Problem Reductions: Models and Transformations

Jin-Guo Liu[1] Xi-Wei Pan[1] Shi-Wen An

[1]Hong Kong University of Science and Technology (Guangzhou)

github.com/CodingThrust/problem-reductions

Abstract. We present formal definitions for computational problems and polynomial-time reductions implemented in the `problem-reductions` library. For each reduction, we state a theorem with a constructive proof; when a reduction is proof-only rather than solver-executable, that restriction is stated explicitly in the rule text.

Contents

1 Introduction

A *reduction* from problem A to problem B, denoted $A \longrightarrow B$, is a polynomial-time transformation of A-instances into B-instances such that: (1) the transformation runs in polynomial time, (2) solutions to B can be efficiently mapped back to solutions of A, and (3) optimal solutions are preserved. The library implements 265 catalogued edges connecting 220 problem types; most are solver-executable witness, aggregate, or Turing reductions, while a few are proof-only NP-hardness embeddings that are excluded from runtime path search.

1.1 Notation

We use the following notation throughout. An *undirected graph* $G = (V, E)$ consists of a vertex set V and edge set $E \subseteq \binom{V}{2}$. For a set S, $\overline{S}$ or $V \setminus S$ denotes its complement. We write $|S|$ for cardinality. A *clique* in G is a subset $K \subseteq V$ where every pair of distinct vertices is adjacent: $(u, v) \in E$ for all distinct $u, v \in K$. A *unit disk graph* is a graph where vertices are points on a 2D lattice and $(u, v) \in E$ iff $d(u, v) \leq r$ for some radius r; a *King's subgraph* uses the 8-connectivity square grid with $r \approx 1.5$. For Boolean variables, $\overline{x}$ denotes negation ($\neg x$). A *literal* is a variable x or its negation $\overline{x}$. A *clause* is a disjunction of literals. A formula in *conjunctive normal form* (CNF) is a conjunction of clauses. We abbreviate Independent Set as IS, Vertex Cover as VC, and use n for problem size, m for number of clauses, and $k_j = |C_j|$ for clause size.

2 Problem Definitions

Each problem definition follows this structure:

Definition N (Problem Name). Formal problem statement defining input, constraints, and objective.

```
ProblemName
field_name    Field description from JSON schema
```

Reduces to: ProblemA, ProblemB.
Reduces from: ProblemC.

The gray schema table shows the JSON field names used in the library's data structures. The reduction links at the bottom connect to the corresponding theorems in Section 3.

2.1 Graph Problems

In all graph problems below, $G = (V, E)$ denotes an undirected graph with $|V| = n$ vertices and $|E|$ edges.

Definition 2.1 (Maximum Independent Set): Given $G = (V, E)$ with vertex weights $w : V \to \mathbb{R}$, find $S \subseteq V$ maximizing $\sum_{v \in S} w(v)$ such that no two vertices in S are adjacent: $\forall u, v \in S : (u, v) \notin E$.

- Complexity: `2^sqrt(num_vertices)`; `1.1996^num_vertices`.
- Reduces to: MaximumIndependentSet, MaximumSetPacking, IntegralFlowBundles, MaximumClique, MinimumVertexCover.
- Reduces from: LongestCommonSubsequence, MaximumClique, MaximumIndependentSet, MaximumSetPacking, MinimumVertexCover, Satisfiability.

```
MaximumIndependentSet
graph                The underlying graph G=(V,E)
weights              Vertex weights w: V -> R
```

One of Karp's 21 NP-complete problems [1], MIS appears in wireless network scheduling, register allocation, and coding theory [2]. Solvable in polynomial time on bipartite graphs (König's theorem), interval graphs, chordal graphs, and cographs. The best known algorithm runs in $O^*(1.1996^n)$ time via measure-and-conquer branching [3]. On geometric graphs (King's subgraph, triangular subgraph, unit disk graphs), MIS admits subexponential $O^*\left(c^{\sqrt{n}}\right)$ algorithms for some constant c, via geometric separation [4].

Example. Consider the Petersen graph G with $n = 10$ vertices, $|E| = 15$ edges, and unit weights $w(v) = 1$ for all $v \in V$. The graph is 3-regular (every vertex has degree 3). A maximum independent set is $S =$

$\{v_0, v_2, v_8, v_9\}$ with $w(S) = \sum_{v \in S} w(v) = 4 = 4(G)$. No two vertices in S share an edge, and no vertex can be added without violating independence.

```
$ pred create --example MIS -o mis.json
$ pred solve mis.json
$ pred evaluate mis.json --config 1,0,1,0,0,0,0,0,1,1
```

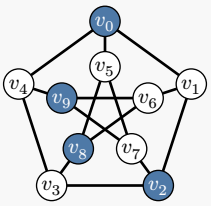


Figure 1: The Petersen graph with a maximum independent set $S = \{v_0, v_2, v_8, v_9\}$ shown in blue ($4(G) = 4$). Outer vertices $v_0, ..., v_4$ form a pentagon; inner vertices $v_5, ..., v_9$ form a pentagram. Unit weights $w(v_i) = 1$.

Definition 2.2 (Minimum Vertex Cover): Given $G = (V, E)$ with vertex weights $w : V \to \mathbb{R}$, find $S \subseteq V$ minimizing $\sum_{v \in S} w(v)$ such that every edge has at least one endpoint in S: $\forall (u, v) \in E : u \in S \vee v \in S$.

- Complexity: `1.1996^num_vertices`.
- Reduces to: EnsembleComputation, LongestCommonSubsequence, MinimumHittingSet, MinimumMaximalMatching, DecisionMinimumVertexCover, MaximumIndependentSet, MinimumFeedbackArcSet, MinimumFeedbackVertexSet, MinimumSetCovering, MinimumWeightAndOrGraph.
- Reduces from: DecisionMinimumVertexCover, KSatisfiability, MaximumIndependentSet.

MinimumVertexCover	
graph	The underlying graph G=(V,E)
weights	Vertex weights w: V -> R

One of Karp's 21 NP-complete problems [1]. Vertex Cover is the complement of Independent Set: S is a vertex cover iff $V \setminus S$ is an independent set, so $|\text{VC}| + |\text{IS}| = n$. Central to parameterized complexity, admitting FPT algorithms in $O^*(1.2738^k)$ time parameterized by solution size k. The best known exact algorithm runs in $O^*(1.1996^n)$ via the MIS complement [3].

Example. Consider the house graph G with $n = 5$ vertices, $|E| = 6$ edges, and unit weights $w(v) = 1$. A minimum vertex cover is $S = \{v_0, v_3, v_4\}$ with $w(S) = 3$: (v_0, v_1) by v_0; (v_0, v_2) by v_0; (v_1, v_3) by v_3; (v_2, v_3) by v_3; (v_2, v_4) by v_4; (v_3, v_4) by both. The complement $\{v_1, v_2\}$ is a maximum independent set ($2(G) = 2$, confirming $|\text{VC}| = n - 2 = 3$).

```
$ pred create --example MVC -o mvc.json
$ pred solve mvc.json
$ pred evaluate mvc.json --config 1,0,0,1,1
```

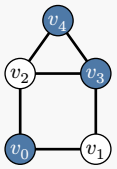


Figure 2: The house graph with a minimum vertex cover $S = \{v_0, v_3, v_4\}$ shown in blue ($w(S) = 3$). Every edge is incident to at least one blue vertex.

Definition 2.3 (Decision Minimum Vertex Cover): Given an undirected graph $G = (V, E)$ with vertex weights $w : V \to \mathbb{R}_{\geq 0}$ and an integer bound k, determine whether there exists a vertex cover $S \subseteq V$ with $\sum_{v \in S} w(v) \leq k$ such that every edge has at least one endpoint in S.

- Complexity: `1.1996^num_vertices`.
- Reduces to: HamiltonianCircuit, MinimumVertexCover.
- Reduces from: KSatisfiability, MinimumVertexCover.

DecisionMinimumVertexCover	
graph	The underlying graph G=(V,E)
weights	Vertex weights w: V -> R
bound	Decision bound (maximum allowed cover cost)

Decision Minimum Vertex Cover is the decision version of Minimum Vertex Cover and one of Karp's 21 NP-complete problems [1], [5]. It asks whether the optimization objective can be achieved within a prescribed budget rather than minimizing the cover weight directly.

Example. Consider a graph on $n = 4$ vertices and $|E| = 4$ edges with threshold $k = 2$. The cover $S = \{v_0, v_2\}$ has total weight $2 \leq 2$ and therefore certifies a yes-instance.

```
$ pred create --example DecisionMinimumVertexCover -o vc.json
$ pred solve vc.json
$ pred evaluate vc.json --config 1,0,1,0
```

Definition 2.4 (Max-Cut): Given $G = (V, E)$ with weights $w : E \to \mathbb{R}$, find partition $\left(S, \overline{S}\right)$ maximizing $\sum_{(u,v) \in E : u \in S, v \in \overline{S}} w(u, v)$.

- Complexity: `2^(0.7907 * num_vertices)`; `2^(2.372 * num_vertices / 3)`.
- Reduces to: MinimumCutIntoBoundedSets, SpinGlass.
- Reduces from: GraphPartitioning, Maximum2Satisfiability, NAESatisfiability, SpinGlass.

MaxCut	
graph	The graph with edge weights
edge_weights	Edge weights w: E -> R

Max-Cut is NP-hard on general graphs [6] but polynomial-time solvable on planar graphs. The Goemans-Williamson SDP relaxation achieves a 0.878-approximation ratio [7], which is optimal assuming the Unique Games Conjecture. The best known exact algorithm runs in $O^*\left(2^{\omega n/3}\right)$ time via algebraic 2-CSP techniques [8], where $\omega < 2.372$ is the matrix multiplication exponent.

Example. Consider the house graph G with $n = 5$ vertices, $|E| = 6$ edges, and unit weights $w(e) = 1$. The partition $S = \{v_0, v_3\}$, $\overline{S} = \{v_1, v_2, v_4\}$ cuts 5 of 6 edges: (v_0, v_1), (v_0, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4). Only the edge (v_2, v_4) is uncut (both endpoints in $\overline{S}$). The cut value is $\sum w(e) = 5$.

```
$ pred create --example MaxCut -o maxcut.json
$ pred solve maxcut.json
$ pred evaluate maxcut.json --config 1,0,0,1,0
```

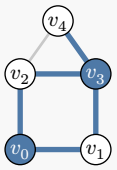


Figure 3: The house graph with max cut $S = \{v_0, v_3\}$ (blue) vs $\overline{S} = \{v_1, v_2, v_4\}$ (white). Cut edges shown in bold blue; 5 of 6 edges are cut.

Definition 2.5 (Graph Partitioning): Given an undirected graph $G = (V, E)$ with $|V| = n$ (even), find a partition of V into two disjoint sets A and B with $|A| = |B| = n/2$ minimizing the number of crossing edges $|\{(u, v) \in E : u \in A, v \in B\}|$.

- Complexity: `2^num_vertices`.
- Reduces to: ILP, MaxCut, QUBO.

GraphPartitioning	
graph	The undirected graph G=(V,E)

Graph Partitioning (Minimum Bisection, Garey & Johnson ND14) is the special case of Minimum Cut Into Bounded Sets with unit weights, $B = n/2$, and no designated s, t vertices. The problem is NP-hard even on 3-regular graphs [9].

The best known exact algorithm is brute-force enumeration over all balanced partitions in $O^*(2^n)$ time[1].

Definition 2.6 (Minimum Cut Into Bounded Sets): Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{Z}^+$, designated vertices $s, t \in V$, and a positive integer $B \leq |V|$, find a partition of V into disjoint sets V_1 and V_2 such that $s \in V_1$, $t \in V_2$, $|V_1| \leq B$, $|V_2| \leq B$, that minimizes the total cut weight

$$\sum_{\{u,v\} \in E : u \in V_1, v \in V_2} w(\{u, v\}).$$

- Complexity: `2^num_vertices`.
- Reduces to: ILP.
- Reduces from: MaxCut.

MinimumCutIntoBoundedSets	
graph	The undirected graph G = (V, E)
edge_weights	Edge weights w: E -> Z+
source	Source vertex s (must be in V1)
sink	Sink vertex t (must be in V2)
size_bound	Maximum size B for each partition set

Minimum Cut Into Bounded Sets (Garey & Johnson ND17) combines the classical minimum s-t cut problem with a balance constraint on partition sizes. Without the balance constraint ($B = |V|$), the problem reduces to standard minimum s-t cut, solvable in polynomial time via network flow. Adding the requirement $|V_1| \leq B$ and $|V_2| \leq B$ makes the problem NP-complete; it remains NP-complete even for $B = |V|$ /2 and unit edge weights (the minimum bisection problem) [9]. Applications include VLSI layout, load balancing, and graph bisection.

The best known exact algorithm is brute-force enumeration of all 2^n vertex partitions in $O(2^n)$ time. For the special case of minimum bisection, Cygan et al. [10] showed fixed-parameter tractability with respect to the cut size. No polynomial-time finite approximation factor exists for balanced graph partition unless $P = NP$ (Andreev and Racke, 2006). Arora, Rao, and Vazirani [11] gave an $O(\sqrt{\log n})$-approximation for balanced separator.

Example. Consider G with 4 vertices and edges (v_0, v_1), (v_1, v_2), (v_2, v_3) with unit weights, $s = v_0$, $t = v_3$, $B = 3$. The optimal partition $V_1 = \{v_0, v_1\}$, $V_2 = \{v_2, v_3\}$ gives minimum cut weight $w(\{v_1, v_2\}) = 1$. Both $|V_1| = 2 \leq 3$ and $|V_2| = 2 \leq 3$.

Definition 2.7 (Biconnectivity Augmentation): Given an undirected graph $G = (V, E)$, a set F of candidate edges on V with $F \cap E = \emptyset$, weights $w : F \to \mathbb{R}$, and a budget $B \in \mathbb{R}$, find $F' \subseteq F$ such that

[1] No algorithm improving on brute-force enumeration is known for general Graph Partitioning.

$\sum_{e \in F'} w(e) \leq B$ and the augmented graph $G' = (V, E \cup F')$ is biconnected, meaning G' is connected and deleting any single vertex leaves it connected.

- Complexity: `2^num_potential_edges`.
- Reduces to: ILP.
- Reduces from: HamiltonianCircuit.

BiconnectivityAugmentation	
graph	The underlying graph G=(V,E)
potential_weights	Potential edges with augmentation weights
budget	Maximum total augmentation weight B

Biconnectivity augmentation is a classical network-design problem: add backup links so the graph survives any single vertex failure. The weighted candidate-edge formulation modeled here captures communication, transportation, and infrastructure planning settings where only a prescribed set of new links is feasible and each carries a cost. In this library, the exact baseline is brute-force enumeration over the $m = |F|$ candidate edges, yielding $O^*(2^m)$ time and matching the exported complexity metadata for the model.

Example. Consider the path graph $v_0 - v_1 - v_2 - v_3 - v_4 - v_5$ with candidate edges (v_0, v_2), (v_0, v_3), (v_0, v_4), (v_1, v_3), (v_1, v_4), (v_1, v_5), (v_2, v_4), (v_2, v_5), (v_3, v_5) carrying weights $(1, 2, 3, 1, 2, 3, 1, 2, 1)$ and budget $B = 4$. Selecting $F' = \{(v_0, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_5)\}$ uses total weight $1 + 1 + 1 + 1 = 4$ and eliminates every articulation point: after deleting any single vertex, the remaining graph is still connected. Reducing the budget to $B = 3$ makes the instance infeasible, because one of the path endpoints remains attached through a single articulation vertex.

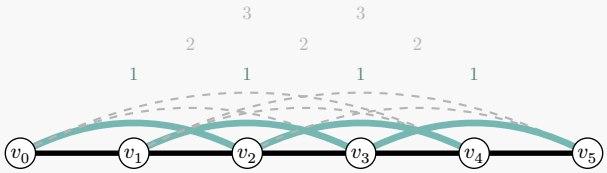


Figure 4: Biconnectivity Augmentation on a 6-vertex path with $B = 4$. Existing edges are black; green arcs show the selected augmentation F' (total weight 4); dashed gray arcs are unselected candidates. The resulting graph $G' = (V, E \cup F')$ is biconnected.

Definition 2.8 (Hamiltonian Circuit): **Instance:** An undirected graph $G = (V, E)$.

Question: Does G contain a *Hamiltonian circuit* — a closed path that visits every vertex exactly once?

- Complexity: `1.657^num_vertices`.
- Reduces to: BiconnectivityAugmentation, BottleneckTravelingSalesman, HamiltonianPath, LongestCircuit, QuadraticAssignment, RuralPostman, StackerCrane, StrongConnectivityAugmentation, TravelingSalesman.
- Reduces from: DecisionMinimumVertexCover.

HamiltonianCircuit	
graph	The undirected graph G=(V,E)

The Hamiltonian Circuit problem is one of Karp's original 21 NP-complete problems [1], and is listed as GT37 in Garey & Johnson [5]. It is closely related to the Traveling Salesman Problem: while TSP seeks to minimize the total weight of a Hamiltonian cycle on a weighted complete graph, the Hamiltonian Circuit problem simply asks whether *any* such cycle exists on a general (unweighted) graph.

A configuration is a permutation π of the vertices, interpreted as the order in which they are visited. The circuit is valid when every consecutive pair $(\pi(i), \pi(i + 1 \bmod n))$ is an edge in G.

Algorithms. The classical Held–Karp dynamic programming algorithm [12] solves the problem in $O(n^2 \cdot 2^n)$ time and $O(n \cdot 2^n)$ space. Björklund's randomized "Determinant Sums" algorithm achieves $O^*(1.657^n)$ time for general graphs and $O^*\left(\sqrt{2}^n\right)$ for bipartite graphs [13].

Example. Consider the triangular prism graph G on 6 vertices with 9 edges. The permutation [0, 1, 2, 5, 4, 3] forms a Hamiltonian circuit: each consecutive pair $(0,1)$,$(1,2)$,$(2,5)$,$(5,4)$,$(4,3)$,$(3,0)$ is an edge of G, and the path returns to the start.

```
$ pred create --example HC -o hc.json
$ pred solve hc.json
$ pred evaluate hc.json --config 0,1,2,5,4,3
```

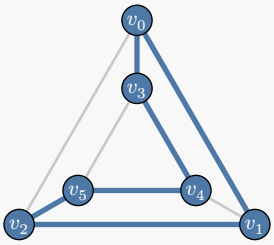


Figure 5: Hamiltonian Circuit in the triangular prism graph. Blue edges show the circuit $v_0 \to v_1 \to v_2 \to v_5 \to v_4 \to v_3 \to v_0$.

Definition 2.9 (Longest Circuit): Given an undirected graph $G = (V, E)$ with positive edge lengths $l : E \to \mathbb{Z}^+$, find a simple circuit $C \subseteq E$ that maximizes $\sum_{e \in C} l(e)$.

- Complexity: `2^num_vertices * num_vertices^2`.
- Reduces to: ILP.
- Reduces from: HamiltonianCircuit.

```
LongestCircuit
graph          The underlying graph G=(V,E)
edge_lengths   Positive edge lengths l: E -> Z_(> 0)
```

Longest Circuit is the optimization version of the classical longest-cycle problem. Hamiltonian Circuit is the special case where every edge has unit length and the optimum equals $|V|$, so Longest Circuit is NP-hard via Karp's original Hamiltonicity result [1]. A standard exact baseline uses Held–Karp-style subset dynamic programming in $O(n^2 \cdot 2^n)$ time [12]; unlike Hamiltonicity, the goal here is to find the longest simple cycle rather than specifically a spanning one.

In the implementation, a configuration selects a subset of edges. A witness is a configuration whose selected edges induce one connected 2-regular subgraph; the objective value is the total selected length.

Example. Consider the canonical 6-vertex instance. The optimal circuit $v_0 \to v_1 \to v_4 \to v_5 \to v_2 \to v_3 \to v_0$ uses edge lengths $3 + 2 + 5 + 1 + 4 + 3 = 18$, which is the maximum circuit length. The remaining edges are available but yield shorter circuits.

```
$ pred create --example LongestCircuit/SimpleGraph/i32 -o longest-circuit.json
$ pred solve longest-circuit.json
$ pred evaluate longest-circuit.json --config 1,0,1,0,1,0,1,1,1,0
```

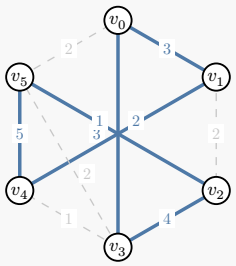


Figure 6: Longest Circuit instance on 6 vertices. The highlighted cycle $v_0 \to v_1 \to v_4 \to v_5 \to v_2 \to v_3 \to v_0$ has maximum total length 18 the remaining edges yield shorter circuits.

Definition 2.10 (Bounded Component Spanning Forest): Given an undirected graph $G = (V, E)$ with vertex weights $w : V \to \mathbb{Z}_{\geq 0}$, a positive integer $K \leq |V|$, and a positive bound B, determine whether there exists a partition of V into t non-empty sets $V_1, ..., V_t$ with $1 \leq t \leq K$ such that each induced subgraph $G[V_i]$ is connected and each part satisfies $\sum_{v \in V_i} w(v) \leq B$.

- Complexity: `3^num_vertices`.
- Reduces to: ILP.
- Reduces from: PartitionIntoPathsOfLength2.

BoundedComponentSpanningForest	
graph	The underlying graph G=(V,E)
weights	Vertex weights w(v) for each vertex v in V
max_components	Upper bound K on the number of connected components
max_weight	Upper bound B on the total weight of each component

Bounded Component Spanning Forest appears as ND10 in Garey and Johnson [5]. It asks for a decomposition into a bounded number of connected pieces, each with bounded total weight, so it naturally captures contiguous districting and redistricting-style constraints where each district must remain connected while respecting a population cap. A direct exhaustive search over component labels gives an $O^*(K^n)$ baseline, but subset-DP techniques via inclusion-exclusion improve the exact running time to $O^*(3^n)$ [14].

Example. Consider the graph on vertices $\{v_0, v_1, ..., v_7\}$ with edges (v_0, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6), (v_6, v_7), (v_0, v_7), (v_1, v_5), (v_2, v_6); vertex weights $(2, 3, 1, 2, 3, 1, 2, 1)$; component limit $K = 3$; and bound $B = 6$. The partition $V_1 = \{v_0, v_1, v_7\}$, $V_2 = \{v_2, v_3, v_4\}$, $V_3 = \{v_5, v_6\}$ is feasible: each set induces a connected subgraph, the component weights are $2 + 3 + 1 = 6$, $1 + 2 + 3 = 6$, and $1 + 2 = 3$, and exactly three non-empty components are used. Therefore this instance is a YES instance.

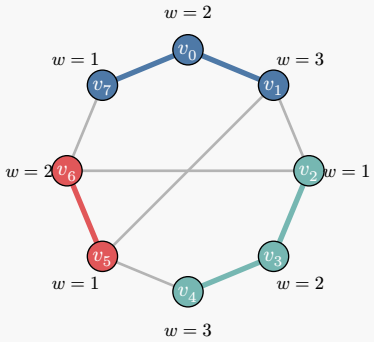


Figure 7: Bounded Component Spanning Forest on 8 vertices with $K = 3$ and $B = 6$. The partition $V_1 = \{v_0, v_1, v_7\}$ (blue, weight 6), $V_2 = \{v_2, v_3, v_4\}$ (green, weight 6), $V_3 = \{v_5, v_6\}$ (red, weight 3) is feasible. Bold colored edges are intra-component; gray edges cross components.

Definition 2.11 (Length-Bounded Disjoint Paths): Given an undirected graph $G = (V, E)$, distinct terminals $s, t \in V$, and a positive integer K, maximize the number of pairwise internally vertex-disjoint paths from s to t, each using at most K edges.

- Complexity: `2^(max_paths * num_vertices)`.
- Reduces to: ILP.

LengthBoundedDisjointPaths	
graph	The underlying graph G=(V,E)
source	The shared source vertex s
sink	The shared sink vertex t
max_paths	Upper bound on the number of path slots
max_length	Maximum path length K in edges

Length-Bounded Disjoint Paths is the bounded-routing version of the classical disjoint-path problem, with applications in network routing and VLSI where multiple connections must fit simultaneously under quality-of-service limits. Garey & Johnson list it as ND41 and summarize the sharp threshold proved by Itai, Perl, and Shiloach: the problem is NP-complete for every fixed $K \geq 5$, polynomial-time solvable for $K \leq 4$, and becomes polynomial again when the length bound is removed entirely [5]. The implementation here uses $M \cdot |V|$ binary variables where $M = \min(\deg(s), \deg(t))$ is an upper bound on the number of vertex-disjoint s-t paths, so brute-force search over configurations runs in $O^*\left(2^{M \cdot |V|}\right)$.

Example. Consider the graph G with $n = 5$ vertices, $|E| = 6$ edges, terminals $s = v_0$, $t = v_4$, and $K = 3$. Here $M = 3$ path slots are available. The three paths $P_1 = v_0 \to v_1 \to v_4$, $P_2 = v_0 \to v_2 \to v_4$, and $P_3 = v_0 \to v_3 \to v_4$ are each of length 2 (at most $K = 3$), and their internal vertex sets $\{v_1\}$, $\{v_2\}$, and $\{v_3\}$ are pairwise disjoint. The optimal value is therefore 3.

```
$ pred create --example LengthBoundedDisjointPaths -o length-bounded-disjoint-paths.json
$ pred solve length-bounded-disjoint-paths.json
$ pred evaluate length-bounded-disjoint-paths.json --config 1,1,0,0,1,1,0,1,0,1,1,0,0,1,1
```

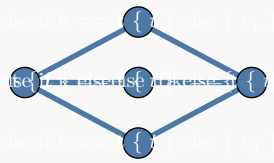


Figure 8: An optimal Length-Bounded Disjoint Paths instance with $s = v_0$, $t = v_4$, and $K = 3$. All three vertex-disjoint paths $v_0 \to v_1 \to v_4$, $v_0 \to v_2 \to v_4$, and $v_0 \to v_3 \to v_4$ are highlighted, giving an optimal value of 3.

Definition 2.12 (Disjoint Connecting Paths): Given an undirected graph $G = (V, E)$ and pairwise disjoint terminal pairs $(s_1, t_1), ..., (s_k, t_k)$, determine whether G contains k mutually vertex-disjoint paths such that path P_i joins s_i to t_i for every i.

- Complexity: `2^num_edges`.
- Reduces to: ILP.

DisjointConnectingPaths	
graph	The underlying graph G=(V,E)
terminal_pairs	Disjoint terminal pairs (s_i, t_i)

Disjoint Connecting Paths is the classical routing form of the vertex-disjoint paths problem, catalogued as ND40 in Garey & Johnson [5]. When the number of terminal pairs k is part of the input, the problem is NP-complete [1]. In contrast, for every fixed k, Robertson and Seymour give an $O(n^3)$ algorithm [15], and Kawarabayashi, Kobayashi, and Reed later improve the dependence on n to $O(n^2)$ [16]. The implementation

in this crate uses one binary variable per undirected edge, so brute-force search explores an $O^*(2^{|E|})$ configuration space.[2]

Example. Consider the repaired YES instance with $n = 6$ vertices, $|E| = 7$ edges, and terminal pairs (v_0, v_3) and (v_2, v_5). Selecting the edges v_0v_1, v_1v_3, v_2v_4, and v_4v_5 yields the two vertex-disjoint paths $v_0 \to v_1 \to v_3$ and $v_2 \to v_4 \to v_5$, so the instance is satisfying.

```
$ pred create --example DisjointConnectingPaths -o disjoint-connecting-paths.json
$ pred solve disjoint-connecting-paths.json
$ pred evaluate disjoint-connecting-paths.json --config 1,0,1,0,1,0,1
```

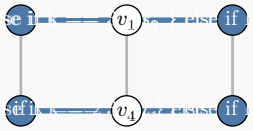


Figure 9: A satisfying Disjoint Connecting Paths instance with terminal pairs (v_0, v_3) and (v_2, v_5). The highlighted edges form the vertex-disjoint paths $v_0 \to v_1 \to v_3$ and $v_2 \to v_4 \to v_5$.

Definition 2.13 (Generalized Hex): Given an undirected graph $G = (V, E)$ and distinct terminals $s, t \in V$, determine whether Player 1 has a forced win in the vertex-claiming Shannon switching game where the players alternately claim vertices of $V \setminus \{s, t\}$, coloring them blue and red respectively, and Player 1 wins iff the final coloring contains an s-t path whose internal vertices are all blue.

- Complexity: `3^num_playable_vertices`.

GeneralizedHex	
graph	The underlying graph G=(V,E)
source	The source terminal s
target	The target terminal t

Generalized Hex is the vertex version of the Shannon switching game listed by Garey & Johnson (A8 GP1). Even and Tarjan proved that deciding whether the first player has a winning strategy is PSPACE-complete [17]. The edge-claiming Shannon switching game is a classical contrast point: Bruno and Weinberg showed that the edge version is polynomial-time solvable via matroid methods [18].

The implementation evaluates the decision problem directly rather than searching over candidate assignments. The instance has `dims() = []`, and `evaluate([])` runs a memoized minimax search over the ternary states (unclaimed, blue, red) of the nonterminal vertices. This preserves the alternating-game semantics of the original problem instead of collapsing the game into a static coloring predicate.

Example. The canonical fixture uses the six-vertex graph with terminals $s = v_0$ and $t = v_5$, and edges (v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_4), (v_2, v_4), (v_3, v_4), (v_4, v_5). Vertex v_4 is the unique neighbor of t, so Player 1 opens by claiming v_4. Player 2 can then block at most one of v_1, v_2, and v_3; Player 1 responds by claiming one of the remaining branch vertices, completing a blue path $v_0 \to v_i \to v_4 \to v_5$. The fixture database therefore has exactly one satisfying configuration: the empty configuration, which triggers the internal game-tree evaluator on the initial board.

```
$ pred create --example GeneralizedHex -o generalized-hex.json
$ pred solve generalized-hex.json
$ pred evaluate generalized-hex.json --config
```

[2] This is the exact-search bound induced by the edge-subset encoding implemented in the codebase; no sharper general exact worst-case bound is claimed here.

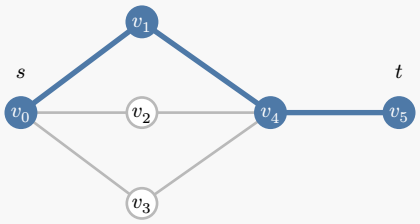


Figure 10: A winning Generalized Hex instance. Player 1 first claims v_4, then answers any red move on $\{v_1, v_2, v_3\}$ by taking a different branch vertex and completing a blue path from $s = v_0$ to $t = v_5$.

Definition 2.14 (Hamiltonian Path): Given a graph $G = (V, E)$, determine whether G contains a *Hamiltonian path*, i.e., a simple path that visits every vertex exactly once.

- Complexity: `1.657^num_vertices`.
- Reduces to: ConsecutiveOnesSubmatrix, DegreeConstrainedSpanningTree, ILP, IsomorphicSpanningTree.
- Reduces from: HamiltonianCircuit.

HamiltonianPath	
graph	The underlying graph G=(V,E)

A classical NP-complete decision problem from Garey & Johnson (A1.3 GT39), closely related to *Hamiltonian Circuit*. Finding a Hamiltonian path in G is equivalent to finding a Hamiltonian circuit in an augmented graph G' obtained by adding a new vertex adjacent to all vertices of G. The problem remains NP-complete for planar graphs, cubic graphs, and bipartite graphs.

The best known exact algorithm is Björklund's randomized $O^*(1.657^n)$ "Determinant Sums" method [13], which applies to both Hamiltonian path and circuit. The classical Held–Karp dynamic programming algorithm solves it in $O(n^2 \cdot 2^n)$ deterministic time.

Variables: $n = |V|$ values forming a permutation. Position i holds the vertex visited at step i. A configuration is satisfying when it forms a valid permutation of all vertices and consecutive vertices are adjacent in G.

Example. Consider the graph G on 6 vertices with edges $\{(0,1), (0,2), (1,3), (2,3), (3,4), (3,5), (4,2), (5,1)\}$. The sequence [0, 2, 4, 3, 1, 5] is a Hamiltonian path: it visits every vertex exactly once, and each consecutive pair is adjacent — $(0,2)$,$(2,4)$,$(4,3)$,$(3,1)$,$(1,5) \in E$.

```
$ pred create --example HamiltonianPath -o hamiltonian-path.json
$ pred solve hamiltonian-path.json
$ pred evaluate hamiltonian-path.json --config 0,2,4,3,1,5
```

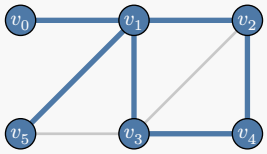


Figure 11: Hamiltonian Path in a 6-vertex graph. Blue edges show the path $v_0 \to v_2 \to v_4 \to v_3 \to v_1 \to v_5$.

Definition 2.15 (Hamiltonian Path Between Two Vertices): Given a graph $G = (V, E)$ and two distinguished vertices $s, t \in V$, determine whether G contains a *Hamiltonian s–t path*, i.e., a simple path that begins at s, ends at t, and visits every vertex of G exactly once.

- Complexity: `1.657^num_vertices`.
- Reduces to: LongestPath.

`HamiltonianPathBetweenTwoVertices`

`graph`	The underlying graph G=(V,E)
`source_vertex`	Source vertex s
`target_vertex`	Target vertex t

A classical NP-complete decision problem from Garey & Johnson (GT39, p. 60), closely related to *Hamiltonian Path*. Fixing the two endpoints of a Hamiltonian path gives a natural and useful specialisation: any Hamiltonian circuit can be converted into a Hamiltonian s–t path by removing the edge (s, t), and conversely a Hamiltonian s–t path plus the edge (s, t) yields a Hamiltonian circuit. The problem remains NP-complete for planar graphs and bipartite graphs.

The best known exact algorithm is Björklund's randomized $O^*(1.657^n)$ "Determinant Sums" method [13]. The classical Held–Karp dynamic programming algorithm solves it in $O(n^2 \cdot 2^n)$ deterministic time by initialising the DP only at s and accepting only solutions that terminate at t [12].

Variables: $n = |V|$ values forming a permutation. Position i holds the vertex visited at step i. A configuration is satisfying when it forms a valid permutation of all vertices, the first element is s, the last element is t, and consecutive vertices are adjacent in G.

Example. Consider the graph G on 6 vertices with edges $\{\{0,1\}, \{0,3\}, \{1,2\}, \{1,4\}, \{2,5\}, \{3,4\}, \{4,5\}, \{2,3\}\}$, source $s = 0$, and target $t = 5$. The sequence [0, 3, 2, 1, 4, 5] is a Hamiltonian s–t path: it starts at s, ends at t, visits every vertex exactly once, and each consecutive pair is adjacent — $\{0,3\}$,$\{3,2\}$,$\{2,1\}$,$\{1,4\}$,$\{4,5\} \in E$.

```
$ pred create --example HamiltonianPathBetweenTwoVertices -o hpbtv.json
$ pred solve hpbtv.json
$ pred evaluate hpbtv.json --config 0,3,2,1,4,5
```

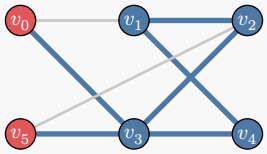


Figure 12: Hamiltonian s–t path in a 6-vertex graph with $s = v_0$ and $t = v_5$ (orange). Blue edges show the path $v_0 \to v_3 \to v_2 \to v_1 \to v_4 \to v_5$.

Definition 2.16 (Directed Hamiltonian Path): Given a directed graph $G = (V, A)$, determine whether G contains a *directed Hamiltonian path*, i.e., a simple directed path that visits every vertex exactly once following arc directions.

- Complexity: `num_vertices^2 * 2^num_vertices`.
- Reduces to: ILP.

`DirectedHamiltonianPath`

`graph`	The directed graph G=(V,A)

A classical NP-complete decision problem from Garey & Johnson (A2.1 GT39). The directed version is NP-complete even for tournaments and remains hard for most restricted digraph classes.

The best known exact algorithm runs in $O(n^2 \cdot 2^n)$ time using Held–Karp style dynamic programming with bitmask DP.

Variables: A permutation of the n vertices, encoded as a Lehmer code with $\text{dims} = [n, n-1, ..., 1]$. A configuration is satisfying when every consecutive pair in the decoded permutation forms a directed arc.

Example. Consider the directed graph G on 6 vertices with arcs $\{(0 \to 1), (0 \to 3), (1 \to 3), (1 \to 4), (2 \to 0), (2 \to 4), (3 \to 2), (3 \to 5), (4 \to 5), (5 \to 1)\}$. The directed Hamiltonian path $0 \to 1 \to 3 \to 2 \to 4 \to 5$ visits every vertex exactly once with all consecutive pairs being arcs.

```
$ pred create --example DirectedHamiltonianPath -o dhp.json
$ pred solve dhp.json
```

Definition 2.17 (Kernel): Given a directed graph $G = (V, A)$, find a *kernel* $V' \subseteq V$ such that (1) V' is *independent* — no arc joins any two vertices in V' — and (2) V' is *absorbing* — every vertex $u \notin V'$ has an arc $(u, v) \in A$ for some $v \in V'$.

- Complexity: `2^num_vertices`.
- Reduces from: KSatisfiability.

Kernel	
graph	The directed graph G=(V,A)

A classical graph-theoretic concept introduced by von Neumann and Morgenstern (1944) in the context of game theory. Deciding whether a directed graph has a kernel is NP-complete in general, though every DAG has a unique kernel. Kernels appear in combinatorial game theory, graph coloring (Galvin's theorem), and stable set problems on digraphs.

Variables: A binary vector of length $|V|$, where $x_v = 1$ iff vertex v is in the kernel.

Example. Consider the directed graph G on 5 vertices with arcs $\{(0 \to 1), (0 \to 2), (1 \to 3), (2 \to 3), (3 \to 4), (4 \to 0), (4 \to 1)\}$. The kernel $V' = \{0, 3\}$ is independent (no arc between 0 and 3) and absorbing (vertex 1 has arc to 3, vertex 2 has arc to 3, vertex 4 has arc to 0).

```
$ pred create --example Kernel -o kernel.json
$ pred solve kernel.json
```

Definition 2.18 (Longest Path): Given an undirected graph $G = (V, E)$ with positive edge lengths $l : E \to \mathbb{Z}^+$ and designated vertices $s, t \in V$, find a simple path P from s to t maximizing $\sum_{e \in P} l(e)$.

- Complexity: `num_vertices * 2^num_vertices`.
- Reduces to: ILP.
- Reduces from: HamiltonianPathBetweenTwoVertices.

LongestPath	
graph	The underlying graph G=(V,E)
edge_lengths	Positive edge lengths l: E -> ZZ_(> 0)
source_vertex	Source vertex s
target_vertex	Target vertex t

Longest Path is problem ND29 in Garey & Johnson [5]. It bridges weighted routing and Hamiltonicity: when every edge has unit length, the optimum reaches $|V| - 1$ exactly when there is a Hamiltonian path from s to t. The implementation catalog records the classical subset-DP exact bound $O\big(|V| \cdot 2^{|V|}\big)$, in the style of Held–Karp dynamic programming [12]. For the parameterized k-path version, color-coding gives randomized $2^{O(k)}\ |V|^{O(1)}$ algorithms [19].

Variables: one binary value per edge. A configuration is valid exactly when the selected edges form a single simple s-t path; otherwise the metric is `Invalid`. For valid selections, the metric is the total selected edge length.

Example. Consider the graph on 7 vertices with source $s = v_0$ and target $t = v_6$. The highlighted path $v_0 \to v_1 \to v_3 \to v_2 \to v_4 \to v_5 \to v_6$ uses edges $\{(v_0, v_1), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_4, v_5), (v_5, v_6)\}$, so its total length is $3 + 4 + 1 + 5 + 3 + 4 = 20$. Another valid path, $v_0 \to v_2 \to v_4 \to v_5 \to v_3 \to v_1 \to v_6$, has total length 17, so the highlighted path is strictly better.

```
$ pred create --example LongestPath -o longest-path.json
$ pred solve longest-path.json
$ pred evaluate longest-path.json --config 1,0,1,1,1,0,1,0,1,0
```

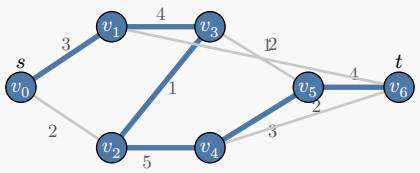


Figure 13: Longest Path instance with edge lengths shown on the edges. The highlighted path from $s = v_0$ to $t = v_6$ has total length 20.

Definition 2.19 (Undirected Flow with Lower Bounds): Given an undirected graph $G = (V, E)$, specified vertices $s, t \in V$, lower bounds $l : E \to \mathbb{Z}_{\geq 0}$, upper capacities $c : E \to \mathbb{Z}^+$ with $l(e) \leq c(e)$ for every edge, and a requirement $R \in \mathbb{Z}^+$, determine whether there exists a flow function $f : \{(u, v), (v, u) : \{u, v\} \in E\} \to \mathbb{Z}_{\geq 0}$ such that each edge carries flow in at most one direction, every edge value lies between its lower and upper bound, flow is conserved at every vertex in $V \setminus \{s, t\}$, and the net flow into t is at least R.

- Complexity: `2^num_edges`.
- Reduces to: ILP.

`UndirectedFlowLowerBounds`	
`graph`	`Undirected graph G=(V,E)`
`capacities`	`Upper capacities c(e) in graph edge order`
`lower_bounds`	`Lower bounds l(e) in graph edge order`
`source`	`Source vertex s`
`sink`	`Sink vertex t`
`requirement`	`Required net inflow R at sink t`

Undirected Flow with Lower Bounds appears as ND37 in Garey and Johnson's catalog [5]. Itai proved that even this single-commodity undirected feasibility problem is NP-complete, contrasting sharply with the directed lower-bounded case, which reduces to ordinary max-flow machinery [20].

The implementation exposes one binary decision per edge rather than raw flow magnitudes. The configuration (0, 0, 0, 0, 0, 0, 0) means "orient every edge exactly as listed in the stored edge order"; once an orientation is fixed, `evaluate()` checks the remaining lower-bounded directed circulation conditions internally. This keeps the explicit search space at 2^m for $m = |E|$, matching the registry complexity bound.

Example. The canonical fixture uses source $s = v_0$, sink $t = v_5$, requirement $R = 3$, edges $\{(v_0, v_1), (v_0, v_2), (v_1, v_3), (v_2, v_3), (v_1, v_4), (v_3, v_5), (v_4, v_5)\}$, and lower/upper pairs $\{(1, 2), (1, 2), (0, 2), (0, 2), (1, 1), (0, 3), (1, 2)\}$ in that order. Under the all-zero orientation config, a feasible witness sends flows (2, 1, 1, 1, 1, 2, 1) along those edges respectively: 2 on (v_0, v_1), 1 on (v_0, v_2), 1 on (v_1, v_3), 1 on (v_2, v_3), 1 on (v_1, v_4), 2 on (v_3, v_5), and 1 on (v_4, v_5). Every lower bound is satisfied, each nonterminal vertex has equal inflow and outflow, and the sink receives $2 + 1 = 3 \geq R$, so the instance evaluates to true. A separate rule issue tracks the natural reduction to ILP; this model PR only documents the standalone verifier.

```
$ pred create --example UndirectedFlowLowerBounds -o undirected-flow-lower-bounds.json
$ pred solve undirected-flow-lower-bounds.json
$ pred evaluate undirected-flow-lower-bounds.json --config 0,0,0,0,0,0,0
```

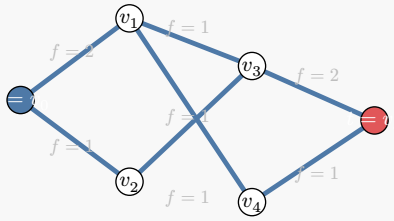


Figure 14: Canonical YES instance for Undirected Flow with Lower Bounds. Blue edges follow the all-zero orientation config, and edge labels show one feasible witness flow.

Definition 2.20 (Undirected Two-Commodity Integral Flow): Given an undirected graph $G = (V, E)$, specified terminals $s_1, s_2, t_1, t_2 \in V$, edge capacities $c : E \to \mathbb{Z}^+$, and requirements $R_1, R_2 \in \mathbb{Z}^+$, determine whether there exist two integral flow functions f_1, f_2 that orient each used edge for each commodity, respect the shared edge capacities, conserve flow at every vertex in $V \setminus \{s_1, s_2, t_1, t_2\}$, and deliver at least R_i units of net flow into t_i for each commodity $i \in \{1, 2\}$.

- Complexity: `5^num_edges`.
- Reduces to: ILP.

UndirectedTwoCommodityIntegralFlow	
graph	Undirected graph G=(V,E)
capacities	Edge capacities c(e) in graph edge order
source_1	Source vertex s_1 for commodity 1
sink_1	Sink vertex t_1 for commodity 1
source_2	Source vertex s_2 for commodity 2
sink_2	Sink vertex t_2 for commodity 2
requirement_1	Required net inflow R_1 at sink t_1
requirement_2	Required net inflow R_2 at sink t_2

Undirected Two-Commodity Integral Flow is the undirected counterpart of the classical two-commodity integral flow problem from Garey & Johnson (ND39) [5]. Even, Itai, and Shamir proved that it remains NP-complete even when every capacity is 1, but becomes polynomial-time solvable when all capacities are even, giving a rare parity-driven complexity dichotomy [21].

The implementation uses four variables per undirected edge $\{u, v\}$: $f_1(u, v)$, $f_1(v, u)$, $f_2(u, v)$, and $f_2(v, u)$. In the unit-capacity regime, each edge has exactly five meaningful local states: unused, commodity 1 in either direction, or commodity 2 in either direction, which matches the catalog bound $O(5^m)$ for $m = |E|$.

Example. Consider the graph with edges $(0, 2)$, $(1, 2)$, and $(2, 3)$, capacities $(1, 1, 2)$, sources $s_1 = v_0$, $s_2 = v_1$, and shared sink $t_1 = t_2 = v_3$. The optimal configuration in the fixture database sets $f_1(0, 2) = 1$, $f_2(1, 2) = 1$, and $f_1(2, 3) = f_2(2, 3) = 1$, with all reverse-direction variables zero. The only nonterminal vertex is v_2, where each commodity has one unit of inflow and one unit of outflow, so conservation holds. Vertex v_3 receives one unit of net inflow from each commodity, and the shared edge $(2, 3)$ uses its full capacity 2. The fixture database contains 1 satisfying configuration for this instance, shown below.

```
$ pred create --example UndirectedTwoCommodityIntegralFlow -o undirected-two-commodity-integral-flow.json
$ pred solve undirected-two-commodity-integral-flow.json
$ pred evaluate undirected-two-commodity-integral-flow.json --config 1,0,0,0,0,0,1,0,1,0,1,0
```

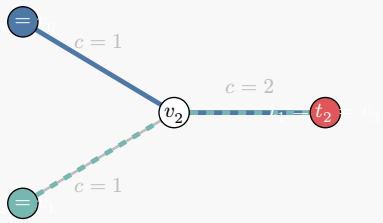


Figure 15: Canonical shared-capacity YES instance for Undirected Two-Commodity Integral Flow. Solid blue carries commodity 1 and dashed teal carries commodity 2; both commodities share the edge (v_2, v_3) of capacity 2.

Definition 2.21 (Path-Constrained Network Flow): Given a directed graph $G = (V, A)$, designated vertices $s, t \in V$, arc capacities $c : A \to \mathbb{Z}^+$, a prescribed collection $\mathcal{P}$ of directed simple s-t paths, and a requirement $R \in \mathbb{Z}^+$, determine whether there exists an integral path-flow function $g : \mathcal{P} \to \mathbb{Z}_{\geq 0}$ such that $\sum_{p \in \mathcal{P} : a \in p} g(p) \leq c(a)$ for every arc $a \in A$ and $\sum_{p \in \mathcal{P}} g(p) \geq R$.

- Complexity: `(max_capacity + 1)^num_paths`.
- Reduces to: ILP.

PathConstrainedNetworkFlow	
graph	Directed graph G = (V, A)
capacities	Capacity c(a) for each arc
source	Source vertex s
sink	Sink vertex t
paths	Prescribed directed s-t paths as arc-index sequences
requirement	Required total flow R

Path-Constrained Network Flow appears as problem ND34 in Garey & Johnson [5]. Unlike ordinary single-commodity flow, the admissible routes are fixed in advance: every unit of flow must be assigned to one of the listed s-t paths. This prescribed-path viewpoint is standard in line planning and unsplittable routing, and Büsing and Stiller give a modern published NP-completeness and inapproximability treatment for exactly this integral formulation [22].

The implementation uses one integer variable per prescribed path, bounded by that path's bottleneck capacity. Exhaustive search over those path-flow variables gives the registered worst-case bound $O^*\big((C+1)^{|\mathcal{P}|}\big)$, where $C = \max_{a \in A} c(a)$.[3]

Example. The canonical fixture uses the directed network with arcs $(0,1)$, $(0,2)$, $(1,3)$, $(1,4)$, $(2,4)$, $(3,5)$, $(4,5)$, $(4,6)$, $(5,7)$, and $(6,7)$, capacities $(2,1,1,1,1,1,1,2,1)$, source $s = 0$, sink $t = 7$, and required flow $R = 3$. The prescribed paths are $p_1 = 0 \to 1 \to 3 \to 5 \to 7$, $p_2 = 0 \to 1 \to 4 \to 5 \to 7$, $p_3 = 0 \to 1 \to 4 \to 6 \to 7$, $p_4 = 0 \to 2 \to 4 \to 5 \to 7$, and $p_5 = 0 \to 2 \to 4 \to 6 \to 7$. The fixture's satisfying configuration is $g = (1,1,0,0,1) = (1,1,0,0,1)$, so one unit is sent along p_1, one along p_2, and one along p_5. The shared arcs $(0,1)$ and $(5,7)$ each carry exactly two units of flow, matching their capacity 2, while every other used arc carries one unit. Therefore the total flow into t is $3 = R$, so the instance is feasible.

```
$ pred create --example PathConstrainedNetworkFlow -o path-constrained-network-flow.json
$ pred solve path-constrained-network-flow.json --solver brute-force
$ pred evaluate path-constrained-network-flow.json --config 1,1,0,0,1
```

[3] This is the brute-force bound induced by the representation used in the library; no sharper general exact algorithm is claimed here for the integral prescribed-path formulation.

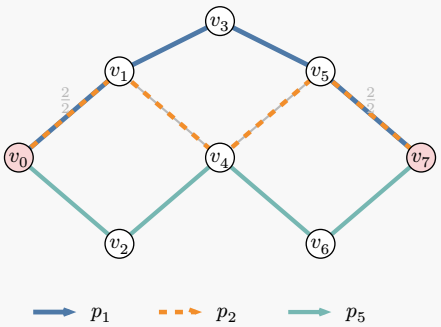


Figure 16: Canonical YES instance for Path-Constrained Network Flow. Blue, dashed orange, and teal show the three prescribed paths used by $g = (1, 1, 0, 0, 1)$. The labels $\frac{2}{2}$ mark the shared arcs $(0, 1)$ and $(5, 7)$, whose flow exactly saturates capacity 2.

Definition 2.22 (Isomorphic Spanning Tree): Given a graph $G = (V, E)$ and a tree $T = (V_T, E_T)$ with $|V| = |V_T|$, determine whether G contains a spanning tree isomorphic to T: does there exist a bijection `(0, 1, 2, 3)` : $V_T \to V$ such that for every edge $\{u, v\} \in E_T$, $\{$`(0, 1, 2, 3)`(u), `(0, 1, 2, 3)`$(v)\} \in E$?

- Complexity: `num_vertices^num_vertices`.
- Reduces to: ILP.
- Reduces from: HamiltonianPath.

IsomorphicSpanningTree	
graph	The host graph G
tree	The target tree T (must be a tree with |V(T)| = |V(G)|)

A classical NP-complete problem listed as ND8 in Garey & Johnson [5]. The Isomorphic Spanning Tree problem strictly generalizes Hamiltonian Path: a graph G has a Hamiltonian path if and only if G contains a spanning tree isomorphic to the path P_n. The problem remains NP-complete even when T is restricted to trees of bounded degree [23].

Brute-force enumeration of all bijections `(0, 1, 2, 3)` : $V_T \to V$ and checking each against the edge set of G runs in $O(n! \cdot n)$ time. No substantially faster exact algorithm is known for general instances.

Variables: $n = |V|$ values forming a permutation. Position i holds the graph vertex that tree vertex i maps to under `(0, 1, 2, 3)`. A configuration is satisfying when it forms a valid permutation and every tree edge maps to a graph edge.

Example. Consider $G = K_4$ (the complete graph on 4 vertices) and T the star S_3 with center 0 and leaves {1, 2, 3}. Since K_4 contains all possible edges, any bijection `(0, 1, 2, 3)` maps the star's edges to edges of G. For instance, the identity mapping `(0, 1, 2, 3)`$(i) = i$ gives the spanning tree $\{(0, 1), (0, 2), (0, 3)\} \subseteq E(K_4)$.

```
$ pred create --example IsomorphicSpanningTree -o isomorphic-spanning-tree.json
$ pred solve isomorphic-spanning-tree.json
$ pred evaluate isomorphic-spanning-tree.json --config 0,1,2,3
```

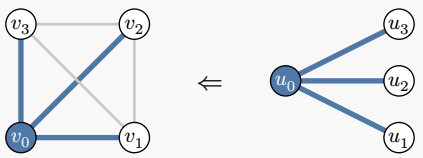


Figure 17: Isomorphic Spanning Tree: the graph $G = K_4$ (left) contains a spanning tree isomorphic to the star S_3 (right, blue edges). The identity mapping `(0, 1, 2, 3)`$(u_i) = v_i$ embeds all 3 star edges into G. Center vertex v_0 shown in blue.

Definition 2.23 (Shortest Weight-Constrained Path): Given an undirected graph $G = (V, E)$ with positive edge lengths $l : E \to \mathbb{Z}^+$, positive edge weights $w : E \to \mathbb{Z}^+$, designated vertices $s, t \in V$, and a weight bound $W \in \mathbb{Z}^+$, find a simple path P from s to t that minimizes $\sum_{e \in P} l(e)$ subject to $\sum_{e \in P} w(e) \leq W$.

- Complexity: `2^num_edges`.
- Reduces to: ILP.
- Reduces from: Partition.

ShortestWeightConstrainedPath	
graph	The underlying graph G=(V,E)
edge_lengths	Edge lengths l: E -> ZZ_(> 0)
edge_weights	Edge weights w: E -> ZZ_(> 0)
source_vertex	Source vertex s
target_vertex	Target vertex t
weight_bound	Upper bound W on total path weight

Also called the *restricted shortest path* or *resource-constrained shortest path* problem. Garey and Johnson list it as ND30 and show NP-completeness via transformation from Partition [5]. The model captures bicriteria routing: one resource measures path length or delay, while the other captures a second consumable budget such as cost, risk, or bandwidth. Because pseudo-polynomial dynamic programming formulations are known [24], the hardness is weak rather than strong; approximation schemes were later developed by Hassin [25] and improved by Lorenz and Raz [26].

The implementation catalog reports the natural brute-force complexity of the edge-subset encoding used here: with $m = |E|$ binary variables, exhaustive search over all candidate subsets costs $O^*(2^m)$. A configuration is feasible when the selected edges form a single simple s-t path whose total weight stays within the bound; the objective is to minimize total length over all such feasible paths.

Example. Consider the graph on 6 vertices with source $s = v_0$, target $t = v_5$, and weight bound $W = 8$. Edge labels are written as $(l(e), w(e))$. The highlighted path $v_0 \to v_2 \to v_3 \to v_5$ uses edges $\{(v_0, v_2), (v_2, v_3), (v_3, v_5)\}$, so its total length is $4 + 1 + 4 = 9$ and its total weight is $1 + 3 + 3 = 7 \leq 8$. This is the minimum-length feasible path; another weight-feasible path $v_0 \to v_1 \to v_4 \to v_5$ has length 10.

```
$ pred create --example ShortestWeightConstrainedPath -o shortest-weight-constrained-path.json
$ pred solve shortest-weight-constrained-path.json
$ pred evaluate shortest-weight-constrained-path.json --config 0,1,0,1,0,1,0,0
```

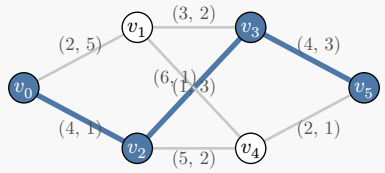


Figure 18: Shortest Weight-Constrained Path instance with edge labels $(l(e), w(e))$. The highlighted path $v_0 \to v_2 \to v_3 \to v_5$ satisfies both bounds.

Definition 2.24 (k-Coloring): Given $G = (V, E)$ and k colors, find $c : V \to \{1, ..., k\}$ minimizing $|\{(u, v) \in E : c(u) = c(v)\}|$.

- Complexity: `num_vertices + num_edges`; `1.3289^num_vertices`; `1.7159^num_vertices`; `2^num_vertices`.
- Reduces to: Clustering, KColoring, TwoDimensionalConsecutiveSets, ILP, PartitionIntoCliques, QUBO.
- Reduces from: KColoring, Satisfiability.

KColoring	
graph	The underlying graph G=(V,E)

Graph coloring arises in register allocation, frequency assignment, and scheduling [5]. Deciding k-colorability is NP-complete for $k \geq 3$ but solvable in $O(n+m)$ for $k=2$ via bipartiteness testing. For $k=3$, the best known algorithm runs in $O^*(1.3289^n)$ [27]; for $k=4$ in $O^*(1.7159^n)$ [28]; for $k=5$ in $O^*((2-\varepsilon)^n)$ [29]. In general, inclusion-exclusion achieves $O^*(2^n)$ [14].

Example. Consider the house graph G with $k=3$ colors. The coloring $c(v_0)=1$, $c(v_1)=2$, $c(v_2)=2$, $c(v_3)=1$, $c(v_4)=3$ is proper: no adjacent pair shares a color, so the number of conflicts is 0. The house graph has chromatic number $\chi(G)=3$ because the triangle (v_2, v_3, v_4) requires 3 colors.

```
$ pred create --example KColoring/SimpleGraph/K3 -o kcoloring.json
$ pred solve kcoloring.json
$ pred evaluate kcoloring.json --config 0,1,1,0,2
```

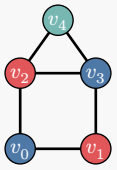


Figure 19: A proper 3-coloring of the house graph. Colors: $c(v_0)=c(v_3)=1$, $c(v_1)=c(v_2)=2$, $c(v_4)=3$. Zero conflicts.

Definition 2.25 (Maximum Achromatic Number): Given an undirected graph $G=(V,E)$, find a proper vertex coloring $c: V \to \{1,...,k\}$ that is *complete* — for every pair of distinct colors i, j there exists an edge $(u,v) \in E$ with $c(u)=i$ and $c(v)=j$ — maximizing the number of colors k.

- Complexity: `num_vertices^num_vertices`.

MaximumAchromaticNumber	
graph	The underlying graph G=(V,E)

The achromatic number $\psi(G)$ is the largest k such that G admits a complete proper k-coloring. It was introduced by Harary and Hedetniemi (1970) and shown NP-hard [5, GT5]. Applications include network partition and information dissemination. Brute-force enumeration runs in $O^*(n^n)$ time.

Example. Consider the 6-cycle C_6 with $n=6$ vertices and $|E|=6$ edges: $\{0,1\}$, $\{1,2\}$, $\{2,3\}$, $\{3,4\}$, $\{4,5\}$, $\{5,0\}$. The coloring $c(v_0)=1$, $c(v_1)=2$, $c(v_2)=3$, $c(v_3)=1$, $c(v_4)=2$, $c(v_5)=3$ uses 3 colors. It is proper (no adjacent pair shares a color) and complete: every pair of color classes is connected by at least one edge. Thus $\psi(C_6) \geq 3$.

```
$ pred create --example MaximumAchromaticNumber -o achromatic.json
$ pred solve achromatic.json
$ pred evaluate achromatic.json --config 0,1,2,0,1,2
```

Definition 2.26 (Minimum Dominating Set): Given $G=(V,E)$ with weights $w: V \to \mathbb{R}$, find $S \subseteq V$ minimizing $\sum_{v \in S} w(v)$ s.t. $\forall v \in V : v \in S \vee \exists u \in S : (u,v) \in E$.

- Complexity: `1.4969^num_vertices`.
- Reduces to: DecisionMinimumDominatingSet, ILP.
- Reduces from: DecisionMinimumDominatingSet, Satisfiability.

MinimumDominatingSet	
graph	The underlying graph G=(V,E)
weights	Vertex weights w: V -> R

Dominating Set models facility location: each vertex in S "covers" itself and its neighbors. Applications include wireless sensor placement and social network influence maximization. W[2]-complete when parameterized by solution size k, making it strictly harder than Vertex Cover in the parameterized hierarchy. The best known exact algorithm runs in $O^*(1.4969^n)$ via measure-and-conquer [30].

Example. Consider the house graph G with $n = 5$ vertices and unit weights $w(v) = 1$. The set $S = \{v_2, v_3\}$ is a minimum dominating set with $w(S) = 2$: vertex v_2 dominates $\{v_0, v_4\}$ and vertex v_3 dominates $\{v_1, v_4\}$ (both also dominate each other). No single vertex can dominate all others, so $\gamma(G) = 2$.

```
$ pred create --example MinimumDominatingSet -o minimum-dominating-set.json
$ pred solve minimum-dominating-set.json
$ pred evaluate minimum-dominating-set.json --config 0,0,1,1,0
```

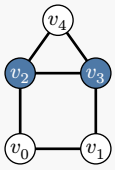


Figure 20: The house graph with minimum dominating set $S = \{v_2, v_3\}$ (blue, $\gamma(G) = 2$). Every white vertex is adjacent to at least one blue vertex.

Definition 2.27 (Minimum Geometric Connected Dominating Set): Given points $P = \{p_1, ..., p_n\}$ in $\mathbb{R}^2$ and distance threshold $B > 0$, find $P' \subseteq P$ minimizing $|P'|$ s.t. (1) $\forall p \in P \setminus P' : \exists q \in P'$ with $d(p, q) \leq B$ (domination), and (2) the unit-disk graph on P' with radius B is connected.

- Complexity: `2^num_points`.

MinimumGeometricConnectedDominatingSet	
points	The set of points P in the plane
radius	The distance threshold B

Geometric Connected Dominating Set arises in wireless ad-hoc networks: selected nodes form a connected backbone that covers all other nodes within communication range. The problem is NP-hard and generalizes both dominating set (dropping connectivity) and connected subgraph (dropping domination).

Example. Consider $n = 8$ points arranged in a 4×2 ladder with spacing 3 and threshold $B = 3.5$. The bottom row $P' = \{p_0, p_1, p_2, p_3\}$ forms a minimum connected dominating set of size 4: each bottom-row point dominates the top-row point directly above it (vertical distance $3 \leq 3.5$), and consecutive bottom-row points are within distance $3 \leq 3.5$ of each other, so P' induces a connected path.

```
$ pred create --example MinimumGeometricConnectedDominatingSet -o mgcds.json
$ pred solve mgcds.json
$ pred evaluate mgcds.json --config 1,1,1,1,0,0,0,0
```

Definition 2.28 (Minimum Covering by Cliques): Given an undirected graph $G = (V, E)$, find a collection of cliques $C_1, ..., C_k$ in G such that every edge $e \in E$ is contained in at least one C_i, and the number of cliques k is minimized.

- Complexity: `2^num_edges`.
- Reduces to: ILP.
- Reduces from: PartitionIntoCliques.

MinimumCoveringByCliques	
graph	The underlying graph G=(V,E)

Minimum Covering by Cliques (also called *edge clique cover*) is NP-hard [5, GT59]. Applications include intersection graph recognition and computational biology. The minimum edge clique cover number equals the minimum dimension of a dot-product representation of the graph. Brute-force enumeration runs in $O^*(2^{|E|})$ time.

Example. Consider G with $n = 6$ vertices and $|E| = 9$ edges: $\{0,1\}$, $\{1,2\}$, $\{2,3\}$, $\{3,0\}$, $\{0,2\}$, $\{4,0\}$, $\{4,1\}$, $\{5,2\}$, $\{5,3\}$. An optimal cover uses 4 cliques: C_1: edges 0, 1, 4; C_2: edges 2, 3; C_3: edges 5, 6; C_4: edges 7, 8.

```
$ pred create --example MinimumCoveringByCliques -o covering-by-cliques.json
$ pred solve covering-by-cliques.json
$ pred evaluate covering-by-cliques.json --config 0,0,1,1,0,2,2,3,3
```

Definition 2.29 (Minimum Intersection Graph Basis): Given an undirected graph $G = (V, E)$, find a universe U of minimum cardinality and an assignment of subsets $S_v \subseteq U$ for each vertex $v \in V$ such that two vertices u, v are adjacent if and only if $S_u \cap S_v \neq \emptyset$. The minimum $|U|$ is the *intersection number* of G.

- Complexity: `num_edges^num_edges`.

MinimumIntersectionGraphBasis	
graph	The underlying graph G=(V,E)

Minimum Intersection Graph Basis is NP-hard [5, GT60]. Every graph is an intersection graph; the intersection number measures how compactly such a representation can be chosen. The intersection number is at most $|E|$. Brute-force enumeration runs in $O^*(|E|^{|E|})$ time.

Example. Consider the path P_3 with $n = 3$ vertices and $|E| = 2$ edges: $\{0,1\}$, $\{1,2\}$. An optimal representation uses 2 elements: $S_0 = \{0\}$, $S_1 = \{0,1\}$, $S_2 = \{1\}$.

```
$ pred create --example MinimumIntersectionGraphBasis -o intersection-basis.json
$ pred solve intersection-basis.json
$ pred evaluate intersection-basis.json --config 1,0,1,1,0,1
```

Definition 2.30 (Maximum Matching): Given $G = (V, E)$ with weights $w : E \to \mathbb{R}$, find $M \subseteq E$ maximizing $\sum_{e \in M} w(e)$ s.t. $\forall e_1, e_2 \in M : e_1 \cap e_2 = \emptyset$.

- Complexity: `num_vertices^3`.
- Reduces to: ILP, MaximumSetPacking.

MaximumMatching	
graph	The underlying graph G=(V,E)
edge_weights	Edge weights w: E -> R

Unlike most combinatorial optimization problems on general graphs, maximum matching is solvable in polynomial time $O(n^3)$ by Edmonds' blossom algorithm [31], which introduced the technique of shrinking odd cycles into pseudo-nodes. Matching theory underpins assignment problems, network flows, and the Tutte-Berge formula for matching deficiency.

Example. Consider the house graph G with $n = 5$ vertices, $|E| = 6$ edges, and unit weights $w(e) = 1$. A maximum matching is $M = \{(v_0, v_1), (v_2, v_4)\}$ with $w(M) = 2$. Each matched edge is vertex-disjoint from the others. Vertex v_3 is unmatched; since n is odd, no perfect matching exists.

```
$ pred create --example MaximumMatching -o maximum-matching.json
$ pred solve maximum-matching.json
$ pred evaluate maximum-matching.json --config 1,0,0,0,1,0
```

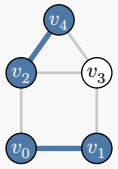


Figure 21: The house graph with a maximum matching $M = \{(v_0, v_1), (v_2, v_4)\}$ (blue edges, $w(M) = 2$). Matched vertices shown in blue; v_3 is unmatched.

Definition 2.31 (Bottleneck Traveling Salesman): Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$, find an edge set $C \subseteq E$ that forms a cycle visiting every vertex exactly once and minimizes $\max_{e \in C} w(e)$.

- Complexity: `num_vertices^2 * 2^num_vertices`.
- Reduces to: ILP.
- Reduces from: HamiltonianCircuit.

BottleneckTravelingSalesman	
graph	The underlying graph G=(V,E)
edge_weights	Edge weights w: E -> Z

This min-max variant models routing where the worst leg matters more than the total distance. Garey and Johnson list the threshold decision version as ND24 [5]: given a bound B, ask whether some Hamiltonian tour has every edge weight at most B. The optimization version implemented here subsumes that decision problem. The classical Held–Karp dynamic programming algorithm still yields an exact $O(n^2 \cdot 2^n)$-time algorithm [12], while Garey and Johnson note the polynomial-time special case of Gilmore and Gomory [32].

Example. Consider the complete graph K_5 with vertices $\{v_0, v_1, v_2, v_3, v_4\}$ and edge weights $w(v_0, v_1) = 5$, $w(v_0, v_2) = 4$, $w(v_0, v_3) = 4$, $w(v_0, v_4) = 5$, $w(v_1, v_2) = 4$, $w(v_1, v_3) = 1$, $w(v_1, v_4) = 2$, $w(v_2, v_3) = 1$, $w(v_2, v_4) = 5$, $w(v_3, v_4) = 4$. The unique optimal bottleneck tour is $v_0 \to v_2 \to v_1 \to v_4 \to v_3 \to v_0$ with edge weights 4, 4, 4, 2, 4 and bottleneck 4. Its total weight is 18. By contrast, the minimum-total-weight TSP tour $v_0 \to v_2 \to v_3 \to v_1 \to v_4 \to v_0$ has total weight 13 but bottleneck 5, because it uses the weight-5 edge (v_0, v_4). Here every other Hamiltonian tour in K_5 contains a weight-5 edge, so the blue tour is the only one that keeps the maximum edge weight at 4.

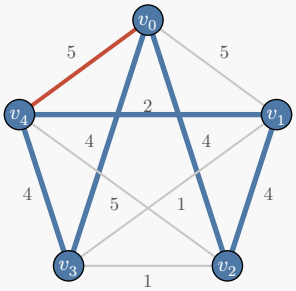


Figure 22: The K_5 bottleneck-TSP instance. Blue edges form the unique optimal bottleneck tour; the red edge (v_0, v_4) is the weight-5 edge used by the minimum-total-weight TSP tour.

Definition 2.32 (Traveling Salesman): Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$, find an edge set $C \subseteq E$ that forms a cycle visiting every vertex exactly once and minimizes $\sum_{e \in C} w(e)$.

- Complexity: `2^num_vertices`.
- Reduces to: ILP, QUBO.
- Reduces from: HamiltonianCircuit.

TravelingSalesman	
graph	The underlying graph G=(V,E)
edge_weights	Edge weights w: E -> R

One of the most intensely studied NP-hard problems, with applications in logistics, circuit board drilling, and DNA sequencing. The best known exact algorithm runs in $O^*(2^n)$ time and space via Held-Karp dynamic programming [12]. No $O^*((2-\varepsilon)^n)$ algorithm is known, and improving the exponential space remains open.

Example. Consider the complete graph K_4 with vertices $\{v_0, v_1, v_2, v_3\}$ and edge weights $w(v_0, v_1) = 1$, $w(v_0, v_2) = 3$, $w(v_0, v_3) = 2$, $w(v_1, v_2) = 2$, $w(v_1, v_3) = 3$, $w(v_2, v_3) = 1$. The optimal tour is $v_0 \to v_1 \to v_2 \to v_3 \to v_0$ with cost $1 + 2 + 2 + 1 = 6$.

```
$ pred create --example TSP -o tsp.json
$ pred solve tsp.json
$ pred evaluate tsp.json --config 1,0,1,1,0,1
```

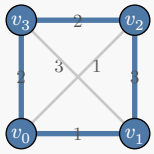


Figure 23: Complete graph K_4 with weighted edges. The optimal tour $v_0 \to v_1 \to v_2 \to v_3 \to v_0$ (blue edges) has cost 6.

Definition 2.33 (Steiner Tree): Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}_{\geq 0}$ and a set of terminal vertices $T \subseteq V$ with $|T| \geq 2$, find a tree $S = (V_S, E_S)$ in G such that $T \subseteq V_S$, minimizing $\sum_{e \in E_S} w(e)$. Vertices in $V_S \setminus T$ are called *Steiner vertices*.

- Complexity: `3^num_terminals * num_vertices + 2^num_terminals * num_vertices^2`.
- Reduces to: ILP.

SteinerTree	
graph	The underlying graph G=(V,E)
edge_weights	Edge weights w: E -> R
terminals	Terminal vertices T that must be connected

One of Karp's 21 NP-complete problems [1], foundational in network design with applications in telecommunications backbone routing, VLSI chip interconnect, pipeline planning, and phylogenetic tree construction. When $T = V$, the problem reduces to the minimum spanning tree (polynomial). The NP-hardness arises from choosing which Steiner vertices to include.

The best known exact algorithm runs in $O^*\left(3^{|T|} \cdot n + 2^{|T|} \cdot n^2\right)$ time via Dreyfus–Wagner dynamic programming over terminal subsets [33]. Byrka *et al.* achieved a $\ln(4) + \varepsilon \approx 1.39$-approximation [34]; the classic 2-approximation uses the minimum spanning tree of the terminal distance graph.

Example. Consider G with $n = 5$ vertices, $m = 7$ edges, and terminals $T = \{v_0, v_2, v_4\}$. The optimal Steiner tree uses edges $\{(v_0, v_1), (v_1, v_2), (v_1, v_3), (v_3, v_4)\}$ with Steiner vertices $\{v_1, v_3\}$ acting as relay points. The total cost is $2+2+1+1 = 6$. Note the only direct terminal–terminal edge (v_2, v_4) has weight 6, equaling the entire Steiner tree cost.

```
$ pred create --example SteinerTree -o steiner-tree.json
$ pred solve steiner-tree.json
$ pred evaluate steiner-tree.json --config 1,0,1,1,0,0,1
```

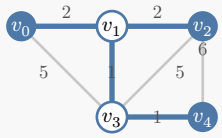


Figure 24: Steiner tree on 5 vertices with terminals $T = \{v_0, v_2, v_4\}$ (filled blue). Steiner vertices v_1, v_3 (outlined) relay connections. Blue edges form the optimal tree with cost 6.

Definition 2.34 (Strong Connectivity Augmentation): Given a directed graph $G = (V, A)$, a set $C \subseteq (V \times V \setminus A) \times \mathbb{Z}_{>0}$ of weighted candidate arcs, and a bound $B \in \mathbb{Z}_{\geq 0}$, determine whether there exists a subset $C' \subseteq C$ such that $\sum_{(u,v,w)\in C'} w \leq B$ and the augmented digraph $(V, A \cup \{(u, v) : (u, v, w) \in C'\})$ is strongly connected.

- Complexity: `2^num_potential_arcs`.
- Reduces to: ILP.
- Reduces from: HamiltonianCircuit.

StrongConnectivityAugmentation	
graph	The initial directed graph G=(V,A)
candidate_arcs	Candidate augmenting arcs (u, v, w(u,v)) not already present in G
bound	Upper bound B on the total added weight

Strong Connectivity Augmentation models network design problems where a partially connected directed communication graph may be repaired by buying additional arcs. Eswaran and Tarjan showed that the unweighted augmentation problem is solvable in linear time, while the weighted variant is substantially harder [35]. The decision version recorded as ND19 in Garey and Johnson is NP-complete [5]. The implementation here uses one binary variable per candidate arc, so brute-force over the candidate set yields a worst-case bound of $O^*(2^m)$ where $m = \text{num\_potential\_arcs}$.[4]

Example. The canonical instance has $n = 5$ vertices, $|A| = 4$ existing arcs, and bound $B = 8$. The base graph is the directed path $v_0 \to v_1 \to v_2 \to v_3 \to v_4$ — every vertex can reach those ahead of it, but vertex v_4 is a sink with no outgoing arcs. The 9 candidate arcs with weights are: $w(v_4, v_0) = 10$, $w(v_4, v_3) = 3$, $w(v_4, v_2) = 3$, $w(v_4, v_1) = 3$, $w(v_3, v_0) = 7$, $w(v_3, v_1) = 3$, $w(v_2, v_0) = 7$, $w(v_2, v_1) = 3$, $w(v_1, v_0) = 5$. The cheapest single arc that closes the cycle is (v_4, v_0), but its weight $10 > B$ exceeds the budget, so strong connectivity must be achieved via a two-hop return path. The pair (v_4, v_1) and (v_1, v_0) with weights 3+5 $= 8 = B$ creates the path $v_4 \to v_1 \to v_0$, making the augmented graph strongly connected at exactly the budget limit. Alternative escape arcs from v_4 (to v_3 or v_2) are equally cheap but land on vertices from which reaching v_0 within the remaining budget is impossible.

```
$ pred create --example StrongConnectivityAugmentation -o strong-connectivity-
augmentation.json
$ pred solve strong-connectivity-augmentation.json
$ pred evaluate strong-connectivity-augmentation.json --config 0,0,0,1,0,0,0,0,1
```

[4] No exact algorithm improving on brute-force is claimed here for the weighted candidate-arc formulation implemented in the codebase.

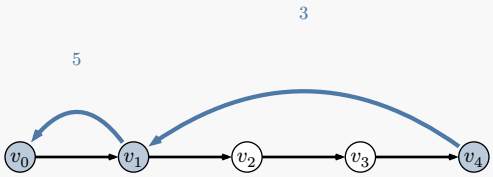


Figure 25: Strong Connectivity Augmentation on a 5-vertex path digraph. Black arcs form the base path A; blue arcs are the unique augmentation $((v_4, v_1), (v_1, v_0))$ with total weight $8 = B = 8$.

Definition 2.35 (Minimum Multiway Cut): Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}_{>0}$ and a set of k terminal vertices $T = \{t_1, ..., t_k\} \subseteq V$, find a minimum-weight set of edges $C \subseteq E$ such that no two terminals remain in the same connected component of $G' = (V, E \setminus C)$.

- Complexity: `1.84^num_terminals * num_vertices^3`.
- Reduces to: ILP, QUBO.

MinimumMultiwayCut	
graph	The undirected graph G=(V,E)
terminals	Terminal vertices that must be separated
edge_weights	Edge weights w: E -> R (same order as graph.edges())

The Minimum Multiway Cut problem generalizes the classical minimum s-t cut: for $k = 2$ it reduces to max-flow and is solvable in polynomial time, but for $k \geq 3$ on general graphs it becomes NP-hard [36]. The problem arises in VLSI design, image segmentation, and network design. A $(2 - 2/k)$-approximation is achievable in polynomial time by taking the union of the $k - 1$ cheapest isolating cuts [36]. The best known exact algorithm runs in $O^*(1.84^k)$ time (suppressing polynomial factors) via submodular functions on isolating cuts [37].

Example. Consider a graph with $n = 5$ vertices, $m = 6$ edges, and $k = 3$ terminals $T = \{0, 2, 4\}$, with edge weights $w(0,1) = 2$, $w(1,2) = 3$, $w(2,3) = 1$, $w(3,4) = 2$, $w(0,4) = 4$, $w(1,3) = 5$. The optimal multiway cut removes edges $\{(0,1), (3,4), (0,4)\}$ with total weight $2+2+4 = 8$, placing each terminal in a distinct component.

```
$ pred create --example MinimumMultiwayCut -o minimum-multiway-cut.json
$ pred solve minimum-multiway-cut.json
$ pred evaluate minimum-multiway-cut.json --config 1,0,0,1,1,0
```

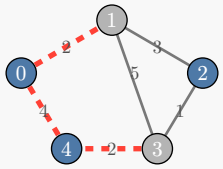


Figure 26: Minimum Multiway Cut with terminals $\{0, 2, 4\}$ (blue). Dashed red edges form the optimal cut (weight 8).

Definition 2.36 (Optimal Linear Arrangement): Given an undirected graph $G = (V, E)$, find a bijection $f : V \to \{0, 1, ..., |V| - 1\}$ that minimizes the total edge length $\sum_{\{u,v\} \in E} |f(u) - f(v)|$.

- Complexity: `2^num_vertices`.
- Reduces to: ILP.

OptimalLinearArrangement	
graph	The undirected graph G=(V,E)

A classical NP-hard optimization problem from Garey & Johnson (GT42) [5], with applications in VLSI design, graph drawing, and sparse matrix reordering. The problem asks for a vertex ordering on a line that minimizes the total "stretch" of all edges.

NP-hardness was established by Garey, Johnson, and Stockmeyer [38], via reduction from Simple Max Cut. The problem remains NP-hard on bipartite graphs, but is solvable in polynomial time on trees. The best known exact algorithm for general graphs uses dynamic programming over subsets in $O^*(2^n)$ time and space (Held-Karp style), analogous to TSP.

Example. Consider a graph with 6 vertices and 7 edges. The arrangement $f = (0, 1, 2, 3, 4, 5)$ gives total cost $|0-1| + |1-2| + |2-3| + |3-4| + |4-5| + |0-3| + |2-5| = 11$, which is optimal.

```
$ pred create --example OptimalLinearArrangement -o optimal-linear-arrangement.json
$ pred solve optimal-linear-arrangement.json
$ pred evaluate optimal-linear-arrangement.json --config 0,1,2,3,4,5
```

Definition 2.37 (Rooted Tree Arrangement): Given an undirected graph $G = (V, E)$ and a non-negative integer K, is there a rooted tree $T = (U, F)$ with $|U| = |V|$ and a bijection $f : V \to U$ such that every edge $\{u, v\} \in E$ maps to two nodes lying on a common root-to-leaf path in T, and $\sum_{\{u,v\}\in E} d_{T(f(u),f(v))} \leq K$?

- Complexity: `2^num_vertices`.
- Reduces to: RootedTreeStorageAssignment.

RootedTreeArrangement	
graph	The undirected graph G=(V,E)
bound	Upper bound K on total tree stretch

Rooted Tree Arrangement is GT45 in Garey and Johnson [5]. It generalizes Optimal Linear Arrangement by allowing the host layout to be any rooted tree rather than a single path. Garey and Johnson cite Gavril's NP-completeness proof via reduction from Optimal Linear Arrangement [39].

The connection to Optimal Linear Arrangement is immediate: if the rooted tree is restricted to a chain, the stretch objective becomes the linear-arrangement objective. This explains why the two problems live in the same arrangement family. For tree-oriented ordering problems, Adolphson and Hu give a polynomial-time algorithm for optimal linear ordering on trees [40], showing that the difficulty here comes from simultaneously choosing both the rooted-tree topology and the vertex-to-node bijection.

Example. Consider the graph with $n = 4$ vertices, $|E| = 4$ edges, and edge set $\{(v_0, v_1), (v_0, v_2), (v_1, v_2), (v_2, v_3)\}$. With bound $K = 5$, the chain tree encoded by parent array $(0, 0, 1, 2)$ and identity mapping $(0, 1, 2, 3)$ is a valid witness: every listed edge lies on the unique root-to-leaf chain, and the total stretch is $1 + 2 + 1 + 1 = 5 \leq 5$. Therefore this canonical instance is a YES instance.

```
$ pred create --example RootedTreeArrangement -o rooted-tree-arrangement.json
$ pred solve rooted-tree-arrangement.json --solver brute-force
$ pred evaluate rooted-tree-arrangement.json --config 0,0,1,2,0,1,2,3
```

Definition 2.38 (k-Clique): Given an undirected graph $G = (V, E)$ and an integer k, determine whether there exists a subset $K \subseteq V$ with $|K| \geq k$ such that every pair of distinct vertices in K is adjacent.

- Complexity: `1.1996^num_vertices`.
- Reduces to: BalancedCompleteBipartiteSubgraph, ConjunctiveBooleanQuery, ILP, SubgraphIsomorphism.
- Reduces from: KSatisfiability.

```
KClique
graph   The underlying graph G=(V,E)
k       Minimum clique size threshold
```

k-Clique is the classical decision version of Clique, one of Karp's original NP-complete problems [1] and listed as GT19 in Garey and Johnson [5]. Unlike Maximum Clique, the threshold k is part of the input, so this formulation is the natural target for decision-to-decision reductions such as 3SAT $\to$ Clique. The best known exact algorithm matches Maximum Clique via the complement reduction to Maximum Independent Set and runs in $O^*(1.1996^n)$ [3].

Example. Consider the house graph G with $n = 5$ vertices, $|E| = 6$ edges, and threshold $k = 3$. The set $K = \{v_2, v_3, v_4\}$ is a valid witness because all three pairs (v_2, v_3), (v_2, v_4), (v_3, v_4) are edges, so $|K| = 3 \geq 3$ and this is a YES instance. This witness is unique, and no 4-clique exists because every vertex outside K misses at least one edge to the other selected vertices.

```
$ pred create --example KClique -o kclique.json
$ pred solve kclique.json
$ pred evaluate kclique.json --config 0,0,1,1,1
```

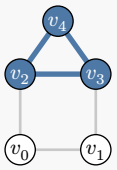


Figure 27: The house graph with satisfying witness $K = \{v_2, v_3, v_4\}$ for $k = 3$. The selected vertices and their internal clique edges are highlighted in blue.

Definition 2.39 (Maximum Clique): Given $G = (V, E)$, find $K \subseteq V$ maximizing $|K|$ such that all pairs in K are adjacent: $\forall u, v \in K : (u, v) \in E$. Equivalent to MIS on the complement graph $\overline{G}$.

- Complexity: `1.1996^num_vertices`.
- Reduces to: ILP, MaximumIndependentSet.
- Reduces from: MaximumIndependentSet.

```
MaximumClique
graph         The underlying graph G=(V,E)
weights       Vertex weights w: V -> R
```

Maximum Clique arises in social network analysis (finding tightly-connected communities), bioinformatics (protein interaction clusters), and coding theory. The problem is equivalent to Maximum Independent Set on the complement graph $\overline{G}$. The best known algorithm runs in $O^*(1.1996^n)$ via the complement reduction to MIS [3]. Robson's direct backtracking algorithm achieves $O^*(1.1888^n)$ using exponential space [41].

Example. Consider the house graph G with $n = 5$ vertices and $|E| = 6$ edges. The triangle $K = \{v_2, v_3, v_4\}$ is a maximum clique of size $3(G) = 3$: all three pairs (v_2, v_3), (v_2, v_4), (v_3, v_4) are edges. No 4-clique exists because vertices v_0 and v_1 each have degree 2 and are not adjacent to all of $\{v_2, v_3, v_4\}$.

```
$ pred create --example MaximumClique -o maximum-clique.json
$ pred solve maximum-clique.json
$ pred evaluate maximum-clique.json --config 0,0,1,1,1
```

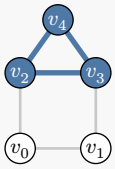


Figure 28: The house graph with maximum clique $K = \{v_2, v_3, v_4\}$ (blue, $3(G) = 3$). All edges within the clique are shown in bold blue.

Definition 2.40 (Maximal Independent Set): Given $G = (V, E)$ with vertex weights $w : V \to \mathbb{R}$, find $S \subseteq V$ maximizing $\sum_{v \in S} w(v)$ such that S is independent ($\forall u, v \in S : (u, v) \notin E$) and maximal (no vertex $u \in V \setminus S$ can be added to S while maintaining independence).

- Complexity: `3^(num_vertices / 3)`.
- Reduces to: ILP.

```
MaximalIS
graph     The underlying graph G=(V,E)
weights   Vertex weights w: V -> R
```

The maximality constraint (no vertex can be added) distinguishes this from MIS, which only requires maximum weight. Every maximum independent set is maximal, but not vice versa. The enumeration bound of $O^*(3^{n/3})$ for listing all maximal independent sets [42] is tight: Moon and Moser [43] showed every n-vertex graph has at most $3^{n/3}$ maximal independent sets, achieved by disjoint triangles.

Example. Consider the path graph P_5 with $n = 5$ vertices, edges (v_i, v_{i+1}) for $i = 0, ..., 3$, and unit weights $w(v) = 1$. The set $S = \{v_1, v_3\}$ is a maximal independent set: no two vertices in S are adjacent, and neither v_0 (adjacent to v_1), v_2 (adjacent to both), nor v_4 (adjacent to v_3) can be added. However, $S' = \{v_0, v_2, v_4\}$ with $w(S') = 3$ is a strictly larger maximal IS, illustrating that maximality does not imply maximum weight.

```
$ pred create --example MaximalIS -o maximal-is.json
$ pred solve maximal-is.json
$ pred evaluate maximal-is.json --config 1,0,1,0,1
```

Figure 29: Path P_5 with maximal IS $S = \{v_1, v_3\}$ (blue, $w(S) = 2$). S is maximal — no white vertex can be added — but not maximum: $\{v_0, v_2, v_4\}$ achieves $w = 3$.

Definition 2.41 (Minimum Maximal Matching): Given $G = (V, E)$, find $M \subseteq E$ of minimum cardinality such that M is a matching and M is maximal: every $e \in E \setminus M$ shares an endpoint with some $e' \in M$.

- Complexity: `1.3160^num_vertices`.
- Reduces to: ILP.
- Reduces from: MinimumVertexCover.

```
MinimumMaximalMatching
graph                 The underlying graph G=(V,E)
```

A maximal matching cannot be extended by any edge, so every non-selected edge must be "blocked" by a selected one. Among all such matchings, the problem seeks one of minimum size. Unlike Maximum Matching (solvable in $O(n^3)$ by Edmonds' algorithm [31]), Minimum Maximal Matching is NP-hard [44]; it can also be viewed as a Minimum Dominating Set in the line graph.

Example. Consider the path graph P_6 with $n = 6$ vertices and 5 edges. A minimum maximal matching is $M = \{(v_1, v_2), (v_3, v_4)\}$ with $|M| = 2$. Every unselected edge shares an endpoint with a selected one, so M is maximal.

```
$ pred create --example MinimumMaximalMatching -o mmm.json
$ pred solve mmm.json
$ pred evaluate mmm.json --config 0,1,0,1,0
```

Figure 30: Path P_6 with minimum maximal matching $M = \{(v_1, v_2), (v_3, v_4)\}$ (blue, $|M| = 2$).

Definition 2.42 (Minimum Dummy Activities in PERT Networks): Given a precedence DAG $G = (V, A)$, find an activity-on-arc PERT event network with one real activity arc for each task $v \in V$, minimizing the number of dummy activity arcs, such that for every ordered pair of tasks (u, v) there is a path from the finish event of u to the start event of v if and only if v is reachable from u in G.

- Complexity: `2^num_arcs`.

MinimumDummyActivitiesPert	
graph	The precedence DAG G=(V,A) whose vertices are tasks and arcs encode direct precedence constraints

The decision version of minimum dummy activities appears as ND44 in Garey and Johnson's compendium [5]. It arises when an activity-on-node precedence DAG must be converted into an activity-on-arc PERT chart: merging compatible finish/start events removes dummy activities, but an over-aggressive merge creates spurious precedence relations between unrelated tasks. The implementation here enumerates, for each direct precedence arc, whether it is realized as an event merge or left as a dummy activity, so brute-force over the $m = 5$ direct precedences yields a worst-case bound of $O^*(2^m)$.[5]

Example. Consider the canonical precedence DAG on $n = 6$ tasks with direct precedences (v_0, v_2), (v_0, v_3), (v_1, v_3), (v_1, v_4), (v_2, v_5). The optimal encoding merges the predecessor-finish/successor-start pairs (v_0, v_2), (v_1, v_4), (v_2, v_5), so those handoffs need no dummy activity at all. The remaining direct precedences (v_0, v_3) and (v_1, v_3) still require dummy activities, so the optimum is 2. Both unresolved precedences enter v_3, and merging either of them would identify unrelated task completions, creating spurious reachability between the two source tasks.

```
$ pred create --example MinimumDummyActivitiesPert -o minimum-dummy-activities-pert.json
$ pred solve minimum-dummy-activities-pert.json --solver brute-force
$ pred evaluate minimum-dummy-activities-pert.json --config 1,0,0,1,1
```

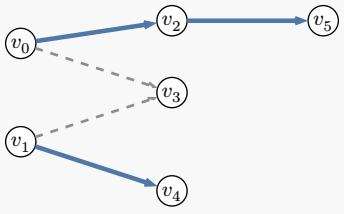


Figure 31: Canonical Minimum Dummy Activities in PERT Networks instance. Blue precedence arcs are encoded by merging the predecessor finish event with the successor start event; dashed gray arcs still require dummy activities. The optimal encoding leaves exactly 2 dummy activities.

[5] No exact algorithm improving on the direct-precedence merge encoding implemented in the codebase is claimed here.

Definition 2.43 (Minimum Feedback Vertex Set): Given a directed graph $G = (V, A)$ with vertex weights $w : V \to \mathbb{R}$, find $S \subseteq V$ minimizing $\sum_{v \in S} w(v)$ such that the induced subgraph $G[V \setminus S]$ is a directed acyclic graph (DAG).

- Complexity: `1.9977^num_vertices`.
- Reduces to: ILP, MinimumCodeGenerationUnlimitedRegisters.
- Reduces from: MinimumVertexCover.

MinimumFeedbackVertexSet	
graph	The directed graph G=(V,A)
weights	Vertex weights w: V -> R

One of Karp's 21 NP-complete problems ("Feedback Node Set") [1]. Applications include deadlock detection in operating systems, loop breaking in circuit design, and Bayesian network structure learning. The directed version is strictly harder than undirected FVS: the best known exact algorithm runs in $O^*(1.9977^n)$ [45], compared to $O^*(1.7548^n)$ for undirected graphs. An $O(\log n \cdot \log \log n)$-approximation exists [46].

Example. Consider the directed graph G with $n = 5$ vertices, $|A| = 7$ arcs, and unit weights. The arcs form two overlapping directed cycles: $C_1 = v_0 \to v_1 \to v_2 \to v_0$ and $C_2 = v_0 \to v_3 \to v_4 \to v_1$. The set $S = \{v_0\}$ with $w(S) = 1$ is a minimum feedback vertex set: removing v_0 breaks both cycles, leaving a DAG with topological order (v_3, v_4, v_1, v_2). No 0-vertex set suffices since C_1 and C_2 overlap only at v_0 and v_1, and removing v_1 alone leaves $C_{1'} = v_0 \to v_3 \to v_4 \to v_1 \to v_2 \to v_0$.

```
$ pred create --example MinimumFeedbackVertexSet -o minimum-feedback-vertex-set.json
$ pred solve minimum-feedback-vertex-set.json
$ pred evaluate minimum-feedback-vertex-set.json --config 1,0,0,0,0
```

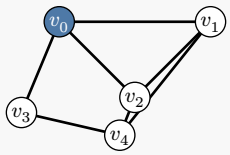


Figure 32: A directed graph with FVS $S = \{v_0\}$ (blue, $w(S) = 1$). Removing v_0 breaks both directed cycles $v_0 \to v_1 \to v_2 \to v_0$ and $v_0 \to v_3 \to v_4 \to v_1$, leaving a DAG.

Definition 2.44 (Partition into Paths of Length 2): Given $G = (V, E)$ with $|V| = 3q$, determine if V can be partitioned into q disjoint sets $V_1, ..., V_q$ of three vertices each, such that each V_t induces at least two edges in G.

- Complexity: `3^num_vertices`.
- Reduces to: BoundedComponentSpanningForest, ILP.

PartitionIntoPathsOfLength2	
graph	The underlying graph G=(V,E) with |V| divisible by 3

A classical NP-complete problem from Garey and Johnson [5, Ch. 3, p. 76], proved hard by reduction from 3-Dimensional Matching. Each triple in the partition must form a path of length 2 (exactly two edges, i.e., a P_3 subgraph) or a triangle (all three edges). The problem models constrained grouping scenarios where cluster connectivity is required. The best known exact approach uses subset DP in $O^*(3^n)$ time.

Example. Consider the graph G with $n = 9$ vertices and edges $\{0,1\}, \{1,2\}, \{3,4\}, \{4,5\}, \{6,7\}, \{7,8\}$ (plus cross-edges $\{0,3\}, \{2,5\}, \{3,6\}, \{5,8\}$). Setting $q = 3$, the partition $V_1 = \{0,1,2\}$, $V_2 = \{3,4,5\}$, $V_3 = \{6,7,8\}$ is valid: V_1 contains edges $\{0,1\}, \{1,2\}$ (path $0 — 1 — 2$), V_2 contains $\{3,4\}, \{4,5\}$, and V_3 contains $\{6,7\}, \{7,8\}$.

Definition 2.45 (Steiner Tree in Graphs): Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}_{\geq 0}$ and a set of terminal vertices $R \subseteq V$, find a subtree T of G that spans all terminals in R and minimizes the total edge weight $\sum_{e \in T} w(e)$.

- Complexity: `2^num_terminals * num_vertices^3`.
- Reduces to: ILP.

SteinerTreeInGraphs	
graph	The underlying graph G=(V,E)
terminals	Required terminal vertices R ⊆ V
edge_weights	Edge weights w: E -> R

A classical NP-complete problem from Karp's list (as "Steiner Tree in Graphs," Garey & Johnson ND12) [1]. Central to network design, VLSI layout, and phylogenetic reconstruction. The problem generalizes minimum spanning tree (where $R = V$) and shortest path (where $|R| = 2$). The Dreyfus–Wagner dynamic programming algorithm [33] solves it in $O\big(3^k \cdot n + 2^k \cdot n^2 + n^3\big)$ time, where $k = |R|$ and $n = |V|$. Bjorklund et al. [47] achieved $O^*\big(2^k\big)$ using subset convolution over the Mobius algebra, and Nederlof [48] gave an $O^*\big(2^k\big)$ polynomial-space algorithm.

Example. Consider a graph G with $n = 6$ vertices and $|E| = 7$ edges. The terminals are $R = \{v_0, v_3, v_5\}$ (blue). The optimal Steiner tree uses Steiner vertex v_2 (gray, dashed border) and edges $\{v_0, v_2\}$, $\{v_2, v_3\}$, $\{v_2, v_5\}$ with total weight $2 + 1 + 2 = 5$.

```
$ pred create --example SteinerTreeInGraphs -o steiner-tree-in-graphs.json
$ pred solve steiner-tree-in-graphs.json
$ pred evaluate steiner-tree-in-graphs.json --config 0,1,0,1,1,0,0
```

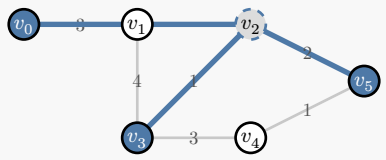


Figure 33: Steiner Tree: terminals $R = \{v_0, v_3, v_5\}$ (blue), Steiner vertex v_2 (dashed). Optimal tree (blue edges) has weight 5.

Definition 2.46 (Minimum Sum Multicenter): Given a graph $G = (V, E)$ with vertex weights $w : V \to \mathbb{Z}_{\geq 0}$, edge lengths $l : E \to \mathbb{Z}_{\geq 0}$, and a positive integer $K \leq |V|$, find a set $P \subseteq V$ of K vertices (centers) that minimizes the total weighted distance $\sum_{v \in V} w(v) \cdot d(v, P)$, where $d(v, P) = \min_{p \in P} d(v, p)$ is the shortest-path distance from v to the nearest center in P.

- Complexity: `2^num_vertices`.
- Reduces to: ILP.
- Reduces from: DecisionMinimumDominatingSet.

MinimumSumMulticenter	
graph	The underlying graph G=(V,E)
vertex_weights	Vertex weights w: V -> R
edge_lengths	Edge lengths l: E -> R
k	Number of centers to place

Also known as the *p-median problem*. This is a classical NP-complete facility location problem from Garey & Johnson (A2 ND51). The goal is to optimally place K service centers (e.g., warehouses, hospitals) to minimize total service cost. NP-completeness was established by Kariv and Hakimi (1979) via transformation from Dominating Set. The problem remains NP-complete even with unit weights and unit edge lengths, but is solvable in polynomial time for fixed K or when G is a tree.

The best known exact algorithm runs in $O^*(2^n)$ time by brute-force enumeration of all $\binom{n}{K}$ vertex subsets. Constant-factor approximation algorithms exist: Charikar et al. (1999) gave the first constant-factor result, and the best known ratio is $(2+\varepsilon)$ by Cohen-Addad et al. (STOC 2022).

Variables: $n = |V|$ binary variables, one per vertex. $x_v = 1$ if vertex v is selected as a center. A configuration is valid when exactly K centers are selected and all vertices are reachable from at least one center.

Example. Consider the graph G on 7 vertices with unit weights $w(v) = 1$ and unit edge lengths, edges $\{(0,1), (1,2), (2,3), (3,4), (4,5), (5,6), (0,6), (2,5)\}$, and $K = 2$. Placing centers at $P = \{v_2, v_5\}$ gives distances $d(v_0) = 2$, $d(v_1) = 1$, $d(v_2) = 0$, $d(v_3) = 1$, $d(v_4) = 1$, $d(v_5) = 0$, $d(v_6) = 1$, for a total cost of $2 + 1 + 0 + 1 + 1 + 0 + 1 = 6$. This is optimal.

```
$ pred create --example MinimumSumMulticenter -o minimum-sum-multicenter.json
$ pred solve minimum-sum-multicenter.json
$ pred evaluate minimum-sum-multicenter.json --config 0,0,1,0,0,1,0
```

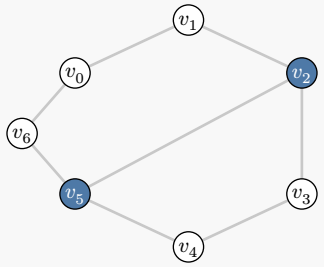


Figure 34: Minimum Sum Multicenter with $K = 2$ on a 7-vertex graph. Centers v_2 and v_5 (blue) achieve optimal total weighted distance 6.

Definition 2.47 (Min-Max Multicenter): Given a graph $G = (V, E)$ with vertex weights $w : V \to \mathbb{Z}_{\geq 0}$, edge lengths $l : E \to \mathbb{Z}_{\geq 0}$, and a positive integer $K \leq |V|$, find $S \subseteq V$ with $|S| = K$ that minimizes $\max_{v \in V} w(v) \cdot d(v, S)$, where $d(v, S) = \min_{s \in S} d(v, s)$ is the shortest weighted-path distance from v to the nearest vertex in S.

- Complexity: `1.4969^num_vertices`.
- Reduces to: ILP.
- Reduces from: DecisionMinimumDominatingSet.

MinMaxMulticenter	
graph	The underlying graph G=(V,E)
vertex_weights	Vertex weights w: V -> R
edge_lengths	Edge lengths l: E -> R
k	Number of centers to place

Also known as the *vertex p-center problem* (Garey & Johnson A2 ND50). The goal is to place K facilities so that the worst-case weighted distance from any demand point to its nearest facility is minimized. NP-hard even with unit weights and unit edge lengths (Kariv and Hakimi, 1979).

Closely related to Dominating Set: on unweighted unit-length graphs, a K-center with optimal radius 1 corresponds to a dominating set of size K. The best known exact algorithm runs in $O^*(1.4969^n)$ via binary search over distance thresholds combined with dominating set computation [30]. An optimal 2-approximation exists (Hochbaum and Shmoys, 1985); no $(2-\varepsilon)$-approximation is possible unless $P = \text{NP}$ (Hsu and Nemhauser, 1979).

Variables: $n = |V|$ binary variables, one per vertex. $x_v = 1$ if vertex v is selected as a center. The objective value is $\min_{|S|=K} \max_{v \in V} w(v) \cdot d(v, S)$; configurations with $|S| \neq K$ or unreachable vertices evaluate to $\bot$ (infeasible).

Example. Consider the graph G on 6 vertices with unit weights $w(v) = 1$, unit edge lengths, edges $\{(0,1), (1,2), (2,3), (3,4), (4,5), (0,5), (1,4)\}$, and $K = 2$. Placing centers at $S = \{v_1, v_4\}$ gives maximum distance $\max_v d(v, S) = 1$, which is optimal.

```
$ pred create --example MinMaxMulticenter -o min-max-multicenter.json
$ pred solve min-max-multicenter.json
$ pred evaluate min-max-multicenter.json --config 0,1,0,0,1,0
```

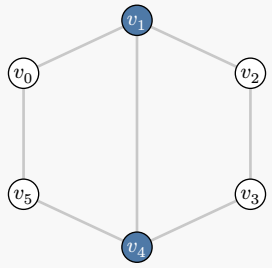


Figure 35: Min-Max Multicenter with $K = 2$ on a 6-vertex graph. Centers v_1 and v_4 (blue) achieve optimal maximum distance 1.

Definition 2.48 (Multiple Copy File Allocation): Given a graph $G = (V, E)$, usage values $u : V \to \mathbb{Z}_{>0}$, and storage costs $s : V \to \mathbb{Z}_{>0}$, find a subset $V' \subseteq V$ that minimizes $\sum_{v \in V'} s(v) + \sum_{v \in V} u(v) \cdot d(v, V')$, where $d(v, V') = \min_{w \in V'} d_{G(v,w)}$ is the shortest-path distance from v to the nearest copy vertex.

- Complexity: `2^num_vertices`.
- Reduces to: ILP.

MultipleCopyFileAllocation	
graph	The network graph G=(V,E)
usage	Usage frequencies u(v) for each vertex
storage	Storage costs s(v) for placing a copy at each vertex

Multiple Copy File Allocation appears in the storage-and-retrieval section of Garey and Johnson (SR6) [5]. The model combines two competing costs: each chosen copy vertex incurs a storage charge, while every vertex pays an access cost weighted by its demand and graph distance to the nearest copy. Garey and Johnson record the problem as NP-hard in the strong sense, even when usage and storage costs are uniform [5].

Example. Consider the 6-cycle C_6 with uniform usage $u(v) = 10$ and uniform storage $s(v) = 1$. Placing copies at every vertex $V' = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ gives storage cost $6 \cdot 1 = 6$ and access cost 0 (each vertex is distance 0 from its own copy), for a total cost of 6. This is optimal: removing any copy saves 1 in storage but adds at least 10 in access cost for each neighbor that must now reach a more distant copy.

```
$ pred create --example MultipleCopyFileAllocation -o multiple-copy-file-allocation.json
$ pred solve multiple-copy-file-allocation.json
$ pred evaluate multiple-copy-file-allocation.json --config 1,1,1,1,1,1
```

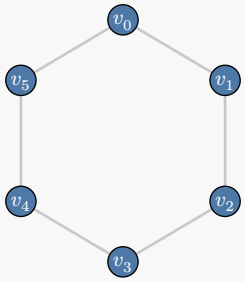


Figure 36: Multiple Copy File Allocation on a 6-cycle. All vertices (shown in blue) host copies; total cost is 6.

Definition 2.49 (Expected Retrieval Cost): Given a set $R = \{r_1, ..., r_n\}$ of records, access probabilities $p(r) \in [0, 1]$ with $\sum_{r \in R} p(r) = 1$, and a positive integer m of circular storage sectors, find a partition $R_1, ..., R_m$ of R that minimizes $\sum_{i=1}^{m} \sum_{j=1}^{m} p(R_i)p(R_j)d(i, j)$, where $p(R_i) = \sum_{r \in R_i} p(r)$ and $d(i, j) = j - i - 1$ for $1 \leq i < j \leq m$, while $d(i, j) = m - i + j - 1$ for $1 \leq j \leq i \leq m$.

- Complexity: `num_sectors ^ num_records`.
- Reduces to: ILP.

ExpectedRetrievalCost	
probabilities	Access probabilities p(r) for each record
num_sectors	Number of sectors on the drum-like device

Expected Retrieval Cost is storage-and-retrieval problem SR4 in Garey and Johnson [5]. The model abstracts a drum-like storage device with fixed read heads: placing probability mass evenly around the cycle reduces the expected waiting time until the next requested sector rotates under the head. Cody and Coffman introduced the formulation and analyzed exact and heuristic record-allocation algorithms for fixed numbers of sectors [49]. Garey and Johnson record that the general decision problem is NP-complete in the strong sense via transformations from Partition and 3-Partition [5]. The implementation in this repository uses one m-ary variable per record, so the registered exact baseline enumerates m^n assignments.

Example. Take six records with probabilities $(0.2, 0.15, 0.15, 0.2, 0.1, 0.2)$ and three sectors. Assign $R_1 = \{r_1, r_5\}$, $R_2 = \{r_2, r_4\}$, and $R_3 = \{r_3, r_6\}$. Then the sector masses are $(p(R_1), p(R_2), p(R_3)) = (0.3, 0.35, 0.35)$. For $m = 3$, the non-zero latencies are $d(1, 1) = d(2, 2) = d(3, 3) = 2$, $d(1, 3) = d(2, 1) = d(3, 2) = 1$, and the remaining pairs contribute 0. Hence the expected retrieval cost is 1.0025, which is optimal for this instance.

```
$ pred create --example ExpectedRetrievalCost -o expected-retrieval-cost.json
$ pred solve expected-retrieval-cost.json --solver brute-force
$ pred evaluate expected-retrieval-cost.json --config 0,1,2,1,0,2
```

Sector	Records	Mass
S_1	r_1, r_5	0.3
S_2	r_2, r_4	0.35
S_3	r_3, r_6	0.35

Table 1: Expected Retrieval Cost example with cyclic sector order $S_1 \to S_2 \to S_3 \to S_1$. The optimal allocation yields masses $(0.3, 0.35, 0.35)$ and minimum cost 1.0025.

2.2 Set Problems

Definition 2.50 (Maximum Set Packing): Given universe U, collection $\mathcal{S} = \{S_1, ..., S_m\}$ with $S_i \subseteq U$, weights $w : \mathcal{S} \to \mathbb{R}$, find $\mathcal{P} \subseteq \mathcal{S}$ maximizing $\sum_{S \in \mathcal{P}} w(S)$ s.t. $\forall S_i, S_j \in \mathcal{P} : S_i \cap S_j = \emptyset$.

- Complexity: `2^num_sets`.
- Reduces to: MaximumIndependentSet, MaximumSetPacking, QUBO, ILP.
- Reduces from: ExactCoverBy3Sets, MaximumIndependentSet, MaximumMatching, MaximumSetPacking.

MaximumSetPacking	
sets	Collection of sets over a universe
weights	Weight for each set

One of Karp's 21 NP-complete problems [1]. Generalizes maximum matching (the special case where all sets have size 2, solvable in polynomial time). Applications include resource allocation, VLSI design, and frequency assignment. The optimization version is as hard to approximate as maximum clique. The best known exact algorithm runs in $O^*(2^m)$ by brute-force enumeration over the m sets[6].

Example. Let $U = \{1, 2, ..., 5\}$ and $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$ with $S_1 = \$\{1, 2\}\$$, $S_2 = \$\{2, 3\}\$$, $S_3 = \$\{3, 4\}\$$, $S_4 = \$\{4, 5\}\$$, and unit weights $w(S_i) = 1$. A maximum packing is $\mathcal{P} = \{S_2, S_4\}$ with $w(\mathcal{P}) = 2$: $S_2 \cap S_4 = \emptyset$. Adding S_2 would conflict with both ($S_1 \cap S_2 = \{2\}$, $S_2 \cap S_3 = \{3\}$), and S_4 conflicts with S_3 ($S_3 \cap S_4 = \{4\}$). The alternative packing $\{S_2, S_4\}$ also achieves weight 2.

```
$ pred create --example MaximumSetPacking/i32 -o maximum-set-packing.json
$ pred solve maximum-set-packing.json
$ pred evaluate maximum-set-packing.json --config 0,1,0,1
```

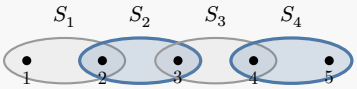


Figure 37: Maximum set packing: $\mathcal{P} = \{S_2, S_4\}$ (blue) are disjoint; S_1, S_3 (gray) conflict with the packing.

Definition 2.51 (Minimum Set Covering): Given universe U, collection $\mathcal{S}$ with weights $w : \mathcal{S} \to \mathbb{R}$, find $\mathcal{C} \subseteq \mathcal{S}$ minimizing $\sum_{S \in \mathcal{C}} w(S)$ s.t. $\bigcup_{S \in \mathcal{C}} S = U$.

- Complexity: `2^num_sets`.
- Reduces to: ILP.
- Reduces from: MinimumVertexCover.

MinimumSetCovering	
universe_size	Size of the universe U
sets	Collection of subsets of U
weights	Weight for each set

One of Karp's 21 NP-complete problems [1]. Arises in facility location, crew scheduling, and test suite minimization. The greedy algorithm achieves an $O(\ln n)$-approximation where $n = |U|$, which is essentially optimal: cannot be approximated within $(1 - o(1)) \ln n$ unless P = NP. The best known exact algorithm runs in $O^*(2^m)$ by brute-force enumeration over the m sets[7].

Example. Let $U = \{1, 2, ..., 5\}$ and $\mathcal{S} = \{S_1, S_2, S_3\}$ with $S_1 = \$\{1, 2, 3\}\$$, $S_2 = \$\{2, 4\}\$$, $S_3 = \$\{3, 4, 5\}\$$, and unit weights $w(S_i) = 1$. A minimum cover is $\mathcal{C} = \{S_1, S_3\}$ with $w(\mathcal{C}) = 2$: $S_1 \cup S_3 = \{1, 2, ..., 5\} = U$. No single set covers all of U, so at least two sets are required.

[6] No algorithm improving on brute-force enumeration is known for general weighted set packing.
[7] No algorithm improving on brute-force enumeration is known for general weighted set covering.

```
$ pred create --example MinimumSetCovering -o minimum-set-covering.json
$ pred solve minimum-set-covering.json
$ pred evaluate minimum-set-covering.json --config 1,0,1
```

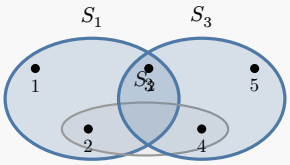


Figure 38: Minimum set covering: $\mathcal{C} = \{S_1, S_3\}$ (blue) cover all of U; S_2 (gray) is redundant.

Definition 2.52 (Minimum Hitting Set): Given a finite universe U and a collection $\mathcal{S} = \{S_1, ..., S_m\}$ of subsets of U, find a subset $H \subseteq U$ minimizing $|H|$ such that $H \cap S_i \neq \emptyset$ for every $i \in \{1, ..., m\}$.

- Complexity: `2^universe_size`.
- Reduces to: ILP.
- Reduces from: MinimumVertexCover.

MinimumHittingSet	
universe_size	Size of the universe U
sets	Collection of subsets of U that must each be hit

Minimum Hitting Set is one of Karp's 21 NP-complete problems [1]. It is the incidence-dual of Set Covering: transposing the set-element incidence matrix swaps the choice of sets with the choice of universe elements. Vertex Cover is the special case in which every set has size 2, so every edge is "hit" by selecting one of its endpoints.

A direct exact algorithm enumerates all 2^n subsets $H \subseteq U$ for $n = |U|$ and checks whether each subset intersects every member of $\mathcal{S}$. This yields an $O^*(2^n)$ exact algorithm[8].

Example. Let $U = \{1, 2, ..., 6\}$ and $\mathcal{S} = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7\}$ with $S_1 =$ \${1, 2, 3}\$, $S_2 =$ \${1, 4, 5}\$, $S_3 =$ \${2, 4, 6}\$, $S_4 =$ \${3, 5, 6}\$, $S_5 =$ \${1, 2, 6}\$, $S_6 =$ \${3, 4}\$, $S_7 =$ \${2, 5}\$. A minimum hitting set is $H =$ \${2, 4, 5}\$ with $|H| = 3$: every set in $\mathcal{S}$ contains at least one of the selected elements. No 2-element subset of U hits all 7 sets, so the optimum is exactly 3.

```
$ pred create --example MinimumHittingSet -o minimum-hitting-set.json
$ pred solve minimum-hitting-set.json
$ pred evaluate minimum-hitting-set.json --config 0,1,0,1,1,0
```

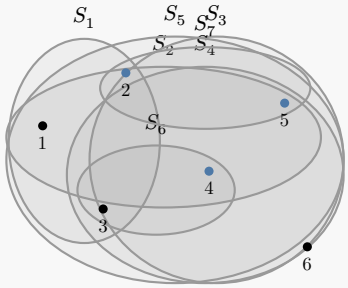


Figure 39: Minimum hitting set: the blue elements \${2, 4, 5}\$ intersect every set region $S_1, ..., S_7$, so they hit the entire collection $\mathcal{S}$.

[8]No exact worst-case algorithm improving on brute-force enumeration over the universe elements is recorded in the standard references used for this catalog entry.

Definition 2.53 (Set Splitting): Given a finite universe U and a collection $\mathcal{C} = \{C_1, ..., C_m\}$ of subsets of U each of size ≥ 2, does there exist a 2-coloring $\chi : U \to \{0, 1\}$ such that every C_i is non-monochromatic — i.e., contains at least one element of each color?

- Complexity: `2^universe_size`.
- Reduces to: Betweenness, ILP.
- Reduces from: NAESatisfiability.

```
SetSplitting
universe_size universe_size
subsets       Subsets that must each contain elements from both parts
```

One of Garey and Johnson's NP-complete problems (SP4 in [5]), shown NP-complete by reduction from 3-SAT. It is equivalent to deciding whether a hypergraph is 2-colorable (also called Property B). The problem is trivially satisfiable when every subset has size exactly 2, reducing to 2-colorability of the corresponding graph; it becomes NP-complete for subsets of size ≥ 3. The best known exact algorithm runs in $O^*(2^n)$ by brute-force enumeration over the $n = |U|$ elements.

Example. Let $U = \{1, 2, ..., 6\}$ and $\mathcal{C} = \{C_1, ..., C_4\}$ with $C_1 =$ \$\{1, 2, 3\}\$, $C_2 =$ \$\{3, 4, 5\}\$, $C_3 =$ \$\{1, 5, 6\}\$, $C_4 =$ \$\{2, 4, 6\}\$. Coloring $S_1 =$ \$\{2, 4, 5\}\$ and $S_2 =$ \$\{1, 3, 6\}\$ splits all subsets: each C_i has at least one element in each part.

```
$ pred create --example SetSplitting -o set-splitting.json
$ pred solve set-splitting.json
$ pred evaluate set-splitting.json --config 1,0,1,0,0,1
```

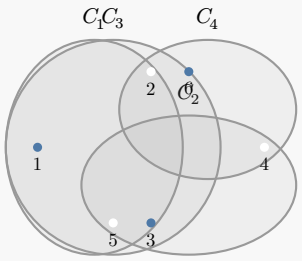


Figure 40: Set splitting: white elements \$\{2, 4, 5\}\$ form S_1 and blue elements \$\{1, 3, 6\}\$ form S_2; every subset $C_1, ..., C_4$ contains both colors.

Definition 2.54 (Consecutive Sets): Given a finite alphabet Σ of size m, a collection $\mathcal{C} = \{\Sigma_1, \Sigma_2, ..., \Sigma_n\}$ of subsets of Σ, and a positive integer K, determine whether there exists a string $w \in \Sigma^*$ with $|w| \leq K$ such that, for each i, the elements of Σ_i occur in a consecutive block of $|\Sigma_i|$ symbols of w.

- Complexity: `alphabet_size^bound_k * num_subsets`.

```
ConsecutiveSets
alphabet_size   Size of the alphabet (elements are 0..alphabet_size-1)
subsets         Collection of subsets of the alphabet
bound_k         Maximum string length K
```

This problem arises in information retrieval and file organization (SR18 in Garey and Johnson [5]). It generalizes the *consecutive ones property* from binary matrices to a string-based formulation: given subsets of an alphabet, construct the shortest string where each subset's elements appear contiguously. The problem is NP-complete, as shown by L. T. Kou [50] via reduction from Hamiltonian Path. The circular variant, where blocks may wrap around from the end of w back to its beginning (considering ww), is also NP-complete [51]. When K equals the number of distinct symbols appearing in the subsets, the problem reduces to testing a binary matrix for the consecutive ones property, which is solvable in linear time using PQ-tree algorithms [51].

Example. Let $\Sigma = \{0, 1, ..., 5\}$, $K = 6$, and $\mathcal{C} = \{\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4, \Sigma_5\}$ with $\Sigma_1 =$ \${0, 4}\$, $\Sigma_2 =$ \${2, 4}\$, $\Sigma_3 =$ \${2, 5}\$, $\Sigma_4 =$ \${1, 5}\$, $\Sigma_5 =$ \${1, 3}\$. A valid string is $w = (0, 4, 2, 5, 1, 3)$ with $|w| = 6 = K$: $\Sigma_1 = \{0, 4\}$ appears as the block $(0, 4)$ at positions 0–1, $\Sigma_2 = \{2, 4\}$ appears as $(4, 2)$ at positions 1–2, $\Sigma_3 = \{2, 5\}$ appears as $(2, 5)$ at positions 2–3, $\Sigma_4 = \{1, 5\}$ appears as $(5, 1)$ at positions 3–4, and $\Sigma_5 = \{1, 3\}$ appears as $(1, 3)$ at positions 4–5.

```
$ pred create --example ConsecutiveSets -o consecutive-sets.json
$ pred solve consecutive-sets.json
$ pred evaluate consecutive-sets.json --config 0,4,2,5,1,3
```

Definition 2.55 (Exact Cover by 3-Sets): Given universe X with $|X| = 3q$ and collection $\mathcal{C}$ of 3-element subsets of X, does $\mathcal{C}$ contain an exact cover — a subcollection $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| = q$ such that every element of X occurs in exactly one member of $\mathcal{C}'$?

- Complexity: `2^universe_size`.
- Reduces to: AlgebraicEquationsOverGF2, ILP, MaximumSetPacking, MinimumAxiomSet, MinimumFaultDetectionTestSet, MinimumWeightSolutionToLinearEquations, StaffScheduling, SubsetProduct.

ExactCoverBy3Sets	
universe_size	Size of universe X (must be divisible by 3)
subsets	Collection C of 3-element subsets of X

Shown NP-complete by Karp (1972) via transformation from 3-Dimensional Matching [1]. X3C remains NP-complete even when no element appears in more than three subsets, but is solvable in polynomial time when no element appears in more than two subsets. It is one of the most widely used source problems for NP-completeness reductions in Garey & Johnson (A3 SP2), serving as the starting point for proving hardness of problems in scheduling, graph theory, set systems, coding, and number theory. The best known exact algorithm runs in $O^*(2^n)$ via inclusion-exclusion over the $n = |X|$ universe elements; a direct brute-force search over the m subsets gives the weaker $O^*(2^m)$ bound.

Example. Let $X = \{1, 2, ..., 9\}$ ($q = 3$) and $\mathcal{C} = \{S_1, ..., S_7\}$ with $S_1 =$ \${1, 2, 3}\$, $S_2 =$ \${1, 3, 5}\$, $S_3 =$ \${4, 5, 6}\$, $S_4 =$ \${4, 6, 8}\$, $S_5 =$ \${7, 8, 9}\$, $S_6 =$ \${2, 5, 7}\$, $S_7 =$ \${3, 6, 9}\$. An exact cover is $\mathcal{C}' = \{S_1, S_3, S_5\}$: S_1 covers \${1, 2, 3}\$, S_3 covers \${4, 5, 6}\$, S_5 covers \${7, 8, 9}\$, their union is X, and they are pairwise disjoint with $|\mathcal{C}'| = 3 = q$.

```
$ pred create --example ExactCoverBy3Sets -o exact-cover-by-3-sets.json
$ pred solve exact-cover-by-3-sets.json
$ pred evaluate exact-cover-by-3-sets.json --config 1,0,1,0,1,0,0
```

Definition 2.56 (Three-Dimensional Matching): Given disjoint sets W, X, Y each with q elements and a set $M \subseteq W \times X \times Y$ of triples, does M contain a *matching* — a subset $M' \subseteq M$ with $|M'| = q$ such that no two triples in M' agree in any coordinate?

- Complexity: `2^num_triples`.
- Reduces to: ILP, ThreePartition.

ThreeDimensionalMatching	
universe_size	Size of each set W, X, Y (q)
triples	Set M of triples (w, x, y)

Shown NP-complete by Karp (1972) [1]. Three-Dimensional Matching (3DM) is one of the six basic NP-complete problems in Garey & Johnson (A3 SP1) and is closely related to Exact Cover by 3-Sets. While X3C asks for a perfect partition of a single universe into disjoint triples, 3DM asks for a system of

distinct representatives across three separate dimensions. The problem remains NP-complete even when each element appears in at most three triples. The direct brute-force algorithm runs in $O^*(2^m)$ time where $m = |M|$.

Example. Let $W = X = Y = \{1, 2, ..., 3\}$ and $M = \{t_1, ..., t_5\}$ with $t_1 =$ \$(1, 2, 3)\$, $t_2 =$ \$(2, 1, 2)\$, $t_3 =$ \$(3, 3, 1)\$, $t_4 =$ \$(1, 1, 1)\$, $t_5 =$ \$(2, 3, 3)\$. A valid matching is $M' = \{t_1, t_2, t_3\}$: the W-coordinates, X-coordinates, and Y-coordinates are each pairwise distinct, and $|M'| = 3 = q$.

```
$ pred create --example ThreeDimensionalMatching -o three-dimensional-matching.json
$ pred solve three-dimensional-matching.json
$ pred evaluate three-dimensional-matching.json --config 1,1,1,0,0
```

Definition 2.57 (Three-Matroid Intersection): Given three partition matroids $(E, \mathcal{F}_1)$, $(E, \mathcal{F}_2)$, $(E, \mathcal{F}_3)$ on a common ground set E with $|E| = n$, and a positive integer $K \leq n$, does there exist a subset $E' \subseteq E$ with $|E'| = K$ that is independent in all three matroids? A partition matroid partitions E into groups; a set S is independent if $|S \cap G| \leq 1$ for every group G.

- Complexity: `2^ground_set_size`.

`ThreeMatroidIntersection`

`ground_set_size`	Number of elements in the ground set E
`partitions`	Three partition matroids, each as a list of groups
`bound`	Required size K of the common independent set

Three-Matroid Intersection is problem SP11 in Garey & Johnson [5] (section A3). While 2-matroid intersection is solvable in polynomial time (Edmonds, 1970) [52], the jump to three matroids captures NP-hardness. NP-completeness is established by transformation from Three-Dimensional Matching, where each dimension induces a partition matroid. The restriction to partition matroids suffices for NP-completeness.

Doron-Arad, Kulik, and Shachnai (2024) [53] showed that brute force essentially cannot be beaten: any algorithm requires $\Omega\left(2^{n-5\sqrt{n}\log n}\right)$ oracle queries. A marginal improvement to $2^{n-\Omega(\log^2 n)}$ exists via Monotone Local Search [54]. The direct brute-force algorithm runs in $O^*(2^n)$ time where $n = |E|$.

Example. Let $E = \{0, 1, ..., 5\}$ with $K = 2$. The three partition matroids have groups: $\mathcal{F}_1$: \$\{0, 1, 2\}\$, \$\{3, 4, 5\}\$ $\mathcal{F}_2$: \$\{0, 3\}\$, \$\{1, 4\}\$, \$\{2, 5\}\$ $\mathcal{F}_3$: \$\{0, 4\}\$, \$\{1, 5\}\$, \$\{2, 3\}\$. The subset $E' =$ \$\{0, 5\}\$ is a valid common independent set of size 2: each matroid has at most one selected element per group.

```
$ pred create --example ThreeMatroidIntersection -o three-matroid-intersection.json
$ pred solve three-matroid-intersection.json
$ pred evaluate three-matroid-intersection.json --config 1,0,0,0,0,1
```

Definition 2.58 (Comparative Containment): Given a finite universe X, two set families $\mathcal{R} = \{R_1, ..., R_k\}$ and $\mathcal{S} = \{S_1, ..., S_l\}$ over X, and positive integer weights $w_{R(R_i)}$ and $w_{S(S_j)}$, does there exist a subset $Y \subseteq X$ such that $\sum_{Y \subseteq R_i} w_{R(R_i)} \geq \sum_{Y \subseteq S_j} w_{S(S_j)}$?

- Complexity: `2^universe_size`.

`ComparativeContainment`

`universe_size`	Size of the universe X
`r_sets`	First set family R over X
`s_sets`	Second set family S over X
`r_weights`	Positive weights for sets in R
`s_weights`	Positive weights for sets in S

Comparative Containment is the set-system comparison problem SP10 in Garey & Johnson [5]. Unlike covering and packing problems, feasibility depends on how the chosen subset Y is nested inside two competing set families: the $\mathcal{R}$ family rewards containment while the $\mathcal{S}$ family penalizes it. The problem remains NP-complete in the unit-weight special case and provides a clean weighted-set comparison primitive for future reduction entries in this catalog.

A direct exact algorithm enumerates all 2^n subsets $Y \subseteq X$ for $n = |X|$ and checks which members of $\mathcal{R}$ and $\mathcal{S}$ contain each candidate. This yields an $O^*(2^n)$ exact algorithm, with the polynomial factor coming from scanning the $k + l$ sets for each subset[9].

Example. Let $X = \{1, 2, ..., 4\}$, $\mathcal{R} = \{R_1, R_2\}$ with $R_1 = \$\{1, 2, 3, 4\}\$$ with $w_{R(R_1)} = 2$, $R_2 = \$\{1, 2\}\$$ with $w_{R(R_2)} = 5$, and $\mathcal{S} = \{S_1, S_2\}$ with $S_1 = \$\{1, 2, 3, 4\}\$$ with $w_{S(S_1)} = 3$, $S_2 = \$\{3, 4\}\$$ with $w_{S(S_2)} = 6$. The subset $Y = \$\{2\}\$$ is satisfying because R_1, R_2 contribute 7 on the left while S_1 contribute only 3 on the right, so $7 \geq 3$. In fact, the satisfying subsets are $\$\{2\}\$$, so this instance has exactly 1 satisfying solutions.

```
$ pred create --example ComparativeContainment -o comparative-containment.json
$ pred solve comparative-containment.json
$ pred evaluate comparative-containment.json --config 0,1,0,0
```

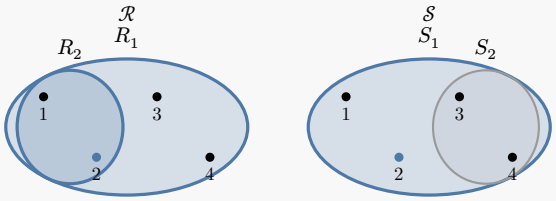


Figure 41: Comparative containment for $Y = \$\{2\}\$$: both R_1 and R_2 contain Y, while only S_1 does, so the $\mathcal{R}$ side dominates the $\mathcal{S}$ side.

Definition 2.59 (Set Basis): Given finite set S, collection $\mathcal{C}$ of subsets of S, and integer k, does there exist a family $\mathcal{B} = \{B_1, ..., B_k\}$ with each $B_i \subseteq S$ such that for every $C \in \mathcal{C}$ there exists $\mathcal{B}_C \subseteq \mathcal{B}$ with $\bigcup_{B \in \mathcal{B}_C} B = C$?

- Complexity: `2^(basis_size * universe_size)`.

SetBasis	
universe_size	Size of the ground set S
collection	Collection C of target subsets of S
k	Required number of basis sets

The Set Basis problem was shown NP-complete by Stockmeyer [55] and appears as SP7 in Garey & Johnson [5]. It asks for an exact union-based description of a family of sets, unlike Set Cover which only requires covering the underlying universe. Applications include data compression, database schema design, and Boolean function minimization. The library's decision encoding uses $k\,|S|$ membership bits, so brute-force over those bits gives an $O^*\left(2^{k\,|S|}\right)$ exact algorithm[10].

Example. Let $S = \{1, 2, 3, 4\}$, $k = 3$, and $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$ with $C_1 = \$\{1, 2\}\$$, $C_2 = \$\{2, 3\}\$$, $C_3 = \$\{1, 3\}\$$, $C_4 = \$\{1, 2, 3\}\$$. The sample basis from the issue is $\mathcal{B} = \{B_1, B_2, B_3\}$ with $B_1 = \$\{3\}\$$, $B_2 = \$\{2\}\$$, $B_3 = \$\{1\}\$$. Then $C_1 = B_1 \cup B_2$, $C_2 = B_2 \cup B_3$, $C_3 = B_1 \cup B_3$, and $C_4 = B_1 \cup B_2 \cup B_3$. The fixture

[9] No specialized exact algorithm improving on brute-force enumeration is recorded in the standard references used for this catalog entry.

[10] This is the direct search bound induced by the encoding implemented here; we are not aware of a faster general exact worst-case algorithm for this representation.

stores one satisfying encoding; other valid encodings exist (e.g., permuting the singleton basis or using the three pair sets C_1, C_2, C_3 as a basis).

```
$ pred create --example SetBasis -o set-basis.json
$ pred solve set-basis.json
$ pred evaluate set-basis.json --config 0,0,1,0,0,1,0,0,1,0,0,0
```

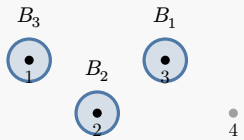


Figure 42: Set Basis example: the singleton basis $\mathcal{B} = \{B_1, B_2, B_3\}$ reconstructs every target set in $\mathcal{C}$; element 4 is unused by the target family.

Definition 2.60 (Prime Attribute Name): Given a set $A = \{0, 1, ..., 5\}$ of attribute names, a collection F of functional dependencies on A, and a specified attribute $x \in A$, determine whether x is a *prime attribute* for $\langle A, F \rangle$ — i.e., whether there exists a candidate key K for $\langle A, F \rangle$ such that $x \in K$.

A *candidate key* is a minimal subset $K \subseteq A$ whose closure $K_F^+ = A$, where the closure K_F^+ is the set of all attributes functionally determined by K under F.

- Complexity: `2^num_attributes * num_dependencies * num_attributes`.

PrimeAttributeName	
num_attributes	Number of attributes
dependencies	Functional dependencies (lhs, rhs) pairs
query_attribute	The query attribute index

Classical NP-complete problem from relational database theory (Lucchesi and Osborn, 1978; Garey & Johnson SR28). Prime attributes are central to database normalization: Second Normal Form (2NF) requires that no non-prime attribute is partially dependent on any candidate key, and Third Normal Form (3NF) requires that for every non-trivial functional dependency $X \to Y$, either X is a superkey or Y consists only of prime attributes. The brute-force approach enumerates all 2^n subsets of A containing x, checking each for the key property; no algorithm significantly improving on this is known for the general problem.

Example. Let $A = \{0, 1, ..., 5\}$ ($n = 6$), query attribute $x = 3$, and $F = \{\{0, 1\} \to \{2, 3, 4, 5\}, \{2, 3\} \to \{0, 1, 4, 5\}, \{0, 3\} \to \{1, 2, 4, 5\}\}$. The subset $K = \{2, 3\}$ is a candidate key containing $x = 3$: its closure is $K_F^+ = A$ (since $\{2, 3\} \to \{0, 1, 4, 5\}$ by the second FD, yielding all of A), and removing either element breaks the superkey property ($\{2\} \not\to A$ and $\{3\} \not\to A$), so K is minimal. Thus attribute 3 is prime. There are 2 candidate keys containing attribute 3 in total.

```
$ pred create --example PrimeAttributeName -o prime-attribute-name.json
$ pred solve prime-attribute-name.json
$ pred evaluate prime-attribute-name.json --config 0,0,1,1,0,0
```

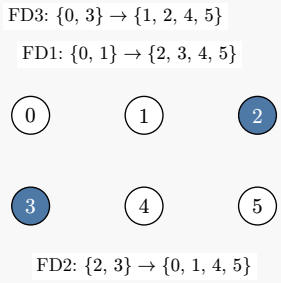


Figure 43: Prime Attribute Name instance with $n = 6$ attributes. Candidate key $K = \{2, 3\}$ is highlighted in blue; query attribute $x = 3$ is a member of K. The three functional dependencies determine the closure of every subset.

Definition 2.61 (Minimum Cardinality Key): Given a set A of attribute names and a collection F of functional dependencies (ordered pairs of subsets of A), find a key $K \subseteq A$ of minimum cardinality, i.e., a subset K such that the closure of K under F^* equals A and $|K|$ is minimized.

- Complexity: `2^num_attributes`.

MinimumCardinalityKey	
num_attributes	Number of attributes in the relation
dependencies	Functional dependencies as (lhs, rhs) pairs

The Minimum Cardinality Key problem arises in relational database theory, where identifying the smallest candidate key determines the most efficient way to uniquely identify rows in a relation. It was shown NP-complete by Lucchesi and Osborn (1978) [56] via transformation from Vertex Cover. The problem appears as SR26 in Garey & Johnson (A4) [5]. The closure F^* is defined by Armstrong's axioms: reflexivity ($B \subseteq C$ implies $C \to B$), transitivity, and union. The best known exact algorithm is brute-force enumeration of all subsets of A, giving $O^*(2^{|A|})$ time[11].

Example. Let $A = \{0, 1, ..., 5\}$ ($|A| = 6$) with functional dependencies $F = \{\$\{0, 1\}\$ \$arrow.r\$ \$\{2\}\$, \$\{0, 2\}\$ \$arrow.r\$ The optimal key $K = \$\{0, 1\}\$$ has $|K| = 2$. Its closure: start with $\{0, 1\}$; apply $\{0, 1\} \to \{2\}$ to get $\{0, 1, 2\}$; apply $\{0, 2\} \to \{3\}$ to get $\{0, 1, 2, 3\}$; apply $\{1, 3\} \to \{4\}$ to get $\{0, 1, 2, 3, 4\}$; apply $\{2, 4\} \to \{5\}$ to get A. Neither $\{0\}$ nor $\{1\}$ alone determines A, so K is a minimum-cardinality key.

```
$ pred create --example MinimumCardinalityKey -o minimum-cardinality-key.json
$ pred solve minimum-cardinality-key.json
$ pred evaluate minimum-cardinality-key.json --config 1,1,0,0,0,0
```

Definition 2.62 (Rooted Tree Storage Assignment): Given a finite set $X = \{0, 1, ..., 4\}$, a collection $\mathcal{C} = \{X_1, ..., X_m\}$ of subsets of X, and a nonnegative integer K, find a directed rooted tree $T = (X, A)$ and supersets $X_{i'} \supseteq X_i$ such that every $X_{i'}$ forms a directed path in T and $\sum_{i=1}^{m} |X_{i'} \setminus X_i| \leq K$.

- Complexity: `universe_size^universe_size`.
- Reduces to: ILP.
- Reduces from: RootedTreeArrangement.

RootedTreeStorageAssignment	
universe_size	Size of the ground set X
subsets	Collection of subsets of X
bound	Upper bound K on the total extension cost

[11] Lucchesi and Osborn give an output-polynomial algorithm for enumerating all candidate keys, but the number of keys can be exponential.

Rooted Tree Storage Assignment is the storage-and-retrieval problem SR5 in Garey and Johnson [5]. Their catalog credits a reduction from Rooted Tree Arrangement, framing the problem as hierarchical file organization: pick a rooted tree on the records so every request set can be completed to a single root-to-leaf path using only a limited number of extra records. The implementation here uses one parent variable per element of X, so the direct exhaustive bound is $|X|^{|X|}$ candidate parent arrays, filtered down to valid rooted trees[12].

Example. Let $X = \{0, 1, ..., 4\}$, $K = 1$, and $\mathcal{C} = \{X_1, X_2, X_3, X_4\}$ with $X_1 = \${0, 2}\$$, $X_2 = \${1, 3}\$$, $X_3 = \${0, 4}\$$, $X_4 = \${2, 4}\$$. The satisfying parent array $p = (0, 0, 0, 1, 2)$ encodes the rooted tree with arcs $(0, 1)$, $(0, 2)$, $(1, 3)$, $(2, 4)$. In this tree, $X_1 = \{0, 2\}$, $X_2 = \{1, 3\}$, and $X_4 = \{2, 4\}$ are already directed paths. The only extension is $X_3 = \{0, 4\}$, which becomes $X_{3'} = \{0, 2, 4\}$ along the path $0 \to 2 \to 4$, so the total extension cost is exactly $1 = K$.

```
$ pred create --example RootedTreeStorageAssignment -o rooted-tree-storage-assignment.json
$ pred solve rooted-tree-storage-assignment.json --solver brute-force
$ pred evaluate rooted-tree-storage-assignment.json --config 0,0,0,1,2
```

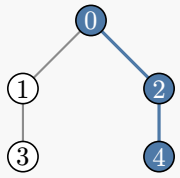


Figure 44: Rooted Tree Storage Assignment example. The rooted tree encoded by $p = (0, 0, 0, 1, 2)$ is shown; the blue path $0 \to 2 \to 4$ is the unique extension needed to realize $X_3 = \{0, 4\}$ within total cost $K = 1$.

Definition 2.63 (2-Dimensional Consecutive Sets): Given finite alphabet $\Sigma = \{0, 1, ..., n-1\}$ and collection $\mathcal{C} = \{\Sigma_1, ..., \Sigma_m\}$ of subsets of Σ, determine whether Σ can be partitioned into disjoint sets $X_1, X_2, ..., X_k$ such that each X_i has at most one element in common with each Σ_j, and for each $\Sigma_j \in \mathcal{C}$ there is an index $l(j)$ with $\Sigma_j \subseteq X_{l(j)} \cup X_{l(j)+1} \cup \cdots \cup X_{l(j)+|\Sigma_j|-1}$.

- Complexity: `alphabet_size^alphabet_size`.
- Reduces from: KColoring.

TwoDimensionalConsecutiveSets	
alphabet_size	Size of the alphabet (elements are 0..alphabet_size-1)
subsets	Collection of subsets of the alphabet

This problem generalizes the Consecutive Sets problem (SR18) by requiring not just that each subset's elements appear consecutively in an ordering, but that they be spread across consecutive groups of a partition where each group contributes at most one element per subset. Shown NP-complete by Lipski [57] via transformation from Graph 3-Colorability. The problem arises in information storage and retrieval where records must be organized in contiguous blocks. It remains NP-complete if all subsets have at most 5 elements, but is solvable in polynomial time if all subsets have at most 2 elements. The brute-force algorithm assigns each of n symbols to one of up to n groups, giving $O^*(n^n)$ time[13].

Example. Let $\Sigma = \{0, 1, ..., 5\}$ and $\mathcal{C} = \{\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4, \Sigma_5\}$ with $\Sigma_1 = \${0, 1, 2}\$$, $\Sigma_2 = \${3, 4, 5}\$$, $\Sigma_3 = \${1, 3}\$$, $\Sigma_4 = \${2, 4}\$$, $\Sigma_5 = \${0, 5}\$$. A valid partition uses $k = 4$ groups: $X_1 = \${0}\$$, $X_2 = \${1, 5}\$$, $X_3 = \${2, 3}\$$, $X_4 = \${4}\$$. Each group intersects every subset in at most one element, and each subset's elements span exactly $|\Sigma_j|$ consecutive groups. For instance, $\Sigma_1 = \{0, 1, 2\}$ maps to groups

[12]No exact algorithm improving on the direct parent-array search bound is claimed here for the general formulation.
[13]No algorithm improving on brute-force enumeration is known for this problem.

X_1, X_2, X_3 (consecutive), and $\Sigma_5 = \{0, 5\}$ maps to groups X_1, X_2 (consecutive). Multiple valid partitions exist for this instance, differing only by unused or shifted group labels.

```
$ pred create --example TwoDimensionalConsecutiveSets -o two-dimensional-consecutive-sets.json
$ pred solve two-dimensional-consecutive-sets.json
$ pred evaluate two-dimensional-consecutive-sets.json --config 0,1,2,2,3,1
```

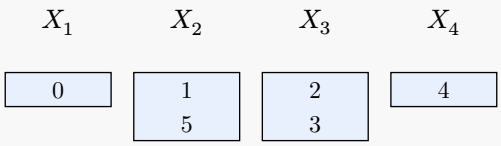


Figure 45: 2-Dimensional Consecutive Sets: partition of $\Sigma = \{0, ..., 5\}$ into 4 groups satisfying intersection and consecutiveness constraints for all 5 subsets.

2.3 Optimization Problems

Definition 2.64 (Spin Glass): Given n spin variables $s_i \in \{-1, +1\}$, pairwise couplings $J_{ij} \in \mathbb{R}$, and external fields $h_i \in \mathbb{R}$, minimize the Hamiltonian (energy function): $H(\boldsymbol{s}) = -\sum_{(i,j)} J_{ij} s_i s_j - \sum_i h_i s_i$.

- Complexity: `2^num_spins`.
- Reduces to: QUBO, MaxCut, SpinGlass.
- Reduces from: CircuitSAT, MaxCut, QUBO, SpinGlass.

```
SpinGlass
graph     The interaction graph
couplings Pairwise couplings J_ij
fields    On-site fields h_i
```

The Ising spin glass is the canonical model in statistical mechanics for disordered magnetic systems [6]. Ground-state computation is NP-hard on general interaction graphs but polynomial-time solvable on planar graphs without external field ($h_i = 0$) via reduction to minimum-weight perfect matching. Central to quantum annealing, where hardware natively encodes spin Hamiltonians. The best known general algorithm runs in $O^*(2^n)$ by brute-force enumeration[14].

Example. Consider $n = 5$ spins on a triangular lattice with uniform antiferromagnetic couplings $J_{ij} = -1$ for all edges and no external field ($h_i = 0$). The Hamiltonian simplifies to $H(\boldsymbol{s}) = \sum_{(i,j)} s_i s_j$, which counts parallel pairs minus antiparallel pairs. The lattice contains 7 edges and 3 triangular faces; since each triangle cannot have all three pairs antiparallel, frustration is unavoidable. A ground state is $\boldsymbol{s} =$ (-, +, -, -, +) achieving $H = -3$: 5 edges are satisfied (antiparallel) and 2 are frustrated (parallel). No configuration can satisfy more than 5 of 7 edges.

```
$ pred create --example SpinGlass -o spinglass.json
$ pred solve spinglass.json
$ pred evaluate spinglass.json --config 1,0,1,1,0
```

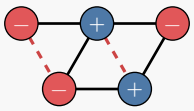


Figure 46: Triangular lattice with $n = 5$ spins and antiferromagnetic couplings ($J = -1$). Ground state $\boldsymbol{s} =$ (-, +, -, -, +) with $H = -3$. Solid edges: satisfied (antiparallel); dashed red: frustrated (parallel).

[14] On general interaction graphs, no algorithm improving on brute-force enumeration is known.

Definition 2.65 (QUBO): Given n binary variables $x_i \in \{0,1\}$, upper-triangular matrix $Q \in \mathbb{R}^{n \times n}$, minimize $f(\boldsymbol{x}) = \sum_{i=1}^{n} Q_{ii} x_i + \sum_{i<j} Q_{ij} x_i x_j$ (using $x_i^2 = x_i$ for binary variables).

- Complexity: `2^num_vars`.
- Reduces to: ILP, SpinGlass.
- Reduces from: ClosestVectorProblem, GraphPartitioning, ILP, KColoring, KSatisfiability, Knapsack, MaximumSetPacking, MinimumMultiwayCut, PaintShop, SpinGlass, TravelingSalesman.

```
QUBO
num_vars Number of binary variables
matrix   Upper-triangular Q matrix
```

Equivalent to the Ising model via the linear substitution $s_i = 2x_i - 1$. The native formulation for quantum annealing hardware (e.g., D-Wave) and a standard target for penalty-method reductions [58]. QUBO unifies many combinatorial problems into a single unconstrained binary framework, making it a universal intermediate representation for quantum and classical optimization. The best known general algorithm runs in $O^*(2^n)$ by brute-force enumeration[15].

Example. Consider $n = 3$ with $Q = (-1, 2, 0; 0, -1, 2; 0, 0, -1)$. The objective is $f(\boldsymbol{x}) = -x_1 - x_2 - x_3 + 2x_1x_2 + 2x_2x_3$. Evaluating all 2^3 assignments: $f(0,0,0) = 0$, $f(1,0,0) = -1$, $f(0,1,0) = -1$, $f(0,0,1) = -1$, $f(1,1,0) = 0$, $f(0,1,1) = 0$, $f(1,0,1) = -2$, $f(1,1,1) = 1$. The minimum is $f^* = -2$ at $\boldsymbol{x}^* = (1, 0, 1)$: selecting x_1 and x_3 avoids the penalty terms $2x_1x_2$ and $2x_2x_3$.

```
$ pred create --example QUBO -o qubo.json
$ pred solve qubo.json
$ pred evaluate qubo.json --config 1,0,1
```

Definition 2.66 (Integer Linear Programming): Given n variables $\boldsymbol{x}$ over a domain $\mathcal{D}$ (binary $\mathcal{D} = \{0,1\}$ or integer $\mathcal{D} = \mathbb{Z}_{\geq 0}$), constraint matrix $A \in \mathbb{R}^{m \times n}$, bounds $\boldsymbol{b} \in \mathbb{R}^m$, and objective $\boldsymbol{c} \in \mathbb{R}^n$, solve

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathcal{D}^n \end{aligned}$$

.

- Complexity: `2^num_vars`; `num_vars^num_vars`.
- Reduces to: ILP, QUBO.
- Reduces from: AcyclicPartition, BMF, BalancedCompleteBipartiteSubgraph, BicliqueCover, BiconnectivityAugmentation, BinPacking, BottleneckTravelingSalesman, BoundedComponentSpanningForest, CapacityAssignment, CircuitSAT, Clustering, ConsecutiveBlockMinimization, ConsecutiveOnesMatrixAugmentation, ConsecutiveOnesSubmatrix, ConsistencyOfDatabaseFrequencyTables, DirectedHamiltonianPath, DirectedTwoCommodityIntegralFlow, DisjointConnectingPaths, ExactCoverBy3Sets, ExpectedRetrievalCost, Factoring, FeasibleRegisterAssignment, FlowShopScheduling, GraphPartitioning, HamiltonianPath, ILP, IntegerKnapsack, IntegralFlowBundles, IntegralFlowHomologousArcs, IntegralFlowWithMultipliers, IsomorphicSpanningTree, KClique, KColoring, Knapsack, LengthBoundedDisjointPaths, LongestCircuit, LongestCommonSubsequence, LongestPath, MaximalIS, Maximum2Satisfiability, MaximumClique, MaximumDomaticNumber, MaximumLeafSpanningTree, MaximumLikelihoodRanking, MaximumMatching, MaximumSetPacking, MinMaxMulticenter, MinimumCapacitatedSpanningTree, MinimumCoveringByCliques, MinimumCutIntoBoundedSets, MinimumDominatingSet, MinimumEdgeCostFlow, MinimumExternalMacroDataCompression, MinimumFaultDetectionTestSet, MinimumFeedbackArcSet, MinimumFeedbackVertexSet, MinimumGraphBandwidth, MinimumHittingSet, MinimumInternalMacroDataCompression, MinimumMatrixCover, MinimumMaximalMatching, MinimumMetricDimension, MinimumMultiwayCut, MinimumSetCovering, MinimumSumMulticenter, MinimumTardinessSequencing, MinimumWeightDecoding, MixedChinesePostman, MonochromaticTriangle, MultipleCopyFileAllocation, MultiprocessorScheduling, NAESatisfiability, NumericalMatchingWithTargetSums, OpenShopScheduling, OptimalLinearArrangement, Optimum-

[15] QUBO inherits the Ising model's complexity; no algorithm improving on brute-force is known for the general case.

CommunicationSpanningTree, PaintShop, PartiallyOrderedKnapsack, PartitionIntoPathsOfLength2, PartitionIntoTriangles, PathConstrainedNetworkFlow, PrecedenceConstrainedScheduling, PreemptiveScheduling, QUBO, QuadraticAssignment, RectilinearPictureCompression, RegisterSufficiency, ResourceConstrainedScheduling, RootedTreeStorageAssignment, RuralPostman, SchedulingToMinimizeWeightedCompletionTime, SchedulingWithIndividualDeadlines, SequencingToMinimizeMaximumCumulativeCost, SequencingToMinimizeTardyTaskWeight, SequencingToMinimizeWeightedCompletionTime, SequencingToMinimizeWeightedTardiness, SequencingWithDeadlinesAndSetUpTimes, SequencingWithReleaseTimesAndDeadlines, SequencingWithinIntervals, SetSplitting, ShortestCommonSupersequence, ShortestWeightConstrainedPath, SparseMatrixCompression, StackerCrane, SteinerTree, SteinerTreeInGraphs, StringToStringCorrection, StrongConnectivityAugmentation, SubgraphIsomorphism, SumOfSquaresPartition, ThreeDimensionalMatching, TimetableDesign, TravelingSalesman, UndirectedFlowLowerBounds, UndirectedTwoCommodityIntegralFlow.

```
ILP
num_vars    Number of integer variables
constraints Linear constraints
objective   Sparse objective coefficients
sense       Optimization direction
```

Integer Linear Programming is a universal modeling framework: virtually every NP-hard combinatorial optimization problem admits an ILP formulation. Relaxing integrality to $\boldsymbol{x} \in \mathbb{R}^n$ yields a linear program solvable in polynomial time, forming the basis of branch-and-bound solvers. When the number of integer variables n is fixed, ILP is solvable in polynomial time by Lenstra's algorithm [59] using the geometry of numbers, making it fixed-parameter tractable in n. The best known general algorithm achieves $O^*(n^n)$ via an FPT algorithm based on lattice techniques [60].

Example. Minimize $\boldsymbol{c}^\top \boldsymbol{x} = \text{-}5x_1\text{-}6x_2$ subject to $x_1 + x_2 \leq 5$, $4x_1 + 7x_2 \leq 28$, x_1,$x_2 \geq 0$, $\boldsymbol{x} \in \mathbb{Z}^2$. The LP relaxation optimum is $p_1 = (7/3, 8/3) \approx (2.33, 2.67)$ with value ≈ -27.67, which is non-integral. Branch-and-bound yields the ILP optimum $\boldsymbol{x}^* = (3,\ 2)$ with $\boldsymbol{c}^\top \boldsymbol{x}^* = -27$.

```
$ pred create --example ILP/i32 -o ilp.json
$ pred solve ilp.json
$ pred evaluate ilp.json --config 3,2
```

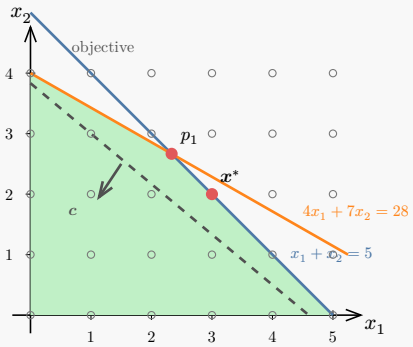


Figure 47: ILP feasible region (green) with constraints $x_1 + x_2 \leq 5$ (blue) and $4x_1 + 7x_2 \leq 28$ (orange). Hollow circles mark the integer lattice. The LP relaxation optimum $p_1 = (7/3, 8/3)$ is non-integral; the ILP optimum $\boldsymbol{x}^* = (3,\ 2)$ gives $\boldsymbol{c}^\top \boldsymbol{x}^* = -27$.

Definition 2.67 (Quadratic Assignment): Given n facilities and m locations ($n \leq m$), a flow matrix $C \in \mathbb{Z}^{n \times n}$ representing flows between facilities, and a distance matrix $D \in \mathbb{Z}^{m \times m}$ representing distances between locations, find an injective assignment $f : \{1, ..., n\} \to \{1, ..., m\}$ that minimizes

$$\sum_{i \neq j} C_{ij} \cdot D_{f(i),f(j)}.$$

- Complexity: `factorial(num_facilities)`.
- Reduces to: ILP.
- Reduces from: HamiltonianCircuit.

QuadraticAssignment	
cost_matrix	Flow/cost matrix between facilities
distance_matrix	Distance matrix between locations

The Quadratic Assignment Problem was introduced by Koopmans and Beckmann (1957) to model the optimal placement of economic activities (facilities) across geographic locations, minimizing total transportation cost weighted by inter-facility flows. It is NP-hard, as shown by Sahni and Gonzalez (1976) via reduction from the Hamiltonian Circuit problem. QAP is widely regarded as one of the hardest combinatorial optimization problems: even moderate instances ($n > 20$) challenge state-of-the-art exact solvers. Best exact approaches use branch-and-bound with Gilmore–Lawler bounds or cutting-plane methods; the best known general algorithm runs in $O^*(n!)$ by exhaustive enumeration of all permutations[16].

Applications include facility layout planning, keyboard and control panel design, scheduling, VLSI placement, and hospital floor planning. As a special case, when D is a distance matrix on a line (i.e., $D_{kl} = |k - l|$), QAP reduces to the Optimal Linear Arrangement problem.

Example. Consider $n = m = 4$ with flow matrix C and distance matrix D:

$$C = \begin{pmatrix} 0 & 5 & 2 & 0 \\ 5 & 0 & 0 & 3 \\ 2 & 0 & 0 & 4 \\ 0 & 3 & 4 & 0 \end{pmatrix}, \quad D = \begin{pmatrix} 0 & 4 & 1 & 1 \\ 4 & 0 & 3 & 4 \\ 1 & 3 & 0 & 4 \\ 1 & 4 & 4 & 0 \end{pmatrix}.$$

The identity assignment $f(i) = i$ gives cost 100. The optimal assignment is $f^* = (4, 1, 2, 3)$ with cost 56: it places the heavily interacting facilities F_1 and F_2 (highest flow $= 5$) at locations L_4 and L_1 (distance $= 1$).

```
$ pred create --example QuadraticAssignment -o quadratic-assignment.json
$ pred solve quadratic-assignment.json
$ pred evaluate quadratic-assignment.json --config 3,0,1,2
```

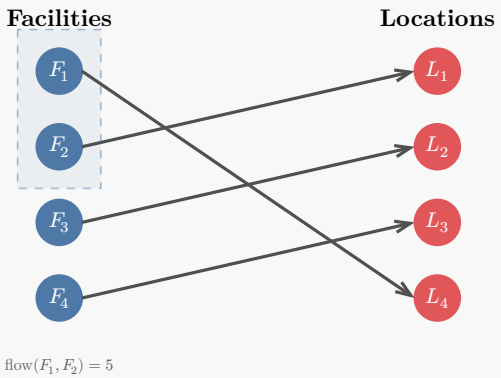


Figure 48: Optimal assignment $f^* = (4, 1, 2, 3)$ for the 4×4 QAP instance. Facilities (blue, left) are assigned to locations (red, right) by arrows. Facilities F_1 and F_2 (highest flow $= 5$) are assigned to locations L_4 and L_1 (distance $= 1$). Total cost $= 56$.

[16]No algorithm significantly improving on brute-force permutation enumeration is known for general QAP.

Definition 2.68 (Algebraic Equations over GF(2)): Given n Boolean variables $x_0, ..., x_{n-1}$ and m multilinear polynomials $p_1, ..., p_m$ over GF(2), determine whether there exists an assignment $(x_0, ..., x_{n-1}) \in \{0,1\}^n$ such that $p_j(x_0, ..., x_{n-1}) = 0$ for all $j = 1, ..., m$.

Each polynomial is a sum (XOR) of monomials over GF(2). Each monomial is a product (AND) of a subset of the variables; the empty product denotes the constant 1.

- Complexity: `2^(0.6943 * num_variables)`.
- Reduces from: ExactCoverBy3Sets.

AlgebraicEquationsOverGF2	
num_variables	Number of Boolean variables
equations	Equations: list of polynomials, each a list of monomials, each a sorted list of variable indices

Algebraic Equations over GF(2) generalises systems of linear equations over GF(2) by allowing higher-degree monomials. The problem is NP-complete in general [5]. When restricted to degree ≤ 1 (linear equations only), the system reduces to Gaussian elimination over GF(2) and is solvable in polynomial time.

Example. Let $n = 3$ with 3 equations. The assignment (1, 0, 0) satisfies all equations.

```
$ pred create --example AlgebraicEquationsOverGF2 -o agf2.json
$ pred solve agf2.json --solver brute-force
$ pred evaluate agf2.json --config 1,0,0
```

Definition 2.69 (Quadratic Congruences): Given non-negative integers a, b, c with $b > 0$ and $a < b$, determine whether there exists a positive integer x with $1 \leq x < c$ such that $x^2 \equiv a \pmod{b}$.

- Complexity: `2^bit_length_c`.
- Reduces from: KSatisfiability.

QuadraticCongruences	
a	a
b	b
c	c

Quadratic Congruences is an NP-complete problem in the setting where b is composite and given in unary (or the factorisation of b is not provided) [5]. The problem asks whether a is a *quadratic residue* modulo b in the range $\{1, ..., c-1\}$. When b is prime, quadratic residuosity can be decided in polynomial time via Euler's criterion or the Legendre symbol, but the general case with composite modulus is believed to be computationally hard without the factorisation of b.

Example. Let $a = 4$, $b = 15$, $c = 10$. We test each $x \in \{1, ..., 9\}$:

```
$ pred create --example QuadraticCongruences -o qc.json
$ pred solve qc.json --solver brute-force
$ pred evaluate qc.json --config 0,1,0,0
```

x	x^2	$x^2 \bmod 15$	Satisfies?
1	1	1	No
2	4	4	Yes
3	9	9	No
4	16	1	No
5	25	10	No

x	x^2	$x^2 \bmod 15$	Satisfies?
6	36	6	No
7	49	4	Yes
8	64	4	Yes
9	81	6	No

The instance is satisfiable: $x = 2$ gives $2^2 = 4 \equiv 4 \pmod{15}$.

Definition 2.70 (Quadratic Diophantine Equations): Given positive integers a, b, c, determine whether there exist positive integers x, y such that $ax^2 + by = c$.

- Complexity: `2^bit_length_c`.
- Reduces from: KSatisfiability.

QuadraticDiophantineEquations	
a	Coefficient of x^2
b	Coefficient of y
c	Right-hand side constant

Quadratic Diophantine equations of the form $ax^2 + by = c$ form one of the simplest families of mixed-degree Diophantine problems. The variable y is entirely determined by x via $y = (c - ax^2)/b$, so the decision problem reduces to checking whether any $x \in \left\{1, ..., \left\lfloor \sqrt{c/a} \right\rfloor\right\}$ yields a positive integer y. This can be done in $O(\sqrt{c})$ time by trial[17].

Example. Let $a = 3$, $b = 5$, $c = 53$. Then x ranges over $1, ..., 4$:

```
$ pred create --example QuadraticDiophantineEquations -o qde.json
$ pred solve qde.json --solver brute-force
$ pred evaluate qde.json --config 1,0,0
```

x	$c - ax^2$	Divisible by b?	y
1	50	Yes	10
2	41	No	–
3	26	No	–
4	5	Yes	1

The instance is satisfiable: $x = 2, y = 8$ gives $3 \cdot 2^2 + 5 \cdot 8 = 53$.

Definition 2.71 (Simultaneous Incongruences): Given a list of pairs (a_i, b_i) with $b_i > 0$ and $1 \leq a_i \leq b_i$ for $i = 1, ..., n$, determine whether there exists a non-negative integer x such that $x \not\equiv a_i \pmod{b_i}$ for all i.

- Complexity: `num_pairs`.
- Reduces from: KSatisfiability.

SimultaneousIncongruences	
pairs	Pairs (aᵢ, bᵢ) with bᵢ > 0 and 1 ≤ aᵢ ≤ bᵢ

Simultaneous Incongruences is an NP-complete problem [5]. It asks whether the complement of a system of congruences — a *covering system* — can be simultaneously avoided. A *covering system* is a finite collection

[17] No algorithm improving on brute-force trial of all candidate x values is known; the registered complexity `sqrt(c)` reflects this direct enumeration bound.

of congruences $\{a_i \pmod{b_i}\}$ that covers every integer; when the system is a covering system there is no valid x and the instance is a "no" instance. The problem generalises checking whether a given set of congruences is a covering system, which has connections to Erdős's covering conjecture and sieve methods in analytic number theory.

Example. Let $n = 4$ with pairs $(2, 2)$, $(1, 3)$, $(2, 5)$, $(3, 7)$. The full period is $L = \operatorname{lcm}(2,\ 3,\ 5,\ 7) = 210$. We test $x = 5$:

```
$ pred create --example SimultaneousIncongruences -o si.json
$ pred solve si.json --solver brute-force
$ pred evaluate si.json --config 5
```

a_i	b_i	$5 \bmod b_i$	$a_i \bmod b_i$	Avoids?
2	2	1	0	Yes
1	3	2	1	Yes
2	5	0	2	Yes
3	7	5	3	Yes

The instance is satisfiable: $x = 5$ avoids all congruences.

Definition 2.72 (Equilibrium Point): Given n players, finite strategy sets $M_i \subset \mathbb{Z}$, and polynomial payoff functions $F_i : M_1 \times \cdots \times M_n \to \mathbb{Z}$ expressed as products of affine factors, determine whether there exists a pure-strategy Nash equilibrium: an assignment $\boldsymbol{y} = (y_1, ..., y_n)$ with $y_i \in M_i$ such that for every player i and every $y_i' \in M_i$, $F_{i(\boldsymbol{y})} \geq F_{i\left(\boldsymbol{y}^{(-i, y_i')}\right)}$, where $\boldsymbol{y}^{(-i,y_i')}$ is $\boldsymbol{y}$ with the i-th component replaced by y_i'.

- Complexity: `2^num_players`.

```
EquilibriumPoint
polynomials      polynomials[i] is a list of affine factors for F_i; each factor [a0,a1,...,an] represents
                 a0 + a1*x1 + ... + an*xn
range_sets       range_sets[i] is the finite strategy set M_i for player i
```

Equilibrium Point (problem AN15 in Garey & Johnson [5]) is NP-complete by reduction from 3-SAT due to Sahni [61]. The problem captures a fundamental question in algorithmic game theory: does a multi-player game with polynomial payoffs admit a stable strategy profile from which no player benefits by deviating? The payoff functions are represented as products of affine factors, enabling compact encoding of degree-k polynomials with $O(kn)$ coefficients per player. The problem remains NP-complete even when all strategy sets are binary.

Example. Consider $n = 3$ players with strategy sets $M_i = \{0,\ 1\}$ for all i, and payoff functions:

$$F_1 = (\text{x1})\cdot(\text{x2})\cdot(\text{x3}),\ F_2 = (1\ +\ \text{-x1})\cdot(\text{x2}),\ F_3 = (\text{x1})\cdot(1\ +\ \text{-x3})$$

The assignment $\boldsymbol{y} = (0,\ 1,\ 0)$ is a Nash equilibrium: $F_{i(\boldsymbol{y})} = 0,\ 1,\ 0$ for $i = 1, 2, 3$, and no player can strictly improve their payoff by deviating.

```
$ pred create --example EquilibriumPoint -o ep.json
$ pred solve ep.json --solver brute-force
$ pred evaluate ep.json --config 0,1,0
```

Definition 2.73 (Closest Vector Problem): Given a lattice basis $\boldsymbol{B} \in \mathbb{R}^{m \times n}$ (columns $\boldsymbol{b}_1, ..., \boldsymbol{b}_n \in \mathbb{R}^m$ spanning lattice $\mathcal{L}(\boldsymbol{B}) = \{\boldsymbol{Bx} : \boldsymbol{x} \in \mathbb{Z}^n\}$) and target $\boldsymbol{t} \in \mathbb{R}^m$, find $\boldsymbol{x} \in \mathbb{Z}^n$ minimizing $\|\boldsymbol{Bx} - \boldsymbol{t}\|_2$.

- Complexity: `2^num_basis_vectors`.
- Reduces to: QUBO.
- Reduces from: SubsetSum.

ClosestVectorProblem	
basis	Basis matrix B as column vectors
target	Target vector t
bounds	Integer bounds per variable

The Closest Vector Problem is a fundamental lattice problem, proven NP-hard by van Emde Boas [62]. CVP appears in lattice-based cryptography, coding theory, and integer programming [59]. Kannan's enumeration algorithm [63] solves CVP in $n^{O(n)}$ time; Micciancio and Voulgaris [64] improved this to deterministic $O^*(4^n)$ using Voronoi cell computations, and Aggarwal, Dadush, and Stephens-Davidowitz [65] achieved randomized $O^*(2^n)$.

Example. Consider the 2D lattice with basis $\boldsymbol{b}_1 = (2, 0)^\top$, $\boldsymbol{b}_2 = (1, 2)^\top$ and target $\boldsymbol{t} = (2.8, 1.5)^\top$. The lattice points near $\boldsymbol{t}$ include $\boldsymbol{B}(1,0)^\top = (2,0)^\top$, $\boldsymbol{B}(0,1)^\top = (1,2)^\top$, and $\boldsymbol{B}(1,1)^\top = (3, 2)^\top$. The closest is $\boldsymbol{B}(1,1)^\top = (3, 2)^\top$ with distance $\left\|\boldsymbol{B}(1,1)^\top - \boldsymbol{t}\right\|_2 \approx 0.539$.

```
$ pred create --example ClosestVectorProblem -o closest-vector-problem.json
$ pred solve closest-vector-problem.json
$ pred evaluate closest-vector-problem.json --config 3,3
```

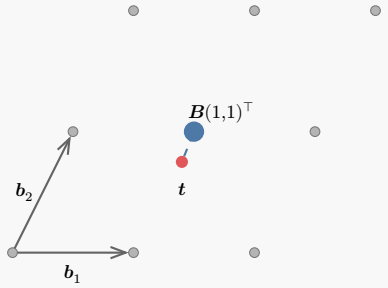


Figure 49: 2D lattice with basis $\boldsymbol{b}_1 = (2, 0)^\top$, $\boldsymbol{b}_2 = (1, 2)^\top$. Target $\boldsymbol{t} = (2.8, 1.5)^\top$ (red) and closest lattice point $\boldsymbol{B}(1,1)^\top = (3, 2)^\top$ (blue). Distance ≈ 0.539.

2.4 Satisfiability Problems

Definition 2.74 (SAT): Given a CNF formula $\varphi = \bigwedge_{j=1}^{m} C_j$ with m clauses over n Boolean variables, where each clause $C_j = \bigvee_i \ell_{ji}$ is a disjunction of literals, find an assignment $\boldsymbol{x} \in \{0,1\}^n$ such that $\varphi(\boldsymbol{x}) = 1$ (all clauses satisfied).

- Complexity: `2^num_variables`.
- Reduces to: CircuitSAT, IntegralFlowHomologousArcs, KColoring, KSatisfiability, Maximum2Satisfiability, MaximumIndependentSet, MinimumDominatingSet, NAESatisfiability, NonTautology.
- Reduces from: CircuitSAT, KSatisfiability.

Satisfiability	
num_vars	Number of Boolean variables
clauses	Clauses in conjunctive normal form

The Boolean Satisfiability Problem (SAT) is the first problem proven NP-complete [66]. SAT serves as the foundation of NP-completeness theory: showing a new problem NP-hard typically proceeds by reduction from SAT or one of its variants. Despite worst-case hardness, conflict-driven clause learning (CDCL) solvers handle industrial instances with millions of variables. The Strong Exponential Time Hypothesis (SETH) [67] conjectures that no $O^*((2-\varepsilon)^n)$ algorithm exists for general CNF-SAT, and the best known algorithm runs in $O^*(2^n)$ by brute-force enumeration[18].

Example. Consider $\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ with $n = 3$ variables and $m = 3$ clauses. The assignment $(x_1,x_2,x_3) = (0, 1, 0)$ satisfies all clauses: $C_1 = (0 \vee 1) = 1$, $C_2 = (1 \vee 0) = 1$, $C_3 = (0 \vee 1) = 1$. Hence $\varphi(0, 1, 0) = 1$.

```
$ pred create --example SAT -o sat.json
$ pred solve sat.json
$ pred evaluate sat.json --config 0,1,0
```

Definition 2.75 (NAE-SAT): Given a CNF formula $\varphi = \bigwedge_{j=1}^{m} C_j$ with m clauses over n Boolean variables, where each clause $C_j = \bigvee_i \ell_{ji}$ is a disjunction of literals, find an assignment $\boldsymbol{x} \in \{0,1\}^n$ such that every clause contains at least one true literal and at least one false literal.

- Complexity: `2^num_variables`.
- Reduces to: ILP, MaxCut, PartitionIntoPerfectMatchings, SetSplitting.
- Reduces from: Satisfiability.

NAESatisfiability	
num_vars	Number of Boolean variables
clauses	Clauses in conjunctive normal form with at least two literals each

Not-All-Equal Satisfiability (NAE-SAT) is a canonical variant in Schaefer's dichotomy theorem [68]. Unlike ordinary SAT, each clause forbids the all-true and all-false patterns, giving the problem a complement symmetry: if an assignment is NAE-satisfying, then flipping every bit is also NAE-satisfying. This makes NAE-SAT a natural intermediate for cut and partition reductions such as Max-Cut. A straightforward exact algorithm enumerates all 2^n assignments; complement symmetry can halve the search space in practice by fixing one variable, but the asymptotic worst-case bound remains $O^*(2^n)$.

Example. Consider $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \neg x_4 \vee x_5) \wedge (\neg x_2 \vee x_3 \vee \neg x_5) \wedge (x_1 \vee \neg x_3 \vee x_5)$ with $n = 5$ variables and $m = 5$ clauses. The assignment $(x_1,x_2,x_3,x_4,x_5) = (0, 0, 0, 1, 1)$ is NAE-satisfying because each clause evaluates to a tuple containing both 0 and 1: $C_1 = (0, 0, 1)$, $C_2 = (1, 0, 1)$, $C_3 = (0, 0, 1)$, $C_4 = (1, 0, 0)$, $C_5 = (0, 1, 1)$. The complementary assignment $(x_1,x_2,x_3,x_4,x_5) = (1, 1, 1, 0, 0)$ is therefore also NAE-satisfying, illustrating the paired-solution structure characteristic of NAE-SAT.

```
$ pred create --example NAESatisfiability -o nae-satisfiability.json
$ pred solve nae-satisfiability.json
$ pred evaluate nae-satisfiability.json --config 0,0,0,1,1
```

Definition 2.76 (k-SAT): SAT with exactly k literals per clause.

- Complexity: `num_variables + num_clauses`; `1.307^num_variables`; `2^num_variables`.
- Reduces to: KSatisfiability, QUBO, AcyclicPartition, CyclicOrdering, DecisionMinimumVertexCover, DirectedTwoCommodityIntegralFlow, FeasibleRegisterAssignment, KClique, Kernel, MinimumVertexCover, MonochromaticTriangle, OneInThreeSatisfiability, PreemptiveScheduling, QuadraticCongruences, QuadraticDiophantineEquations, RegisterSufficiency, SimultaneousIncongruences, SubsetSum, TimetableDesign, Satisfiability.
- Reduces from: KSatisfiability, Satisfiability.

KSatisfiability	
num_vars	Number of Boolean variables
clauses	Clauses each with exactly K literals

[18] SETH conjectures this is optimal; no $O^*((2-\varepsilon)^n)$ algorithm is known.

The restriction of SAT to exactly k literals per clause reveals a sharp complexity transition: 2-SAT is polynomial-time solvable via implication graph SCC decomposition [69] in $O(n+m)$, while k-SAT for $k \geq 3$ is NP-complete. Random k-SAT exhibits a satisfiability threshold at clause density $m/n \approx 2^k \ln 2$, a key phenomenon in computational phase transitions. The best known algorithm for 3-SAT runs in $O^*(1.307^n)$ via biased-PPSZ [70]. Under SETH, k-SAT requires time $O^*(c_k^n)$ with $c_k \to 2$ as $k \to \infty$.

Example. Consider the 3-SAT formula $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$ with $n = 3$ variables and $m = 3$ clauses, each containing exactly 3 literals. The assignment $(x_1,x_2,x_3) = (0, 0, 1)$ satisfies all clauses: $C_1 = (0 \vee 0 \vee 1) = 1$, $C_2 = (1 \vee 1 \vee 1) = 1$, $C_3 = (0 \vee 1 \vee 0) = 1$.

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred solve ksat.json
$ pred evaluate ksat.json --config 0,0,1
```

Definition 2.77 (Planar 3-SAT): Given a 3-CNF formula $\varphi = \bigwedge_{j=1}^{m} C_j$ with m clauses over n Boolean variables, where each clause C_j contains exactly 3 literals, and the variable-clause incidence graph $H(\varphi)$ is planar, find a satisfying assignment $\boldsymbol{x} \in \{0,1\}^n$.

- Complexity: `1.307^num_variables`.

Planar3Satisfiability	
num_vars	Number of Boolean variables
clauses	Clauses each with exactly 3 literals

Planar 3-SAT is a restricted variant of 3-SAT introduced by Lichtenstein [71], who proved it NP-complete. The incidence graph $H(\varphi)$ is bipartite with variable nodes and clause nodes, connected by edges when a variable appears in a clause. Requiring $H(\varphi)$ to be planar is a strong structural constraint that enables reductions to geometric and planar problems (e.g., rectilinear Steiner tree, planar vertex cover). The best known algorithm shares the 3-SAT bound of $O^*(1.307^n)$ via biased-PPSZ [70], since any Planar 3-SAT instance is also a valid 3-SAT instance.

Example. Consider $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$ with $n = 4$ variables and $m = 4$ clauses. The assignment $(x_1,x_2,x_3,x_4) = (1, 1, 1, 0)$ satisfies all clauses: $C_1 = (1 \vee 1 \vee 1) = 1$, $C_2 = (0 \vee 1 \vee 0) = 1$, $C_3 = (1 \vee 0 \vee 0) = 1$, $C_4 = (0 \vee 1 \vee 1) = 1$.

```
$ pred create --example Planar3Satisfiability -o planar3sat.json
$ pred solve planar3sat.json
$ pred evaluate planar3sat.json --config 1,1,1,0
```

Definition 2.78 (1-in-3 SAT): Given a CNF formula $\varphi = \bigwedge_{j=1}^{m} C_j$ with m clauses over n Boolean variables, where each clause C_j contains exactly 3 literals, find a truth assignment $\boldsymbol{x} \in \{0,1\}^n$ such that each clause has *exactly one* true literal.

- Complexity: `1.307^num_variables`.
- Reduces from: KSatisfiability.

OneInThreeSatisfiability	
num_vars	Number of Boolean variables
clauses	Clauses each with exactly 3 literals

One-in-Three Satisfiability (1-in-3 SAT) was introduced by Schaefer [68] as part of his dichotomy theorem for generalized satisfiability. Unlike standard 3-SAT which requires at least one true literal per clause, 1-in-3 SAT requires exactly one. The problem is NP-complete even for monotone instances (no negations).

The best known algorithm runs in $O^*(1.307^n)$ time via biased-PPSZ [70], since every 1-in-3 SAT instance reduces trivially to 3-SAT.

Example. Consider $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$ with $n = 4$ variables and $m = 3$ clauses. The assignment $(x_1,x_2,x_3,x_4) = (1, 0, 0, 1)$ satisfies the 1-in-3 condition: C_1 has 1 true literal, C_2 has 1 true literal, C_3 has 1 true literal.

```
$ pred create --example OneInThreeSatisfiability -o 1in3sat.json
$ pred solve 1in3sat.json
$ pred evaluate 1in3sat.json --config 1,0,0,1
```

Definition 2.79 (Maximum 2-Satisfiability): Given a set U of n Boolean variables and a collection $C = \{C_1, ..., C_m\}$ of m clauses over U with $|C_j| = 2$ for each j, find a truth assignment $\boldsymbol{x} \in \{0,1\}^n$ that maximizes the number of simultaneously satisfied clauses.

- Complexity: `2^(0.7905 * num_variables)`.
- Reduces to: ILP, MaxCut.
- Reduces from: Satisfiability.

Maximum2Satisfiability	
num_vars	Number of Boolean variables
clauses	Collection of 2-literal clauses

Maximum 2-Satisfiability (MAX-2-SAT) is one of the fundamental NP-hard optimization problems. While the decision version of 2-SAT is solvable in linear time by implication-graph analysis, the optimization variant—maximizing the number of satisfied clauses—is NP-hard [5]. The best known exact algorithm by Williams [8] runs in $O^*(2^{0.7905n})$ time by reducing to a maximum-weight triangle problem and applying fast matrix multiplication.

Example. Consider $m = 7$ clauses over $n = 4$ variables: $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee x_4) \wedge (\neg x_3 \vee \neg x_4) \wedge (x_3 \vee x_4)$. The assignment $(x_1,x_2,x_3,x_4) = (1, 1, 0, 1)$ satisfies 6 out of 7 clauses.

```
$ pred create --example Maximum2Satisfiability -o max2sat.json
$ pred solve max2sat.json
$ pred evaluate max2sat.json --config 1,1,0,1
```

Rule 2.1: (Maximum 2-Satisfiability → Max-Cut (weighted)) This $O(n + m)$ reduction [1], [5] builds a signed weighted graph with one reference vertex s and one vertex per Boolean variable. Each 2-clause contributes two reference-variable terms and, when the clause uses two different variables, one variable-variable term. After doubling the affine clause identity to clear fractions, the target has $n + 1$ vertices and at most $n + m$ nonzero edges.
Overhead: `num_vertices = num_vars + 1`, `num_edges = num_vars + num_clauses`.

Proof: *Construction.* Let φ be a MAX-2-SAT instance on variables $x_1, ..., x_n$. Create one reference vertex $s = v_0$ and one vertex v_i for each variable x_i. For a literal ℓ over variable x_i, define $\sigma(\ell) = 1$ when $\ell = x_i$ and $\sigma(\ell) = -1$ when $\ell = \neg x_i$. For each clause $C = (\ell_a \vee \ell_b)$, add $-\sigma(\ell_a)$ to edge (s, v_a) and $-\sigma(\ell_b)$ to edge (s, v_b). If $a \neq b$, also add $\sigma(\ell_a)\sigma(\ell_b)$ to edge (v_a, v_b). Repeated contributions accumulate; zero-weight edges are omitted. Interpret a cut by setting $x_i = 1$ exactly when v_i lies on the same side of the cut as s.

Correctness. Let $\delta(u, v) \in \{0, 1\}$ indicate whether vertices u and v lie on opposite sides of the cut. For each variable define $y_i = 1 - 2\delta(s, v_i)$, so $y_i = 1$ iff $x_i = 1$ and $y_i = -1$ iff $x_i = 0$. For clause $C = (\ell_a \vee \ell_b)$ with $\sigma_a = \sigma(\ell_a)$ and $\sigma_b = \sigma(\ell_b)$, its satisfaction indicator is

$$S_C = \frac{3 + \sigma_a y_a + \sigma_b y_b - \sigma_a \sigma_b y_a y_b}{4}.$$

Since $y_i = 1 - 2\delta(s, v_i)$ and $y_a y_b = 1 - 2\delta(v_a, v_b)$, multiplying by 2 yields

$$2S_C = K_C - \sigma_a \delta(s, v_a) - \sigma_b \delta(s, v_b) + \sigma_a \sigma_b \delta(v_a, v_b),$$

where $K_C = \frac{3+\sigma_a+\sigma_b-\sigma_a\sigma_b}{2}$ is independent of the chosen cut. Summing over all clauses gives

$$2S(\varphi, \boldsymbol{x}) = C_0 + w(\delta)$$

for the constant $C_0 = \sum_C K_C$.

($\Rightarrow$) Any truth assignment $\boldsymbol{x}$ induces a cut by placing v_i with s iff $x_i = 1$. The displayed identity shows that an assignment satisfying k clauses yields cut value $2k - C_0$.

($\Leftarrow$) Any cut δ extracts a truth assignment by comparing each v_i with s. If another assignment satisfied more clauses, its induced cut would have strictly larger cut value by the same identity, contradicting maximality. Therefore every maximum cut extracts to an optimal MAX-2-SAT assignment.

Solution extraction. Return the source bit $x_i = 1$ iff v_i and s lie on the same side of the cut. Because this depends only on equality with s, globally swapping the two cut sides leaves the extracted assignment unchanged. □

Example: $n = 3$ variables, $m = 5$ clauses, target has 4 vertices and 4 edges

Source: Maximum2Satisfiability **Target:** MaxCut

```
$ pred create --example Maximum2Satisfiability -o max2sat.json
$ pred reduce max2sat.json --to MaxCut/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate max2sat.json --config 0,1,1
```

Step 1 – Source instance. The canonical source has $n = 3$ variables and 5 two-literal clauses. The stored optimal assignment is (0, 1, 1), which satisfies all five clauses.

Step 2 – Accumulate the cut weights. Introduce the reference vertex $s = v_0$ and variable vertices v_1, v_2, v_3. After summing the per-clause contributions and deleting zero-weight edges, the target graph has the four signed edges (s, v_2) with weight -1, (s, v_3) with weight -1, (v_1, v_2) with weight 2, and (v_2, v_3) with weight -1.

Step 3 – Verify the witness. The target witness (0, 1, 0, 0) puts v_2 and v_3 on the same side as s and v_1 on the opposite side, so extraction recovers (0, 1, 1). Only edge (v_1, v_2) crosses, so the cut value is 2 and the affine objective identity certifies optimality ✓.

Multiplicity: The fixture stores one canonical witness. Flipping every target bit yields the complementary cut partition but extracts the same source assignment because extraction compares each variable vertex to s.

Rule 2.2: (Maximum 2-Satisfiability → Integer Linear Programming) A MAX-2-SAT instance maps directly to a binary ILP [5]: each Boolean variable becomes a binary decision variable, each clause gets a binary indicator variable, and a single linear inequality per clause links the indicator to its literals. The objective maximizes the sum of clause indicators, so the ILP optimum equals the maximum number of satisfiable clauses.
Overhead: `num_vars = num_vars + num_clauses`, `num_constraints = num_clauses`.

Proof: *Construction.* Given n Boolean variables and m clauses, introduce binary variables $y_0, ..., y_{n-1} \in \{0, 1\}$ (truth assignment) and $z_0, ..., z_{m-1} \in \{0, 1\}$ (clause indicators). For each clause C_j with literals ℓ_1, ℓ_2, define $\ell_{i'} = y_i$ if positive and $\ell_{i'} = 1 - y_i$ if negated. Add the constraint $z_j \leq \ell_{1'} + \ell_{2'}$, ensuring $z_j = 1$ only when the clause is satisfied. The ILP is:

$$\begin{aligned} \max \quad & \sum_{j=0}^{m-1} z_j \\ \text{subject to} \quad & z_j \leq \ell_{1'} + \ell_{2'} \quad \forall j \in \{0, ..., m-1\} \\ & y_i \in \{0,1\} \quad \forall i \in \{0, ..., n-1\} \\ & z_j \in \{0,1\} \quad \forall j \in \{0, ..., m-1\} \end{aligned}$$

. The target has $n + m$ variables and m constraints.

Correctness. ($\Rightarrow$) Any truth assignment $\boldsymbol{y}$ satisfying k clauses yields a feasible ILP solution by setting $z_j = 1$ iff clause j is satisfied, achieving objective k. ($\Leftarrow$) Any feasible ILP solution with $z_j = 1$ has clause j satisfied by the constraint, so the truth assignment satisfies at least $\sum z_j$ clauses. Thus optimal values coincide.

Solution extraction. Return the first n components $(y_0, ..., y_{n-1})$ as the truth assignment. □

Example: $n = 4$ **variables,** $m = 7$ **clauses**

Source: Maximum2Satisfiability **Target:** ILP

```
$ pred create --example Maximum2Satisfiability -o max2sat.json
$ pred reduce max2sat.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate max2sat.json --config 1,1,0,1
```

Step 1 – Source instance. The canonical MAX-2-SAT instance has $n = 4$ Boolean variables and $m = 7$ clauses.

Step 2 – Build the binary ILP. Introduce n binary truth variables $y_0, ..., y_{n-1} \in \{0, 1\}$ and m binary clause-indicator variables $z_0, ..., z_{m-1} \in \{0, 1\}$. The objective is

$$\max \sum_{j=0}^{m-1} z_j$$

subject to one constraint per clause j: $z_j \leq l_{1'} + l_{2'}$ where $l_{i'} = y_i$ for a positive literal and $l_{i'} = 1 - y_i$ for a negated literal. The resulting ILP has $n + m = 11$ variables and $m = 7$ constraints.

Step 3 – Verify a solution. The ILP optimum extracts the first n variables as the truth assignment $\boldsymbol{y}^* = (1, 1, 0, 1)$, satisfying 4 source variables ✓.

Definition 2.80 (Non-Tautology): Given a Boolean formula in DNF $\varphi = \bigvee_{j=1}^{m} D_j$ with m disjuncts over n Boolean variables, where each disjunct D_j is a conjunction of literals, find a truth assignment $\boldsymbol{x} \in \{0,1\}^n$ such that $\varphi(\boldsymbol{x}) = 0$ (i.e., every disjunct is false).

- Complexity: `1.307^num_variables`.
- Reduces from: Satisfiability.

NonTautology	
num_vars	Number of Boolean variables
disjuncts	Disjuncts (each a conjunction of literals) in disjunctive normal form

The Non-Tautology problem asks whether a given DNF formula is *not* a tautology, by finding a falsifying assignment. A disjunct $D_j = \ell_1 \wedge ... \wedge \ell_k$ is false when at least one of its literals evaluates to false; the formula is false when all disjuncts are false. The problem is coNP-complete in general and closely related to SAT through De Morgan duality: a DNF formula φ is a tautology iff $\neg\varphi$ (a CNF formula) is unsatisfiable.

Example. Consider $\varphi = (x_1 \wedge x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3)$ with $n = 3$ variables and $m = 2$ disjuncts. The assignment $(x_1,x_2,x_3) = (1, 0, 0)$ falsifies the formula: D_1 is false, D_2 is false.

```
$ pred create --example NonTautology -o nontaut.json
$ pred solve nontaut.json
$ pred evaluate nontaut.json --config 1,0,0
```

Definition 2.81 (CircuitSAT): Given a Boolean circuit C composed of logic gates (AND, OR, NOT, XOR) with n input variables, find an input assignment $\boldsymbol{x} \in \{0,1\}^n$ such that $C(\boldsymbol{x}) = 1$.

- Complexity: `2^num_variables`.
- Reduces to: ILP, Satisfiability, SpinGlass.
- Reduces from: Factoring, Satisfiability.

```
CircuitSAT
circuit    The boolean circuit
```

Circuit Satisfiability is the most natural NP-complete problem: the Cook-Levin theorem [66] proves NP-completeness by showing any nondeterministic polynomial-time computation can be encoded as a Boolean circuit. CircuitSAT is strictly more succinct than CNF-SAT, since a circuit with g gates may require an exponentially larger CNF formula without auxiliary variables. The Tseitin transformation reduces CircuitSAT to CNF-SAT with only $O(g)$ clauses by introducing one auxiliary variable per gate. The best known algorithm runs in $O^*(2^n)$ by brute-force enumeration[19].

Example. Consider the circuit $C(x_1, x_2) = (x_1 \text{ AND } x_2) \text{ XOR } (x_1 \text{ OR } x_2)$ with $n = 2$ inputs and $g = 3$ gates. Evaluating: $C(0,0) = (0) \text{ XOR } (0) = 0$, $C(0,1) = (0) \text{ XOR } (1) = 1$, $C(1,0) = (0) \text{ XOR } (1) = 1$, $C(1,1) = (1) \text{ XOR } (1) = 0$. The satisfying assignments are (0, 0) – precisely the inputs where exactly one variable is true.

```
$ pred create --example CircuitSAT -o circuitsat.json
$ pred solve circuitsat.json
$ pred evaluate circuitsat.json --config 0,0,0,0,0
```

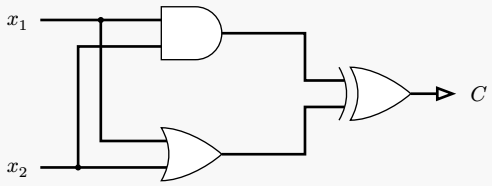


Figure 50: Circuit $C(x_1, x_2) = (x_1 \wedge x_2) \oplus (x_1 \vee x_2)$. Junction dots mark where inputs fork to both gates. Satisfying assignments: (0, 0).

Definition 2.82 (Conjunctive Query Foldability): Given a finite domain D, relation symbols $R_1, ..., R_m$ with fixed arities $d_1, ..., d_m$, a set X of *distinguished* variables, a set Y of *undistinguished* variables (with $X \cap Y = \emptyset$), and two conjunctive queries Q_1 and Q_2 — each a set of atoms of the form $R_j\big(t_1, ..., t_{d_j}\big)$ with $t_i \in D \cup X \cup Y$ — determine whether there exists a substitution $\sigma : Y \to D \cup X \cup Y$ such that $\sigma(Q_1) = Q_2$ as sets of atoms, where σ fixes all elements of $D \cup X$.

- Complexity: `(num_distinguished + num_undistinguished + domain_size)^num_undistinguished * num_conjuncts_q1`.

```
ConjunctiveQueryFoldability
domain_size                Size of the finite domain D
num_distinguished          Number of distinguished variables X
num_undistinguished        Number of undistinguished variables Y
```

[19] No algorithm improving on brute-force is known for general circuits.

ConjunctiveQueryFoldability	
relation_arities	Arity of each relation
query1_conjuncts	Atoms of query Q1
query2_conjuncts	Atoms of query Q2

Conjunctive query foldability is equivalent to conjunctive query containment and was shown NP-complete by Chandra and Merlin (1977) via reduction from Graph 3-Colorability.[20] If Q_1 folds into Q_2, then Q_1 is subsumed by Q_2, making Q_1 redundant — a key step in query optimization. The brute-force algorithm enumerates all $|D \cup X \cup Y|^{|Y|}$ possible substitutions and checks set equality; no general exact algorithm with a better worst-case bound is known.[21]

Example. Let $D = \emptyset$, $X = \{x\}$, $Y = \{u, v, a\}$, and R a single binary relation. The query $Q_1 = \{R(x,u), R(u,v), R(v,x), R(u,u)\}$ is a directed triangle (x, u, v) with a self-loop on u. The query $Q_2 = \{R(x,a), R(a,a), R(a,x)\}$ is a "lollipop": a self-loop on a with edges $x \to a$ and $a \to x$. The substitution $\sigma : u \mapsto a$,
$v \mapsto a$,
$a \mapsto a$ maps Q_1 to $\{R(x,a), R(a,a), R(a,x), R(a,a)\} = Q_2$ (as a set), so Q_1 folds into Q_2.

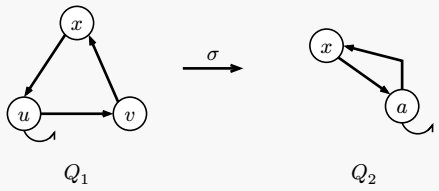


Figure 51: Conjunctive Query Foldability example. Left: query Q_1 — directed triangle (x, u, v) with self-loop on u. Right: query Q_2 — lollipop with node a having a self-loop and two edges to x. The substitution $\sigma : u \mapsto a, v \mapsto a$ (with $a \mapsto a$) folds Q_1 into Q_2.

Definition 2.83 (Ensemble Computation): Given a finite set A and a collection C of subsets of A, find the minimum number of union operations in a sequence $S = (z_1 \leftarrow x_1 \cup y_1, z_2 \leftarrow x_2 \cup y_2, ..., z_j \leftarrow x_j \cup y_j)$ such that each operand x_i, y_i is either a singleton $\{a\}$ for some $a \in A$ or a previously computed set z_k with $k < i$, the two operands are disjoint for every step, and every target subset $c \in C$ is equal to some computed set z_i.

- Complexity: `(universe_size + budget)^(2 * budget)`.
- Reduces from: MinimumVertexCover.

EnsembleComputation	
universe_size	Number of elements in the universe A
subsets	Required subsets that must appear among the computed z_i values
budget	Maximum number of union operations J

Ensemble Computation is problem PO9 in Garey and Johnson [5]. It can be viewed as monotone circuit synthesis over set union: each operation introduces one reusable intermediate set, and the objective is to realize all targets in the fewest operations. The original GJ formulation is a decision problem with a budget parameter J; this library models the optimization variant that minimizes the sequence length, using J as a search-space bound. The implementation uses $2J$ operand variables with domain size $|A| + J$ and reports the first step at which all targets are produced. The resulting search space yields a straightforward exact upper bound of $(|A| + J)^{2J}$. Järvisalo, Kaski, Koivisto, and Korhonen study SAT encodings for finding efficient ensemble computations in a monotone-circuit setting [72].

[20] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational data bases," *Proc. 9th ACM STOC*, 1977, pp. 77–90.

[21] No algorithm improving on brute-force substitution enumeration is known for general conjunctive query foldability.

Example. Let $A = \{0, 1, 2, 3\}$, $C = \{\{0, 1, 2\}, \{0, 1, 3\}\}$, and $J = 4$. A satisfying witness uses three essential unions: $z_1 = \{0\} \cup \{1\} = \{0, 1\}$, $z_2 = z_1 \cup \{2\} = \{0, 1, 2\}$, and $z_3 = z_1 \cup \{3\} = \{0, 1, 3\}$. Thus both target subsets appear among the computed z_i values while staying within the budget.

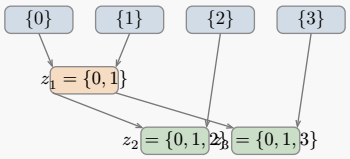


Figure 52: An ensemble computation for $A = \{0, 1, 2, 3\}$ and $C = \{\{0, 1, 2\}, \{0, 1, 3\}\}$. The intermediate set $z_1 = \{0, 1\}$ is reused to produce both target subsets.

Definition 2.84 (Factoring): Given a composite integer N and bit sizes m, n, find integers $p \in [2, 2^m - 1]$ and $q \in [2, 2^n - 1]$ such that $p \times q = N$. Here p has m bits and q has n bits.

- Complexity: `exp((m + n)^(1/3) * log(m + n)^(2/3))`.
- Reduces to: CircuitSAT, ILP.

Factoring	
m	Bits for first factor
n	Bits for second factor
target	Number to factor

The hardness of integer factorization underpins RSA cryptography and other public-key systems. Unlike most problems in this collection, Factoring is not known to be NP-complete; it lies in NP ∩ co-NP, suggesting it may be of intermediate complexity. The best classical algorithm is the General Number Field Sieve [73] running in sub-exponential time $e^{O\left(b^{1/3}(\log b)^{2/3}\right)}$ where b is the bit length. Shor's algorithm [74] solves Factoring in polynomial time on a quantum computer.

Example. Let $N = 15$ with $m = 2$ bits and $n = 3$ bits, so $p \in [2, 3]$ and $q \in [2, 7]$. The solution is $p = 5$, $q = 7$, since $5 \times 7 = 15 = N$. Note $p = 5$ fits in 2 bits and $q = 7$ fits in 3 bits. The alternative factorization 7×5 requires $m = 3$, $n = 2$.

```
$ pred create --example Factoring -o factoring.json
$ pred solve factoring.json
$ pred evaluate factoring.json --config 1,1,1,0,1
```

Definition 2.85 (Quantified Boolean Formulas (QBF)): Given a set $U = \{u_1, ..., u_n\}$ of Boolean variables and a fully quantified Boolean formula $F = (Q_1 u_1)(Q_2 u_2) \cdots (Q_n u_n) E$, where each $Q_i \in \{\exists, \forall\}$ is a quantifier and E is a Boolean expression in CNF with m clauses, determine whether F is true.

- Complexity: `2^num_vars`.

QuantifiedBooleanFormulas	
num_vars	Number of Boolean variables
quantifiers	Quantifier for each variable (Exists or ForAll)
clauses	CNF clauses of the Boolean expression E

Quantified Boolean Formulas (QBF) is the canonical PSPACE-complete problem, established by L. J. Stockmeyer and A. R. Meyer [75]. QBF generalizes SAT by adding universal quantifiers (∀) alongside existential ones (∃), creating a two-player game semantics: the existential player chooses values for ∃-variables, the universal player for ∀-variables, and the formula is true iff the existential player has a winning strategy ensuring all clauses are satisfied. This quantifier alternation is the source of PSPACE-hardness and makes QBF the primary source of PSPACE-completeness reductions for combinatorial game problems.

The problem remains PSPACE-complete even when E is restricted to 3-CNF (Quantified 3-SAT), but is polynomial-time solvable when each clause has at most 2 literals [68]. The best known exact algorithm is brute-force game-tree evaluation in $O^*(2^n)$ time. For QBF with m CNF clauses, R. Williams [76] achieves $O^*(1.709^m)$ time.

Example. Consider $F = \exists u_1\ \forall u_2\ (u_1 \vee u_2) \wedge (u_1 \vee \neg u_2)$ with $n = 2$ variables and $m = 2$ clauses. The existential player chooses $u_1 = 1$: then $C_1 = (1 \vee u_2) = 1$ and $C_2 = (1 \vee \neg u_2) = 1$ for any value of u_2. Hence F is `true` — the existential player has a winning strategy.

```
$ pred create --example QuantifiedBooleanFormulas -o quantified-boolean-formulas.json
$ pred solve quantified-boolean-formulas.json
$ pred evaluate quantified-boolean-formulas.json --config
```

2.5 Specialized Problems

Definition 2.86 (Boolean Matrix Factorization): Given an $m \times n$ boolean matrix A and rank k, find boolean matrices $B \in \{0,1\}^{m \times k}$ and $C \in \{0,1\}^{k \times n}$ minimizing the Hamming distance $d_H(A, B \circ C)$, where the boolean product $(B \circ C)_{ij} = \bigvee_\ell (B_{i\ell} \wedge C_{\ell j})$.

- Complexity: `2^(rows * rank + rank * cols)`.
- Reduces to: ILP.

```
BMF
matrix Target boolean matrix A
k      Factorization rank
```

Boolean Matrix Factorization decomposes binary data into interpretable boolean factors, unlike real-valued SVD which loses the discrete structure. NP-hard even to approximate, BMF arises in data mining, text classification, and role-based access control where factors correspond to latent binary features. Practical algorithms use greedy rank-1 extraction or alternating fixed-point methods. The best known exact algorithm runs in $O^*(2^{mk+kn})$ by brute-force search over B and C[22].

Example. Let $A = (1, 1, 0; 1, 1, 1; 0, 1, 1)$ and $k = 2$. Set $B = (0, 1; 1, 1; 1, 0)$ and $C = (0, 1, 1; 1, 1, 0)$. Then $B \circ C = (1, 1, 0; 1, 1, 1; 0, 1, 1) = A$, achieving Hamming distance $d_H = 0$ (exact factorization). The two boolean factors capture overlapping row/column patterns: factor 1 selects rows $\{1, 2\}$ and columns $\{1, 2\}$; factor 2 selects rows $\{2, 3\}$ and columns $\{2, 3\}$.

```
$ pred create --example BMF -o bmf.json
$ pred solve bmf.json
$ pred evaluate bmf.json --config 0,1,1,1,1,0,0,1,1,1,1,0
```

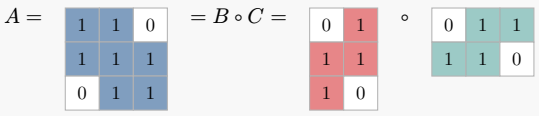


Figure 53: Boolean matrix factorization: $A = B \circ C$ with rank $k = 2$. Factor 1 (red) covers the top-left block; factor 2 (teal) covers the bottom-right block.

Definition 2.87 (Consecutive Block Minimization): Given an $m \times n$ binary matrix A and a positive integer K, determine whether there exists a permutation of the columns of A such that the resulting matrix has at most K maximal blocks of consecutive 1-entries (summed over all rows). A *block* is a maximal contiguous run of 1-entries within a single row.

[22] No algorithm improving on brute-force enumeration is known for general BMF.

- Complexity: `factorial(num_cols) * num_rows * num_cols`.
- Reduces to: ILP.

ConsecutiveBlockMinimization	
matrix	Binary matrix A (m x n)
bound	Upper bound K on total consecutive blocks

Consecutive Block Minimization (SR17 in Garey & Johnson) arises in consecutive file organization for information retrieval systems, where records stored on a linear medium must be arranged so that each query's relevant records form a contiguous segment. Applications also include scheduling, production planning, the glass cutting industry, and data compression. NP-complete by reduction from Hamiltonian Path [50]. When K equals the number of non-all-zero rows, the problem reduces to testing the *consecutive ones property*, solvable in polynomial time via PQ-trees [77]. A 1.5-approximation is known [78]. The best known exact algorithm runs in $O^*(n!)$ by brute-force enumeration of all column permutations.

Example. Let A be the 6×6 matrix with rows $r_0 = (0,1,0,0,0,0)$, $r_1 = (1,0,1,0,0,0)$, $r_2 = (0,1,0,1,0,0)$, $r_3 = (0,0,1,0,1,0)$, $r_4 = (0,0,0,1,0,1)$, $r_5 = (0,0,0,0,1,0)$ and $K = 6$. The column permutation $\pi = (0, 2, 4, 1, 3, 5)$ yields 6 total blocks, so $6 \leq 6$ and the answer is YES.

```
$ pred create --example ConsecutiveBlockMinimization -o consecutive-block-minimization.json
$ pred solve consecutive-block-minimization.json
$ pred evaluate consecutive-block-minimization.json --config 0,2,4,1,3,5
```

Definition 2.88 (Paint Shop): Given a sequence of $2n$ positions where each of n cars appears exactly twice, assign a binary color to each car (each car's two occurrences receive opposite colors) to minimize the number of color changes between consecutive positions.

- Complexity: `2^num_cars`.
- Reduces to: ILP, QUBO.

PaintShop	
sequence	Car labels (each must appear exactly twice)

NP-hard and APX-hard [79]. Arises in automotive manufacturing where color changes between consecutive cars on an assembly line require costly purging of paint nozzles. Each car appears twice in the sequence (two coats), and each car's two occurrences must receive opposite colors (one per side). A natural benchmark for quantum annealing due to its binary structure and industrial relevance. The best known algorithm runs in $O^*(2^n)$ by brute-force enumeration[23].

Example. Consider $n = 3$ cars with sequence (A, B, A, C, B, C). Each car gets one occurrence colored 0 and the other colored 1. The assignment A: 0/1, B: 0/1, C: 1/0 yields color sequence (0, 0, 1, 1, 1, 0) with 2 color changes. The minimum is 2 changes.

```
$ pred create --example PaintShop -o paint-shop.json
$ pred solve paint-shop.json
$ pred evaluate paint-shop.json --config 0,0,1
```

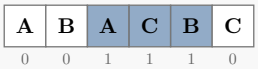


Figure 54: Paint Shop: sequence (A, B, A, C, B, C) with optimal coloring. White = color 0, blue = color 1. 2 color changes (marked ×).

[23] No algorithm improving on brute-force is known for general Paint Shop.

Definition 2.89 (Biclique Cover): Given a bipartite graph $G = (L, R, E)$ and integer k, find k bicliques $(L_1, R_1), ..., (L_k, R_k)$ that cover all edges ($E \subseteq \bigcup_i L_i \times R_i$) while minimizing the total size $\sum_i (|L_i| + |R_i|)$.

- Complexity: `2^num_vertices`.
- Reduces to: ILP.

BicliqueCover	
left_size	Vertices in left partition
right_size	Vertices in right partition
edges	Bipartite edges
k	Number of bicliques

Biclique Cover is equivalent to factoring the biadjacency matrix M of the bipartite graph as a Boolean sum of rank-1 binary matrices, connecting it to Boolean matrix rank and nondeterministic communication complexity. Applications include data compression, database optimization (covering queries with materialized views), and bioinformatics (gene expression biclustering). NP-hard even for fixed $k \geq 2$. The best known algorithm runs in $O^*(2^{|L|+|R|})$ by brute-force enumeration[24].

Example. Consider $G = (L, R, E)$ with $L = \{\ell_1, \ell_2\}$, $R = \{r_1, r_2, r_3\}$, and edges $E = \{(\ell_1, r_1), (\ell_1, r_2), (\ell_2, r_2), (\ell_2, r_3)\}$. A biclique cover with $k = 2$: $(L_1, R_1) = (\{\ell_1\}, \{r_1, r_2\})$ covering edges $\{(\ell_1, r_1), (\ell_1, r_2)\}$, and $(L_2, R_2) = (\{\ell_2\}, \{r_2, r_3\})$ covering $\{(\ell_2, r_2), (\ell_2, r_3)\}$. Total size $= (1 + 2) + (1 + 2) = 5$. Merging into a single biclique is impossible since $(\ell_1, r_3) \notin E$.

```
$ pred create --example BicliqueCover -o biclique-cover.json
$ pred solve biclique-cover.json
$ pred evaluate biclique-cover.json --config 0,1,0,1,0,1,0,1,0,1
```

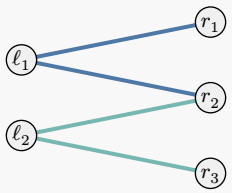


Figure 55: Biclique cover of a bipartite graph: biclique 1 (blue) = $(\{\ell_1\}, \{r_1, r_2\})$, biclique 2 (teal) = $(\{\ell_2\}, \{r_2, r_3\})$. Edge (ℓ_1, r_3) is absent, preventing a single biclique.

Definition 2.90 (Balanced Complete Bipartite Subgraph): Given a bipartite graph $G = (A, B, E)$ and an integer k, determine whether there exist subsets $A' \subseteq A$ and $B' \subseteq B$ such that $|A'| = |B'| = k$ and every cross pair is present: $A' \times B' \subseteq E$.

- Complexity: `1.3803^num_vertices`.
- Reduces to: ILP.
- Reduces from: KClique.

BalancedCompleteBipartiteSubgraph	
graph	The bipartite graph G = (A, B, E)
k	Balanced biclique size

Balanced Complete Bipartite Subgraph is a classical NP-complete bipartite containment problem from Garey and Johnson [5]. Unlike Biclique Cover, which asks for a collection of bicliques covering all edges, this problem asks for a *single* balanced biclique of prescribed size. It arises naturally in biclustering, dense submatrix discovery, and pattern mining on bipartite data. Chen et al. give an exact $O^*(1.3803^n)$ algorithm for dense bipartite graphs, and the registry records that best-known bound in the catalog metadata. A

[24]No algorithm improving on brute-force enumeration is known for general Biclique Cover.

straightforward baseline still enumerates all k-subsets of A and B and checks whether they induce a complete bipartite graph, taking $O\left(\binom{|A|}{k} \cdot \binom{|B|}{k} \cdot k^2\right) = O^*\left(2^{|A|+|B|}\right)$ time.

Example. Consider the bipartite graph with $A = \{\ell_1, \ell_2, \ell_3, \ell_4\}$, $B = \{r_1, r_2, r_3, r_4\}$, and edges $E = \{(\ell_1, r_1), (\ell_1, r_2), (\ell_1, r_3), (\ell_2, r_1), (\ell_2, r_2), (\ell_2, r_3), (\ell_3, r_1), (\ell_3, r_2), (\ell_3, r_3), (\ell_4, r_1), (\ell_4, r_2), (\ell_4, r_4)\}$. For $k = 3$, the selected sets $A' = \{\ell_1, \ell_2, \ell_3\}$ and $B' = \{r_1, r_2, r_3\}$ form a balanced complete bipartite subgraph: all 9 required cross edges are present. Vertex ℓ_4 is excluded because $(\ell_4, r_3) \notin E$, so any witness using ℓ_4 cannot realize $K_{3,3}$.

```
$ pred create --example BalancedCompleteBipartiteSubgraph -o balanced-complete-bipartite-subgraph.json
$ pred solve balanced-complete-bipartite-subgraph.json
$ pred evaluate balanced-complete-bipartite-subgraph.json --config 1,1,1,0,1,1,1,0
```

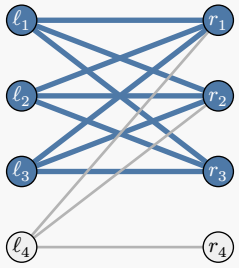


Figure 56: Balanced complete bipartite subgraph with $k = 3$: the selected vertices $A' = \{\ell_1, \ell_2, \ell_3\}$ and $B' = \{r_1, r_2, r_3\}$ are blue, and the 9 edges of the induced $K_{3,3}$ are highlighted. The missing edge (ℓ_4, r_3) prevents including ℓ_4.

Definition 2.91 (Partition Into Triangles): Given a graph $G = (V, E)$ with $|V| = 3q$ for some integer q, determine whether the vertices of G can be partitioned into q disjoint triples $V_1, ..., V_q$, each containing exactly 3 vertices, such that for each $V_i = \{u_i, v_i, w_i\}$, all three edges $\{u_i, v_i\}$, $\{u_i, w_i\}$, and $\{v_i, w_i\}$ belong to E.

- Complexity: `2^num_vertices`.
- Reduces to: ILP.

PartitionIntoTriangles	
graph	The underlying graph G=(V,E) with |V| divisible by 3

Partition Into Triangles is NP-complete by transformation from 3-Dimensional Matching [5, GT11]. It remains NP-complete on graphs of maximum degree 4, with an exact algorithm running in $O^*(1.0222^n)$ for bounded-degree-4 graphs [80]. The general brute-force bound is $O^*(2^n)$[25].

Example. Consider G with $n = 6$ vertices ($q = 2$) and edges $\{0, 1\}$, $\{0, 2\}$, $\{1, 2\}$, $\{3, 4\}$, $\{3, 5\}$, $\{4, 5\}$, $\{0, 3\}$. The partition $V_1 = \{v_0, v_1, v_2\}$, $V_2 = \{v_3, v_4, v_5\}$ is valid: V_1 forms a triangle and V_2 forms a triangle. The cross-edge $\{0, 3\}$ is unused. Swapping v_2 and v_3 yields $V_1' = \{v_0, v_1, v_3\}$, which fails because $\{1, 3\} \notin E$.

```
$ pred create --example PartitionIntoTriangles -o partition-into-triangles.json
$ pred solve partition-into-triangles.json
$ pred evaluate partition-into-triangles.json --config 0,0,0,1,1,1
```

[25] No algorithm improving on brute-force enumeration is known for general Partition Into Triangles.

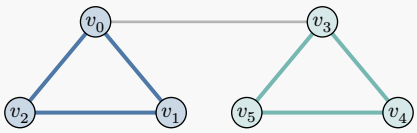


Figure 57: Partition Into Triangles: $V_1 = \{v_0, v_1, v_2\}$ and $V_2 = \{v_3, v_4, v_5\}$ each form a triangle. Cross-edges (gray) are unused.

Definition 2.92 (Partition into Forests): Given an undirected graph $G = (V, E)$ and a positive integer K, determine whether the vertex set V can be partitioned into K non-empty classes $V_1, ..., V_K$ such that the subgraph $G[V_i]$ induced by each class V_i is a forest (acyclic graph).

- Complexity: `num_forests^num_vertices`.

PartitionIntoForests	
graph	The underlying graph G=(V,E)
num_forests	num_forests: number of forest classes K (>= 1)

Partition Into Forests is NP-complete [5, GT18]. The problem asks whether the vertex set can be split into K classes each inducing an acyclic subgraph; it generalises arboricity decomposition (covering all edges with K forests, solvable in polynomial time) to the decision problem where the partition need not cover all edges. The best known exact algorithm uses brute-force enumeration in $O^*(K^n)$ time.

Example. Consider G with $n = 6$ vertices and edges $\{0,1\}$, $\{1,2\}$, $\{2,0\}$, $\{2,3\}$, $\{3,4\}$, $\{4,5\}$, $\{5,3\}$. With $K = 2$, the partition $V_1 = \{v_0, v_3\}$, $V_2 = \{v_1, v_2, v_4, v_5\}$ is valid: each induced subgraph is acyclic.

```
$ pred create --example PartitionIntoForests -o partition-into-forests.json
$ pred solve partition-into-forests.json
$ pred evaluate partition-into-forests.json --config 0,1,1,0,1,1
```

Definition 2.93 (Degree-Constrained Spanning Tree): Given an undirected graph $G = (V, E)$ and a positive integer K, determine whether G contains a spanning tree T in which every vertex has degree at most K.

- Complexity: `2^num_vertices`.
- Reduces from: HamiltonianPath.

DegreeConstrainedSpanningTree	
graph	The underlying graph G=(V,E)
max_degree	max_degree: maximum allowed vertex degree K (>= 1)

Degree-Constrained Spanning Tree is NP-complete [5, ND1]. The problem generalises the Hamiltonian Path problem (set $K = 2$). The best known exact algorithm uses brute-force enumeration in $O^*(2^n)$ time, where $n = |V|$.

Example. Consider G with $n = 5$ vertices and $m = 7$ edges $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,2\}$, $\{1,4\}$, $\{2,3\}$, $\{3,4\}$. With $K = 2$, the spanning tree using edges $\{0,2\}$, $\{0,3\}$, $\{1,2\}$, $\{1,4\}$ has maximum vertex degree ≤ 2.

```
$ pred create --example DegreeConstrainedSpanningTree -o dcst.json
$ pred solve dcst.json
$ pred evaluate dcst.json --config 0,1,1,1,1,0,0
```

Definition 2.94 (Bounded Diameter Spanning Tree): Given an undirected graph $G = (V, E)$ with positive edge weights $w : E \to \mathbb{Z}_{>0}$, a weight bound B, and a diameter bound D, determine whether G contains a spanning tree T such that the total weight $\sum_{e \in T} w(e) \leq B$ and the diameter of T (the longest shortest path in number of edges) is at most D.

- Complexity: `num_vertices ^ num_vertices`.

BoundedDiameterSpanningTree	
graph	The underlying graph G=(V,E)
edge_weights	Edge weights w: E -> ZZ_(> 0)
weight_bound	Upper bound B on total tree weight
diameter_bound	Upper bound D on tree diameter (in edges)

Bounded Diameter Spanning Tree is NP-hard [5, ND3]. The problem asks for a spanning tree that simultaneously satisfies a budget constraint on total edge weight and a structural constraint on tree diameter. The best known exact algorithm uses brute-force enumeration in $O^*(n^n)$ time, where $n = |V|$.

Example. Consider G with $n = 5$ vertices and $m = 7$ edges $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,2\}$, $\{1,4\}$, $\{2,3\}$, $\{3,4\}$ with edge weights $w(\{0,1\}) = 1$, $w(\{0,2\}) = 2$, $w(\{0,3\}) = 1$, $w(\{1,2\}) = 1$, $w(\{1,4\}) = 2$, $w(\{2,3\}) = 1$, $w(\{3,4\}) = 1$. With $B = 5$ and $D = 3$, the spanning tree using edges $\{0,1\}$, $\{0,3\}$, $\{2,3\}$, $\{3,4\}$ has total weight $4 \leq 5$ and diameter ≤ 3.

```
$ pred create --example BoundedDiameterSpanningTree -o bdst.json
$ pred solve bdst.json
$ pred evaluate bdst.json --config 1,0,1,0,0,1,1
```

Definition 2.95 (Maximum Leaf Spanning Tree): Given a connected undirected graph $G = (V, E)$, find a spanning tree T of G that maximizes the number of leaves (degree-1 vertices) in T.

- Complexity: `1.8966^num_vertices`.
- Reduces to: ILP.

MaximumLeafSpanningTree	
graph	The underlying graph G=(V,E)

Maximum Leaf Spanning Tree is NP-hard [5, ND2]. The problem has applications in network design and broadcasting, where maximizing the number of leaf nodes reduces the set of internal (relay) nodes. The best known exact algorithm runs in $O^*(1.8966^n)$ time, where $n = |V|$.[26]

Example. Consider G with $n = 6$ vertices and $m = 9$ edges $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,4\}$, $\{2,4\}$, $\{2,5\}$, $\{3,5\}$, $\{4,5\}$, $\{1,3\}$. The spanning tree using edges $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{2,4\}$, $\{2,5\}$ has 4 leaves (vertices 1, 3, 4, 5), each with degree 1 in the tree.

```
$ pred create --example MaximumLeafSpanningTree -o mlst.json
$ pred solve mlst.json
$ pred evaluate mlst.json --config 1,1,1,0,1,1,0,0,0
```

Rule 2.3: (Maximum Leaf Spanning Tree → Integer Linear Programming) An MLST instance reduces to an integer linear program with $3m + n$ variables and $3n + 2m + 1$ constraints, using a single-commodity flow formulation to enforce spanning-tree connectivity and binary leaf indicators to encode the objective [5]. *Overhead:* `num_vars = 3 * num_edges + num_vertices`, `num_constraints = 3 * num_vertices + 2 * num_edges + 1`.

[26] H. Fernau, J. Kneis, D. Kratsch, A. Langer, M. Liedloff, D. Raible, and P. Rossmanith, An exact algorithm for the Maximum Leaf Spanning Tree problem, *Theoretical Computer Science*, 412(45):6290–6302, 2011.

Proof: *Construction.* Given $G = (V, E)$ with $n = |V|$, $m = |E|$, root the flow at vertex 0. Introduce binary edge selectors $y_e \in \{0, 1\}$ for each $e \in E$ (m variables), binary leaf indicators $z_v \in \{0, 1\}$ for each $v \in V$ (n variables), and directed flow variables $f_{uv}, f_{vu} \geq 0$ for each undirected edge $\{u, v\}$ ($2m$ variables).

Constraints:

1. **Tree cardinality:** $\sum_e y_e = n - 1$.
2. **Flow conservation:** for the root, net outflow $= n - 1$; for each non-root vertex v, net inflow $= 1$.
3. **Flow–edge linking:** $f_{uv} + f_{vu} \leq (n - 1)y_e$ for each edge $e = \{u, v\}$.
4. **Leaf detection:** $\sum_{e \ni v} y_e + (n - 2)z_v \leq n - 1$ for each vertex v.
5. **Binary bounds:** $y_e \leq 1$, $z_v \leq 1$.

The objective is $\max \sum_{v \in V} z_v$.

Correctness. ($\Rightarrow$) A spanning tree T of G with ℓ leaves induces a feasible ILP solution: route one unit of flow from the root to every other vertex along the unique tree path, set $y_e = 1$ for tree edges, and set $z_v = 1$ for degree-1 vertices; constraint (4) is tight when $\deg_{T(v)} = 1$ and slack otherwise, achieving objective ℓ. ($\Leftarrow$) Any feasible ILP solution with $\sum y_e = n - 1$ and connectivity enforced by the flow yields a spanning tree; constraint (4) forces $z_v = 0$ whenever $\deg_{T(v)} > 1$, and maximization ensures $z_v = 1$ for all leaves.

Solution extraction. Return the edge-selector prefix $(y_0, ..., y_{m-1})$ as the source configuration. □

Definition 2.96 (Monochromatic Triangle): Given an undirected graph $G = (V, E)$, determine whether the edges of G can be 2-colored (each edge assigned color 0 or 1) so that no triangle is monochromatic — that is, for every three mutually adjacent vertices u, v, w, the three edges $\{u, v\}$, $\{u, w\}$, $\{v, w\}$ do not all receive the same color.

- Complexity: `2^num_edges`.
- Reduces to: ILP.
- Reduces from: KSatisfiability.

MonochromaticTriangle	
graph	The underlying graph G=(V,E)

Monochromatic Triangle is closely related to Ramsey theory. By the classical result $R(3, 3) = 6$, the complete graph K_6 admits no valid 2-coloring, while K_5 does. The problem is NP-complete in general [5, GT6]. The best known exact algorithm uses brute-force enumeration in $O^*(2^m)$ time, where $m = |E|$.

Example. Consider K_4 with $n = 4$ vertices and $m = 6$ edges. A valid coloring assigns colors edge $0 \to 0$, edge $1 \to 0$, edge $2 \to 1$, edge $3 \to 1$, edge $4 \to 0$, edge $5 \to 1$, so that no triangle has all three edges the same color.

```
$ pred create --example MonochromaticTriangle -o monochromatic-triangle.json
$ pred solve monochromatic-triangle.json
$ pred evaluate monochromatic-triangle.json --config 0,0,1,1,0,1
```

Definition 2.97 (Partition into Cliques): Given an undirected graph $G = (V, E)$ and a positive integer $K \leq |V|$, determine whether the vertex set V can be partitioned into $k \leq K$ groups $V_1, ..., V_k$ such that each group V_i induces a complete subgraph (clique) in G.

- Complexity: `2^num_vertices`.
- Reduces to: MinimumCoveringByCliques.
- Reduces from: KColoring.

PartitionIntoCliques	
graph	The underlying graph G=(V,E)
num_cliques	num_cliques: maximum number of clique groups K (>= 1)

Partition Into Cliques is NP-complete [5, GT15]. The problem is the complement of Graph Coloring: a valid clique cover of G corresponds to a valid coloring of the complement graph $\overline{G}$. The best known exact algorithm uses brute-force enumeration in $O^*(2^n)$ time.

Example. Consider G with $n = 6$ vertices and edges $\{0,1\}$, $\{0,2\}$, $\{1,2\}$, $\{3,4\}$, $\{3,5\}$, $\{4,5\}$, $\{0,3\}$, $\{1,4\}$, $\{2,5\}$. With $K = 3$, the partition $V_1 = \{v_0, v_1, v_2\}$, $V_2 = \{v_3, v_4, v_5\}$ is valid: each group induces a clique.

```
$ pred create --example PartitionIntoCliques -o partition-into-cliques.json
$ pred solve partition-into-cliques.json
$ pred evaluate partition-into-cliques.json --config 0,0,0,1,1,1
```

Definition 2.98 (Partition into Perfect Matchings): Given an undirected graph $G = (V, E)$ and a positive integer $K \leq |V|$, determine whether the vertex set V can be partitioned into $k \leq K$ groups $V_1, ..., V_k$ such that the subgraph induced by each group V_i is a perfect matching: every vertex in V_i has exactly one neighbor within V_i. Empty groups are permitted.

- Complexity: `num_matchings^num_vertices`.
- Reduces from: NAESatisfiability.

PartitionIntoPerfectMatchings	
graph	The underlying graph G=(V,E)
num_matchings	num_matchings: maximum number of matching groups K (>= 1)

Partition Into Perfect Matchings is NP-complete [5, GT16]. The problem asks whether the edges of a graph can be decomposed into perfect matchings of vertex-induced subgraphs. The best known exact algorithm uses brute-force enumeration in $O^*(K^n)$ time.

Example. Consider G with $n = 4$ vertices and edges $\{0,1\}$, $\{2,3\}$, $\{0,2\}$, $\{1,3\}$. With $K = 2$, the partition $V_1 = \{v_0, v_1\}$, $V_2 = \{v_2, v_3\}$ is valid: each group induces a perfect matching.

```
$ pred create --example PartitionIntoPerfectMatchings -o partition-into-perfect-matchings.json
$ pred solve partition-into-perfect-matchings.json
$ pred evaluate partition-into-perfect-matchings.json --config 0,0,1,1
```

Definition 2.99 (Bin Packing): Given n items with sizes $s_1, ..., s_n \in \mathbb{R}^+$ and bin capacity $C > 0$, find an assignment $x : \{1, ..., n\} \to \mathbb{N}$ minimizing $|\{x(i) : i = 1, ..., n\}|$ (the number of distinct bins used) subject to $\forall j : \sum_{i:x(i)=j} s_i \leq C$.

- Complexity: `2^num_items`.
- Reduces to: ILP.
- Reduces from: Partition.

BinPacking	
sizes	Item sizes s_i for each item
capacity	Bin capacity C

Bin Packing is one of the classical NP-hard optimization problems [5], with applications in logistics, cutting stock, and cloud resource allocation. The best known exact algorithm runs in $O^*(2^n)$ time via inclusion-exclusion over set partitions [14].

Example. Consider $n = 3$ items with sizes (3, 3, 4) and capacity $C = 7$. An optimal packing uses 2 bins.

```
$ pred create --example BinPacking -o bin-packing.json
$ pred solve bin-packing.json
$ pred evaluate bin-packing.json --config 0,1,0
```

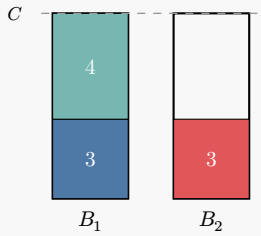


Figure 58: Optimal packing of 3 items into 2 bins of capacity $C = 7$. Numbers indicate item sizes.

Definition 2.100 (Knapsack): Given n items with weights $w_0, ..., w_{n-1} \in \mathbb{N}$ and values $v_0, ..., v_{n-1} \in \mathbb{N}$, and a capacity $C \in \mathbb{N}$, find $S \subseteq \{0, ..., n-1\}$ maximizing $\sum_{i \in S} v_i$ subject to $\sum_{i \in S} w_i \leq C$.

- Complexity: `2^(num_items / 2)`.
- Reduces to: ILP, QUBO.
- Reduces from: Partition.

```
Knapsack
weights  Nonnegative item weights w_i
values   Nonnegative item values v_i
capacity Nonnegative knapsack capacity C
```

One of Karp's 21 NP-complete problems [1]. Knapsack is only *weakly* NP-hard: a classical dynamic-programming algorithm runs in $O(nC)$ pseudo-polynomial time, and a fully polynomial-time approximation scheme (FPTAS) achieves $(1-\varepsilon)$-optimal value in $O(n^2/\varepsilon)$ time [81]. The special case $v_i = w_i$ for all i is the Subset Sum problem. Knapsack is also a special case of Integer Linear Programming with a single constraint. The best known exact algorithm is the $O^*(2^{n/2})$ meet-in-the-middle approach of Horowitz and Sahni [82], which partitions items into two halves and combines sorted sublists.

Example. Let $n = 4$ items with weights (2, 3, 4, 5), values (3, 4, 5, 7), and capacity $C = 7$. Selecting $S = \{0, 3\}$ gives total weight $7 \leq C$ and total value 10, which is optimal.

```
$ pred create --example Knapsack -o knapsack.json
$ pred solve knapsack.json
$ pred evaluate knapsack.json --config 1,0,0,1
```

Definition 2.101 (Integer Knapsack): Given n items with sizes $s_0, ..., s_{n-1} \in \mathbb{Z}^+$ and values $v_0, ..., v_{n-1} \in \mathbb{Z}^+$, and a capacity $B \in \mathbb{N}$, find non-negative integer multiplicities $c_0, ..., c_{n-1} \in \mathbb{N}$ maximizing $\sum_{i=0}^{n-1} c_i \cdot v_i$ subject to $\sum_{i=0}^{n-1} c_i \cdot s_i \leq B$.

- Complexity: `(capacity + 1)^num_items`.
- Reduces to: ILP.
- Reduces from: SubsetSum.

```
IntegerKnapsack
sizes          Positive item sizes s(u)
values         Positive item values v(u)
capacity       Nonnegative knapsack capacity B
```

The Integer Knapsack (also called the *unbounded knapsack problem*) generalizes the 0-1 Knapsack by allowing each item to be selected more than once. Like 0-1 Knapsack, it admits a pseudo-polynomial $O(nB)$

dynamic-programming algorithm [5]. The problem is weakly NP-hard: when item sizes are bounded by a polynomial in n, DP runs in polynomial time. The brute-force approach enumerates all multiplicity vectors, giving $O\left(\prod_{i=0}^{n-1}(\lfloor B/s_i \rfloor + 1)\right)$ configurations.[27]

Example. Let $n = 5$ items with sizes (3, 4, 5, 2, 7), values (4, 5, 7, 3, 9), and capacity $B = 15$. Setting multiplicities (0, 0, 1, 5, 0) gives total size $15 \leq B$ and total value 22, which is optimal.

```
$ pred create --example IntegerKnapsack -o ik.json
$ pred solve ik.json
$ pred evaluate ik.json --config 0,0,1,5,0
```

Definition 2.102 (Partially Ordered Knapsack): Given n items with weights $w_0, ..., w_{n-1} \in \mathbb{N}$ and values $v_0, ..., v_{n-1} \in \mathbb{N}$, a partial order $\prec$ on the items (given by its cover relations), and a capacity $C \in \mathbb{N}$, find a downward-closed subset $S \subseteq \{0, ..., n-1\}$ (i.e., if $i \in S$ and $j \prec i$ then $j \in S$) maximizing $\sum_{i \in S} v_i$ subject to $\sum_{i \in S} w_i \leq C$.

- Complexity: `2^num_items`.
- Reduces to: ILP.

PartiallyOrderedKnapsack	
weights	Item weights w(u) for each item
values	Item values v(u) for each item
precedences	Precedence pairs (a, b) meaning a must be included before b
capacity	Knapsack capacity B

Garey and Johnson's problem A6 MP12 [5]. Unlike standard Knapsack, the partial order constraint makes the problem *strongly* NP-complete — it remains NP-complete even when $w_i = v_i$ for all i, so no pseudo-polynomial algorithm exists unless $P = NP$. The problem arises in manufacturing scheduling, project selection, and mining operations. For tree partial orders, Johnson and Niemi [83] gave pseudo-polynomial $O(n \cdot C)$ tree DP and an FPTAS. Kolliopoulos and Steiner [84] extended the FPTAS to 2-dimensional partial orders with $O(n^4/\varepsilon)$ running time.

Example. Let $n = 6$ items with weights $(2, 3, 4, 1, 2, 3)$, values $(3, 2, 5, 4, 3, 8)$, and capacity $C = 11$. The partial order has cover relations $0 \prec 2$, $0 \prec 3$, $1 \prec 4$, $3 \prec 5$, $4 \prec 5$. Selecting $S = \{0, 1, 3, 4, 5\}$ is downward-closed (all predecessors included), has total weight $2 + 3 + 1 + 2 + 3 = 11 \leq C$, and total value $3 + 2 + 4 + 3 + 8 = 20$. Adding item 2 would exceed capacity ($15 > 11$).

Definition 2.103 (Rectilinear Picture Compression): Given an $m \times n$ binary matrix M and a nonnegative integer K, determine whether there exists a collection of at most K axis-aligned rectangles that covers precisely the 1-entries of M. Each rectangle is a quadruple (a, b, c, d) with $a \leq b$ and $c \leq d$, covering entries M_{ij} for $a \leq i \leq b$ and $c \leq j \leq d$, where every covered entry must satisfy $M_{ij} = 1$.

- Complexity: `2^(num_rows * num_cols)`.
- Reduces to: ILP.

RectilinearPictureCompression	
matrix	m x n binary matrix
bound	Maximum number of rectangles allowed

Rectilinear Picture Compression is a classical NP-complete problem from Garey & Johnson (A4 SR25, p. 232) [5]. It arises naturally in image compression, DNA microarray design, integrated circuit manufacturing, and access control list minimization. NP-completeness was established by Masek (1978) via transformation from 3SAT. A straightforward exact baseline, including the brute-force solver in this crate,

[27] No algorithm improving on brute-force enumeration of multiplicity vectors is known for the general Integer Knapsack problem.

enumerates subsets of the maximal all-1 rectangles. If an instance has R such rectangles, this gives an $O^*(2^R)$ exact search, so the worst-case behavior remains exponential in the instance size.

```
M = (
  (true, true, false, false),
  (true, true, false, false),
  (false, false, true, true),
  (false, false, true, true),
```

Example. Let $)(1, 1, 0, 0; 1, 1, 0, 0; 0, 0, 1, 1; 0, 0, 1, 1)$ (a 4×4 matrix) and $K = 2$. The two maximal all-1 rectangles cover rows 0..1, columns 0..1 and rows 2..3, columns 2..3. Selecting both gives $|\{R_1, R_2\}| = 2 \leq K = 2$ and their union covers all eight 1-entries, so the answer is YES.

```
$ pred create --example RectilinearPictureCompression -o rectilinear-picture-compression.json
$ pred solve rectilinear-picture-compression.json
$ pred evaluate rectilinear-picture-compression.json --config 1,1
```

1	**1**	0	0
1	**1**	0	0
0	0	**1**	**1**
0	0	**1**	**1**

Figure 59: Rectilinear Picture Compression: matrix M covered by two rectangles R_1 (blue, top-left 2×2) and R_2 (teal, bottom-right 2×2), with $|\{R_1, R_2\}| = 2 \leq K = 2$.

Definition 2.104 (Feasible Register Assignment): Given a directed acyclic graph $G = (V, A)$ with $n = |V|$ vertices, where each arc $(v, u) \in A$ means vertex v depends on vertex u, K registers, and a register assignment $f : V \to \{0, ..., K - 1\}$, determine whether there exists a topological evaluation ordering such that no register conflict arises: when a vertex v is evaluated and placed in register $f(v)$, no other vertex currently occupying register $f(v)$ still has uncomputed dependents.

- Complexity: `factorial(num_vertices)`.
- Reduces to: ILP.
- Reduces from: KSatisfiability.

FeasibleRegisterAssignment	
num_vertices	Number of vertices n = |V|
arcs	Directed arcs (v, u) meaning v depends on u
num_registers	Number of registers K
assignment	Register assignment f(v) for each vertex

Feasible Register Assignment is NP-complete [85]. It is closely related to Register Sufficiency (SS19 in Garey & Johnson [5]), but here the register assignment is given and only the scheduling order is sought.

Example. Let $n = 4$ vertices with arcs: $v_0 \to v_1$, $v_0 \to v_2$, $v_1 \to v_3$. Registers $K = 2$, assignment $f = (0, 1, 0, 0)$. The evaluation order (v_3, v_1, v_2, v_0) is feasible.

```
$ pred create --example FeasibleRegisterAssignment -o feasible-register-assignment.json
$ pred solve feasible-register-assignment.json --solver brute-force
$ pred evaluate feasible-register-assignment.json --config 3,1,2,0
```

Definition 2.105 (Minimum Register Sufficiency for Loops): Given a loop of length N (representing N timesteps arranged in a circle) and a set of n variables, each active during a contiguous circular arc of timesteps specified by (s_i, l_i) covering timesteps $\{s_i, s_i + 1, ..., s_i + l_i - 1\} \bmod N$, assign a register $r_i \in$

$\{0, ..., n-1\}$ to each variable minimizing the number of distinct registers used, such that no two variables with overlapping arcs share the same register.

- Complexity: `num_variables ^ num_variables`.

MinimumRegisterSufficiencyForLoops	
loop_length	Loop length N (number of timesteps)
variables	Variables as (start_time, duration) circular arcs

Minimum Register Sufficiency for Loops is problem SS20 in Garey & Johnson [5]. It is equivalent to minimum coloring of circular arc graphs. NP-complete via reduction from Chromatic Number. No algorithm improving on brute-force $O(n^n)$ enumeration is known for arbitrary circular arc instances.

Example. Let $N = 6$ timesteps and $n = 3$ variables with arcs: $x_0 : [0, 0+3)$, $x_1 : [2, 2+3)$, $x_2 : [4, 4+3)$ mod 6. All pairs of arcs overlap (each arc covers half the circle and any two arcs share at least one timestep), forming a complete conflict graph K_3. The assignment (0, 1, 2) uses 3 distinct registers, which is optimal.

```
$ pred create --example MinimumRegisterSufficiencyForLoops -o mrsfl.json
$ pred solve mrsfl.json --solver brute-force
$ pred evaluate mrsfl.json --config 0,1,2
```

Definition 2.106 (Register Sufficiency): Given a directed acyclic graph $G = (V, A)$ with $n = |V|$ vertices, where each arc $(v, u) \in A$ means vertex v depends on vertex u, and a positive integer K, determine whether there exists an evaluation ordering $v_{\pi(0)}, v_{\pi(1)}, ..., v_{\pi(n-1)}$ of all vertices such that the computation can be performed using at most K registers. A value must reside in a register from the moment it is computed until all vertices that depend on it have been evaluated.

- Complexity: `num_vertices ^ 2 * 2 ^ num_vertices`.
- Reduces to: ILP.
- Reduces from: KSatisfiability.

RegisterSufficiency	
num_vertices	Number of vertices n = |V|
arcs	Directed arcs (v, u) meaning v depends on u
bound	Register bound K

Register Sufficiency is problem SS19 (also A11 PO1) in Garey & Johnson [5]. NP-complete via reduction from 3-SAT [86]. Remains NP-complete even when all vertices have out-degree at most 2. For expression trees (DAGs with tree structure), the Sethi–Ullman algorithm solves the problem in $O(n)$ time [87]. For general DAGs, Kessler's dynamic programming over register states runs in $O(n^2 \cdot 2^n)$ time [88].

Example. Let $n = 7$ vertices with arcs (dependencies): $v_2 \to v_0$, $v_2 \to v_1$, $v_3 \to v_1$, $v_4 \to v_2$, $v_4 \to v_3$, $v_5 \to v_0$, $v_6 \to v_4$, $v_6 \to v_5$. Bound $K = 3$. The evaluation order $\pi = (v_0, v_1, v_2, v_3, v_5, v_4, v_6)$ achieves a maximum of 3 registers.

```
$ pred create --example RegisterSufficiency -o register-sufficiency.json
$ pred solve register-sufficiency.json --solver brute-force
$ pred evaluate register-sufficiency.json --config 0,1,2,3,5,4,6
```

Definition 2.107 (Minimum Code Generation (One Register)): Given a directed acyclic graph $G = (V, A)$ with maximum out-degree 2 representing an expression DAG, where leaf vertices (out-degree 0) are input values stored in memory, internal vertices are operations, and root vertices (in-degree 0) are the values

to compute, find a program of minimum number of instructions for a one-register machine (supporting LOAD, STORE, and OP instructions) that computes all root vertices.

- Complexity: `2 ^ num_vertices`.

MinimumCodeGenerationOneRegister	
num_vertices	Number of vertices n = |V|
edges	Directed arcs (parent, child) in the expression DAG
num_leaves	Number of leaf vertices (out-degree 0)

Minimum Code Generation on a One-Register Machine is problem A11 PO4 in Garey & Johnson [5]. NP-complete via transformation from 3-Satisfiability [89]. Remains NP-complete even when the only vertices with in-degree greater than 1 have arcs only to leaves. For directed forests (expression trees), the Sethi–Ullman algorithm finds an optimal instruction sequence in $O(n)$ time [87]. For general DAGs, brute-force enumeration of all valid evaluation orderings runs in $O^*(2^n)$ time.[28]

Example. Consider $n = 7$ vertices with arcs: $v_0 \to v_1$, $v_0 \to v_2$, $v_1 \to v_3$, $v_1 \to v_4$, $v_2 \to v_3$, $v_2 \to v_5$, $v_3 \to v_5$, $v_3 \to v_6$. Leaves (out-degree 0): $\{v_4, v_5, v_6\}$. The evaluation order (v_3, v_2, v_1, v_0) yields an optimal program of 8 instructions.

```
$ pred create --example MinimumCodeGenerationOneRegister -o mcgor.json
$ pred solve mcgor.json --solver brute-force
$ pred evaluate mcgor.json --config 3,2,1,0
```

Definition 2.108 (Minimum Code Generation (Unlimited Registers)): Given a directed acyclic graph $G = (V, A)$ with maximum out-degree 2 representing an expression DAG, and a partition of arcs into left (L) and right (R) operand sets, find a program of minimum number of instructions for an unlimited-register machine using 2-address instructions (OP and LOAD/copy) that computes all root vertices. The OP instruction computes a vertex and overwrites the left operand's register; a LOAD instruction copies a register to preserve a value before destruction.

- Complexity: `2 ^ num_vertices`.
- Reduces from: MinimumFeedbackVertexSet.

MinimumCodeGenerationUnlimitedRegisters	
num_vertices	Number of vertices n = |V|
left_arcs	Left operand arcs L: (parent, child) — child's register is destroyed
right_arcs	Right operand arcs R: (parent, child) — child's register is preserved

Minimum Code Generation with Unlimited Registers is problem A11 PO5 in Garey & Johnson [5]. NP-complete via transformation from Feedback Vertex Set [90]. Remains NP-complete even if the only vertices with in-degree greater than 1 are leaves. Polynomial for forests and when 3-address instructions are allowed. For general DAGs, brute-force enumeration runs in $O^*(2^n)$ time.[29]

Example. Consider $n = 5$ vertices with left arcs L: $v_1 \to v_3$, $v_2 \to v_3$, $v_0 \to v_1$ and right arcs R: $v_1 \to v_4$, $v_2 \to v_4$, $v_0 \to v_2$. Leaves (out-degree 0): $\{v_3, v_4\}$. The evaluation order (v_1, v_2, v_0) yields an optimal program of 4 instructions.

```
$ pred create --example MinimumCodeGenerationUnlimitedRegisters -o mcgur.json
$ pred solve mcgur.json --solver brute-force
$ pred evaluate mcgur.json --config 2,0,1
```

[28] No algorithm improving on brute-force is known for general expression DAGs.

[29] No algorithm improving on brute-force is known for general expression DAGs with 2-address instructions.

Definition 2.109 (Minimum Code Generation (Parallel Assignments)): Given a set V of variables and a collection of simultaneous assignments $A_i : v_i \leftarrow B_i$ where $v_i \in V$ is the target variable and $B_i \subseteq V$ is the set of variables read, find a permutation π of the assignments that minimizes the number of backward dependencies. A backward dependency occurs when $v_{\pi(i)} \in B_{\pi(j)}$ for some $j > i$, i.e., assignment $\pi(i)$ overwrites a variable that a later assignment $\pi(j)$ still needs to read.

- Complexity: `2^num_assignments`.

MinimumCodeGenerationParallelAssignments	
num_variables	Number of variables
assignments	Each assignment (target_var, read_vars)

Minimum Code Generation for Parallel Assignments is problem A11 PO6 in Garey & Johnson [5]. NP-complete via transformation from Feedback Vertex Set [86]. Remains NP-complete even when $|B_i| \leq 2$ for all assignments. For general instances, brute-force enumeration of all permutations runs in $O^*(2^m)$ time via dynamic programming over subsets.[30]

Example. Consider 4 variables and 4 assignments: $A_0 : v_0 \leftarrow \{v_1, v_2\}$, $A_1 : v_1 \leftarrow \{v_0\}$, $A_2 : v_2 \leftarrow \{v_3\}$, $A_3 : v_3 \leftarrow \{v_1, v_2\}$. The execution order (A_0, A_2, A_3, A_1) yields 2 backward dependencies.

```
$ pred create --example MinimumCodeGenerationParallelAssignments -o mcgpa.json
$ pred solve mcgpa.json --solver brute-force
$ pred evaluate mcgpa.json --config 0,3,1,2
```

Definition 2.110 (Minimum Decision Tree): Given a set S of n objects and m binary tests $T_1, ..., T_m$ where each test maps S to $\{0, 1\}$ and every pair of objects is distinguished by at least one test, find a decision tree using tests from T that identifies each object and minimizes the total external path length (sum of leaf depths).

- Complexity: `num_tests^num_objects`.

MinimumDecisionTree	
test_matrix	Binary matrix: test_matrix[j][i] = object i passes test j
num_objects	Number of objects to identify
num_tests	Number of available binary tests

Minimum Decision Tree (MS15) models optimal test sequencing for object identification [5]. NP-hard even when each test has at most 3 positive objects (Hyafil and Rivest 1976, via reduction from Exact Cover by 3-Sets). The Sethi–Ullman algorithm solves the tree (forest) case in polynomial time. The brute-force bound is $O^*(m^n)$.

Example. Consider $n = 4$ objects and $m = 3$ tests. The optimal decision tree has total external path length 8, achieved by a balanced split at the root.

```
$ pred create --example MinimumDecisionTree -o mdt.json
$ pred solve mdt.json
$ pred evaluate mdt.json --config 0,2,1,3,3,3,3
```

Definition 2.111 (Minimum Disjunctive Normal Form): Given n Boolean variables and a Boolean function $f : \{0, 1\}^n \to \{0, 1\}$ specified by its truth table, find a disjunctive normal form (DNF) formula with the minimum number of terms (disjuncts) that is equivalent to f.

- Complexity: `2^(3^num_variables)`.

[30] No algorithm improving on brute-force is known for general parallel assignment instances.

MinimumDisjunctiveNormalForm	
num_variables	Number of Boolean variables
truth_table	Truth table of length 2^n

Minimum Disjunctive Normal Form (LO9) is the classic two-level logic minimization problem [5]. NP-hard (Masek 1979, via reduction from Minimum Cover). The Quine–McCluskey algorithm enumerates all prime implicants, reducing the problem to minimum set cover. The worst-case number of prime implicants is $\Theta(3^n/\sqrt{n})$ (Chandra and Markowsky 1978).

Example. A function on $n = 3$ variables with 6 minterms has 6 prime implicants. The minimum cover requires 3 terms.

```
$ pred create --example MinimumDisjunctiveNormalForm -o mdnf.json
$ pred solve mdnf.json
```

Definition 2.112 (Rural Postman): Given an undirected graph $G = (V, E)$ with edge lengths $l : E \to \mathbb{Z}_{\geq 0}$ and a subset $E' \subseteq E$ of required edges, find a circuit (closed walk) in G that traverses every edge in E' and has minimum total length.

- Complexity: `2^num_vertices * num_vertices^2`.
- Reduces to: ILP.
- Reduces from: HamiltonianCircuit.

RuralPostman	
graph	The underlying graph G=(V,E)
edge_weights	Edge lengths l(e) for each e in E
required_edges	Edge indices of the required subset E' ⊆ E

The Rural Postman Problem (RPP) is a fundamental NP-complete arc-routing problem [91] that generalizes the Chinese Postman Problem. When $E' = E$, the problem reduces to finding an Eulerian circuit with minimum augmentation (polynomial-time solvable via T-join matching). For general $E' \subseteq E$, exact algorithms use dynamic programming over subsets of required edges in $O(n^2 \cdot 2^r)$ time, where $r = |E'|$ and $n = |V|$, analogous to the Held-Karp algorithm for TSP. The problem admits a 3/2-approximation for metric instances [92].

Example. Consider a graph with 6 vertices and 8 edges, where 6 outer edges have length 1 and 2 diagonal edges have length 2. The required edges are $E' = \{(v_0, v_1), (v_2, v_3), (v_4, v_5)\}$. The outer cycle $v_0 \to v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_0$ covers all 3 required edges with minimum total length 6.

```
$ pred create --example RuralPostman -o rural-postman.json
$ pred solve rural-postman.json
$ pred evaluate rural-postman.json --config 1,1,1,1,1,1,0,0
```

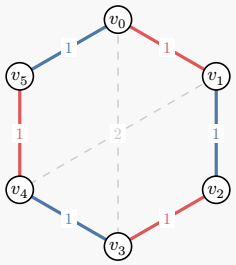


Figure 60: Rural Postman instance: 6 vertices, 8 edges, 3 required edges (red, bold). The outer cycle (blue + red edges) has minimum total cost 6, covering all required edges.

Definition 2.113 (Mixed Chinese Postman): Given a mixed graph $G = (V, A, E)$ with directed arcs A, undirected edges E, and integer lengths $l(e) \geq 0$ for every $e \in A \cup E$, find a closed walk in G that traverses every arc in its prescribed direction and every undirected edge at least once in some direction, minimizing total length.

- Complexity: `2^num_edges * num_vertices^3`.
- Reduces to: ILP.

MixedChinesePostman	
graph	The mixed graph G=(V,A,E)
arc_weights	Lengths for the directed arcs in A
edge_weights	Lengths for the undirected edges in E

Mixed Chinese Postman is the mixed-graph arc-routing problem ND25 in Garey and Johnson [5]. Papadimitriou proved the mixed case NP-complete even when all lengths are 1, the graph is planar, and the maximum degree is 3 [93]. In contrast, the pure undirected and pure directed cases are polynomial-time solvable via matching / circulation machinery [94]. The implementation here uses one binary variable per undirected edge orientation, so the search space contributes the $2^{|E|}$ factor visible in the registered exact bound.

Example. Consider the instance on 5 vertices with directed arcs (v_0, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_0) of lengths $2, 3, 1, 4$ and undirected edges $\{v_0, v_2\}$, $\{v_1, v_3\}$, $\{v_0, v_4\}$, $\{v_4, v_2\}$ of lengths $2, 3, 1, 2$. The config (1, 1, 0, 0) orients those edges as (v_2, v_0), (v_3, v_1), (v_0, v_4), and (v_4, v_2), producing a strongly connected digraph. The base traversal cost is 18, and the minimum balancing cost brings the total to 21.

```
$ pred create --example MixedChinesePostman/i32 -o mixed-chinese-postman.json
$ pred solve mixed-chinese-postman.json --solver brute-force
$ pred evaluate mixed-chinese-postman.json --config 1,1,0,0
```

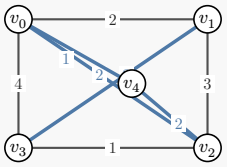


Figure 61: Mixed Chinese Postman example. Gray arrows are the original directed arcs, while blue arrows are the chosen orientations of the former undirected edges under config (1, 1, 0, 0). The optimal walk has total cost 21.

Definition 2.114 (Stacker Crane): Given a mixed graph $G = (V, A, E)$ with directed arcs A, undirected edges E, and nonnegative lengths $l : A \cup E \to \mathbb{Z}_{\geq 0}$, find a closed walk in G that traverses every arc in A in its prescribed direction and has minimum total length.

- Complexity: `num_vertices^2 * 2^num_arcs`.
- Reduces to: ILP.
- Reduces from: HamiltonianCircuit.

StackerCrane	
num_vertices	Number of vertices in the mixed graph
arcs	Required directed arcs that must be traversed
edges	Undirected edges available for connector paths
arc_lengths	Nonnegative lengths of the required directed arcs
edge_lengths	Nonnegative lengths of the undirected connector edges

Stacker Crane is the mixed-graph arc-routing problem listed as ND26 in Garey and Johnson [5]. Frederickson, Hecht, and Kim prove the problem NP-complete via reduction from Hamiltonian Circuit and give the classical 9/5-approximation for the metric case [95]. The problem stays difficult even on trees [96].

The standard Held-Karp-style dynamic program over (current vertex, covered-arc subset) runs in $O\big(|V|^2 \cdot 2^{|A|}\big)$ time[31].

A configuration is a permutation of the required arcs, interpreted as the order in which those arcs are forced into the tour. The verifier traverses each chosen arc, then inserts the shortest available connector path from that arc's head to the tail of the next arc, wrapping around at the end to close the walk.

Example. The canonical instance has 6 vertices, 5 required arcs, and 7 undirected edges. The optimal configuration [0, 2, 1, 4, 3] orders the required arcs as a_0, a_2, a_1, a_4, a_3. Traversing those arcs contributes 17 units of required-arc length, and the shortest connector paths contribute $1 + 1 + 1 + 0 + 0 = 3$, so the resulting closed walk has minimum total length 20.

```
$ pred create --example StackerCrane -o stacker-crane.json
$ pred solve stacker-crane.json --solver brute-force
$ pred evaluate stacker-crane.json --config 0,2,1,4,3
```

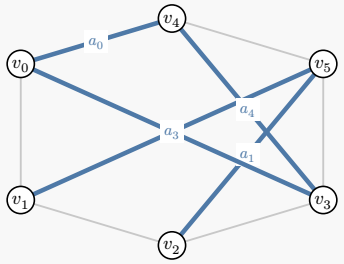


Figure 62: Stacker Crane hourglass instance. Required directed arcs are shown in blue and labeled a_0 through a_4; undirected connector edges are gray. The optimal order a_0, a_2, a_1, a_4, a_3 yields minimum total length 20.

Definition 2.115 (Subgraph Isomorphism): Given graphs $G = (V_1, E_1)$ (host) and $H = (V_2, E_2)$ (pattern), determine whether G contains a subgraph isomorphic to H: does there exist an injective function $f : V_2 \to V_1$ such that $\{u, v\} \in E_2 \Rightarrow \{f(u), f(v)\} \in E_1$?

- Complexity: `num_host_vertices ^ num_pattern_vertices`.
- Reduces to: ILP.
- Reduces from: KClique.

SubgraphIsomorphism	
graph	The host graph G = (V_1, E_1) to search in
pattern	The pattern graph H = (V_2, E_2) to find as a subgraph

Subgraph Isomorphism (GT48 in Garey & Johnson [5]) is NP-complete by transformation from Clique [5]. It strictly generalizes Clique (where $H = K_k$) and also contains Hamiltonian Circuit ($H = C_n$) and Hamiltonian Path ($H = P_n$) as special cases. Brute-force enumeration of all injective mappings $f : V_2 \to V_1$ runs in $O\big(|V_1|^{|V_2|} \cdot |E_2|\big)$ time. For fixed-size patterns, the color-coding technique of Alon, Yuster, and Zwick [19] gives a randomized algorithm in $2^{O(|V_2|)} \cdot |V_1|^{O(\mathrm{tw}(H))}$ time. Practical algorithms include VF2 [97] and VF2++ [98].

Example. Host graph $G = K_4$ (4 vertices, 6 edges), pattern $H = K_3$ (3 vertices, 3 edges). The mapping $f = (0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2)$ is injective and preserves all 3 pattern edges, confirming a subgraph isomorphism exists.

[31] Included as a straightforward exact dynamic-programming baseline over subsets of required arcs; no sharper exact bound was independently verified while preparing this entry.

```
$ pred create --example SubgraphIsomorphism -o subgraph-isomorphism.json
$ pred solve subgraph-isomorphism.json
$ pred evaluate subgraph-isomorphism.json --config 0,1,2
```

Definition 2.116 (Grouping by Swapping): Given a finite alphabet Σ, a string $x \in \Sigma^*$, and a positive integer K, determine whether there exists a sequence of at most K adjacent symbol interchanges that transforms x into a string $y \in \Sigma^*$ in which every symbol $a \in \Sigma$ appears in a single contiguous block. Equivalently, y contains no subsequence aba with distinct $a, b \in \Sigma$.

- Complexity: `string_len ^ budget`.

```
GroupingBySwapping
alphabet_size      Size of the alphabet
string             Input string over {0, ..., alphabet_size-1}
budget             Maximum number of adjacent swaps allowed
```

Grouping by Swapping is the storage-and-retrieval problem SR21 in Garey and Johnson [5]. It asks whether a string can be locally reorganized, using only adjacent transpositions, until equal symbols coalesce into blocks. The implementation in this crate uses a fixed-length swap program with one slot per allowed operation, so the direct brute-force search explores $O(|x|^K)$ configurations.[32]

Example. Let $\Sigma = \{\text{a, b, c}\}$, $x = \text{abcabc}$, and $K = 5$. The configuration $p = (2, 1, 3, 5, 5)$ performs adjacent swaps at positions $(2, 3)$, $(1, 2)$, and $(3, 4)$, then uses two trailing no-op slots. The resulting string is $y = \text{aabbcc}$, so every symbol now appears in one contiguous block and the verifier returns YES.

```
$ pred create --example GroupingBySwapping -o grouping-by-swapping.json
$ pred solve grouping-by-swapping.json --solver brute-force
$ pred evaluate grouping-by-swapping.json --config 2,1,3,5,5
```

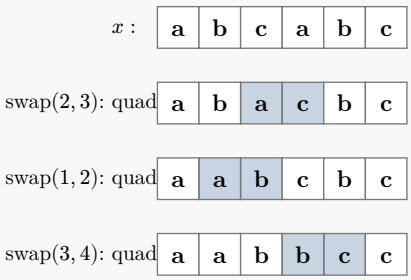


Figure 63: Grouping by Swapping on $x = \text{abcabc}$: three effective adjacent swaps turn the alternating string into $y = \text{aabbcc}$. The remaining two slots in $p = (2, 1, 3, 5, 5)$ are no-ops at position 5.

The final row has exactly one block of a, one block of b, and one block of c, so it satisfies the grouping constraint within the allotted budget.

Definition 2.117 (Longest Common Subsequence): Given a finite alphabet Σ and a set $R = \{r_1, ..., r_m\}$ of strings over Σ^*, find a longest string $w \in \Sigma^*$ such that every string $r_i \in R$ contains w as a *subsequence*: there exist indices $1 \leq j_1 < j_2 < ... < j_{|w|} \leq |r_i|$ with $r_{i[j_t]} = w[t]$ for all t.

- Complexity: `(alphabet_size + 1) ^ max_length`.
- Reduces to: ILP, MaximumIndependentSet.
- Reduces from: MinimumVertexCover.

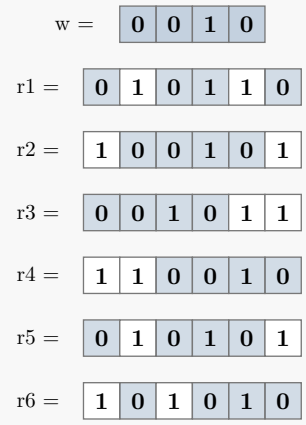
[32] This is the exact search bound induced by the fixed-length witness encoding implemented in the codebase; no sharper exact worst-case bound is claimed here.

```
LongestCommonSubsequence
alphabet_size            Size of the alphabet
strings                  Input strings over the alphabet {0, ..., alphabet_size-1}
max_length               Maximum possible subsequence length (min of string lengths)
```

A classic NP-hard string problem, listed as problem SR10 in Garey and Johnson [5]. D. Maier [99] proved NP-completeness of the decision version, while Garey and Johnson note polynomial-time cases for fixed $|R|$. For the special case of two strings, the classical dynamic-programming algorithm of R. A. Wagner and M. J. Fischer [100] runs in $O(|r_1| \cdot |r_2|)$ time. The optimization model implemented in this repository maximizes the subsequence length directly using a padding-based encoding.

Example. Let $\Sigma = \{0, 1\}$ and let the input set R contain the strings "010110", "100101", "001011", "110010", "010101", "101010". The witness $w =$ "0010" is a longest common subsequence of every string in R, with $|w| = 4$.

```
$ pred create --example LongestCommonSubsequence -o longest-common-subsequence.json
$ pred solve longest-common-subsequence.json
$ pred evaluate longest-common-subsequence.json --config 0,0,1,0,2,2
```



The highlighted positions show one left-to-right embedding of $w =$ "0010" in each input string, certifying that the longest common subsequence has length 4.

Definition 2.118 (Subset Sum): Given a finite set $A = \{a_0, ..., a_{n-1}\}$ with sizes $s(a_i) \in \mathbb{Z}^+$ and a target $B \in \mathbb{Z}^+$, determine whether there exists a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = B$.

- Complexity: `2^(num_elements / 2)`.
- Reduces to: CapacityAssignment, ClosestVectorProblem, IntegerExpressionMembership, IntegerKnapsack, Partition.
- Reduces from: KSatisfiability, Partition.

```
SubsetSum
sizes     Positive integer sizes s(a) for each element
target    Target sum B
```

One of Karp's 21 NP-complete problems [1]. Subset Sum is the special case of Knapsack where $v_i = w_i$ for all items and we seek an exact sum rather than an inequality. Though NP-complete, it is only *weakly* NP-hard: a dynamic-programming algorithm runs in $O(nB)$ pseudo-polynomial time. The best known exact algorithm is the $O^*(2^{n/2})$ meet-in-the-middle approach of Horowitz and Sahni [82].

Example. Let $A = \{3, 7, 1, 8, 2, 4\}$ ($n = 6$) and target $B = 11$. Selecting $A' = \{3, 8\}$ gives sum $3 + 8 = 11 = B$.

```
$ pred create --example SubsetSum -o subset-sum.json
$ pred solve subset-sum.json
$ pred evaluate subset-sum.json --config 1,0,0,1,0,0
```

Definition 2.119 (Subset Product): Given a finite set $A = \{a_0, ..., a_{n-1}\}$ with sizes $s(a_i) \in \mathbb{Z}^+$ and a target $B \in \mathbb{Z}^+$, determine whether there exists a subset $A' \subseteq A$ such that $\prod_{a \in A'} s(a) = B$.

- Complexity: `2^num_elements`.
- Reduces from: ExactCoverBy3Sets.

SubsetProduct	
sizes	Positive integer sizes s(a) for each element
target	Target product B

The multiplicative analogue of Subset Sum. Instead of seeking a subset whose elements sum to a target, we seek one whose product equals the target. NP-complete; the best known exact algorithm is an $O^*(2^{n/2})$ meet-in-the-middle approach analogous to Horowitz–Sahni for Subset Sum.

Example. Let $A = \{2, 3, 5, 7, 6, 10\}$ ($n = 6$) and target $B = 210$. Selecting $A' = \{2, 3, 5, 7\}$ gives product 2 times 3 times 5 times $7 = 210 = B$.

```
$ pred create --example SubsetProduct -o subset-product.json
$ pred solve subset-product.json
$ pred evaluate subset-product.json --config 1,1,1,1,0,0
```

Definition 2.120 (Resource Constrained Scheduling): Given a set T of n unit-length tasks, m identical processors, r resources with bounds B_i ($1 \leq i \leq r$), resource requirements $R_i(t)$ for each task t and resource i ($0 \leq R_i(t) \leq B_i$), and an overall deadline $D \in \mathbb{Z}^+$, determine whether there exists an m-processor schedule $\sigma : T \to \{0, ..., D-1\}$ such that for every time slot u, at most m tasks are scheduled at u and $\sum_{t:\sigma(t)=u} R_i(t) \leq B_i$ for each resource i.

- Complexity: `deadline ^ num_tasks`.
- Reduces to: ILP.
- Reduces from: ThreePartition.

ResourceConstrainedScheduling	
num_processors	Number of identical processors m
resource_bounds	Resource bound B_i for each resource i
resource_requirements	R_i(t) for each task t and resource i (n x r matrix)
deadline	Overall deadline D

RESOURCE CONSTRAINED SCHEDULING is problem SS10 in Garey & Johnson's compendium [5]. It is NP-complete in the strong sense, even for $r = 1$ resource and $m = 3$ processors, by reduction from 3-PARTITION [5]. For $m = 2$ processors with arbitrary r, the problem is solvable in polynomial time via bipartite matching. The general case subsumes bin-packing-style constraints across multiple resource dimensions.

Example. Let $n = 6$ tasks, $m = 3$ processors, $r = 1$ resource with $B_1 = 20$, and deadline $D = 2$. Resource requirements: $R_1(t_1) = 6$, $R_1(t_2) = 7$, $R_1(t_3) = 7$, $R_1(t_4) = 6$, $R_1(t_5) = 8$, $R_1(t_6) = 6$. Schedule: slot $0 \leftarrow \{t_1, t_2, t_3\}$ (3 tasks, resource $= 20$), slot $1 \leftarrow \{t_4, t_5, t_6\}$ (3 tasks, resource $= 20$). Both constraints satisfied; answer: YES.

Definition 2.121 (Boyce-Codd Normal Form Violation): **Instance:** A set A of attribute names, a collection F of functional dependencies on A, and a subset $A' \subseteq A$.

Question: Is there a subset $X \subseteq A'$ and two attributes $y, z \in A' \setminus X$ such that $y \in X^+$ but $z \notin X^+$, where X^+ is the closure of X under F?

- Complexity: `2^num_target_attributes * num_target_attributes^2 * num_functional_deps`.

BoyceCoddNormalFormViolation	
num_attributes	Total number of attributes in A
functional_deps	Functional dependencies (lhs_attributes, rhs_attributes)
target_subset	Subset A' of attributes to test for BCNF violation

A relation satisfies *Boyce-Codd Normal Form* (BCNF) if every non-trivial functional dependency $X \to Y$ has X as a superkey — that is, $X^+ = A'$. This classical NP-complete problem from database theory asks whether the given attribute subset A' violates BCNF. The NP-completeness was established by Beeri and Bernstein (1979) via reduction from Hitting Set. It appears as problem SR29 in Garey and Johnson's compendium (category A4: Storage and Retrieval).

Definition 2.122 (Consistency of Database Frequency Tables): Given a finite set V of objects, a finite set A of attributes, a domain D_a for each $a \in A$, a collection of pairwise frequency tables $f_{a,b} : D_a \times D_b \to \mathbb{Z}^{\geq 0}$ whose entries sum to $|V|$, and a set $K \subseteq V \times A \times \bigcup_{a \in A} D_a$ of known triples (v, a, x), determine whether there exist functions $g_a : V \to D_a$ such that $g_{a(v)} = x$ for every $(v, a, x) \in K$ and, for every published table $f_{a,b}$, exactly $f_{a,b}(x, y)$ objects satisfy $\left(g_{a(v)}, g_{b(v)}\right) = (x, y)$.

- Complexity: `domain_size_product^num_objects`.
- Reduces to: ILP.

ConsistencyOfDatabaseFrequencyTables	
num_objects	Number of objects in the database
attribute_domains	Domain size for each attribute
frequency_tables	Published pairwise frequency tables
known_values	Known object-attribute-value triples

Consistency of Database Frequency Tables is Garey and Johnson's storage-and-retrieval problem SR35 [5]. It asks whether released pairwise marginals can come from some hidden microdata table while respecting already known individual attribute values, making it a natural decision problem in statistical disclosure control. The direct witness space implemented in this crate assigns one categorical variable to each object-attribute pair, so exhaustive search runs in $O^*\left(\left(\prod_{a \in A} |D_a|\right)^{|V|}\right)$.[33]

Example. Let $|V| = 6$ with attributes a_0, a_1, a_2 having domain sizes 2, 3, and 2 respectively. Publish the pairwise tables

f_{a_0,a_1}	0	1	2
0	1	1	1
1	1	1	1

and

f_{a_1,a_2}	0	1
0	1	1
1	0	2

[33] This is the exact search bound induced by the implementation's configuration space; no faster general exact worst-case algorithm is claimed here.

f_{a_1,a_2}	0	1
2	1	1

together with the known values $K = \{(v_0, a_0, 0), (v_3, a_0, 1), (v_1, a_2, 1)\}$. One consistent completion is:

object	a_0	a_1	a_2
v_0	0	0	0
v_1	0	1	1
v_2	0	2	1
v_3	1	0	1
v_4	1	1	1
v_5	1	2	0

This witness satisfies every published count: in f_{a_0,a_1} each of the six cells appears exactly once, while in f_{a_1,a_2} the five occupied cells have multiplicities $1, 1, 2, 1, 1$ exactly as listed above. It also respects all three known triples, so the answer is YES.

```
$ pred create --example ConsistencyOfDatabaseFrequencyTables -o consistency-of-database-frequency-tables.json
$ pred solve consistency-of-database-frequency-tables.json
$ pred evaluate consistency-of-database-frequency-tables.json --config 0,0,0,0,1,1,0,2,1,1,0,1,1,1,1,1,2,0
```

Rule 2.4: (Consistency of Database Frequency Tables → Integer Linear Programming) Each object-attribute pair is encoded by a one-hot binary vector over its domain, and each pairwise frequency count becomes a linear equality over McCormick auxiliary variables that linearize the product of two one-hot indicators. Known values are fixed by pinning the corresponding indicator to 1. The resulting ILP is a pure feasibility problem (trivial objective).
Overhead: `num_vars = num_assignment_indicators + num_auxiliary_frequency_indicators`, `num_constraints = num_assignment_variables + num_known_values + num_frequency_cells + 3 * num_auxiliary_frequency_indicators`.

Proof: *Construction.* Let V be the set of objects, A the set of attributes with domains D_a, $\mathcal{T}$ the set of published frequency tables, and K the set of known triples (v, a, x).

Variables: (1) Binary one-hot indicators $y_{v,a,x} \in \{0, 1\}$ for each object $v \in V$, attribute $a \in A$, and value $x \in D_a$: $y_{v,a,x} = 1$ iff object v takes value x for attribute a. (2) Binary auxiliary variables $z_{t,v,x,x'} \in \{0, 1\}$ for each table $t \in \mathcal{T}$ (with attribute pair (a, b)), object $v \in V$, and cell $(x, x') \in D_a \times D_b$: $z_{t,v,x,x'} = 1$ iff object v realizes cell (x, x') in table t.

Constraints: (1) One-hot: $\sum_{x \in D_a} y_{v,a,x} = 1$ for all $v \in V$, $a \in A$. (2) Known values: $y_{v,a,x} = 1$ for each $(v, a, x) \in K$. (3) McCormick linearization for $z_{t,v,x,x'} = y_{v,a,x} \cdot y_{v,b,x'}$: $z_{t,v,x,x'} \leq y_{v,a,x}$, $z_{t,v,x,x'} \leq y_{v,b,x'}$, $z_{t,v,x,x'} \geq y_{v,a,x} + y_{v,b,x'} - 1$. (4) Frequency counts: $\sum_{v \in V} z_{t,v,x,x'} = f_t(x, x')$ for each table t and cell (x, x').

Objective: Minimize 0 (feasibility problem).

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{x \in D_a} y_{v,a,x} = 1 \quad \forall v \in V, a \in A \\
& y_{v,a,x} = 1 \quad \forall (v,a,x) \in K \\
& z_{t,v,x,x'} \leq y_{v,a,x} \quad \forall t \in \mathcal{T}, v \in V, (x,x') \in D_a \times D_b \\
& z_{t,v,x,x'} \leq y_{v,b,x'} \quad \forall t \in \mathcal{T}, v \in V, (x,x') \in D_a \times D_b \\
& z_{t,v,x,x'} \geq y_{v,a,x} + y_{v,b,x'} - 1 \quad \forall t \in \mathcal{T}, v \in V, (x,x') \in D_a \times D_b \\
& \sum_{v \in V} z_{t,v,x,x'} = f_{t(x,x')} \quad \forall t \in \mathcal{T}, (x,x') \in D_a \times D_b \\
& y_{v,a,x}, z_{t,v,x,x'} \in \{0,1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A consistent assignment defines one-hot indicators and their products; all constraints hold by construction, and the frequency equalities match the published counts. ($\Leftarrow$) Any feasible binary solution assigns exactly one value per object-attribute (one-hot), respects known values, and the McCormick constraints force $z_{t,v,x,x'} = y_{v,a,x} \cdot y_{v,b,x'}$ for binary variables, so the frequency equalities certify consistency.

Solution extraction. For each object v and attribute a, find x with $y_{v,a,x} = 1$; assign value x to (v, a). □

Definition 2.123 (Sum of Squares Partition): Given a finite set $A = \{a_0, ..., a_{n-1}\}$ with sizes $s(a_i) \in \mathbb{Z}^+$ and a positive integer $K \leq |A|$ (number of groups), find a partition of A into K disjoint sets $A_1, ..., A_K$ that minimizes $\sum_{i=1}^{K} \left(\sum_{a \in A_i} s(a)\right)^2$.

- Complexity: `num_groups^num_elements`.
- Reduces to: ILP.

SumOfSquaresPartition	
sizes	Positive integer size s(a) for each element a in A
num_groups	Number of groups K in the partition

Problem SP19 in Garey and Johnson [5]. NP-complete in the strong sense, so no pseudo-polynomial time algorithm exists unless $P = \text{NP}$. For fixed K, a dynamic-programming algorithm runs in $O(nS^{K-1})$ pseudo-polynomial time, where $S = \sum s(a)$. The problem remains NP-complete when the exponent 2 is replaced by any fixed rational $\alpha > 1$.[34] The squared objective penalizes imbalanced partitions, connecting it to variance minimization, load balancing, and k-means clustering. Sum of Squares Partition generalizes Partition ($K = 2$, $J = S^2/2$).

Example. Let $A = \{5, 3, 8, 2, 7, 1\}$ ($n = 6$) and $K = 3$ groups. The partition $A_1 = \{8, 1\}$, $A_2 = \{5, 2\}$, $A_3 = \{3, 7\}$ gives group sums $9, 7, 10$ and sum of squares $81 + 49 + 100 = 230$. The optimal partition has group sums $\{9, 9, 8\}$ yielding $81 + 81 + 64 = 226$.

Definition 2.124 (3-Partition): Given a set $A = \{a_0, ..., a_{3m-1}\}$ of $3m$ elements, a bound $B \in \mathbb{Z}^+$, and sizes $s(a) \in \mathbb{Z}^+$ such that $\frac{B}{4} < s(a) < \frac{B}{2}$ for every $a \in A$ and $\sum_{a \in A} s(a) = mB$, determine whether A can be partitioned into m disjoint triples $A_1, ..., A_m$ with $\sum_{a \in A_i} s(a) = B$ for every i.

- Complexity: `3^num_elements`.
- Reduces to: FlowShopScheduling, JobShopScheduling, ResourceConstrainedScheduling, SequencingToMinimizeWeightedTardiness, SequencingWithReleaseTimesAndDeadlines.
- Reduces from: ThreeDimensionalMatching.

[34] No algorithm improving on brute-force $O(K^n)$ enumeration is known for the general case.

ThreePartition	
sizes	Positive integer sizes s(a) for each element a in A
bound	Target sum B for each triple

3-Partition is Garey and Johnson's strongly NP-complete benchmark SP15 [5]. Unlike ordinary Partition, the strict size window forces every feasible block to contain exactly three elements, making the problem the canonical source for strong NP-completeness reductions to scheduling, packing, and layout models. The implementation in this repository uses one group-assignment variable per element, so the exported exact-search baseline is $O^*(3^n)$[35].

Example. Let $B = 15$ and consider the 6-element instance with sizes (4, 5, 6, 4, 6, 5). The witness triples $A_1 = \{4, 5, 6\}$ and $A_2 = \{4, 6, 5\}$ both sum to 15, so this instance is satisfiable.

```
$ pred create --example ThreePartition -o three-partition.json
$ pred solve three-partition.json
$ pred evaluate three-partition.json --config 0,0,0,1,1,1
```

Triple	Elements	Sum
A_1	4, 5, 6	15
A_2	4, 6, 5	15

Definition 2.125 (Numerical 3-Dimensional Matching): Given disjoint sets W, X, Y each with m elements, positive integer sizes $s(a)$ with $B/4 < s(a) < B/2$ for every element, and a bound $B \in \mathbb{Z}^+$ such that $\sum s(a) = mB$, determine whether $W \cup X \cup Y$ can be partitioned into m triples, each containing one element from W, X, and Y, with each triple summing to exactly B.

- Complexity: `num_groups^(2 * num_groups)`.

Numerical3DimensionalMatching	
sizes_w	Positive integer sizes for each element of W
sizes_x	Positive integer sizes for each element of X
sizes_y	Positive integer sizes for each element of Y
bound	Target sum B for each triple

Numerical 3-Dimensional Matching is strongly NP-complete (SP16 in Garey and Johnson [5]). The strict size window $B/4 < s(a) < B/2$ forces every feasible triple to contain exactly one element from each set. The problem is a key intermediate in strong NP-completeness reductions to bin packing, scheduling, and layout problems. Brute-force enumeration runs in $O^*(m^{2m})$ time.

Example. Let $m = 2$ and $B = 15$. The sizes are $W = (4, 5)$, $X = (4, 5)$, $Y = (5, 7)$. The matching pairs each w_i with $x_{\pi(i)}$ and $y_{\sigma(i)}$: $w_0 + x_0 + y_1 = 15$, $w_1 + x_1 + y_0 = 15$, all equal to B.

```
$ pred create --example Numerical3DimensionalMatching -o n3dm.json
$ pred solve n3dm.json
$ pred evaluate n3dm.json --config 0,1,1,0
```

Definition 2.126 (Numerical Matching with Target Sums): Given two disjoint sets X and Y each with m elements, integer sizes $s(x_i)$ for $x_i \in X$ and $s(y_j)$ for $y_j \in Y$, and a multiset of m target values $B_1, ..., B_m$,

[35] This is the direct worst-case bound induced by the implementation's configuration space and matches the registered catalog expression `3^num_elements`; no sharper general exact bound was independently verified while preparing this entry.

determine whether $X \cup Y$ can be partitioned into m pairs, each containing one element from X and one from Y, such that the multiset of pair sums $\{s(x_i) + s(y_{\pi(i)})\}$ equals the target multiset.

- Complexity: `2^num_pairs`.
- Reduces to: ILP.

NumericalMatchingWithTargetSums	
sizes_x	Integer sizes for each element of X
sizes_y	Integer sizes for each element of Y
targets	Target sums for each pair

Numerical Matching with Target Sums is NP-complete in the strong sense (SP17 in Garey and Johnson [5]). It generalizes bipartite perfect matching by imposing sum constraints on each pair. Brute-force enumeration runs in $O^*(2^m)$ time by trying all $m!$ permutations.

Example. Let $m = 3$, $X = (1, 4, 7)$, $Y = (2, 5, 3)$, targets $= (3, 7, 12)$. The matching $\pi = (0, 2, 1)$ yields sums $1 + 2 = 3$, $4 + 3 = 7$, $7 + 5 = 12$, which as a multiset equals the targets.

```
$ pred create --example NumericalMatchingWithTargetSums -o nmts.json
$ pred solve nmts.json
$ pred evaluate nmts.json --config 0,2,1
```

Rule 2.5: (Numerical Matching with Target Sums → Integer Linear Programming) Introduce a binary variable $z_{i,j,k} \in \{0, 1\}$ for each *compatible triple* (i, j, k) where $s(x_i) + s(y_j) = B_k$. The constraints ensure a perfect matching: $\sum_{j,k} z_{i,j,k} = 1$ for each i (every x_i matched once), $\sum_{i,k} z_{i,j,k} = 1$ for each j (every y_j matched once), $\sum_{i,j} z_{i,j,k} = 1$ for each k (every target used once). The objective is trivial (minimize 0), since this is a feasibility problem.
Overhead: `num_vars = num_pairs * num_pairs * num_pairs`, `num_constraints = 3 * num_pairs`.

Proof: *Correctness.* By construction, variables are only created for triples satisfying $s(x_i) + s(y_j) = B_k$. The three families of equality constraints enforce that the assignment is a bijection on X, Y, and the target indices. Any feasible ILP solution therefore defines a permutation π with $s(x_i) + s(y_{\pi(i)})$ matching a distinct target, and conversely any valid matching maps to a feasible binary assignment. The number of variables is at most m^3 (all triples compatible), and the number of constraints is $3m$. □

Example: Numerical Matching with Target Sums to ILP via compatible-triple assignment variables.

Source: NumericalMatchingWithTargetSums **Target:** ILP

Definition 2.127 (Non-Liveness Free Petri Net): Given a free-choice Petri net $P = (S, T, F, M_0)$ with $|S|$ places, $|T|$ transitions, flow relation F, and initial marking M_0, determine whether P is *not live*: does there exist a transition $t \in T$ and a marking M reachable from M_0 such that t can never fire again from M?

- Complexity: `(initial_token_sum + 1) ^ num_places * num_transitions`.

NonLivenessFreePetriNet	
num_places	Number of places |S|
num_transitions	Number of transitions |T|
place_to_transition	Arcs from places to transitions
transition_to_place	Arcs from transitions to places
initial_marking	Initial marking M_0 (tokens per place)

Non-Liveness of free-choice Petri nets is NP-complete (Garey and Johnson [5]). A Petri net is *free-choice* if every two transitions sharing an input place have identical presets. The implementation explores the

bounded reachability graph (capped at the initial token sum per place) and checks whether any transition becomes permanently dead.

Example. A chain net with 4 places and 3 transitions: t_0 moves a token from s_0 to s_1, t_1 from s_1 to s_2, t_2 from s_2 to s_3. Starting from $M_0 = (1, 0, 0, 0)$, after all transitions fire once the net reaches deadlock at $(0, 0, 0, 1)$ and all transitions are permanently dead. The witness configuration (1, 1, 1) confirms all transitions are globally dead.

```
$ pred create --example NonLivenessFreePetriNet -o petri.json
$ pred solve petri.json --solver brute-force
$ pred evaluate petri.json --config 1,1,1
```

Definition 2.128 (Minimum Axiom Set): Given a finite set of sentences $S = \{s_0, ..., s_{n-1}\}$, a subset $T \subseteq S$ of true sentences, and a set of implications $\{(A_j, c_j)\}$ where each $A_j \subseteq S$ and $c_j \in S$, find a smallest subset $S_0 \subseteq T$ such that the deductive closure of S_0 under the implications equals T. That is, starting from S_0, repeatedly applying every rule "if all sentences in A_j hold then c_j holds" until no new sentences are added must yield exactly T.

- Complexity: `2^num_true_sentences`.
- Reduces from: ExactCoverBy3Sets.

MinimumAxiomSet	
num_sentences	Total number of sentences |S|
true_sentences	Indices of true sentences T ⊆ S
implications	Implication rules (antecedents, consequent)

Minimum Axiom Set is problem LO6 in Garey and Johnson [5]. The problem models finding a minimal set of assumptions (axioms) from which all truths in a theory can be derived. It generalises set cover: when every implication has a single antecedent, the problem reduces to finding a minimum dominating set in the implication graph. No algorithm improving on brute-force ($O(2^{|T|})$) is known for the general case.[36]

Example. Let $S = \{s_0, ..., s_7\}$ ($n = 8$) with $T = S$ (all sentences true) and implications ($\{0\} \to 2$), ($\{0\} \to 3$), ($\{1\} \to 4$), ($\{1\} \to 5$), ($\{2, 4\} \to 6$), ($\{3, 5\} \to 7$), ($\{6, 7\} \to 0$), ($\{6, 7\} \to 1$). Selecting axioms $S_0 = \{0, 1\}$ generates the full deductive closure T in three rounds: first $\{0, 1\} \to \{2, 3, 4, 5\}$, then $\{2, 4\}, \{3, 5\} \to \{6, 7\}$, then $\{6, 7\} \to \{0, 1\}$ (already present). The optimal value is 2.

```
$ pred create --example MinimumAxiomSet -o axiom.json
$ pred solve axiom.json --solver brute-force
$ pred evaluate axiom.json --config 1,1,0,0,0,0,0,0
```

Definition 2.129 (Betweenness): Given a finite set $A = \{a_0, ..., a_{n-1}\}$ of n elements and a collection C of ordered triples (a, b, c), determine whether there exists a linear ordering $f : A \to \{0, ..., n-1\}$ (a bijection) such that for every $(a, b, c) \in C$, either $f(a) < f(b) < f(c)$ or $f(c) < f(b) < f(a)$ — that is, b is *between* a and c in the ordering.

- Complexity: `2^num_elements`.
- Reduces from: SetSplitting.

Betweenness	
num_elements	Number of elements in the set A
triples	Collection of ordered triples (a, b, c) requiring b between a and c

[36] No algorithm improving on brute-force is known for the general Minimum Axiom Set problem.

Betweenness is problem MS1 in Garey and Johnson [5]. It arises in seriation, archaeological sequencing, and DNA physical mapping. The problem is NP-complete even when restricted to dense constraint sets. The implementation represents a solution as a permutation f where $f(i)$ is the position assigned to element i.

Example. Consider $n = 5$ elements with triples $(0, 1, 2)$, $(2, 3, 4)$, $(0, 2, 4)$, $(1, 3, 4)$. The witness ordering $f = (0, 1, 2, 3, 4)$ (the identity permutation) satisfies all constraints: each middle element of every triple lies between the other two in the ordering.

```
$ pred create --example Betweenness -o betweenness.json
$ pred solve betweenness.json
$ pred evaluate betweenness.json --config 0,1,2,3,4
```

Definition 2.130 (Cyclic Ordering): Given a finite set $A = \{a_0, ..., a_{n-1}\}$ of n elements and a collection C of ordered triples (a, b, c), determine whether there exists a permutation $f : A \to \{0, ..., n-1\}$ (a bijection) such that for every $(a, b, c) \in C$, the values $f(a)$, $f(b)$, $f(c)$ appear in cyclic order — i.e., $f(a) < f(b) < f(c)$ or $f(b) < f(c) < f(a)$ or $f(c) < f(a) < f(b)$.

- Complexity: `factorial(num_elements)`.
- Reduces from: KSatisfiability.

CyclicOrdering	
num_elements	Number of elements in the set A
triples	Collection of ordered triples (a, b, c) requiring cyclic order

Cyclic Ordering is problem MS2 in Garey and Johnson [5]. It is closely related to Betweenness (MS1) but enforces a cyclic rather than linear ordering constraint. The problem is NP-complete. The implementation represents a solution as a permutation f where $f(i)$ is the position assigned to element i.

Example. Consider $n = 5$ elements with triples $(0, 1, 2)$, $(2, 3, 0)$, $(1, 3, 4)$. The witness ordering $f = (1, 3, 4, 0, 2)$ satisfies all constraints: each triple's elements appear in cyclic order under f.

```
$ pred create --example CyclicOrdering -o cyclic_ordering.json
$ pred solve cyclic_ordering.json
$ pred evaluate cyclic_ordering.json --config 1,3,4,0,2
```

Definition 2.131 (Clustering): Given a set of n elements with a symmetric distance function $d : \binom{A}{2} \to \mathbb{N}$ (represented as a matrix with zero diagonal), a cluster count bound K, and a diameter bound B, determine whether there exists a partition of the elements into at most K non-empty clusters such that for every cluster C, all pairwise distances within C satisfy $d(i, j) \leq B$.

- Complexity: `num_clusters^num_elements`.
- Reduces to: ILP.
- Reduces from: KColoring.

Clustering	
distances	Symmetric distance matrix with zero diagonal
num_clusters	Maximum number of clusters K
diameter_bound	Maximum allowed intra-cluster pairwise distance B

Clustering is a fundamental problem in unsupervised learning and data analysis. The variant considered here is the diameter-bounded formulation, which is NP-complete. No algorithm improving on brute-force (K^n enumeration) is known for the general case.[37]

[37] No algorithm improving on brute-force is known for general diameter-bounded clustering.

Example. Consider $n = 6$ elements with $K = 2$ clusters and diameter bound $B = 1$. The distance matrix has two tight groups $\{0, 1, 2\}$ and $\{3, 4, 5\}$ with intra-group distance 1 and inter-group distance 3. The witness assignment (0, 0, 0, 1, 1, 1) partitions elements into clusters $\{0, 1, 2\}$ and $\{3, 4, 5\}$; each cluster has maximum pairwise distance $1 \leq 1$.

```
$ pred create --example Clustering -o clustering.json
$ pred solve clustering.json
$ pred evaluate clustering.json --config 0,0,0,1,1,1
```

Definition 2.132 (Dynamic Storage Allocation): Given n items, each with arrival time $r(a)$, departure time $d(a)$, and size $s(a)$, and a storage size D, determine whether there exists a starting address $\sigma(a) \in \{0, ..., D - s(a)\}$ for each item a such that for every pair of items a, a' with overlapping time intervals ($r(a) < d(a')$ and $r(a') < d(a)$), the memory intervals $[\sigma(a), \sigma(a) + s(a) - 1]$ and $[\sigma(a'), \sigma(a') + s(a') - 1]$ are disjoint.

- Complexity: `(memory_size + 1)^num_items`.

DynamicStorageAllocation	
items	Items as (arrival, departure, size) tuples
memory_size	Total memory size D

Dynamic Storage Allocation is Garey and Johnson's SR2 [5] and models memory allocation for processes with known lifetimes. It generalises strip-packing and bin-packing with time constraints. The implementation encodes each item's starting address as a single variable with domain $D - s(a) + 1$.

Example. Let $D = 6$ and consider 5 items with (r, d, s) tuples $(0, 3, 2)$, $(0, 2, 3)$, $(1, 4, 1)$, $(2, 5, 3)$, $(3, 5, 2)$. The witness assignment $\sigma = (0, 2, 5, 2, 0)$ places every item within $[0, 5]$ and ensures no two time-overlapping items share memory cells.

```
$ pred create --example DynamicStorageAllocation -o dynamic-storage-allocation.json
$ pred solve dynamic-storage-allocation.json
$ pred evaluate dynamic-storage-allocation.json --config 0,2,5,2,0
```

Item	Arrival	Departure	Size	σ
a_0	0	3	2	0
a_1	0	2	3	2
a_2	1	4	1	5
a_3	2	5	3	2
a_4	3	5	2	0

Definition 2.133 (Kth Largest m-Tuple): Given m finite sets $X_1, ..., X_m$ of positive integers, a bound $B \in \mathbb{Z}^+$, and a threshold $K \in \mathbb{Z}^+$, count the number of distinct m-tuples $(x_1, ..., x_m) \in X_1 \times \cdots \times X_m$ satisfying $\sum_{i=1}^{m} x_i \geq B$. The answer is *yes* iff this count is at least K.

- Complexity: `total_tuples * num_sets`.

KthLargestMTuple	
sets	m sets, each containing positive integer sizes
k	Threshold K (answer YES iff count >= K)
bound	Lower bound B on tuple sum

The Kth Largest m-Tuple problem is MP10 in Garey and Johnson's appendix [5]. It is *not known to be in NP*, because a "yes" certificate may need to exhibit K qualifying tuples and K can be exponentially large. The problem is PP-complete under polynomial-time Turing reductions [101], though the special case $m = 2$, $K = 1$ is NP-complete via reduction from Subset Sum. In the general case, the only known exact approach is brute-force enumeration of all $\prod_{i=1}^{m} |X_i|$ tuples, so the registered catalog complexity is `total_tuples * num_sets`[38].

Example. Let $m = 3$, $B = 12$, and $K = 14$ with sets $X_1 = \{2, 5, 8\}$, $X_2 = \{3, 6\}$, $X_3 = \{1, 4, 7\}$. The Cartesian product has 18 tuples. For instance, the tuple (8, 6, 7) has sum $21 \geq 12$, contributing 1 to the count. In total, 14 of the 18 tuples satisfy the bound, so the answer is *yes* (count $= K$).

```
$ pred create --example KthLargestMTuple -o kth-largest-m-tuple.json
$ pred solve kth-largest-m-tuple.json --solver brute-force
$ pred evaluate kth-largest-m-tuple.json --config 2,1,2
```

Definition 2.134 (Sequencing with Release Times and Deadlines): Given a set T of n tasks and, for each task $t \in T$, a processing time $\ell(t) \in \mathbb{Z}^+$, a release time $r(t) \in \mathbb{Z}^{\geq 0}$, and a deadline $d(t) \in \mathbb{Z}^+$, determine whether there exists a one-processor schedule $\sigma : T \to \mathbb{Z}^{\geq 0}$ such that for all $t \in T$: $\sigma(t) \geq r(t)$, $\sigma(t) + \ell(t) \leq d(t)$, and no two tasks overlap (i.e., $\sigma(t) > \sigma(t')$ implies $\sigma(t) \geq \sigma(t') + \ell(t')$).

- Complexity: `2^num_tasks * num_tasks`.
- Reduces to: ILP.
- Reduces from: ThreePartition.

SequencingWithReleaseTimesAndDeadlines	
lengths	Processing time l(t) for each task (positive)
release_times	Release time r(t) for each task (non-negative)
deadlines	Deadline d(t) for each task (positive)

Problem SS1 in Garey and Johnson's appendix [5], and a fundamental single-machine scheduling feasibility problem. It is strongly NP-complete by reduction from 3-Partition, so no pseudo-polynomial time algorithm exists unless P = NP. The problem becomes polynomial-time solvable when: (1) all task lengths equal 1, (2) preemption is allowed, or (3) all release times are zero. The best known exact algorithm for the general case runs in $O^*(2^n \cdot n)$ time via dynamic programming on task subsets.

Example. Consider 5 tasks:

	t_1	t_2	t_3	t_4	t_5
$\ell(t)$	3	2	4	1	2
$r(t)$	0	1	5	0	8
$d(t)$	5	6	10	3	12

A feasible schedule: $\sigma(t_4) = 0$ (runs $[0, 1)$), $\sigma(t_1) = 1$ (runs $[1, 4)$), $\sigma(t_2) = 4$ (runs $[4, 6)$), $\sigma(t_3) = 6$ (runs $[6, 10)$), $\sigma(t_5) = 10$ (runs $[10, 12)$). All release and deadline constraints are satisfied with no overlap.

```
$ pred create --example SequencingWithReleaseTimesAndDeadlines -o sequencing-with-release-
times-and-deadlines.json
$ pred solve sequencing-with-release-times-and-deadlines.json
$ pred evaluate sequencing-with-release-times-and-deadlines.json --config 3,0,0,0,0
```

[38] No algorithm improving on brute-force is known for the general Kth Largest m-Tuple problem.

Definition 2.135 (Partition): Given a finite set $A = \{a_0, ..., a_{n-1}\}$ with sizes $s(a_i) \in \mathbb{Z}^+$, determine whether there exists a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)$.

- Complexity: `2^(num_elements / 2)`.
- Reduces to: BinPacking, CosineProductIntegration, Knapsack, MultiprocessorScheduling, OpenShopScheduling, ProductionPlanning, SequencingToMinimizeTardyTaskWeight, SequencingWithinIntervals, ShortestWeightConstrainedPath, SubsetSum.
- Reduces from: SubsetSum.

```
Partition
sizes     Positive integer size for each element
```

One of Karp's 21 NP-complete problems [1], listed as SP12 in Garey & Johnson [5]. Partition is the special case of Subset Sum where the target equals half the total sum. Though NP-complete, it is only *weakly* NP-hard: a dynamic-programming algorithm runs in $O(n \cdot B_{\text{total}})$ pseudo-polynomial time, where $B_{\text{total}} = \sum_i s(a_i)$. The best known exact algorithm is the $O^*(2^{n/2})$ meet-in-the-middle approach of Schroeppel and Shamir (1981).

Example. Let $A = \{3, 1, 1, 2, 2, 1\}$ ($n = 6$, total sum $= 10$). Setting $A' = \{3, 2\}$ (indices 0, 3) gives sum $3 + 2 = 5 = 10/2$, and $A \setminus A' = \{1, 1, 2, 1\}$ also sums to 5. Hence a balanced partition exists.

Definition 2.136 (Cosine Product Integration): Given a sequence of integers $(a_1, a_2, ..., a_n)$, determine whether there exists a sign assignment $\varepsilon \in \{-1, +1\}^n$ such that $\sum_{i=1}^{n} \varepsilon_i a_i = 0$.

- Complexity: `2^(num_coefficients / 2)`.
- Reduces from: Partition.

```
CosineProductIntegration
coefficients            Integer cosine frequencies
```

Garey & Johnson problem A7/AN14. The original formulation asks whether $\int_0^{2\pi} \prod_{i=1}^{n} \cos(a_i \theta) d\theta = 0$; by expanding each cosine as $\left(e^{ia_i\theta} + e^{-ia_i\theta}\right)/2$ via Euler's formula and integrating, the integral equals $(2\pi/2^n)$ times the number of sign assignments ε with $\sum \varepsilon_i a_i = 0$. Hence the integral is nonzero if and only if a balanced sign assignment exists, making this equivalent to a generalisation of Partition to signed integers. NP-complete by reduction from Partition [102]. Solvable in pseudo-polynomial time via dynamic programming on achievable partial sums.

Example. Let $(a_1, a_2, a_3) = (2, 3, 5)$. The sign assignment $(+1, +1, -1)$ gives $2 + 3 - 5 = 0$, so the integral is nonzero.

Definition 2.137 (Shortest Common Supersequence): Given a finite alphabet Σ and a set $R = \{r_1, ..., r_m\}$ of strings over Σ^*, find a string $w \in \Sigma^*$ of minimum length such that every string $r_i \in R$ is a *subsequence* of w: there exist indices $1 \leq j_1 < j_2 < ... < j_{|r_i|} \leq |w|$ with $w[j_k] = r_i[k]$ for all k.

- Complexity: `(alphabet_size + 1) ^ max_length`.
- Reduces to: ILP.

```
ShortestCommonSupersequence
alphabet_size               Size of the alphabet
strings                     Input strings over the alphabet {0, ..., alphabet_size-1}
max_length                  Maximum possible supersequence length (sum of all string lengths)
```

A classic NP-hard string problem, listed as problem SR8 in Garey and Johnson [5]. D. Maier [99] proved NP-completeness of the decision version; K.-J. Räihä and E. Ukkonen [103] showed the problem remains NP-complete even over a binary alphabet ($|\Sigma| = 2$). Note that *subsequence* (characters may be non-

contiguous) differs from *substring* (contiguous block): the Shortest Common Supersequence asks that each input string can be embedded into w by selecting characters in order but not necessarily adjacently.

For $|R| = 2$ strings, the problem is solvable in polynomial time via the duality with the Longest Common Subsequence (LCS): if $\mathrm{LCS}(r_1, r_2)$ has length ℓ, then the shortest common supersequence has length $|r_1| + |r_2| - \ell$, computable in $O(|r_1| \cdot |r_2|)$ time by dynamic programming. For general $|R| = m$, the brute-force search explores all candidate supersequences up to the maximum possible length $\sum_i |r_i|$. Applications include bioinformatics (reconstructing ancestral sequences from fragments), data compression (representing multiple strings compactly), and scheduling (merging instruction sequences).

Example. Let $\Sigma = \{\mathrm{a}, \mathrm{b}\}$ and $R = \{\text{"ab"}, \text{"ba"}\}$. We seek the shortest string w that contains every r_i as a subsequence.

```
$ pred create --example ShortestCommonSupersequence -o shortest-common-supersequence.json
$ pred solve shortest-common-supersequence.json
$ pred evaluate shortest-common-supersequence.json --config 0,1,0,2
```

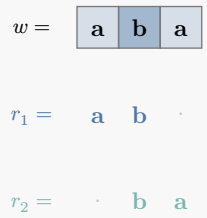


Figure 65: Shortest Common Supersequence: $w =$ "aba" (length 3) contains $r_1 =$ "ab" (positions 0,1), $r_2 =$ "ba" (positions 1,2) as subsequences. Dots mark unused positions.

The optimal supersequence $w =$ "aba" has length 3 and contains all 2 input strings as subsequences.

Definition 2.138 (String-to-String Correction): Given a finite alphabet Σ, a source string $x \in \Sigma^*$, a target string $y \in \Sigma^*$, and a positive integer K, determine whether y can be derived from x by a sequence of at most K operations, where each operation is either a *single-symbol deletion* (remove one character at a chosen position) or an *adjacent-symbol interchange* (swap two neighboring characters).

- Complexity: `(2 * source_length + 1) ^ bound`.
- Reduces to: ILP.

StringToStringCorrection	
alphabet_size	Size of the finite alphabet
source	Source string (symbol indices)
target	Target string (symbol indices)
bound	Maximum number of operations allowed

A classical NP-complete problem listed as SR20 in Garey and Johnson [5]. R. A. Wagner [104] proved NP-completeness via transformation from Set Covering. The standard edit distance problem — allowing insertion, deletion, and substitution — is solvable in $O(|x| \cdot |y|)$ time by the Wagner–Fischer dynamic programming algorithm [105]. However, restricting the operation set to only deletions and adjacent swaps makes the problem NP-complete for unbounded alphabets. When only adjacent swaps are allowed (no deletions), the problem reduces to counting inversions and is polynomial [104].[39]

Example. Let $\Sigma = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$, source $x =$ abcdba (length 6), target $y =$ abdcb (length 5), and $K = 2$.

[39] No algorithm improving on brute-force is known for the general swap-and-delete variant.

```
$ pred create --example StringToStringCorrection -o string-to-string-correction.json
$ pred solve string-to-string-correction.json
$ pred evaluate string-to-string-correction.json --config 8,5
```

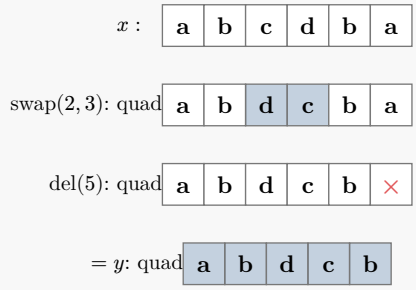


Figure 66: String-to-String Correction: transforming $x = \text{abcdba}$ into $y = \text{abdcb}$ with $K = 2$ operations. Step 1 swaps adjacent symbols at positions 2 and 3; step 2 deletes the symbol at position 5.

The transformation uses exactly $K = 2$ operations (1 swap + 1 deletion), which is the minimum: a single operation cannot account for both the transposition of two symbols and the removal of one.

Definition 2.139 (Minimum External Macro Data Compression): Given a finite alphabet Σ of size k, a string $s \in \Sigma^*$ of length n, and a pointer cost $h \in \mathbb{Z}^+$, find a dictionary string $D \in \Sigma^*$ and a compressed string $C \in (\Sigma \cup \{p_1, ..., p_n\})^*$, where each p_i is a pointer referencing a contiguous substring of D, such that s can be obtained from C by replacing every pointer with its referenced substring, minimizing the total cost $|D| + |C| + (h - 1) \times$ (number of pointer occurrences in C).

- Complexity: `(alphabet_size + 1) ^ string_length * (alphabet_size + 1 + string_length * (string_length + 1) / 2) ^ string_length`.
- Reduces to: ILP.

MinimumExternalMacroDataCompression	
alphabet_size	Size of the alphabet (symbols indexed 0..alphabet_size)
string	Source string as symbol indices
pointer_cost	Pointer cost h (each pointer contributes h to the cost)

A classical NP-hard data compression problem, listed as SR22 in Garey and Johnson [5]. The macro model of data compression was introduced by J. A. Storer [106], who proved NP-completeness via transformation from Vertex Cover. J. A. Storer and T. G. Szymanski [107] provided a comprehensive analysis of the macro compression framework, showing that NP-completeness persists even when h is any fixed integer ≥ 2, when the alphabet has ≥ 3 symbols, and when D contains no pointers (the "external" variant). The LZ-family of practical compression algorithms (LZ77, LZSS, LZ78) are restricted forms of this general macro model. The related Smallest Grammar Problem is APX-hard [108].[40]

Example. Let $\Sigma = \{\text{a, b, c, d, e, f}\}$ and $s = \text{abcdefabcdefabcdef}$ (length 18) with pointer cost $h = 2$.

```
$ pred create --example MinimumExternalMacroDataCompression -o min-emdc.json
$ pred solve min-emdc.json
$                pred                 evaluate                 min-emdc.json                --
config 0,1,2,3,4,5,6,6,6,6,6,6,6,6,6,6,6,12,12,12,6,6,6,6,6,6,6,6,6,6,6,6,6,6
```

[40] No algorithm improving on brute-force enumeration is known for optimal external macro compression.

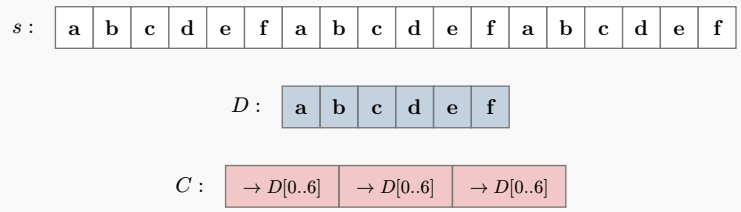


Figure 67: Minimum External Macro Data Compression: with $s = \text{abcdefabcdefabcdef}$ (length 18) and pointer cost $h = 2$, the optimal compression stores $D = \text{abcdef}$ (6 symbols) and uses 3 pointers in C, achieving cost $6 + 3 + (2 - 1) \times 3 = 12$ vs. uncompressed cost 18.

This instance has a repeating pattern of length 6, allowing the dictionary D to store one copy and the compressed string C to reference it via pointers. Each pointer costs $h = 2$ (the pointer symbol itself plus $h - 1 = 1$ extra), so the total cost is $|D| + |C| + (h - 1) \times |\text{pointers}| = 6 + 3 + 1 \times 3 = 12$, saving 6 over the uncompressed cost of 18.

Rule 2.6: (Minimum External Macro Data Compression $\longrightarrow$ Integer Linear Programming) The compression problem decomposes into a dictionary selection (which symbols appear at which positions in D) and a string partitioning (which segments of s are literals vs. pointers). Both are naturally expressed with binary variables and linear constraints. The partition structure is modeled as a flow on a DAG whose nodes are string positions and whose arcs are candidate segments.
Overhead: `num_vars = string_length * alphabet_size + 2 * string_length + string_length^3`, `num_constraints = string_length + string_length * alphabet_size + string_length + string_length + 1 + string_length^3 * string_length`.

Proof: *Construction.* For alphabet Σ of size k, string s of length n, and pointer cost h:

Variables: (1) Binary $d_{j,c} \in \{0, 1\}$ for each dictionary position $j \in \{0, ..., n - 1\}$ and symbol $c \in \Sigma$: $d_{j,c} = 1$ iff $D[j] = c$. (2) Binary $u_j \in \{0, 1\}$: $u_j = 1$ iff dictionary position j is used. (3) Binary $\ell_i \in \{0, 1\}$ for each string position i: $\ell_i = 1$ iff position i is covered by a literal. (4) Binary $p_{i,\lambda,\delta} \in \{0, 1\}$ for each valid triple (i, λ, δ) with $i + \lambda \leq n$ and $\delta + \lambda \leq n$: $p_{i,\lambda,\delta} = 1$ iff positions $[i, i + \lambda)$ are covered by a pointer referencing $D[\delta..\delta + \lambda)$.

Constraints: (1) Dictionary one-hot: $\sum_{c \in \Sigma} d_{j,c} \leq 1$ for all j. (2) Linking: $d_{j,c} \leq u_j$ for all j, c. (3) Contiguity: $u_{j+1} \leq u_j$ for all $j < n - 1$. (4) Partition flow: the segments form a partition of $\{0, ..., n - 1\}$ via flow conservation on nodes $0, ..., n$. (5) Pointer matching: $p_{i,\lambda,\delta} \leq d_{\delta+r, s[i+r]}$ for all offsets $r \in \{0, ..., \lambda - 1\}$.

Objective: Minimize $\sum_j u_j + \sum_i \ell_i + h \sum_{i,\lambda,\delta} p_{i,\lambda,\delta}$.

Correctness. ($\Rightarrow$) An optimal (D, C) pair determines a feasible ILP assignment: set $d_{j,c} = 1$ for each symbol in D, $u_j = 1$ for used positions, and activate the corresponding literal or pointer variables for each C-slot. The partition flow is satisfied by construction. ($\Leftarrow$) Any feasible ILP solution defines a valid dictionary (one-hot + contiguity) and a valid partition of s into literal and pointer segments (flow conservation + matching), with cost equal to the objective.

Solution extraction. Read D from the $d_{j,c}$ indicators. Walk through the active segments (via ℓ_i and $p_{i,\lambda,\delta}$) to reconstruct C. □

Definition 2.140 (Minimum Internal Macro Data Compression): Given a finite alphabet Σ of size k, a string $s \in \Sigma^*$ of length n, and a pointer cost $h \in \mathbb{Z}^+$, find a single compressed string $C \in (\Sigma \cup \{\text{pointers}\})^*$ such that s can be obtained from C by resolving all pointer references *within* C *itself* (left-to-right), minimizing the total cost $|C| + (h - 1) \times$ (number of pointer occurrences in C).

- Complexity: `(alphabet_size + string_len + 1) ^ string_len`.
- Reduces to: ILP.

MinimumInternalMacroDataCompression	
alphabet_size	Size of the alphabet (symbols indexed 0..alphabet_size)
string	Source string as symbol indices
pointer_cost	Pointer cost h (each pointer adds h−1 extra to the cost)

A classical NP-hard data compression problem, listed as SR23 in Garey and Johnson [5]. Unlike the external variant (Definition 2.139), there is no separate dictionary — the compressed string C serves as both dictionary and output, with pointers referencing substrings within C itself. J. A. Storer [106] proved NP-completeness via transformation from Vertex Cover. J. A. Storer and T. G. Szymanski [107] showed that NP-completeness persists even when h is any fixed integer ≥ 2. The internal macro model is closely related to the smallest grammar problem, which is APX-hard [108].[41]

Example. Let $\Sigma = \{\text{a, b, c}\}$ and $s = \text{abcabcabc}$ (length 9) with pointer cost $h = 2$.

```
$ pred create --example MinimumInternalMacroDataCompression -o min-imdc.json
$ pred solve min-imdc.json
$ pred evaluate min-imdc.json --config 0,1,2,4,4,3,3,3,3
```

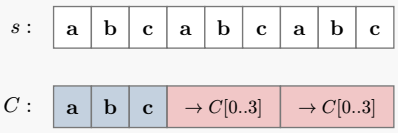


Figure 68: Minimum Internal Macro Data Compression: with $s = \text{abcabcabc}$ (length 9) and pointer cost $h = 2$, the optimal self-referencing compression C starts with 3 literals, then uses 2 pointers back to $C[0..3]$, achieving cost $5 + (2-1) \times 2 = 7$ vs. uncompressed cost 9.

The compressed string C has 3 literal symbols followed by 2 pointers, each referencing $C[0..3]$ to copy "abc". Each pointer costs $h = 2$ (the pointer token plus $h - 1 = 1$ extra), so the total cost is $|C| + (h - 1) \times |\text{pointers}| = 5 + 1 \times 2 = 7$, saving 2 over the uncompressed cost of 9.

Rule 2.7: (Minimum Internal Macro Data Compression → Integer Linear Programming) The self-referencing compression problem is formulated as a binary ILP. Since there is no separate dictionary, only the string partition structure needs to be modeled. The partition is expressed as a flow on a DAG whose nodes are string positions and whose arcs are candidate segments.
Overhead: `num_vars = string_len + string_len^3`, `num_constraints = string_len + 1`.

Proof: *Construction.* For alphabet Σ of size k, string s of length n, and pointer cost h:

Variables: (1) Binary $\ell_i \in \{0, 1\}$ for each string position $i \in \{0, ..., n-1\}$: $\ell_i = 1$ iff position i is covered by a literal. (2) Binary $p_{i,\lambda,r} \in \{0, 1\}$ for each valid triple (i, λ, r) where $r + \lambda \leq i$ and $s[r..r+\lambda) = s[i..i+\lambda)$: $p_{i,\lambda,r} = 1$ iff positions $[i, i+\lambda)$ are covered by a pointer referencing the decoded output starting at source position r.

Constraints: Partition flow: the segments form a partition of $\{0, ..., n-1\}$ via flow conservation on nodes $0, ..., n$. The string-matching constraint ($s[r..r+\lambda) = s[i..i+\lambda)$) and the precedence constraint ($r + \lambda \leq i$) are enforced structurally by only generating valid triples.

Objective: Minimize $\sum_i \ell_i + h \sum_{i,\lambda,r} p_{i,\lambda,r}$.

Correctness. ($\Rightarrow$) An optimal compressed string C determines a feasible ILP assignment: activate the literal or pointer variable for each segment in the partition. The flow is satisfied by construction. ($\Leftarrow$) Any feasible ILP solution defines a valid partition of s into literal and pointer segments with cost equal to the objective.

[41] No algorithm improving on brute-force enumeration is known for optimal internal macro compression.

Solution extraction. Walk through the active segments (via ℓ_i and $p_{i,\lambda,r}$) to reconstruct C, mapping source reference positions to compressed-string positions. □

Definition 2.141 (Minimum Weight AND/OR Graph): Given a directed acyclic graph $G = (V, A)$ with $n = |V|$ vertices, a source vertex $s \in V$, a gate-type function $g : V \to \{\text{AND}, \text{OR}, \text{leaf}\}$, and arc weights $w : A \to \mathbb{Z}$, find a solution subgraph $S \subseteq A$ of minimum total weight $\sum_{a \in S} w(a)$. A solution subgraph is valid when: (1) the source is solved, (2) for each solved AND-gate vertex v, all outgoing arcs from v are in S, (3) for each solved OR-gate vertex v, at least one outgoing arc from v is in S, and (4) leaf vertices are trivially solved. A vertex $v \neq s$ is solved if there exists an arc $(u, v) \in S$ with u solved.

- Complexity: `2^num_arcs`.
- Reduces from: MinimumVertexCover.

MinimumWeightAndOrGraph	
num_vertices	Number of vertices in the DAG
arcs	Directed arcs (u, v)
source	Source vertex index
gate_types	Gate type per vertex: Some(true)=AND, Some(false)=OR, None=leaf
arc_weights	Weight of each arc

AND/OR graphs generalize search trees and game trees and arise in AI planning, logic programming, and design-space exploration. Dynamic-programming algorithms on tree-structured AND/OR graphs run in linear time, but the general DAG case requires exponential enumeration.[42]

Example. Consider $n = 7$ vertices with source v_0 (AND gate). Vertices v_1, v_2 are OR gates; v_3, v_4, v_5, v_6 are leaves. Arcs with weights: $v_0 \to v_1(1)$, $v_0 \to v_2(2)$, $v_1 \to v_3(3)$, $v_1 \to v_4(1)$, $v_2 \to v_5(4)$, $v_2 \to v_6(2)$. Since v_0 is AND, both outgoing arcs must be selected (cost $1 + 2 = 3$). For OR gates, pick the cheapest outgoing arc: $v_1 \to v_4$ (cost 1) and $v_2 \to v_6$ (cost 2). Total weight: 6.

```
$ pred create --example MinimumWeightAndOrGraph -o mwaog.json
$ pred solve mwaog.json --solver brute-force
$ pred evaluate mwaog.json --config 1,1,0,1,0,1
```

Definition 2.142 (Minimum Fault Detection Test Set): Given a directed acyclic graph $G = (V, A)$ with $n = |V|$ vertices, designated input vertices $I \subseteq V$, and designated output vertices $O \subseteq V$, find the minimum number of input-output pairs $(i, o) \in I \times O$ such that the union of their coverage sets covers every internal vertex in $V - (I \cup O)$. For a pair (i, o), the coverage set is the set of vertices reachable from i that can also reach o.

- Complexity: `2^(num_inputs * num_outputs)`.
- Reduces to: ILP.
- Reduces from: ExactCoverBy3Sets.

MinimumFaultDetectionTestSet	
num_vertices	Number of vertices in the DAG
arcs	Directed arcs (u, v)
inputs	Input vertex indices
outputs	Output vertex indices

Fault detection test sets arise in hardware testing: each input-output path through a circuit's DAG representation exercises the internal components it traverses, while the boundary pins themselves are fixed sources and sinks. The problem therefore asks for the fewest test pairs whose induced paths cover all internal vertices. It generalises Set Cover over a structured family of subsets induced by DAG reachability.[43]

[42] No algorithm improving on brute-force enumeration of all $2^{|A|}$ arc subsets is known for general AND/OR DAGs.

[43] No algorithm improving on brute-force enumeration of all $2^{|I| \cdot |O|}$ input-output pair subsets is known for the general case.

Example. Consider $n = 7$ vertices with inputs $I = \{0, 1\}$ and outputs $O = \{5, 6\}$. The internal vertices are $\{2, 3, 4\}$. Arcs: $0 \to 2$, $0 \to 3$, $1 \to 3$, $1 \to 4$, $2 \to 5$, $3 \to 5$, $3 \to 6$, $4 \to 6$. Selecting pair $(0, 5)$ covers internal vertices $\{2, 3\}$, and pair $(1, 6)$ covers internal vertices $\{3, 4\}$. Their union is all internal vertices, giving an optimal count of 2.

```
$ pred create --example MinimumFaultDetectionTestSet -o mfdts.json
$ pred solve mfdts.json --solver brute-force
$ pred evaluate mfdts.json --config 1,0,0,1
```

Definition 2.143 (Minimum Feedback Arc Set): Given a directed graph $G = (V, A)$, find a minimum-size subset $A' \subseteq A$ such that $G - A'$ is a directed acyclic graph (DAG). Equivalently, A' must contain at least one arc from every directed cycle in G.

- Complexity: `2^num_vertices`.
- Reduces to: ILP, MaximumLikelihoodRanking.
- Reduces from: MinimumVertexCover.

MinimumFeedbackArcSet	
graph	The directed graph G=(V,A)
weights	Arc weights w: A -> R

Feedback Arc Set (FAS) is a classical NP-complete problem from Karp's original list [1] (via transformation from Vertex Cover, as presented in Garey & Johnson GT8). The problem arises in ranking aggregation, sports scheduling, deadlock avoidance, and causal inference. Unlike the undirected analogue (which is trivially polynomial — the number of non-tree edges in a spanning forest), the directed version is NP-hard due to the richer structure of directed cycles. The best known exact algorithm uses dynamic programming over vertex subsets in $O^*(2^n)$ time, generalizing the Held–Karp TSP technique to vertex ordering problems [109]. FAS is fixed-parameter tractable with parameter $k = |A'|$: an $O\big(4^k \cdot k! \cdot n^{O(1)}\big)$ algorithm exists via iterative compression [110]. Polynomial-time solvable for planar digraphs via the Lucchesi–Younger theorem [111].

Example. Consider G with $V = \{0, 1, 2\}$ and arcs $(0 \to 1)$,$(1 \to 2)$,$(2 \to 0)$. Removing $A' = \{(2 \to 0)\}$ (weight 1) breaks all directed cycles, yielding a DAG.

```
$ pred create --example MinimumFeedbackArcSet -o minimum-feedback-arc-set.json
$ pred solve minimum-feedback-arc-set.json
$ pred evaluate minimum-feedback-arc-set.json --config 0,0,1
```

Definition 2.144 (Partial Feedback Edge Set): Given an undirected graph $G = (V, E)$, a budget $K \in \mathbb{Z}_{\geq 0}$, and a cycle-length bound $L \in \mathbb{Z}_{\geq 0}$, determine whether there exists a subset $E' \subseteq E$ with $|E'| \leq K$ such that every simple cycle in G of length at most L contains at least one edge of E'.

- Complexity: `2^num_edges`.

PartialFeedbackEdgeSet	
graph	The underlying graph G=(V,E)
budget	Maximum number K of edges that may be removed
max_cycle_length	Cycle length bound L; every cycle with length at most L must be hit

Partial Feedback Edge Set is the bounded-cycle edge-deletion problem GT9 in Garey and Johnson [5]. Bounding the cycle length is what makes the problem hard: hitting only the short cycles is NP-complete, whereas the unrestricted undirected feedback-edge-set problem is polynomial-time solvable by reducing

to a spanning forest. The implementation here uses one binary variable per edge, so brute-force search explores $O^*(2^{|E|})$ candidate edge subsets.[44]

Example. Consider the graph G with $n = 6$ vertices, $|E| = 9$ edges, budget $K = 3$, and length bound $L = 4$. Removing $E' = \{\{v_2, v_0\}, \{v_2, v_3\}, \{v_3, v_4\}\}$ hits the triangles (v_0, v_1, v_2), (v_0, v_2, v_3), (v_2, v_3, v_4), and (v_3, v_4, v_5), together with the 4-cycles (v_0, v_1, v_2, v_3), (v_0, v_2, v_4, v_3), and (v_2, v_3, v_5, v_4). Hence every cycle of length at most 4 is hit. Brute-force search on this instance finds exactly five satisfying 3-edge deletions and none of size 2, so the displayed configuration certifies a YES-instance.

```
$ pred create --example PartialFeedbackEdgeSet -o partial-feedback-edge-set.json
$ pred solve partial-feedback-edge-set.json
$ pred evaluate partial-feedback-edge-set.json --config 0,0,1,1,1,0,0,0,0
```

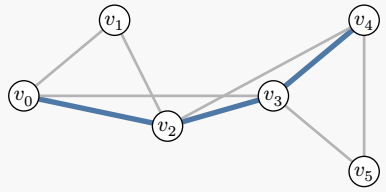


Figure 69: Partial Feedback Edge Set example with $K = 3$ and $L = 4$. Blue edges $\{v_0, v_2\}$, $\{v_2, v_3\}$, and $\{v_3, v_4\}$ form a satisfying edge set that hits every cycle of length at most 4.

Definition 2.145 (Multiple Choice Branching): Given a directed graph $G = (V, A)$, arc weights $w : A \to \mathbb{Z}^+$, a partition $A_1, A_2, ..., A_m$ of A, and a threshold $K \in \mathbb{Z}^+$, determine whether there exists a subset $A' \subseteq A$ with $\sum_{a \in A'} w(a) \geq K$ such that every vertex has in-degree at most one in (V, A'), the selected subgraph (V, A') is acyclic, and $|A' \cap A_i| \leq 1$ for every partition group.

- Complexity: `2^num_arcs`.

MultipleChoiceBranching	
graph	The directed graph G=(V,A)
weights	Arc weights w(a) for each arc a in A
partition	Partition of arc indices; each arc index must appear in exactly one group
threshold	Weight threshold K

Multiple Choice Branching is the directed-graph problem ND11 in Garey & Johnson [5]. The partition constraint turns the polynomial-time maximum branching setting into an NP-complete decision problem: Garey and Johnson note that the problem remains NP-complete even when the digraph is strongly connected and all weights are equal, while the special case in which every partition group has size 1 reduces to ordinary maximum branching and becomes polynomial-time solvable [5].

A conservative exact algorithm enumerates all $2^{\{|A|\}}$ arc subsets and checks the partition, in-degree, acyclicity, and threshold constraints in polynomial time. This is the brute-force search space used by the implementation.[45]

Example. Consider the digraph on $n = 6$ vertices with arcs $(0 \to 1), (0 \to 2), (1 \to 3), (2 \to 3), (1 \to 4), (3 \to 5), (4 \to 5), (2 \to 4)$, partition groups $A_1 = \{(0 \to 1), (0 \to 2)\}$, $A_2 = \{(1 \to 3), (2 \to 3)\}$, $A_3 = \{(1 \to 4), (2 \to 4)\}$, $A_4 = \{(3 \to 5), (4 \to 5)\}$, and threshold $K = 10$. The highlighted selection $A' = \{(0 \to 1), (1 \to 3), (2 \to 4), (3 \to 5)\}$ has total weight $3 + 4 + 3 + 3 = 13 \geq 10$, uses exactly one arc from each partition group, and gives in-degrees 1 at vertices $1, 3, 4$, and 5. Because every selected arc points strictly left-to-right in the drawing, the selected subgraph is acyclic. The figure highlights one satisfying selection for this instance.

[44] No sharper general exact worst-case bound is claimed here.
[45] We use the registry complexity bound $O^*(2^{\{|A|\}})$ for the full partitioned problem.

```
$ pred create --example MultipleChoiceBranching -o multiple-choice-branching.json
$ pred solve multiple-choice-branching.json
$ pred evaluate multiple-choice-branching.json --config 1,0,1,0,0,1,0,1
```

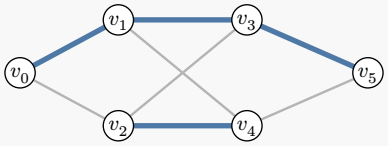


Figure 70: Directed graph for Multiple Choice Branching. Blue arcs show the satisfying branching $(0 \to 1), (1 \to 3), (2 \to 4), (3 \to 5)$ of total weight 13; gray arcs are available but unselected.

Definition 2.146 (Acyclic Partition): Given a directed graph $G = (V, A)$ with vertex weights $w : V \to \mathbb{Z}^+$, arc costs $c : A \to \mathbb{Z}^+$, and bounds $B, K \in \mathbb{Z}^+$, determine whether there exists a partition $V = V_1 \cup ... \cup V_m$ such that every part satisfies $\sum_{v \in V_i} w(v) \leq B$, the total cost of arcs crossing between different parts is at most K, and the quotient digraph on the parts is acyclic.

- Complexity: `num_vertices^num_vertices`.
- Reduces to: ILP.
- Reduces from: KSatisfiability.

AcyclicPartition	
graph	The directed graph G=(V,A)
vertex_weights	Vertex weights w(v) for each vertex v in V
arc_costs	Arc costs c(a) for each arc a in A, matching graph.arcs() order
weight_bound	Maximum total vertex weight B for each partition
cost_bound	Maximum total inter-partition arc cost K

Acyclic Partition is the directed partitioning problem ND15 in Garey & Johnson [5]. Unlike ordinary graph partitioning, the goal is not merely to minimize the cut: the partition must preserve a global topological order after every part is contracted to a super-node. This makes the model a natural abstraction for DAG-aware task clustering in compiler scheduling, parallel execution pipelines, and automatic differentiation systems where coarse-grained blocks must still communicate without creating cyclic dependencies.

The implementation uses the natural witness encoding in which each of the $n = 6$ vertices chooses one of at most n part labels, so direct brute-force search explores n^n assignments.[46]

Example. Consider the six-vertex digraph in the figure with vertex weights $w = (2, 3, 2, 1, 3, 1)$, part bound $B = 5$, and cut-cost bound $K = 5$. The witness $V_0 = \{v_0, v_2\}$, $V_1 = \{v_1\}$, $V_2 = \{v_3, v_4, v_5\}$ has part weights 4, 3, and 5, so every part respects the weight cap. Exactly 5 arcs cross between different parts, namely $(v_0 \to v_1)$,$(v_1 \to v_3)$,$(v_1 \to v_4)$,$(v_2 \to v_4)$,$(v_2 \to v_5)$, so the total crossing cost is $5 \leq K$. These crossings induce quotient arcs $V_0 \to V_1$, $V_0 \to V_2$, and $V_1 \to V_2$, which form a DAG; hence this instance is a YES-instance.

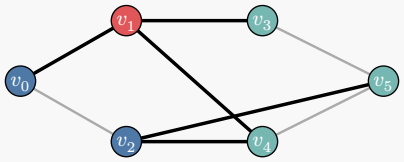


Figure 71: A YES witness for Acyclic Partition. Node colors indicate the parts V_0, V_1, and V_2. Black arcs cross parts and define the quotient DAG $V_0 \to V_1$, $V_0 \to V_2$, $V_1 \to V_2$; gray arcs stay inside a part and therefore do not contribute to the quotient graph.

[46] Many labelings represent the same unordered partition, but the full configuration space exposed to the solver is still n^n.

Definition 2.147 (Flow Shop Scheduling): Given m processors and a set J of n jobs, where each job $j \in J$ consists of m tasks $t_1[j], t_2[j], ..., t_m[j]$ with lengths $\ell(t_i[j]) \in \mathbb{Z}_0^+$, and a deadline $D \in \mathbb{Z}^+$, determine whether there exists a permutation schedule π of the jobs such that all jobs complete by time D. Each job must be processed on machines $1, 2, ..., m$ in order, and job j cannot start on machine $i+1$ until its task on machine i is completed.

- Complexity: `factorial(num_jobs)`.
- Reduces to: ILP.
- Reduces from: ThreePartition.

```
FlowShopScheduling
num_processors      Number of machines m
task_lengths        task_lengths[j][i] = length of job j's task on machine i
deadline            Global deadline D
```

Flow Shop Scheduling is a classical NP-complete problem from Garey & Johnson (A5 SS15), strongly NP-hard for $m \geq 3$ [9]. For $m = 2$, it is solvable in $O(n \log n)$ by Johnson's rule [112]. The problem is fundamental in operations research, manufacturing planning, and VLSI design. When restricted to permutation schedules (same job order on all machines), the search space is $n!$ orderings. The best known exact algorithm for $m = 3$ runs in $O^*(3^n)$ time [113]; for general m, brute-force over $n!$ permutations gives $O(n! \cdot mn)$.

Example. Let $m = 3$ machines, $n = 5$ jobs with task lengths:

$$\ell = \begin{pmatrix} 3 & 4 & 2 \\ 2 & 3 & 5 \\ 4 & 1 & 3 \\ 1 & 5 & 4 \\ 3 & 2 & 3 \end{pmatrix}$$

and deadline $D = 25$. The job order $\pi = (j_4, j_1, j_5, j_3, j_2)$ yields makespan $23 \leq 25$, so a feasible schedule exists.

```
$ pred create --example FlowShopScheduling -o flow-shop-scheduling.json
$ pred solve flow-shop-scheduling.json
$ pred evaluate flow-shop-scheduling.json --config 3,0,2,1,0
```

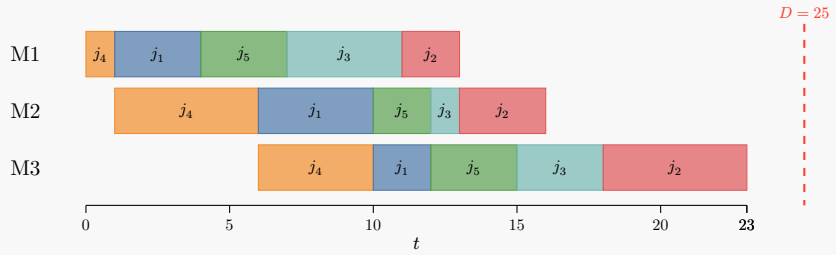


Figure 72: Flow shop schedule for 5 jobs on 3 machines. Job order $(j_4, j_1, j_5, j_3, j_2)$ achieves makespan 23, within deadline $D = 25$ (dashed red line).

Definition 2.148 (Job-Shop Scheduling): Given a positive integer m, a set J of jobs, where each job $j \in J$ consists of an ordered list of tasks $t_1[j], ..., t_{n_j}[j]$ with processor assignments $p\left(t_{k[j]}\right) \in \{1, ..., m\}$, processing lengths $\ell\left(t_{k[j]}\right) \in \mathbb{Z}_0^+$, and consecutive-processor constraint $p\left(t_{k[j]}\right) \neq p(t_{k+1}[j])$, find start times $\sigma\left(t_{k[j]}\right) \in \mathbb{Z}_0^+$ such that tasks sharing a processor do not overlap, each job respects $\sigma(t_{k+1}[j]) \geq \sigma\left(t_{k[j]}\right) + \ell\left(t_{k[j]}\right)$, and the makespan $\max_{j \in J}\left(\sigma\left(t_{n_j}[j]\right) + \ell\left(t_{n_j}[j]\right)\right)$ is minimized.

- Complexity: `factorial(num_tasks)`.
- Reduces from: ThreePartition.

JobShopScheduling	
num_processors	Number of processors m
jobs	jobs[j][k] = (processor, length) for the k-th task of job j

Job-Shop Scheduling is the classical disjunctive scheduling problem SS18 in Garey & Johnson; Garey, Johnson, and Sethi proved it strongly NP-hard already for two machines [9]. Unlike Flow Shop Scheduling, each job carries its own machine route, so the difficulty lies in choosing a compatible relative order on every machine and then finding the schedule with minimum makespan. This implementation follows the original Garey-Johnson formulation, including the requirement that consecutive tasks of the same job use different processors, and evaluates a witness by orienting the machine-order edges and propagating longest paths through the resulting precedence DAG. The registered baseline therefore exposes a factorial upper bound over task orders[47].

Example. The canonical fixture has 2 machines and 5 jobs

$$J_1 = ((M_1, 3), (M_2, 4)), J_2 = ((M_2, 2), (M_1, 3), (M_2, 2)), J_3 = ((M_1, 4), (M_2, 3)), J_4 = ((M_2, 5), (M_1, 2)), J_5 = ((M_1, 2), (M_2, 3), (M_1, 1)).$$

The witness stored in the example DB orders the six tasks on M_1 as $(J_1^1, J_2^2, J_3^1, J_4^2, J_5^1, J_5^3)$ and the six tasks on M_2 as $(J_2^1, J_4^1, J_1^2, J_3^2, J_5^2, J_2^3)$. Taking the earliest schedule consistent with those machine orders yields the Gantt chart in Figure 73, whose makespan is 19.

```
$ pred create --example JobShopScheduling -o job-shop-scheduling.json
$ pred solve job-shop-scheduling.json --solver brute-force
$ pred evaluate job-shop-scheduling.json --config 0,0,0,0,0,0,1,3,0,1,1,0
```

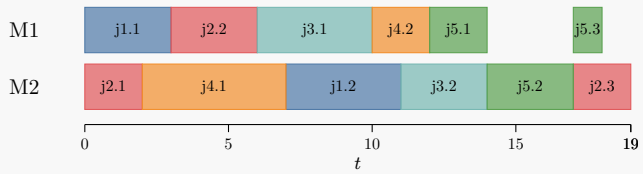


Figure 73: Job-shop schedule induced by the canonical machine-order witness. The optimal makespan is 19.

Definition 2.149 (Open Shop Scheduling): Given m machines and a set J of n jobs, where each job $j \in J$ has one task per machine i with processing time $p(j,i) \in \mathbb{Z}_0^+$, find a non-preemptive schedule minimizing the **makespan** $\max_{j,i}(\sigma(j,i) + p(j,i))$, subject to:

1. **Machine constraint:** Each machine processes at most one job at a time.
2. **Job constraint:** Each job occupies at most one machine at a time.

Unlike flow-shop or job-shop scheduling, there is no prescribed order for a job's tasks across machines.

- Complexity: `factorial(num_jobs)^num_machines`.
- Reduces to: ILP.
- Reduces from: Partition.

OpenShopScheduling	
num_machines	Number of machines m
processing_times	processing_times[j][i] = processing time of job j on machine i (n x m)

Open Shop Scheduling is problem SS14 in Garey and Johnson's catalog [5] (decision version: does a schedule exist with makespan $\leq D$?). NP-completeness for $m \geq 3$ machines was established by Gonzalez and Sahni via reduction from Partition [114]. The problem is solvable in polynomial time for $m = 2$ and also for the preemptive variant with any m [114]. This codebase evaluates a candidate schedule by simulating a greedy active schedule: for each step, the machine with the earliest feasible next-job start is processed next.

[47] The auto-generated complexity table records the concrete upper bound used by the Rust implementation; no sharper exact bound is cited here.

The configuration encodes one permutation of jobs per machine (direct indices), giving $(n!)^m$ candidate orderings.

Example. Let $m = 3$ machines and $n = 4$ jobs with processing times

$$P = \begin{pmatrix} 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 2 & 3 \\ 2 & 2 & 1 \end{pmatrix}$$

The canonical optimal orderings are:

Machine	Job order
M1	J_1,J_2,J_3,J_4
M2	J_2,J_1,J_4,J_3
M3	J_3,J_4,J_1,J_2

giving the Gantt chart in Figure 74 and makespan **8**.

```
$ pred create --example OpenShopScheduling -o open-shop-scheduling.json
$ pred solve open-shop-scheduling.json
$ pred evaluate open-shop-scheduling.json --config 0,1,2,3,1,0,3,2,2,3,0,1
```

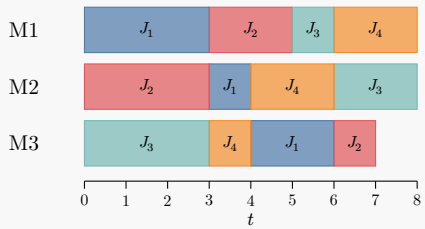


Figure 74: Open-shop schedule for 4 jobs on 3 machines. Optimal makespan is 8. Each color represents one job; no two tasks of the same job overlap in time.

Definition 2.150 (Staff Scheduling): Given a collection C of binary schedule patterns of length m, where each pattern has exactly k ones, a requirement vector $\overline{R} \in \mathbb{Z}^m_{\geq 0}$, and a worker budget $n \in \mathbb{Z}_{\geq 0}$, determine whether there exists a function $f : C \to \mathbb{Z}_{\geq 0}$ such that $\sum_{c \in C} f(c) \leq n$ and $\sum_{c \in C} f(c) \cdot c \geq \overline{R}$ component-wise.

- Complexity: `(num_workers + 1)^num_schedules`.
- Reduces from: ExactCoverBy3Sets.

```
StaffScheduling
shifts_per_schedule Required number of active periods in each schedule pattern
schedules           Binary schedule patterns available to workers
requirements        Minimum staffing requirement for each period
num_workers         Maximum number of workers available
```

Staff Scheduling is problem SS20 in Garey and Johnson's catalog [5]. It models workforce planning with reusable shift templates: each pattern describes the periods covered by one worker, and the multiplicity function f chooses how many workers receive each template. The general problem is NP-complete [5], while the circular-ones special case admits a polynomial-time algorithm via network-flow structure [115]. In this codebase the registered baseline enumerates all assignments of $0, \ldots, n$ workers to each pattern, matching the $(n+1)^{|C|}$ configuration space exposed by the model.

Example. Consider a 7-day week with $k = 5$ working days per schedule, worker budget $n = 4$, and schedule patterns

$c_1 = (1,1,1,1,1,0,0), c_2 = (0,1,1,1,1,1,0), c_3 = (0,0,1,1,1,1,1), c_4 = (1,0,0,1,1,1,1), c_5 = (1,1,0,0,1,1,1)$

with requirement vector $\overline{R} = (2,2,2,3,3,2,1)$. Choosing

$$f(c_1) = f(c_2) = f(c_3) = f(c_4) = 1$$

and

$$f(c_5) = 0$$

uses exactly 4 workers and yields coverage vector $(2,2,3,4,4,3,2) \geq \overline{R}$, so the instance is feasible.

Schedule	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Workers
c_1	1	1	1	1	1	0	0	1
c_2	0	1	1	1	1	1	0	1
c_3	0	0	1	1	1	1	1	1
c_4	1	0	0	1	1	1	1	1
c_5	1	1	0	0	1	1	1	0
$\overline{R}$	2	2	2	3	3	2	1	-
Coverage	2	2	3	4	4	3	2	4

Table 2: Worked Staff Scheduling instance. The last column shows the chosen multiplicities $f(c_i)$; the final row verifies that daily coverage dominates the requirement vector while using 4 workers.

Definition 2.151 (Timetable Design): Given a set H of work periods, a set C of craftsmen, a set T of tasks, availability sets $A_{C(c)} \subseteq H$ for each craftsman $c \in C$, availability sets $A_{T(t)} \subseteq H$ for each task $t \in T$, and exact workload requirements $R : C \times T \to \mathbb{Z}_{\geq 0}$, determine whether there exists a function $f : C \times T \times H \to \{0,1\}$ such that:

$$f(c,t,h) = 1 \Rightarrow h \in A_{C(c)} \cap A_{T(t)},$$

$$\forall c \in C, h \in H : \sum_{t \in T} f(c,t,h) \leq 1,$$

$$\forall t \in T, h \in H : \sum_{c \in C} f(c,t,h) \leq 1,$$

and

$$\forall c \in C, t \in T : \sum_{h \in H} f(c,t,h) = R(c,t).$$

- Complexity: `2^(num_craftsmen * num_tasks * num_periods)`.
- Reduces to: ILP.
- Reduces from: KSatisfiability.

```
TimetableDesign
num_periods     Number of work periods |H|
num_craftsmen   Number of craftsmen |C|
num_tasks       Number of tasks |T|
craftsman_avail Availability matrix A(c) for craftsmen (|C| x |H|)
task_avail      Availability matrix A(t) for tasks (|T| x |H|)
requirements    Required work periods R(c,t) for each craftsman-task pair (|C| x |T|)
```

Timetable Design is the classical timetabling feasibility problem catalogued as SS19 in Garey & Johnson [5]. Even, Itai, and Shamir showed that it is NP-complete even when there are only three work periods,

every task is available in every period, and every requirement is binary [21]. The same paper also identifies polynomial-time islands, including cases where each craftsman is available in at most two periods or where all craftsmen and tasks are available in every period [21]. The implementation in this repository uses one binary variable for each triple (c, t, h), so the registered baseline explores a configuration space of size $2^{|C|\,|T|\,|H|}$.

Example. The canonical instance has three periods $H = \{h_1, h_2, h_3\}$, five craftsmen, five tasks, and seven nonzero workload requirements. The satisfying timetable stored in the example database assigns (c_1, t_1), (c_2, t_5) during h_1, (c_2, t_2), (c_3, t_4) during h_2, and (c_1, t_3), (c_4, t_5), (c_5, t_2) during h_3. Every listed assignment lies in the corresponding availability intersection $A_{C(c)} \cap A_{T(t)}$, no craftsman or task appears twice in the same period, and each required pair is scheduled exactly once, so the verifier returns YES.

```
$ pred create --example TimetableDesign -o timetable-design.json
$ pred solve timetable-design.json
$ pred evaluate timetable-design.json --config 1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0
```

Period	Assignments
h_1	(c_1, t_1), (c_2, t_5)
h_2	(c_2, t_2), (c_3, t_4)
h_3	(c_1, t_3), (c_4, t_5), (c_5, t_2)

Table 3: Worked Timetable Design instance derived from the canonical example DB. Each row lists the craftsman-task pairs assigned in one work period.

Definition 2.152 (Multiprocessor Scheduling): Given a finite set T of tasks with processing lengths $\ell : T \to \mathbb{Z}^+$, a number $m \in \mathbb{Z}^+$ of identical processors, and a deadline $D \in \mathbb{Z}^+$, determine whether there exists an assignment $p : T \to \{1, ..., m\}$ such that for every processor $i \in \{1, ..., m\}$ we have $\sum_{t \in T : p(t) = i} \ell(t) \leq D$.

- Complexity: `2^num_tasks`.
- Reduces to: ILP.
- Reduces from: Partition.

MultiprocessorScheduling	
lengths	Processing time l(t) for each task
num_processors	Number of identical processors m
deadline	Global deadline D

Multiprocessor Scheduling is problem SS8 in Garey & Johnson [5]. Their original formulation uses start times on identical processors, but because tasks are independent and non-preemptive, any feasible schedule can be packed contiguously on each processor. The model implemented here therefore uses processor-assignment variables, and feasibility reduces to checking that every processor's total load is at most D. For fixed m, dynamic programming over load vectors gives pseudo-polynomial algorithms; for general m, the best known exact algorithm runs in $O^*(2^n)$ time via inclusion-exclusion over set partitions [14].

Example. Let $T = \{t_1, ..., t_5\}$ with lengths $(4, 5, 3, 2, 6)$, $m = 2$, and $D = 10$. The satisfying assignment $(1, 2, 2, 2, 1)$ places t_1 and t_5 on processor 1 and t_2, t_3, t_4 on processor 2. The verifier computes the processor loads $4 + 6 = 10$ and $5 + 3 + 2 = 10$, so both meet the deadline exactly.

```
$ pred create --example MultiprocessorScheduling -o multiprocessor-scheduling.json
$ pred solve multiprocessor-scheduling.json
$ pred evaluate multiprocessor-scheduling.json --config 0,1,1,1,0
```

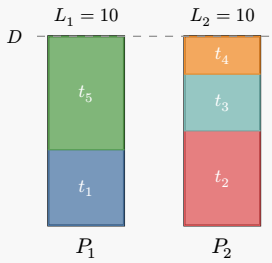


Figure 75: Canonical Multiprocessor Scheduling instance with 5 tasks on 2 processors. Stacked blocks show the satisfying assignment $(1, 2, 2, 2, 1)$; both processor loads equal the deadline $D = 10$.

Definition 2.153 (Production Planning): Given a positive integer n, period demands $r_1, ..., r_n \in \mathbb{Z}_{\geq 0}$, production capacities $c_1, ..., c_n \in \mathbb{Z}_{\geq 0}$, setup costs $b_1, ..., b_n \in \mathbb{Z}_{\geq 0}$, per-unit production costs $p_1, ..., p_n \in \mathbb{Z}_{\geq 0}$, per-unit inventory costs $h_1, ..., h_n \in \mathbb{Z}_{\geq 0}$, and a bound $B \in \mathbb{Z}_{\geq 0}$, determine whether there exist production quantities $x_1, ..., x_n$ such that $0 \leq x_i \leq c_i$ for every period i, the inventory prefix $I_i = \sum_{j=1}^{i} (x_j - r_j)$ satisfies $I_i \geq 0$ for every i, and $\sum_{i=1}^{n} (p_i x_i + h_i I_i) + \sum_{i:x_i>0} b_i \leq B$.

- Complexity: `(max_capacity + 1)^num_periods`.
- Reduces from: Partition.

ProductionPlanning	
num_periods	Number of planning periods n
demands	Demand r_i for each period
capacities	Production capacity c_i for each period
setup_costs	Setup cost b_i incurred when x_i > 0
production_costs	Per-unit production cost coefficient p_i
inventory_costs	Per-unit inventory cost coefficient h_i
cost_bound	Total cost bound B

Production Planning is the lot-sizing feasibility problem SS21 in Garey & Johnson [5]. Florian, Lenstra, and Rinnooy Kan show that the general problem is NP-complete even under strong restrictions, while also giving pseudo-polynomial dynamic-programming algorithms for capacitated variants [116]. The implementation in this repository uses one bounded integer variable per period, so the registered exact baseline explores the direct witness space $\prod_i (c_i + 1)$; under the uniform-capacity bound $C = \max_i c_i$, this becomes $O^*((C+1)^n)$[48].

Example. Consider the canonical instance with 4 periods, demands (2, 1, 3, 2), capacities (4, 4, 4, 4), setup costs (2, 2, 2, 2), production costs (1, 1, 1, 1), inventory costs (1, 1, 1, 1), and budget $B = 16$. The satisfying production plan $x = (3, 0, 4, 1)$ yields prefix inventories (1, 0, 1, 0). The verifier therefore accepts, and its cost breakdown is $8 + 2 + 6 = 16 \leq 16$.

```
$ pred create --example ProductionPlanning -o production-planning.json
$ pred solve production-planning.json --solver brute-force
$ pred evaluate production-planning.json --config 3,0,4,1
```

[48] This is the search bound induced by the configuration space exposed by the implementation, not a literature-best exact algorithm claim.

Period	1	2	3	4
r_i	2	1	3	2
c_i	4	4	4	4
b_i	2	2	2	2
p_i	1	1	1	1
h_i	1	1	1	1
x_i	3	0	4	1
I_i	1	0	1	0

Table 4: Canonical Production Planning instance from the example DB. The documented plan meets every prefix-demand constraint and stays within the budget $B = 16$.

Definition 2.154 (Capacity Assignment): Given a finite set C of communication links, an ordered set $M \subset \mathbb{Z}_{>0}$ of capacities, cost and delay functions $g : C \times M \to \mathbb{Z}_{\geq 0}$ and $d : C \times M \to \mathbb{Z}_{\geq 0}$ such that for every $c \in C$ and $i < j$ in the order of M we have $g(c, i) \leq g(c, j)$ and $d(c, i) \geq d(c, j)$, and a delay budget $J \in \mathbb{Z}_{\geq 0}$, find an assignment $\sigma : C \to M$ minimizing $\sum_{c \in C} g(c, \sigma(c))$ subject to $\sum_{c \in C} d(c, \sigma(c)) \leq J$.

- Complexity: `num_capacities ^ num_links`.
- Reduces to: ILP.
- Reduces from: SubsetSum.

```
CapacityAssignment
capacities          Ordered capacity levels M
cost                Cost matrix g(c, m) for each link and capacity
delay               Delay matrix d(c, m) for each link and capacity
delay_budget        Budget J on total delay penalty
```

Capacity Assignment is the bicriteria communication-network design problem SR7 in Garey & Johnson [5]. The original NP-completeness proof, via reduction from Subset Sum, is due to Van Sickle and Chandy [117]. The model captures discrete provisioning of communication links, where upgrading a link increases installation cost but decreases delay. The direct witness encoding implemented in this repository yields an $O^*\left(|M|^{|C|}\right)$ exact algorithm by brute-force enumeration[49]. Garey and Johnson also note a pseudo-polynomial dynamic-programming formulation when the budgets are small [5].

Example. Let $C = \{c_1, c_2, c_3\}$, $M = \{1, 2, 3\}$, and $J = 12$. With cost rows $(1, 3, 6)$, $(2, 4, 7)$, $(1, 2, 5)$ and delay rows $(8, 4, 1)$, $(7, 3, 1)$, $(6, 3, 1)$, the optimal assignment is $\sigma = (2, 2, 2)$ with total cost $3 + 4 + 2 = 9$ and total delay $4 + 3 + 3 = 10 \leq 12$. For contrast, $\sigma = (1, 1, 1)$ has total delay $8 + 7 + 6 = 21 > 12$ and is therefore infeasible.

```
$ pred create --example CapacityAssignment -o capacity-assignment.json
$ pred solve capacity-assignment.json --solver brute-force
$ pred evaluate capacity-assignment.json --config 1,1,1
```

Link	Cost row	Delay row
c_1	$(1, 3, 6)$	$(8, 4, 1)$
c_2	$(2, 4, 7)$	$(7, 3, 1)$
c_3	$(1, 2, 5)$	$(6, 3, 1)$

Table 5: Canonical Capacity Assignment instance with delay budget $J = 12$. Each row lists the cost-delay trade-off for one communication link.

[49] No algorithm improving on brute-force enumeration is known for the exact witness encoding used in this repository.

Definition 2.155 (Precedence Constrained Scheduling): Given a set T of n unit-length tasks, a partial order $\prec$ on T, a number $m \in \mathbb{Z}^+$ of processors, and a deadline $D \in \mathbb{Z}^+$, determine whether there exists a schedule `(0, 0, 1, 1, 1, 2, 2, 3)` $: T \to \{0, ..., D-1\}$ such that (i) for every time slot t, at most m tasks are assigned to t, and (ii) for every precedence $t_i \prec t_j$, we have `(0, 0, 1, 1, 1, 2, 2, 3)`$(t_j) \geq$ `(0, 0, 1, 1, 1, 2, 2, 3)`$(t_i) + 1$.

- Complexity: `2^num_tasks`.
- Reduces to: ILP.

PrecedenceConstrainedScheduling	
num_tasks	Number of tasks n = |T|
num_processors	Number of processors m
deadline	Global deadline D
precedences	Precedence pairs (i, j) meaning task i must finish before task j starts

Precedence Constrained Scheduling is problem SS9 in Garey & Johnson [5]. NP-complete via reduction from 3SAT [118]. Remains NP-complete even for $D = 3$ [119]. Solvable in polynomial time for $m = 2$ by the Coffman–Graham algorithm [120], for forest-structured precedences [121], and for chordal complement precedences [122]. A subset dynamic programming approach solves the general case in $O(2^n \cdot n)$ time by enumerating subsets of completed tasks at each time step.

Example. Let $n = 8$ tasks, $m = 3$ processors, $D = 4$. Precedences: $t_0 \prec t_2$, $t_0 \prec t_3$, $t_1 \prec t_3$, $t_1 \prec t_4$, $t_2 \prec t_5$, $t_3 \prec t_6$, $t_4 \prec t_6$, $t_5 \prec t_7$, $t_6 \prec t_7$. A feasible schedule assigns `(0, 0, 1, 1, 1, 2, 2, 3)` $=$ (0, 0, 1, 1, 1, 2, 2, 3): slot 0 has $\{t_0, t_1\}$, slot 1 has $\{t_2, t_3, t_4\}$, slot 2 has $\{t_5, t_6\}$, slot 3 has $\{t_7\}$. All precedences are satisfied and no slot exceeds $m = 3$.

```
$ pred create --example PrecedenceConstrainedScheduling -o precedence-constrained-scheduling.json
$ pred solve precedence-constrained-scheduling.json
$ pred evaluate precedence-constrained-scheduling.json --config 0,0,1,1,1,2,2,3
```

Definition 2.156 (Scheduling With Individual Deadlines): Given a set T of n unit-length tasks, a number $m \in \mathbb{Z}^+$ of identical processors, a deadline function $d : T \to \mathbb{Z}^+$, and a partial order $\preceq$ on T, determine whether there exists a schedule $\sigma : T \to \{0, 1, ..., D-1\}$, where $D = \max_{t \in T} d(t)$, such that every task meets its own deadline ($\sigma(t) + 1 \leq d(t)$), every precedence constraint is respected (if $t_i \preceq t_j$ then $\sigma(t_i) + 1 \leq \sigma(t_j)$), and at most m tasks are scheduled in each time slot.

- Complexity: `max_deadline^num_tasks`.
- Reduces to: ILP.

SchedulingWithIndividualDeadlines	
num_tasks	Number of tasks |T|
num_processors	Number of identical processors m
deadlines	Deadline d(t) for each task
precedences	Precedence pairs (predecessor, successor)

Scheduling With Individual Deadlines is the parallel-machine feasibility problem catalogued as A5 SS11 in Garey & Johnson [5]. Garey & Johnson record NP-completeness via reduction from Vertex Cover, and Brucker, Garey, and Johnson sharpen the complexity picture: the problem remains NP-complete for out-tree precedence constraints, but becomes polynomial-time solvable for in-trees [123]. The two-processor case is also polynomial-time solvable [5].

[50] This is the worst-case search bound induced by the implementation's configuration space; deadlines can be smaller on individual tasks, so practical instances may enumerate fewer than D^n assignments.

The direct encoding in this library uses one start-time variable per task, with each variable ranging over its allowable deadline window. If $D = \max_t d(t)$, exhaustive search over that encoding yields an $O^*(D^n)$ brute-force bound.[50]

Example. Consider $n = 7$ tasks on $m = 3$ processors with deadlines $d(t_1) = 2$, $d(t_2) = 1$, $d(t_3) = 2$, $d(t_4) = 2$, $d(t_5) = 3$, $d(t_6) = 3$, $d(t_7) = 2$ and precedence constraints $t_1 \preceq t_4$, $t_2 \preceq t_4$, $t_2 \preceq t_5$, $t_3 \preceq t_5$, $t_3 \preceq t_6$. The sample schedule $\sigma = [0, 0, 0, 1, 2, 1, 1]$ assigns slot 0: t_1, t_2, t_3; slot 1: t_4, t_6, t_7; slot 2: t_5. Every slot uses at most 3 processors, and the tight tasks t_2, t_4, t_5, t_7 finish exactly at their deadlines.

```
$ pred create --example SchedulingWithIndividualDeadlines -o scheduling-with-individual-deadlines.json
$ pred solve scheduling-with-individual-deadlines.json
$ pred evaluate scheduling-with-individual-deadlines.json --config 0,0,0,1,2,1,1
```

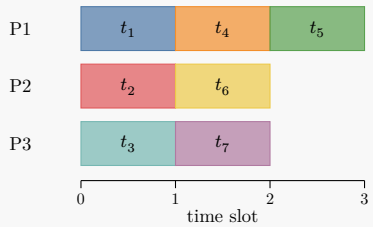


Figure 76: A feasible 3-processor schedule for Scheduling With Individual Deadlines. Tasks sharing a column run in the same unit-length time slot; the sample assignment uses slots $0, 1, 2$ and meets every deadline.

Definition 2.157 (Preemptive Scheduling): Given a set T of n tasks with processing lengths $\ell : T \to \mathbb{Z}^+$, a number $m \in \mathbb{Z}^+$ of identical processors, and a set of precedence constraints $\prec$ on T, find a preemptive schedule that minimizes the makespan.

A preemptive schedule assigns each task t a (possibly non-contiguous) set $S(t) \subseteq \{0, 1, ..., D_{\max} - 1\}$ of unit time slots, where $D_{\max} = \sum_t \ell(t)$, such that $|S(t)| = \ell(t)$ for all t, at most m tasks are active at each slot, and for every precedence $(t_i \prec t_j)$, the last slot of t_i precedes the first slot of t_j.

The makespan is $\max_{\{t \in T\}}(\max S(t) + 1)$.

- Complexity: `2^(num_tasks * num_tasks)`.
- Reduces to: ILP.
- Reduces from: KSatisfiability.

PreemptiveScheduling	
lengths	Processing length l(t) for each task
num_processors	Number of identical processors m
precedences	Precedence pairs (pred, succ) — pred must finish before succ starts

Preemptive Scheduling is problem A5 SS6 in Garey & Johnson [5]. NP-complete in general; the special case without precedences (m arbitrary) is solvable in polynomial time (McNaughton's wrap-around algorithm), and the preemptive open-shop variant is also polynomial. The configuration representation is a binary vector of length $n \cdot D_{\max}$ encoding per-slot assignments.

Example. Let $n = 5$ tasks with lengths (2, 1, 3, 2, 1), $m = 2$ processors, and precedences $t_0 \prec t_2$, $t_1 \prec t_3$. Optimal makespan: 5. Schedule: t_0 at slots [0, 1]; t_1 at slots [0]; t_2 at slots [2, 3, 4]; t_3 at slots [2, 3]; t_4 at slots [1].

```
$ pred create --example PreemptiveScheduling -o preemptive-scheduling.json
$ pred solve preemptive-scheduling.json
```

```
$           pred           evaluate           preemptive-scheduling.json           --config
1,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0
```

Definition 2.158 (Scheduling to Minimize Weighted Completion Time): Given a finite set T of tasks with processing lengths $\ell : T \to \mathbb{Z}^+$ and weights $w : T \to \mathbb{Z}^+$, and a number $m \in \mathbb{Z}^+$ of identical processors, find an assignment $p : T \to \{1, ..., m\}$ that minimizes the total weighted completion time $\sum_{t \in T} w(t) \cdot C(t)$, where on each processor tasks are ordered by Smith's rule (non-decreasing $\ell(t)/w(t)$ ratio) and $C(t)$ is the completion time of task t (i.e., the cumulative processing time up to and including t on its assigned processor).

- Complexity: `num_processors^num_tasks`.
- Reduces to: ILP.

SchedulingToMinimizeWeightedCompletionTime	
lengths	Processing time l(t) for each task
weights	Weight w(t) for each task
num_processors	Number of identical processors m

Scheduling to Minimize Weighted Completion Time is problem A5 SS13 in Garey & Johnson [5]. NP-complete for $m = 2$ by reduction from Partition [124], and NP-complete in the strong sense for arbitrary m. For a fixed assignment of tasks to processors, Smith's rule gives the optimal ordering on each processor, reducing the search space to m^n processor assignments [125]. The problem is solvable in polynomial time when all lengths are equal or when all weights are equal [126], [127].

Example. Let $T = \{t_1, ..., t_5\}$ with lengths (1, 2, 3, 4, 5), weights (6, 4, 3, 2, 1), and $m = 2$ processors. The optimal assignment (1, 2, 1, 2, 1) achieves total weighted completion time 47:

- Processor 1: $\{t_1, t_3, t_5\}$ – contributions: $1 \times 6 = 6$, $4 \times 3 = 12$, $9 \times 1 = 9$
- Processor 2: $\{t_2, t_4\}$ – contributions: $2 \times 4 = 8$, $6 \times 2 = 12$

```
$ pred create --example SchedulingToMinimizeWeightedCompletionTime -o scheduling-wct.json
$ pred solve scheduling-wct.json --solver brute-force
$ pred evaluate scheduling-wct.json --config 0,1,0,1,0
```

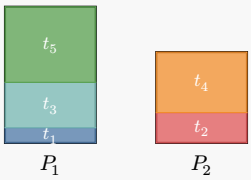


Figure 77: Canonical Scheduling to Minimize Weighted Completion Time instance with 5 tasks on 2 processors. Tasks are ordered on each processor by Smith's rule.

Rule 2.8: (Scheduling to Minimize Weighted Completion Time → Integer Linear Programming) This $O(n^2 m)$ reduction constructs an ILP with binary assignment variables $x_{t,p}$, integer completion-time variables C_t, and binary ordering variables $y_{i,j}$ for task pairs. Big-M disjunctive constraints enforce non-overlapping execution on shared processors.

Overhead: `num_vars = num_tasks * num_processors + num_tasks + num_tasks * (num_tasks + -1 * 1) * 2^-1`, `num_constraints = num_tasks + num_tasks * num_processors + 2 * num_tasks + 2 * num_tasks * (num_tasks + -1 * 1) * 2^-1 * num_processors + num_tasks * (num_tasks + -1 * 1) * 2^-1`.

Proof: *Construction.* Let $n = |T|$ and m be the number of processors. Create nm binary assignment variables $x_{t,p} \in \{0, 1\}$ (task t on processor p), n integer completion-time variables C_t, and $\frac{n(n-1)}{2}$ binary ordering variables $y_{i,j}$ for $i < j$. The constraints are: (1) Assignment: $\sum_p x_{t,p} = 1$ for each t. (2) Completion

bounds: $C_t \geq \ell(t)$ for each t. (3) Disjunctive: for each pair (i, j) with $i < j$ and each processor p, big-M constraints ensure that if both tasks are on processor p, one must complete before the other starts. The objective minimizes $\sum_t w(t) \cdot C_t$.

Correctness. ($\Rightarrow$) Any valid schedule gives a feasible ILP solution with the same objective. ($\Leftarrow$) Any ILP solution encodes a valid assignment and non-overlapping schedule.

Solution extraction. For each task t, find the processor p with $x_{t,p} = 1$. □

Definition 2.159 (Sequencing Within Intervals): Given a finite set T of tasks and, for each $t \in T$, a release time $r(t) \geq 0$, a deadline $d(t) \geq 0$, and a processing length $\ell(t) \in \mathbb{Z}^+$ satisfying $r(t) + \ell(t) \leq d(t)$, determine whether there exists a feasible schedule $\sigma : T \to \mathbb{Z}_{\geq 0}$ such that for each $t \in T$: (1) $\sigma(t) \geq r(t)$, (2) $\sigma(t) + \ell(t) \leq d(t)$, and (3) for all $t' \in T \setminus \{t\}$, either $\sigma(t') + \ell(t') \leq \sigma(t)$ or $\sigma(t') \geq \sigma(t) + \ell(t)$.

- Complexity: `2^num_tasks`.
- Reduces to: ILP.
- Reduces from: Partition.

SequencingWithinIntervals	
release_times	Release time r(t) for each task
deadlines	Deadline d(t) for each task
lengths	Processing length l(t) for each task

Sequencing Within Intervals is problem SS1 in Garey & Johnson [5], proved NP-complete via reduction from Partition (Theorem 3.8). Each task t must execute non-preemptively during the interval $[r(t), d(t))$, occupying $\ell(t)$ consecutive time units on a single machine, and no two tasks may overlap.

Example. Consider 5 tasks with overlapping availability windows:

Task	t_1	t_2	t_3	t_4	t_5
$r(t)$	0	1	3	6	0
$d(t)$	5	8	9	12	12
$\ell(t)$	2	2	2	3	2

```
$ pred create --example SequencingWithinIntervals -o sequencing-within-intervals.json
$ pred solve sequencing-within-intervals.json
$ pred evaluate sequencing-within-intervals.json --config 0,1,1,0,9
```

Each task can only start within its window $[r(t), d(t) - \ell(t)]$, and the windows overlap, so finding a non-overlapping assignment is non-trivial. One feasible schedule places the tasks at $[0, 2), [2, 4), [4, 6), [6, 9), [9, 11)$:

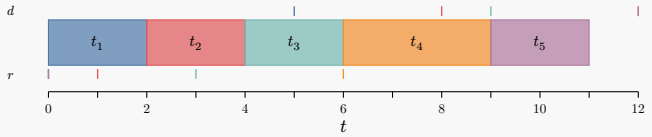


Figure 78: A feasible schedule for the SWI instance. Ticks below and above mark release times r and deadlines d for each task.

Definition 2.160 (Minimum Tardiness Sequencing): Given a set T of n unit-length tasks, a deadline function $d : T \to \mathbb{Z}^+$, and a partial order $\preceq$ on T, find a one-machine schedule $\sigma : T \to \{1, 2, ..., n\}$ that respects the precedence constraints (if $t_i \preceq t_j$ then $\sigma(t_i) < \sigma(t_j)$) and minimizes the number of *tardy* tasks, i.e., tasks t with $\sigma(t) > d(t)$.

- Complexity: `2^num_tasks`.
- Reduces to: ILP.

MinimumTardinessSequencing	
lengths	Processing time l(t) for each task
deadlines	Deadline d(t) for each task
precedences	Precedence pairs (predecessor, successor)

Minimum Tardiness Sequencing is a classical NP-complete scheduling problem catalogued as SS2 in Garey & Johnson [5]. In standard scheduling notation it is written $1 \mid \text{prec}, p_j = 1 \mid \sum U_j$, where $U_j = 1$ if job j finishes after its deadline and $U_j = 0$ otherwise.

The problem is NP-complete by reduction from Clique (Theorem 3.10 in [5]). When the precedence constraints are empty, the problem becomes solvable in $O(n \log n)$ time by Moore's algorithm [128]: sort tasks by deadline and greedily schedule each task on time, removing the task with the largest processing time whenever a deadline violation occurs. With arbitrary precedence constraints and unit processing times, the problem remains strongly NP-hard.

Example. Consider $n = 4$ tasks with deadlines $d = (2, 3, 1, 4)$ and precedence constraint $t_0 \preceq t_2$. An optimal schedule places tasks in order (t_0, t_1, t_2, t_3), giving 1 tardy task (t_2 finishes after its deadline).

```
$ pred create --example MinimumTardinessSequencing -o minimum-tardiness-sequencing.json
$ pred solve minimum-tardiness-sequencing.json
$ pred evaluate minimum-tardiness-sequencing.json --config 0,0,0,0
```

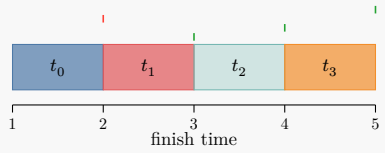


Figure 79: Optimal schedule for 4 tasks. Faded block indicates the tardy task (finish time exceeds deadline).

Definition 2.161 (Sequencing to Minimize Weighted Completion Time): Given a set T of n tasks, a processing-time function $l : T \to \mathbb{Z}^+$, a weight function $w : T \to \mathbb{Z}^+$, and a partial order $\preceq$ on T, find a one-machine schedule minimizing $\sum_{t \in T} w(t)C(t)$, where $C(t)$ is the completion time of task t and every precedence relation $t_i \preceq t_j$ requires task t_i to complete before task t_j starts.

- Complexity: `factorial(num_tasks)`.
- Reduces to: ILP.

SequencingToMinimizeWeightedCompletionTime	
lengths	Processing time l(t) for each task
weights	Weight w(t) for each task
precedences	Precedence pairs (predecessor, successor)

Sequencing to Minimize Weighted Completion Time is the single-machine precedence-constrained scheduling problem catalogued as SS4 in Garey & Johnson [5], usually written $1 \mid \text{prec} \mid \sum w_j C_j$. Lawler showed that arbitrary precedence constraints make the problem NP-complete, while series-parallel precedence orders admit an $O(n \log n)$ algorithm [129]. Without precedence constraints, Smith's ratio rule orders jobs by non-increasing $\frac{w_j}{l_j}$ and is optimal [125].

Example. Consider tasks with lengths $l = (2, 1, 3, 1, 2)$, weights $w = (3, 5, 1, 4, 2)$, and precedence constraints $t_0 \preceq t_2$, $t_1 \preceq t_4$. An optimal schedule is $(t_1, t_3, t_0, t_4, t_2)$, with completion times $(1, 2, 4, 6, 9)$ along the machine timeline and objective value 46.

```
$ pred create --example SequencingToMinimizeWeightedCompletionTime -o sequencing-to-minimize-
weighted-completion-time.json
```

```
$ pred solve sequencing-to-minimize-weighted-completion-time.json
$ pred evaluate sequencing-to-minimize-weighted-completion-time.json --config 1,2,0,1,0
```

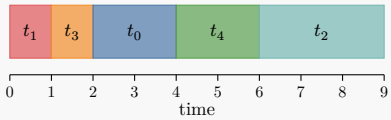


Figure 80: Optimal single-machine schedule for the canonical weighted-completion-time instance. Each block width equals the processing time l_j.

Definition 2.162 (Sequencing to Minimize Weighted Tardiness): Given a set J of n jobs, processing times $\ell_j \in \mathbb{Z}^+$, tardiness weights $w_j \in \mathbb{Z}^+$, deadlines $d_j \in \mathbb{Z}^+$, and a bound $K \in \mathbb{Z}^+$, determine whether there exists a one-machine schedule whose total weighted tardiness $\sum_{j \in J} w_j \max\bigl(0, C_j - d_j\bigr)$ is at most K, where C_j is the completion time of job j.

- Complexity: `factorial(num_tasks)`.
- Reduces to: ILP.
- Reduces from: ThreePartition.

SequencingToMinimizeWeightedTardiness	
lengths	Processing times l_j for each job
weights	Tardiness weights w_j for each job
deadlines	Deadlines d_j for each job
bound	Upper bound K on total weighted tardiness

Sequencing to Minimize Weighted Tardiness is the classical single-machine scheduling problem $1 \,\|\, \sum w_j T_j$, where $T_j = \max\bigl(0, C_j - d_j\bigr)$. It appears as SS5 in Garey & Johnson [5] and is strongly NP-complete via transformation from 3-Partition, which rules out pseudo-polynomial algorithms in general. When all weights are equal, the special case reduces to ordinary total tardiness and admits a pseudo-polynomial dynamic program [130]. Garey & Johnson also note that the equal-length case is polynomial-time solvable by bipartite matching [5].

Exact algorithms remain exponential in the worst case. Brute-force over all $n!$ schedules evaluates the implementation's decision encoding in $O(n! \cdot n)$ time. More refined exact methods include the branch-and-bound algorithm of Potts and Van Wassenhove [131] and the dynamic-programming style exact algorithm of Tanaka, Fujikuma, and Araki [132].

Example. Consider the five jobs with processing times $\ell = (3, 4, 2, 5, 3)$, weights $w = (2, 3, 1, 4, 2)$, deadlines $d = (5, 8, 4, 15, 10)$, and bound $K = 13$. The unique satisfying schedule is $(t_1, t_2, t_5, t_4, t_3)$, with completion times $(3, 7, 10, 15, 17)$. Only job t_3 is tardy; the per-job weighted tardiness contributions are $(0, 0, 0, 0, 13)$, so the total weighted tardiness is $13 \leq K$.

```
$ pred create --example SequencingToMinimizeWeightedTardiness -o sequencing-to-minimize-weighted-tardiness.json
$ pred solve sequencing-to-minimize-weighted-tardiness.json
$ pred evaluate sequencing-to-minimize-weighted-tardiness.json --config 0,0,2,1,0
```

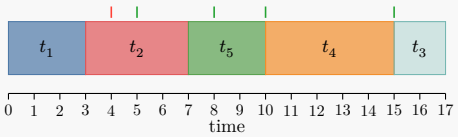


Figure 81: Single-machine schedule for the canonical weighted-tardiness example. The faded job is tardy; colored ticks mark the individual deadlines d_j.

Definition 2.163 (Sequencing to Minimize Maximum Cumulative Cost): Given a set T of n tasks, a precedence relation $\preceq$ on T, and an integer cost function $c : T \to \mathbb{Z}$ (negative values represent profits), find a one-machine schedule $\sigma : T \to \{1, 2, ..., n\}$ that respects the precedence constraints and minimizes the maximum cumulative cost $\min_\sigma \max_{t \in T} \sum_{\sigma(t') \leq \sigma(t)} c(t')$.

- Complexity: `factorial(num_tasks)`.
- Reduces to: ILP.

SequencingToMinimizeMaximumCumulativeCost	
costs	Task costs in schedule order-independent indexing
precedences	Precedence pairs (predecessor, successor)

Sequencing to Minimize Maximum Cumulative Cost is the scheduling problem SS7 in Garey & Johnson [5]. It is NP-complete by transformation from Register Sufficiency, even when every task cost is in $\{-1, 0, 1\}$ [5]. The problem models precedence-constrained task systems with resource consumption and release, where a negative cost corresponds to a profit or resource refund accumulated as the schedule proceeds.

When the precedence constraints form a series-parallel digraph, H. M. Abdel-Wahab and T. Kameda [133] gave a polynomial-time algorithm running in $O(n^2)$ time. C. L. Monma and J. B. Sidney [134] placed the problem in a broader family of sequencing objectives solvable efficiently on series-parallel precedence structures. The implementation here uses Lehmer-code enumeration of task orders, so the direct exact search induced by the model runs in $O(n!)$ time.

Example. Consider $n = 6$ tasks with costs $(2, -1, 3, -2, 1, -3)$ and precedence constraints $t_1 \preceq t_3$, $t_2 \preceq t_3$, $t_2 \preceq t_4$, $t_3 \preceq t_5$, $t_4 \preceq t_6$, $t_5 \preceq t_6$. The optimal schedule $(t_2, t_1, t_4, t_3, t_5, t_6)$ has cumulative sums $(-1, 1, -1, 2, 3, 0)$, achieving a maximum cumulative cost of 3.

```
$ pred create --example SequencingToMinimizeMaximumCumulativeCost -o sequencing-to-minimize-maximum-cumulative-cost.json
$ pred solve sequencing-to-minimize-maximum-cumulative-cost.json
$ pred evaluate sequencing-to-minimize-maximum-cumulative-cost.json --config 1,0,1,0,0,0
```

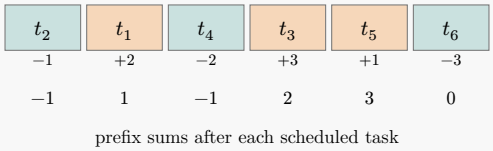


Figure 82: An optimal schedule for Sequencing to Minimize Maximum Cumulative Cost. Orange boxes add cost, teal boxes release cost, and the displayed prefix sums $(-1, 1, -1, 2, 3, 0)$ achieve a maximum of 3.

Definition 2.164 (Sequencing to Minimize Tardy Task Weight): Given a set T of n tasks, a processing-time function $\ell : T \to \mathbb{Z}^+$, a weight function $w : T \to \mathbb{Z}^+$, and a deadline function $d : T \to \mathbb{Z}^+$, find a one-machine schedule that minimizes $\sum_{t \in T} w(t)U(t)$, where $U(t) = 1$ if the completion time $C(t) > d(t)$ (task t is tardy) and $U(t) = 0$ otherwise.

- Complexity: `factorial(num_tasks)`.
- Reduces to: ILP.
- Reduces from: Partition.

SequencingToMinimizeTardyTaskWeight	
lengths	Processing time for each task
weights	Weight w(t) for each task
deadlines	Deadline d(t) for each task

Sequencing to Minimize Tardy Task Weight is problem SS8 in Garey & Johnson [5], usually written $1 \,\|\, \sum w_j U_j$. The unweighted variant $1 \,\|\, \sum U_j$ (minimize number of tardy tasks) is solvable in $O(n \log n)$ by Moore's algorithm [128], but the weighted version is NP-complete. The problem is closely related to

$1 \parallel \sum w_j T_j$ (Sequencing to Minimize Weighted Tardiness) but differs in using a 0/1 indicator U_j instead of the actual tardiness $T_j = \max(0, C_j - d_j)$.

Configurations are direct permutation encodings: the config vector $(\sigma_0, ..., \sigma_{\{n-1\}})$ specifies which task occupies each position, i.e., σ_p is the index of the task scheduled at position p. A configuration is valid iff it is a permutation of $\{0, ..., n-1\}$.

Example. Consider $n = 5$ tasks with lengths $\ell = (3, 2, 4, 1, 2)$, weights $w = (5, 3, 7, 2, 4)$, and deadlines $d = (6, 4, 10, 2, 8)$. The optimal schedule $(t_4, t_1, t_5, t_3, t_2)$ achieves completion times $(1, 4, 6, 10, 12)$. Tasks with completion time exceeding their deadline contribute weights $(0, 0, 0, 0, 3)$, giving total tardy weight 3.

```
$ pred create --example SequencingToMinimizeTardyTaskWeight -o sequencing-to-minimize-tardy-task-weight.json
$ pred solve sequencing-to-minimize-tardy-task-weight.json
$ pred evaluate sequencing-to-minimize-tardy-task-weight.json --config 3,0,4,2,1
```

Definition 2.165 (Sequencing with Deadlines and Set-Up Times): Given a set T of n tasks, a processing-time function $\ell : T \to \mathbb{Z}^+$, a deadline function $d : T \to \mathbb{Z}^+$, a compiler assignment $k : T \to C$ for a finite set C of compilers, and a setup-time function $s : C \to \mathbb{Z}_{\geq 0}$, determine whether there exists a single-machine schedule such that every task t completes by its deadline $d(t)$, where an additional setup time $s(k(t))$ is charged before t whenever $k(t) \neq k(t')$ for the immediately preceding task t'.

- Complexity: `factorial(num_tasks)`.
- Reduces to: ILP.

SequencingWithDeadlinesAndSetUpTimes	
lengths	Processing time for each task
deadlines	Deadline d(t) for each task
compilers	Compiler index k(t) for each task
setup_times	Setup time s(c) charged when switching to compiler c

Sequencing with Deadlines and Set-Up Times is problem SS14 in Garey & Johnson [5], usually written $1 \mid s_{ij} \mid$ feasibility. The problem is NP-complete even when all setup times are equal. It generalises Sequencing with Release Times and Deadlines (SS13) by replacing release-time windows with compiler-switch penalties.

Configurations are direct permutation encodings: the config vector $(\sigma_0, ..., \sigma_{n-1})$ specifies which task occupies each position, and a configuration is valid iff it is a permutation of $\{0, ..., n-1\}$.

Example. Consider $n = 5$ tasks with lengths $\ell = (2, 3, 1, 2, 2)$, deadlines $d = (4, 11, 3, 16, 7)$, compilers $k = (0, 1, 0, 1, 0)$, and setup times $s = (1, 2)$. The schedule $(t_3, t_1, t_5, t_2, t_4)$ achieves completion times $(1, 3, 5, 10, 12)$; every task meets its deadline, so the instance is feasible.

```
$ pred create --example SequencingWithDeadlinesAndSetUpTimes -o sequencing-with-deadlines-and-set-up-times.json
$ pred solve sequencing-with-deadlines-and-set-up-times.json
$ pred evaluate sequencing-with-deadlines-and-set-up-times.json --config 2,0,4,1,3
```

Definition 2.166 (Integral Flow with Homologous Arcs): Given a directed graph $G = (V, A)$ with source $s \in V$, sink $t \in V$, arc capacities $c : A \to \mathbb{Z}^+$, requirement $R \in \mathbb{Z}^+$, and a set $H \subseteq A \times A$ of homologous arc pairs, determine whether there exists an integral flow function $f : A \to \mathbb{Z}_{\geq 0}$ such that $f(a) \leq c(a)$ for

every $a \in A$, flow is conserved at every vertex in $V \setminus \{s, t\}$, $f(a) = f(a')$ for every $(a, a') \in H$, and the net flow into t is at least R.

- Complexity: `(max_capacity + 1)^num_arcs`.
- Reduces to: ILP.
- Reduces from: Satisfiability.

IntegralFlowHomologousArcs	
graph	Directed graph G = (V, A)
capacities	Capacity c(a) for each arc
source	Source vertex s
sink	Sink vertex t
requirement	Required net inflow R at the sink
homologous_pairs	Arc-index pairs (a, a') with f(a) = f(a')

Integral Flow with Homologous Arcs is the single-commodity equality-constrained flow problem listed as ND35 in Garey & Johnson [5]. Their catalog records the NP-completeness result attributed to Sahni and notes that the unit-capacity restriction remains hard, while the corresponding non-integral relaxation is polynomial-time equivalent to linear programming [5].

The implementation uses one integer variable per arc, so exhaustive search over the induced configuration space runs in $O((C + 1)^m)$ for $m = |A|$ and $C = \max_{a \in A} c(a)$[51].

Example. The canonical fixture instance has source $s = v_0$, sink $t = v_5$, unit capacities on all eight arcs, requirement $R = 2$, and homologous pairs (a_2, a_5) and (a_4, a_3). The stored satisfying configuration routes one unit along $0 \to 1 \to 3 \to 5$ and one unit along $0 \to 2 \to 4 \to 5$. Thus the paired arcs $(1, 3)$ and $(2, 4)$ both carry 1, while $(1, 4)$ and $(2, 3)$ both carry 0. Every nonterminal vertex has equal inflow and outflow, and the sink receives two units of flow, so the verifier returns true.

```
$ pred create --example IntegralFlowHomologousArcs -o integral-flow-homologous-arcs.json
$ pred solve integral-flow-homologous-arcs.json --solver brute-force
$ pred evaluate integral-flow-homologous-arcs.json --config 1,1,1,0,0,1,1,1
```

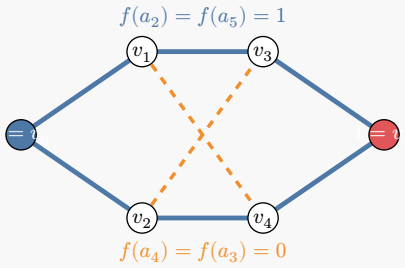


Figure 83: Canonical YES instance for Integral Flow with Homologous Arcs. Solid blue arcs carry the satisfying integral flow; dashed orange arcs form the second homologous pair, constrained to equal zero.

Definition 2.167 (Directed Two-Commodity Integral Flow): Given a directed graph $G = (V, A)$ with arc capacities $c : A \to \mathbb{Z}^+$, two source-sink pairs (s_1, t_1) and (s_2, t_2), and requirements $R_1, R_2 \in \mathbb{Z}^+$, determine whether there exist two integral flow functions $f_1, f_2 : A \to \mathbb{Z}_{\geq 0}$ such that (1) $f_1(a) + f_2(a) \leq c(a)$ for all $a \in A$, (2) for each commodity $i \in \{1, 2\}$, flow f_i is conserved at every vertex except s_i and t_i, and (3) the net flow into t_i under f_i is at least R_i.

- Complexity: `(max_capacity + 1)^(2 * num_arcs)`.
- Reduces to: ILP.
- Reduces from: KSatisfiability.

[51] This is the exact search bound induced by the implemented per-arc domains $f(a) \in \{0, \ldots, c(a)\}$. In the unit-capacity special case, it simplifies to $O(2^m)$.

DirectedTwoCommodityIntegralFlow	
graph	Directed graph G = (V, A)
capacities	Capacity c(a) for each arc
source_1	Source vertex s_1 for commodity 1
sink_1	Sink vertex t_1 for commodity 1
source_2	Source vertex s_2 for commodity 2
sink_2	Sink vertex t_2 for commodity 2
requirement_1	Flow requirement R_1 for commodity 1
requirement_2	Flow requirement R_2 for commodity 2

Directed Two-Commodity Integral Flow is a fundamental NP-complete problem in multicommodity flow theory, catalogued as ND38 in Garey & Johnson [5]. While single-commodity max-flow is solvable in polynomial time and fractional multicommodity flow reduces to linear programming, requiring integral flows with just two commodities makes the problem NP-complete.

NP-completeness was proved by Even, Itai, and Shamir via reduction from 3-SAT [135]. The problem remains NP-complete even when all arc capacities are 1 and $R_1 = 1$. No sub-exponential exact algorithm is known; brute-force enumeration over $(C+1)^{2|A|}$ flow assignments dominates, where $C = \max_{a \in A} c(a)$.[52]

Example. Consider a directed graph with 6 vertices and 8 arcs (all with unit capacity), sources $s_1 = 0$, $s_2 = 1$, sinks $t_1 = 4$, $t_2 = 5$, and requirements $R_1 = R_2 = 1$. Commodity 1 routes along the path $0 \to 2 \to 4$ and commodity 2 along $1 \to 3 \to 5$, satisfying all capacity and conservation constraints.

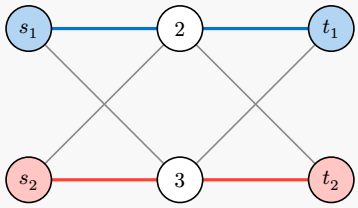


Figure 84: Two-commodity flow: commodity 1 (blue, $s_1 \to 2 \to t_1$) and commodity 2 (red, $s_2 \to 3 \to t_2$).

Definition 2.168 (Minimum Edge-Cost Flow): Given a directed graph $G = (V, A)$ with arc capacities $c : A \to \mathbb{Z}^+$, arc prices $p : A \to \mathbb{Z}$, a source vertex s, a sink vertex t, and a flow requirement $R \in \mathbb{Z}^+$, find an integral flow $f : A \to \mathbb{Z}_{\geq 0}$ of value at least R that minimizes the total edge cost $\sum_{a \in A : f(a) > 0} p(a)$ — the sum of prices of arcs carrying nonzero flow.

- Complexity: `(max_capacity + 1)^num_edges`.
- Reduces to: ILP.

MinimumEdgeCostFlow	
graph	Directed graph G = (V, A)
prices	Price p(a) for each arc
capacities	Capacity c(a) for each arc
source	Source vertex s
sink	Sink vertex t
required_flow	Flow requirement R

Minimum Edge-Cost Flow is an NP-hard network design problem that arises in telecommunications and logistics, where there is a fixed cost (price) for activating each link, independent of the actual traffic volume. Unlike the classical minimum-cost flow problem (where cost is proportional to flow), the edge-cost variant introduces a combinatorial selection aspect: choosing which arcs to activate. The problem is closely related to fixed-charge network flow problems [5].

[52] No algorithm improving on brute-force is known for Directed Two-Commodity Integral Flow.

The brute-force bound $(C+1)^{|A|}$ arises from enumerating all possible integral flow vectors, where $C = \max_{a \in A} c(a)$.[53]

Example. Consider a directed graph with 5 vertices, source $s = 0$, sink $t = 4$, requirement $R = 3$, and 6 arcs with capacities $c(a) = 2$ for all arcs. The prices are $p(0,1) = 3$, $p(0,2) = 1$, $p(0,3) = 2$, and all arcs entering the sink have price 0. The optimal flow routes 1 unit via vertex 2 and 2 units via vertex 3, activating 4 arcs for a total edge cost of $1 + 2 + 0 + 0 = 3$.

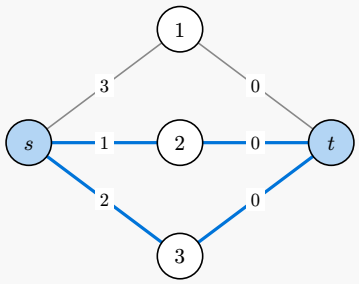


Figure 85: Minimum Edge-Cost Flow: optimal flow (blue) routes via vertices 2 and 3, total edge cost 3. Arc labels show prices.

Definition 2.169 (Integral Flow with Bundles): Given a directed graph $G = (V, A)$, specified vertices $s, t \in V$, a family of arc bundles $I_1, ..., I_k \subseteq A$ whose union covers A, positive bundle capacities $c_1, ..., c_k$, and a requirement $R \in \mathbb{Z}^+$, determine whether there exists an integral flow $f : A \to \mathbb{Z}_{\geq 0}$ such that (1) $\sum_{a \in I_j} f(a) \leq c_j$ for every bundle j, (2) flow is conserved at every vertex in $V \setminus \{s, t\}$, and (3) the net flow into t is at least R.

- Complexity: `2^num_arcs`.
- Reduces to: ILP.
- Reduces from: MaximumIndependentSet.

IntegralFlowBundles	
graph	Directed graph G=(V,A)
source	Source vertex s
sink	Sink vertex t
bundles	Bundles of arc indices covering A
bundle_capacities	Capacity c_j for each bundle I_j
requirement	Required net inflow R at the sink

Integral Flow with Bundles is the shared-capacity single-commodity flow problem listed as ND36 in Garey & Johnson [5]. Sahni introduced it as one of a family of computationally related network problems and showed that the bundled-capacity variant is NP-complete even in a very sparse unit-capacity regime [61].

The implementation keeps one non-negative integer variable per directed arc. Unlike ordinary max-flow, the usable range of an arc is not determined by an intrinsic per-arc capacity; it is bounded instead by the smallest bundle capacity among the bundles that contain that arc. The registered $O(2^m)$ catalog bound therefore reflects the unit-capacity case with $m = |A|$, which is exactly the regime highlighted by Garey & Johnson and Sahni.[54]

Example. The canonical YES instance has source $s = v_0$, sink $t = v_3$, and arcs $(0,1)$, $(0,2)$, $(1,3)$, $(2,3)$, $(1,2)$, $(2,1)$. The three bundles are $I_1 = \{(0,1),(0,2)\}$, $I_2 = \{(1,3),(2,1)\}$, and $I_3 = \{(2,3),(1,2)\}$, each with capacity 1. Sending one unit along the path $0 \to 1 \to 3$ yields the flow vector $(1,0,1,0,0,0)$: bundle I_1 contributes $1 + 0 = 1$, bundle I_2 contributes $1 + 0 = 1$, bundle I_3 contributes $0 + 0 = 0$, and the only

[53] No sub-exponential exact algorithm is known for Minimum Edge-Cost Flow.

[54] No exact worst-case algorithm improving on brute-force is claimed here for the bundled-capacity formulation.

nonterminal vertices v_1, v_2 satisfy conservation. If the requirement is raised from $R = 1$ to $R = 2$, the same gadget becomes infeasible because I_1 caps the total outflow leaving the source at one unit.

```
$ pred create --example IntegralFlowBundles -o integral-flow-bundles.json
$ pred solve integral-flow-bundles.json
$ pred evaluate integral-flow-bundles.json --config 1,0,1,0,0,0
```

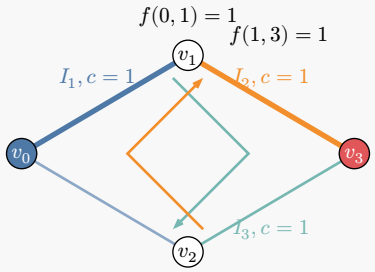


Figure 86: Canonical YES instance for Integral Flow with Bundles. Thick blue/orange arcs carry the satisfying flow $0 \to 1 \to 3$, while the lighter arcs show the two unused alternatives coupled into bundles I_1, I_2, and I_3.

Definition 2.170 (Integral Flow With Multipliers): Given a directed graph $G = (V, A)$, distinguished vertices $s, t \in V$, arc capacities $c : A \to \mathbb{Z}^+$, vertex multipliers $h : V \setminus \{s, t\} \to \mathbb{Z}^+$, and a requirement $R \in \mathbb{Z}^+$, determine whether there exists an integral flow function $f : A \to \mathbb{Z}_{\geq 0}$ such that (1) $f(a) \leq c(a)$ for every $a \in A$, (2) for each nonterminal vertex $v \in V \setminus \{s, t\}$, the value $h(v)$ times the total inflow into v equals the total outflow from v, and (3) the net flow into t is at least R.

- Complexity: `(max_capacity + 1)^num_arcs`.
- Reduces to: ILP.

IntegralFlowWithMultipliers	
graph	Directed graph G = (V, A)
source	Source vertex s
sink	Sink vertex t
multipliers	Vertex multipliers h(v) in vertex order; source/sink entries are ignored
capacities	Arc capacities c(a) in graph arc order
requirement	Required net inflow R at the sink

Integral Flow With Multipliers is Garey and Johnson's gain/loss network problem ND33 [5]. Sahni includes the same integral vertex-multiplier formulation among his computationally related problems, where partition-style reductions show that adding discrete gain factors destroys the ordinary max-flow structure [61]. The key wrinkle is that conservation is no longer symmetric: one unit entering a vertex may force several units to leave, so the feasible integral solutions behave more like multiplicative gadgets than classical flow balances.

When every multiplier equals 1, the model collapses to ordinary single-commodity max flow and becomes polynomial-time solvable by the standard network-flow machinery summarized in Garey and Johnson [5]. Jewell studies a different continuous flow-with-gains model in which gain factors live on arcs and the flow may be fractional [136]. That continuous relaxation remains polynomially tractable, so it should not be conflated with the NP-complete integral vertex-multiplier decision problem catalogued here. In this implementation the witness stores one bounded integer variable per arc, giving the direct exact-search bound $O((C + 1)^m)$ where $m = |A|$ and $C = \max_{a \in A} c(a)$.

Example. The canonical fixture encodes the Partition multiset $\{2, 3, 4, 5, 6, 4\}$ using source $s = v_0$, sink $t = v_7$, six unit-capacity arcs out of s, six sink arcs with capacities $(2, 3, 4, 5, 6, 4)$, and multipliers $(2, 3, 4, 5, 6, 4)$ on the intermediate vertices. Setting the source arcs to v_1, v_3, and v_5 to 1 forces outgoing sink arcs of 2, 4,

and 6, respectively. The sink therefore receives net inflow $2 + 4 + 6 = 12$, exactly meeting the requirement $R = 12$.

```
$ pred create --example IntegralFlowWithMultipliers -o integral-flow-with-multipliers.json
$ pred solve integral-flow-with-multipliers.json --solver brute-force
$ pred evaluate integral-flow-with-multipliers.json --config 1,0,1,0,1,0,2,0,4,0,6,0
```

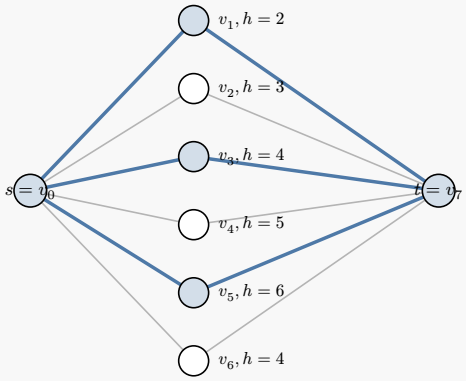


Figure 87: Integral Flow With Multipliers: the blue branches send one unit from s into v_1, v_3, and v_5, forcing sink inflow $2 + 4 + 6 = 12$ at t.

Definition 2.171 (Additional Key): Given a set A of attribute names, a collection F of functional dependencies on A, a subset $R \subseteq A$, and a set K of candidate keys for the relational scheme $\langle R, F \rangle$, determine whether there exists a subset $R' \subseteq R$ such that $R' \notin K$, the closure R'^{+} under F equals R, and no proper subset of R' also has this property.

- Complexity: `2^num_relation_attrs * num_dependencies * num_attributes`.

```
AdditionalKey
num_attributes Number of attributes in A
dependencies   Functional dependencies F; each (lhs, rhs)
relation_attrs Relation scheme attributes R ⊆ A
known_keys     Known candidate keys K
```

A classical NP-complete problem from relational database theory [137]. Enumerating all candidate keys is necessary to verify Boyce-Codd Normal Form (BCNF), and the NP-completeness of Additional Key implies that BCNF testing is intractable in general. The best known exact algorithm is brute-force enumeration of all $2^{|R|}$ subsets, checking each for the key property via closure computation under Armstrong's axioms.[55]

Example. Consider attribute set $A = \{0, 1, 2, 3, 4, 5\}$ with functional dependencies $F = \{\{0, 1\} \to \{2, 3\}, \{2, 3\} \to \{4, 5\}, \{4, 5\} \to \{0, 1\}, \{0, 2\} \to \{3\}, \{3, 5\} \to \{1\}\}$, relation $R = A$, and known keys $K = \{\{0, 1\}, \{2, 3\}, \{4, 5\}\}$. The subset $\{0, 2\}$ is an additional key: starting from $\{0, 2\}$, we apply $\{0, 2\} \to \{3\}$ to get $\{0, 2, 3\}$, then $\{2, 3\} \to \{4, 5\}$ to get $\{0, 2, 3, 4, 5\}$, then $\{4, 5\} \to \{0, 1\}$ to reach $R^{+} = A$. The set $\{0, 2\}$ is minimal (neither $\{0\}$ nor $\{2\}$ alone determines A) and $\{0, 2\} \notin K$, so the answer is YES.

Definition 2.172 (Conjunctive Boolean Query): Given a finite domain $D = \{0, ..., d - 1\}$, a collection of relations $R_0, R_1, ..., R_{m-1}$ where each R_i is a set of a_i-tuples with entries from D, and a conjunctive Boolean query

$$Q = (\exists y_0, y_1, ..., y_{l-1})(A_0 \wedge A_1 \wedge \cdots \wedge A_{r-1})$$

[55] No algorithm improving on brute-force is known for the Additional Key problem.

where each *atom* A_j has the form $R_{i_j}(u_0, u_1, ...)$ with every u in $\{y_0, ..., y_{l-1}\} \cup D$, determine whether there exists an assignment to the variables that makes Q true — i.e., the resolved tuple of every atom belongs to its relation.

- Complexity: `domain_size ^ num_variables`.
- Reduces from: KClique.

ConjunctiveBooleanQuery	
domain_size	Size of the finite domain D
relations	Collection of relations R
num_variables	Number of existentially quantified variables
conjuncts	Query conjuncts: (relation_index, arguments)

The Conjunctive Boolean Query (CBQ) problem is one of the most fundamental problems in database theory and finite model theory. A. K. Chandra and P. M. Merlin [138] showed that evaluating conjunctive queries is NP-complete by reduction from the Clique problem. CBQ is equivalent to the Constraint Satisfaction Problem (CSP) and to the homomorphism problem for relational structures; this equivalence connects database query evaluation, constraint programming, and graph theory under a single computational framework [139].

For queries of bounded *hypertree-width*, evaluation becomes polynomial-time [140]. The general brute-force algorithm enumerates all d^l variable assignments and checks every atom, running in $O\big(d^l \cdot r \cdot \max_i a_i\big)$ time.[56]

Example. Let $D = \{0, ..., 5\}$ ($d = 6$), with 2 relations:

R_0	c_0	c_1
τ_0	0	3
τ_1	1	3
τ_2	2	4
τ_3	3	4
τ_4	4	5

R_1	c_0	c_1	c_2
τ_0	0	1	5
τ_1	1	2	5
τ_2	2	3	4
τ_3	0	4	3

The query has 2 variables (y_0, y_1) and 3 atoms:

$$Q = (\exists y_0, y_1)(A_0 = R_0(y_0, 3) \wedge A_1 = R_0(y_1, 3) \wedge A_2 = R_1(y_0, y_1, 5))$$

Under the assignment $y_0 = 0$, $y_1 = 1$: atom A_0 resolves to $(0, 3) \in R_0$ (row τ_0), atom A_1 resolves to $(1, 3) \in R_0$ (row τ_1), and atom A_2 resolves to $(0, 1, 5) \in R_1$ (row τ_0). All three atoms are satisfied, so Q is true.

```
$ pred create --example ConjunctiveBooleanQuery -o conjunctive-boolean-query.json
$ pred solve conjunctive-boolean-query.json
$ pred evaluate conjunctive-boolean-query.json --config 0,1
```

Definition 2.173 (Consecutive Ones Matrix Augmentation): Given an $m \times n$ binary matrix A and a nonnegative integer K, determine whether there exists a matrix A', obtained from A by changing at most K zero entries to one, such that some permutation of the columns of A' has the consecutive ones property.

- Complexity: `factorial(num_cols) * num_rows * num_cols`.
- Reduces to: ILP.

ConsecutiveOnesMatrixAugmentation	
matrix	m x n binary matrix A
bound	Upper bound K on zero-to-one augmentations

[56] No substantially faster general algorithm is known for arbitrary conjunctive Boolean queries.

Consecutive Ones Matrix Augmentation is problem SR16 in Garey & Johnson [5]. It asks whether a binary matrix can be repaired by a bounded number of augmenting flips so that every row's 1-entries become contiguous after reordering the columns. This setting appears in information retrieval and DNA physical mapping, where matrices close to the consecutive ones property can still encode useful interval structure. Booth and Lueker showed that testing whether a matrix already has the consecutive ones property is polynomial-time via PQ-trees [141], but allowing bounded augmentation makes the decision problem NP-complete [77]. The direct exhaustive search tries all $n!$ column permutations and, for each one, computes the minimum augmentation cost by filling the holes between the first and last 1 in every row[57].

Example. Consider the 4×5 matrix $A = (1, 0, 0, 1, 1; 1, 1, 0, 0, 0; 0, 1, 1, 0, 1; 0, 0, 1, 1, 0)$ with $K = 2$. Under the permutation $\pi = (0, 1, 4, 2, 3)$, the reordered rows are $r_1 = (1, 0, 1, 0, 1)$, $r_2 = (1, 1, 0, 0, 0)$, $r_3 = (0, 1, 1, 1, 0)$, $r_4 = (0, 0, 0, 1, 1)$. The first row becomes $(1, 0, 1, 0, 1)$, so filling the two interior gaps yields $(1, 1, 1, 1, 1)$. The other three rows already have consecutive 1-entries under the same order, so the total augmentation cost is 2 and $2 \leq 2$, making the instance satisfiable.

```
$ pred create --example ConsecutiveOnesMatrixAugmentation -o consecutive-ones-matrix-augmentation.json
$ pred solve consecutive-ones-matrix-augmentation.json --solver brute-force
$ pred evaluate consecutive-ones-matrix-augmentation.json --config 0,1,4,2,3
```

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Figure 88: The canonical 4×5 example matrix for Consecutive Ones Matrix Augmentation. The permutation $\pi = (0, 1, 4, 2, 3)$ makes only the first row need augmentation, and exactly two zero-to-one flips suffice.

Definition 2.174 (Consecutive Ones Submatrix): Given an $m \times n$ binary matrix A and an integer K with $0 \leq K \leq n$, determine whether there exists a subset of K columns of A whose columns can be permuted so that in each row all 1′s occur consecutively (the *consecutive ones property*).

- Complexity: `2^(num_cols) * (num_rows + num_cols)`.
- Reduces to: ILP.
- Reduces from: HamiltonianPath.

ConsecutiveOnesSubmatrix	
matrix	m×n binary matrix A
bound	Required number of columns K

The Consecutive Ones Property (C1P) — that the columns of a binary matrix can be ordered so that all 1′s in each row are contiguous — is fundamental in computational biology (DNA physical mapping), interval graph recognition, and PQ-tree algorithms. Testing whether a full matrix has the C1P is polynomial: Booth and Lueker [141] gave a linear-time PQ-tree algorithm running in $O(m + n + f)$ where f is the number of 1-entries. However, finding the largest column subset with the C1P is NP-complete, proven by Booth [77] via transformation from Hamiltonian Path. This implementation permits the vacuous case $K = 0$, where the empty submatrix is immediately satisfying. The best known exact algorithm is brute-force enumeration of all $\binom{n}{K}$ column subsets, testing each for the C1P in $O(m + n)$ time[58].

Example. Consider the 3×4 matrix $A = (1, 1, 0, 1; 1, 0, 1, 1; 0, 1, 1, 0)$ with $K = 3$. Selecting columns $\{0, 1, 3\}$ yields a 3×3 submatrix. Under column permutation $[1, 0, 3]$, each row's 1-entries are contiguous: row 1 has

[57] No algorithm improving on brute-force permutation enumeration is known for the general problem in this repository's supported setting.

[58] No algorithm improving on brute-force subset enumeration is known for the general Consecutive Ones Submatrix problem.

$[1, 1, 1]$, row 2 has $[0, 1, 1]$, and row 3 has $[1, 0, 0]$. The full 3×4 matrix does *not* have the C1P (it contains a Tucker obstruction), but two of the four 3-column subsets do.

```
$ pred create --example ConsecutiveOnesSubmatrix -o consecutive-ones-submatrix.json
$ pred solve consecutive-ones-submatrix.json
$ pred evaluate consecutive-ones-submatrix.json --config 1,1,0,1
```

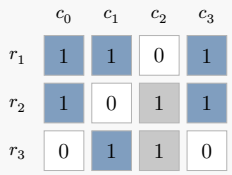


Figure 89: Binary matrix A (3×4) with $K = 3$. Blue-highlighted columns $\{0, 1, 3\}$ form a submatrix with the consecutive ones property under a suitable column permutation. Grey cells are 1-entries in non-selected columns.

Definition 2.175 (Sparse Matrix Compression): Given an $m \times n$ binary matrix A and a positive integer K, determine whether there exist a shift function $s : \{1, ..., m\} \to \{1, ..., K\}$ and a storage vector $b \in \{0, 1, ..., m\}^{\{n+K\}}$ such that, for every row i and column j, $A_{ij} = 1$ if and only if $b_{s(i)+j-1} = i$.

- Complexity: `(bound_k ^ num_rows) * num_rows * num_cols`.
- Reduces to: ILP.

SparseMatrixCompression	
matrix	m x n binary matrix A
bound_k	Maximum shift range K

Sparse Matrix Compression appears as problem SR13 in Garey and Johnson [5]. It models row-overlay compression for sparse lookup tables: rows may share storage positions only when their shifted 1-entries never demand different row labels from the same slot. The implementation in this crate searches over row shifts only, then reconstructs the implied storage vector internally. This yields the direct exact bound $O(K^m \cdot m \cdot n)$ for m rows and n columns.[59]

Example. Let $A = (1, 0, 0, 1; 0, 1, 0, 0; 0, 0, 1, 0; 1, 0, 0, 0)$ and $K = 2$. The stored config $(1, 1, 1, 0)$ encodes the one-based shifts $s = (2, 2, 2, 1)$. These shifts place the four row supports at positions $\{2, 5\}$, $\{3\}$, $\{4\}$, and $\{1\}$ respectively, so the supports are pairwise disjoint. The implied overlay vector is therefore $b = (4, 1, 2, 3, 1, 0)$, and this is the unique satisfying shift assignment among the $2^4 = 16$ configs in the canonical fixture.

```
$ pred create --example SparseMatrixCompression -o sparse-matrix-compression.json
$ pred solve sparse-matrix-compression.json
$ pred evaluate sparse-matrix-compression.json --config 1,1,1,0
```

[59] The storage vector is not enumerated as part of the configuration space. Once the shifts are fixed, every occupied slot is forced by the 1-entries of the shifted rows.

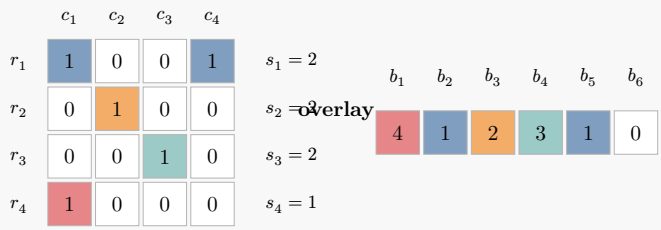


Figure 90: Canonical Sparse Matrix Compression YES instance. Row-colored 1-entries on the left are shifted into the overlay vector on the right, producing $b = (4, 1, 2, 3, 1, 0)$.

Definition 2.176 (Minimum Matrix Cover): Given an $n \times n$ nonnegative integer matrix A, find a function $f : \{1, ..., n\} \to \{-1, +1\}$ minimizing $\sum_{i,j} a_{ij} \cdot f(i) \cdot f(j)$.

- Complexity: `2^num_rows`.
- Reduces to: ILP.

MinimumMatrixCover	
matrix	n×n nonnegative integer matrix

Minimum Matrix Cover asks for a sign assignment to rows (equivalently columns) of a square matrix that minimizes the resulting quadratic form. Each binary variable $x_i \in \{0, 1\}$ encodes a sign $f(i) = 2x_i - 1$. Since $f(i)^2 = 1$, diagonal entries contribute a constant $\sum_i a_{ii}$; the optimization depends only on off-diagonal structure. The brute-force complexity is $O(2^n)$ where n is the matrix dimension.[60]

Example. Let A be the 4×4 symmetric matrix with zero diagonal shown below. The optimal config (0, 1, 1, 0) assigns signs $(-1, +1, +1, -1)$, yielding value $= -20$.

```
$ pred create --example MinimumMatrixCover -o mmc.json
$ pred solve mmc.json
$ pred evaluate mmc.json --config 0,1,1,0
```

Definition 2.177 (Minimum Matrix Domination): Given an $m \times n$ binary matrix M, find a minimum-cardinality subset C of 1-entries such that every 1-entry not in C shares a row or column with some entry in C.

- Complexity: `2^num_ones`.

MinimumMatrixDomination	
matrix	n×n binary matrix M

Minimum Matrix Domination is a matrix analogue of the dominating set problem. Each binary variable corresponds to a 1-entry in row-major order; the evaluator checks that every unselected 1-entry shares a row or column with at least one selected entry. The brute-force complexity is $O(2^k)$ where k is the number of 1-entries.

Example. Let M be the 6×6 adjacency matrix of P_6 (the path on 6 vertices), which has 10 non-zero entries. The optimal config (1, 1, 0, 0, 0, 0, 1, 1, 0, 0) selects entries at positions (0, 1), (1, 0), (3, 4), (4, 3), yielding value $= 4$.

```
$ pred create --example MinimumMatrixDomination -o mmd.json
$ pred solve mmd.json
$ pred evaluate mmd.json --config 1,1,0,0,0,0,1,1,0,0
```

[60] No algorithm improving on brute-force enumeration of all 2^n sign assignments is known for the general case.

Definition 2.178 (Minimum Weight Decoding): Given an $n \times m$ binary parity-check matrix H and a binary syndrome vector $s \in \{0,1\}^n$, find a binary vector $x \in \{0,1\}^m$ minimizing the Hamming weight $|x| = \sum_{j=0}^{m-1} x_j$ subject to $Hx \equiv s \pmod 2$.

- Complexity: `2^(0.0494 * num_cols)`.
- Reduces to: ILP.

MinimumWeightDecoding	
matrix	n×m binary parity-check matrix H
target	binary syndrome vector s of length n

Minimum Weight Decoding is a fundamental problem in coding theory. Given a linear code with parity-check matrix H, the task is to find the minimum-weight error pattern consistent with a received syndrome. The problem is equivalent to finding the closest codeword to a received word and is central to the hardness of decoding random linear codes.

The best known algorithms for general instances use information set decoding techniques, achieving $O(2^{0.0494n})$ where n is the block length.

Example. Let H be the 3×4 binary matrix and $s = (1, 1, 0)$. The optimal config (0, 0, 1, 0) has Hamming weight 1.

```
$ pred create --example MinimumWeightDecoding -o mwd.json
$ pred solve mwd.json
$ pred evaluate mwd.json --config 0,0,1,0
```

Rule 2.9: (Minimum Weight Decoding → Integer Linear Programming) The GF(2) constraint $Hx \equiv s \pmod 2$ is linearized by introducing integer slack variables: for each row i, $\sum_j H_{ij} x_j - 2k_i = s_i$ where $k_i \geq 0$ is an integer. Binary bounds $x_j \leq 1$ are added, and the objective minimizes $\sum x_j$.
Overhead: `num_vars = num_cols + num_rows`, `num_constraints = num_rows + num_cols`.

Proof: *Construction.* Given $H \in \{0,1\}^{n \times m}$ and $s \in \{0,1\}^n$, create an ILP with $m + n$ variables: $x_0, \ldots, x_{m-1}$ (binary) and $k_0, \ldots, k_{n-1}$ (non-negative integer). Add n equality constraints $\sum_j H_{ij} x_j - 2k_i = s_i$ and m binary bounds $x_j \leq 1$. The objective is $\min \sum_{j=0}^{m-1} x_j$.

Correctness. ($\Rightarrow$) If x^* is feasible for the source, then $Hx^* \equiv s \pmod 2$, so $\sum_j H_{ij} x_j^* = s_i + 2k_i$ for some $k_i \geq 0$. Setting these k_i values gives a feasible ILP solution with the same objective. ($\Leftarrow$) If (x^*, k^*) is feasible for the ILP, then $\sum_j H_{ij} x_j^* = s_i + 2k_i^*$ implies $\sum_j H_{ij} x_j^* \equiv s_i \pmod 2$ for all i, and $x_j^* \in \{0,1\}$ by the binary bounds.

Solution extraction. Take the first m variables as the source configuration. □

Example: Minimum Weight Decoding to ILP (3 rows, 4 columns)

Source: MinimumWeightDecoding **Target:** ILP

Definition 2.179 (Minimum Weight Solution to Linear Equations): Given an $n \times m$ integer matrix A and an integer vector $b \in \mathbb{Z}^n$, find a rational vector $y \in \mathbb{Q}^m$ satisfying $Ay = b$ that minimizes $\|y\|_0$ (the number of non-zero entries of y).

- Complexity: `2^num_variables`.
- Reduces from: ExactCoverBy3Sets.

MinimumWeightSolutionToLinearEquations	
matrix	n×m integer matrix A
rhs	right-hand side vector b of length n

Minimum Weight Solution to Linear Equations is a sparsity-seeking variant of solving linear systems. Each binary variable x_j indicates whether the j-th component of y may be non-zero; the evaluator forms the restricted submatrix A' from the selected columns and checks whether b lies in its column space via integer Gaussian elimination (using i128 arithmetic for exact rational consistency). If the restricted system $A'y' = b$ is consistent, the value is the number of selected columns; otherwise the configuration is infeasible.

Example. Let A be the 2×4 matrix $(1, 2, 3, 1\ 2, 1, 1, 3)$ with $b = (5, 4)$. The optimal config $(1, 1, 0, 0)$ selects columns 0, 1, yielding value $= 2$.

```
$ pred create --example MinimumWeightSolutionToLinearEquations -o mwsle.json
$ pred solve mwsle.json
$ pred evaluate mwsle.json --config 1,1,0,0
```

Definition 2.180 (Integer Expression Membership): Given a recursive integer expression e over union ($\cup$) and Minkowski sum ($+$) operations on singleton positive integers, and a target $K \in \mathbb{N}^+$, determine whether $K \in \text{eval}(e)$, where $\text{eval}(\text{Atom}(n)) = \{n\}$, $\text{eval}(F \cup G) = \text{eval}(F) \cup \text{eval}(G)$, and $\text{eval}(F + G) = \{m + n : m \in \text{eval}(F), n \in \text{eval}(G)\}$.

- Complexity: `2^num_union_nodes`.
- Reduces from: SubsetSum.

IntegerExpressionMembership	
expression	Recursive expression tree
target	Target integer K

Integer Expression Membership asks whether a specific integer is reachable by selecting one branch at each union node in a recursive expression tree. Each configuration assigns a binary choice (left or right) at every union node in depth-first order; with all unions resolved, the expression reduces to a chain of sums and atoms that evaluates to a single integer.[61]

Example. Consider $e = (1 \cup 4) + (3 \cup 6) + (2 \cup 5)$ with $K = 12$. There are $u = 3$ union nodes, giving $2^3 = 8$ configurations. The reachable set is $\{6, 9, 12, 15\}$. Since $12 \in \{6, 9, 12, 15\}$, the answer is YES. One witness: choose right (4), right (6), left (2) at the three union nodes, giving $4 + 6 + 2 = 12 = K$.

```
$ pred create --example IntegerExpressionMembership -o iem.json
$ pred solve iem.json --solver brute-force
$ pred evaluate iem.json --config 1,1,0
```

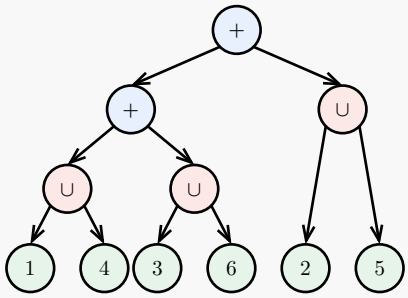


Figure 91: Expression tree $e = (1 \cup 4) + (3 \cup 6) + (2 \cup 5)$ with target $K = 12$. Blue nodes are sums ($+$), red nodes are unions ($\cup$), green leaves are atoms. Choosing right at the first two unions and left at the third yields $4 + 6 + 2 = 12$.

[61] No algorithm improving on brute-force enumeration of all 2^u union-branch combinations (where u is the number of union nodes) is known for the general case.

Definition 2.181 (Feasible Basis Extension): Given an $m \times n$ integer matrix A with $m < n$, a column vector $\overline{a} \in \mathbb{Z}^m$, and a subset S of column indices with $|S| < m$, determine whether there exists a *feasible basis* B — a set of m column indices including S — such that the $m \times m$ submatrix A_B is nonsingular and $A_B^{-1}\overline{a} \geq 0$ (componentwise).

- Complexity: `2^num_columns * num_rows^3`.

FeasibleBasisExtension	
matrix	m x n integer matrix A (row-major)
rhs	Column vector a_bar of length m
required_columns	Subset S of column indices that must be in the basis

The Feasible Basis Extension problem arises in linear programming theory and the study of simplex method pivoting rules. It was shown NP-complete by Murty [142] via a reduction from Hamiltonian Circuit, establishing that determining whether a partial basis can be extended to a feasible one is computationally intractable in general. The problem is closely related to the question of whether a given linear program has a feasible basic solution containing specified variables. The best known exact algorithm is brute-force enumeration of all $\binom{n-|S|}{m-|S|}$ candidate extensions, testing each for nonsingularity and non-negativity of the solution in $O(m^3)$ time.[62]

Example. Consider the 3×6 matrix $A = (1, 0, 1, 2, -1, 0; 0, 1, 0, 1, 1, 2; 0, 0, 1, 1, 0, 1)$ with $\overline{a} = (7, 5, 3)^\top$ and required columns $S = \{0, 1\}$. We need 1 additional column from the free set $\{2, 3, 4, 5\}$. Selecting column 2 gives basis $B = \{0, 1, 2\}$, which yields $A_B^{-1}\overline{a} = (4, 5, 3)^\top \geq 0$. Column 4 makes A_B singular, and column 5 produces a negative component.

```
$ pred create --example FeasibleBasisExtension -o feasible-basis-extension.json
$ pred solve feasible-basis-extension.json --solver brute-force
$ pred evaluate feasible-basis-extension.json --config 1,0,0,0
```

	c_0	c_1	c_2	c_3	c_4	c_5	$\overline{a}$
r_1	1	0	1	2	−1	0	7
r_2	0	1	0	1	1	2	5
r_3	0	0	1	1	0	1	3

Figure 92: Feasible Basis Extension instance (3×6). Orange columns are required ($S = \{0, 1\}$), blue column is the selected extension. Together they form a nonsingular basis with non-negative solution.

Definition 2.182 (Maximum Likelihood Ranking): Given an $n \times n$ comparison matrix A with $a_{ii} = 0$ for all i, find a permutation π of $\{0, ..., n-1\}$ minimizing the *disagreement cost* $\sum_{i>j} a_{\pi(i),\pi(j)}$, where π maps rank positions to items.

- Complexity: `num_items * num_items * 2^num_items`.
- Reduces to: ILP.
- Reduces from: MinimumFeedbackArcSet.

MaximumLikelihoodRanking	
matrix	Antisymmetric comparison matrix A (a_ij + a_ji = c, a_ii = 0)

The Maximum Likelihood Ranking problem arises in voting theory and tournament aggregation: given pairwise comparison counts between n alternatives, find a linear ordering that best explains the observed data [5]. The problem is equivalent to finding a minimum-weight linear extension of a tournament. When $a_{ij} + a_{ji} = c$ for all $i \neq j$, the problem is also known as the *Kemeny ranking* or *minimum feedback arc set in*

[62] No algorithm improving on brute-force enumeration is known for the general Feasible Basis Extension problem.

tournaments. The best known exact algorithm uses dynamic programming over subsets (Held–Karp style), running in $O(n^2 \cdot 2^n)$ time.[63]

Example. Consider $n = 4$ items with comparison matrix A, where rows and columns index items $0, ..., 3$. Ranking all items by identity ($\pi(i) = i$) yields disagreement cost $1 + 2 + 1 + 0 + 2 + 1 = 7$, which is optimal.

```
$ pred create --example MaximumLikelihoodRanking -o mlr.json
$ pred solve mlr.json --solver brute-force
$ pred evaluate mlr.json --config 0,1,2,3
```

	0	1	2	3
0	0	4	3	5
1	1	0	4	3
2	2	1	0	4
3	0	2	1	0

Figure 93: Comparison matrix A ($n = 4$). Highlighted cells are disagreement entries under the identity ranking.

Definition 2.183 (Optimum Communication Spanning Tree): Given a complete graph K_n with edge weights $w(e) \geq 0$ and communication requirements $r(u, v) \geq 0$ for each vertex pair, find a spanning tree T minimizing the total communication cost $\sum_{u<v} r(u, v) \cdot W_T(u, v)$, where $W_T(u, v)$ is the sum of edge weights on the unique path from u to v in T.

- Complexity: `2^num_edges`.
- Reduces to: ILP.

OptimumCommunicationSpanningTree	
num_vertices	Number of vertices n
edge_weights	Symmetric weight matrix w(i,j)
requirements	Symmetric requirement matrix r(i,j)

The Optimum Communication Spanning Tree problem (ND7 in [5]) models communication network design where edge weights represent link costs and requirements represent traffic demands between vertex pairs. NP-hard even when all requirements are equal (reducing to the Minimum Routing Cost Spanning Tree). Polynomial when all edge weights are equal, solved by the Gomory-Hu tree. The best known exact approach enumerates all n^{n-2} labeled spanning trees.

Example. Consider K_4 with edge weight matrix W and requirement matrix R. The optimal spanning tree uses edges $\{(0, 1), (0, 3), (2, 3)\}$ with total communication cost 20.

```
$ pred create --example OptimumCommunicationSpanningTree -o ocst.json
$ pred solve ocst.json --solver brute-force
$ pred evaluate ocst.json --config 1,0,1,0,0,1
```

Definition 2.184 (Square Tiling): Given a set C of colors, a collection $T \subseteq C^4$ of tile types (where $\langle a, b, c, d \rangle$ denotes a tile whose top, right, bottom, and left sides are colored a, b, c, d respectively), and a positive integer N, determine whether there exists an assignment of a tile $f(i, j) \in T$ to each grid cell

[63] No algorithm with a significantly better exponential base than 2^n is known for the general Maximum Likelihood Ranking problem.

(i,j), $0 \le i,j < N$, such that (1) if $f(i,j) = \langle a,b,c,d\rangle$ and $f(i+1,j) = \langle a',b',c',d'\rangle$ then $c = a'$ (bottom of upper tile matches top of lower tile), and (2) if $f(i,j) = \langle a,b,c,d\rangle$ and $f(i,j+1) = \langle a',b',c',d'\rangle$ then $b = d'$ (right of left tile matches left of right tile). Tiles may be reused but not rotated or reflected.

- Complexity: `num_tiles^(grid_size^2)`.

SquareTiling	
num_colors	Number of colors
tiles	Collection of tile types (top, right, bottom, left)
grid_size	Grid dimension N for N x N tiling

Square Tiling (also known as Bounded Wang Tiling) is problem GP13 in Garey and Johnson [5]. It was shown NP-complete via transformation from Directed Hamiltonian Path. The infinite variant (tiling the entire plane) is famously undecidable (Berger, 1966). The best known exact approach enumerates all $|T|^{N^2}$ assignments.

Example. Consider $|C| = 3$ colors, $|T| = 4$ tiles, and grid size $N = 2$. The tiles are $t_0 = \langle 0,1,2,0\rangle$, $t_1 = \langle 0,0,2,1\rangle$, $t_2 = \langle 2,1,0,0\rangle$, $t_3 = \langle 2,0,0,1\rangle$. The witness assignment (0, 1, 2, 3) places t_0, t_1 in row 0 and t_2, t_3 in row 1, satisfying all edge-color constraints.

```
$ pred create --example SquareTiling -o square_tiling.json
$ pred solve square_tiling.json
$ pred evaluate square_tiling.json --config 0,1,2,3
```

3 Reductions

Each reduction is presented as a **Rule** (with linked problem names and overhead from the graph data), followed by a **Proof** (construction, correctness, variable mapping, solution extraction), and optionally a **Concrete Example** (a small instance with verified solution). Problem names in the rule title link back to their definitions in Section 2.

3.1 Trivial Reductions

Rule 3.1: (Minimum Vertex Cover (weighted) $\to$ Maximum Independent Set (weighted)) Vertex cover and independent set are set complements: removing a cover from V leaves vertices with no edges between them (an independent set), and vice versa. Since $|S| + |C| = |V|$ is constant, maximizing one is equivalent to minimizing the other. The reduction preserves the graph and weights unchanged.
Overhead: `num_vertices = num_vertices`, `num_edges = num_edges`.

Proof: *Construction.* Given VC instance $(G, \boldsymbol{w})$, create IS instance $(G, \boldsymbol{w})$ with identical graph and weights. Variables correspond one-to-one: vertex v in the source maps to vertex v in the target.

Correctness. ($\Rightarrow$) If C is a vertex cover, then for any $u, v \in V \setminus C$, the edge $(u,v) \notin E$ (otherwise C would miss it), so $V \setminus C$ is independent. ($\Leftarrow$) If S is independent, then for any $(u,v) \in E$, at most one endpoint lies in S, so $V \setminus S$ covers every edge. Since $|S| + |C| = |V|$ is constant, a minimum vertex cover corresponds to a maximum independent set.

Solution extraction. For IS solution S, return $C = V \setminus S$, i.e.
flip each variable: $c_v = 1 - s_v$. □

Example: Petersen graph ($n = 10$): VC $\leftrightarrow$ IS

Source: MinimumVertexCover **Target:** MaximumIndependentSet

```
$ pred create --example MVC -o mvc.json
$ pred reduce mvc.json --to MaximumIndependentSet/SimpleGraph/i32 -o bundle.json
```

```
$ pred solve bundle.json
$ pred evaluate mvc.json --config 0,1,1,0,1,1,0,0,1,1
```

Source VC: $C = \{1, 2, 4, 5, 8, 9\}$ (size 6) Target IS: $S = \{0, 3, 6, 7\}$ (size 4)
$|\text{VC}| + |\text{IS}| = 10 = |V|$ ✓

Rule 3.2: (Decision Minimum Dominating Set → Min-Max Multicenter) This $O(n + m)$ parameter-setting reduction [5, ND50] keeps the graph unchanged, replaces all vertex weights and edge lengths by 1, and copies the decision budget K into the target center count k. On such unit graphs, a k-center solution of radius at most 1 exists exactly when every vertex is itself chosen or adjacent to a chosen vertex, which is the dominating-set condition.
Overhead: `num_vertices = num_vertices`, `num_edges = num_edges`.

Proof: *Construction.* Given a unit-weight decision dominating-set instance $(G = (V, E), K)$, build a Min-Max Multicenter instance on the same graph G. Set $w(v) = 1$ for every vertex, set $l(e) = 1$ for every edge, and set the number of centers to $k = K$.

Correctness. ($\Rightarrow$) If $D \subseteq V$ is a dominating set with $|D| \leq K$, pad D with arbitrary additional vertices until exactly K centers are chosen. Every vertex is then either a center (distance 0) or adjacent to one (distance 1), so the target maximum weighted distance is at most 1. ($\Leftarrow$) If a set $P \subseteq V$ of exactly K centers has maximum weighted distance at most 1, then every vertex lies at graph distance 0 or 1 from some vertex of P. Hence every vertex is either in P or adjacent to a vertex of P, so P is a dominating set of size K.

Solution extraction. Return the same indicator vector: every chosen target center becomes a chosen source dominating-set vertex. □

Example: 6-vertex unit graph: dominating set of size 2 equals a 2-center of radius 1

Source: DecisionMinimumDominatingSet **Target:** MinMaxMulticenter

```
$ pred create --example DecisionMinimumDominatingSet/SimpleGraph/One -o dmds.json
$ pred reduce dmds.json --to MinMaxMulticenter/SimpleGraph/One -o bundle.json
$ pred solve bundle.json
$ pred evaluate dmds.json --config 1,0,0,1,0,0
```

Step 1 – Source instance. The source graph has vertices $\{0, 1, 2, 3, 4, 5\}$, edges $(0, 1)$, $(0, 2)$, $(1, 3)$, $(2, 3)$, $(3, 4)$, $(3, 5)$, $(4, 5)$, and bound $K = 2$. The stored dominating-set witness is $D = \{0, 3\}$.

Step 2 – Build the target instance. Keep the graph unchanged, assign weight 1 to every vertex, assign length 1 to every edge, and set the number of centers to $k = 2$. The target therefore still has 6 vertices and 7 edges.

Step 3 – Verify a witness. Choosing centers $P = \{0, 3\}$ yields distances $(0, 1, 1, 0, 1, 1)$ to the nearest center, so the maximum weighted distance is 1. The extracted source witness is the same indicator vector, hence a dominating set of size 2 ✓

Rule 3.3: (Decision Minimum Dominating Set → Minimum Sum Multicenter (weighted)) This $O(n + m)$ parameter-setting reduction [5, ND51] keeps the graph unchanged, sets every vertex weight and edge length to 1, copies the decision budget K into the target center count k, and compares the target optimum against $B = |V| - K$. On such unit graphs, every exact-K center placement has total distance at least $n - K$, with equality exactly when every non-center vertex is adjacent to a center.
Overhead: `num_vertices = num_vertices`, `num_edges = num_edges`.

Proof: *Construction.* Given a decision dominating-set instance $(G = (V, E), K)$ with unit vertex weights, build a Minimum Sum Multicenter instance on the same graph G. Set $w(v) = 1$ for every vertex, set $l(e) = 1$ for every edge, and set the number of centers to $k = K$. Let $n = |V|$, and define the decision threshold $B = n - K$ for the target optimum.

Correctness. ($\Rightarrow$) If $D \subseteq V$ is a dominating set with $|D| \leq K$, pad D with arbitrary additional vertices until exactly K centers are chosen. Every chosen center contributes distance 0, and every non-center vertex is adjacent to at least one chosen center, so every non-center contributes distance 1. Hence the total weighted distance is exactly $n - K = B$.

($\Leftarrow$) Suppose a set $P \subseteq V$ of exactly K centers has total weighted distance at most $B = n - K$. Every non-center vertex has distance at least 1 from P, so any exact-K center placement has total distance at least $n - K$. Therefore total distance at most $n - K$ forces equality, meaning every non-center contributes exactly 1. Thus every non-center vertex is adjacent to some center in P, so P is a dominating set of size K.

Solution extraction. Return the same indicator vector: every chosen target center becomes a chosen source dominating-set vertex. □

Example: 6-vertex unit graph: dominating set of size 2 gives total distance 4

Source: DecisionMinimumDominatingSet **Target:** MinimumSumMulticenter

```
$ pred create --example DecisionMinimumDominatingSet/SimpleGraph/One -o dmds.json
$ pred reduce dmds.json --to MinimumSumMulticenter/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate dmds.json --config 1,0,0,1,0,0
```

Step 1 – Source instance. The source graph has vertices $\{0, 1, 2, 3, 4, 5\}$, edges $(0, 1)$, $(0, 2)$, $(1, 3)$, $(2, 3)$, $(3, 4)$, $(3, 5)$, $(4, 5)$, and decision bound $K = 2$. The stored dominating-set witness is $D = \{0, 3\}$.

Step 2 – Build the target instance. Keep the graph unchanged, assign vertex weight 1 everywhere, assign edge length 1 everywhere, and set the target center count to $k = 2$. The comparison threshold is $B = |V| - K = 6 - 2 = 4$.

Step 3 – Verify a witness. Choosing centers $P = \{0, 3\}$ yields distances $(0, 1, 1, 0, 1, 1)$ to the nearest center, so the total weighted distance is $4 = B$. The extracted source witness is the same indicator vector, hence a valid YES witness for the original decision instance ✓

Rule 3.4: (Minimum Vertex Cover → Minimum Maximal Matching) This size-preserving identity map records the forward implication used in the classical NP-hardness proof for Minimum Maximal Matching (equivalently, Minimum Edge Dominating Set) on bounded-degree graphs: every unit-weight vertex cover of G can be greedily converted into a maximal matching of size at most the cover size. The converse loses a factor of two in general, so the edge is documented but intentionally disabled for runtime reduction search. *Overhead:* `num_vertices = num_vertices`, `num_edges = num_edges`.

Proof: *Construction.* Given a unit-weight Minimum Vertex Cover instance $(G = (V, E), K)$, build the Minimum Maximal Matching instance on the same graph G. The target uses one binary variable per source edge, so the graph structure and size fields are unchanged.

Correctness. ($\Rightarrow$) Let $C \subseteq V$ be a vertex cover with $|C| \leq K$. Start with $M = \emptyset$ and process the vertices of C in arbitrary order. Whenever $v \in C$ is unmatched, choose any edge $\{v, u\} \in E$ whose other endpoint u is also unmatched, add that edge to M, and mark both endpoints matched. Because only unmatched endpoints are paired, M is a matching. If some edge $\{x, y\} \in E$ were disjoint from every edge of M at the end, then both x and y would still be unmatched. Since C covers every edge, at least one endpoint, say x, lies in C, and when the algorithm processed x it could have added $\{x, y\}$, a contradiction. Hence M is maximal and $|M| \leq |C| \leq K$.

($\Leftarrow$) Let M be any maximal matching. The set of all endpoints of edges in M is a vertex cover, so $\text{mvc}(G) \leq 2 \cdot |M|$. This yields the standard bound $\text{mmm}(G) \leq \text{mvc}(G) \leq 2 \cdot \text{mmm}(G)$, but it does not recover an exact same-bound inverse. On C_5, the target optimum is 2 while the source optimum is 3.

Solution extraction. No runtime extractor is registered. The endpoint map always returns a valid vertex cover and greedy pruning can shrink it, but neither approach guarantees an optimal cover from an optimal maximal matching witness, and the target optimum value does not determine the source optimum value exactly. □

Example: Cycle C_5: the forward implication is exact, but the backward gap is strict

Source: MinimumVertexCover **Target:** MinimumMaximalMatching

```
$ pred create MinimumVertexCover --graph 0-1,1-2,2-3,3-4,4-0 --weights 1,1,1,1,1 -o mvc.json
$ pred solve mvc.json
$ pred create MinimumMaximalMatching --graph 0-1,1-2,2-3,3-4,4-0 -o mmm.json
$ pred solve mmm.json
```

Step 1 – Shared instance. Both problems use the same 5-cycle, so $n = 5$ and $|E| = 5$.

Step 2 – Source optimum. The canonical minimum vertex cover is $C = \{0, 1, 3\}$, so $\text{mvc}(C_5) = 3 = 3$.

Step 3 – Target optimum. The canonical minimum maximal matching is $M = \{(0, 1), (2, 3)\}$, so $\text{mmm}(C_5) = 2 = 2$.

Step 4 – Backward gap. The endpoint set of M is $\{0, 1, 2, 3\}$, a valid vertex cover of size 4. Pruning can recover an optimal cover of size 3, but not one of size 2, so the same-bound backward implication fails.

Runtime note: This catalog edge is proof-only. The CLI can solve the two instances separately, but runtime reduction search does not traverse this edge because there is no exact witness or aggregate extractor.

Rule 3.5: (Minimum Vertex Cover $\rightarrow$ Longest Common Subsequence) This $O(|V|\ |E|)$ reduction follows D. Maier [99] and the Garey–Johnson entry SR10 [5]. Each source vertex becomes one alphabet symbol, one template string lists the vertices in sorted order, and each source edge contributes a two-block constraint string. The longest common subsequence length equals the maximum independent-set size, so a size-K vertex cover exists exactly when the target LCS has length at least $|V| - K$.
Overhead: `alphabet_size = num_vertices`, `num_strings = num_edges + 1`, `max_length = num_vertices`, `total_length = num_vertices + 2 * num_edges * num_vertices + -1 * 2 * num_edges`.

Proof: *Construction.* Given a unit-weight Minimum Vertex Cover instance $(G = (V, E), K)$ with $V = \{0, 1, ..., n-1\}$, build a Longest Common Subsequence instance as follows. Let $\Sigma = V$. Create the template string $S_0 = (0, 1, ..., n-1)$. For each edge $\{u, v\} \in E$, rename the endpoints so that $u < v$, then add the edge string

$$S_{\{u,v\}} = (0, ..., \hat{u}, ..., n-1) \parallel (0, ..., \hat{v}, ..., n-1),$$

where $\hat{u}$ means that symbol u is omitted. The target string family is $R = \{S_0\} \cup \{S_e : e \in E\}$. Its alphabet size is n, it has $|E| + 1$ strings, `max_length` $= n$, and total input length $n + 2\ |E|\ (n-1)$. Set the target threshold to $K' = n - K$.

Correctness. ($\Rightarrow$) If $C \subseteq V$ is a vertex cover of size K, then $I = V \setminus C$ is an independent set of size $n - K$. List the vertices of I in increasing order. This sequence is a subsequence of S_0 immediately. For an edge string $S_{\{u,v\}}$, at most one endpoint lies in I. If neither endpoint lies in I, every symbol of I appears in both halves, so the subsequence is immediate. If only the larger endpoint v lies in I, then every symbol of I still appears in the first half $(V \setminus \{u\})$, so the same ordered list is a subsequence there. If only the smaller endpoint u lies in I, then all symbols of I smaller than u appear in the first half, while u and every

larger symbol appear in the second half ($V \setminus \{v\}$); concatenating the two halves therefore still contains the sorted list of I as a subsequence. Hence every edge string contains that sequence, so the target LCS has length at least $|I| = n - K = K'$.

($\Leftarrow$) Let w be a common subsequence of the target strings with $|w| \geq K'$. Because w is a subsequence of $S_0 = (0, 1, ..., n-1)$, its symbols are distinct and already appear in increasing order. Consider any edge $\{u, v\}$ with $u < v$. In $S_{\{u,v\}}$, the symbol v appears only in the first half and the symbol u appears only in the second half, so any embedding of both symbols would force v to be matched before u. That contradicts the increasing order forced by S_0. Therefore w contains at most one endpoint of every edge, so its symbols form an independent set I of size at least $K' = n - K$. The complement $V \setminus I$ is then a vertex cover of size at most K.

Solution extraction. Given an LCS witness w, mark every symbol that appears before the padding symbol as "outside the cover" and return its complement: $c_v = 1$ iff $v \notin w$, and $c_v = 0$ iff $v \in w$. □

Example: Path graph P_4: VC → LCS via vertex symbols

Source: MinimumVertexCover **Target:** LongestCommonSubsequence

```
$ pred create --example MinimumVertexCover/SimpleGraph/One -o mvc.json
$ pred reduce mvc.json --to LongestCommonSubsequence -o bundle.json
$ pred solve bundle.json
$ pred evaluate mvc.json --config 0,1,1,0
```

Step 1 – Source instance. Path graph P_4 with $n = 4$ vertices and $|E| = 3$ edges. The canonical minimum vertex cover is $C = \{1, 2\}$.

Step 2 – Construct the LCS instance. Alphabet $\Sigma = \{0, ..., 3\}$ and 4 strings: $S_0 = (0, 1, 2, 3)$, $S_1 = (1, 2, 3, 0, 2, 3)$, $S_2 = (0, 2, 3, 0, 1, 3)$, $S_3 = (0, 1, 3, 0, 1, 2)$. The target has `max_length` 4 and total input length 22.

Step 3 – Verify the witness. The stored target config is (0, 3, 4, 4), so the non-padding common subsequence is $w = (0, 3)$ and the corresponding independent set is $\{0, 3\}$. Taking the complement gives $V \setminus w = \{1, 2\} = C$ ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.6: (Minimum Vertex Cover (weighted) → Minimum Feedback Vertex Set (weighted)) Each undirected edge $\{u, v\}$ can be viewed as the directed 2-cycle $u \to v \to u$. Replacing every source edge this way turns the task "hit every edge with a chosen endpoint" into "hit every directed cycle with a chosen vertex." The vertex set, weights, and budget are preserved, so the reduction is size-preserving up to doubling the edge count into arcs.
Overhead: `num_vertices = num_vertices, num_arcs = 2 * num_edges`.

Proof: *Construction.* Given a Minimum Vertex Cover instance $(G = (V, E), \boldsymbol{w})$, build the directed graph $D = (V, A)$ on the same vertex set, where for every undirected edge $\{u, v\} \in E$ we add both arcs (u, v) and (v, u) to A. Keep the vertex weights unchanged and reuse the same decision variables $x_v \in \{0, 1\}$.

Correctness. ($\Rightarrow$) If $C \subseteq V$ is a vertex cover of G, then every source edge $\{u, v\}$ has an endpoint in C, so the corresponding 2-cycle $u \to v \to u$ in D is hit by C. Any longer directed cycle in D is also made from source edges, so one of its vertices lies in C as well. Therefore removing C destroys all directed cycles, and C is a feedback vertex set of D. ($\Leftarrow$) If $F \subseteq V$ is a feedback vertex set of D, then for every source edge $\{u, v\}$ the digraph contains the 2-cycle $u \to v \to u$, which must be hit by F. Hence at least one of u, v lies in F, so F covers every edge of G and is a vertex cover.

Solution extraction. Return the target solution vector unchanged: a selected vertex in the feedback vertex set is selected in the vertex cover, and vice versa. □

Example: 7-vertex graph: each source edge becomes a directed 2-cycle

Source: MinimumVertexCover **Target:** MinimumFeedbackVertexSet

```
$ pred create --example MVC -o mvc.json
$ pred reduce mvc.json --to MinimumFeedbackVertexSet/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate mvc.json --config 1,1,0,1,0,1,0
```

Source VC: $C = \{0, 1, 3, 5\}$ (size 4) on a graph with $n = 7$ vertices and $|E| = 8$ edges
Target FVS: $F = \{0, 1, 3, 5\}$ (size 4) on a digraph with the same $n = 7$ vertices and $|A| = 16 = 2\,|E|$ arcs
Canonical witness is preserved exactly: $C = F$ ✓

Rule 3.7: (Maximum Independent Set (weighted) → Minimum Vertex Cover (weighted)) The exact reverse of VC → IS: complementing an independent set yields a vertex cover. The graph and weights are preserved unchanged, and $|\text{IS}| + |\text{VC}| = |V|$ ensures optimality carries over.
Overhead: `num_vertices = num_vertices`, `num_edges = num_edges`.

Proof: *Construction.* Given IS instance $(G, \boldsymbol{w})$, create VC instance $(G, \boldsymbol{w})$ with identical graph and weights.

Correctness. ($\Rightarrow$) If S is independent, no edge has both endpoints in S, so every edge has at least one endpoint in $V \setminus S$, making $V \setminus S$ a cover. ($\Leftarrow$) If C is a vertex cover, every edge is incident to some vertex in C, so no edge connects two vertices of $V \setminus C$, making $V \setminus C$ independent.

Solution extraction. For VC solution C, return $S = V \setminus C$, i.e.
flip each variable: $s_v = 1 - c_v$. □

Rule 3.8: (Maximum Independent Set (weighted) → Maximum Clique (weighted)) An independent set in G is exactly a clique in the complement graph $\overline{G}$: vertices with no edges between them in G are pairwise adjacent in $\overline{G}$. Both problems maximize total vertex weight, so optimal values are preserved. This is Karp's classical complement graph reduction.
Overhead: `num_vertices = num_vertices`, `num_edges = num_vertices * (num_vertices + -1 * 1) * 2^-1 + -1 * num_edges`.

Proof: *Construction.* Given IS instance $(G = (V, E), \boldsymbol{w})$, build $\overline{G} = \left(V, \overline{E}\right)$ where $\overline{E} = \{(u, v) : u \neq v, (u, v) \notin E\}$. Create MaxClique instance $\left(\overline{G}, \boldsymbol{w}\right)$ with the same weights. Variables correspond one-to-one: vertex v in the source maps to vertex v in the target.

Correctness. ($\Rightarrow$) If S is independent in G, then for any $u, v \in S$, $(u, v) \notin E$, so $(u, v) \in \overline{E}$ — all pairs in S are adjacent in $\overline{G}$, making S a clique. ($\Leftarrow$) If C is a clique in $\overline{G}$, then for any $u, v \in C$, $(u, v) \in \overline{E}$, so $(u, v) \notin E$ — no pair in C is adjacent in G, making C independent. Weight sums are identical, so optimality is preserved.

Solution extraction. For clique solution C in $\overline{G}$, return IS $= C$ (identity mapping: $s_v = c_v$). □

Example: Path graph P_5: IS → Clique via complement

Source: MaximumIndependentSet **Target:** MaximumClique

```
$ pred create --example MIS -o mis.json
$ pred reduce mis.json --to MaximumClique/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate mis.json --config 1,0,1,0,1
```

Source IS: $S = \{0, 2, 4\}$ (size 3) Target Clique: $C = \{0, 2, 4\}$ (size 3)
Source $|E| = 4$, complement $|\overline{E}| = 6$ ✓

Rule 3.9: (Maximum Independent Set → Maximum Set Packing) The key insight is that two vertices are adjacent if and only if they share an edge. By representing each vertex v as the set of its incident edges S_v, adjacency becomes set overlap: $S_u \cap S_v \neq \emptyset$ iff $(u, v) \in E$. Thus an independent set (no two adjacent) maps exactly to a packing (no two overlapping).
Overhead: `num_sets = num_vertices`, `universe_size = num_edges`.

Proof: *Construction.* Universe $U = E$ (edges, indexed $0, ..., |E| - 1$). For each vertex v, define $S_v = \{e \in E : v \in e\}$ (the set of edge indices incident to v), with weight $w(S_v) = w(v)$. Variables correspond one-to-one: vertex v maps to set S_v.

Correctness. ($\Rightarrow$) If I is independent, then for any $u, v \in I$, edge $(u, v) \notin E$, so $S_u \cap S_v = \emptyset$ — the sets are mutually disjoint, forming a valid packing. ($\Leftarrow$) If $\{S_v : v \in P\}$ is a packing, then for any $u, v \in P$, $S_u \cap S_v = \emptyset$, meaning u and v share no edge, so P is independent. Weight sums are identical, so optimality is preserved.

Solution extraction. For packing $\{S_v : v \in P\}$, return IS $= P$ (same variable assignment). □

Rule 3.10: (Maximum Set Packing → Maximum Independent Set) The *intersection graph* captures set overlap as adjacency: two sets that share an element become neighbors, so a packing (mutually disjoint sets) corresponds exactly to an independent set (mutually non-adjacent vertices). This is the standard reduction from set packing to independent set.
Overhead: `num_vertices = num_sets`, `num_edges = num_sets^2`.

Proof: *Construction.* Build the intersection graph $G' = (V', E')$: create one vertex v_i per set S_i ($i = 1, ..., m$), and add edge (v_i, v_j) iff $S_i \cap S_j \neq \emptyset$. Set $w(v_i) = w(S_i)$. Variables correspond one-to-one: set S_i maps to vertex v_i.

Correctness. ($\Rightarrow$) If $\mathcal{P}$ is a packing (all sets mutually disjoint), then for any $S_i, S_j \in \mathcal{P}$, $S_i \cap S_j = \emptyset$, so $(v_i, v_j) \notin E'$, meaning $\{v_i : S_i \in \mathcal{P}\}$ is independent. ($\Leftarrow$) If $I \subseteq V'$ is independent, then for any $v_i, v_j \in I$, $(v_i, v_j) \notin E'$, so $S_i \cap S_j = \emptyset$, meaning $\{S_i : v_i \in I\}$ is a valid packing. Weight sums match, so optimality is preserved.

Solution extraction. For IS $I \subseteq V'$, return packing $\mathcal{P} = \{S_i : v_i \in I\}$ (same variable assignment). □

Rule 3.11: (Minimum Vertex Cover (weighted) → Minimum Set Covering (weighted)) A vertex cover must "hit" every edge; set covering must "hit" every universe element. By making each edge a universe element and each vertex the set of its incident edges, the two covering conditions become identical. This is the canonical embedding of vertex cover as a special case of set covering.
Overhead: `num_sets = num_vertices`, `universe_size = num_edges`.

Proof: *Construction.* Universe $U = \{0, ..., |E| - 1\}$ (one element per edge). For each vertex v, define $S_v = \{i : e_i \text{ incident to } v\}$ (the indices of edges touching v), with weight $w(S_v) = w(v)$. Variables correspond one-to-one: vertex v maps to set S_v.

Correctness. ($\Rightarrow$) If C is a vertex cover, every edge e_i has at least one endpoint $v \in C$, so $i \in S_v$ for some selected set — hence $\bigcup_{v \in C} S_v = U$, a valid covering. ($\Leftarrow$) If $\{S_v : v \in C\}$ covers U, then every edge index $i \in U$ appears in some S_v with $v \in C$, meaning edge e_i is incident to some $v \in C$ — hence C is a vertex cover. Weight sums are identical, so optimality is preserved.

Solution extraction. For covering $\{S_v : v \in C\}$, return VC $= C$ (same variable assignment). □

Rule 3.12: (Minimum Vertex Cover → Ensemble Computation) This $O(|V| + |E|)$ reduction [5] encodes the unit-weight vertex-cover problem as an ensemble-computation minimization over disjoint unions. A

fresh element a_0 is introduced, and each edge becomes a 3-element target subset. The minimum sequence length equals $K^* + |E|$, where K^* is the minimum vertex cover size.
Overhead: `universe_size = num_vertices + 1`, `num_subsets = num_edges`.

Proof: *Construction.* Given a unit-weight VC instance $G = (V, E)$, let a_0 be a fresh element not in V. Set the universe $A = V \cup \{a_0\}$ with $|A| = |V| + 1$. For each edge $\{u, v\} \in E$, add the subset $\{a_0, u, v\}$ to the collection C. Set the search-space bound $J = |V| + |E|$.

Correctness. ($\Rightarrow$) If C' is a vertex cover of size K, label its elements $v_1, ..., v_K$ and the edges $e_1, ..., e_m$. Since C' covers every edge, each $e_j = \{u_j, v_{r[j]}\}$ where $v_{r[j]} \in C'$. The sequence of $K + m$ operations $z_i = \{a_0\} \cup \{v_i\}$ for $i = 1, ..., K$ followed by $z_{K+j} = \{u_j\} \cup z_{r[j]}$ for $j = 1, ..., m$ produces every target subset in exactly $K + |E|$ steps. ($\Leftarrow$) An exchange argument (Garey & Johnson, PO9) shows that any minimum-length sequence can be normalized to use only $\{a_0\} \cup \{u\}$ and $\{v\} \cup z_k$ forms. Each edge contributes exactly one operation of the second form, so the number of first-form operations equals the sequence length minus $|E|$. Since the first-form vertices must cover all edges, the minimum sequence length is $K^* + |E|$.

Solution extraction. From an optimal witness, collect all vertices appearing as singleton operands (indices $< |V|$). In a minimum-length normalized sequence, exactly the K^* cover vertices appear as $\{a_0\}$-paired singletons. □

Rule 3.13: (Minimum Vertex Cover (weighted) $\rightarrow$ Minimum Weight AND/OR Graph) This reduction encodes vertex cover as a minimum-weight solution subgraph problem on a three-layer AND/OR DAG. The root AND gate requires all edges to be covered; each edge becomes an OR gate selecting which endpoint covers it; and each vertex becomes a sink whose arc weight equals the vertex weight. The minimum-weight solution subgraph selects exactly the arcs corresponding to a minimum vertex cover.
Overhead: `num_vertices = 1 + num_edges + 2 * num_vertices`, `num_arcs = 3 * num_edges + num_vertices`.

Proof: *Construction.* Given a Minimum Vertex Cover instance $(G = (V, E), \boldsymbol{w})$ with $n = |V|$ vertices and $m = |E|$ edges, build an AND/OR graph D with $1 + m + 2n$ vertices arranged in three layers:

- **Root (AND gate):** A single vertex r (index 0) with gate type AND.
- **Edge layer (OR gates):** For each edge $e_i = \{u, v\}$ ($i = 0, ..., m - 1$), create vertex e_i (index $1 + i$) with gate type OR and an arc (r, e_i) of weight 1.
- **Cover layer (OR gates):** For each source vertex v_j ($j = 0, ..., n - 1$), create vertex c_j (index $1 + m + j$) with gate type OR. For each edge $e_i = \{u, v\}$, add arcs (e_i, c_u) and (e_i, c_v), each of weight 1.
- **Sink layer (leaves):** For each source vertex v_j, create leaf s_j (index $1 + m + n + j$) and arc (c_j, s_j) with weight w_j.

Since r is AND, any solution subgraph must include all arcs from r to the edge-layer vertices (cost m). Each edge-OR vertex e_i requires at least one of its two outgoing arcs to c_u and c_v (selecting which endpoint covers edge i). Each activated cover vertex c_j requires its outgoing arc to s_j (contributing w_j). The total weight is $m + |\{\text{activated cover arcs}\}| + \sum_{j \in C} w_j$.

Correctness. ($\Rightarrow$) If $C \subseteq V$ is a vertex cover with weight W, then for each edge $e_i = \{u, v\}$, at least one endpoint lies in C; select the arc from e_i to that endpoint's cover vertex. Activate all cover-to-sink arcs for vertices in C. This satisfies the AND gate at the root (all edge arcs selected), every edge OR gate (at least one child selected), and all activated cover vertices (sink arc selected). The total weight is $m + |\{\text{edge-to-cover arcs}\}| + W$. ($\Leftarrow$) In any valid solution subgraph, the AND root forces all m edge arcs. Each edge OR vertex selects at least one arc to a cover vertex, activating that cover vertex and its sink arc. The set of activated cover vertices forms a vertex cover (every edge has at least one endpoint activated). The sink arc weights sum to the cover weight, so any minimum-weight solution subgraph corresponds to a minimum vertex cover.

Solution extraction. Examine the cover-to-sink arcs (indices $3m, ..., 3m + n - 1$ in the arc list): $c_j = 1$ if arc (c_j, s_j) is selected, $c_j = 0$ otherwise. □

Example: Path P_3: vertex cover $\{1\}$ maps to an AND/OR graph of weight 5

Source: MinimumVertexCover **Target:** MinimumWeightAndOrGraph

```
$ pred create --example MVC -o mvc.json
$ pred reduce mvc.json --to MinimumWeightAndOrGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate mvc.json --config 0,1,0
```

Source VC: $C = \{1\}$ (size 1) on graph with $n = 3$ vertices, $m = 2$ edges
Target AND/OR graph: 9 vertices, source v_0 (AND), arcs: $v_0 \to v_1$, $v_0 \to v_2$, $v_1 \to v_3$, $v_1 \to v_4$, $v_2 \to v_4$, $v_2 \to v_5$, $v_3 \to v_6$, $v_4 \to v_7$, $v_5 \to v_8$
Selected arcs: $v_0 \to v_1$, $v_0 \to v_2$, $v_1 \to v_4$, $v_2 \to v_4$, $v_4 \to v_7$ (weight 5) ✓

Rule 3.14: (Maximum Matching (weighted) → Maximum Set Packing (weighted)) A matching selects edges that share no endpoints; set packing selects sets that share no elements. By representing each edge as the 2-element set of its endpoints and using vertices as the universe, two edges conflict (share an endpoint) if and only if their sets overlap. This embeds matching as a special case of set packing where every set has size exactly 2.
Overhead: `num_sets = num_edges`, `universe_size = num_vertices`.

Proof: *Construction.* Universe $U = V$ (vertices, indexed $0, ..., |V| - 1$). For each edge $e = (u, v)$, define $S_e = \{u, v\}$ with weight $w(S_e) = w(e)$. Variables correspond one-to-one: edge e maps to set S_e.

Correctness. ($\Rightarrow$) If M is a matching, then for any $e_1, e_2 \in M$, the edges share no endpoint, so $S_{e_1} \cap S_{e_2} = \emptyset$ — the sets are mutually disjoint, forming a valid packing. ($\Leftarrow$) If $\{S_e : e \in P\}$ is a packing, then for any $e_1, e_2 \in P$, $S_{e_1} \cap S_{e_2} = \emptyset$, meaning the edges share no vertex, so P is a valid matching. Weight sums are identical, so optimality is preserved.

Solution extraction. For packing $\{S_e : e \in P\}$, return matching $= P$ (same variable assignment). □

Rule 3.15: (QUBO (real-weighted) → Spin Glass (real-weighted)) QUBO uses binary variables $x_i \in \{0, 1\}$; the Ising model uses spin variables $s_i \in \{-1, +1\}$. The affine substitution $x_i = (s_i + 1)/2$ converts between the two encodings. Since every quadratic binary function maps to a quadratic spin function (and vice versa), the two models are polynomially equivalent. This is the reverse of SpinGlass → QUBO.
Overhead: `num_spins = num_vars`.

Proof: *Construction.* Substitute $x_i = (s_i + 1)/2$ into $f(\boldsymbol{x}) = \sum_{i \le j} Q_{ij} x_i x_j$. For diagonal terms ($i = j$): $Q_{ii} x_i = Q_{ii}(s_i + 1)/2$, contributing $Q_{ii}/2$ to h_i. For off-diagonal terms ($i < j$): $Q_{ij} x_i x_j = Q_{ij}(s_i + 1)(s_j + 1)/4$, contributing $Q_{ij}/4$ to J_{ij}, $Q_{ij}/4$ to both h_i and h_j, plus a constant. Collecting terms:

$$J_{ij} = \frac{Q_{ij}}{4}, \quad h_i = \frac{1}{2}\left(Q_{ii} + \sum_{j \neq i} \frac{Q_{ij}}{2}\right)$$

Correctness. ($\Rightarrow$) Any binary assignment $\boldsymbol{x}$ maps to a spin assignment $\boldsymbol{s}$ with $s_i = 2x_i - 1$, and the QUBO objective equals the Ising energy up to a global constant. ($\Leftarrow$) Any spin ground state maps back to a binary minimizer via $x_i = (s_i + 1)/2$. The constant offset does not affect the argmin.

Solution extraction. Convert spins to binary: $x_i = (s_i + 1)/2$, i.e.
$s_i = +1 \to x_i = 1$, $s_i = -1 \to x_i = 0$. □

Rule 3.16: (Spin Glass (real-weighted) → QUBO (real-weighted)) The Ising model and QUBO are both quadratic functions over finite domains: spins $\{-1, +1\}$ and binary variables $\{0, 1\}$, respectively. The affine map $s_i = 2x_i - 1$ establishes a bijection between the two domains and preserves the quadratic structure. Substituting into the Ising Hamiltonian yields a QUBO objective that differs from the original energy by

a constant, so ground states correspond exactly.
Overhead: `num_vars = num_spins`.

Proof: *Construction.* Substitute $s_i = 2x_i - 1$ into $H = -\sum_{i<j} J_{ij} s_i s_j - \sum_i h_i s_i$. Expanding:

$$s_i s_j = (2x_i - 1)(2x_j - 1) = 4x_i x_j - 2x_i - 2x_j + 1$$

Collecting terms and using $x_i^2 = x_i$:

$$Q_{ij} = -4J_{ij} \quad (i < j), \quad Q_{ii} = 2\sum_{j \neq i} J_{ij} - 2h_i$$

The constant offset $-\sum_{i<j} J_{ij} + \sum_i h_i$ does not affect the minimizer.

Correctness. ($\Rightarrow$) Any spin configuration $\boldsymbol{s}$ maps to a unique binary vector $\boldsymbol{x}$ via $x_i = (s_i + 1)/2$, and $H_{\text{SG}(\boldsymbol{s})} = H_{\text{QUBO}(\boldsymbol{x})} + \text{const}$, so a ground state of the Ising model maps to a QUBO minimizer. ($\Leftarrow$) Any QUBO minimizer $\boldsymbol{x}$ maps back to spins $s_i = 2x_i - 1$ with the same energy relationship, so optimality is preserved in both directions.

Solution extraction. Convert binary to spins: $s_i = 2x_i - 1$, i.e.
$x_i = 1 \to s_i = +1$, $x_i = 0 \to s_i = -1$. □

Example: 10-spin Ising model on Petersen graph

Source: SpinGlass **Target:** QUBO

```
$ pred create --example SpinGlass -o spinglass.json
$ pred reduce spinglass.json --to QUBO/f64 -o bundle.json
$ pred solve bundle.json
$ pred evaluate spinglass.json --config 1,0,1,1,1,0,1,0,0,1
```

Source: $n = 10$ spins, $h_i = 0$, couplings $J_{ij} \in \{\pm 1\}$
Mapping: $s_i = 2x_i - 1$ converts spins $\{-1, +1\}$ to binary $\{0, 1\}$
Canonical ground-state witness: $\boldsymbol{x} = (1, 0, 1, 1, 1, 0, 1, 0, 0, 1)$ ✓

Rule 3.17: (Closest Vector Problem (weighted) → QUBO (real-weighted)) A bounded Closest Vector Problem instance already supplies a finite integer box $x_i \in [\ell_i, u_i]$ for each coefficient. Following the direct quadratic-form reduction of Canale, Qureshi, and Viola [143], encoding each offset $c_i = x_i - \ell_i$ with an exact in-range binary basis turns the squared-distance objective into an unconstrained quadratic over binary variables. Unlike penalty-method encodings, no auxiliary feasibility penalty is needed: every bit pattern decodes to a legal coefficient vector by construction.
Overhead: `num_vars = num_encoding_bits`.

Proof: *Construction.* Let $A \in \mathbb{Z}^{m \times n}$ be the basis matrix with columns $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$, let $\boldsymbol{t} \in \mathbb{R}^m$ be the target, and let $x_i \in [\ell_i, u_i]$ with range $r_i = u_i - \ell_i$. Define $L_i = \lceil \log_2(r_i + 1) \rceil$ when $r_i > 0$ and omit bits when $r_i = 0$. For each variable, introduce binary variables $z_{i,0}, \ldots, z_{i,L_i-1}$ with exact-range weights

$$w_{i,p} = 2^p \quad (0 \le p < L_i - 1), \quad w_{i,L_i-1} = r_i + 1 - 2^{L_i - 1}$$

so that every bit vector represents an offset in $\{0, \ldots, r_i\}$. Then

$$x_i = \ell_i + \sum_{p=0}^{L_i-1} w_{i,p} z_{i,p}$$

and the total number of QUBO variables is $N = \sum_i L_i$, exactly the exported overhead `num_vars = num_encoding_bits`.

Let $G = A^\top A$ and $h = A^\top \boldsymbol{t}$. Writing $\boldsymbol{x} = \ell + B\boldsymbol{z}$ for the encoding matrix $B \in \mathbb{R}^{n \times N}$ gives

$$\|A\boldsymbol{x} - \boldsymbol{t}\|_2^2 = \boldsymbol{z}^\top (B^\top G B)\boldsymbol{z} + 2\boldsymbol{z}^\top B^\top (G\ell - h) + \text{const}$$

where the constant $\|A\ell - \boldsymbol{t}\|_2^2$ is dropped. Therefore the QUBO coefficients are

$$Q_{u,u} = (B^\top G B)_{u,u} + 2(B^\top (G\ell - h))_u, \quad Q_{u,v} = 2(B^\top G B)_{u,v} \quad (u < v)$$

using the usual upper-triangular convention.

Correctness. ($\Rightarrow$) Every binary vector $\boldsymbol{z} \in \{0,1\}^N$ decodes to a coefficient vector $\boldsymbol{x}$ inside the prescribed bounds because each exact-range basis reaches only offsets in $\{0, \ldots, r_i\}$. Substituting this decoding into the CVP objective yields $\boldsymbol{z}^\top Q \boldsymbol{z} + \text{const}$, so any QUBO minimizer maps to a bounded CVP minimizer. ($\Leftarrow$) Every bounded CVP solution $\boldsymbol{x}$ has at least one bit encoding for each coordinate offset, hence at least one binary vector $\boldsymbol{z}$ with the same objective value up to the dropped constant. Thus the minimizers correspond exactly, although several binary witnesses may decode to the same CVP solution.

Solution extraction. For each source variable, sum its selected encoding weights to recover the source configuration offset $c_i = x_i - \ell_i$. This is exactly the configuration format expected by the `ClosestVectorProblem` model. □

Example: 2D bounded CVP with two 3-bit exact-range encodings

Source: ClosestVectorProblem **Target:** QUBO

```
$ pred create --example CVP -o cvp.json
$ pred reduce cvp.json --to QUBO/f64 -o bundle.json
$ pred solve bundle.json
$ pred evaluate cvp.json --config 3,3
```

Step 1 – Source instance. The canonical CVP example uses basis columns $\boldsymbol{b}_1 = (2,\, 0)^\top$ and $\boldsymbol{b}_2 = (1,\, 2)^\top$, target $\boldsymbol{t} = (2.8,\, 1.5)^\top$, and bounds $x_1, x_2 \in [-2, 4]$.

Step 2 – Exact bounded encoding. Each variable has 4 - bounds.at(0).lower + 1 admissible values, so the implementation uses the capped binary basis $(1, 2, 3)$ rather than $(1, 2, 4)$: the first two bits are powers of two, and the last weight is capped so every bit pattern reconstructs an offset in $\{0, \ldots, 6\}$. Thus

$$x_1 = -2 + z_0 + 2z_1 + 3z_2, \quad x_2 = -2 + z_3 + 2z_4 + 3z_5$$

giving 6 QUBO variables in total.

Step 3 – Build the QUBO. For this instance, $G = A^\top A = ((4, 2), (2, 5))$ and $h = A^\top \boldsymbol{t} = (5.6, 5.8)^\top$. Expanding the shifted quadratic form yields the exported upper-triangular matrix with representative entries $Q_{0,0} = -31.200000000000003$, $Q_{0,1} = 16$, $Q_{0,2} = 24$, $Q_{2,5} = 36$, and $Q_{5,5} = -73.80000000000001$.

Step 4 – Verify a solution. The fixture stores the canonical witness $\boldsymbol{z} = (0,\, 0,\, 1,\, 0,\, 0,\, 1)$, which extracts to source offsets $\boldsymbol{c} = (3,\, 3)$ and actual lattice coordinates $\boldsymbol{x} = (1,\, 1)$. The QUBO value is $\boldsymbol{z}^\top Q \boldsymbol{z} = -107.4$; adding back the dropped constant 107.69 yields the original squared distance 0.29, so the extracted point is the closest lattice vector ✓.

Multiplicity. Offset 3 has two bit encodings ($(0, 0, 1)$ and $(1, 1, 0)$), so the fixture stores one canonical witness even though the QUBO has multiple optimal binary assignments representing the same CVP solution.

3.2 Penalty-Method QUBO Reductions

The *penalty method* [58], [144] converts a constrained optimization problem into an unconstrained QUBO by adding quadratic penalty terms. Given an objective $\text{obj}(\boldsymbol{x})$ to minimize and constraints $g_k(\boldsymbol{x}) = 0$, construct:

$$f(\boldsymbol{x}) = \text{obj}(\boldsymbol{x}) + P \sum_k g_k(\boldsymbol{x})^2$$

where P is a penalty weight large enough that any constraint violation costs more than the entire objective range. Since $g_k(\boldsymbol{x})^2 \geq 0$ with equality iff $g_k(\boldsymbol{x}) = 0$, minimizers of f are feasible and optimal for the original problem. Because binary variables satisfy $x_i^2 = x_i$, the resulting f is a quadratic in $\boldsymbol{x}$, i.e.
a QUBO.

Rule 3.18: (k-Coloring (k-ary) → QUBO (real-weighted)) The k-coloring problem has two requirements: each vertex gets exactly one color, and adjacent vertices get different colors. Both can be expressed as quadratic penalties over binary variables. Introduce nk binary variables $x_{v,c} \in \{0, 1\}$ (indexed by $v \cdot k + c$), where $x_{v,c} = 1$ means vertex v receives color c. The first requirement becomes a *one-hot constraint* penalizing vertices with zero or multiple colors; the second becomes an *edge conflict penalty* penalizing same-color neighbors. The combined QUBO matrix $Q \in \mathbb{R}^{nk \times nk}$ encodes both penalties.
Overhead: `num_vars = num_vertices^2`.

Proof: *Construction.* Applying the penalty method (Section 3.2), the two requirements translate into two penalty terms:

$$f(\boldsymbol{x}) = \underbrace{P_1 \sum_{v \in V} \left(1 - \sum_{c=1}^{k} x_{v,c}\right)^2}_{\text{one-hot: exactly one color per vertex}} + \underbrace{P_2 \sum_{(u,v) \in E} \sum_{c=1}^{k} x_{u,c} x_{v,c}}_{\text{edge conflict: neighbors differ}}$$

One-hot expansion. The constraint $\left(1 - \sum_c x_{v,c}\right)^2$ penalizes any vertex with $\neq 1$ active color. Expanding using $x_{v,c}^2 = x_{v,c}$ (binary variables):

$$\left(1 - \sum_{c} x_{v,c}\right)^2 = 1 - \sum_{c} x_{v,c} + 2 \sum_{c_1 < c_2} x_{v,c_1} x_{v,c_2}$$

Reading off the QUBO coefficients: diagonal $Q_{vk+c,vk+c} = -P_1$ (favors assigning a color) and intra-vertex off-diagonal $Q_{vk+c_1,vk+c_2} = 2P_1$ for $c_1 < c_2$ (discourages multiple colors).

Edge conflict. For each edge (u, v) and color c, the product $x_{u,c} x_{v,c}$ equals 1 iff both endpoints share color c. The penalty $P_2 x_{u,c} x_{v,c}$ adds P_2 to $Q_{uk+c,vk+c}$ (with appropriate index ordering).

In our implementation, $P_1 = P = 1 + n$ and $P_2 = P/2$. The penalty P_1 exceeds the number of vertices, ensuring that any constraint violation outweighs any objective gain.

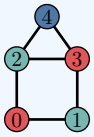
Correctness. ($\Rightarrow$) If $\boldsymbol{x}$ violates any one-hot constraint (some vertex has 0 or ≥ 2 colors), the penalty $P_1 > n$ exceeds the objective range, so $\boldsymbol{x}$ is not a minimizer. ($\Leftarrow$) Among valid one-hot encodings, f reduces to the edge conflict term, minimized when no two adjacent vertices share a color — exactly the k-coloring objective.

Solution extraction. For each vertex v, find c with $x_{v,c} = 1$. □

Example: House graph ($n = 5$, $|E| = 6$, $\chi = 3$) with $k = 3$ colors

Source: KColoring **Target:** QUBO

```
$ pred create --example KColoring/SimpleGraph/KN -o kcoloring.json
$ pred reduce kcoloring.json --to QUBO/f64 -o bundle.json
$ pred solve bundle.json
$ pred evaluate kcoloring.json --config 1,2,2,1,0
```



Step 1 – Encode each color choice as a binary variable. A coloring assigns each vertex one of k colors. To express this in binary, introduce k indicator variables per vertex: $x_{v,c} = 1$ means "vertex v gets color c." For the house graph with $k = 3$, this gives $nk = 5 \times 3 = 15$ QUBO variables:

$$\underbrace{x_{0,0}x_{0,1}x_{0,2}}_{\text{vertex 0}}\ \underbrace{x_{1,0}x_{1,1}x_{1,2}}_{\text{vertex 1}}\ \cdots\ \underbrace{x_{4,0}x_{4,1}x_{4,2}}_{\text{vertex 4}}$$

Step 2 – Penalize invalid color assignments (one-hot constraint). A valid coloring requires each vertex to have *exactly one* color, i.e. $\sum_c x_{v,c} = 1$. The penalty $\left(1 - \sum_c x_{v,c}\right)^2$ equals zero when exactly one variable is 1, and is positive otherwise. Weighted by $P_1 = 1 + n = 6$, this contributes diagonal entries $Q_{vk+c,vk+c} = -6$ and off-diagonal entries $Q_{vk+c_1,vk+c_2} = 12$ between colors of the same vertex. These form the 5×5 diagonal blocks of Q.

Step 3 – Penalize same-color neighbors (edge conflict). For each edge $(u, v) \in E$ and each color c, the product $x_{u,c}x_{v,c} = 1$ iff both endpoints receive color c — exactly the coloring conflict we want to forbid. The penalty $P_2 \cdot x_{u,c}x_{v,c}$ with $P_2 = P_1/2 = 3$ makes such conflicts costly. The house has 6 edges, each contributing 3 color-conflict penalties → 18 off-diagonal entries of value 3 in Q.

Step 4 – Verify a solution. The first valid 3-coloring is $(c_0, ..., c_4) = (1, 2, 2, 1, 0)$, shown in the figure above. The one-hot encoding is $\boldsymbol{x} = (0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0)$. Check: each 3-bit group has exactly one 1 (valid one-hot ✓), and for every edge the two endpoints have different colors (e.g. edge 0–1: colors $1, 2$ ✓).

Multiplicity: The fixture stores one canonical coloring witness. The house graph has $3! \times 3 = 18$ valid colorings overall: the triangle 2–3–4 forces 3 distinct colors ($3! = 6$ permutations), and for each, the base vertices $0, 1$ have exactly 3 compatible ordered pairs.

Rule 3.19: (Maximum Set Packing (real-weighted) → QUBO (real-weighted)) Set packing selects mutually disjoint sets of maximum total weight. Two sets conflict if and only if they share a universe element — the same adjacency structure as an independent set on the *intersection graph*. This reduction builds the intersection graph implicitly and applies the IS penalty method directly: each set becomes a QUBO variable, diagonal entries reward selection, and off-diagonal entries penalize pairs of overlapping sets with a penalty large enough to forbid any overlap.
Overhead: `num_vars = num_sets`.

Proof: *Construction.* Given sets $S_1, ..., S_m$ with weights $w_1, ..., w_m$, introduce binary variables $x_i \in \{0, 1\}$ for each set. Two sets S_i, S_j *conflict* iff $S_i \cap S_j \neq \emptyset$. The packing objective is: maximize $\sum_i w_i x_i$ subject to $x_i x_j = 0$ for every conflicting pair. Applying the penalty method (Section 3.2):

$$f(\boldsymbol{x}) = -\sum_i w_i x_i + P \sum_{S_i \cap S_j \neq \emptyset,\, i<j} x_i x_j$$

with $P = 1 + \sum_i w_i$. The QUBO coefficients are: diagonal $Q_{ii} = -w_i$ (reward for selecting set S_i), off-diagonal $Q_{ij} = P$ for each conflicting pair $i < j$ (penalty for overlap).

Correctness. (⇒) If $\boldsymbol{x}$ encodes a maximum-weight packing, all selected sets are mutually disjoint, so all penalty terms vanish and $f(\boldsymbol{x}) = -\sum_{i \in \mathcal{P}} w_i$. Any assignment selecting overlapping sets incurs penalty $P > \sum_i w_i$, making it suboptimal. (⇐) Among feasible assignments (no overlapping sets selected), the penalty terms vanish and $f(\boldsymbol{x}) = -\sum_{i \in \mathcal{P}} w_i$, minimized exactly when $\mathcal{P}$ is a maximum-weight packing.

Solution extraction. Return $\boldsymbol{x}$ directly — each $x_i = 1$ indicates set S_i is in the packing. □

Rule 3.20: (k-SAT (k-ary) → QUBO (real-weighted)) Each clause in a k-SAT formula is falsified by exactly one assignment to its literals. For $k = 2$, this falsifying pattern is a product of two (possibly complemented) binary variables — already quadratic, so each clause maps directly to QUBO terms. For $k = 3$, the

falsifying pattern $y_1y_2y_3$ is cubic; Rosenberg quadratization replaces the product y_1y_2 with an auxiliary variable a, enforced by a penalty that makes $a \neq y_1y_2$ suboptimal. The total QUBO counts unsatisfied clauses, so minimizers maximize satisfiability.
Overhead: `num_vars = num_vars`.

Proof: **Case $k = 2$.**

Construction. Each 2-literal clause has exactly one falsifying assignment (both literals false). The penalty for that assignment is a quadratic function of x_i, x_j:

Clause	Falsified when	Penalty	QUBO contributions
$x_i \vee x_j$	$x_i = 0, x_j = 0$	$(1 - x_i)(1 - x_j)$	$Q_{ii} -= 1, Q_{jj} -= 1, Q_{ij} += 1$
$\overline{x_i} \vee x_j$	$x_i = 1, x_j = 0$	$x_{i(1-x_j)}$	$Q_{ii} += 1, Q_{ij} -= 1$
$x_i \vee \overline{x_j}$	$x_i = 0, x_j = 1$	$(1 - x_i)x_j$	$Q_{jj} += 1, Q_{ij} -= 1$
$\overline{x_i} \vee \overline{x_j}$	$x_i = 1, x_j = 1$	x_ix_j	$Q_{ij} += 1$

Summing over all clauses, $f(\boldsymbol{x}) = \sum_j \text{penalty}_j(\boldsymbol{x})$ counts falsified clauses.

Correctness. ($\Rightarrow$) Each penalty term is non-negative and equals 1 exactly when its clause is falsified. If $\boldsymbol{x}$ satisfies all clauses, $f(\boldsymbol{x}) = 0$. ($\Leftarrow$) Any minimizer of f achieves the fewest falsified clauses, hence maximizes satisfiability.

Case $k = 3$ (Rosenberg quadratization).

Construction. For each clause $(\ell_1 \vee \ell_2 \vee \ell_3)$, define complement variables $y_i = \overline{\ell_i}$ (so $y_i = x_i$ if the literal is negated, $y_i = 1 - x_i$ if positive). The clause is violated when $y_1y_2y_3 = 1$. This cubic penalty is reduced to quadratic form by introducing an auxiliary variable a and the substitution $a = y_1y_2$, enforced via a Rosenberg penalty with weight M:

$$H = a \cdot y_3 + M(y_1y_2 - 2y_1a - 2y_2a + 3a)$$

where $M = 2$ suffices. Each clause adds one auxiliary variable (indices $n, n+1, ..., n+m-1$), so the total QUBO has $n + m$ variables.

Correctness. ($\Rightarrow$) If $a = y_1y_2$, the Rosenberg penalty term vanishes and $H = y_1y_2y_3$ counts the clause violation faithfully. ($\Leftarrow$) If $a \neq y_1y_2$, the penalty $M(\cdots) \geq 1$ strictly exceeds the clause-counting contribution (at most 1), so any minimizer must have $a = y_1y_2$ for every clause. Among such assignments, H counts unsatisfied clauses, and minimizers maximize satisfiability.

Solution extraction. Discard auxiliary variables: return $\boldsymbol{x}[0..n]$. □

Rule 3.21: (*k*-SAT (*k*-ary) → Quadratic Congruences) Manders and Adleman's number-theoretic reduction encodes a 3-SAT assignment as a pattern of signs $\alpha_j \in \{-1, +1\}$ in a bounded knapsack-style congruence, then uses carefully chosen prime powers and the Chinese Remainder Theorem to realize those signs as divisibility conditions on $H - x$ and $H + x$. Squaring removes the sign ambiguity and yields one quadratic congruence $x^2 \equiv a \bmod b$ together with an explicit bound $x < c$. The bound is essential: without it, the composite-modulus quadratic residuosity problem becomes much easier once the factorization of b is known.
Overhead: `bit_length_a = (num_vars + num_clauses)^2 * log(num_vars + num_clauses + 1)`, `bit_length_b = (num_vars + num_clauses)^2 * log(num_vars + num_clauses + 1)`, `bit_length_c = (num_vars + num_clauses)^2 * log(num_vars + num_clauses + 1)`.

Proof: *Construction.* Given a 3-CNF formula φ with n variables, first deduplicate clauses and restrict to the active variables. Enumerate all signed 3-clauses over those variables as $\sigma_1, ..., \sigma_M$. Define the base-8 clause weight 8^j for σ_j, form $\tau_\varphi = -\sum_{\sigma_j \in \varphi} 8^j$, and for each variable compute the positive and negative occurrence sums f_i^+ and f_i^-. In doubled form, set $N = 2M + l$ and coefficients

$$d_0 = 2, \quad d_{2k-1} = -8^k, \quad d_{2k} = -2 \cdot 8^k, \quad d_{2M+i} = f_i^+ - f_i^-$$

together with

$$\tau_2 = 2\tau_\varphi + \sum_{j=0}^{N} d_j + 2\sum_{i=1}^{l} f_i^-$$

modulo $2 \cdot 8^{M+1}$.

Choose distinct odd primes $p_0, \ldots, p_N \geq 13$ and let $K = \prod_{j=0}^{N} p_j^{N+1}$. For each j, construct θ_j so that

$$\theta_j \equiv d_j \bmod 2 \cdot 8^{M+1}, \quad \theta_j \equiv 0 \bmod \prod_{i \neq j} p_i^{N+1}$$

and p_j does not divide θ_j. Set $H = \sum_j \theta_j$, $b = 2 \cdot 8^{M+1} \cdot K$, and

$$a = (2 \cdot 8^{M+1} + K)^{-1}(K\tau_2^2 + 2 \cdot 8^{M+1} H^2) \bmod b, \quad c = H + 1.$$

Correctness. ($\Rightarrow$) A satisfying assignment determines signs $\alpha_j \in \{-1, +1\}$ for the lifted knapsack system so that $x = \sum_j \alpha_j \theta_j$ obeys both $x \equiv \tau_2 \bmod 2 \cdot 8^{M+1}$ and $(H + x)(H - x) \equiv 0 \bmod K$. These together imply $x^2 \equiv a \bmod b$ with $0 \leq x \leq H < c$. ($\Leftarrow$) Any witness $x < c$ with $x^2 \equiv a \bmod b$ yields, for each j, a unique sign from whether p_j^{N+1} divides $H - x$ or $H + x$. Those signs recover an exact knapsack solution and hence a satisfying assignment of the original 3-SAT instance.

Solution extraction. Recover each sign α_j from the divisibility of $H - x$ and $H + x$ by p_j^{N+1}. For variable coordinates $j = 2M + i$, interpret $\alpha_j = -1$ as $x_i = 1$ and $\alpha_j = +1$ as $x_i = 0$. □

Example: 3-SAT with $n = 3$ variables and $m = 1$ clause mapped to a quadratic congruence with a 669-digit modulus

Source: KSatisfiability **Target:** QuadraticCongruences

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to QuadraticCongruences -o bundle.json
$ pred evaluate ksat.json --config 1,0,0
```

Step 1 – Source instance. The canonical formula is the single clause $(x_1 \vee x_2 \vee x_3)$, witnessed by the satisfying assignment (1, 0, 0).

Step 2 – Enumerate standard clauses. With $l = 3$ active variables, the construction lists all $M = 8$ signed 3-clauses on $\{x_1, x_2, x_3\}$. This yields $N = 2M + l = 19$ lifted coefficients in the doubled knapsack encoding.

Step 3 – Lift by CRT. Using $N + 1 = 20$ odd primes starting at 13, the reduction builds the CRT gadgets $\theta_0, \ldots, \theta_N$ and outputs (a, b, c). In the canonical fixture these numbers have 668, 669, and 646 decimal digits respectively, so the paper reports their sizes rather than expanding them inline.

Step 4 – Verify the stored witness. The example DB keeps the target witness in binary using 2144 bits. Evaluating that witness satisfies $x^2 \equiv a \bmod b$ with $1 \leq x < c$, and extraction recovers the original source assignment (1, 0, 0) ✓.

Multiplicity: The fixture stores one canonical satisfying witness.

Rule 3.22: (*k*-SAT (*k*-ary) → Quadratic Diophantine Equations) This reduction chains through the Manders–Adleman quadratic congruence construction. Given a 3-SAT instance φ, first reduce to a Quadratic Congruences instance (a, b, c) with $x^2 \equiv a \bmod b$ and $x < c$, then convert the bounded congruence into a Diophantine equation $x^2 + b'y = c'$ with $a' = 1$. The conversion exploits the fact that $x < c$ implies x^2 is bounded, so the residue $c' - x^2$ is always positive and divisible by b' precisely when the congruence holds. *Overhead:* `bit_length_a = 1`, `bit_length_b = (num_vars + num_clauses)^2 * log(num_vars + num_clauses + 1)`, `bit_length_c = (num_vars + num_clauses)^2 * log(num_vars + num_clauses + 1)`.

Proof: *Construction.* Given a 3-CNF formula φ with n variables and m clauses:

Step 1. Apply the Manders–Adleman reduction (KSatisfiability → QuadraticCongruences) to obtain (a, b, c) such that φ is satisfiable iff $\exists x < c : x^2 \equiv a \bmod b$.

Step 2. Convert to a Diophantine equation. Let $h = c - 1$. Compute a padding value $p = \lfloor (h^2 - a)/b \rfloor + 1$ and set $c' = a + b \cdot p$. Output the Diophantine equation $x^2 + by = c'$ (i.e., $a' = 1$, $b' = b$). A positive integer x with $x^2 + by = c'$ must satisfy $y = (c' - x^2)/b > 0$, which requires $1 \leq x \leq h = c - 1$.

Correctness. ($\Rightarrow$) If φ is satisfiable, the congruence has a witness $x_0 < c$ with $x_0^2 \equiv a \bmod b$. Then $x_0^2 - a = bk$ for some non-negative integer k. Since $c' = a + bp$ and $x_0 \leq c - 1 = h$, we have $c' - x_0^2 = b(p - k) > 0$, and $y = p - k$ is a positive integer. So (x_0, y) is a solution to $x^2 + by = c'$. ($\Leftarrow$) If (x, y) satisfies $x^2 + by = c'$ with $x, y \geq 1$, then $x^2 = c' - by \equiv c' \bmod b \equiv a \bmod b$ (since $c' = a + bp$). Also $x^2 < c' = a + bp \leq h^2 + b$, and since $y \geq 1$ we have $x^2 = c' - by \leq c' - b < h^2 + b - b = h^2$, so $x \leq h < c$. Thus x is a valid congruence witness, and the original formula is satisfiable.

Solution extraction. Decode the Diophantine witness x from its little-endian binary encoding. Then extract a 3-SAT assignment by passing x through the congruence-to-SAT extraction (sign recovery from divisibility by prime powers). □

Rule 3.23: (*k*-SAT (*k*-ary) → Subset Sum) Base-10 digit encoding reduction following Sipser [145, Thm 7.56] and CLRS [146, §34.5.5]. (Karp [1] established SubsetSum NP-completeness via Exact Cover; this direct 3-SAT construction is a later textbook formulation.) Each integer has $(n + m)$ digits, where the first n positions correspond to variables and the last m to clauses. For variable x_i, two integers y_i, z_i encode positive and negative literal occurrences. For clause C_j, slack integers g_j, h_j pad the clause digit to exactly 4. Since each clause has at most 3 literals and slacks add at most 2, no digit exceeds 5, so no carries occur.
Overhead: `num_elements = 2 * num_vars + 2 * num_clauses`.

Proof: *Construction.* Given a 3-CNF formula φ with n variables and m clauses, create $2n + 2m$ integers in $(n + m)$-digit base-10 representation:

(i) *Variable integers* ($2n$): For each x_i, create y_i with $d_i = 1$ and $d_{n+j} = 1$ if $x_i \in C_j$, and z_i with $d_i = 1$ and $d_{n+j} = 1$ if $\overline{x_i} \in C_j$.

(ii) *Slack integers* ($2m$): For each clause C_j, create g_j with $d_{n+j} = 1$ and h_j with $d_{n+j} = 2$.

(iii) *Target* T: $d_i = 1$ for $i \in [1, n]$ and $d_{n+j} = 4$ for $j \in [1, m]$.

Correctness. ($\Rightarrow$) If assignment α satisfies φ, select y_i when $x_i = \top$ and z_i when $x_i = \bot$. Variable digits sum to exactly 1 (one of y_i, z_i per variable). Each satisfied clause has 1–3 true literals contributing 1–3 to its digit; slacks g_j, h_j with values 1, 2 can pad any value in $\{1, 2, 3\}$ to 4. ($\Leftarrow$) Variable digits force exactly one of y_i, z_i per variable, defining a truth assignment. Clause digits reach 4 only if the literal contribution is ≥ 1, meaning each clause is satisfied.

Solution extraction. For each i: if y_i is selected ($x_{2i} = 1$), set $x_i = 1$; if z_i is selected ($x_{2i+1} = 1$), set $x_i = 0$. □

Example: 3-SAT with 3 variables and 2 clauses

Source: KSatisfiability **Target:** SubsetSum

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to SubsetSum -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 0,0,1
```

Source: $n = 3$ variables, $m = 2$ clauses
Target: 10 elements, target = 11144
Source config: `(0, 0, 1)` Target config: `(0, 1, 0, 1, 1, 0, 1, 1, 1, 0)`

Rule 3.24: (Subset Sum → Closest Vector Problem (weighted)) Classical lattice embedding for Subset Sum following Lagarias and Odlyzko [147], with the $\frac{1}{2}$-target CVP formulation in the style of Coster et al. [148]. For an instance with n elements, the reduction produces n basis vectors in ambient dimension $n + 1$: the first n coordinates enforce binary structure and the last coordinate records the subset sum error. *Overhead:* `ambient_dimension = num_elements + 1`, `num_basis_vectors = num_elements`.

Proof: *Construction.* Given sizes $s_0, ..., s_{n-1} \in \mathbb{Z}^+$ and target $B \in \mathbb{Z}^+$, define one basis vector per element:

$$\boldsymbol{b}_i = \boldsymbol{e}_i + s_i \boldsymbol{e}_{n+1}$$

for $i \in \{0, ..., n-1\}$. Equivalently, the basis matrix has columns $\boldsymbol{b}_0, ..., \boldsymbol{b}_{n-1}$, so its first n rows form the identity matrix and its last row is $(s_0, ..., s_{n-1})$. Set the target vector to

$$\boldsymbol{t} = \left(\frac{1}{2}, ..., \frac{1}{2}, B\right)^\top$$

and restrict every CVP variable to $x_i \in \{0, 1\}$.

Correctness. ($\Rightarrow$) If $\boldsymbol{x} \in \{0,1\}^n$ is a satisfying Subset Sum solution, then $\sum_i s_i x_i = B$ and

$$\|\boldsymbol{B}\boldsymbol{x} - \boldsymbol{t}\|_2^2 = \sum_{i=0}^{n-1} \left(x_i - \frac{1}{2}\right)^2 + \left(\sum_i s_i x_i - B\right)^2 = \frac{n}{4}.$$

Hence every satisfying subset becomes a CVP solution at distance $\sqrt{\frac{n}{4}}$. ($\Leftarrow$) Conversely, binary bounds force every CVP candidate to lie in $\{0,1\}^n$. The first n coordinates always contribute exactly $\frac{n}{4}$ to the squared distance, so a CVP minimizer attains distance $\sqrt{\frac{n}{4}}$ if and only if the last coordinate contributes 0, i.e. $\sum_i s_i x_i = B$. When the Subset Sum instance is unsatisfiable, every binary vector has strictly larger distance.

Solution extraction. Return the binary CVP vector unchanged. □

Example: 4 elements, target sum $B = 11$

Source: SubsetSum **Target:** ClosestVectorProblem

```
$ pred create --example SubsetSum -o subsetsum.json
$ pred reduce subsetsum.json --to ClosestVectorProblem/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate subsetsum.json --config 1,0,0,1
```

Step 1 – Source instance. The canonical Subset Sum instance has sizes (3, 7, 1, 8) and target $B = 11$.

Step 2 – Build the lattice. The reduction creates the basis

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 3 \\ 0 & 1 & 0 & 0 & 7 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 8 \end{pmatrix}$$

together with target

$$\boldsymbol{t} = (0.5, 0.5, 0.5, 0.5, 11)^\top$$

and binary bounds $x_i \in \{0, 1\}$ for all 4 coordinates.

Step 3 – Verify the canonical witness. The fixture stores $\boldsymbol{x} = (1, 0, 0, 1)$, which selects sizes 3 and 8 and therefore satisfies $3 + 8 = 11$. Since $\boldsymbol{Bx} = (1, 0, 0, 1, 11)^\top$, the difference vector is $(0.5, -0.5, -0.5, 0.5, 0)^\top$ and the Euclidean distance is $\sqrt{\frac{4}{4}} = 1$.

Witness semantics. The example DB stores one canonical minimizer. This source instance also has another satisfying subset, $(1, 1, 1, 0)$, so the reduction has multiple optimal CVP witnesses even though only one is serialized.

Rule 3.25: (Subset Sum → Capacity Assignment) This $O(n)$ reduction from Subset Sum to Capacity Assignment follows the original NP-completeness proof of Van Sickle and Chandy [117] (GJ SR7 [5]). Each element becomes a communication link with two capacity levels; the cost/delay duality encodes complementary subset selection.
Overhead: `num_links = num_elements`, `num_capacities = 2`.

Proof: *Construction.* Given sizes $a_1, ..., a_n \in \mathbb{Z}^+$ and target B, let $S = \sum_{i=1}^{n} a_i$. Create n communication links with capacity set $M = \{1, 2\}$. For each link c_i:

- Cost: $g(c_i, 1) = 0$, $g(c_i, 2) = a_i$ (non-decreasing since $0 \leq a_i$).
- Delay: $d(c_i, 1) = a_i$, $d(c_i, 2) = 0$ (non-increasing since $a_i \geq 0$).

Set the delay budget $J = S - B$.

Correctness. For any assignment σ, the total cost is $\sum_{i:\sigma(c_i)=2} a_i$ and the total delay is $\sum_{i:\sigma(c_i)=1} a_i$. Since every element contributes to exactly one of these sums, cost + delay $= S$.

($\Rightarrow$) If $A' \subseteq A$ sums to B, assign $\sigma(c_i) = 2$ for $a_i \in A'$ and $\sigma(c_i) = 1$ otherwise. Total cost $= B$, total delay $= S - B = J$.

($\Leftarrow$) The delay constraint forces delay $\leq S - B$, so cost $\geq S - (S - B) = B$. If the optimal cost equals B, the high-capacity links form a subset summing to exactly B. If no such subset exists, the minimum cost is strictly greater than B.

Solution extraction. Return the target configuration unchanged: capacity index 1 (high) for link c_i means element a_i is selected. □

Example: 6 elements, target sum $B = 11$

Source: SubsetSum **Target:** CapacityAssignment

```
$ pred create --example SubsetSum -o subsetsum.json
$ pred reduce subsetsum.json --to CapacityAssignment -o bundle.json
$ pred solve bundle.json
$ pred evaluate subsetsum.json --config 1,0,0,1,0,0
```

Step 1 – Source instance. The canonical Subset Sum instance has sizes (3, 7, 1, 8, 2, 4) and target $B = 11$. The total sum is $S = 25$.

Step 2 – Build the Capacity Assignment instance. The reduction creates 6 communication links with two capacities $\{1, 2\}$. For each link c_i with element value a_i: cost row $(0, a_i)$ and delay row $(a_i, 0)$. The delay budget is $J = S - B = 25 - 11 = 14$.

Step 3 – Verify the canonical witness. The fixture stores source config (1, 0, 0, 1, 0, 0), selecting elements at indices 0, 3 with values $3 + 8 = 11 = B$. In the target, these links get high capacity (index 1) with total cost 11 and the remaining links get low capacity (index 0) with total delay $14 \leq 14 = J$.

Witness semantics. The example DB stores one canonical witness. Other subsets summing to B would also yield valid witnesses.

Rule 3.26: (Integer Linear Programming → QUBO (real-weighted)) A binary ILP optimizes a linear objective over binary variables subject to linear constraints. The penalty method converts each equality constraint $\boldsymbol{a}_k^\top \boldsymbol{x} = b_k$ into the quadratic penalty $\left(\boldsymbol{a}_k^\top \boldsymbol{x} - b_k\right)^2$, which is zero if and only if the constraint is satisfied. Inequality constraints are first converted to equalities using binary slack variables with powers-of-two coefficients. The resulting unconstrained quadratic over binary variables is a QUBO whose matrix Q combines the negated objective (as diagonal terms) with the expanded constraint penalties (as a Gram matrix $A^\top A$).
Overhead: `num_vars = num_vars + num_constraints * num_vars`.

Proof: *Construction.* First, normalize all constraints to equalities. Inequalities $\boldsymbol{a}_k^\top \boldsymbol{x} \leq b_k$ become $\boldsymbol{a}_k^\top \boldsymbol{x} + \sum_{s=0}^{S_k-1} 2^s y_{k,s} = b_k$ where $S_k = \lceil \log_2(b_k + 1) \rceil$ binary slack bits. For $\geq$ constraints, the slack has a negative sign. The extended system is $A'\boldsymbol{x}' = \boldsymbol{b}$ with $\boldsymbol{x}' = (\boldsymbol{x}, \boldsymbol{y}) \in \{0,1\}^{n'}$. For minimization, negate $\boldsymbol{c}$ to convert to maximization.

Applying the penalty method (Section 3.2), combine the negated objective with quadratic constraint penalties:

$$f(\boldsymbol{x}') = -\boldsymbol{c}'^\top \boldsymbol{x}' + P \sum_{k=1}^{m} \left(\boldsymbol{a}_k'^\top \boldsymbol{x}' - b_k\right)^2$$

where $\boldsymbol{c}' = (\boldsymbol{c}, \boldsymbol{0})$ and $P = 1 + \|\boldsymbol{c}\|_1 + \|\boldsymbol{b}\|_1$. Expanding the quadratic penalty:

$$\sum_k \left(\boldsymbol{a}_k'^\top \boldsymbol{x}' - b_k\right)^2 = \boldsymbol{x}'^\top A'^\top A' \boldsymbol{x}' - 2\boldsymbol{b}^\top A' \boldsymbol{x}' + \|\boldsymbol{b}\|_2^2$$

Combining with $-\boldsymbol{c}'^\top \boldsymbol{x}'$ and dropping the constant $\|\boldsymbol{b}\|_2^2$:

$$Q = -\mathrm{diag}(\boldsymbol{c}' + 2P\boldsymbol{b}^\top A') + PA'^\top A'$$

The diagonal contains linear terms (objective plus constraint); the upper triangle of $A'^\top A'$ gives quadratic cross-terms.

Correctness. ($\Rightarrow$) If $\boldsymbol{x}'^*$ is an optimal ILP solution, then $A'\boldsymbol{x}'^* = \boldsymbol{b}$ and all penalty terms vanish, so $f(\boldsymbol{x}'^*) = -\boldsymbol{c}'^\top \boldsymbol{x}'^*$. ($\Leftarrow$) If any constraint is violated, $\left(\boldsymbol{a}_k'^\top \boldsymbol{x}' - b_k\right)^2 \geq 1$ and the penalty $P > \|\boldsymbol{c}\|_1$ exceeds the entire objective range, so $\boldsymbol{x}'$ cannot be a QUBO minimizer. Among feasible assignments (all penalties zero), f reduces to $-\boldsymbol{c}'^\top \boldsymbol{x}'$, minimized at the ILP optimum.

Solution extraction. Discard slack variables: return $\boldsymbol{x}'[0..n]$. □

Rule 3.27: (Partition → Cosine Product Integration) This $O(n)$ identity reduction casts each positive integer size s_i to the corresponding integer coefficient $a_i = s_i$. A balanced partition (two subsets of equal sum) exists if and only if a balanced sign assignment ($\sum \varepsilon_i a_i = 0$) exists, because assigning element i to subset A' corresponds to $\varepsilon_i = -1$ and to $A \setminus A'$ corresponds to $\varepsilon_i = +1$. Reference: Plaisted (1976) [102].
Overhead: `num_coefficients = num_elements`.

Proof: *Construction.* Given Partition sizes $s_0, ..., s_{n-1} \in \mathbb{Z}^+$, set the CosineProductIntegration coefficients to $a_i = s_i$ for each $i \in \{0, ..., n-1\}$.

Correctness. ($\Rightarrow$) If a balanced partition exists with subset A' having $\sum_{a \in A'} s(a) = S/2$, then the sign assignment $\varepsilon_i = -1$ for $i \in A'$ and $\varepsilon_i = +1$ otherwise gives $\sum \varepsilon_i a_i = S - 2 \cdot S/2 = 0$. ($\Leftarrow$) If a balanced sign assignment exists with $\sum \varepsilon_i a_i = 0$, the elements with $\varepsilon_i = -1$ form a subset summing to $S/2$, which is a valid partition.

Solution extraction. Return the same binary vector: $x_i = 1$ (element in second subset) corresponds to $\varepsilon_i = -1$ (negative sign). □

> **Example: 6 elements**
>
> **Source:** Partition **Target:** CosineProductIntegration

Rule 3.28: (Partition → Knapsack) This $O(n)$ reduction[64] [5, MP9] constructs a 0-1 Knapsack instance by copying each Partition size into both the item weight and item value and setting the capacity to half the total size sum. For n source elements it produces n knapsack items.
Overhead: `num_items = num_elements`.

Proof: *Construction.* Given positive sizes $s_0, ..., s_{n-1}$ with total sum $S = \sum_{i=0}^{n-1} s_i$, create one knapsack item per element and set

$$w_i = s_i, \quad v_i = s_i$$

for every $i \in \{0, ..., n-1\}$. Set the knapsack capacity to

$$C = \left\lfloor \frac{S}{2} \right\rfloor.$$

Every feasible knapsack solution is therefore a subset of the original elements, and because $w_i = v_i$, its objective value equals the same subset sum.

Correctness. ($\Rightarrow$) If the Partition instance is satisfiable, some subset A' has sum $\frac{S}{2}$. In particular S is even, so $C = \frac{S}{2}$, and selecting exactly the corresponding knapsack items is feasible with value $\frac{S}{2}$. No feasible knapsack solution can have value larger than C, because value equals weight for every item and total weight is bounded by C. Thus the knapsack optimum is exactly $\frac{S}{2}$. ($\Leftarrow$) If the knapsack optimum is $\frac{S}{2}$, then the optimum is an integer and hence S must be even. The selected items have total value $\frac{S}{2}$, so they also have total weight $\frac{S}{2}$ because $w_i = v_i$ itemwise. Those items therefore form a subset of the original multiset whose complement has the same sum, giving a valid balanced partition.

Solution extraction. Return the same binary selection vector on the original elements: item i is selected in the knapsack witness if and only if element i belongs to the extracted partition subset. □

Example: 6 elements, total sum $S = 10$

Source: Partition **Target:** Knapsack

```
$ pred create --example Partition -o partition.json
$ pred reduce partition.json --to Knapsack -o bundle.json
$ pred solve bundle.json
$ pred evaluate partition.json --config 1,0,0,1,0,0
```

Step 1 – Source instance. The canonical Partition instance has sizes (3, 1, 1, 2, 2, 1) with total sum $S = 10$, so a balanced witness must hit exactly $\frac{S}{2} = 5$.

Step 2 – Build the knapsack instance. The reduction copies each size into both the weight and the value list, producing weights (3, 1, 1, 2, 2, 1), values (3, 1, 1, 2, 2, 1), and capacity $C = 5$. No auxiliary variables are introduced, so the target has the same 6 binary coordinates as the source.

Step 3 – Verify the canonical witness. The serialized witness uses the same binary vector on both sides, $\boldsymbol{x} = (1, 0, 0, 1, 0, 0)$. It selects elements at indices $\{0, 3\}$ with sizes (3, 2), so the chosen subset has total weight and value $5 = 5$. Hence the knapsack solution saturates the capacity and certifies a balanced partition.

Witness semantics. The example DB stores one canonical balanced subset. This instance has multiple balanced partitions because several different subsets sum to 5, but one witness is enough to demonstrate the reduction.

Rule 3.29: (Partition → Subset Sum) This $O(n)$ reduction [1], [5, SP13] embeds a Partition instance into Subset Sum by copying the element sizes and setting the target to half the total sum. For n source elements

[64] The linear-time bound follows from a single pass that copies the source sizes into item weights and values.

it produces n Subset Sum items.
Overhead: `num_elements = num_elements`.

Proof: *Construction.* Given positive sizes $s_0, \ldots, s_{n-1}$ with total sum $S = \sum_{i=0}^{n-1} s_i$, construct a Subset Sum instance with the same sizes and target

$$B = \frac{S}{2}.$$

If S is odd, return a trivially infeasible Subset Sum instance (empty sizes, target $= 1$).

Correctness. ($\Rightarrow$) If the Partition instance is satisfiable, some subset A' has sum $\frac{S}{2} = B$, so the Subset Sum instance is satisfiable. ($\Leftarrow$) If the Subset Sum instance is satisfiable, some subset sums to $B = \frac{S}{2}$, so its complement sums to $S - \frac{S}{2} = \frac{S}{2}$, giving a balanced partition. When S is odd, $\frac{S}{2}$ is not an integer and no subset of positive integers can sum to it; the trivially infeasible target instance correctly reflects this.

Solution extraction. Return the same binary selection vector: element i is in the partition subset if and only if it is selected in the Subset Sum witness. □

Example: 6 elements, total sum $S = 10$

Source: Partition **Target:** SubsetSum

```
$ pred create --example Partition -o partition.json
$ pred reduce partition.json --to SubsetSum -o bundle.json
$ pred solve bundle.json
$ pred evaluate partition.json --config 1,0,0,1,0,0
```

Step 1 – Source instance. The canonical Partition instance has sizes (3, 1, 1, 2, 2, 1) with total sum $S = 10$, so a balanced witness must hit exactly $\frac{S}{2} = 5$.

Step 2 – Build the Subset Sum instance. The reduction copies the sizes directly: (3, 1, 1, 2, 2, 1), and sets the target $B = \frac{S}{2} = 5$. The number of binary variables is unchanged ($n = 6$).

Step 3 – Verify the canonical witness. The serialized witness uses the same binary vector on both sides, $\boldsymbol{x} = (1, 0, 0, 1, 0, 0)$. It selects elements at indices $\{0, 3\}$ with sizes (3, 2), so the chosen subset sums to $5 = 5 = B$ ✓.

Witness semantics. The example DB stores one canonical balanced subset. Multiple subsets may sum to B, but one witness suffices to demonstrate the reduction.

Rule 3.30: (Knapsack → QUBO (real-weighted)) For a standard 0-1 Knapsack instance with nonnegative weights, nonnegative values, and nonnegative capacity, the inequality $\sum_i w_i x_i \leq C$ is converted to equality using binary slack variables that encode the unused capacity. When $C > 0$, one can take $B = \lfloor \log_2 C \rfloor + 1$ slack bits; when $C = 0$, a single slack bit also suffices. The penalty method (Section 3.2) combines the negated value objective with a quadratic constraint penalty, producing a QUBO with $n + B$ binary variables.
Overhead: `num_vars = num_items + num_slack_bits`.

Proof: *Construction.* Given n items with nonnegative weights $w_0, \ldots, w_{n-1}$, nonnegative values $v_0, \ldots, v_{n-1}$, and nonnegative capacity C, introduce $B = \lfloor \log_2 C \rfloor + 1$ binary slack variables $s_0, \ldots, s_{B-1}$ when $C > 0$ (or one slack bit when $C = 0$) to convert the capacity inequality to equality:

$$\sum_{i=0}^{n-1} w_i x_i + \sum_{j=0}^{B-1} 2^j s_j = C$$

Let a_k denote the constraint coefficient of the k-th binary variable ($a_k = w_k$ for $k < n$, $a_{n+j} = 2^j$ for $j < B$). The QUBO objective is:

$$f(\boldsymbol{z}) = -\sum_{i=0}^{n-1} v_i x_i + P\left(\sum_k a_k z_k - C\right)^2$$

where $\boldsymbol{z} = (x_0, ..., x_{n-1}, s_0, ..., s_{B-1})$ and $P = 1 + \sum_i v_i$. Expanding the quadratic penalty using $z_k^2 = z_k$ (binary):

$$Q_{kk} = Pa_k^2 - 2PCa_k - [k < n]v_k, \quad Q_{ij} = 2Pa_i a_j \quad (i < j)$$

Correctness. ($\Rightarrow$) If $\boldsymbol{x}^*$ is a feasible knapsack solution with value V^*, then there exist slack values $\boldsymbol{s}^*$ satisfying the equality constraint (encoding $C - \sum w_i x_i^*$ in binary), so $f(\boldsymbol{z}^*) = -V^*$. ($\Leftarrow$) If the equality constraint is violated, the penalty $\left(\sum a_k z_k - C\right)^2 \geq 1$ contributes at least $P > \sum_i v_i$ to the objective. Since all values are nonnegative, every feasible assignment has objective in the range $\left[-\sum_i v_i, 0\right]$, so that penalty exceeds the entire feasible value range. Among feasible assignments (penalty zero), f reduces to $-\sum v_i x_i$, minimized at the knapsack optimum.

Solution extraction. Discard slack variables: return $\boldsymbol{z}[0..n]$. □

Example: $n = 4$ **items, capacity** $C = 7$

Source: Knapsack **Target:** QUBO

```
$ pred create --example Knapsack -o knapsack.json
$ pred reduce knapsack.json --to QUBO/f64 -o bundle.json
$ pred solve bundle.json
$ pred evaluate knapsack.json --config 1,0,0,1
```

Step 1 – Source instance. The canonical knapsack instance has weights (2, 3, 4, 5), values (3, 4, 5, 7), and capacity $C = 7$.

Step 2 – Introduce slack variables. The inequality $\sum_i w_i x_i \leq C$ becomes an equality by adding $B = 3$ binary slack bits that encode unused capacity:

$$2x_0 + 3x_1 + 4x_2 + 5x_3 + 1s_0 + 2s_1 + 4s_2 = 7$$

This gives $n + B = 4 + 3 = 7$ QUBO variables.

Step 3 – Add the penalty objective. With penalty $P = 1 + \sum_i v_i = 20$, the QUBO minimizes

$$H = -(3x_0 + 4x_1 + 5x_2 + 7x_3) + 20(2x_0 + 3x_1 + 4x_2 + 5x_3 + 1s_0 + 2s_1 + 4s_2 - 7)^2$$

so any violation of the equality is more expensive than the entire knapsack value range.

Step 4 – Verify a solution. The QUBO ground state $\boldsymbol{z} = (1, 0, 0, 1, 0, 0, 0)$ extracts to the knapsack choice $\boldsymbol{x} = (1, 0, 0, 1)$. This selects items $\{0, 3\}$ with total weight $2 + 5 = 7$ and total value $3 + 7 = 10$, so the slack bits are all zero and the penalty term vanishes ✓.

Uniqueness: The fixture stores one canonical optimal witness. The source optimum is unique because items $\{0, 3\}$ are the only feasible selection achieving value 10.

Rule 3.31: (Minimum Multiway Cut (weighted) → QUBO (real-weighted)) The multiway cut problem requires a partition of vertices into k components — one per terminal — minimizing the total weight of edges crossing components. The penalty method (Section 3.2) encodes two constraints as QUBO penalties: (1) each vertex belongs to exactly one component (one-hot), and (2) each terminal is pinned to its own component. The cut-cost Hamiltonian counts edge weight across distinct components. Reference: [149]. *Overhead:* `num_vars = num_terminals * num_vertices`.

Proof: *Construction.* Given $G = (V, E)$ with $n = |V|$, edge weights $w : E \to \mathbb{R}_{>0}$, and k terminals $T = \{t_0, ..., t_{k-1}\}$. Introduce nk binary variables $x_{u,t} \in \{0, 1\}$ (indexed by $u \cdot k + t$), where $x_{u,t} = 1$ means vertex u is in terminal t's component. Let $\alpha = 1 + \sum_{e \in E} w(e)$.

The QUBO Hamiltonian is $H = H_A + H_B$ where:

$$H_A = \alpha \left(\sum_{u \in V} \left(1 - \sum_{t=0}^{k-1} x_{u,t} \right)^2 + \sum_{i=0}^{k-1} \sum_{s \neq i} x_{t_i,s} \right)$$

The first term is a *one-hot constraint* ensuring each vertex is assigned to exactly one component. The second term *pins* each terminal t_i to position i by penalizing any other assignment. Expanding the one-hot term using $x^2 = x$:

$$Q_{uk+t,uk+t} = -\alpha, \quad Q_{uk+s,uk+t} = 2\alpha \quad (s < t)$$

Terminal pinning adds α to the diagonal $Q_{t_i k+s, t_i k+s}$ for $s \neq i$, canceling the one-hot incentive.

The cut-cost Hamiltonian:

$$H_B = \sum_{(u,v) \in E} \sum_{s \neq t} w(u,v) \cdot x_{u,s} \cdot x_{v,t}$$

counts the total weight of edges whose endpoints lie in different components.

Correctness. ($\Rightarrow$) A valid multiway cut with cost C maps to a QUBO solution with $H_A = 0$ (valid partition with correct terminal pinning) and $H_B = C$. ($\Leftarrow$) If $H_A > 0$, the penalty $\alpha > \sum_e w(e)$ exceeds the entire cut-cost range, so any QUBO minimizer has $H_A = 0$, encoding a valid partition. Among valid partitions, H_B equals the cut cost, and the minimizer achieves the minimum multiway cut.

Solution extraction. For each vertex u, find terminal position t with $x_{u,t} = 1$. For each edge (u, v), output 1 (cut) if u and v are in different components, 0 otherwise. □

Example: $n = 5$ **vertices,** $k = 3$ **terminals** $T = \{0, 2, 4\}$**,** $|E| = 6$ **edges**

Source: MinimumMultiwayCut **Target:** QUBO

```
$ pred create --example MinimumMultiwayCut -o minimummultiwaycut.json
$ pred reduce minimummultiwaycut.json --to QUBO/f64 -o bundle.json
$ pred solve bundle.json
$ pred evaluate minimummultiwaycut.json --config 1,0,0,1,1,0
```

Step 1 – Source instance. The canonical graph has $n = 5$ vertices, $m = 6$ edges with weights (2, 3, 1, 2, 4, 5), and $k = 3$ terminals $T = \{0, 2, 4\}$.

Step 2 – Introduce binary variables. Assign $k = 3$ indicator variables per vertex: $x_{u,t} = 1$ means vertex u belongs to terminal t's component. This gives $nk = 5 \times 3 = 15$ QUBO variables:

$$\underbrace{x_{0,0}x_{0,1}x_{0,2}}_{\text{vertex 0}} \; \underbrace{x_{1,0}x_{1,1}x_{1,2}}_{\text{vertex 1}} \cdots \underbrace{x_{4,0}x_{4,1}x_{4,2}}_{\text{vertex 4}}$$

Step 3 – Penalty coefficient. $\alpha = 1 + \sum_{e \in E} w(e) = 1 + 2 + 3 + 1 + 2 + 4 + 5 = 18$.

Step 4 – Build H_A (constraints). One-hot: diagonal entries $Q_{uk+t,uk+t} = -18$, off-diagonal $Q_{uk+s,uk+t} = 36$ within each vertex's group. Terminal pinning: for each terminal vertex t_i, the wrong-position diagonal entries $Q_{t_i k+s, t_i k+s} \mathrel{+}= 18$ for $s \neq i$, effectively canceling the one-hot incentive for those positions.

Step 5 – Build H_B (cut cost). For each edge (u, v) with weight w and each pair $s \neq t$, add w to $Q_{uk+s,vk+t}$. For example, edge $(0, 1)$ with weight 2 contributes 2 to positions $(x_{0,0}, x_{1,1})$, $(x_{0,0}, x_{1,2})$, $(x_{0,1}, x_{1,0})$, $(x_{0,1}, x_{1,2})$, $(x_{0,2}, x_{1,0})$, and $(x_{0,2}, x_{1,1})$.

Step 6 – Verify a solution. The QUBO ground state $\boldsymbol{x} = (1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1)$ decodes to the partition: vertex 0 in component 0, vertices 1–3 in component 1, vertex 4 in component 2. Cut edges: {(0,1), (3,4), (0,4)} with total weight $2 + 2 + 4 = 8$ ✓.

Rule 3.32: (QUBO (real-weighted) → Integer Linear Programming) QUBO minimizes a quadratic form $\boldsymbol{x}^\top Q\boldsymbol{x}$ over binary variables. Every quadratic term $Q_{ij}x_ix_j$ can be *linearized* by introducing an auxiliary variable y_{ij} constrained to equal the product x_ix_j via three McCormick inequalities. Diagonal terms $Q_{ii}x_i^2 = Q_{ii}x_i$ are already linear for binary x_i. The result is a binary ILP with a linear objective and $3m$ constraints (where m is the number of non-zero off-diagonal entries), whose minimizer corresponds exactly to the QUBO minimizer.
Overhead: `num_vars = num_vars^2`, `num_constraints = num_vars^2`.

Proof: *Construction.* For $Q \in \mathbb{R}^{n\times n}$ (upper triangular) with m non-zero off-diagonal entries:

Diagonal terms. For binary x_i: $Q_{ii}x_i^2 = Q_{ii}x_i$, which is directly linear.

Off-diagonal terms. For each non-zero Q_{ij} $(i < j)$, introduce binary $y_{ij} = x_i \cdot x_j$ with McCormick constraints:

$$y_{ij} \le x_i, \quad y_{ij} \le x_j, \quad y_{ij} \ge x_i + x_j - 1$$

ILP formulation. Minimize $\sum_i Q_{ii}x_i + \sum_{i<j} Q_{ij}y_{ij}$ subject to the McCormick constraints and $x_i, y_{ij} \in \{0,1\}$.

The ILP is:

$$\begin{aligned} \min \quad & \sum_i Q_{ii}x_i + \sum_{i<j} Q_{ij}y_{ij} \\ \text{subject to} \quad & y_{ij} \le x_i \quad \forall i<j, Q_{ij} \neq 0 \\ & y_{ij} \le x_j \quad \forall i<j, Q_{ij} \neq 0 \\ & y_{ij} \ge x_i + x_j - 1 \quad \forall i<j, Q_{ij} \neq 0 \\ & x_i, y_{ij} \in \{0,1\} \end{aligned}$$

.

Correctness. ($\Rightarrow$) For binary x_i, x_j, the three McCormick inequalities are tight: $y_{ij} = x_ix_j$ is the unique feasible value. Hence the ILP objective equals $\boldsymbol{x}^\top Q\boldsymbol{x}$, and any ILP minimizer is a QUBO minimizer. ($\Leftarrow$) Given a QUBO minimizer $\boldsymbol{x}^*$, setting $y_{ij} = x_i^*x_j^*$ satisfies all constraints and achieves the same objective value.

Solution extraction. Return the first n variables (discard auxiliary y_{ij}). □

Example: 4-variable QUBO with 3 quadratic terms

Source: QUBO **Target:** ILP

```
$ pred create --example QUBO -o qubo.json
$ pred reduce qubo.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate qubo.json --config 1,1,1,1
```

Source: $n = 4$ binary variables, 3 off-diagonal terms
Target: 7 ILP variables (4 original +3 auxiliary), 9 McCormick constraints
Canonical optimal witness: $\boldsymbol{x} = (1, 1, 1, 1)$ ✓

Rule 3.33: (CircuitSAT → Integer Linear Programming) Each boolean gate (AND, OR, NOT, XOR) has a truth table that can be captured exactly by a small set of linear inequalities over binary variables. By Tseitin-style flattening, each internal expression node gets an auxiliary ILP variable constrained to match its gate's output, so the conjunction of all gate constraints is feasible if and only if the circuit is satisfiable. The ILP has a trivial objective (minimize 0), making it a pure feasibility problem.

Overhead: `num_vars = num_variables + num_assignments`, `num_constraints = num_variables + num_assignments`.

Proof: *Construction.* Recursively assign an ILP variable to each expression node. Named circuit variables keep their identity; internal nodes get auxiliary variables.

Gate encodings (output c, inputs $a_1, ..., a_k$, all binary):
- NOT: $c + a = 1$
- AND: $c \leq a_i$ $(\forall i)$, $c \geq \sum a_i - (k-1)$
- OR: $c \geq a_i$ $(\forall i)$, $c \leq \sum a_i$
- XOR (binary, chained pairwise): $c \leq a + b$, $c \geq a - b$, $c \geq b - a$, $c \leq 2 - a - b$

Objective. Minimize 0 (feasibility problem): any feasible solution satisfies the circuit.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & c + a = 1 \quad \text{for each NOT gate} \\
& c \leq a_i \quad \forall i \quad \text{for each AND gate} \\
& c \geq \sum_i a_i - (k-1) \quad \text{for each AND gate} \\
& c \geq a_i \quad \forall i \quad \text{for each OR gate} \\
& c \leq \sum_i a_i \quad \text{for each OR gate} \\
& c \leq a + b \quad \text{for each XOR gate} \\
& c \geq a - b \quad \text{for each XOR gate} \\
& c \geq b - a \quad \text{for each XOR gate} \\
& c \leq 2 - a - b \quad \text{for each XOR gate} \\
& \text{all gate and input variables are binary}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Each gate encoding is the convex hull of the gate's truth table rows (viewed as binary vectors), so a satisfying circuit assignment satisfies all constraints. ($\Leftarrow$) Any binary feasible solution respects every gate's input-output relation, and since gates are composed in topological order, the full circuit evaluates to true.

Solution extraction. Return values of the named circuit variables. □

Example: 1-bit full adder to ILP

Source: CircuitSAT **Target:** ILP

```
$ pred create --example CircuitSAT -o circuitsat.json
$ pred reduce circuitsat.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate circuitsat.json --config 0,0,0,0,0,0,0,0
```

Circuit: 5 gates (2 XOR, 2 AND, 1 OR), 8 variables
Target: 13 ILP variables (circuit vars + auxiliary), trivial objective
Canonical feasible witness shown (2^3 valid input combinations exist for the full adder) ✓

3.3 Non-Trivial Reductions

Rule 3.34: (SAT → Maximum Independent Set) [1] A satisfying assignment must make at least one literal true in every clause, and different clauses cannot assign contradictory values to the same variable. These two requirements map naturally to an independent set problem: *intra-clause cliques* force exactly one

literal per clause to be selected, while *conflict edges* between complementary literals across clauses enforce consistency. The target IS size equals the number of clauses m, so an IS of size m exists iff the formula is satisfiable.
Overhead: `num_vertices = num_literals`, `num_edges = num_literals^2`.

Proof: *Construction.* For $\varphi = \bigwedge_{j=1}^{m} C_j$ with $C_j = \left(\ell_{j,1} \vee ... \vee \ell_{j,k_j}\right)$:

Vertices: For each literal $\ell_{j,i}$ in clause C_j, create $v_{j,i}$. Total: $|V| = \sum_j k_j$.

Edges: (1) Intra-clause cliques: $E_{\text{clause}} = \left\{(v_{j,i}, v_{j,i'}) : i \neq i'\right\}$. (2) Conflict edges: $E_{\text{conflict}} = \left\{(v_{j,i}, v_{j',i'}) : j \neq j', \ell_{j,i} = \overline{\ell_{j',i'}}\right\}$.

Correctness. ($\Rightarrow$) A satisfying assignment selects one true literal per clause; these vertices form an IS of size m (no clause edges by selection, no conflict edges by consistency). ($\Leftarrow$) An IS of size m must contain exactly one vertex per clause (by clause cliques); the corresponding literals are consistent (by conflict edges) and satisfy φ.

Solution extraction. For $v_{j,i} \in S$ with literal x_k: set $x_k = 1$; for $\overline{x_k}$: set $x_k = 0$. □

Example: 3-SAT with 5 variables and 7 clauses

Source: Satisfiability **Target:** MaximumIndependentSet

```
$ pred create --example SAT -o sat.json
$ pred reduce sat.json --to MaximumIndependentSet/SimpleGraph/One -o bundle.json
$ pred solve bundle.json
$ pred evaluate sat.json --config 1,1,1,1,0
```

SAT assignment: $(x_1, ..., x_5) = (1, 1, 1, 1, 0)$
IS graph: 21 vertices ($= 3 \times 7$ literals), 43 edges
IS of size $7 = m$: one vertex per clause → satisfying assignment ✓

Rule 3.35: (SAT → *k*-Coloring (*k*-ary)) [5] A 3-coloring partitions vertices into three classes. The key insight is that three colors suffice to encode Boolean logic: one color represents TRUE, one FALSE, and a third (AUX) serves as a neutral ground. Variable gadgets force each variable's positive and negative literals to receive opposite truth colors, while clause gadgets use an OR-chain that can only receive the TRUE color when at least one input literal is TRUE-colored. Connecting the output of each clause gadget to the FALSE vertex forces it to be TRUE-colored, encoding the requirement that every clause is satisfied.
Overhead: `num_vertices = num_vars + num_literals`, `num_edges = num_vars + num_literals`.

Proof: *Construction.* (1) *Base triangle:* vertices TRUE, FALSE, AUX, all mutually connected. This fixes three distinct colors and establishes the color semantics. (2) *Variable gadget* for x_i: vertices pos_i, neg_i connected to each other and to AUX. Since pos_i and neg_i are both adjacent to AUX, neither can receive the AUX color; since they are adjacent to each other, one must be TRUE-colored and the other FALSE-colored. (3) *Clause gadget* for $(\ell_1 \vee ... \vee \ell_k)$: apply OR-gadgets iteratively — $o_1 = \text{OR}(\ell_1, \ell_2)$, $o_2 = \text{OR}(o_1, \ell_3)$, etc. — producing final output o, then connect o to both FALSE and AUX.

OR-gadget$(a, b) \mapsto o$*:* Introduces five auxiliary vertices with edges arranged so that o can receive the TRUE color iff at least one of a, b has the TRUE color. When both inputs have the FALSE color, the gadget's internal constraints force o into the AUX color.

Correctness. ($\Rightarrow$) A satisfying assignment colors pos_i as TRUE when $x_i = 1$ and FALSE otherwise. Each clause has at least one TRUE literal, so the OR-chain output receives the TRUE color, which is compatible with edges to FALSE and AUX. ($\Leftarrow$) In any valid 3-coloring, the variable gadgets assign consistent truth values and the clause gadget connections to FALSE force each clause output to be TRUE-colored, meaning at least one literal per clause is TRUE.

Solution extraction. Set $x_i = 1$ iff $\text{color}(\text{pos}_i) = \text{color}(\text{TRUE})$. □

Example: 5-variable SAT with 3 unit clauses to 3-coloring

Source: Satisfiability **Target:** KColoring

```
$ pred create --example SAT -o sat.json
$ pred reduce sat.json --to KColoring/SimpleGraph/K3 -o bundle.json
$ pred solve bundle.json
$ pred evaluate sat.json --config 1,1,0,1,1
```

SAT assignment: $(x_1, ..., x_5) = (1, 1, 0, 1, 1)$
Construction: 3 base + 2×5 variable gadgets + OR-gadgets $\rightarrow$ 13 vertices, 24 edges
Canonical 3-coloring witness shown (the construction also has the expected color-symmetry multiplicity for satisfying assignments) ✓

Rule 3.36: (SAT $\rightarrow$ Minimum Dominating Set (weighted)) [5] Each variable is represented by a triangle whose three vertices correspond to the positive literal, negative literal, and a dummy. Any dominating set must include at least one vertex from each triangle to dominate the dummy. The clause vertices are connected only to the literal vertices that appear in the clause, so a dominating set of minimum size n (one vertex per triangle) dominates all clause vertices iff the chosen literals satisfy every clause.
Overhead: `num_vertices = 3 * num_vars + num_clauses`, `num_edges = 3 * num_vars + num_literals`.

Proof: *Construction.* (1) *Variable triangle* for x_i: vertices $\text{pos}_i = 3i$, $\text{neg}_i = 3i + 1$, $\text{dum}_i = 3i + 2$ forming a triangle. The dummy vertex dum_i is adjacent only to pos_i and neg_i, so it can only be dominated by a vertex from its own triangle. (2) *Clause vertex* $c_j = 3n + j$ connected to pos_i if $x_i \in C_j$, to neg_i if $\overline{x_i} \in C_j$.

Correctness. ($\Rightarrow$) Given a satisfying assignment, select pos_i if $x_i = 1$, else neg_i. This dominates all triangle vertices (each triangle has one selected vertex adjacent to both others). Each clause C_j has at least one true literal, so c_j is adjacent to at least one selected vertex. Total size: n. ($\Leftarrow$) Any dominating set needs $\geq$ 1 vertex per triangle (to dominate dum_i). A set of size n has exactly one per triangle. If dum_i is selected, it does not dominate any clause vertex; but it does dominate pos_i and neg_i, which still need to cover clauses. Since dum_i has no clause neighbors, we can swap it for pos_i or neg_i without losing domination of the triangle. After swapping, each clause vertex c_j must be dominated by some pos_i or neg_i, defining a consistent satisfying assignment.

Solution extraction. Set $x_i = 1$ if pos_i selected; $x_i = 0$ if neg_i selected. □

Example: 5-variable 7-clause 3-SAT to dominating set

Source: Satisfiability **Target:** MinimumDominatingSet

```
$ pred create --example SAT -o sat.json
$ pred reduce sat.json --to MinimumDominatingSet/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate sat.json --config 1,0,1,1,1
```

SAT assignment: $(x_1, ..., x_5) = (1, 0, 1, 1, 1)$
Vertex structure: $22 = 3 \times 5 + 7$ (variable triangles + clause vertices)
Dominating set of size $n = 5$: one vertex per variable triangle ✓

Rule 3.37: (SAT $\rightarrow$ Integral Flow with Homologous Arcs) This $O(nm + L)$ reduction [5], [61] constructs a directed network with one unit-capacity variable path per Boolean variable and one capacity-$(|C_j| - 1)$ bottleneck per clause. A feasible integral flow of value n exists if and only if the SAT formula is satisfiable. For n variables, m clauses, and total literal count $L = \sum_j |C_j|$, the target has $2nm + 3n + 2m + 2$ vertices, $2nm + 5n + m + L$ arcs, and L homologous pairs.

Overhead: `num_vertices = 2 * num_vars * num_clauses + 3 * num_vars + 2 * num_clauses + 2`, `num_arcs = 2 * num_vars * num_clauses + 5 * num_vars + num_clauses + num_literals`.

Proof: *Construction.* Let $\varphi = \bigwedge_{j=1}^{m} C_j$ with variables $x_1, ..., x_n$ and clause widths $k_j = |C_j|$. Introduce source s, sink t, split vertices split_i for $i \in \{1, ..., n\}$, true/false pipeline vertices $T_{j,i}$ and $F_{j,i}$ for every boundary $j \in \{0, ..., m\}$, and a collector/distributor pair (γ_j, δ_j) for each clause stage $j \in \{1, ..., m\}$.

Variable stage. For each variable x_i, add (s, split_i), $(\text{split}_i, T_{0,i})$, and $(\text{split}_i, F_{0,i})$, all with capacity 1. Any feasible flow of value n must therefore choose exactly one of the two channels for each variable.

Clause stage. For clause C_j, add a bottleneck arc (γ_j, δ_j) of capacity $k_j - 1$. For each variable x_i:

- if $x_i \in C_j$, route the false channel through the bottleneck via $(F_{j-1,i}, \gamma_j)$ and $(\delta_j, F_{j,i})$, while the true channel bypasses directly from $T_{j-1,i}$ to $T_{j,i}$;
- if $\overline{x_i} \in C_j$, route the true channel through the bottleneck via $(T_{j-1,i}, \gamma_j)$ and $(\delta_j, T_{j,i})$, while the false channel bypasses directly from $F_{j-1,i}$ to $F_{j,i}$;
- otherwise both channels bypass directly from boundary $j-1$ to boundary j.

Finally add $(T_{m,i}, t)$ and $(F_{m,i}, t)$ of capacity 1 for every variable, and set the requirement to $R = n$.

For every literal occurrence in C_j, declare the corresponding collector-entry and distributor-exit arcs homologous: pair $(F_{j-1,i}, \gamma_j)$ with $(\delta_j, F_{j,i})$ when $x_i \in C_j$, and pair $(T_{j-1,i}, \gamma_j)$ with $(\delta_j, T_{j,i})$ when $\overline{x_i} \in C_j$.

Variable mapping. Choosing the T-channel for x_i represents $x_i = 1$; choosing the F-channel represents $x_i = 0$. A literal of $\neg C_j$ is true exactly when its designated channel is forced through the stage-j bottleneck.

Correctness. ($\Rightarrow$) Let σ satisfy φ. Route one unit from s through split_i along the T-channel when $\sigma(x_i) = 1$ and along the F-channel otherwise. In clause stage j, exactly those literals of $\neg C_j$ that are true under σ attempt to use (γ_j, δ_j). Since σ satisfies C_j, at least one literal of C_j is true, so at least one literal of $\neg C_j$ is false. Hence at most $k_j - 1$ chosen channels use the bottleneck, respecting its capacity. The homologous equalities are satisfied because every chosen collector entry for variable x_i is matched with the corresponding distributor exit on the same variable/channel.

($\Leftarrow$) Let f be a feasible integral flow of value at least n. The only outgoing arcs of s are the n unit-capacity arcs (s, split_i), so each variable contributes exactly one unit. Conservation at split_i forces that unit onto exactly one of $(\text{split}_i, T_{0,i})$ or $(\text{split}_i, F_{0,i})$, defining a truth assignment σ. In clause stage j, the bottleneck allows at most $k_j - 1$ literal channels of $\neg C_j$ to carry flow. Because homologous pairs prevent flow from entering through one variable's channel and exiting through another's, this load counts genuine literals of $\neg C_j$ made true by σ. Therefore at least one literal of $\neg C_j$ is false, so the corresponding literal of C_j is true. Every clause is satisfied.

Solution extraction. Read the base channel choice for each variable: output $x_i = 1$ iff the flow on $(\text{split}_i, T_{0,i})$ is 1, and $x_i = 0$ iff the flow on $(\text{split}_i, F_{0,i})$ is 1. □

Example: 3-variable 4-clause SAT to equality-constrained integral flow

Source: Satisfiability **Target:** IntegralFlowHomologousArcs

```
$ pred create --example Satisfiability -o sat.json
$ pred reduce sat.json --to IntegralFlowHomologousArcs -o bundle.json
$ pred solve bundle.json
$ pred evaluate sat.json --config 1,0,1
```

SAT assignment: $(x_1, x_2, x_3) = (1, 0, 1)$
Target network: 43 vertices, 51 arcs, 8 homologous pairs, and $R = 3$
The stored flow witness gives bottleneck loads $1, 1, 1, 0$ across the four clause stages, so every stage respects its unit capacity ✓

Step 1 – Choose variable channels. The fixture assignment sends one unit on the T channel of x_1, one unit on the F channel of x_2, and one unit on the T channel of x_3. Because $R = 3$, any feasible witness must route exactly one unit out of each split node.

Step 2 – Process the clauses. Clause $C_1 = (x_1 \vee x_2)$ sends the false channels of x_1 and x_2 through the first bottleneck, so only the chosen F_2 path contributes. Clause $C_2 = (\neg x_1 \vee x_3)$ sends T_1 and F_3 through stage 2, giving load 1. Clause $C_3 = (\neg x_2 \vee \neg x_3)$ sends T_2 and T_3 through stage 3, again giving load 1. Clause $C_4 = (x_1 \vee x_3)$ sends F_1 and F_3 through stage 4, so the chosen assignment contributes load 0.

Step 3 – Verify a witness. The sink receives all three units, and each homologous pair forces the flow entering a collector from variable x_i to leave the matching distributor arc for the same variable/channel. The unsatisfying assignment $(1, 1, 1)$ is rejected because stage 3 would send both T_2 and T_3 through a unit-capacity bottleneck.

Multiplicity: The fixture stores one canonical satisfying assignment / flow pair. Other satisfying assignments induce different T/F channel choices, but the clause-stage bottleneck test is always determined by which literals of each clause are false.

Rule 3.38: (*k*-SAT (*k*-ary) → SAT) Every k-SAT instance is already a SAT instance — clauses happen to have exactly k literals, but SAT places no restriction on clause width. The embedding is the identity.
Overhead: `num_clauses = num_clauses`, `num_vars = num_vars`, `num_literals = num_literals`.

Proof: *Construction.* Variables and clauses are unchanged.

Correctness. (⇒) Any k-SAT satisfying assignment satisfies the same clauses under SAT. (⇐) Any SAT satisfying assignment satisfies the same clauses (which all have width k). *Solution extraction.* Identity. □

Rule 3.39: (SAT → *k*-SAT (*k*-ary)) [5], [66] Clauses shorter than k can be padded with a complementary pair $y, \overline{y}$ that is always satisfiable; clauses longer than k can be split into a chain of width-k clauses linked by auxiliary variables that propagate truth values. Both transformations preserve satisfiability while enforcing uniform clause width.
Overhead: `num_clauses = 4 * num_clauses + num_literals`, `num_vars = num_vars + 3 * num_clauses + num_literals`.

Proof: *Construction.*

Small clauses ($|C| < k$): Pad $(\ell_1 \vee ... \vee \ell_r)$ with fresh auxiliary y: $(\ell_1 \vee ... \vee \ell_r \vee y \vee \overline{y} \vee ...)$ to length k. The pair $y, \overline{y}$ is a tautology, so the padded clause is satisfiable iff the original is.

Large clauses ($|C| > k$): Split $(\ell_1 \vee ... \vee \ell_r)$ with auxiliaries $y_1, ..., y_{r-k}$:

$$(\ell_1 \vee ... \vee \ell_{k-1} \vee y_1) \wedge (\overline{y_1} \vee \ell_k \vee ... \vee y_2) \wedge ... \wedge (\overline{y_{r-k}} \vee \ell_{r-k+2} \vee ... \vee \ell_r)$$

Correctness. (⇒) If the original clause is satisfied by some literal ℓ_j, set the auxiliary chain so that $y_i = 1$ for all i before ℓ_j's sub-clause and $y_i = 0$ after. Each sub-clause then contains either a true original literal or a true auxiliary. (⇐) If all sub-clauses are satisfied but every original literal is false, the first clause forces $y_1 = 1$, which forces $y_2 = 1$ (since $\overline{y_1}$ is false), and so on until the last clause has $\overline{y_{r-k}} = 0$ and all remaining literals false — a contradiction.

Solution extraction. Discard auxiliary variables; return original variable assignments. □

Example: Mixed-size clauses (sizes 1 to 5) to 3-SAT

Source: Satisfiability **Target:** KSatisfiability

```
$ pred create --example SAT -o sat.json
$ pred reduce sat.json --to KSatisfiability/K3 -o bundle.json
$ pred solve bundle.json
$ pred evaluate sat.json --config 1,1,1,0,1
```

Source: 5 variables, 6 clauses (sizes 1, 2, 3, 3, 4, 5)
Target 3-SAT: $12 = 5 + 7$ variables, 13 clauses (small padded, large split)
First solution: $(x_1, ..., x_5) = (1, 1, 1, 0, 1)$, auxiliary vars are don't-cares ✓

Rule 3.40: (SAT → Maximum 2-Satisfiability) This reduction composes a satisfiability-preserving normalization to 3-CNF with the Garey–Johnson–Stockmeyer MAX-2-SAT gadget [5], [9]. Each normalized 3-clause contributes a 10-clause MAX-2-SAT block with a one-clause gap between the satisfied and unsatisfied cases. For a SAT instance with n variables, m clauses, and total literal count $L = \sum_j |C_j|$, the implementation produces at most $n + 2L + 4m$ target variables and $10(L + 3m)$ target clauses.
Overhead: `num_vars = num_vars + 2 * num_literals + 4 * num_clauses`, `num_clauses = 10 * (num_literals + 3 * num_clauses)`.

Proof: *Construction.* Let $\varphi = \bigwedge_{j=1}^{m} C_j$ be a CNF formula on variables $x_1, ..., x_n$.

Step 1: normalize each clause to width 3. Replace every clause independently:
- if $C = ()$, introduce a fresh variable y and use $(y \vee y \vee y) \wedge (\overline{y} \vee \overline{y} \vee \overline{y})$;
- if $C = (\ell_1)$, introduce fresh y, z and use the four clauses $(\ell_1 \vee y \vee z)$, $(\ell_1 \vee y \vee \overline{z})$, $(\ell_1 \vee \overline{y} \vee z)$, $(\ell_1 \vee \overline{y} \vee \overline{z})$;
- if $C = (\ell_1 \vee \ell_2)$, introduce fresh y and use $(\ell_1 \vee \ell_2 \vee y) \wedge (\ell_1 \vee \ell_2 \vee \overline{y})$;
- if C already has width 3, keep it unchanged;
- if $C = (\ell_1 \vee ... \vee \ell_k)$ with $k > 3$, introduce fresh $y_1, ..., y_{k-3}$ and replace C by

$$(\ell_1 \vee \ell_2 \vee y_1) \wedge (\overline{y_1} \vee \ell_3 \vee y_2) \wedge ... \wedge (\overline{y_{k-3}} \vee \ell_{k-1} \vee \ell_k).$$

Let the resulting 3-CNF formula be $\psi = \bigwedge_{t=1}^{m'} D_t$.

Step 2: convert each 3-clause to MAX-2-SAT. For every normalized clause $D_t = (a_t \vee b_t \vee c_t)$, introduce a fresh variable w_t and add the ten 2-clauses

$$(a_t \vee a_t), (b_t \vee b_t), (c_t \vee c_t), (w_t \vee w_t),$$
$$\left(\overline{a_t} \vee \overline{b_t}\right), \left(\overline{b_t} \vee \overline{c_t}\right), (\overline{a_t} \vee \overline{c_t}),$$
$$(a_t \vee \overline{w_t}), (b_t \vee \overline{w_t}), (c_t \vee \overline{w_t}).$$

The repeated literals encode unit clauses inside the exact-2-literal MAX-2-SAT model.

Correctness. For one gadget corresponding to D_t, exhaustive case analysis shows that the best choice of w_t satisfies exactly 7 of the ten clauses when D_t is true and at most 6 when D_t is false. Therefore, for every assignment to the normalized variables,

$$\text{MAX2SAT(target)} = 6m' + \#\{t : D_t \text{ is true}\}.$$

($\Rightarrow$) If φ is satisfiable, then the normalized formula ψ is satisfiable by Step 1. Every normalized clause is true, so each gadget attains value 7 and the target optimum is $7m'$. ($\Leftarrow$) If the target optimum is $7m'$, then every gadget must contribute 7, so every normalized clause D_t is true. Hence ψ is satisfiable, and the normalization in Step 1 implies that the restriction to the original variables satisfies φ. If φ is unsatisfiable, then ψ is unsatisfiable, so every assignment leaves at least one normalized clause false and the target optimum is strictly less than $7m'$.

Solution extraction. Discard all auxiliary normalization variables and all gadget variables w_t; return only the first n Boolean variables $(x_1, ..., x_n)$. □

Example: 3-variable 2-clause SAT to MAX-2-SAT

Source: Satisfiability **Target:** Maximum2Satisfiability

```
$ pred create --example Satisfiability -o sat.json
$ pred reduce sat.json --to Maximum2Satisfiability -o bundle.json
$ pred solve bundle.json
$ pred evaluate sat.json --config 1,1,1
```

Step 1 – Source instance. The canonical SAT formula has $n = 3$ variables and $m = 2$ clauses:

$$C_1 = (x_1 \vee \overline{x_2} \vee x_3), \quad C_2 = (\overline{x_1} \vee x_2).$$

The stored satisfying assignment is $(x_1, x_2, x_3) = (1, 1, 1)$.

Step 2 – Normalize to 3-CNF. Clause C_1 already has width 3. Clause C_2 introduces one auxiliary variable y_1 and becomes

$$(\overline{x_1} \vee x_2 \vee y_1) \wedge (\overline{x_1} \vee x_2 \vee \overline{y_1}).$$

The normalized formula therefore has 4 variables and 3 clauses.

Step 3 – Build the MAX-2-SAT gadgets. Introduce one gadget variable per normalized clause, so the target has 7 variables and 30 clauses. The stored witness is $(x_1, x_2, x_3, y_1, w_1, w_2, w_3) = (1, 1, 1, 0, 1, 0, 1)$. With $(y_1, w_1, w_2, w_3) = (0, 1, 0, 1)$, each of the three gadgets satisfies exactly 7 clauses, so the target objective reaches $21 = 7 \times 3$ ✓.

Multiplicity: The fixture stores one canonical optimum. Auxiliary variables such as y_1 can vary across optimal witnesses, but truncating any optimal target assignment to the first 3 coordinates still yields a satisfying assignment of the original SAT formula.

Rule 3.41: (SAT → CircuitSAT) CNF is inherently an AND-of-ORs structure, which maps directly to a boolean circuit: each clause becomes an OR gate over its literals, and a final AND gate combines all clause outputs. The circuit is constrained to output *true*, so a satisfying circuit assignment exists iff the original formula is satisfiable.
Overhead: `num_variables = num_vars + num_clauses + 1`, `num_assignments = num_clauses + 2`.

Proof: *Construction.* For $\varphi = C_1 \wedge ... \wedge C_k$ with $C_i = (\ell_{i1} \vee ... \vee \ell_{im_i})$: (1) Create circuit inputs $x_1, ..., x_n$ corresponding to SAT variables. (2) For each clause C_i, add an OR gate g_i with inputs from the clause's literals (negated inputs use NOT gates). (3) Add a final AND gate with inputs $g_1, ..., g_k$, constrained to output *true*.

Correctness. (⇒) A satisfying assignment makes at least one literal true in each clause, so each OR gate outputs true and the AND gate outputs true. (⇐) A satisfying circuit assignment has all OR gates true (forced by the AND output constraint), meaning at least one literal per clause is true — exactly a SAT solution.

Solution extraction. Return the values of the circuit input variables $x_1, ..., x_n$. □

Example: 3-variable SAT formula to boolean circuit

Source: Satisfiability **Target:** CircuitSAT

```
$ pred create --example SAT -o sat.json
$ pred reduce sat.json --to CircuitSAT -o bundle.json
$ pred solve bundle.json
$ pred evaluate sat.json --config 1,1,1
```

Rule 3.42: (CircuitSAT → SAT) This linear-time Tseitin reduction [66] rewrites each circuit assignment into CNF by keeping every named circuit variable as a SAT variable and introducing one auxiliary variable for each non-leaf subexpression after constant folding and binary balancing. The target therefore has one SAT variable per circuit variable plus one per introduced gate.
Overhead: `num_vars = tseitin_num_vars`, `num_clauses = tseitin_num_clauses`.

Proof: *Construction.* Consider one circuit assignment $o_1, ..., o_t = e$. First simplify constant subexpressions inside e and rewrite every n-ary AND, OR, and XOR node as a balanced binary tree. For each non-leaf subexpression α, introduce a fresh SAT variable v_α; named circuit variables are reused directly. Add the standard Tseitin clauses

$$\begin{aligned} v_{\neg a} &\text{ iff } \neg a, \\ v_{a \wedge b} &\text{ iff } a \wedge b, \\ v_{a \vee b} &\text{ iff } a \vee b, \\ v_{a \oplus b} &\text{ iff } a \oplus b \end{aligned}$$

using the 2-clause NOT gadget, the 3-clause AND/OR gadgets, and the 4-clause XOR gadget. If the simplified right-hand side becomes a variable or auxiliary variable z_e, add $(\overline{o_i} \vee z_e)$ and $(o_i \vee \overline{z_e})$ for every output o_i. If it simplifies to a constant, add the unit clause o_i or $\overline{o_i}$ accordingly. Repeat this independently for every assignment in the circuit.

Correctness. ($\Rightarrow$) Let σ be a satisfying CircuitSAT assignment. Set every auxiliary variable v_α to the truth value of the corresponding subexpression α under σ. Each Tseitin gadget is then satisfied because its output variable matches the gate semantics, and every output-equivalence or unit clause holds because σ already makes each circuit assignment $o_1, ..., o_t = e$ true. Hence the CNF is satisfiable. ($\Leftarrow$) Let τ satisfy the constructed CNF. Every Tseitin gadget forces its auxiliary variable to equal the truth value of its subexpression, so the root variable z_e equals the value of e. The output-equivalence clauses therefore force every output o_i to equal e, and unit clauses force the required constants. Restricting τ to the named circuit variables yields an assignment satisfying every original circuit equation.

Solution extraction. Return the values of the named circuit variables and discard the auxiliary Tseitin variables. □

Example: Tseitin encoding of a circuit equation

Source: CircuitSAT **Target:** Satisfiability

```
$ pred create --example CircuitSAT -o circuitsat.json
$ pred reduce circuitsat.json --to Satisfiability -o bundle.json
$ pred solve bundle.json
$ pred evaluate circuitsat.json --config 1,1,1,0,1
```

Circuit: 1 assignment, 5 named variables
Target: 9 SAT variables, 13 clauses

Step 1 – Flatten the expression. The canonical example is the circuit equation $r = (x_1 \wedge x_2) \vee (\neg x_3 \wedge x_4)$. Introduce auxiliary variables $a = x_1 \wedge x_2$, $b = \neg x_3$, $c = b \wedge x_4$, and $d = a \vee c$. This yields 9 SAT variables: the five named circuit variables (r, x_1, x_2, x_3, x_4) plus four Tseitin variables (a, b, c, d).

Step 2 – Emit clauses. Add three clauses for $a = x_1 \wedge x_2$, two for $b = \neg x_3$, three for $c = b \wedge x_4$, three for $d = a \vee c$, and two clauses for the output identity $r \equiv d$. The CNF therefore has 13 clauses.

Step 3 – Verify a witness. The fixture stores source config 1, 1, 1, 0, 1, meaning $(r, x_1, x_2, x_3, x_4) = (1, 1, 1, 0, 1)$. Extending with $(a, b, c, d) = (1, 1, 1, 1)$ gives the SAT witness 1, 1, 1, 0, 1, 1, 1, 1, 1, which satisfies every gate-definition clause and the two clauses enforcing $r \equiv d$.

Multiplicity: The fixture stores one canonical consistent assignment. Other satisfying assignments may exist whenever the circuit equations leave some named variables unconstrained.

Rule 3.43: (CircuitSAT → Spin Glass (weighted)) [144], [150] Each logic gate can be represented as an Ising gadget — a small set of spins with couplings J_{ij} and fields h_i chosen so that the gadget's ground states correspond exactly to the gate's truth table rows. Composing gadgets for all gates in the circuit yields a spin glass whose ground states encode precisely the satisfying assignments of the circuit. The energy gap between valid and invalid I/O patterns ensures that any global ground state respects every gate's logic.
Overhead: `num_spins = num_assignments`, `num_interactions = num_assignments`.

Proof: *Construction.*

Spin mapping: Boolean variables $\sigma \in \{0, 1\}$ map to Ising spins $s = 2\sigma - 1 \in \{-1, +1\}$. Each circuit variable is assigned a unique spin index; gate gadgets reference these indices for their inputs and outputs.

Gate gadgets (inputs 0,1; output 2; auxiliary 3 for XOR) are listed in Table 6. For each gate, instantiate the gadget's couplings and fields. The total Hamiltonian is the sum over all gadgets: $H = -\sum_{i<j} J_{ij} s_i s_j - \sum_i h_i s_i$.

Correctness. (⇒) A satisfying circuit assignment maps to a spin configuration where every gadget is in a ground state (valid I/O), so the total energy is minimized. (⇐) Any global ground state must minimize each gadget's contribution. Since each gadget's ground states match its gate's truth table, the spin configuration encodes a valid circuit evaluation. The output spin is constrained to +1 (true), so the circuit is satisfied.

Solution extraction. Map spins back to Boolean: $\sigma_i = \frac{s_i+1}{2}$. Return the circuit input variables. □

Example: 1-bit full adder to Ising model

Source: CircuitSAT **Target:** SpinGlass

```
$ pred create --example CircuitSAT -o circuitsat.json
$ pred reduce circuitsat.json --to SpinGlass/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate circuitsat.json --config 0,0,0,0,0,0,0,0
```

Circuit: 5 gates (2 XOR, 2 AND, 1 OR), 8 variables
Target: 15 spins (each gate allocates I/O + auxiliary spins)
Canonical ground-state witness shown (2^3 valid input combinations exist for the full adder) ✓

Gate	Couplings J	Fields h
AND	$J_{01} = 1, J_{02} = J_{12} = -2$	$h_0 = h_1 = -1, h_2 = 2$
OR	$J_{01} = 1, J_{02} = J_{12} = -2$	$h_0 = h_1 = 1, h_2 = -2$
NOT	$J_{01} = 1$	$h_0 = h_1 = 0$
XOR	$J_{01} = 1, J_{02} = J_{12} = -1, J_{03} = J_{13} = -2, J_{23} = 2$	$h_0 = h_1 = -1, h_2 = 1, h_3 = 2$

Table 6: Ising gadgets for logic gates. Ground states match truth tables.

Rule 3.44: (Factoring → CircuitSAT) Integer multiplication can be implemented as a boolean circuit: an $m \times n$ array multiplier computes $p \times q$ using only AND, XOR, and OR gates arranged in a grid of full adders. Constraining the output bits to match N turns the circuit into a satisfiability problem — the circuit is satisfiable iff $N = p \times q$ for some p, q within the given bit widths. *(Folklore; no canonical reference.)*
Overhead: `num_variables = 6 * num_bits_first * num_bits_second + num_bits_first + num_bits_second`, `num_assignments = 6 * num_bits_first * num_bits_second + num_bits_first + num_bits_second`.

Proof: *Construction.* Build $m \times n$ array multiplier for $p \times q$:

Full adder (i, j)*:* Each cell computes one partial product bit $p_i \wedge q_j$ and adds it to the running sum from previous cells. The sum and carry are: $s_{i,j} + 2c_{i,j} = (p_i \wedge q_j) + s_{\text{prev}} + c_{\text{prev}}$, implemented via:

$$a := p_i \wedge q_j, \quad t_1 := a \oplus s_{\text{prev}}, \quad s_{i,j} := t_1 \oplus c_{\text{prev}}$$

$$t_2 := t_1 \wedge c_{\text{prev}}, \quad t_3 := a \wedge s_{\text{prev}}, \quad c_{i,j} := t_2 \vee t_3$$

Output constraint: Fix output wires to the binary representation of N: $M_k := \text{bit}_{k(N)}$ for $k = 1, ..., m+n$.

Correctness. ($\Rightarrow$) If $N = p \times q$ with $p < 2^m$ and $q < 2^n$, setting the input bits to the binary representations of p and q produces output bits matching N, satisfying all constraints. ($\Leftarrow$) Any satisfying assignment to the circuit computes a valid multiplication (the gates enforce arithmetic correctness), and the output constraint ensures the product equals N.

Solution extraction. Read off factor bits: $p = \sum_i p_i 2^{i-1}$, $q = \sum_j q_j 2^{j-1}$. □

Example: Factor $N = 35$

Source: Factoring **Target:** CircuitSAT

```
$ pred create --example Factoring -o factoring.json
$ pred reduce factoring.json --to CircuitSAT -o bundle.json
$ pred solve bundle.json
$ pred evaluate factoring.json --config 1,0,1,1,1,1
```

Circuit: 3×3 array multiplier with 60 gates, 60 variables
Canonical witness: $5 \times 7 = 35$ ✓

Rule 3.45: (Max-Cut (weighted) → Spin Glass (weighted)) [6] A maximum cut partitions vertices into two groups to maximize the total weight of edges crossing the partition. In the Ising model, two spins with opposite signs contribute $-J_{ij}s_is_j = J_{ij}$ to the energy, while same-sign spins contribute $-J_{ij}$. Setting $J_{ij} = w_{ij}$ and $h_i = 0$ makes each cut edge lower the energy by $2J_{ij}$ relative to an uncut edge, so the Ising ground state corresponds to the maximum cut.
Overhead: `num_spins = num_vertices`, `num_interactions = num_edges`.

Proof: *Construction.* Map each vertex to a spin with $J_{ij} = w_{ij}$ for each edge and $h_i = 0$ (no external field). Spins are $s_i = 2\sigma_i - 1$ where $\sigma_i \in \{0, 1\}$ is the partition label.

Correctness. ($\Rightarrow$) A maximum cut assigns $\sigma_i \in \{0, 1\}$. For cut edges, $s_is_j = -1$, contributing $-J_{ij}(-1) = +J_{ij}$. For uncut edges, $s_is_j = +1$, contributing $-J_{ij}$. Maximizing cut weight is equivalent to minimizing $-\sum J_{ij}s_is_j$, the Ising energy. ($\Leftarrow$) An Ising ground state minimizes $-\sum J_{ij}s_is_j$, which is maximized when opposite-sign pairs (cut edges) have the largest possible weights — exactly the maximum cut.

Solution extraction. Partition $= \{i : s_i = +1\}$. □

Example: Petersen graph ($n = 10$, unit weights) to Ising

Source: MaxCut **Target:** SpinGlass

```
$ pred create --example MaxCut -o maxcut.json
$ pred reduce maxcut.json --to SpinGlass/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate maxcut.json --config 0,1,0,1,0,1,0,0,0,1
```

Direct 1:1 mapping: vertices → spins, $J_{ij} = w_{ij} = 1$, $h_i = 0$
Partition: $S = \{1, 3, 5, 9\}$ vs $\overline{S} = \{0, 2, 4, 6, 7, 8\}$
Cut value = 12 (canonical witness shown) ✓

Rule 3.46: (Spin Glass (weighted) → Max-Cut (weighted)) [6], [144] The Ising Hamiltonian $H = -\sum J_{ij}s_is_j - \sum h_is_i$ has two types of terms. The pairwise couplings J_{ij} map directly to MaxCut edge

weights, since minimizing $-J_{ij}s_is_j$ favors opposite spins (cut edges) when $J_{ij} > 0$. The local fields h_i have no direct MaxCut analogue, but can be absorbed by introducing a single ancilla vertex connected to every spin with weight h_i: fixing the ancilla's partition side effectively creates a linear bias on each spin.
Overhead: `num_vertices = num_spins`, `num_edges = num_interactions`.

Proof: *Construction.* If all $h_i = 0$: set $w_{ij} = J_{ij}$ directly (1:1 mapping, no ancilla). If some $h_i \neq 0$: add ancilla vertex a with edges $w_{i,a} = h_i$ for each spin i. The Ising energy $-\sum J_{ij}s_is_j - \sum h_is_i$ equals $-\sum J_{ij}s_is_j - \sum h_is_is_a$ when $s_a = +1$, which is a pure pairwise Hamiltonian on $n+1$ spins.

Correctness. ($\Rightarrow$) An Ising ground state assigns spins to minimize H. The equivalent MaxCut graph has the same objective (up to a constant), so the spin configuration defines a maximum cut. ($\Leftarrow$) A maximum cut on the constructed graph maximizes $\sum_{\text{cut}} w_{ij}$, which corresponds to minimizing $-\sum J_{ij}s_is_j - \sum h_is_is_a$. With s_a fixed, this is the Ising energy, so the cut defines a ground state.

Solution extraction. Without ancilla: partition labels are the spin values directly. With ancilla: if $\sigma_a = 1$ (ancilla on the $+1$ side), the spin values are read directly; if $\sigma_a = 0$, flip all spins before reading (since the ancilla should represent $s_a = +1$). □

Example: 10-spin Ising with alternating $J_{ij} \in \{\pm 1\}$

Source: SpinGlass **Target:** MaxCut

```
$ pred create --example SpinGlass -o spinglass.json
$ pred reduce spinglass.json --to MaxCut/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate spinglass.json --config 1,0,1,1,1,0,1,0,0,1
```

All $h_i = 0$: no ancilla needed, direct 1:1 vertex mapping
Edge weights $w_{ij} = J_{ij} \in \{\pm 1\}$ (alternating couplings)
Canonical ground-state witness: partition $S = \{0, 2, 3, 4, 6, 9\}$ ✓

Rule 3.47: (*k*-Coloring (*k*-ary) → Integer Linear Programming) A k-coloring assigns each vertex exactly one of k colors such that adjacent vertices differ. Both requirements are naturally linear: the "exactly one color" condition is an equality constraint on k binary indicators per vertex, and the "neighbors differ" condition bounds each color's indicator sum to at most one per edge. The resulting ILP has $|V|$ k variables and $|V| + |E|$ k constraints with a trivial objective.
Overhead: `num_vars = num_vertices^2`, `num_constraints = num_vertices + num_vertices * num_edges`.

Proof: *Construction.* For graph $G = (V, E)$ with k colors:

Variables: Binary $x_{v,c} \in \{0, 1\}$ for each vertex $v \in V$ and color $c \in \{1, ..., k\}$. Interpretation: $x_{v,c} = 1$ iff vertex v has color c.

Constraints: (1) Each vertex has exactly one color: $\sum_{c=1}^{k} x_{v,c} = 1$ for all $v \in V$. (2) Adjacent vertices have different colors: $x_{u,c} + x_{v,c} \leq 1$ for all $(u, v) \in E$ and $c \in \{1, ..., k\}$.

Objective: Feasibility problem (minimize 0).

The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{c=1}^{k} x_{v,c} = 1 \quad \forall v \in V \\ & x_{u,c} + x_{v,c} \leq 1 \quad \forall (u, v) \in E, c \in \{1, ..., k\} \\ & x_{v,c} \in \{0, 1\} \end{aligned}$$

.

Correctness. ($\Rightarrow$) A valid k-coloring assigns exactly one color per vertex with different colors on adjacent vertices; setting $x_{v,c} = 1$ for the assigned color satisfies all constraints. ($\Leftarrow$) Any feasible ILP solution has exactly one $x_{v,c} = 1$ per vertex; this defines a coloring, and constraint (2) ensures adjacent vertices differ.

Solution extraction. For each vertex v, find c with $x_{v,c} = 1$; assign color c to v. □

Rule 3.48: (k-Coloring (k-ary) → 2-Dimensional Consecutive Sets) [57] A graph 3-coloring can be encoded as a partition problem on an alphabet. Each edge becomes a size-3 subset containing the two endpoint symbols plus a unique dummy symbol, and a valid 3-coloring corresponds to partitioning the alphabet into 3 groups where each edge-subset spans exactly 3 consecutive groups with one element per group. The reduction uses $n + m$ alphabet symbols and m subsets for a graph with n vertices and m edges.
Overhead: `alphabet_size = num_vertices + num_edges`, `num_subsets = num_edges`.

Proof: *Construction.* Given $G = (V, E)$ with $|V| = n$ and $|E| = m$, build alphabet $\Sigma = V \cup \{d_e : e \in E\}$ of size $n + m$. For each edge $e = \{u, v\} \in E$, define subset $\Sigma_e = \{u, v, d_e\}$. The collection is $\mathcal{C} = \{\Sigma_e : e \in E\}$ with $|\mathcal{C}| = m$ subsets, each of size 3.

Correctness. ($\Rightarrow$) Given a valid 3-coloring $\chi : V \to \{1, 2, 3\}$, define partition groups $X_c = \{v \in V : \chi(v) = c\}$ for $c \in \{1, 2, 3\}$. For each edge $e = \{u, v\}$, assign dummy d_e to the unique third color $c^* \in \{1, 2, 3\} \setminus \{\chi(u), \chi(v)\}$ (which exists since $\chi(u) \neq \chi(v)$). Then $\Sigma_e = \{u, v, d_e\}$ has its three elements in three distinct groups $\{X_{\chi(u)}, X_{\chi(v)}, X_{c^*}\} = \{X_1, X_2, X_3\}$, which are consecutive with one element per group. ($\Leftarrow$) If a valid partition into k groups exists, each size-3 subset $\{u, v, d_e\}$ must occupy 3 distinct consecutive groups. In particular, u and v are in different groups. Mapping groups to colors gives a valid 3-coloring.

Solution extraction. The first n symbols in the target configuration correspond to the graph vertices. Their group assignments, compressed to $0, 1, 2$, yield the 3-coloring. □

Rule 3.49: (Factoring → Integer Linear Programming) Integer multiplication $p \times q = N$ is a system of bilinear equations over binary factor bits with carry propagation. Each bit-product $p_i q_j$ is a bilinear term that McCormick linearization replaces with an auxiliary variable and three inequalities. The carry-chain equations are already linear, so the full system becomes a binary ILP with $O(mn)$ variables and constraints.
Overhead: `num_vars = num_bits_first * num_bits_second`, `num_constraints = num_bits_first * num_bits_second`.

Proof: *Construction.* For target N with m-bit factor p and n-bit factor q:

Variables: Binary $p_i, q_j \in \{0, 1\}$ for factor bits; binary $z_{ij} \in \{0, 1\}$ for products $p_i \cdot q_j$; integer $c_k \geq 0$ for carries at each bit position.

Product linearization (McCormick): For each $z_{ij} = p_i \cdot q_j$:

$$z_{ij} \leq p_i, \quad z_{ij} \leq q_j, \quad z_{ij} \geq p_i + q_j - 1$$

Bit-position equations: For each bit position k:

$$\sum_{i+j=k} z_{ij} + c_{k-1} = N_k + 2c_k$$

where N_k is the k-th bit of N and $c_{-1} = 0$.

No overflow: $c_{m+n-1} = 0$.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & z_{ij} \leq p_i \quad \forall i, j \\
& z_{ij} \leq q_j \quad \forall i, j \\
& z_{ij} \geq p_i + q_j - 1 \quad \forall i, j \\
& \sum_{i+j=k} z_{ij} + c_{k-1} = N_k + 2c_k \quad \forall k \in \{0, \ldots, m+n-1\} \\
& c_{m+n-1} = 0 \\
& p_i, q_j, z_{ij} \in \{0,1\}, c_k \in \mathbb{Z}_{\geq 0}
\end{aligned}
$$

.

Correctness. The McCormick constraints enforce $z_{ij} = p_i \cdot q_j$ for binary variables. The bit equations encode $p \times q = N$ via carry propagation, matching array multiplier semantics.

Solution extraction. Read $p = \sum_i p_i 2^i$ and $q = \sum_j q_j 2^j$ from the binary variables. □

3.4 ILP Formulations

The following reductions to Integer Linear Programming are straightforward formulations where problem constraints map directly to linear inequalities.

Rule 3.50: (Maximum Set Packing (weighted) → Integer Linear Programming) Each set is either selected or not, and every universe element may belong to at most one selected set – an element-based constraint that is directly linear in binary indicator variables.
Overhead: `num_vars = num_sets`, `num_constraints = universe_size`.

Proof: *Construction.* Variables: $x_i \in \{0,1\}$ for each set $S_i \in \mathcal{S}$. The ILP is:

$$
\begin{aligned}
\max \quad & \sum_i w_i x_i \\
\text{subject to} \quad & \sum_{S_i \ni e} x_i \leq 1 \quad \forall e \in U \\
& x_i \in \{0,1\} \quad \forall i
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A valid packing chooses pairwise disjoint sets, so each element is covered at most once. ($\Leftarrow$) Any feasible binary solution covers each element at most once, hence the chosen sets are pairwise disjoint; the objective maximizes total weight.

Solution extraction. $\mathcal{P} = \{S_i : x_i = 1\}$. □

Rule 3.51: (Maximum Matching (weighted) → Integer Linear Programming) Each edge is either selected or not, and each vertex may be incident to at most one selected edge – a degree-bound constraint that is directly linear in binary edge indicators.
Overhead: `num_vars = num_edges`, `num_constraints = num_vertices`.

Proof: *Construction.* Variables: $x_e \in \{0,1\}$ for each $e \in E$. The ILP is:

$$
\begin{aligned}
\max \quad & \sum_e w_e x_e \\
\text{subject to} \quad & \sum_{e \ni v} x_e \leq 1 \quad \forall v \in V \\
& x_e \in \{0,1\} \quad \forall e \in E
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A matching has at most one edge per vertex, so all degree constraints hold. ($\Leftarrow$) Any feasible solution is a matching by construction; the objective maximizes total weight.

Solution extraction. $M = \{e : x_e = 1\}$. □

Rule 3.52: (Minimum Set Covering (weighted) $\rightarrow$ Integer Linear Programming) Every universe element must be covered by at least one selected set – a lower-bound constraint on the sum of indicators for sets containing that element, which is directly linear.
Overhead: `num_vars = num_sets`, `num_constraints = universe_size`.

Proof: *Construction.* Variables: $x_i \in \{0, 1\}$ for each $S_i \in \mathcal{S}$. The ILP is:

$$\begin{aligned} \min \quad & \sum_i w_i x_i \\ \text{subject to} \quad & \sum_{S_i \ni u} x_i \geq 1 \quad \forall u \in U \\ & x_i \in \{0, 1\} \quad \forall i \end{aligned}$$

.

Correctness. ($\Rightarrow$) A set cover includes at least one set containing each element, satisfying all constraints. ($\Leftarrow$) Any feasible solution covers every element; the objective minimizes total weight.

Solution extraction. $\mathcal{C} = \{S_i : x_i = 1\}$. □

Rule 3.53: (Minimum Dominating Set (weighted) $\rightarrow$ Integer Linear Programming) Every vertex must be dominated – either selected itself or adjacent to a selected vertex – which is a lower-bound constraint on the sum of indicators over its closed neighborhood.
Overhead: `num_vars = num_vertices`, `num_constraints = num_vertices`.

Proof: *Construction.* Variables: $x_v \in \{0, 1\}$ for each $v \in V$. The ILP is:

$$\begin{aligned} \min \quad & \sum_v w_v x_v \\ \text{subject to} \quad & x_v + \sum_{u \in N(v)} x_u \geq 1 \quad \forall v \in V \\ & x_v \in \{0, 1\} \quad \forall v \in V \end{aligned}$$

.

Correctness. ($\Rightarrow$) A dominating set includes a vertex or one of its neighbors for every vertex, satisfying all constraints. ($\Leftarrow$) Any feasible solution dominates every vertex; the objective minimizes total weight.

Solution extraction. $D = \{v : x_v = 1\}$. □

Rule 3.54: (Minimum Fault Detection Test Set $\rightarrow$ Integer Linear Programming) This direct $O((|I| + |O|)(|V| + |A|) + |I|\ |O|\ |V|)$ ILP encoding[65] precomputes the coverage set of each input-output pair, then introduces one binary selection variable per pair and one covering inequality per internal vertex ($|I|\ |O|$ variables and $|V \setminus (I \cup O)|$ constraints).
Overhead: `num_vars = num_inputs * num_outputs`, `num_constraints = num_vertices + -1 * num_inputs + -1 * num_outputs`.

Proof: *Construction.* Let the source DAG be $G = (V, A)$ with input set $I = \{i_0, ..., i_{a-1}\}$ and output set $O = \{o_0, ..., o_{b-1}\}$. For each ordered pair $(i_r, o_s) \in I \times O$, define its coverage set

$$C_{r,s} = \{v \in V : v \text{ is reachable from } i_r \text{ and can reach } o_s\}.$$

[65] Standard set-cover ILP formulation over input-output pair coverage sets; no dedicated bibliography key is currently registered in `references.bib`.

Let $W = V \setminus (I \cup O)$ be the internal vertices. Use `ILP<bool>` with binary variables $t_{r,s}$ indexed by the source pair order $p = rb + s$, so the target has ab variables. For every internal vertex $v \in W$, add the covering constraint

$$\sum_{(r,s):v \in C_{r,s}} t_{r,s} \geq 1.$$

The objective is

$$\min \sum_{r=0}^{a-1} \sum_{s=0}^{b-1} t_{r,s},$$

so the ILP minimizes the number of selected input-output pairs.

Correctness. ($\Rightarrow$) If $T \subseteq I \times O$ is a feasible source witness, set $t_{r,s} = 1$ exactly for pairs in T. Every internal vertex lies in the coverage set of at least one chosen pair, so each covering inequality holds, and the ILP objective equals $|T|$. ($\Leftarrow$) If the ILP is feasible, let $T = \{(i_r, o_s) : t_{r,s} = 1\}$. For every internal vertex $v \in W$, the corresponding inequality ensures that some selected pair satisfies $v \in C_{r,s}$, so the union of the chosen coverage sets covers all internal vertices. The source value is therefore exactly the ILP objective.

Solution extraction. The target variables already form the source binary selection vector in the same pair order, so return them unchanged. □

Example: DAG with 7 vertices, 2 inputs, 2 outputs, and 3 internal vertices

Source: MinimumFaultDetectionTestSet **Target:** ILP

```
$ pred create --example MinimumFaultDetectionTestSet -o mfdts.json
$ pred reduce mfdts.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate mfdts.json --config 1,0,0,1
```

Step 1 – Source instance. The canonical DAG has 7 vertices and arcs $(0,2)$, $(0,3)$, $(1,3)$, $(1,4)$, $(2,5)$, $(3,5)$, $(3,6)$, $(4,6)$. Inputs are $\{0, 1\}$, outputs are $\{5, 6\}$, so the internal vertices are $\{2,3,4\}$. The source configuration therefore has $2 \cdot 2 = 4$ input-output pair bits.

Step 2 – Build the covering ILP. Order the pair variables as $(0,5)$, $(0,6)$, $(1,5)$, and $(1,6)$. Their internal coverage sets are $\{2,3\}$, $\{3\}$, $\{3\}$, and $\{3,4\}$, so the target has 4 binary variables, 3 covering constraints, and objective $\min(x_0 + x_1 + x_2 + x_3)$. The exported constraints are exactly $x_0 \geq 1$, $x_0 + x_1 + x_2 + x_3 \geq 1$, and $x_3 \geq 1$.

Step 3 – Verify the canonical witness. The stored ILP witness is (1, 0, 0, 1). Because extraction is identity, the source witness is the same vector (1, 0, 0, 1), which selects pairs $(0,5)$ and $(1,6)$. These two pairs cover internal vertices $\{2,3\}$ and $\{3,4\}$ respectively, so their union covers every internal vertex and the optimum value is 2 ✓.

Multiplicity: The fixture stores one canonical witness. Any feasible solution must include $x_0 = 1$ to cover internal vertex 2 and $x_3 = 1$ to cover internal vertex 4, so the unique optimum is $(1,0,0,1)$.

Rule 3.55: (Minimum Feedback Vertex Set (weighted) → Integer Linear Programming) A directed graph is a DAG iff it admits a topological ordering. MTZ-style ordering variables enforce this: for each kept vertex, an integer position variable must increase strictly along every arc. Removed vertices relax the ordering constraints via big-M terms.
Overhead: `num_vars = 2 * num_vertices`, `num_constraints = num_arcs + 2 * num_vertices`.

Proof: *Construction.* Given directed graph $G = (V, A)$ with $n = |V|$, $m = |A|$, and weights w_v:

Variables: Binary $x_v \in \{0,1\}$ for each $v \in V$: $x_v = 1$ iff v is removed. Integer $o_v \in \{0, ..., n-1\}$ for each $v \in V$: topological order position. Total: $2n$ variables.

Constraints: (1) For each arc $(u \to v) \in A$: $o_v - o_u \geq 1 - n(x_u + x_v)$. When both endpoints are kept ($x_u = x_v = 0$), this forces $o_v > o_u$ (strict topological order). When either is removed, the constraint relaxes to $o_v - o_u \geq 1 - n$ (trivially satisfied). (2) Binary bounds: $x_v \leq 1$. (3) Order bounds: $o_v \leq n - 1$. Total: $m + 2n$ constraints.

Objective: Minimize $\sum_v w_v x_v$.

The ILP is:

$$\begin{aligned} \min \quad & \sum_v w_v x_v \\ \text{subject to} \quad & o_v - o_u \geq 1 - n(x_u + x_v) \quad \forall (u \to v) \in A \\ & x_v \in \{0,1\}, o_v \in \{0, ..., n-1\} \quad \forall v \in V \end{aligned}$$

.

Correctness. ($\Rightarrow$) If S is a feedback vertex set, then $G[V \setminus S]$ is a DAG with a topological ordering. Set $x_v = 1$ for $v \in S$, o_v to the topological position for kept vertices, and $o_v = 0$ for removed vertices. All constraints are satisfied. ($\Leftarrow$) If the ILP is feasible with all arc constraints satisfied, no directed cycle can exist among kept vertices: a cycle $v_1 \to ... \to v_k \to v_1$ would require $o_{v_1} < o_{v_2} < ... < o_{v_k} < o_{v_1}$, a contradiction.

Solution extraction. $S = \{v : x_v = 1\}$. □

Rule 3.56: (Minimum Feedback Vertex Set (weighted) → Minimum Code Generation (Unlimited Registers))
The Aho–Johnson–Ullman chain gadget construction [90] encodes a feedback vertex set problem as a code generation problem on an expression DAG with unlimited registers and 2-address instructions. Each source vertex becomes a leaf (input register), and each outgoing arc becomes an internal chain node. The number of LOAD (copy) instructions needed in an optimal program equals the size of a minimum feedback vertex set.
Overhead: `num_vertices = num_vertices + num_arcs`.

Proof: *Construction.* Given a directed graph $G = (V, A)$ with $n = |V|$ vertices and $m = |A|$ arcs, build an expression DAG D with $n + m$ vertices as follows. Vertices $0, ..., n-1$ are *leaves* (one per source vertex), each stored in its own register. For each source vertex x with outgoing arcs $(x, y_1), ..., (x, y_d)$, create a chain of d internal nodes $x^1, ..., x^d$ where:

- x^1 has left child x^0 (the leaf for x) and right child y_1^0 (the leaf for y_1),
- x^i ($i \geq 2$) has left child x^{i-1} (previous chain node) and right child y_i^0 (the leaf for y_i).

The left operand's register is destroyed by the OP instruction; a LOAD (copy) is needed whenever a leaf register must survive past its destruction.

Correctness. ($\Rightarrow$) If $F \subseteq V$ is a feedback vertex set of size k, then $G[V \setminus F]$ is a DAG. The topological order of $G[V \setminus F]$ induces an evaluation order of the chain nodes such that each leaf x^0 with $x \notin F$ is consumed (as a left child) only after all chain nodes that reference it as a right child have been evaluated. Only leaves corresponding to vertices in F need a LOAD instruction (their register is destroyed before some right-child usage). Hence the program uses exactly $n + m - n + k = m + k$ instructions, of which k are LOADs. ($\Leftarrow$) If an optimal program uses k LOAD instructions, the k leaves that require LOADs form a set F: removing F from G leaves a DAG (otherwise a directed cycle $v_1 \to ... \to v_l \to v_1$ would require each v_i^0 to be consumed before v_{i+1}^0, creating a circular register dependency that demands at least one additional LOAD for each cycle). Thus F is a feedback vertex set of size k.

Solution extraction. Given a target evaluation order (permutation of internal nodes), identify which leaves require a LOAD: leaf x^0 needs a LOAD iff the chain-start node x^1 is evaluated before some other internal node that uses x^0 as a right child. Set $c_x = 1$ for such vertices and $c_x = 0$ otherwise. □

Example: 3-cycle digraph: FVS of size 1 maps to an expression DAG needing 1 LOAD

Source: MinimumFeedbackVertexSet **Target:** MinimumCodeGenerationUnlimitedRegisters

```
$ pred create --example MinimumFeedbackVertexSet -o fvs.json
$ pred reduce fvs.json --to MinimumCodeGenerationUnlimitedRegisters -o bundle.json
$ pred solve bundle.json
$ pred evaluate fvs.json --config 1,0,0
```

Source FVS: $F = \{0\}$ (size 1) on a digraph with $n = 3$ vertices and $m = 3$ arcs
Target DAG: 6 vertices, left arcs L: $3 \to 0$, $4 \to 1$, $5 \to 2$, right arcs R: $3 \to 1$, $4 \to 2$, $5 \to 0$
Target evaluation order: (0, 1, 2) with 3 instructions ✓

Rule 3.57: (Maximum Clique (weighted) $\to$ Integer Linear Programming) A clique requires every pair of selected vertices to be adjacent; equivalently, no two selected vertices may share a *non*-edge. This is the independent set formulation on the complement graph $\overline{G}$.
Overhead: `num_vars = num_vertices`, `num_constraints = num_vertices^2`.

Proof: *Construction.* Variables: $x_v \in \{0, 1\}$ for each $v \in V$. The ILP is:

$$\begin{aligned} \max \quad & \sum_{v} x_v \\ \text{subject to} \quad & x_u + x_v \leq 1 \quad \forall (u, v) \notin E \\ & x_v \in \{0, 1\} \quad \forall v \in V \end{aligned}$$

.

Correctness. ($\Rightarrow$) In a clique, every pair of selected vertices is adjacent, so no non-edge constraint is violated. ($\Leftarrow$) Any feasible solution selects only mutually adjacent vertices, forming a clique; the objective maximizes its size.

Solution extraction. $K = \{v : x_v = 1\}$. □

Rule 3.58: (Knapsack $\to$ Integer Linear Programming) A 0-1 Knapsack instance is already a binary Integer Linear Program [151]: each item-selection bit becomes a binary variable, the capacity condition is a single linear inequality, and the value objective is linear. The reduction preserves the number of decision variables exactly, producing an ILP with n variables and one constraint.
Overhead: `num_vars = num_items`, `num_constraints = 1`.

Proof: *Construction.* Given nonnegative weights $w_0, ..., w_{n-1}$, nonnegative values $v_0, ..., v_{n-1}$, and capacity C, introduce binary variables $x_0, ..., x_{n-1} \in \{0, 1\}$ where $x_i = 1$ iff item i is selected. The ILP is:

$$\begin{aligned} \max \quad & \sum_{i=0}^{n-1} v_i x_i \\ \text{subject to} \quad & \sum_{i=0}^{n-1} w_i x_i \leq C \\ & x_i \in \{0, 1\} \quad \forall i \in \{0, ..., n-1\} \end{aligned}$$

. The target therefore has exactly n variables and one linear constraint.

Correctness. ($\Rightarrow$) Any feasible knapsack solution $\boldsymbol{x}$ satisfies $\sum_i w_i x_i \leq C$, so the same binary vector is feasible for the ILP and attains identical objective value $\sum_i v_i x_i$. ($\Leftarrow$) Any feasible binary ILP solution selects exactly the items with $x_i = 1$; the single inequality guarantees the chosen set fits in the knapsack, and the ILP objective equals the knapsack value. Therefore optimal solutions correspond one-to-one and preserve the optimum value.

Solution extraction. Identity: return the binary variable vector $\boldsymbol{x}$ as the knapsack selection. □

Example: $n = 4$ **items, capacity** $C = 7$

Source: Knapsack **Target:** ILP

```
$ pred create --example Knapsack -o knapsack.json
$ pred reduce knapsack.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate knapsack.json --config 0,1,1,0
```

Step 1 – Source instance. The canonical knapsack instance has weights (1, 3, 4, 5), values (1, 4, 5, 7), and capacity $C = 7$.

Step 2 – Build the binary ILP. Introduce one binary variable per item: x_0, x_1, x_2, $x_3 \in \{0, 1\}$. The objective is

$$\max 1x_0 + 4x_1 + 5x_2 + 7x_3$$

subject to the single capacity inequality

$$1x_0 + 3x_1 + 4x_2 + 5x_3 \leq 7$$

.

Step 3 – Verify a solution. The ILP optimum $\boldsymbol{x}^* = (0, 1, 1, 0)$ extracts directly to the knapsack selection $\boldsymbol{x}^* = (0, 1, 1, 0)$, choosing items $\{1, 2\}$. Their total weight is $3 + 4 = 7$ and their total value is $4 + 5 = 9$ ✓.

Uniqueness: The fixture stores one canonical optimal witness. For this instance the optimum is unique: items $\{1, 2\}$ are the only feasible choice achieving value 9.

Rule 3.59: (Integer Knapsack → Integer Linear Programming) This linear-size reduction reformulates Integer Knapsack as a non-negative integer program [151]: each item multiplicity becomes an ILP variable, the knapsack capacity is a single linear inequality, and explicit upper bounds $c_i \leq \lfloor \frac{B}{s_i} \rfloor$ preserve the exact witness domain of the source problem. The target therefore has n variables and $n + 1$ constraints. *Overhead:* `num_vars = num_items`, `num_constraints = num_items + 1`.

Proof: *Construction.* Given item sizes $s_0, ..., s_{n-1} \in \mathbb{Z}^+$, values $v_0, ..., v_{n-1} \in \mathbb{Z}^+$, and capacity $B \in \mathbb{N}$, introduce one non-negative integer variable c_i for each item. Add the capacity constraint

$$\sum_{i=0}^{n-1} s_i c_i \leq B$$

and, for each i, the upper bound

$$c_i \leq \lfloor \frac{B}{s_i} \rfloor.$$

Maximize the linear objective

$$\sum_{i=0}^{n-1} v_i c_i.$$

Correctness. ($\Rightarrow$) Any feasible Integer Knapsack multiplicity vector $\boldsymbol{c}$ already satisfies $\sum_i s_i c_i \leq B$, and every source multiplicity also satisfies $c_i \leq \lfloor \frac{B}{s_i} \rfloor$, so the same vector is feasible for the ILP and attains exactly the same objective value $\sum_i v_i c_i$. ($\Leftarrow$) Any feasible ILP solution satisfies the same capacity inequality and the same per-item multiplicity bounds, so it is a valid Integer Knapsack witness with identical total value. Therefore optimal solutions correspond one-to-one and preserve the optimum value.

Solution extraction. Identity: return the ILP variable vector $\boldsymbol{c}$ as the Integer Knapsack multiplicities. □

Example: $n = 3$ **items, capacity** $B = 10$

Source: IntegerKnapsack **Target:** ILP

```
$ pred create --example IntegerKnapsack -o integer-knapsack.json
$ pred reduce integer-knapsack.json --to ILP/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate integer-knapsack.json --config 0,0,2
```

Step 1 – Source instance. The canonical Integer Knapsack instance has sizes (3, 4, 5), values (4, 5, 7), and capacity $B = 10$.

Step 2 – Build the ILP. Introduce one integer variable per item multiplicity: c_0, c_1, $c_2 \in \mathbb{N}$. The capacity constraint is

$$3c_0 + 4c_1 + 5c_2 \leq 10$$

, and the explicit upper bounds are (3, 2, 2), i.e. $c_i \leq \lfloor \frac{B}{s_i} \rfloor$ for every item.

Step 3 – Verify the canonical witness. The ILP optimum is (0, 0, 2), which extracts identically to the source multiplicities (0, 0, 2). The selected terms contribute total size $10 \leq B$ and total value 14 ✓.

Uniqueness: The fixture stores one canonical optimum, here $(0, 0, 2)$.

Rule 3.60: (Maximum Clique (weighted) → Maximum Independent Set (weighted)) A clique in G is an independent set in the complement graph $\overline{G}$, where $\overline{G} = (V, \overline{E})$ with $\overline{E} = \{(u, v) : u \neq v, (u, v) \notin E\}$. This classical reduction [1] preserves vertices and weights; only the edge set changes.
Overhead: `num_vertices = num_vertices`, `num_edges = num_vertices * (num_vertices + -1 * 1) * 2^-1 + -1 * num_edges`.

Proof: *Construction.* Given MaximumClique instance $(G = (V, E), \boldsymbol{w})$ with $n = |V|$ and $m = |E|$, create MaximumIndependentSet instance $\left(\overline{G} = (V, \overline{E}), \boldsymbol{w}\right)$ where $\overline{E} = \{(u, v) : u \neq v, (u, v) \notin E\}$. The complement graph has $\frac{n(n-1)}{2} - m$ edges. Weights are preserved identically.

Correctness. ($\Rightarrow$) If S is a clique in G, then all pairs in S are adjacent in G, so no pair in S is adjacent in $\overline{G}$, making S an independent set in $\overline{G}$. ($\Leftarrow$) If S is an independent set in $\overline{G}$, then no pair in S is adjacent in $\overline{G}$, so all pairs in S are adjacent in G, making S a clique. Since both problems maximize $\sum_{v \in S} w_v$, optimal values coincide.

Solution extraction. Identity: the configuration is the same in both problems, since vertices are preserved one-to-one. □

Example: Path graph P_4**: clique in** G **maps to independent set in complement** $\overline{G}$**.**

Source: MaximumClique **Target:** MaximumIndependentSet

```
$ pred create --example MaximumClique -o maximumclique.json
$ pred reduce maximumclique.json --to MaximumIndependentSet/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate maximumclique.json --config 0,1,1,0
```

Rule 3.61: (Bin Packing (weighted) → Integer Linear Programming) The assignment-based formulation introduces a binary indicator for each item–bin pair and a binary variable for each bin being open. Assignment constraints ensure each item is placed in exactly one bin; capacity constraints link bin usage to item weights.
Overhead: `num_vars = num_items * num_items + num_items`, `num_constraints = 2 * num_items`.

Proof: *Construction.* Given n items with sizes $s_1, ..., s_n$ and bin capacity C:

Variables: $x_{ij} \in \{0,1\}$ for $i, j \in \{0, ..., n-1\}$: item i is assigned to bin j. $y_j \in \{0,1\}$: bin j is used. Total: $n^2 + n$ variables.

Constraints: (1) Assignment: $\sum_{j=0}^{n-1} x_{ij} = 1$ for each item i (each item in exactly one bin). (2) Capacity + linking: $\sum_{i=0}^{n-1} s_i \cdot x_{ij} \leq C \cdot y_j$ for each bin j (bin capacity respected; y_j forced to 1 if bin j is used).

Objective: Minimize $\sum_{j=0}^{n-1} y_j$.

The ILP is:

$$\begin{aligned} \min \quad & \sum_{j=0}^{n-1} y_j \\ \text{subject to} \quad & \sum_{j=0}^{n-1} x_{ij} = 1 \quad \forall i \in \{0, ..., n-1\} \\ & \sum_{i=0}^{n-1} s_i x_{ij} \leq C y_j \quad \forall j \in \{0, ..., n-1\} \\ & x_{ij}, y_j \in \{0,1\} \end{aligned}$$

.

Correctness. ($\Rightarrow$) A valid packing assigns each item to exactly one bin (satisfying (1)); each bin's load is at most C and $y_j = 1$ for any used bin (satisfying (2)). ($\Leftarrow$) Any feasible solution assigns each item to one bin by (1), respects capacity by (2), and the objective counts the number of open bins.

Solution extraction. For each item i, find the unique j with $x_{ij} = 1$; assign item i to bin j. □

Rule 3.62: (Integral Flow with Bundles → Integer Linear Programming) The feasibility conditions are already linear: one integer variable per arc, one inequality per bundle, one conservation equality per nonterminal vertex, and one lower bound on sink inflow.
Overhead: `num_vars = num_arcs`, `num_constraints = num_bundles + num_vertices + -1 * 1`.

Proof: *Construction.* Given Integral Flow with Bundles instance $\left(G = (V, A), s, t, (I_j, c_j)_{j=1}^{k}, R\right)$ with arc set $A = \{a_0, ..., a_{m-1}\}$, create one non-negative integer variable x_i for each arc a_i. The ILP therefore has m variables.

Bundle constraints. For every bundle I_j, add $\sum_{a_i \in I_j} x_i \leq c_j$.

Flow conservation. For every nonterminal vertex $v \in V \setminus \{s, t\}$, add $\sum_{a_i=(u,v)\in A} x_i - \sum_{a_i=(v,w)\in A} x_i = 0$.

Requirement constraint. Add the sink inflow lower bound $\sum_{a_i=(u,t)\in A} x_i - \sum_{a_i=(t,w)\in A} x_i \geq R$.

Objective. Minimize 0. The target is a pure feasibility ILP, so any constant objective works.

The ILP is:

$$\begin{aligned} \text{find} \quad & (x_i)_{i=0}^{m-1} \\ \text{subject to} \quad & \sum_{a_i \in I_j} x_i \leq c_j \quad \forall j \in \{1, ..., k\} \\ & \sum_{a_i=(u,v)\in A} x_i - \sum_{a_i=(v,w)\in A} x_i = 0 \quad \forall v \in V \setminus \{s, t\} \\ & \sum_{a_i=(u,t)\in A} x_i - \sum_{a_i=(t,w)\in A} x_i \geq R \\ & x_i \in \mathbb{Z}_{\geq 0} \quad \forall i \in \{0, ..., m-1\} \end{aligned}$$

.

Correctness. ($\Rightarrow$) Any satisfying bundled flow assigns a non-negative integer to each arc, satisfies every bundle inequality by definition, satisfies every nonterminal conservation equality, and yields sink inflow at least R, so it is a feasible ILP solution. ($\Leftarrow$) Any feasible ILP solution gives non-negative integral arc values obeying the same bundle, conservation, and sink-inflow constraints, hence it is a satisfying solution to the original Integral Flow with Bundles instance.

Solution extraction. Identity: read the ILP vector $(x_0, ..., x_{m-1})$ directly as the arc-flow vector of the source problem. □

Rule 3.63: (Sequencing to Minimize Weighted Completion Time → Integer Linear Programming) Completion times are natural integer variables, precedence constraints compare those completion times directly, and one binary order variable per task pair enforces that a single machine cannot overlap two jobs.
Overhead: `num_vars = num_tasks + num_tasks * (num_tasks + -1 * 1) * 2^-1`, `num_constraints = 2 * num_tasks + 3 * num_tasks * (num_tasks + -1 * 1) * 2^-1 + num_precedences`.

Proof: *Construction.* For each task j, introduce an integer completion-time variable C_j. For each unordered pair $i < j$, introduce a binary order variable y_{ij} with $y_{ij} = 1$ meaning task i finishes before task j. Let $M = \sum_h l_h$.

Bounds. $l_j \le C_j \le M$ for every task j, and $y_{ij} \in \{0, 1\}$.

Precedence constraints. If $i \preceq j$, require $C_j - C_i \ge l_j$.

Single-machine disjunction. For every pair $i < j$, require $C_j - C_i + M(1 - y_{ij}) \ge l_j$ and $C_i - C_j + My_{ij} \ge l_i$. Exactly one of the two orderings is therefore active.

Objective. Minimize $\sum_j w_j C_j$.

The ILP is:

$$\begin{aligned} \min \quad & \sum_j w_j C_j \\ \text{subject to} \quad & l_j \le C_j \le M \quad \forall j \\ & C_j - C_i \ge l_j \quad \forall i \preceq j \\ & C_j - C_i + M(1 - y_{ij}) \ge l_j \quad \forall i < j \\ & C_i - C_j + My_{ij} \ge l_i \quad \forall i < j \\ & y_{ij} \in \{0, 1\}, C_j \in \mathbb{Z}_{\ge 0} \end{aligned}$$

.

Correctness. ($\Rightarrow$) Any feasible schedule defines completion times and pairwise order values satisfying the bounds, precedence inequalities, and disjunctive machine constraints; its weighted completion time is exactly the ILP objective. ($\Leftarrow$) Any feasible ILP solution assigns a strict order to every task pair and forbids overlap, so the completion times correspond to a valid single-machine schedule that respects all precedences. Minimizing the ILP objective therefore minimizes the original weighted completion-time objective.

Solution extraction. Sort tasks by their completion times C_j and encode that order back into the source schedule representation. □

Rule 3.64: (Hamiltonian Circuit → Traveling Salesman (weighted)) [5] This $O(n^2)$ reduction constructs the complete graph on the same vertex set and uses edge weights to distinguish source edges from non-edges: weight 1 means "present in the source" and weight 2 means "missing in the source" ($n\frac{n-1}{2}$ target edges).
Overhead: `num_vertices = num_vertices`, `num_edges = num_vertices * (num_vertices + -1 * 1) * 2^-1`.

Proof: *Construction.* Given a Hamiltonian Circuit instance $G = (V, E)$ with $n = |V|$, construct the complete graph K_n on the same vertex set. For each pair $u < v$, set $w(u, v) = 1$ if $(u, v) \in E$ and $w(u, v) =$

2 otherwise. The target TSP instance asks for a minimum-weight Hamiltonian cycle in this weighted complete graph.

Correctness. ($\Rightarrow$) If G has a Hamiltonian circuit $v_0, v_1, ..., v_{n-1}, v_0$, then the same cycle exists in K_n. Every chosen edge belongs to E, so each edge has weight 1 and the resulting TSP tour has total cost n. ($\Leftarrow$) Every TSP tour on n vertices uses exactly n edges, and every target edge has weight at least 1, so any tour has cost at least n. If the optimum cost is exactly n, every selected edge must therefore have weight 1. Those edges are precisely edges of G, so the optimal TSP tour is already a Hamiltonian circuit in the source graph.

Solution extraction. Read the selected TSP edges, traverse the unique degree-2 cycle they form, and return the resulting vertex permutation as the source Hamiltonian-circuit witness. □

Example: Cycle graph on 4 vertices to weighted K_4

Source: HamiltonianCircuit **Target:** TravelingSalesman

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to TravelingSalesman/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2,3
```

Step 1 – Start from the source graph. The canonical source fixture is the cycle on vertices $\{0, 1, 2, 3\}$ with edges $(0, 1)$, $(1, 2)$, $(2, 3)$, $(3, 0)$. The stored Hamiltonian-circuit witness is the permutation [0, 1, 2, 3].

Step 2 – Complete the graph and encode adjacency by weights. The target keeps the same 4 vertices but adds the missing diagonals, so it becomes K_4 with 6 undirected edges. The original cycle edges $(0, 1)$, $(0, 3)$, $(1, 2)$, $(2, 3)$ receive weight 1, while the diagonals $(0, 2)$, $(1, 3)$ receive weight 2.

Step 3 – Verify the canonical witness. The stored target configuration [1, 0, 1, 1, 0, 1] selects the tour edges $(0, 1)$, $(0, 3)$, $(1, 2)$, $(2, 3)$. Its total cost is $1 + 1 + 1 + 1 = 4$, so every chosen edge is a weight-1 source edge, and traversing the selected cycle recovers the Hamiltonian circuit [0, 1, 2, 3].

Multiplicity: The fixture stores one canonical witness. For the 4-cycle there are $4 \times 2 = 8$ Hamiltonian-circuit permutations (choice of start vertex and direction), but they all induce the same undirected target edge set.

Rule 3.65: (Traveling Salesman (weighted) $\rightarrow$ Integer Linear Programming) A Hamiltonian tour is a permutation of vertices. Position-based encoding assigns each vertex a tour position via binary indicators, with permutation constraints ensuring a valid bijection. The tour cost involves products of position indicators for consecutive positions, which McCormick linearization converts to auxiliary variables with linear constraints.
Overhead: `num_vars = num_vertices^2 + 2 * num_vertices * num_edges`, `num_constraints = num_vertices^3 + -1 * 1 * num_vertices^2 + 2 * num_vertices + 4 * num_vertices * num_edges`.

Proof: *Construction.* For graph $G = (V, E)$ with $n = |V|$ and $m = |E|$:

Variables: Binary $x_{v,k} \in \{0, 1\}$ for each vertex $v \in V$ and position $k \in \{0, ..., n-1\}$. Interpretation: $x_{v,k} = 1$ iff vertex v is at position k in the tour.

Auxiliary variables: For each edge $(u, v) \in E$ and position k, introduce $y_{u,v,k}$ and $y_{v,u,k}$ to linearize the products $x_{u,k} \cdot x_{v,(k+1) \bmod n}$ and $x_{v,k} \cdot x_{u,(k+1) \bmod n}$ respectively.

Constraints: (1) Each vertex has exactly one position: $\sum_{k=0}^{n-1} x_{v,k} = 1$ for all $v \in V$. (2) Each position has exactly one vertex: $\sum_{v \in V} x_{v,k} = 1$ for all k. (3) Non-edge consecutive prohibition: if $\{v, w\} \notin E$, then $x_{v,k} + x_{w,(k+1) \bmod n} \leq 1$ for all k. (4) McCormick: $y \leq x_{v,k}$, $y \leq x_{w,(k+1) \bmod n}$, $y \geq x_{v,k} + x_{w,(k+1) \bmod n} - 1$.

Objective: Minimize $\sum_{(u,v) \in E} w(u, v) \cdot \sum_k \left(y_{u,v,k} + y_{v,u,k}\right)$.

The ILP is:

$$
\begin{aligned}
\min \quad & \sum_{(u,v)\in E} w(u,v) \sum_{k} \left(y_{u,v,k} + y_{v,u,k}\right) \\
\text{subject to} \quad & \sum_{k=0}^{n-1} x_{v,k} = 1 \quad \forall v \in V \\
& \sum_{v\in V} x_{v,k} = 1 \quad \forall k \in \{0,...,n-1\} \\
& x_{v,k} + x_{w,(k+1) \bmod n} \leq 1 \quad \forall \{v,w\} \notin E, k \in \{0,...,n-1\} \\
& y_{u,v,k} \leq x_{u,k} \quad \forall (u,v) \in E, k \in \{0,...,n-1\} \\
& y_{u,v,k} \leq x_{v,(k+1) \bmod n} \quad \forall (u,v) \in E, k \in \{0,...,n-1\} \\
& y_{u,v,k} \geq x_{u,k} + x_{v,(k+1) \bmod n} - 1 \quad \forall (u,v) \in E, k \in \{0,...,n-1\} \\
& x_{v,k}, y_{u,v,k} \in \{0,1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A valid tour defines a permutation matrix $\left(x_{v,k}\right)$ satisfying constraints (1)–(2); consecutive vertices are adjacent by construction, so (3) holds; McCormick constraints (4) force $y = x_{u,k}x_{v,k+1}$, making the objective equal to the tour cost. ($\Leftarrow$) Any feasible binary solution defines a permutation (by (1)–(2)) where consecutive positions are connected by edges (by (3)), forming a Hamiltonian tour; the linearized objective equals the tour cost.

Solution extraction. For each position k, find vertex v with $x_{v,k} = 1$ to recover the tour permutation; then select edges between consecutive positions. □

Example: Weighted K_4: the optimal tour $0 \to 1 \to 3 \to 2 \to 0$ with cost 80 is found by position-based ILP.

Source: TravelingSalesman **Target:** ILP

```
$ pred create --example TSP -o tsp.json
$ pred reduce tsp.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate tsp.json --config 1,1,0,0,1,1
```

Rule 3.66: (Longest Path (weighted) → Integer Linear Programming) A simple s-t path can be represented as one unit of directed flow from s to t on oriented copies of the undirected edges. Integer order variables then force the selected arcs to move strictly forward, which forbids detached directed cycles.
Overhead: `num_vars = 2 * num_edges + num_vertices`, `num_constraints = 5 * num_edges + 4 * num_vertices + 1`.

Proof: *Construction.* For graph $G = (V, E)$ with $n = |V|$ and $m = |E|$:

Variables: For each undirected edge $\{u, v\} \in E$, introduce two binary arc variables $x_{u,v}, x_{v,u} \in \{0, 1\}$. Interpretation: $x_{u,v} = 1$ iff the path traverses edge $\{u, v\}$ from u to v. For each vertex $v \in V$, add an integer order variable $o_v \in \{0, ..., n-1\}$. Total: $2m + n$ variables.

Constraints: (1) Flow balance: $\sum_{w:\{v,w\}\in E} x_{v,w} - \sum_{u:\{u,v\}\in E} x_{u,v} = 1$ at the source, equals -1 at the target, and equals 0 at every other vertex. (2) Degree bounds: every vertex has at most one selected outgoing arc and at most one selected incoming arc. (3) Edge exclusivity: $x_{u,v} + x_{v,u} \leq 1$ for each undirected edge. (4) Ordering: for every oriented edge $u \to v$, $o_v - o_u \geq 1 - n\left(1 - x_{u,v}\right)$. (5) Anchor the path at the source with $o_s = 0$.

Objective. Maximize $\sum_{\{u,v\}\in E} l(\{u, v\}) \cdot \left(x_{u,v} + x_{v,u}\right)$.

The ILP is:

$$\begin{aligned}
\max \quad & \sum_{\{u,v\}\in E} l(\{u,v\})\big(x_{u,v}+x_{v,u}\big) \\
\text{subject to} \quad & \sum_{w:\{v,w\}\in E} x_{v,w} - \sum_{u:\{u,v\}\in E} x_{u,v} = b_v \quad \forall v \in V \\
& \sum_{w:\{v,w\}\in E} x_{v,w} \le 1 \quad \forall v \in V \\
& \sum_{u:\{u,v\}\in E} x_{u,v} \le 1 \quad \forall v \in V \\
& x_{u,v} + x_{v,u} \le 1 \quad \forall \{u,v\} \in E \\
& o_v - o_u \ge 1 - n\big(1 - x_{u,v}\big) \quad \forall u \to v \\
& o_s = 0 \\
& x_{u,v} \in \{0,1\}, o_v \in \{0,\ldots,n-1\}
\end{aligned}$$

, where $b_s = 1$, $b_t = -1$, and $b_v = 0$ otherwise.

Correctness. ($\Rightarrow$) Any simple s-t path can be oriented from s to t, giving exactly one outgoing arc at s, one incoming arc at t, balanced flow at every internal vertex, and strictly increasing order values along the path. ($\Leftarrow$) Any feasible ILP solution satisfies the flow equations and degree bounds, so the selected arcs form vertex-disjoint directed paths and cycles. The ordering inequalities make every selected arc increase the order value by at least 1, so directed cycles are impossible. The only remaining positive-flow component is therefore a single directed s-t path, whose objective is exactly the total selected edge length.

Solution extraction. For each undirected edge $\{u,v\}$, select it in the source configuration iff either $x_{u,v}$ or $x_{v,u}$ is 1. □

Example: The 3-vertex path $0 \to 1 \to 2$ encoded as a 7-variable ILP with optimum 5.

Source: LongestPath **Target:** ILP

```
$ pred create --example LongestPath -o longest-path.json
$ pred reduce longest-path.json --to ILP/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate longest-path.json --config 1,1
```

Step 1 – Orient each undirected edge. The canonical witness has two source edges, so the reduction creates four directed-arc variables. The optimal witness sets $x_{0,1} = 1$ and $x_{1,2} = 1$, leaving the reverse directions at 0.

Step 2 – Add order variables. The target has 7 variables and 23 constraints in total. The order block $\boldsymbol{o} = (0, 1, 2)$ certifies the increasing path positions $0 < 1 < 2$.

Step 3 – Check the objective. The target witness $\boldsymbol{z} = (1, 0, 1, 0, 0, 1, 2)$ selects lengths 2 and 3, so the ILP objective is 5, matching the source optimum. ✓

Rule 3.67: (Traveling Salesman (weighted) → QUBO (real-weighted)) Position-based QUBO encoding [144] maps a Hamiltonian tour to n^2 binary variables $x_{v,p}$, where $x_{v,p} = 1$ iff city v is visited at position p. The QUBO Hamiltonian $H = H_A + H_B + H_C$ combines permutation constraints with the distance objective (n^2 variables indexed by $v \cdot n + p$).
Overhead: `num_vars = num_vertices^2`.

Proof: *Construction.* For graph $G = (V, E)$ with $n = |V|$ and edge weights w_{uv}. Let $A = 1 + \sum_{(u,v)\in E} |w_{uv}|$ be the penalty coefficient.

Variables: Binary $x_{v,p} \in \{0,1\}$ for vertex $v \in V$ and position $p \in \{0,\ldots,n-1\}$. QUBO variable index: $v \cdot n + p$.

QUBO matrix: (1) Row constraint $H_A = A\sum_v \left(1 - \sum_p x_{v,p}\right)^2$: diagonal $Q[vn+p, vn+p] \mathrel{+}= -A$, off-diagonal $Q[vn+p, vn+p'] \mathrel{+}= 2A$ for $p < p'$. (2) Column constraint $H_B = A\sum_p \left(1 - \sum_v x_{v,p}\right)^2$: symmetric to H_A. (3) Distance $H_C = \sum_{(u,v)\in E} w_{uv} \sum_p \left(x_{u,p}x_{v,(p+1) \bmod n} + x_{v,p}x_{u,(p+1) \bmod n}\right)$. For non-edges, penalty A replaces w_{uv}.

Correctness. ($\Rightarrow$) A valid tour defines a permutation matrix satisfying $H_A = H_B = 0$; the H_C terms sum to the tour cost. ($\Leftarrow$) The minimum-energy state has $H_A = H_B = 0$ (penalty A exceeds any tour cost), so it encodes a valid permutation; H_C equals the tour cost, selecting the shortest tour.

Solution extraction. From QUBO solution x^*, for each position p find the unique vertex v with $x^*_{vn+p} = 1$. Map consecutive position pairs to edge indices. □

Example: TSP on K_3 with weights $w_{01} = 1$, $w_{02} = 2$, $w_{12} = 3$: the QUBO ground state encodes the optimal tour with cost $1 + 2 + 3 = 6$.

Source: TravelingSalesman **Target:** QUBO

```
$ pred create --example TSP -o tsp.json
$ pred reduce tsp.json --to QUBO/f64 -o bundle.json
$ pred solve bundle.json
$ pred evaluate tsp.json --config 1,1,1
```

Step 1 – Encode each tour position as a binary variable. A tour is a permutation of n vertices. Introduce $n^2 = 9$ binary variables $x_{v,p}$: vertex v is at position p.

$$\underbrace{x_{0,0}x_{0,1}x_{0,2}}_{\text{vertex 0}}\ \underbrace{x_{1,0}x_{1,1}x_{1,2}}_{\text{vertex 1}}\ \underbrace{x_{2,0}x_{2,1}x_{2,2}}_{\text{vertex 2}}$$

Step 2 – Penalize invalid permutations. The penalty $A = 1 + |w_{01}| + |w_{02}| + |w_{12}| = 1 + 1 + 2 + 3 = 7$ ensures any row/column constraint violation outweighs any tour cost. Row constraints (each vertex at exactly one position) and column constraints (each position has one vertex) contribute diagonal -7 and off-diagonal $+14$ within each group.

Step 3 – Encode edge costs. For each edge (u, v) and position p, the products $x_{u,p}x_{v,(p+1) \bmod 3}$ and $x_{v,p}x_{u,(p+1) \bmod 3}$ add the edge weight w_{uv} when vertices u, v are consecutive in the tour. Since K_3 is complete, all pairs are edges with their actual weights.

Step 4 – Verify a solution. The QUBO ground state $\boldsymbol{x} = (0, 0, 1, 1, 0, 0, 0, 1, 0)$ encodes a valid tour. Reading the permutation: each 3-bit group has exactly one 1 (valid permutation ✓). The tour cost equals $w_{01} + w_{02} + w_{12} = 1 + 2 + 3 = 6$.

Multiplicity: The fixture stores one canonical optimal witness. On K_3 with distinct edge weights $1, 2, 3$, every Hamiltonian cycle has cost $1 + 2 + 3 = 6$ (all edges used), and 3 cyclic tours × 2 directions yield 6 permutation matrices overall.

Rule 3.68: (Longest Common Subsequence → Maximum Independent Set) A match-node construction transforms a k-string LCS instance into a Maximum Independent Set problem on a conflict graph. Each vertex represents a k-tuple of positions (one per string) that all share the same character, and edges connect pairs that cannot coexist in any valid common subsequence. The MIS of this graph equals the LCS length.
Overhead: `num_vertices = cross_frequency_product`, `num_edges = cross_frequency_product^2`.

Proof: *Construction.* Given k strings $s_1, \ldots, s_k$ over alphabet Σ (size $|\Sigma|$):

Vertices: For each character $c \in \Sigma$, create a vertex for every k-tuple $(p_1, ..., p_k)$ where $s_i[p_i] = c$ for all i. The total vertex count equals $\sum_{c \in \Sigma} \prod_{i=1}^{k} \text{count}(c, s_i)$.

Edges: Two vertices $u = (a_1, ..., a_k)$ and $v = (b_1, ..., b_k)$ are connected if they *conflict* — they cannot both appear in a valid common subsequence. A conflict occurs when the position differences are not consistently ordered: $\neg(\forall i : a_i < b_i)$ and $\neg(\forall i : a_i > b_i)$.

Correctness. ($\Rightarrow$) A common subsequence of length ℓ selects ℓ match nodes whose positions are strictly increasing in every string, so no two are adjacent — forming an independent set of size ℓ. ($\Leftarrow$) An independent set of size ℓ consists of ℓ mutually non-conflicting match nodes, meaning their positions are consistently ordered across all strings. Sorting by any string's position yields a valid common subsequence of length ℓ.

Solution extraction. Sort the selected vertices by position in s_1. Read off the characters to obtain the common subsequence, then pad to `max_length` with the padding symbol. □

Example: LCS of two strings over a 3-symbol alphabet

Source: LongestCommonSubsequence **Target:** MaximumIndependentSet

```
$ pred create --example LCS -o lcs.json
$ pred reduce lcs.json --to MaximumIndependentSet/SimpleGraph/One -o bundle.json
$ pred solve bundle.json
$ pred evaluate lcs.json --config 1,0,2,3
```

Source LCS: config (1, 0, 2, 3)
Target MIS: $S = \{2, 4, 5\}$ (size 3)
MIS size = LCS length = 3 ✓

Rule 3.69: (Longest Common Subsequence → Integer Linear Programming) An optimization ILP formulation maximizes the length of a common subsequence. Binary variables choose a symbol (or padding) at each witness position. Match variables link active positions to source string indices, and the objective maximizes the number of non-padding positions.
Overhead: `num_vars = max_length * (alphabet_size + 1) + max_length * total_length`, `num_constraints = max_length + num_transitions + max_length * num_strings + max_length * total_length + num_transitions * sum_triangular_lengths`.

Proof: *Construction.* Given alphabet Σ (size k), strings $R = \{r_1, ..., r_m\}$, and maximum length $L = \min_i |r_i|$:

Variables: Binary $x_{p,a} \in \{0, 1\}$ for witness position $p \in \{1, ..., L\}$ and symbol $a \in \Sigma \cup \{\bot\}$ (where $\bot$ is the padding symbol), with $x_{p,a} = 1$ iff position p holds symbol a. For every input string r_i, witness position p, and source index $j \in \{1, ..., |r_i|\}$, binary $y_{i,p,j} = 1$ iff position p is matched to index j of r_i.

Constraints: (1) Exactly one symbol (including padding) per position: $\sum_{a \in \Sigma \cup \{\bot\}} x_{p,a} = 1$ for all p. (2) Contiguity: $x_{p+1,\bot} \geq x_{p,\bot}$ for consecutive positions. (3) Conditional matching: $\sum_{j=1}^{|r_i|} y_{i,p,j} + x_{p,\bot} = 1$ for each (i, p), so active positions select exactly one match and padding positions select none. (4) Character consistency: $y_{i,p,j} \leq x_{p,r_{i[j]}}$. (5) Strictly increasing matches: for consecutive positions p and $p+1$, forbid $y_{i,p,j'} = y_{i,p+1,j} = 1$ whenever $j' \geq j$.

Objective: Maximize $\sum_p \sum_{a \in \Sigma} x_{p,a}$ (the number of non-padding positions).

The ILP is:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{p} \sum_{a \in \Sigma} x_{p,a} \\
\text{subject to} \quad & \sum_{a \in \Sigma \cup \{\bot\}} x_{p,a} = 1 \quad \forall p \in \{1, ..., L\} \\
& x_{p+1,\bot} \geq x_{p,\bot} \quad \forall p \\
& \sum_{j=1}^{|r_i|} y_{i,p,j} + x_{p,\bot} = 1 \quad \forall i, p \\
& y_{i,p,j} \leq x_{p,r_i[j]} \quad \forall i, p, j \\
& y_{i,p,j'} + y_{i,p+1,j} \leq 1 \quad \forall i, p, j' \geq j \\
& x_{p,a}, y_{i,p,j} \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Given an optimal common subsequence w of length ℓ, set $x_{p,w_p} = 1$ for $p \leq \ell$ and $x_{p,\bot} = 1$ for $p > \ell$. For active positions, choose the embedding indices in each source string. All constraints are satisfied and the objective equals ℓ. ($\Leftarrow$) Any optimal ILP solution selects contiguous non-padding positions followed by padding. The active prefix, together with character consistency and ordering constraints, forms a valid common subsequence whose length equals the objective value.

Solution extraction. For each position p, read the selected symbol a (which may be $\bot$). The resulting length-L vector with padding is the source configuration. □

Rule 3.70: (Minimum Multiway Cut (weighted) $\rightarrow$ Integer Linear Programming) The vertex-assignment + edge-cut indicator formulation [152] introduces binary variables for vertex-to-component membership and edge-cut indicators. Terminal vertices are fixed to their own components, partition constraints ensure every vertex belongs to exactly one component, and linking inequalities force the cut indicator on whenever an edge's endpoints are in different components.
Overhead: `num_vars = num_terminals * num_vertices + num_edges`, `num_constraints = num_vertices + 2 * num_terminals * num_edges + num_terminals * num_terminals`.

Proof: *Construction.* Given graph $G = (V, E, w)$ with $n = |V|$ vertices, $m = |E|$ edges, edge weights $w_e > 0$, and k terminals $T = \{t_0, ..., t_{k-1}\}$:

Variables: (1) $y_{iv} \in \{0, 1\}$ for $i \in \{0, ..., k-1\}$, $v \in V$: vertex v belongs to the component of terminal t_i. (2) $x_e \in \{0, 1\}$ for $e \in E$: edge e is in the cut. Total: $kn + m$ variables.

Constraints: (1) Terminal fixing: $y_{i,t_i} = 1$ for each i (terminal t_i is in its own component); $y_{j,t_i} = 0$ for $j \neq i$ (each terminal excluded from other components). (2) Partition: $\sum_{i=0}^{k-1} y_{iv} = 1$ for each $v \in V$ (each vertex in exactly one component). (3) Edge-cut linking: for each edge $e = (u, v)$ and each terminal i: $x_e \geq y_{iu} - y_{iv}$ and $x_e \geq y_{iv} - y_{iu}$ (force $x_e = 1$ when endpoints are in different components). Total: $k^2 + n + 2km$ constraints.

Objective: Minimize $\sum_{e \in E} w_e \cdot x_e$.

The ILP is:

$$
\begin{aligned}
\min \quad & \sum_{e \in E} w_e x_e \\
\text{subject to} \quad & y_{i,t_i} = 1 \quad \forall i \in \{0, ..., k-1\} \\
& y_{j,t_i} = 0 \quad \forall i \neq j \\
& \sum_{i=0}^{k-1} y_{iv} = 1 \quad \forall v \in V \\
& x_e \geq y_{iu} - y_{iv} \quad \forall e = (u, v) \in E, i \in \{0, ..., k-1\} \\
& x_e \geq y_{iv} - y_{iu} \quad \forall e = (u, v) \in E, i \in \{0, ..., k-1\} \\
& x_e, y_{iv} \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A multiway cut C partitions V into k components, one per terminal. Setting $y_{iv} = 1$ iff v is in t_i's component and $x_e = 1$ iff $e \in C$ satisfies all constraints: partition by construction, terminal fixing by definition, and linking because any edge with endpoints in different components is in C. The objective equals the cut weight. ($\Leftarrow$) Any feasible ILP solution defines a valid partition (by constraint (2)) separating all terminals (by constraint (1)). The linking constraints (3) force $x_e = 1$ for all cross-component edges, so the objective is at least the multiway cut weight; minimization ensures optimality.

Solution extraction. For each edge e at index idx, read $x_e = x^*_{kn+\text{ idx}}$. The source configuration is config$[e] = x_e$ (1 = cut, 0 = keep). □

Rule 3.71: (Steiner Tree (weighted) $\rightarrow$ Integer Linear Programming) The rooted multi-commodity flow formulation [153], [154] introduces one binary selector y_e for each source edge and, for every non-root terminal t, one binary flow variable on each directed source edge. Flow conservation sends one unit from the root to each terminal, while the linking inequalities $f^t_{u,v} \leq y_e$ ensure that every used flow arc is backed by a selected source edge. The resulting binary ILP has $m + 2m(k-1)$ variables and $n(k-1) + 2m(k-1)$ constraints.
Overhead: `num_vars = num_edges + 2 * num_edges * (num_terminals + -1 * 1)`, `num_constraints = num_vertices * (num_terminals + -1 * 1) + 2 * num_edges * (num_terminals + -1 * 1)`.

Proof: *Construction.* Given an undirected weighted graph $G = (V, E, w)$ with strictly positive edge weights, terminals $T = \{t_0, ..., t_{k-1}\}$, and root $r = t_0$, introduce binary edge selectors $y_e \in \{0, 1\}$ for every $e \in E$. For each non-root terminal $t \in T \setminus \{r\}$ and each directed copy of an undirected edge $(u, v) \in E$, introduce a binary flow variable $f^t_{u,v} \in \{0, 1\}$. The target objective is

$$\min \sum_{e \in E} w_e y_e.$$

For every commodity t and vertex v, enforce flow conservation:

$$\sum_{u:(u,v) \in A} f^t_{u,v} - \sum_{u:(v,u) \in A} f^t_{v,u} = b_{t,v},$$

where A contains both orientations of every undirected edge, $b_{t,v} = -1$ at the root $v = r$, $b_{t,v} = 1$ at the sink $v = t$, and $b_{t,v} = 0$ otherwise. For every commodity t and undirected edge $e = \{u, v\}$, add the capacity-linking inequalities

$$f^t_{u,v} \leq y_e \quad \text{and} \quad f^t_{v,u} \leq y_e.$$

Binary flow variables suffice because any Steiner tree yields a unique simple root-to-terminal path for each commodity, so every commodity can be realized as a 0/1 path indicator.

The ILP is:

$$
\begin{aligned}
\text{min} \quad & \sum_{e \in E} w_e y_e \\
\text{subject to} \quad & \sum_{u:(u,v)\in A} f^t_{u,v} - \sum_{u:(v,u)\in A} f^t_{v,u} = b_{t,v} \quad \forall t \in T \setminus \{r\}, v \in V \\
& f^t_{u,v} \le y_e \quad \forall t \in T \setminus \{r\}, e = \{u,v\} \in E \\
& f^t_{v,u} \le y_e \quad \forall t \in T \setminus \{r\}, e = \{u,v\} \in E \\
& y_e, f^t_{u,v} \in \{0,1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) If $S \subseteq E$ is a Steiner tree, set $y_e = 1$ exactly for $e \in S$. For each non-root terminal t, the unique path from r to t inside the tree defines a binary flow assignment satisfying the conservation equations, and every used arc lies on a selected edge, so all linking inequalities hold. The ILP objective equals $\sum_{e \in S} w_e$. ($\Leftarrow$) Any feasible ILP solution with edge selector set $Y = \{e \in E : y_e = 1\}$ supports one unit of flow from r to every non-root terminal, so the selected edges contain a connected subgraph spanning all terminals. Because all edge weights are strictly positive, any cycle in the selected subgraph has positive total cost; the optimizer therefore never includes redundant edges, so the selected subgraph is already a Steiner tree. Therefore an optimal ILP solution induces a minimum-cost Steiner tree.

Variable mapping. The first m ILP variables are the source-edge indicators $y_0, ..., y_{m-1}$ in source edge order. For terminal t_p with $p \in \{1, ..., k-1\}$, the next block of $2m$ variables stores the directed arc indicators $f^{t_p}_{u,v}$ and $f^{t_p}_{v,u}$ for each source edge (u, v).

Solution extraction. Read the first m target variables as the source edge-selection vector. Since those coordinates are exactly the y_e variables, the extracted source configuration is valid whenever the selected subgraph is pruned to its Steiner tree witness.

Remark. Zero-weight edges are excluded because they allow degenerate optimal ILP solutions containing redundant cycles at no cost; following the convention of practical solvers (e.g., SCIP-Jack [154]), such edges should be contracted before applying the reduction. □

Example: Canonical Steiner tree instance ($n = 5$, $m = 7$, $|T| = 3$)

Source: SteinerTree **Target:** ILP

```
$ pred create --example SteinerTree -o steinertree.json
$ pred reduce steinertree.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate steinertree.json --config 1,1,1,1,0,0,0
```

Step 1 – Choose a root and one commodity per remaining terminal. The canonical source instance has terminals $T = \{v_0, v_2, v_4\}$. The reduction fixes the first terminal as root $r = v_0$ and creates one flow commodity for each remaining terminal: v_2 and v_4.

Step 2 – Count the variables from the source edge order. The first 7 target variables are the edge selectors $\boldsymbol{y} = (1, 1, 1, 1, 0, 0, 0)$, one per source edge in the order $e_0 = (0, 1)$, $e_1 = (1, 2)$, $e_2 = (1, 3)$, $e_3 = (3, 4)$, $e_4 = (0, 3)$, $e_5 = (3, 2)$, $e_6 = (2, 4)$. The remaining 28 variables are directed flow indicators: $2m(|T| - 1) = 2 \times 7 \times 2 = 28$.

Step 3 – Count the constraints commodity-by-commodity. Each non-root terminal contributes one flow-conservation equality per vertex and two capacity inequalities per source edge. For this fixture that is $5 \times 2 = 10$ equalities plus $14 \times 2 = 28$ inequalities, totaling 38 constraints.

Step 4 – Read the canonical witness pair. The source witness selects edges $\{(v_0, v_1), (v_1, v_2), (v_1, v_3), (v_3, v_4)\}$, so $\boldsymbol{y}$ already encodes the Steiner tree. In the target witness, the

commodity for v_2 routes along $v_0 \to v_1 \to v_2$, while the commodity for v_4 routes along $v_0 \to v_1 \to v_3 \to v_4$. Every flow 1-entry therefore sits under a selected edge variable ✓

Step 5 – Verify the objective end-to-end. The selected-edge prefix is $\boldsymbol{y} = (1, 1, 1, 1, 0, 0, 0)$, matching the source witness (1, 1, 1, 1, 0, 0, 0). The ILP objective is $2+2+1+1 = 6$, exactly the Steiner tree optimum stored in the fixture.

Multiplicity: The fixture stores one canonical witness. Other optimal Steiner trees could yield different feasible ILP witnesses, but every valid witness still exposes the source solution in the first m variables.

Rule 3.72: (Minimum Vertex Cover → Minimum Hitting Set) Vertex Cover is the special case of Hitting Set where every set has exactly two elements [5]. Given a unit-weight VC instance $G = (V, E)$, let the universe $U = V$ and define one 2-element subset $\{u, v\}$ per edge $(u, v) \in E$. The budget is unchanged.
Overhead: `universe_size = num_vertices`, `num_sets = num_edges`.

Proof: *Construction.* Given unit-weight VC instance (G, k) with $G = (V, E)$, construct Hitting Set instance $(U, \mathcal{S}, k)$:

- Universe: $U = V$ with $|U| = |V|$ elements.
- Collection: $\mathcal{S} = \{\{u, v\} : (u, v) \in E\}$ with $|\mathcal{S}| = |E|$ subsets, each of size 2.
- Budget: $k' = k$ (unchanged).

Correctness. ($\Rightarrow$) If $C \subseteq V$ is a vertex cover, then for every edge $(u, v) \in E$, at least one of u, v lies in C, so C intersects the subset $\{u, v\} \in \mathcal{S}$. Hence C is a hitting set. ($\Leftarrow$) If $H \subseteq U$ hits every subset $\{u, v\} \in \mathcal{S}$, then for every edge $(u, v) \in E$, H contains u or v, so H is a vertex cover.

Since both problems minimise cardinality (unit weights), an optimal vertex cover of size k corresponds to an optimal hitting set of the same size.

Solution extraction. The hitting set H is directly the vertex cover: $c_v = h_v$ for each $v \in V$. □

Example: Unit-weight VC to Hitting Set ($n = 6$, $|E| = 8$)

Source: MinimumVertexCover **Target:** MinimumHittingSet

```
$ pred create --example 'MVC {weight: One}' -o mvc.json
$ pred reduce mvc.json --to MinimumHittingSet -o bundle.json
$ pred solve bundle.json
$ pred evaluate mvc.json --config 1,0,0,1,1,0
```

Source VC: $C = \{0, 3, 4\}$ (size 3) Target HS: $H = \{0, 3, 4\}$ (size 3)
The hitting set H is identical to the vertex cover C because the universe elements are the vertices and the subsets are the edges.

Rule 3.73: (Minimum Hitting Set → Integer Linear Programming) Each set must contain at least one selected element – a standard set-covering constraint on the element indicators.
Overhead: `num_vars = universe_size`, `num_constraints = num_sets`.

Proof: *Construction.* Variables: $x_e \in \{0, 1\}$ for each element $e \in U$. The ILP is:

$$\begin{aligned} \min \quad & \sum_{e} x_e \\ \text{subject to} \quad & \sum_{e \in S} x_e \geq 1 \quad \forall S \in \mathcal{S} \\ & x_e \in \{0, 1\} \quad \forall e \in U \end{aligned}$$

.

Correctness. ($\Rightarrow$) A hitting set includes at least one element from each set. ($\Leftarrow$) Any feasible solution hits every set.

Solution extraction. $H = \{e : x_e = 1\}$. □

Rule 3.74: (Exact Cover by 3-Sets $\to$ Integer Linear Programming) Each element must be covered by exactly one triple, and the number of selected triples must equal $|U|/3$.
Overhead: `num_vars = num_subsets`, `num_constraints = universe_size + 1`.

Proof: *Construction.* Variables: $x_j \in \{0,1\}$ for each triple T_j. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{j : e \in T_j} x_j = 1 \quad \forall e \in U \\ & \sum_j x_j = |U| \frac{|}{3} \\ & x_j \in \{0,1\} \quad \forall j \end{aligned}$$

.

Correctness. The equality constraints force each element to appear in exactly one selected triple, which is the definition of an exact cover.

Solution extraction. $\mathcal{C} = \{T_j : x_j = 1\}$. □

Rule 3.75: (NAE-SAT $\to$ Integer Linear Programming) Each clause must have at least one true and at least one false literal, encoded as a pair of linear inequalities per clause.
Overhead: `num_vars = num_vars`, `num_constraints = 2 * num_clauses`.

Proof: *Construction.* Variables: $x_i \in \{0,1\}$ per Boolean variable. For each clause C with literals $l_1, ..., l_k$, substitute $l_i = x_i$ for positive and $l_i = 1 - x_i$ for negative literals. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_i \text{coeff}_{C,i} x_i \geq 1 - \text{neg}(C) \quad \text{for each clause } C \\ & \sum_i \text{coeff}_{C,i} x_i \leq |C| - 1 - \text{neg}(C) \quad \text{for each clause } C \\ & x_i \in \{0,1\} \quad \forall i \end{aligned}$$

.

Correctness. The two constraints per clause jointly enforce the not-all-equal condition.

Solution extraction. Direct: $x_i = 1$ iff variable i is true. □

Rule 3.76: (Set Splitting $\to$ Integer Linear Programming) Each subset must contain at least one element of each color, encoded as a pair of linear inequalities per subset.
Overhead: `num_vars = universe_size`, `num_constraints = 2 * num_subsets`.

Proof: *Construction.* Variables: $x_i \in \{0,1\}$ for each $u_i \in U$ ($x_i = 1$ means $u_i \in S_2$). For each subset $C = \{i_1, ..., i_k\}$ the ILP enforces:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{j \in C} x_j \geq 1 \quad \text{for each } C \in \mathcal{C} \\ & \sum_{j \in C} x_j \leq |C| - 1 \quad \text{for each } C \in \mathcal{C} \\ & x_i \in \{0,1\} \quad \forall i \end{aligned}$$

.

Correctness. ($\Rightarrow$) A valid splitting has at least one element in S_2 ($\sum \geq 1$) and at least one in S_1 ($\sum \leq |C| - 1$) for every C. ($\Leftarrow$) Any feasible ILP solution defines a valid 2-coloring.

Solution extraction. $S_2 = \{u_i : x_i = 1\}$, $S_1 = U \setminus S_2$. □

Rule 3.77: (Monochromatic Triangle $\rightarrow$ Integer Linear Programming) This $O(m + t)$ reduction uses one binary variable per edge and two linear inequalities per triangle, where $m = |E|$ and t is the number of triangles in the source graph. The target ILP is a pure feasibility problem with m variables and $2t$ constraints.
Overhead: `num_vars = num_edges`, `num_constraints = 2 * num_triangles`.

Proof: *Construction.* Let the source graph edges be indexed as $e_0, ..., e_{m-1}$. Introduce binary variables $x_0, ..., x_{m-1}$, where $x_i = 0$ or 1 is the color assigned to edge e_i. For every triangle $T = \{e_a, e_b, e_c\}$ in the source graph, add the pair of inequalities

$$x_a + x_b + x_c \geq 1 \quad \text{and} \quad x_a + x_b + x_c \leq 2.$$

The objective is empty, so the ILP asks only for feasibility.

Correctness. ($\Rightarrow$) Any triangle-free 2-edge-coloring assigns each triangle at least one edge of color 0 and at least one edge of color 1, so the corresponding sum is either 1 or 2 and both inequalities hold. ($\Leftarrow$) Any feasible ILP assignment gives a 0/1 color to every edge, and the triangle bounds forbid sums 0 and 3, so no triangle is monochromatic.

Solution extraction. Return the ILP variables unchanged as the target edge-coloring vector. □

Example: K_4 with $n = 4$ vertices, $m = 6$ edges, and 4 triangles

Source: MonochromaticTriangle **Target:** ILP

```
$ pred create --example MonochromaticTriangle/SimpleGraph -o monochromatic-triangle.json
$ pred reduce monochromatic-triangle.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate monochromatic-triangle.json --config 1,1,0,0,0,1
```

Step 1 – Source instance. The canonical Monochromatic Triangle fixture is K_4 on vertices $0, 1, 2, 3$ with edges $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$. It has 4 triangles, so the reduction creates one pair of inequalities for each of those four triangles.

Step 2 – Build the ILP. Introduce one binary variable per edge, so the target has $m = 6$ variables. For each triangle, add the lower bound $x_a + x_b + x_c \geq 1$ and the upper bound $x_a + x_b + x_c \leq 2$, giving 8 total constraints.

Step 3 – Verify a witness. The stored ILP witness is (1, 1, 0, 0, 0, 1). Because extraction is identity, it immediately yields the edge coloring (1, 1, 0, 0, 0, 1), and evaluating that coloring on the source returns `true` ✓. Every triangle therefore uses both colors.

Multiplicity: The fixture stores one canonical edge coloring. Any binary ILP solution satisfying all triangle pairs is a valid Monochromatic Triangle witness.

Rule 3.78: (Set Splitting $\rightarrow$ Betweenness) This $O\left(n + \sum_{S \in \mathcal{C}} |S|\right)$ reduction [5, MS1] first normalizes each large subset to size 2 or 3 with complementarity pairs, then builds a Betweenness instance with one pole element p, one element a_u per normalized universe element, and one auxiliary betweenness element for each size-3 subset. If the normalized Set Splitting instance has universe size n' with s_2 size-2 subsets and s_3 size-3 subsets, the target has $n' + 1 + s_3$ elements and $s_2 + 2s_3$ triples.

Overhead: `num_elements = normalized_universe_size + 1 + normalized_num_size3_subsets`, `num_triples = normalized_num_size2_subsets + 2 * normalized_num_size3_subsets`.

Proof: *Construction.* Given Set Splitting instance $(U, \mathcal{C})$, first normalize every subset to size 2 or 3. For a subset $S = \{s_1, ..., s_k\}$ with $k \geq 4$, introduce fresh elements y^+, y^-, replace S by the size-3 subset $\{s_1, s_2, y^+\}$ and the complementarity subset $\{y^+, y^-\}$, and continue recursively on $\{y^-, s_3, ..., s_k\}$. Repeating this step yields an equivalent normalized instance $(U', \mathcal{C}')$ in which every subset has size 2 or 3.

Create one Betweenness element a_u for each $u \in U'$ and one distinguished pole p. For every size-2 subset $\{u, v\} \in \mathcal{C}'$, add triple (a_u, p, a_v). For every size-3 subset $\{u, v, w\} \in \mathcal{C}'$, introduce a fresh auxiliary element $d_{u,v,w}$ and add triples $(a_u, d_{u,v,w}, a_v)$ and $(d_{u,v,w}, p, a_w)$.

Correctness. The normalization identity preserves splittability: a coloring splits $\{s_1, ..., s_k\}$ if and only if it can be extended to fresh elements y^+, y^- so that $\{s_1, s_2, y^+\}$, $\{y^+, y^-\}$, and $\{y^-, s_3, ..., s_k\}$ are all non-monochromatic. Thus it suffices to reason about normalized subsets.

($\Rightarrow$) Let $\chi : U' \to \{0, 1\}$ split every subset of $\mathcal{C}'$. Place all a_u with $\chi(u) = 0$ to the left of p and all a_u with $\chi(u) = 1$ to the right. For a size-2 subset $\{u, v\}$, non-monochromaticity gives $\chi(u) \neq \chi(v)$, so p lies between a_u and a_v, satisfying (a_u, p, a_v). For a size-3 subset $\{u, v, w\}$, not all three colors are equal. If u and v lie on the same side of p, then w lies on the opposite side; place $d_{u,v,w}$ between a_u and a_v on their shared side. If u and v lie on opposite sides of p, place $d_{u,v,w}$ between them on the side opposite a_w. In both cases $(a_u, d_{u,v,w}, a_v)$ and $(d_{u,v,w}, p, a_w)$ hold.

($\Leftarrow$) Suppose an ordering satisfies all Betweenness triples. Define $\chi(u) = 0$ iff a_u lies left of p, and $\chi(u) = 1$ otherwise. For a size-2 subset $\{u, v\}$, the triple (a_u, p, a_v) forces a_u and a_v onto opposite sides of p, so the subset is non-monochromatic. For a size-3 subset $\{u, v, w\}$, the triple $(a_u, d_{u,v,w}, a_v)$ puts $d_{u,v,w}$ between a_u and a_v. If a_u, a_v, a_w were all on the same side of p, then $d_{u,v,w}$ would also lie on that side, making $(d_{u,v,w}, p, a_w)$ impossible because p cannot lie between two elements on the same side. Hence every size-3 subset is non-monochromatic. By normalization equivalence, the original Set Splitting instance is splittable.

Solution extraction. For each original universe element $i \in \{0, ..., |U| - 1\}$, return color 0 when $f(a_i) < f(p)$ and color 1 otherwise. □

Example: $|U| = 5$, 4 subsets, no normalization auxiliaries needed

Source: SetSplitting **Target:** Betweenness

```
$ pred create --example SetSplitting -o set-splitting.json
$ pred reduce set-splitting.json --to Betweenness -o bundle.json
$ pred solve bundle.json
$ pred evaluate set-splitting.json --config 1,0,1,0,0
```

Step 1 – Source instance. The canonical Set Splitting fixture has universe $U = \{0, 1, 2, 3, 4\}$ and subsets $S_1 = \{0, 1, 2\}$, $S_2 = \{2, 3, 4\}$, $S_3 = \{0, 3, 4\}$, and $S_4 = \{1, 2, 3\}$. The stored splitting is (1, 0, 1, 0, 0), so colors 0 and 1 both appear in every subset.

Step 2 – Add the pole and clause auxiliaries. Because every subset already has size 3, normalization adds no universe elements. The target therefore uses pole $p = a_5$ together with one auxiliary element for each subset, namely $d_1 = 6$, $d_2 = 7$, $d_3 = 8$, and $d_4 = 9$, for a total of 10 elements.

Step 3 – Form the betweenness triples. The four subsets become the triple pairs (0, 6, 1), (6, 5, 2); (2, 7, 3), (7, 5, 4); (0, 8, 3), (8, 5, 4); and (1, 9, 2), (9, 5, 3). Each pair uses one auxiliary d_j to force the corresponding 3-set to place at least one element on each side of the pole.

Step 4 – Verify the ordering and extraction. The stored Betweenness witness is $f =$ (8, 2, 9, 0, 1, 4, 3, 6, 7, 5), so the pole $p = 5$ sits at position $f(p) = 4$. Original elements $1, 3, 4$ lie to the left of the pole, while $0, 2$ lie to the right, so extraction returns (1, 0, 1, 0, 0) exactly. For example,

$(0, 6, 1)$ holds because $f(1) = 2 < f(6) = 3 < f(0) = 8$, and $(6, 5, 2)$ holds because $f(6) = 3 < f(5) = 4 < f(2) = 9$ ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.79: (*k*-Clique → Integer Linear Programming) A k-clique requires at least k selected vertices with no non-edge between any pair.
Overhead: `num_vars = num_vertices`, `num_constraints = num_vertices^2`.

Proof: *Construction.* Variables: $x_v \in \{0, 1\}$ for each $v \in V$. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_v x_v \geq k \\ & x_u + x_v \leq 1 \quad \forall (u, v) \notin E \\ & x_v \in \{0, 1\} \quad \forall v \in V \end{aligned}$$

.

Correctness. ($\Rightarrow$) A k-clique selects $\geq k$ mutually adjacent vertices, satisfying all constraints. ($\Leftarrow$) Any feasible solution selects $\geq k$ vertices with no non-edge pair, forming a clique of size $\geq k$.

Solution extraction. $K = \{v : x_v = 1\}$. □

Rule 3.80: (*k*-Clique → Balanced Complete Bipartite Subgraph) This $O(n^2 + nm)$ reduction (Johnson, 1987; Garey and Johnson GT24) constructs a bipartite graph $H = (A \cup B, F)$ with $|A| = n + \binom{k}{2}$ and $|B| = m + n - k$ using a non-incidence encoding. The target biclique size is $K' = n + \binom{k}{2} - k$.
Overhead: `left_size = num_vertices + k * (k + -1 * 1) * 2^-1`, `right_size = num_edges + num_vertices + -1 * k`, `k = num_vertices + k * (k + -1 * 1) * 2^-1 + -1 * k`.

Proof: *Construction.* Given k-Clique instance $(G = (V, E), k)$ with $n = |V|$, $m = |E|$: Let $C = \binom{k}{2} = \frac{k(k-1)}{2}$. Add C isolated vertices to V, giving $V' = \{v_0, ..., v_{n'-1}\}$ with $n' = n + C$. Part $A = V'$. Part B has m *edge elements* $\{e_0, ..., e_{m-1}\}$ (one per edge of G) and $n - k$ *padding elements* $\{w_0, ..., w_{n-k-1}\}$. Add bipartite edge (v, e_j) iff v is NOT an endpoint of e_j (non-incidence). Add (v, w_i) for all $v \in A$, $w_i \in W$ (full padding). Set $K' = n' - k$.

Correctness. ($\Rightarrow$) If $S \subseteq V$ is a k-clique, let $A' = V' \setminus S$ ($|A'| = n' - k = K'$) and $B' = E(S) \cup W$ where $E(S)$ is the set of intra-clique edges. Since $|E(S)| = C$ and $|W| = n - k$, we have $|B'| = K'$. For any $v \in A'$ and $e_j \in E(S)$: both endpoints of e_j lie in S but $v \notin S$, so v is not an endpoint — the non-incidence edge exists. For padding elements, all edges exist by construction. ($\Leftarrow$) If (A', B') is a balanced K'-biclique, let $S = \{v \in V : v \notin A'\}$ with $|S| = k$. Any edge e_j with an endpoint $u \in A'$ cannot be in B' (since $(u, e_j) \notin F$). So $B' \cap E \subseteq E(S)$. Since $|B'| = K'$ and $|W| = n - k$, we need $|B' \cap E| \geq K' - |W| = C$. But $|E(S)| \leq \binom{k}{2} = C$, so $|E(S)| = C$, meaning S is a k-clique.

Solution extraction. For each original vertex $v \in \{0, ..., n-1\}$: source$[v] = 1 -$ target$[v]$ (vertices NOT selected on the left side form the clique). □

Example: 4-vertex graph with $k = 3$: non-incidence gadget construction

Source: KClique **Target:** BalancedCompleteBipartiteSubgraph

```
$ pred create --example KClique/SimpleGraph -o kclique.json
$ pred reduce kclique.json --to BalancedCompleteBipartiteSubgraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate kclique.json --config 1,1,1,0
```

Step 1 – Pad the vertex set. $C(3,2) = 3$ padding vertices are added, giving $n' = 4 + 3 = 7$ left vertices (Part A).

Step 2 – Build Part B. Part B has 4 edge elements (one per original edge) plus $4 - 3 = 1$ padding elements, for $|B| = 5$.

Step 3 – Bipartite edges. For each $v \in A$ and edge element $e_j = \{u, w\}$, add (v, e_j) iff $v \notin \{u, w\}$ (non-incidence). All padding elements connect to all left vertices.

Step 4 – Set target parameter. $K' = n' - k = 7 - 3 = 4$.

Step 5 – Verify a solution. The 3-clique is $S = \{0, 1, 2\}$. The 4 left vertices NOT in S plus the 3 padding vertices form the left side A'. The right side B' contains the 3 intra-clique edge elements plus 1 padding elements ($|B'| = 4$). All 4×4 cross-edges are present because no $v \in A'$ is an endpoint of any selected edge element.

Multiplicity: The fixture stores one canonical witness.

Rule 3.81: (Maximal Independent Set (weighted) → Integer Linear Programming) An independent set that is also maximal: no vertex outside the set can be added without violating independence.
Overhead: `num_vars = num_vertices`, `num_constraints = num_edges + num_vertices`.

Proof: *Construction.* Variables: $x_v \in \{0, 1\}$ for each $v \in V$. The ILP is:

$$\begin{aligned} \max \quad & \sum_v w_v x_v \\ \text{subject to} \quad & x_u + x_v \leq 1 \quad \forall (u, v) \in E \\ & x_v + \sum_{u \in N(v)} x_u \geq 1 \quad \forall v \in V \\ & x_v \in \{0, 1\} \quad \forall v \in V \end{aligned}$$

.

Correctness. Independence constraints prevent adjacent selections; maximality constraints ensure every vertex is either selected or has a selected neighbor.

Solution extraction. $I = \{v : x_v = 1\}$. □

Rule 3.82: (Minimum Maximal Matching → Integer Linear Programming) Each edge is either selected or not; matching and maximality constraints are both directly linear in binary edge indicators.
Overhead: `num_vars = num_edges`, `num_constraints = num_vertices + num_edges`.

Proof: *Construction.* Variables: $x_e \in \{0, 1\}$ for each $e \in E$. The ILP is:

$$\begin{aligned} \min \quad & \sum_e x_e \\ \text{subject to} \quad & \sum_{e \ni v} x_e \leq 1 \quad \forall v \in V \\ & x_j + \sum_{\substack{i : i \sim j, \\ i \neq j}} x_i \geq 1 \quad \forall j \in E \\ & x_e \in \{0, 1\} \quad \forall e \in E \end{aligned}$$

, where $i \sim j$ denotes that edges i and j share an endpoint.

Correctness. Degree constraints enforce the matching property. For each edge j, the maximality constraint requires that j itself or at least one adjacent edge is selected, ensuring the matching cannot be extended. ($\Rightarrow$) A minimum maximal matching satisfies both constraints and minimizes cardinality. ($\Leftarrow$) Any feasible solution is a maximal matching; the objective minimizes its size.

Solution extraction. $M = \{e : x_e = 1\}$. □

Rule 3.83: (Minimum Covering by Cliques → Integer Linear Programming) Use one potential clique slot per source edge, with binary vertex-membership, slot-activation, and edge-covered-by-slot variables.
Overhead: `num_vars = num_vertices * num_edges + num_edges + num_edges * num_edges`, `num_constraints = num_vertices * num_edges + (num_vertices * (num_vertices + -1 * 1) * 2^-1 + -1 * num_edges) * num_edges + 3 * num_edges * num_edges + num_edges`.

Proof: *Construction.* Let $m = |E|$ and index the clique slots by $k \in \{0, ..., m-1\}$. Introduce binary variables $x_{v,k}$ for $v \in V$ and $k \in \{0, ..., m-1\}$, where $x_{v,k} = 1$ means vertex v is placed in clique slot k; binary activation variables z_k; and binary variables $y_{e,k}$ for $e = \{u, v\} \in E$, where $y_{e,k} = 1$ means edge e is covered by slot k. The ILP is:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{k=0}^{m-1} z_k \\
\text{subject to} \quad & x_{u,k} + x_{v,k} \leq 1 \quad \forall k, \\
& \forall \{u, v\} \notin E \\
& x_{v,k} \leq z_k \quad \forall v \in V, \\
& \forall k \\
& y_{\{u,v\},k} \leq x_{u,k} \quad \forall \{u, v\} \in E, \\
& \forall k \\
& y_{\{u,v\},k} \leq x_{v,k} \quad \forall \{u, v\} \in E, \\
& \forall k \\
& y_{\{u,v\},k} \geq x_{u,k} + x_{v,k} - 1 \quad \forall \{u, v\} \in E, \\
& \forall k \\
& \sum_{k=0}^{m-1} y_{e,k} \geq 1 \quad \forall e \in E \\
& x_{v,k}, z_k, y_{e,k} \in \{0, 1\}
\end{aligned}
$$

.

Correctness. (⇒) Given an edge-clique cover $C_0, ..., C_{t-1}$ with $t \leq m$, map clique C_k to slot k: set $z_k = 1$, set $x_{v,k} = 1$ exactly for $v \in C_k$, and set $y_{e,k} = 1$ exactly for the edges e whose endpoints both lie in C_k. Because each C_k is a clique, no non-edge constraint is violated. Every covered edge satisfies at least one coverage inequality, so the ILP objective is at most t.

(⇐) Conversely, let (x, z, y) be any feasible ILP solution. For each slot k, the vertices with $x_{v,k} = 1$ form a clique because every non-edge pair is forbidden from appearing together in that slot. If $y_{\{u,v\},k} = 1$, the McCormick constraints force both endpoints u and v into slot k, so the edge is indeed contained in that clique. The coverage inequalities therefore certify that every source edge lies in at least one clique slot, giving a valid edge-clique cover. Since the objective counts active slots, minimizing it yields a minimum cover.

Solution extraction. For each source edge e, choose any slot k with $y_{e,k} = 1$ and output the label k. The extracted edge-to-slot labeling is valid because every slot induces a clique and every edge is assigned to at least one covering slot. □

Rule 3.84: (Partially Ordered Knapsack → Integer Linear Programming) Standard knapsack with precedence constraints: item b can only be selected if item a is also selected for each precedence (a, b).
Overhead: `num_vars = num_items`, `num_constraints = num_precedences + 1`.

Proof: *Construction.* Variables: $x_i \in \{0, 1\}$ per item. The ILP is:

$$\begin{aligned} \max \quad & \sum_i v_i x_i \\ \text{subject to} \quad & \sum_i w_i x_i \leq C \\ & x_b \leq x_a \quad \text{for each precedence } (a, b) \\ & x_i \in \{0, 1\} \quad \forall i \end{aligned}$$

.

Correctness. Capacity and precedence constraints are directly linear. Any feasible ILP solution is a valid knapsack packing respecting the partial order.

Solution extraction. Selected items: $\{i : x_i = 1\}$. □

Rule 3.85: (Rectilinear Picture Compression → Integer Linear Programming) Cover all 1-cells with at most B maximal all-1 rectangles.
Overhead: `num_vars = num_rows * num_cols`, `num_constraints = num_rows * num_cols + 1`.

Proof: *Construction.* Variables: $x_r \in \{0, 1\}$ per maximal rectangle r. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{r \text{ covers } (i,j)} x_r \geq 1 \quad \forall (i, j) \text{ with source cell} = 1 \\ & \sum_r x_r \leq B \\ & x_r \in \{0, 1\} \quad \forall r \end{aligned}$$

.

Correctness. Coverage constraints ensure every 1-cell is covered; the cardinality bound limits the number of rectangles.

Solution extraction. Selected rectangles: $\{r : x_r = 1\}$. □

Rule 3.86: (Shortest Weight-Constrained Path (weighted) → Integer Linear Programming) Find a minimum-length s-t path subject to a weight budget, using directed arc variables with MTZ ordering $o_v - o_u \geq 1 - M(1 - a_{u,v})$ on selected arcs to prevent subtours.
Overhead: `num_vars = 2 * num_edges + num_vertices`, `num_constraints = 5 * num_edges + 4 * num_vertices + 2`.

Proof: *Construction.* Let A contain both orientations of every undirected edge and let $M = n$. Variables: binary $a_{u,v} \in \{0, 1\}$ for each directed arc $(u, v) \in A$, plus integer $o_v \in \{0, ..., n - 1\}$ per vertex. The ILP is:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{(u,v)\in A} l_{u,v} a_{u,v} \\
\text{subject to} \quad & \sum_{w:(v,w)\in A} a_{v,w} - \sum_{u:(u,v)\in A} a_{u,v} = b_v \quad \forall v \in V \\
& \sum_{w:(v,w)\in A} a_{v,w} \leq 1 \quad \forall v \in V \\
& \sum_{u:(u,v)\in A} a_{u,v} \leq 1 \quad \forall v \in V \\
& a_{u,v} + a_{v,u} \leq 1 \quad \forall \{u,v\} \in E \\
& o_v - o_u \geq 1 - M\big(1 - a_{u,v}\big) \quad \forall (u,v) \in A \\
& \sum_{(u,v)\in A} w_{u,v} a_{u,v} \leq W \\
& a_{u,v} \in \{0,1\}, o_v \in \{0,\ldots,n-1\}
\end{aligned}
$$

, where $b_s = 1$, $b_t = -1$, and $b_v = 0$ otherwise.

Correctness. Flow balance forces an s-t path; the MTZ inequalities apply only on selected arcs and therefore eliminate subtours; the weight constraint enforces the budget; the objective minimizes total path length.

Solution extraction. Edge $\{u, v\}$ is selected iff $a_{u,v} + a_{v,u} > 0$. □

Rule 3.87: (Multiple Copy File Allocation → Integer Linear Programming) Place file copies at vertices to minimize total storage plus weighted access cost.
Overhead: `num_vars = num_vertices + num_vertices^2`, `num_constraints = num_vertices^2 + num_vertices`.

Proof: *Construction.* Variables: binary x_v (copy at v) and $y_{v,u}$ (vertex v served by copy at u). The ILP is:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{v} s_v x_v + \sum_{v,u} \text{usage}_v d(v,u) y_{v,u} \\
\text{subject to} \quad & \sum_{u} y_{v,u} = 1 \quad \forall v \\
& y_{v,u} \leq x_u \quad \forall v, u \\
& x_v, y_{v,u} \in \{0,1\}
\end{aligned}
$$

.

Correctness. Assignment constraints ensure each vertex is served by exactly one copy; capacity links prevent assignment to non-copy vertices; the objective linearizes the total cost.

Solution extraction. Copy placement: $\{v : x_v = 1\}$. □

Rule 3.88: (Minimum Sum Multicenter (weighted) → Integer Linear Programming) Select k centers and assign each vertex to a center, minimizing the total weighted distance.
Overhead: `num_vars = num_vertices + num_vertices^2`, `num_constraints = num_vertices^2 + 2 * num_vertices + 1`.

Proof: *Construction.* Variables: binary x_j (vertex j is center), $y_{i,j}$ (vertex i assigned to center j). The ILP is:

$$\begin{aligned} \min \quad & \sum_{i,j} w_i d(i,j) y_{i,j} \\ \text{subject to} \quad & \sum_j x_j = k \\ & y_{i,j} \leq x_j \quad \forall i, j \\ & \sum_j y_{i,j} = 1 \quad \forall i \\ & x_j, y_{i,j} \in \{0,1\} \end{aligned}$$

.

Correctness. The assignment structure and cardinality constraint directly encode the k-median objective with precomputed shortest-path distances.

Solution extraction. Centers: $\{j : x_j = 1\}$. □

Rule 3.89: (Min-Max Multicenter (weighted) → Integer Linear Programming) Select k centers minimizing the maximum weighted distance from any vertex to its assigned center.
Overhead: `num_vars = num_vertices + num_vertices^2 + 1`, `num_constraints = 2 * num_vertices^2 + 3 * num_vertices + 2`.

Proof: *Construction.* Same assignment structure as MinimumSumMulticenter (binary x_j, $y_{i,j}$), plus an integer variable z. The ILP is:

$$\begin{aligned} \text{minimize} \quad & z \\ \text{subject to} \quad & \sum_j x_j = k \\ & y_{i,j} \leq x_j \quad \forall i, j \\ & \sum_j y_{i,j} = 1 \quad \forall i \\ & \sum_j w_i d(i,j) y_{i,j} \leq z \quad \forall i \\ & x_j, y_{i,j} \in \{0,1\}, z \in \mathbb{Z} \end{aligned}$$

.

Correctness. Each minimax constraint forces z to be at least the weighted distance from vertex i to its assigned center. Minimizing z yields the optimal maximum weighted distance.

Solution extraction. Centers: $\{j : x_j = 1\}$. □

Rule 3.90: (Multiprocessor Scheduling → Integer Linear Programming) Assign tasks to processors so that no processor's total load exceeds the deadline.
Overhead: `num_vars = num_tasks * num_processors`, `num_constraints = num_tasks + num_processors`.

Proof: *Construction.* Variables: binary $x_{j,p}$ (task j on processor p), one-hot per task. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_p x_{j,p} = 1 \quad \forall j \\ & \sum_j l_j x_{j,p} \leq D \quad \forall p \\ & x_{j,p} \in \{0,1\} \end{aligned}$$

.

Correctness. One-hot constraints ensure each task is assigned to exactly one processor; load constraints enforce the deadline on every processor.

Solution extraction. Task j goes to processor $\arg\max_p x_{j,p}$. □

Rule 3.91: (Capacity Assignment → Integer Linear Programming) Assign a capacity level to each link to minimize total cost subject to a delay budget.
Overhead: `num_vars = num_links * num_capacities`, `num_constraints = num_links + 1`.

Proof: *Construction.* Variables: binary $x_{l,c}$ (link l gets capacity c), one-hot per link. The ILP is:

$$\begin{aligned} \text{minimize} \quad & \sum_{l,c} \text{cost}[l][c] x_{l,c} \\ \text{subject to} \quad & \sum_{c} x_{l,c} = 1 \quad \forall l \\ & \sum_{l,c} \text{delay}[l][c] x_{l,c} \leq J \\ & x_{l,c} \in \{0,1\} \end{aligned}$$

.

Correctness. One-hot constraints fix one capacity per link; the delay budget constraint is linear in the indicators; the objective sums the selected costs.

Solution extraction. Link l gets capacity $\arg\max_c x_{l,c}$. □

Rule 3.92: (Expected Retrieval Cost → Integer Linear Programming) Assign records to sectors to minimize expected retrieval cost, using product linearization for the quadratic cost terms.
Overhead: `num_vars = num_records * num_sectors + num_records^2 * num_sectors^2`, `num_constraints = num_records + 3 * num_records^2 * num_sectors^2`.

Proof: *Construction.* Variables: binary $x_{r,s}$ (record r in sector s), one-hot per record, plus linearization variables $z_{(r,s),(r',s')} = x_{r,s} \cdot x_{r',s'}$. The ILP is:

$$\begin{aligned} \text{minimize} \quad & \sum d(s,s') p_r p_{r'} z_{(r,s),(r',s')} \\ \text{subject to} \quad & \sum_{s} x_{r,s} = 1 \quad \forall r \\ & z_{(r,s),(r',s')} \leq x_{r,s} \quad \forall r, s, r', s' \\ & z_{(r,s),(r',s')} \leq x_{r',s'} \quad \forall r, s, r', s' \\ & z_{(r,s),(r',s')} \geq x_{r,s} + x_{r',s'} - 1 \quad \forall r, s, r', s' \\ & x_{r,s}, z_{(r,s),(r',s')} \in \{0,1\} \end{aligned}$$

.

Correctness. McCormick constraints force z to equal the product of binary indicators, linearizing the quadratic cost. The ILP objective directly encodes the expected retrieval cost.

Solution extraction. Record r goes to sector $\arg\max_s x_{r,s}$. □

Rule 3.93: (Partition Into Triangles → Integer Linear Programming) Partition vertices into groups of 3 such that each group forms a triangle in the graph.
Overhead: `num_vars = num_vertices^2`, `num_constraints = num_vertices^2 * num_vertices`.

Proof: *Construction.* Variables: binary $x_{v,g}$ (vertex v in group g), one-hot per vertex, $q = n/3$ groups. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{g} x_{v,g} = 1 \quad \forall v \\ & \sum_{v} x_{v,g} = 3 \quad \forall g \in \{1, ..., q\} \\ & x_{u,g} + x_{v,g} \leq 1 \quad \forall g \in \{1, ..., q\}, (u, v) \notin E \\ & x_{v,g} \in \{0, 1\} \end{aligned}$$

.

Correctness. Size-3 groups with no non-edge pair within any group forces each group to be a triangle.

Solution extraction. Vertex v goes to group $\arg\max_g x_{v,g}$. □

Rule 3.94: (Partition into Paths of Length 2 → Integer Linear Programming) Partition vertices into groups of 3 such that each group induces a path of length 2 (at least 2 edges within the group).
Overhead: `num_vars = num_vertices^2 + num_edges * num_vertices`, `num_constraints = num_vertices^2 + num_edges * num_vertices + num_vertices`.

Proof: *Construction.* Variables: binary $x_{v,g}$ plus product linearization variables $z_{(u,v),g} = x_{u,g} \cdot x_{v,g}$ for edges (u, v). The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{g} x_{v,g} = 1 \quad \forall v \\ & \sum_{v} x_{v,g} = 3 \quad \forall g \\ & \sum_{(u,v) \in E} z_{(u,v),g} \geq 2 \quad \forall g \\ & z_{(u,v),g} \leq x_{u,g} \quad \forall (u, v) \in E, g \\ & z_{(u,v),g} \leq x_{v,g} \quad \forall (u, v) \in E, g \\ & z_{(u,v),g} \geq x_{u,g} + x_{v,g} - 1 \quad \forall (u, v) \in E, g \\ & x_{v,g}, z_{(u,v),g} \in \{0, 1\} \end{aligned}$$

.

Correctness. The edge count constraint ensures connectivity within each group. Combined with group size 3, this forces a path of length 2.

Solution extraction. Vertex v goes to group $\arg\max_g x_{v,g}$. □

Rule 3.95: (Partition into Paths of Length 2 → Bounded Component Spanning Forest (weighted)) This $O(n + m)$ parameter-setting reduction (Hadlock, 1974; Garey and Johnson [5, ND10, p. 208]) constructs a Bounded Component Spanning Forest instance on the same graph with unit vertex weights, $K = |V| \ /3$ components, and weight bound $B = 3$.
Overhead: `num_vertices = num_vertices`, `num_edges = num_edges`, `max_components = num_vertices * 3^-1`.

Proof: *Construction.* Given a Partition into Paths of Length 2 instance on graph $G = (V, E)$ with $|V| = 3q$:

- Graph: use G unchanged.
- Vertex weights: $w(v) = 1$ for all $v \in V$.
- Component bound: $K = q = |V| \ /3$.
- Weight bound: $B = 3$.

Correctness. ($\Rightarrow$) Suppose V has a valid P_3-partition $V_1, ..., V_q$ where each V_t induces at least 2 edges. Since a graph on 3 vertices with at least 2 edges is connected, each V_t is a connected component of weight

$1+1+1=3 \leq B$. There are $q=K$ components. ($\Leftarrow$) Suppose V has a partition into at most $K=q$ connected components each of weight at most $B=3$. Since all weights are 1, each component has at most 3 vertices. With $3q$ vertices and at most q components, the pigeonhole principle forces exactly q components of exactly 3 vertices each. A connected graph on 3 vertices has at least 2 edges (a path P_3 or a triangle K_3), satisfying the P_3-partition requirement.

Solution extraction. Identity: the component assignment in BCSF is directly a group assignment in PPL2.□

Example: 6-vertex graph ($n=6$, $q=2$): two P_3 paths

Source: PartitionIntoPathsOfLength2 **Target:** BoundedComponentSpanningForest

```
$ pred create --example PartitionIntoPathsOfLength2 -o ppl2.json
$ pred reduce ppl2.json --to BoundedComponentSpanningForest/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate ppl2.json --config 0,0,0,1,1,1
```

Source PPL2: groups = {0, 0, 0, 1, 1, 1} on a graph with $n=6$ vertices and $|E|=4$ edges
Target BCSF: components = {0, 0, 0, 1, 1, 1}, $K=2$, $B=3$
Identity mapping: source and target configs coincide ✓

Rule 3.96: (Sum of Squares Partition → Integer Linear Programming) Partition elements into groups minimizing $\sum_g \left(\sum_{i \in g} s_i\right)^2$.
Overhead: `num_vars = num_elements * num_groups + num_elements^2 * num_groups`, `num_constraints = num_elements + 3 * num_elements^2 * num_groups`.

Proof: *Construction.* Variables: binary $x_{i,g}$ (element i in group g), plus $z_{(i,j),g} = x_{i,g} \cdot x_{j,g}$. The ILP is:

$$\begin{aligned}
\text{minimize} \quad & \sum_g \sum_{i,j} s_i s_j z_{(i,j),g} \\
\text{subject to} \quad & \sum_g x_{i,g} = 1 \quad \forall i \\
& z_{(i,j),g} \leq x_{i,g} \quad \forall i,j,g \\
& z_{(i,j),g} \leq x_{j,g} \quad \forall i,j,g \\
& z_{(i,j),g} \geq x_{i,g} + x_{j,g} - 1 \quad \forall i,j,g \\
& x_{i,g}, z_{(i,j),g} \in \{0,1\}
\end{aligned}$$

.

Correctness. Product linearization captures the quadratic sum-of-squares objective; the ILP minimizes the linearized form directly.

Solution extraction. Element i goes to group $\arg\max_g x_{i,g}$. □

Rule 3.97: (Precedence Constrained Scheduling → Integer Linear Programming) Assign unit-length tasks to time slots on m processors, respecting precedence constraints and a deadline.
Overhead: `num_vars = num_tasks * deadline`, `num_constraints = num_tasks + deadline + num_tasks^2`.

Proof: *Construction.* Variables: binary $x_{j,t}$ (task j at time t), one-hot per task. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_t x_{j,t} = 1 \quad \forall j \\
& \sum_j x_{j,t} \leq m \quad \forall t \\
& \sum_t t x_{j,t} \geq \sum_t t x_{i,t} + 1 \quad \text{for each precedence } (i,j) \\
& x_{j,t} \in \{0,1\}
\end{aligned}
$$

.

Correctness. One-hot ensures each task is scheduled once; capacity limits processors per slot; precedence is linearized via weighted time indicators.

Solution extraction. Task j is scheduled at time $\arg\max_t x_{j,t}$. □

Rule 3.98: (Scheduling With Individual Deadlines → Integer Linear Programming) Schedule unit-length tasks on m processors, each task j must complete before its individual deadline d_j.
Overhead: `num_vars = num_tasks * max_deadline`, `num_constraints = num_tasks + max_deadline + num_precedences`.

Proof: *Construction.* Variables: binary $x_{j,t}$ (task j at time $t \in \{0, ..., d_j - 1\}$), one-hot per task. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{t=0}^{d_j-1} x_{j,t} = 1 \quad \forall j \\
& \sum_j x_{j,t} \leq m \quad \forall t \\
& \sum_t t x_{j,t} \geq \sum_t t x_{i,t} + 1 \quad \text{for each precedence } (i,j) \\
& x_{j,t} \in \{0,1\}
\end{aligned}
$$

.

Correctness. Per-task deadline is enforced by restricting the time domain of each task's indicator variables.

Solution extraction. Task j is scheduled at time $\arg\max_t x_{j,t}$. □

Rule 3.99: (Preemptive Scheduling → Integer Linear Programming) Minimize makespan for preemptive parallel scheduling with variable-length tasks and precedence constraints.
Overhead: `num_vars = num_tasks * d_max + 1`, `num_constraints = num_tasks + d_max + num_precedences * d_max + 2 * num_tasks * d_max`.

Proof: *Construction.* Let $D = \sum_t \ell(t)$ be the horizon. Variables: binary $x_{t,u} \in \{0,1\}$ (task t processed at slot u) for $t \in \{0, ..., n-1\}$, $u \in \{0, ..., D-1\}$; integer $M \in \{0, ..., D\}$ (makespan). The ILP is:

$$
\begin{aligned}
\min \quad & M \\
\text{subject to} \quad & \sum_u x_{t,u} = \ell(t) \quad \forall t \quad \text{(work)} \\
& \sum_t x_{t,u} \leq m \quad \forall u \quad \text{(capacity)} \\
& \sum_u u \cdot x_{j,u} - \sum_u u \cdot x_{i,u} \geq 1 \quad \text{for each } (i \prec j) \quad \text{(precedence)} \\
& M - (u+1) \cdot x_{t,u} \geq 0 \quad \forall t, u \quad \text{(makespan)} \\
& x_{t,u} \in \{0,1\}, \quad M \in \mathbb{Z}_{\geq 0}
\end{aligned}
$$

.

Correctness. Work constraints enforce each task runs for exactly $\ell(t)$ slots. Capacity limits at most m tasks per slot. Precedences are enforced by weighted time indicators. Makespan lower bounds force $M \geq u + 1$ whenever task t is active at slot u.

Solution extraction. Config$[t \cdot D + u] = x_{t,u}$ for all t, u. □

Rule 3.100: (Sequencing Within Intervals → Integer Linear Programming) Schedule tasks with release times, deadlines, and processing lengths on a single machine without overlap.
Overhead: `num_vars = num_tasks^2`, `num_constraints = num_tasks^2 + num_tasks`.

Proof: *Construction.* Variables: binary $x_{j,t}$ (task j starts at time $t \in [r_j, d_j - l_j]$), one-hot per task. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{t=r_j}^{d_j - l_j} x_{j,t} = 1 \quad \forall j \\ & \sum_{j,t:t \leq \tau < t+l_j} x_{j,t} \leq 1 \quad \forall \tau \\ & x_{j,t} \in \{0,1\} \end{aligned}$$

.

Correctness. One-hot ensures each task starts once within its feasible window; non-overlap prevents simultaneous execution.

Solution extraction. Task j starts at time $\arg\max_t x_{j,t}$; config$[j] = t - r_j$. □

Rule 3.101: (Minimum Feedback Arc Set (weighted) → Integer Linear Programming) Remove minimum-weight arcs to make a directed graph acyclic, using MTZ-style ordering to enforce acyclicity among kept arcs.
Overhead: `num_vars = num_arcs + num_vertices`, `num_constraints = num_arcs + num_arcs + num_vertices`.

Proof: *Construction.* Variables: binary $y_a \in \{0,1\}$ per arc ($y_a = 1$ iff removed), integer $o_v \in \{0, ..., n-1\}$ per vertex. The ILP is:

$$\begin{aligned} \min \quad & \sum_a w_a y_a \\ \text{subject to} \quad & o_v - o_u \geq 1 - n y_a \quad \forall a = (u \to v) \\ & y_a \in \{0,1\} \quad \forall a \\ & o_v \in \{0, ..., n-1\} \quad \forall v \end{aligned}$$

.

Correctness. ($\Rightarrow$) Removing a FAS leaves a DAG with a topological ordering satisfying all constraints. ($\Leftarrow$) Among kept arcs, the ordering variables enforce acyclicity: a cycle would require $o_{v_1} < ... < o_{v_k} < o_{v_1}$, a contradiction.

Solution extraction. Removed arcs: $\{a : y_a = 1\}$. □

Rule 3.102: (Undirected Two-Commodity Integral Flow → Integer Linear Programming) Route two commodities on an undirected graph with shared edge capacities, using direction indicators to enforce anti-parallel flow constraints.
Overhead: `num_vars = 6 * num_edges`, `num_constraints = 7 * num_edges + 2 * num_nonterminal_vertices + 2`.

Proof: *Construction.* Variables: integer flow variables $f^k_{u,v}, f^k_{v,u}$ per edge per commodity ($k \in \{1,2\}$), plus binary direction indicators d^k_e. The ILP is:

$$\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & f^k_{u,v} \le \text{cap}_e d^k_e \quad \forall e = \{u,v\} \in E, k \in \{1,2\} \\
& f^k_{v,u} \le \text{cap}_e (1 - d^k_e) \quad \forall e = \{u,v\} \in E, k \in \{1,2\} \\
& \sum_{k=1}^{2} (f^k_{u,v} + f^k_{v,u}) \le \text{cap}_e \quad \forall e = \{u,v\} \in E \\
& \sum_w f^k_{v,w} - \sum_u f^k_{u,v} = b^k_v \quad \forall k \in \{1,2\}, v \in V \\
& d^k_e \in \{0,1\}, f^k_{u,v} \in \mathbb{Z}_{\ge 0}
\end{aligned}$$

.

Correctness. Direction indicators linearize the capacity-sharing constraint; flow conservation and demand constraints ensure valid multi-commodity flow.

Solution extraction. Flow variables (first $4|E|$ variables). □

Rule 3.103: (Directed Two-Commodity Integral Flow → Integer Linear Programming) Route two commodities on a directed graph with shared arc capacities.
Overhead: `num_vars = 2 * num_arcs`, `num_constraints = num_arcs + 2 * num_vertices + 2`.

Proof: *Construction.* Variables: integer $f^1_a, f^2_a \ge 0$ per arc a. The ILP is:

$$\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & f^1_a + f^2_a \le \text{cap}(a) \quad \forall a \in A \\
& \sum_{a \in \delta^+(v)} f^k_a - \sum_{a \in \delta^-(v)} f^k_a = b^k_v \quad \forall k \in \{1,2\}, v \in V \\
& \sum_{a \in \delta^-(t_k)} f^k_a - \sum_{a \in \delta^+(t_k)} f^k_a \ge R_k \quad \forall k \in \{1,2\} \\
& f^1_a, f^2_a \in \mathbb{Z}_{\ge 0} \quad \forall a \in A
\end{aligned}$$

.

Correctness. Joint capacity and conservation constraints directly encode the two-commodity flow problem.

Solution extraction. Direct: $2|A|$ flow variables. □

Rule 3.104: (Undirected Flow with Lower Bounds → Integer Linear Programming) Find a feasible single-commodity flow on an undirected graph with both upper and lower capacity bounds per edge.
Overhead: `num_vars = 3 * num_edges`, `num_constraints = 4 * num_edges + num_vertices + 1`.

Proof: *Construction.* Variables: integer $f_{u,v}, f_{v,u} \ge 0$ per edge, plus direction indicator $z_e \in \{0,1\}$. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & f_{u,v} \leq \text{cap}_e z_e \quad \forall e = \{u,v\} \in E \\
& f_{v,u} \leq \text{cap}_e (1 - z_e) \quad \forall e = \{u,v\} \in E \\
& f_{u,v} \geq \text{lower}_e z_e \quad \forall e = \{u,v\} \in E \\
& f_{v,u} \geq \text{lower}_e (1 - z_e) \quad \forall e = \{u,v\} \in E \\
& \sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a = b_v \quad \forall v \in V \\
& \sum_{a \in \delta^-(t)} f_a - \sum_{a \in \delta^+(t)} f_a \geq R \\
& z_e \in \{0,1\}, f_{u,v} \in \mathbb{Z}_{\geq 0}
\end{aligned}
$$

.

Correctness. Direction indicators force flow in one direction per edge; bounds enforce both upper and lower capacity limits.

Solution extraction. Edge orientations: z_e values. □

Rule 3.105: (Integral Flow with Homologous Arcs → Integer Linear Programming) Use one integer flow variable per arc, with standard conservation plus equality constraints on every homologous pair.
Overhead: `num_vars = num_arcs`, `num_constraints = num_arcs + num_vertices + -1 * 2 + 1`.

Proof: *Construction.* Variables: integer $f_a \geq 0$ per arc $a \in A$. The ILP is:

$$
\begin{aligned}
\text{find} \quad & (f_a)_{a \in A} \\
\text{subject to} \quad & f_a \leq c_a \quad \forall a \in A \\
& \sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a \quad \forall v \in V \setminus \{s,t\} \\
& f_a = f_b \quad \forall (a,b) \\
& \sum_{a \in \delta^-(t)} f_a - \sum_{a \in \delta^+(t)} f_a \geq R \\
& f_a \in \mathbb{Z}_{\geq 0} \quad \forall a \in A
\end{aligned}
$$

.

Correctness. (⇒) Any feasible integral flow already satisfies the capacity, conservation, equality, and sink-demand constraints. (⇐) Any feasible ILP assignment is exactly an integral arc-flow meeting the homologous-pair and requirement conditions.

Solution extraction. Output the arc-flow vector $(f_a)_{a \in A}$ in the source arc order. □

Rule 3.106: (Integral Flow With Multipliers → Integer Linear Programming) The source constraints are linear after writing one integer flow variable per arc and enforcing multiplier-scaled conservation at each non-terminal.
Overhead: `num_vars = num_arcs`, `num_constraints = num_arcs + num_vertices + -1 * 1`.

Proof: *Construction.* Variables: integer $f_a \geq 0$ per arc $a = (u \to v)$. The ILP is:

$$
\begin{aligned}
\text{find} \quad & (f_a)_{a \in A} \\
\text{subject to} \quad & f_a \le c_a \quad \forall a \in A \\
& \sum_{a \in \delta^+(v)} f_a = h(v) \sum_{a \in \delta^-(v)} f_a \quad \forall v \in V \setminus \{s,t\} \\
& \sum_{a \in \delta^-(t)} f_a - \sum_{a \in \delta^+(t)} f_a \ge R \\
& f_a \in \mathbb{Z}_{\ge 0} \quad \forall a \in A
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A valid multiplier flow satisfies these linear equalities and inequalities by definition. ($\Leftarrow$) Any feasible ILP solution gives an integral arc flow whose non-terminal outflow equals the prescribed multiple of its inflow and whose sink inflow meets the requirement.

Solution extraction. Output the arc-flow vector $(f_a)_{a \in A}$. □

Rule 3.107: (Path-Constrained Network Flow $\rightarrow$ Integer Linear Programming) Because flow may use only the prescribed s-t paths, it suffices to assign an integer amount to each allowed path and aggregate those loads on every arc.
Overhead: `num_vars = num_paths`, `num_constraints = num_arcs + 1`.

Proof: *Construction.* Let $P_1, ..., P_q$ be the prescribed paths. Variables: integer $f_i \ge 0$ for each path P_i. The ILP is:

$$
\begin{aligned}
\text{find} \quad & (f_i)_{i=1}^{q} \\
\text{subject to} \quad & \sum_{i : a \in P_i} f_i \le c_a \quad \forall a \in A \\
& \sum_{i} f_i \ge R \\
& f_i \in \mathbb{Z}_{\ge 0} \quad \forall i \in \{1, ..., q\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any valid path-flow assignment respects every arc capacity and delivers at least R units in total. ($\Leftarrow$) Any feasible ILP solution assigns integral flow only to the prescribed paths, and the aggregated arc loads satisfy the network capacities.

Solution extraction. Output the path-flow vector $(f_1, ..., f_q)$. □

Rule 3.108: (Disjoint Connecting Paths $\rightarrow$ Integer Linear Programming) Route one unit of flow for each terminal pair on an oriented copy of the graph, and forbid internal vertices from carrying more than one commodity.
Overhead: `num_vars = num_pairs * 2 * num_edges`, `num_constraints = num_pairs * num_vertices + num_pairs * num_edges + num_edges + num_vertices`.

Proof: *Construction.* For terminal pairs (s_k, t_k), variables: binary $f^k_{u,v}$ on each orientation of each edge and integer order variables h^k_v. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{w} f^k_{s_k,w} - \sum_{u} f^k_{u,s_k} = 1 \quad \forall k \\
& \sum_{u} f^k_{u,t_k} - \sum_{w} f^k_{t_k,w} = 1 \quad \forall k \\
& \sum_{w} f^k_{v,w} - \sum_{u} f^k_{u,v} = 0 \quad \forall k, v \in V \setminus \{s_k, t_k\} \\
& f^k_{u,v} + f^k_{v,u} \leq 1 \quad \forall \{u,v\} \in E, k \\
& \sum_{k} \sum_{w \in N(v)} f^k_{v,w} \leq 1 \quad \forall \text{ non-terminal } v \\
& h^k_v \geq h^k_u + 1 - M\big(1 - f^k_{u,v}\big) \quad \forall k, u \to v \\
& f^k_{u,v} \in \{0,1\}, h^k_v \in \mathbb{Z}_{\geq 0}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A family of pairwise internally vertex-disjoint connecting paths orients each path from its source to its sink and satisfies all constraints. ($\Leftarrow$) The conservation, disjointness, and ordering constraints force each commodity to trace one simple path, and different commodities can intersect only at terminals.

Solution extraction. Mark an edge selected in the source config iff some orientation of that edge carries flow for some commodity. □

Rule 3.109: (Length-Bounded Disjoint Paths → Integer Linear Programming) Use one unit-flow commodity for each requested path and add hop variables so every chosen path has at most the source bound K edges. *Overhead:* `num_vars = max_paths * 2 * num_edges + max_paths`, `num_constraints = max_paths * num_vertices + max_paths * num_edges + max_paths + num_edges + num_vertices + max_paths`.

Proof: *Construction.* Variables: binary $f^k_{u,v}$ on each orientation of each edge for each path slot k, plus integer hop variables $h^k_v \in \{0, ..., K\}$, where K is the path-length bound and $M = K + 1$. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{w} f^k_{s,w} - \sum_{u} f^k_{u,s} = 1 \quad \forall k \\
& \sum_{u} f^k_{u,t} - \sum_{w} f^k_{t,w} = 1 \quad \forall k \\
& \sum_{w} f^k_{v,w} - \sum_{u} f^k_{u,v} = 0 \quad \forall k, v \in V \setminus \{s,t\} \\
& f^k_{u,v} + f^k_{v,u} \leq 1 \quad \forall \{u,v\} \in E, k \\
& \sum_{k} \sum_{w \in N(v)} f^k_{v,w} \leq 1 \quad \forall v \in V \setminus \{s,t\} \\
& h^k_s = 0 \quad \forall k \\
& h^k_v \geq h^k_u + 1 - M\big(1 - f^k_{u,v}\big) \quad \forall k, u \to v \\
& h^k_t \leq K \quad \forall k \\
& f^k_{u,v} \in \{0,1\}, h^k_v \in \{0, ..., K\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A collection of J internally disjoint s-t paths of length at most K yields feasible commodity flows and consistent hop labels. ($\Leftarrow$) The flow and hop constraints force each commodity to be a simple s-t path, while the vertex-disjointness inequalities match the source requirement.

Solution extraction. For each path slot k, set the source vertex-indicator block to 1 exactly on the vertices incident to the commodity-k path, including s and t. □

Rule 3.110: (Mixed Chinese Postman (weighted) → Integer Linear Programming) Choose an orientation for every undirected edge, then add integer traversal variables on the available directed arcs to balance the oriented required multigraph within the length bound.

Overhead: `num_vars = num_edges + 4 * (num_arcs + 2 * num_edges) + 3 * num_vertices + 1`, `num_constraints = num_vertices + 2 * (num_arcs + 2 * num_edges) + 2 * (num_arcs + 2 * num_edges) + num_vertices + 1 + num_vertices + 4 * num_vertices + 2 * (num_arcs + 2 * num_edges) + 2 * num_vertices`.

Proof: *Construction.* Let $n = |V|$, let the original directed arcs be $A = \{a_0, ..., a_{m-1}\}$ with $a_i = (\alpha_i, \beta_i)$, and let the undirected edges be $E = \{e_0, ..., e_{q-1}\}$ with $e_k = \{u_k, v_k\}$. Set $R = m + q$. If $R = 0$, return the empty feasible ILP: the empty walk already has length 0. Otherwise form the available directed-arc list $A^* = \{b_0, ..., b_{L-1}\}$ with $L = m + 2q$, where $b_i = a_i$ for $0 \le i < m$, $b_{m+2k} = (u_k, v_k)$, and $b_{m+2k+1} = (v_k, u_k)$. Write $b_j = (\text{tail}_j, \text{head}_j)$ and let ℓ_j be the corresponding length. Use `ILP<i32>` with binary variables encoded by bounds $0 \le x \le 1$. Order the variables as $(d_0, ..., d_{q-1}, g_0, ..., g_{L-1}, y_0, ..., y_{L-1}, z_0, ..., z_{n-1}, \rho_0, ..., \rho_{n-1}, s, b_0, ..., b_{n-1}, f_0, ..., f_{L-1}, h_0, ..., h_{L-1})$, so d_k has index k, g_j has index $q + j$, y_j has index $q + L + j$, z_v has index $q + 2L + v$, ρ_v has index $q + 2L + n + v$, s has index $q + 2L + 2n$, b_v has index $q + 2L + 2n + 1 + v$, f_j has index $q + 2L + 3n + 1 + j$, and h_j has index $q + 3L + 3n + 1 + j$. There are $q + 4L + 3n + 1$ variables in total.

The orientation bit $d_k \in \{0, 1\}$ means $d_k = 0$ chooses $u_k \to v_k$ and $d_k = 1$ chooses $v_k \to u_k$. Define the required multiplicity on each available arc explicitly by $r_{i(d)} = 1$ for $0 \le i < m$, $r_{m+2k}(d) = 1 - d_k$, and $r_{m+2k+1}(d) = d_k$. Thus the two oriented copies of each undirected edge are already linear in the orientation bit.

Let $G = R(n-1)$ and $M_{\text{use}} = 1 + G$. The variable $g_j \in \{0, ..., G\}$ counts extra traversals of b_j beyond the required multiplicity, so the total multiplicity of b_j is $r_{j(d)} + g_j$. The bound $G = R(n-1)$ is exact for this formulation: any closed walk can be shortcut so that between consecutive required traversals it uses a simple connector path of at most $n - 1$ arcs, and there are exactly R such connector segments in the cyclic order of the required traversals.

The constraints are: $\sum_{j:\text{tail}_j=v} \left(r_{j(d)} + g_j\right) - \sum_{j:\text{head}_j=v} \left(r_{j(d)} + g_j\right) = 0$ for every $v \in V$; $r_{j(d)} + g_j \le M_{\text{use}} y_j$ and $y_j \le r_{j(d)} + g_j$ for every $j \in \{0, ..., L-1\}$, so $y_j = 1$ iff arc b_j is used at least once; $y_j \le z_{\text{tail}_j}$ and $y_j \le z_{\text{head}_j}$ for every j, and $z_v \le \sum_{j:\text{tail}_j=v \text{ or } \text{head}_j=v} y_j$ for every vertex v, so $z_v = 1$ iff v is incident to some used arc; $s = \sum_v z_v$; $\sum_v \rho_v = 1$, $\rho_v \le z_v$ for every v, and the product linearization $b_v \le s$, $b_v \le n\rho_v$, $b_v \ge s - n(1 - \rho_v)$, $b_v \ge 0$ for every v, so $b_v = s\rho_v$ and therefore the unique root chosen by ρ supplies $s - 1$ units of connectivity flow; $0 \le f_j \le (n-1)y_j$ and $0 \le h_j \le (n-1)y_j$ for every available arc b_j; here the exact big-M for arc activation is $n - 1$, because at most one unit is demanded by each non-root active vertex; $\sum_{j:\text{tail}_j=v} f_j - \sum_{j:\text{head}_j=v} f_j = b_v - z_v$ for every vertex v; and $\sum_{j:\text{head}_j=v} h_j - \sum_{j:\text{tail}_j=v} h_j = b_v - z_v$ for every vertex v. The f-flow makes every active vertex reachable from the chosen root, and the h-flow makes the root reachable from every active vertex on the same used support. Finally impose the length bound $\sum_{j=0}^{L-1} \ell_j \left(r_{j(d)} + g_j\right) \le B$.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{j:\mathrm{tail}_j=v} \left(r_{j(d)} + g_j\right) - \sum_{j:\mathrm{head}_j=v} \left(r_{j(d)} + g_j\right) = 0 \quad \forall v \in V \\
& r_{j(d)} + g_j \le M_{\mathrm{use}} y_j, y_j \le r_{j(d)} + g_j \quad \forall j \in \{0, ..., L-1\} \\
& y_j \le z_{\mathrm{tail}_j}, y_j \le z_{\mathrm{head}_j} \quad \forall j \\
& z_v \le \sum_{j:\mathrm{tail}_j=v \text{ or } \mathrm{head}_j=v} y_j \quad \forall v \in V \\
& s = \sum_v z_v; \sum_v \rho_v = 1; \rho_v \le z_v \quad \forall v \in V \\
& \text{the standard product linearization enforces } b_v = s\rho_v \quad \forall v \in V \\
& 0 \le f_j, h_j \le (n-1) y_j \quad \forall j \in \{0, ..., L-1\} \\
& \text{forward and reverse root-flow conservation hold on the used support} \\
& \sum_{j=0}^{L-1} \ell_j \left(r_{j(d)} + g_j\right) \le B \\
& d_k, y_j, z_v, \rho_v \in \{0,1\}; g_j \in \{0, ..., G\}; f_j, h_j \in \{0, ..., n-1\}; s, b_v \in \mathbb{Z}_{\ge 0}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) From any feasible mixed-postman tour, set d_k from the direction in which edge e_k is first required, let g_j be the number of extra copies of b_j beyond the required multiplicity, and let y_j mark the positive-support arcs. The tour itself visits exactly the active vertices, so some active vertex can be chosen as the root. Taking one outgoing spanning arborescence and one incoming spanning arborescence of the used Eulerian digraph gives feasible f- and h-flows. The walk length is exactly $\sum_j \ell_j \left(r_{j(d)} + g_j\right)$, hence the ILP is feasible. ($\Leftarrow$) A feasible ILP solution chooses one direction for every undirected edge, and the balance equations make the directed multigraph with multiplicities $r_{j(d)} + g_j$ Eulerian. The two root-flow systems imply that the positive-support digraph on the active vertices is strongly connected. Therefore the used multigraph admits an Euler tour, and its total length is exactly the bounded linear form above, so the source instance is a YES-instance.

Solution extraction. Return the orientation bits d_e in the source edge order. □

Rule 3.111: (Rural Postman (weighted) → Integer Linear Programming) Use one traversal-multiplicity variable per edge, together with activation and connectivity constraints, to encode an Eulerian connected subgraph covering all required edges.
Overhead: `num_vars = num_edges + num_vertices + num_edges + num_vertices + 2 * num_edges`, `num_constraints = 2 * num_edges + num_required_edges + num_vertices + 2 * num_edges + num_vertices + 2 * num_edges + num_vertices + num_edges + num_edges + num_vertices`.

Proof: *Construction.* If $E' = \emptyset$, the empty circuit already satisfies the source instance whenever $B \ge 0$, so use the empty ILP. Otherwise fix a root vertex r incident to some required edge and let $n = |V|$. Variables: integer $t_e \in \{0, 1, 2\}$ and parity variables q_v, binary edge-activation flags y_e, binary vertex-activity flags z_v, and nonnegative connectivity-flow variables $f_{u,v}$ on both orientations of every edge. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & y_e \le t_e \le 2y_e \quad \forall e \in E \\
& t_e \ge 1 \quad \forall e \in E' \\
& \sum_{e : v \in e} t_e = 2q_v \quad \forall v \\
& y_e \le z_u, y_e \le z_v \quad \forall e = \{u, v\} \in E \\
& z_v \le \sum_{e : v \in e} y_e \quad \forall v \in V \\
& f_{u,v} \le (n-1)y_e, f_{v,u} \le (n-1)y_e \quad \forall e = \{u, v\} \in E \\
& \sum_{w : \{r,w\} \in E} f_{r,w} - \sum_{u : \{u,r\} \in E} f_{u,r} = \sum_v z_v - 1 \\
& \sum_{u : \{u,v\} \in E} f_{u,v} - \sum_{w : \{v,w\} \in E} f_{v,w} = z_v \quad \forall v \in V \setminus \{r\} \\
& \sum_e \ell_e t_e \le B \\
& y_e, z_v \in \{0, 1\}, t_e \in \{0, 1, 2\}, q_v, f_{u,v} \in \mathbb{Z}_{\ge 0}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any feasible rural-postman circuit uses each edge at most twice, has even degrees, is connected on its positive-support edges, and satisfies the bound. ($\Leftarrow$) A feasible ILP solution defines a connected Eulerian multigraph containing every required edge, hence an Eulerian circuit of total length at most B.

Solution extraction. Output the traversal multiplicities $(t_e)_{e \in E}$. □

Rule 3.112: (Stacker Crane → Integer Linear Programming) Encode the required-arc order by a one-hot position assignment and charge the shortest connector distance between each consecutive pair of required arcs.
Overhead: `num_vars = num_arcs * num_arcs + num_arcs * num_arcs * num_arcs`, `num_constraints = num_arcs + num_arcs + 3 * num_arcs * num_arcs * num_arcs`.

Proof: *Construction.* Let the required arcs be $A = \{a_0, ..., a_{m-1}\}$ with $a_i = (\text{tail}_i, \text{head}_i)$. Build the mixed connector graph $H = (V, A \cup \{(u, v), (v, u) : \{u, v\} \in E\})$, where the original required arcs keep their given lengths and each undirected edge contributes both orientations with the same length. Because all lengths are nonnegative, compute the all-pairs connector distances $D[u, v] = \text{dist}_{H(u,v)}$ either by running Dijkstra from every source vertex or by Floyd–Warshall on the n-vertex graph H; this is exactly the graph queried by `mixed_graph_adjacency()` and `shortest_path_length()` in the model. If $D[u, v] = \infty$, the pair is impossible and will be forbidden explicitly.

Use `ILP<bool>`. The binary position variables are $x_{i,p}$ for $i, p \in \{0, ..., m-1\}$, with index $\text{idx}_{x(i,p)} = im + p$. The binary McCormick variables are $z_{i,j,p}$ for $i, j, p \in \{0, ..., m-1\}$, where position $p+1$ is interpreted cyclically as $(p+1) \bmod m$; their indices are $\text{idx}_{z(i,j,p)} = m^2 + pm^2 + im + j$. There are $m^2 + m^3$ binary variables.

The constraints are: $\sum_{p=0}^{m-1} x_{i,p} = 1$ for each required arc i; $\sum_{i=0}^{m-1} x_{i,p} = 1$ for each position p; $z_{i,j,p} \le x_{i,p}$, $z_{i,j,p} \le x_{j,(p+1) \bmod m}$, and $z_{i,j,p} \ge x_{i,p} + x_{j,(p+1) \bmod m} - 1$ for all i, j, p; if $D[\text{head}_i, \text{tail}_j] = \infty$, then either set $z_{i,j,p} = 0$ for all p or, equivalently, impose $x_{i,p} + x_{j,(p+1) \bmod m} \le 1$ for all p; and finally $\sum_{i=0}^{m-1} \ell_i + \sum_{p=0}^{m-1} \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} D[\text{head}_i, \text{tail}_j] z_{i,j,p} \le B$. The first term is the total length of the required traversals, and the second term charges exactly one connector distance for each consecutive pair in the cyclic order.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{p=0}^{m-1} x_{i,p} = 1 \quad \forall i \in \{0, ..., m-1\} \\
& \sum_{i=0}^{m-1} x_{i,p} = 1 \quad \forall p \in \{0, ..., m-1\} \\
& z_{i,j,p} \le x_{i,p}, z_{i,j,p} \le x_{j,(p+1) \bmod m} \quad \forall i, j, p \\
& z_{i,j,p} \ge x_{i,p} + x_{j,(p+1) \bmod m} - 1 \quad \forall i, j, p \\
& z_{i,j,p} = 0 \quad \text{whenever } D\big[\text{head}_i, \text{tail}_j\big] = \infty \\
& \sum_{i=0}^{m-1} \ell_i + \sum_{p=0}^{m-1} \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} D\big[\text{head}_i, \text{tail}_j\big] z_{i,j,p} \le B \\
& x_{i,p}, z_{i,j,p} \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any feasible Stacker Crane permutation determines a one-hot assignment and consecutive-pair indicators whose connector costs equal the route length. ($\Leftarrow$) Any feasible ILP solution yields a permutation of the required arcs, and the linearized connector term is exactly the sum of shortest paths between consecutive arcs.

Solution extraction. Decode the permutation by taking, for each position p, the unique arc a with $x_{a,p} = 1$. □

Rule 3.113: (Steiner Tree in Graphs (weighted) → Integer Linear Programming) Select edges and certify terminal connectivity by sending one unit of flow from a root terminal to every other terminal through the selected subgraph.
Overhead: `num_vars = num_edges + 2 * num_edges * (num_terminals + -1 * 1)`, `num_constraints = num_vertices * (num_terminals + -1 * 1) + 2 * num_edges * (num_terminals + -1 * 1)`.

Proof: *Construction.* Fix a root terminal $r \in R$. Variables: binary $y_{u,v}$ for each undirected edge $\{u, v\}$ and nonnegative flow variables $f^t_{u,v}$ on each directed edge orientation for every terminal $t \in R \setminus \{r\}$. The ILP is:

$$
\begin{aligned}
\min \quad & \sum_{\{u,v\} \in E} w_{u,v} y_{u,v} \\
\text{subject to} \quad & \sum_{u} f^t_{u,v} - \sum_{w} f^t_{v,w} = b_{t,v} \quad \forall t \in R \setminus \{r\}, v \in V \\
& f^t_{u,v} \le y_{u,v} \quad \forall \{u, v\} \in E, t \in R \setminus \{r\} \\
& f^t_{v,u} \le y_{u,v} \quad \forall \{u, v\} \in E, t \in R \setminus \{r\} \\
& y_{u,v} \in \{0, 1\}, f^t_{u,v} \in \mathbb{Z}_{\ge 0}
\end{aligned}
$$

, where $b_{t,v} = -1$ if $v = r$, $b_{t,v} = 1$ if $v = t$, and $b_{t,v} = 0$ otherwise.

Correctness. ($\Rightarrow$) A Steiner tree supports a unit flow from the root to every other terminal using exactly its selected edges, with the same total weight. ($\Leftarrow$) Any feasible ILP solution selects a connected subgraph spanning all terminals, and with nonnegative edge weights an optimum solution is a minimum-weight Steiner tree.

Solution extraction. Output the binary edge-selection vector $(y_e)_{e \in E}$. □

Rule 3.114: (Flow Shop Scheduling → Integer Linear Programming) Order the jobs with pairwise precedence bits and completion-time variables on every machine; the deadline becomes a makespan bound.
Overhead: `num_vars = num_jobs * (num_jobs + -1 * 1) * 2^-1 + num_jobs * num_processors`,

`num_constraints = num_jobs * (num_jobs + -1 * 1) * 2^-1 + num_jobs + num_jobs * (num_processors + -1 * 1) + num_jobs * (num_jobs + -1 * 1) * num_processors + num_jobs`.

Proof: *Construction.* Let $q \in \{1, ..., m\}$ index the machines, let $p_{j,q} = \ell(t_q[j])$ be the processing time of job j on machine q, and let $M = D + \max_{j,q} p_{j,q}$. Variables: binary $y_{i,j}$ with $y_{i,j} = 1$ iff job i precedes job j, and integer completion times $C_{j,q}$. The ILP is:

$$\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & y_{i,j} + y_{j,i} = 1 \quad \forall i \neq j \\
& C_{j,1} \geq p_{j,1} \quad \forall j \\
& C_{j,q+1} \geq C_{j,q} + p_{j,q+1} \quad \forall j, q \in \{1, ..., m-1\} \\
& C_{j,q} \geq C_{i,q} + p_{j,q} - M(1 - y_{i,j}) \quad \forall i \neq j, q \in \{1, ..., m\} \\
& C_{j,m} \leq D \quad \forall j \\
& y_{i,j} \in \{0, 1\}, C_{j,q} \in \mathbb{Z}_{\geq 0}
\end{aligned}$$

.

Correctness. ($\Rightarrow$) Any feasible flow-shop permutation induces a total order and completion times satisfying the machine and deadline constraints. ($\Leftarrow$) Any feasible ILP solution defines one common order of the jobs on all machines, and the resulting schedule completes by the deadline.

Solution extraction. Sort the jobs by their final-machine completion times $C_{j,m}$ and convert that permutation to Lehmer code. □

Rule 3.115: (Open Shop Scheduling → Integer Linear Programming) Binary ordering variables and integer start times encode the disjunctive non-overlap constraints for both machines and jobs; the makespan is the minimized objective.
Overhead: `num_vars = num_jobs * (num_jobs + -1 * 1) * 2^-1 * num_machines + num_jobs * num_machines + num_jobs * num_machines * (num_machines + -1 * 1) * 2^-1 + 1`, `num_constraints = num_jobs * (num_jobs + -1 * 1) * 2^-1 * num_machines + num_jobs * num_machines + 1 + 2 * num_jobs * (num_jobs + -1 * 1) * 2^-1 * num_machines + num_jobs * num_machines * (num_machines + -1 * 1) * 2^-1 + 2 * num_jobs * num_machines * (num_machines + -1 * 1) * 2^-1 + num_jobs * num_machines`.

Proof: *Construction.* Let $M = \sum_{j,i} p(j, i)$ be the big-M constant (an upper bound on the makespan). For each pair $j < k$ and each machine i, let $x_{\{jki\}} \in \{0, 1\}$ with $x_{\{jki\}} = 1$ iff job j precedes job k on machine i. For each job j and pair of machines $i < i'$, let $y_{\{jii'\}} \in \{0, 1\}$ with $y_{\{jii'\}} = 1$ iff machine i is processed before machine i' for job j. Let $s_{\{j,i\}} \in \mathbb{Z}_{\{\geq 0\}}$ be the start time of job j on machine i, and C be the integer makespan variable. The ILP is:

$$\begin{aligned}
\min \quad & C \\
\text{subject to} \quad & s_{k,i} - s_{j,i} - M x_{jki} \geq p(j, i) - M \quad \forall j < k, i \\
& s_{j,i} - s_{k,i} + M x_{jki} \geq p(k, i) \quad \forall j < k, i \\
& s_{j,i'} - s_{j,i} - M y_{jii'} \geq p(j, i) - M \quad \forall j, i < i' \\
& s_{j,i} - s_{j,i'} + M y_{jii'} \geq p(j, i') \quad \forall j, i < i' \\
& C - s_{j,i} \geq p(j, i) \quad \forall j, i \\
& x_{jki}, y_{jii'} \in \{0, 1\},; s_{j,i}, C \in \mathbb{Z}_{\geq 0}
\end{aligned}$$

.

Correctness. ($\Rightarrow$) Any feasible open-shop schedule with the given permutations σ_i induces valid ordering bits $x_{\{jki\}}$ and $y_{\{jii'\}}$ and start times satisfying all non-overlap constraints. ($\Leftarrow$) Any feasible ILP solution defines non-overlapping start times for all tasks, respecting both machine and job constraints.

Solution extraction. For each machine i, sort jobs by their ILP start times $s_{\{j,i\}}$ to recover the per-machine permutation; output the concatenation of these m direct-index permutations. □

Rule 3.116: (Minimum Tardiness Sequencing → Integer Linear Programming) A position-assignment ILP captures the permutation, the precedence constraints, and a binary tardy indicator for each unit-length task.
Overhead: `num_vars = num_tasks * num_tasks + num_tasks`, `num_constraints = 2 * num_tasks + num_precedences + num_tasks`.

Proof: *Construction.* Variables: binary $x_{j,p}$ placing task j in position $p \in \{0, ..., n-1\}$ and binary tardy indicators u_j, where $M = n$. The ILP is:

$$\begin{aligned}
\min \quad & \sum_j u_j \\
\text{subject to} \quad & \sum_p x_{j,p} = 1 \quad \forall j \\
& \sum_j x_{j,p} = 1 \quad \forall p \\
& \sum_p p x_{i,p} + 1 \leq \sum_p p x_{j,p} \quad \text{for each precedence } (i,j) \\
& \sum_p (p+1) x_{j,p} - d_j \leq M u_j \quad \forall j \\
& x_{j,p}, u_j \in \{0,1\}
\end{aligned}$$

.

Correctness. (⇒) Any feasible schedule gives a permutation and tardy bits with objective equal to the number of tardy tasks. (⇐) Any feasible ILP assignment decodes to a precedence-respecting permutation, and each u_j is forced to record whether task j misses its deadline.

Solution extraction. Decode the permutation from $x_{j,p}$ and encode it as Lehmer code. □

Rule 3.117: (Resource Constrained Scheduling → Integer Linear Programming) The source witness is already a time-slot assignment, so a standard time-indexed ILP suffices.
Overhead: `num_vars = num_tasks * deadline`, `num_constraints = num_tasks + deadline + num_resources * deadline`.

Proof: *Construction.* Variables: binary $x_{j,t}$ with $x_{j,t} = 1$ iff task j is run in slot $t \in \{0, ..., D-1\}$, where $r_{j,q} = R_{q(t_j)}$ denotes the amount of resource q consumed by task j. The ILP is:

$$\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_t x_{j,t} = 1 \quad \forall j \\
& \sum_j x_{j,t} \leq m \quad \forall t \\
& \sum_j r_{j,q} x_{j,t} \leq B_q \quad \forall q, t \\
& x_{j,t} \in \{0,1\}
\end{aligned}$$

.

Correctness. (⇒) Any feasible schedule chooses one slot per task while respecting processor and resource capacities in every period. (⇐) Any feasible ILP solution directly gives such a slot assignment.

Solution extraction. Task j is assigned to the unique slot t with $x_{j,t} = 1$. □

Rule 3.118: (Sequencing to Minimize Maximum Cumulative Cost → Integer Linear Programming) Assign each task to one position in the permutation and bound the running cumulative cost at every prefix.
Overhead: `num_vars = num_tasks * num_tasks + 1`, `num_constraints = 2 * num_tasks + num_precedences + num_tasks + num_tasks * num_tasks`.

Proof: *Construction.* Variables: binary $x_{j,p}$ with $x_{j,p} = 1$ iff task j is scheduled in position p. The ILP is:

$$\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{p} x_{j,p} = 1 \quad \forall j \\
& \sum_{j} x_{j,p} = 1 \quad \forall p \\
& \sum_{p} p x_{i,p} + 1 \leq \sum_{p} p x_{j,p} \quad \text{for each precedence } (i,j) \\
& \sum_{j} \sum_{p \in \{0,\ldots,q\}} c_j x_{j,p} \leq K \quad \forall q \\
& x_{j,p} \in \{0,1\}
\end{aligned}$$

.

Correctness. ($\Rightarrow$) A feasible permutation satisfies the precedence constraints and keeps every prefix sum at most K. ($\Leftarrow$) Any feasible ILP assignment is a permutation whose cumulative cost after each prefix is exactly the linear expression being bounded.

Solution extraction. Decode the position assignment and convert the resulting permutation to Lehmer code. □

Rule 3.119: (Sequencing to Minimize Tardy Task Weight → Integer Linear Programming) Place each task in exactly one schedule position with a binary tardy indicator forced on whenever the completion time at that position exceeds the task's deadline.
Overhead: `num_vars = num_tasks * num_tasks + num_tasks`, `num_constraints = 2 * num_tasks + num_tasks * num_tasks`.

Proof: *Construction.* Variables: binary $x_{j,p}$ with $x_{j,p} = 1$ iff task j occupies position p, and binary tardy indicator u_j. Let $M = \sum_j \ell_j$. The ILP is:

$$\begin{aligned}
\text{minimize} \quad & \sum_{j} w_j u_j \\
\text{subject to} \quad & \sum_{p} x_{j,p} = 1 \quad \forall j \\
& \sum_{j} x_{j,p} = 1 \quad \forall p \\
& M x_{j,p} + \sum_{p' < p} \sum_{j'} \ell_{j'} x_{j',p'} - M u_j \leq d_j - \ell_j + M \quad \forall j, p \\
& x_{j,p} \in \{0,1\}, u_j \in \{0,1\}
\end{aligned}$$

. The third family of constraints enforces: if task j is at position p (so $x_{j,p} = 1$), then its completion time $\ell_j + \sum_{p' < p} \sum_{j'} \ell_{j'} x_{j',p'}$ exceeds d_j only when $u_j = 1$.

Correctness. ($\Rightarrow$) Any schedule induces completion times; for each tardy task the big-M constraint forces $u_j = 1$, so the objective counts exactly the total tardy weight. ($\Leftarrow$) Any feasible ILP assignment is a valid permutation (by the assignment constraints) and the tardy indicators agree with the actual completion times.

Solution extraction. Read the unique position p with $x_{j,p} = 1$ for each task j to recover the schedule permutation. □

Rule 3.120: (Sequencing with Deadlines and Set-Up Times → Integer Linear Programming) Assign tasks to positions with switch-detection auxiliaries that gate per-compiler setup costs into the deadline constraints.
Overhead: `num_vars = num_tasks * num_tasks + num_tasks + -1 * 1 + num_tasks * (num_tasks + -1 * 1)`, `num_constraints = 2 * num_tasks + num_tasks^2 * (num_tasks + -1 * 1) + 3 * num_tasks * (num_tasks + -1 * 1) + num_tasks * num_tasks`.

Proof: *Construction.* Let n be the number of tasks. Variables: binary $x_{j,p}$ with $x_{j,p} = 1$ iff task j occupies position p; binary sw_p for $p \geq 1$ indicating a compiler switch before position p; binary $a_{j,p} = x_{j,p} \cdot \mathrm{sw}_p$ (linearised product). Let $M = \sum_j \ell_j + \max_c s(c) \cdot (n-1)$. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{p} x_{j,p} = 1 \quad \forall j \\
& \sum_{j} x_{j,p} = 1 \quad \forall p \\
& x_{j,p} + x_{j',p-1} - \mathrm{sw}_p \leq 1 \quad \forall p \geq 1, j, j' : k(j) \neq k(j') \\
& a_{j,p} \leq x_{j,p}, \quad a_{j,p} \leq \mathrm{sw}_p, \quad x_{j,p} + \mathrm{sw}_p - a_{j,p} \leq 1 \quad \forall j, p \geq 1 \\
& M x_{j,p} + \sum_{p'<p} \sum_{j'} \ell_{j'} x_{j',p'} + \sum_{p'=1}^{p} \sum_{j'} s(k(j')) a_{j',p'} \leq d_j - \ell_j + M \quad \forall j, p \\
& x_{j,p}, \mathrm{sw}_p, a_{j,p} \in \{0,1\}
\end{aligned}
$$

. The switch-detection row forces $\mathrm{sw}_p = 1$ whenever the tasks at positions $p-1$ and p use different compilers. The $a_{j,p}$ linearisation then routes the correct per-compiler setup time into the completion-time bound for each position.

Correctness. ($\Rightarrow$) Any feasible schedule assigns each task to a position; the switch indicator equals one exactly when consecutive compilers differ, and the deadline constraint is satisfied by hypothesis. ($\Leftarrow$) Any feasible ILP solution is a valid permutation and the deadline bound ensures each task finishes on time accounting for all setup penalties.

Solution extraction. Read the unique position p with $x_{j,p} = 1$ for each task j to recover the schedule permutation. □

Rule 3.121: (Sequencing to Minimize Weighted Tardiness → Integer Linear Programming) Encode the single-machine order with pairwise precedence bits and completion times, then linearize the weighted tardiness bound with nonnegative tardiness variables.
Overhead: `num_vars = num_tasks * (num_tasks + -1 * 1) * 2^-1 + 2 * num_tasks`, `num_constraints = num_tasks * (num_tasks + -1 * 1) * 2^-1 + num_tasks + num_tasks * (num_tasks + -1 * 1) + 2 * num_tasks + 1`.

Proof: *Construction.* Variables: binary $y_{i,j}$ with $y_{i,j} = 1$ iff job i precedes job j, integer completion times C_j, and nonnegative tardiness variables T_j, where $M = \sum_j \ell_j$ is a valid schedule-horizon bound. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & y_{i,j} + y_{j,i} = 1 \quad \forall i \neq j \\
& C_j \geq \ell_j \quad \forall j \\
& C_j \geq C_i + \ell_j - M(1 - y_{i,j}) \quad \forall i \neq j \\
& T_j \geq C_j - d_j \quad \forall j \\
& T_j \geq 0 \quad \forall j \\
& \sum_j w_j T_j \leq K \\
& y_{i,j} \in \{0,1\}, C_j \in \mathbb{Z}_{\geq 0}, T_j \in \mathbb{Z}_{\geq 0}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any job order induces completion times and tardiness values satisfying the bound exactly when the source instance is feasible. ($\Leftarrow$) Any feasible ILP solution yields a single-machine order whose weighted tardiness equals the encoded linear objective term.

Solution extraction. Sort the jobs by C_j and encode that permutation as Lehmer code. □

Rule 3.122: (Sequencing with Release Times and Deadlines → Integer Linear Programming) A time-indexed formulation captures the admissible start window of each task and forbids overlap on the single machine. *Overhead:* `num_vars = num_tasks * time_horizon`, `num_constraints = num_tasks + time_horizon`.

Proof: *Construction.* Variables: binary $x_{j,t}$ with $x_{j,t} = 1$ iff task j starts at time t, where $p_j = \ell(t_j)$ is the processing time (length) of task j. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{t=r_j}^{d_j - p_j} x_{j,t} = 1 \quad \forall j \\
& \sum_{j,t:t \leq \tau < t+p_j} x_{j,t} \leq 1 \quad \forall \tau \\
& x_{j,t} \in \{0,1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any feasible non-preemptive schedule chooses one valid start time per task and never overlaps two active jobs. ($\Leftarrow$) Any feasible ILP solution gives exactly such a start-time assignment, so executing the jobs in increasing start order solves the source instance.

Solution extraction. Read each task's chosen start time, sort the tasks by that order, and encode the resulting permutation as Lehmer code. □

Rule 3.123: (Timetable Design → Integer Linear Programming) The source witness is a binary craftsman-task-period incidence table, and all feasibility conditions are already linear.
Overhead: `num_vars = num_craftsmen * num_tasks * num_periods`, `num_constraints = num_craftsmen * num_periods + num_tasks * num_periods + num_craftsmen * num_tasks`.

Proof: *Construction.* Variables: binary $x_{c,t,h}$ with $x_{c,t,h} = 1$ iff craftsman c works on task t in period h. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & x_{c,t,h} = 0 \quad \text{whenever either side is unavailable} \\
& \sum_{t} x_{c,t,h} \leq 1 \quad \forall c, h \\
& \sum_{c} x_{c,t,h} \leq 1 \quad \forall t, h \\
& \sum_{h} x_{c,t,h} = r_{c,t} \quad \forall c, t \\
& x_{c,t,h} \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any valid timetable satisfies availability, exclusivity, and exact requirement counts. ($\Leftarrow$) Any feasible ILP solution is exactly such a timetable because the variable layout matches the source configuration.

Solution extraction. Output the flattened binary array $(x_{c,t,h})$ in source order. □

Rule 3.124: (Hamiltonian Path $\rightarrow$ Integer Linear Programming) Place each vertex in exactly one path position and use auxiliary variables for consecutive pairs so only graph edges may appear between adjacent positions.
Overhead: `num_vars = num_vertices^2 + 2 * num_edges * num_vertices`, `num_constraints = 2 * num_vertices + 6 * num_edges * num_vertices + num_vertices`.

Proof: *Construction.* Variables: binary $x_{v,p}$ with $x_{v,p} = 1$ iff vertex v is placed at position p, and binary $z_{(u,v),p}$ linearizing $x_{u,p} x_{v,p+1}$. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{p} x_{v,p} = 1 \quad \forall v \\
& \sum_{v} x_{v,p} = 1 \quad \forall p \\
& z_{(u,v),p} \leq x_{u,p} \quad \forall (u,v), p \\
& z_{(u,v),p} \leq x_{v,p+1} \quad \forall (u,v), p \\
& z_{(u,v),p} \geq x_{u,p} + x_{v,p+1} - 1 \quad \forall (u,v), p \\
& \sum_{(u,v) \in E} z_{(u,v),p} = 1 \quad \forall p \\
& x_{v,p}, z_{(u,v),p} \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A Hamiltonian path defines a permutation of the vertices and therefore a feasible assignment matrix with one admissible graph edge between every consecutive pair. ($\Leftarrow$) Any feasible ILP solution is a vertex permutation whose consecutive pairs are graph edges, hence a Hamiltonian path.

Solution extraction. For each position p, output the unique vertex v with $x_{v,p} = 1$. □

Rule 3.125: (Directed Hamiltonian Path $\rightarrow$ Integer Linear Programming) Assign each vertex to exactly one path position and forbid non-arc pairs at consecutive positions.
Overhead: `num_vars = num_vertices^2`, `num_constraints = 3 * num_vertices + (num_vertices + -1 * 1) * (num_vertices^2 + -1 * num_arcs)`.

Proof: *Construction.* Variables: binary $x_{v,k}$ with $x_{v,k} = 1$ iff vertex v is placed at position k. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{k} x_{v,k} = 1 \quad \forall v \\
& \sum_{v} x_{v,k} = 1 \quad \forall k \\
& x_{v,k} + x_{w,k+1} \leq 1 \quad \forall k, (v,w) \notin A \\
& x_{v,k} \in \{0,1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A directed Hamiltonian path yields a permutation where every consecutive pair is a directed arc; the arc-exclusion constraints are satisfied by definition. ($\Leftarrow$) Any feasible ILP solution defines a vertex permutation whose consecutive pairs are all directed arcs, hence a directed Hamiltonian path.

Solution extraction. For each position k, decode the unique vertex v with $x_{v,k} = 1$ to recover the permutation; convert to Lehmer code for the source configuration. □

Rule 3.126: (Bottleneck Traveling Salesman → Integer Linear Programming) Use a cyclic position assignment for the tour and a bottleneck variable that dominates the weight of every chosen tour edge.
Overhead: `num_vars = num_vertices^2 + 2 * num_edges * num_vertices + 1`, `num_constraints = 2 * num_vertices + num_vertices^2 + 2 * num_edges * num_vertices + 6 * num_edges * num_vertices + num_vertices + 2 * num_edges * num_vertices`.

Proof: *Construction.* Variables: binary $x_{v,p}$ for city-position assignment, binary $z_{(u,v),p}$ for consecutive tour edges, and integer bottleneck variable b. The ILP is:

$$
\begin{aligned}
\min \quad & b \\
\text{subject to} \quad & \sum_{p} x_{v,p} = 1 \quad \forall v \\
& \sum_{v} x_{v,p} = 1 \quad \forall p \\
& z_{(u,v),p} \leq x_{u,p} \quad \forall (u,v), p \\
& z_{(u,v),p} \leq x_{v,(p+1) \bmod n} \quad \forall (u,v), p \\
& z_{(u,v),p} \geq x_{u,p} + x_{v,(p+1) \bmod n} - 1 \quad \forall (u,v), p \\
& \sum_{(u,v) \in E} z_{(u,v),p} = 1 \quad \forall p \\
& b \geq w_{u,v} z_{(u,v),p} \quad \forall (u,v), p \\
& x_{v,p}, z_{(u,v),p} \in \{0,1\}, b \in \mathbb{Z}_{\geq 0}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any Hamiltonian tour yields a feasible assignment and sets b to the maximum selected edge weight. ($\Leftarrow$) Any feasible ILP solution encodes a Hamiltonian cycle, and the minimax constraints force b to equal its bottleneck edge weight.

Solution extraction. Mark an edge selected in the source config iff it appears between two consecutive positions in the decoded cycle. □

Rule 3.127: (Hamiltonian Circuit → Longest Circuit (weighted)) [5] This $O(m)$ reduction copies the graph unchanged and assigns unit weight to every edge (n target vertices, m target edges). A Hamiltonian circuit exists iff the optimal circuit length equals n.
Overhead: `num_vertices = num_vertices`, `num_edges = num_edges`.

Proof: *Construction.* Given a Hamiltonian Circuit instance $G = (V, E)$ with $n = |V|$ and $m = |E|$, construct a Longest Circuit instance on the same graph $G' = G$ with edge lengths $l(e) = 1$ for every $e \in E$.

Correctness. ($\Rightarrow$) If G has a Hamiltonian circuit $v_0, v_1, ..., v_{n-1}, v_0$, then this circuit uses n edges each of length 1, giving total length n. Since a simple circuit on n vertices can use at most n edges, this is optimal. ($\Leftarrow$) If the longest circuit in G' has length n, it uses n unit-weight edges and therefore visits n distinct vertices, i.e., every vertex exactly once. This circuit is therefore a Hamiltonian circuit in G.

Solution extraction. Read the selected target edges, traverse the unique degree-2 cycle they form, and return the resulting vertex permutation as the source Hamiltonian-circuit witness. □

Example: Cycle graph on 4 vertices with unit edge lengths

Source: HamiltonianCircuit **Target:** LongestCircuit

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to LongestCircuit/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2,3
```

Step 1 – Start from the source graph. The canonical source fixture is the cycle on vertices $\{0, 1, ..., 3\}$ with 4 edges. The stored Hamiltonian-circuit witness is the permutation [0, 1, 2, 3].

Step 2 – Assign unit edge lengths. The target keeps the same 4 vertices and 4 edges. Every edge receives length 1, so the edge-length vector is [1, 1, 1, 1].

Step 3 – Verify the canonical witness. The stored target configuration [1, 1, 1, 1] selects the edges $(0, 1)$, $(1, 2)$, $(2, 3)$, $(3, 0)$. The total circuit length is $4 \times 1 = 4 = n$, confirming a Hamiltonian circuit. Traversing the selected edges recovers the vertex permutation [0, 1, 2, 3].

Multiplicity: The fixture stores one canonical witness. For the 4-cycle there are $4 \times 2 = 8$ directed Hamiltonian circuits (choice of start vertex and direction), but they all select the same undirected edge set.

Rule 3.128: (Longest Circuit (weighted) $\rightarrow$ Integer Linear Programming) A direct cycle-selection ILP uses binary edge variables, degree constraints, and a connectivity witness to force exactly one simple circuit of length at least the bound.
Overhead: `num_vars = num_edges + num_vertices + 2 * num_edges * (num_vertices + -1 * 1)`, `num_constraints = 1 + num_vertices^2 + 2 * num_edges * (num_vertices + -1 * 1)`.

Proof: *Construction.* Variables: binary y_e for edges, binary s_v indicating whether vertex v lies on the circuit, and root-flow variables on selected edges. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{e : v \in e} y_e = 2 s_v \quad \forall v \\
& \sum_{e} y_e \geq 3 \\
& \sum_{e} l_e y_e \geq K \\
& \text{root-flow connectivity constraints hold on the selected edges} \\
& y_e, s_v \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A simple circuit has degree 2 at each used vertex, is connected, and meets the length bound K. ($\Leftarrow$) The degree and connectivity constraints force the selected edges to form exactly one simple circuit, and the final inequality enforces the required total length.

Solution extraction. Output the binary edge-selection vector $(y_e)_{e \in E}$. □

Rule 3.129: (Quadratic Assignment → Integer Linear Programming) Assign each facility to exactly one location, enforce injectivity, and linearize every quadratic cost term with McCormick products.
Overhead: `num_vars = num_facilities * num_locations + num_facilities^2 * num_locations^2`, `num_constraints = num_facilities + num_locations + 3 * num_facilities^2 * num_locations^2`.

Proof: *Construction.* Variables: binary $x_{i,p}$ with $x_{i,p} = 1$ iff facility i is placed at location p, and binary $z_{(i,p),(j,q)}$ for the products $x_{i,p}x_{j,q}$. The ILP is:

$$\begin{aligned}
\min \quad & \sum_{i \neq j} \sum_{p,q} c_{i,j} d_{p,q} z_{(i,p),(j,q)} \\
\text{subject to} \quad & \sum_{p} x_{i,p} = 1 \quad \forall i \\
& \sum_{i} x_{i,p} \leq 1 \quad \forall p \\
& z_{(i,p),(j,q)} \leq x_{i,p} \quad \forall i, p, j, q \\
& z_{(i,p),(j,q)} \leq x_{j,q} \quad \forall i, p, j, q \\
& z_{(i,p),(j,q)} \geq x_{i,p} + x_{j,q} - 1 \quad \forall i, p, j, q \\
& x_{i,p}, z_{(i,p),(j,q)} \in \{0, 1\}
\end{aligned}$$

.

Correctness. ($\Rightarrow$) Any injective facility placement gives a feasible ILP assignment with exactly the same quadratic cost. ($\Leftarrow$) Any feasible ILP solution decodes to an injective facility-to-location map, and the linearized objective equals the source objective term by term.

Solution extraction. For each facility i, output the unique location p with $x_{i,p} = 1$. □

Rule 3.130: (Optimal Linear Arrangement → Integer Linear Programming) Assign each vertex to one position and use absolute-value auxiliaries to measure the length of every edge in the arrangement.
Overhead: `num_vars = num_vertices^2 + num_vertices + num_edges`, `num_constraints = 2 * num_vertices + num_vertices^2 + num_vertices + num_vertices + 3 * num_edges`.

Proof: *Construction.* Variables: binary $x_{v,p}$ with $x_{v,p} = 1$ iff vertex v gets position p, integer position variables $p_v = \sum_p p x_{v,p}$, and nonnegative $z_{u,v}$ per edge $\{u, v\}$. The ILP is:

$$\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{p} x_{v,p} = 1 \quad \forall v \\
& \sum_{v} x_{v,p} = 1 \quad \forall p \\
& z_{u,v} \geq p_u - p_v \quad \forall \{u, v\} \in E \\
& z_{u,v} \geq p_v - p_u \quad \forall \{u, v\} \in E \\
& \sum_{\{u,v\} \in E} z_{u,v} \leq K \\
& x_{v,p} \in \{0, 1\}, z_{u,v} \in \mathbb{Z}_{\geq 0}
\end{aligned}$$

.

Correctness. ($\Rightarrow$) Any valid linear arrangement satisfies the permutation constraints and gives edge lengths $|p_u - p_v|$ within the bound. ($\Leftarrow$) Any feasible ILP solution is a bijection from vertices to positions, and the auxiliary variables exactly upper-bound the edge lengths, so the total arrangement cost is at most K.

Solution extraction. For each vertex v, output its decoded position p_v. □

Rule 3.131: (Subgraph Isomorphism → Integer Linear Programming) Choose an injective image of every pattern vertex in the host graph and forbid any mapped pattern edge from landing on a host non-edge.
Overhead: `num_vars = num_pattern_vertices * num_host_vertices`, `num_constraints = num_pattern_vertices + num_host_vertices + num_pattern_edges * num_host_vertices^2`.

Proof: *Construction.* Variables: binary $x_{v,u}$ with $x_{v,u} = 1$ iff pattern vertex v maps to host vertex u. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{u} x_{v,u} = 1 \quad \forall v \\ & \sum_{v} x_{v,u} \le 1 \quad \forall u \\ & x_{v,u} + x_{w,u'} \le 1 \quad \forall \{v,w\} \in E_{\text{pat}}, \{u,u'\} \notin E_{\text{host}} \\ & x_{v,u'} + x_{w,u} \le 1 \quad \forall \{v,w\} \in E_{\text{pat}}, \{u,u'\} \notin E_{\text{host}} \\ & x_{v,u} \in \{0,1\} \end{aligned}$$

.

Correctness. ($\Rightarrow$) Any injective edge-preserving embedding satisfies the assignment and non-edge constraints. ($\Leftarrow$) Any feasible ILP solution is an injective vertex map, and the non-edge inequalities ensure every pattern edge is sent to a host edge.

Solution extraction. For each pattern vertex v, output the unique host vertex u with $x_{v,u} = 1$. □

Rule 3.132: (Acyclic Partition (weighted) → Integer Linear Programming) Assign every vertex to one partition class, bound the weight and crossing cost of those classes, and impose a topological order on the quotient digraph.
Overhead: `num_vars = num_vertices * num_vertices + num_arcs * num_vertices + num_arcs + 2 * num_vertices`, `num_constraints = num_vertices + num_vertices + num_arcs * num_vertices + num_arcs + 1 + 2 * num_vertices + 2 * num_vertices * num_vertices + num_arcs`.

Proof: *Construction.* Let $n = |V|$ and let the directed arcs be $A = \{a_0, ..., a_{m-1}\}$ with $a_t = (u_t \to v_t)$. The source witness already allows every vertex to choose one label in $\{0, ..., n-1\}$, so the ILP uses exactly the same label range. Use `ILP<i32>` with variable order $(x_{v,c})_{v,c}, (s_{t,c})_{t,c}, (y_t)_t, (o_c)_c, (p_v)_v$. The indices are $\text{idx}_{x(v,c)} = vn + c$, $\text{idx}_{s(t,c)} = n^2 + tn + c$, $\text{idx}_{y(t)} = n^2 + mn + t$, $\text{idx}_{o(c)} = n^2 + mn + m + c$, and $\text{idx}_{p(v)} = n^2 + mn + m + n + v$. There are $n^2 + mn + m + 2n$ variables.

Here $x_{v,c} \in \{0,1\}$ means vertex v is assigned to class c, $s_{t,c} \in \{0,1\}$ means both endpoints of arc a_t lie in class c, $y_t \in \{0,1\}$ marks that arc a_t crosses between two different classes, $o_c \in \{0, ..., n-1\}$ is the order assigned to class c, and $p_v \in \{0, ..., n-1\}$ copies the order of the class chosen by vertex v.

The constraints are: $\sum_{c=0}^{n-1} x_{v,c} = 1$ for every vertex v; $\sum_v w_v x_{v,c} \le B$ for every class c; $s_{t,c} \le x_{u_t,c}$, $s_{t,c} \le x_{v_t,c}$, and $s_{t,c} \ge x_{u_t,c} + x_{v_t,c} - 1$ for every arc a_t and class c; $y_t + \sum_{c=0}^{n-1} s_{t,c} = 1$ for every arc a_t, so $y_t = 1$ exactly for crossing arcs; $\sum_{t=0}^{m-1} \text{cost}(a_t) y_t \le K$; $0 \le o_c \le n-1$ and $0 \le p_v \le n-1$ for all classes c and vertices v; $p_v - o_c \le (n-1)(1 - x_{v,c})$ and $o_c - p_v \le (n-1)(1 - x_{v,c})$ for all v, c, so $p_v = o_c$ whenever $x_{v,c} = 1$; and for every arc $a_t = (u_t \to v_t)$, $p_{v_t} - p_{u_t} \ge 1 - n \sum_{c=0}^{n-1} s_{t,c}$. The exact big-$M$ here is $M = n$: if u_t and v_t lie in the same class, then $\sum_c s_{t,c} = 1$ and the right-hand side is $1 - n = -(n-1)$, which is precisely the smallest possible difference between two order variables in $\{0, ..., n-1\}$. If the arc crosses between two distinct classes, then $\sum_c s_{t,c} = 0$ and the inequality becomes $p_{v_t} - p_{u_t} \ge 1$. For the realized classes c and d of the endpoints, this is exactly the requested form $o_d - o_c \ge 1 - M \sum_h s_{t,h}$.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{c=0}^{n-1} x_{v,c} = 1 \quad \forall v \in V \\
& \sum_{v} w_v x_{v,c} \leq B \quad \forall c \in \{0, ..., n-1\} \\
& s_{t,c} \leq x_{u_t,c}, s_{t,c} \leq x_{v_t,c} \quad \forall t, c \\
& s_{t,c} \geq x_{u_t,c} + x_{v_t,c} - 1 \quad \forall t, c \\
& y_t + \sum_{c=0}^{n-1} s_{t,c} = 1 \quad \forall t \in \{0, ..., m-1\} \\
& \sum_{t=0}^{m-1} \text{cost}(a_t) y_t \leq K \\
& p_v - o_c \leq (n-1)\big(1 - x_{v,c}\big), o_c - p_v \leq (n-1)\big(1 - x_{v,c}\big) \quad \forall v, c \\
& p_{v_t} - p_{u_t} \geq 1 - n \sum_{c=0}^{n-1} s_{t,c} \quad \forall t \in \{0, ..., m-1\} \\
& x_{v,c}, s_{t,c}, y_t \in \{0,1\}; o_c, p_v \in \{0, ..., n-1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any valid acyclic partition gives a class assignment whose quotient arcs respect some topological ordering, with the same class weights and crossing cost. ($\Leftarrow$) Any feasible ILP solution partitions the vertices, keeps every class within the weight bound, charges exactly the inter-class arcs, and the order variables force the quotient digraph to be acyclic.

Solution extraction. For each vertex v, output the unique class label c with $x_{v,c} = 1$. □

Rule 3.133: (Balanced Complete Bipartite Subgraph → Integer Linear Programming) Choose exactly k vertices on each side of the bipartite graph and forbid any selected left-right pair that is not an edge.
Overhead: `num_vars = num_vertices`, `num_constraints = num_vertices * num_vertices`.

Proof: *Construction.* Let L and R be the bipartition. Variables: binary x_l for $l \in L$ and y_r for $r \in R$. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{l \in L} x_l = k \\
& \sum_{r \in R} y_r = k \\
& x_l + y_r \leq 1 \quad \forall (l, r) \notin E \\
& x_l, y_r \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A balanced complete bipartite subgraph of size $k + k$ satisfies the cardinality constraints and has no selected non-edge pair. ($\Leftarrow$) Any feasible ILP solution selects k left vertices and k right vertices with every cross-pair present, hence a balanced biclique.

Solution extraction. Output the concatenated left/right binary selection vector. □

Rule 3.134: (Biclique Cover → Integer Linear Programming) Use k candidate bicliques, assign vertices to any of them, force every graph edge to be covered by some common biclique, and minimize the total membership size.
Overhead: `num_vars = num_vertices * rank + num_vertices * num_vertices * rank`, `num_constraints = num_vertices * num_vertices * rank + num_edges`.

Proof: *Construction.* Variables: binary $x_{l,b}$ for left vertices, binary $y_{r,b}$ for right vertices, and binary $z_{(l,r),b}$ linearizing $x_{l,b}y_{r,b}$. The ILP is:

$$\begin{aligned}
\min \quad & \sum_{l,b} x_{l,b} + \sum_{r,b} y_{r,b} \\
\text{subject to} \quad & z_{(l,r),b} \le x_{l,b} \quad \forall l, r, b \\
& z_{(l,r),b} \le y_{r,b} \quad \forall l, r, b \\
& z_{(l,r),b} \ge x_{l,b} + y_{r,b} - 1 \quad \forall l, r, b \\
& \sum_{b} z_{(l,r),b} \ge 1 \quad \forall (l,r) \in E \\
& x_{l,b} + y_{r,b} \le 1 \quad \forall (l,r) \notin E, b \\
& x_{l,b}, y_{r,b}, z_{(l,r),b} \in \{0,1\}
\end{aligned}$$

.

Correctness. ($\Rightarrow$) Any valid k-biclique cover assigns each covered edge to a biclique containing both endpoints, with objective equal to the total biclique size. ($\Leftarrow$) Any feasible ILP solution defines k complete bipartite subgraphs whose union covers every edge, and the objective is exactly the source objective.

Solution extraction. Output the flattened vertex-by-biclique membership bits and discard the coverage auxiliaries. □

Rule 3.135: (Biconnectivity Augmentation (weighted) → Integer Linear Programming) Select candidate edges under the budget and, for every deleted vertex, certify that the remaining augmented graph stays connected by a flow witness.
Overhead: `num_vars = num_potential_edges + 2 * num_vertices * num_vertices * (num_edges + num_potential_edges)`, `num_constraints = 1 + 2 * num_vertices * num_vertices * num_potential_edges + num_vertices * num_vertices * num_vertices`.

Proof: *Construction.* Let the base graph edges be $E = \{e_0, ..., e_{m-1}\}$ with $e_i = \{u_i, v_i\}$, and let the candidate edges be $F = \{f_0, ..., f_{p-1}\}$ with $f_j = \{s_j, t_j\}$. If $n = |V| \le 1$, return the empty feasible ILP, since every 0- or 1-vertex graph is already biconnected in the model. Otherwise fix, for each deleted vertex q, the surviving root $r_q = 0$ if $q \neq 0$, and $r_0 = 1$. This choice is explicit and valid because $n \ge 2$.

Use `ILP<i32>`. The candidate-selection bits are $y_j \in \{0,1\}$ with index j. For the connectivity witnesses, allocate the full (q,t) commodity grid with $q, t \in \{0, ..., n-1\}$, even though the commodities with $t = q$ or $t = r_q$ will be pinned to 0. For each base edge e_i and orientation flag $\eta \in \{0,1\}$, let $\eta = 0$ mean $u_i \to v_i$ and $\eta = 1$ mean $v_i \to u_i$; define binary flow variables $f^{q,t}_{i,\eta}$ with index $p + (((qn + t)m + i)2 + \eta)$. For each candidate edge f_j and orientation flag $\eta \in \{0,1\}$, let $\eta = 0$ mean $s_j \to t_j$ and $\eta = 1$ mean $t_j \to s_j$; define binary flow variables $g^{q,t}_{j,\eta}$ with index $p + 2mn^2 + (((qn + t)p + j)2 + \eta)$. There are $p + 2n^2(m + p)$ variables in total.

The constraints are: $\sum_{j=0}^{p-1} w_j y_j \le B$; for every deleted vertex q and target t, if $t = q$ or $t = r_q$, set all $f^{q,t}_{i,\eta}$ and $g^{q,t}_{j,\eta}$ equal to 0; if the deleted vertex q is incident to base edge e_i or candidate edge f_j, set the corresponding directed flow variables for that (q,t) to 0, because they do not exist in $G - q$; for each candidate edge variable, the exact activation big-M is 1: $g^{q,t}_{j,\eta} \le y_j$ for every q, t, j, η; and for every valid pair (q,t) with $t \notin \{q, r_q\}$ and every surviving vertex $v \neq q$, impose flow conservation $\sum_{\text{out of v}}(f^{q,t} + g^{q,t}) - \sum_{\text{into v}}(f^{q,t} + g^{q,t}) = 1$ when $v = r_q$, $= -1$ when $v = t$, and $= 0$ otherwise. The sums range over both orientations of all base and candidate edges that avoid q. Since every commodity carries exactly one unit, binary flows are sufficient.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{j=0}^{p-1} w_j y_j \leq B \\
& f_{i,\eta}^{q,t} = 0 \quad \text{whenever } t \in \{q, r_q\} \text{ or } e_i \text{ is incident to } q \\
& g_{j,\eta}^{q,t} = 0 \quad \text{whenever } t \in \{q, r_q\} \text{ or } f_j \text{ is incident to } q \\
& g_{j,\eta}^{q,t} \leq y_j \quad \forall q, t \in \{0, ..., n-1\}, j \in \{0, ..., p-1\}, \eta \in \{0, 1\} \\
& \text{for each valid pair } (q, t) \text{ , unit-flow conservation from } r_q \text{ to } t \text{ holds in } G - q \\
& y_j, f_{i,\eta}^{q,t}, g_{j,\eta}^{q,t} \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) If the chosen augmentation makes the graph biconnected, then every vertex-deleted graph is connected and therefore supports the required flows. ($\Leftarrow$) If the ILP is feasible, then removing any single vertex leaves a connected graph, which is exactly the definition of biconnectivity for the augmented graph.

Solution extraction. Output the binary selection vector of candidate edges. □

Rule 3.136: (Bounded Component Spanning Forest (weighted) → Integer Linear Programming) Assign every vertex to one of at most K components, bound each component's total weight, and certify connectivity inside each used component by a flow witness.
Overhead: `num_vars = 3 * num_vertices * max_components + 2 * max_components + 2 * num_edges * max_components`, `num_constraints = num_vertices + max_components + max_components + 2 * max_components + num_vertices * max_components + 4 * num_vertices * max_components + 4 * num_edges * max_components + num_vertices * max_components`.

Proof: *Construction.* Let $n = |V|$, let the graph edges be $E = \{e_0, ..., e_{m-1}\}$ with $e_i = \{u_i, v_i\}$, and let the allowed component labels be $c \in \{0, ..., K-1\}$. Use `ILP<i32>` with variables ordered as $(x_{v,c})_{v,c}, (u_c)_c, (r_{v,c})_{v,c}, (s_c)_c, (b_{v,c})_{v,c}, (f_{i,\eta,c})_{i,\eta,c}$. Their indices are $\text{idx}_{x(v,c)} = vK + c$, $\text{idx}_{u(c)} = nK + c$, $\text{idx}_{r(v,c)} = nK + K + vK + c$, $\text{idx}_{s(c)} = 2nK + K + c$, $\text{idx}_{b(v,c)} = 2nK + 2K + vK + c$, and, with $\eta = 0$ meaning $u_i \to v_i$ and $\eta = 1$ meaning $v_i \to u_i$, $\text{idx}_{f(i,\eta,c)} = 3nK + 2K + (i2 + \eta)K + c$. There are $3nK + 2K + 2mK$ variables.

Here $x_{v,c} \in \{0, 1\}$ means vertex v is placed in component c, $u_c \in \{0, 1\}$ says that component c is nonempty, $r_{v,c} \in \{0, 1\}$ chooses the root of nonempty component c, $s_c \in \{0, ..., n\}$ is its size, $b_{v,c} \in \{0, ..., n\}$ linearizes the product $s_c r_{v,c}$, and $f_{i,\eta,c} \in \{0, ..., n-1\}$ is the root-flow on the chosen component edges.

The constraints are: $\sum_{c=0}^{K-1} x_{v,c} = 1$ for every vertex v; $\sum_v w_v x_{v,c} \leq B$ for every component label c; $s_c = \sum_v x_{v,c}$ for every c; $u_c \leq s_c$ and $s_c \leq n u_c$ for every c, so $u_c = 1$ iff the component is nonempty; $\sum_v r_{v,c} = u_c$ and $r_{v,c} \leq x_{v,c}$ for every c and v, which chooses exactly one root in every nonempty component; the product linearization $b_{v,c} \leq s_c$, $b_{v,c} \leq n r_{v,c}$, $b_{v,c} \geq s_c - n(1 - r_{v,c})$, $b_{v,c} \geq 0$ for every v, c, so $b_{v,c} = s_c r_{v,c}$; the exact big-M here is n; for every edge $e_i = \{u_i, v_i\}$, orientation flag $\eta \in \{0, 1\}$, and component c, $0 \leq f_{i,\eta,c} \leq (n-1) x_{u_i,c}$ and $0 \leq f_{i,\eta,c} \leq (n-1) x_{v_i,c}$. The exact capacity big-M is $n - 1$: a component of size s_c needs to route at most $s_c - 1 \leq n - 1$ units across any oriented edge of a spanning tree. Finally, for every vertex v and component c, $\sum_{\text{out of v in c}} f - \sum_{\text{into v in c}} f = b_{v,c} - x_{v,c}$. If v is the chosen root of component c, then the right-hand side is $s_c - 1$; every other assigned vertex consumes one unit; unassigned vertices have right-hand side 0.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{c=0}^{K-1} x_{v,c} = 1 \quad \forall v \in V \\
& \sum_{v} w_v x_{v,c} \leq B \quad \forall c \in \{0, ..., K-1\} \\
& s_c = \sum_{v} x_{v,c} \quad \forall c \in \{0, ..., K-1\} \\
& u_c \leq s_c \leq n u_c \quad \forall c \in \{0, ..., K-1\} \\
& \sum_{v} r_{v,c} = u_c, r_{v,c} \leq x_{v,c} \quad \forall v, c \\
& \text{the standard product linearization enforces } b_{v,c} = s_c r_{v,c} \quad \forall v, c \\
& 0 \leq f_{i,\eta,c} \leq (n-1) x_{u_i,c}, 0 \leq f_{i,\eta,c} \leq (n-1) x_{v_i,c} \quad \forall i, \eta, c \\
& \sum_{\text{out of v in c}} f - \sum_{\text{into v in c}} f = b_{v,c} - x_{v,c} \quad \forall v, c \\
& x_{v,c}, u_c, r_{v,c} \in \{0,1\}; s_c \in \{0, ..., n\}; b_{v,c}, f_{i,\eta,c} \in \mathbb{Z}_{\geq 0}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any valid bounded-component partition assigns each component a connected supporting subgraph and respects the weight bound. ($\Leftarrow$) Any feasible ILP solution partitions the vertices into at most K connected sets, each of total weight at most B, exactly as required by the source problem.

Solution extraction. For each vertex v, output the unique component label c with $x_{v,c} = 1$. □

Rule 3.137: (Minimum Cut Into Bounded Sets (weighted) → Integer Linear Programming) A binary side variable for each vertex, together with cut indicators on the edges, directly linearizes the bounded two-way cut conditions.
Overhead: `num_vars = num_vertices + num_edges`, `num_constraints = 2 + 2 + 2 * num_edges`.

Proof: *Construction.* Variables: binary x_v with $x_v = 1$ iff v is placed on the sink side, and binary y_e for edges. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & x_s = 0 \\
& x_t = 1 \\
& \sum_{v} x_v \leq B \\
& \sum_{v} (1 - x_v) \leq B \\
& y_e \geq x_u - x_v \quad \forall e = \{u, v\} \in E \\
& y_e \geq x_v - x_u \quad \forall e = \{u, v\} \in E \\
& \sum_{e} w_e y_e \leq K \\
& x_v, y_e \in \{0,1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any feasible bounded cut determines a 0/1 side assignment, and the edge indicators are 1 exactly on the cut edges. ($\Leftarrow$) Any feasible ILP solution partitions the vertices into two bounded sets with s and t separated and total cut weight at most K.

Solution extraction. Output the partition bit-vector $(x_v)_{v \in V}$. □

Rule 3.138: (Strong Connectivity Augmentation (weighted) → Integer Linear Programming) Select candidate arcs under the budget and certify strong connectivity by sending flow both from a root to every vertex and back again.
Overhead: `num_vars = num_potential_arcs + 2 * num_vertices * (num_arcs + num_potential_arcs)`, `num_constraints = 1 + 2 * num_vertices * num_potential_arcs + 2 * num_vertices * num_vertices`.

Proof: *Construction.* Let the base arcs be $A = \{a_0, ..., a_{m-1}\}$ with $a_i = (u_i, v_i)$, let the candidate arcs be $C = \{c_0, ..., c_{p-1}\}$ with $c_j = (s_j, t_j)$, and, when $n = |V| \geq 1$, fix the root to be vertex $r = 0$. If $n \leq 1$, return the empty feasible ILP. Use `ILP<i32>` with variables ordered as $(y_j)_j, (f_i^t)_{t,i}, \left(|(f)_j^t\right)_{t,j}, (g_i^t)_{t,i}, \left(|(g)_j^t\right)_{t,j}$, where f^t is the forward root-to-t flow on base arcs, $|(f)^t$ is the forward flow on candidate arcs, g^t is the backward t-to-root flow on base arcs, and $|(g)^t$ is the backward flow on candidate arcs. The indices are $\text{idx}_{y(j)} = j$, $\text{idx}_{f_i^t} = p + tm + i$, $\text{idx}_{|(f)_j^t} = p + nm + tp + j$, $\text{idx}_{g_i^t} = p + n(m + p) + tm + i$, and $\text{idx}_{|(g)_j^t} = p + n(2m + p) + tp + j$. There are $p + 2n(m + p)$ variables.

The constraints are: $\sum_{j=0}^{p-1} w_j y_j \leq B$; for the dummy commodity $t = r$, set all four flow blocks f_i^r, $|(f)_j^r$, g_i^r, $|(g)_j^r$ to 0; for every candidate arc and target vertex, use the exact activation big-$M = 1$: $|(f)_j^t \leq y_j$ and $|(g)_j^t \leq y_j$ for all t, j; for every $t \neq r$ and every vertex v, $\sum_{\text{out of v}}(f^t + |(f)^t) - \sum_{\text{into v}}(f^t + |(f)^t) = 1$ when $v = r$, $= -1$ when $v = t$, and $= 0$ otherwise; and $\sum_{\text{out of v}}(g^t + |(g)^t) - \sum_{\text{into v}}(g^t + |(g)^t) = 1$ when $v = t$, $= -1$ when $v = r$, and $= 0$ otherwise. All flow variables are binary, because each commodity carries a single unit.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{j=0}^{p-1} w_j y_j \leq B \\
& f_i^r = 0, |(f)_j^r = 0, g_i^r = 0, |(g)_j^r = 0 \\
& |(f)_j^t \leq y_j, |(g)_j^t \leq y_j \quad \forall t \in \{0, ..., n-1\}, j \in \{0, ..., p-1\} \\
& \text{root-to-target unit-flow conservation holds on } f^t, |(f)^t \quad \forall t \neq r \\
& \text{target-to-root unit-flow conservation holds on } g^t, |(g)^t \quad \forall t \neq r \\
& y_j, f_i^t, |(f)_j^t, g_i^t, |(g)_j^t \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A strongly connected augmentation provides both directions of reachability between the root and every other vertex, hence all required flows. ($\Leftarrow$) If those flows exist for every vertex, then every vertex is reachable from the root and can reach the root, so the augmented digraph is strongly connected.

Solution extraction. Output the binary candidate-arc selection vector (y_a). □

Rule 3.139: (Boolean Matrix Factorization → Integer Linear Programming) Split the witness into binary factor matrices B and C, reconstruct their Boolean product with McCormick auxiliaries, and minimize the Hamming distance to the target matrix.
Overhead: `num_vars = rows * rank + rank * cols + rows * rank * cols + rows * cols + rows * cols`, `num_constraints = 3 * rows * rank * cols + rank * rows * cols + rows * cols + 2 * rows * cols`.

Proof: *Construction.* Variables: binary $b_{i,r}$, binary $c_{r,j}$, binary $p_{i,r,j}$ linearizing $b_{i,r}c_{r,j}$, binary $w_{i,j}$ for the reconstructed entry, and nonnegative error variables $e_{i,j}$. The ILP is:

$$
\begin{aligned}
\min \quad & \sum_{i,j} e_{i,j} \\
\text{subject to} \quad & p_{i,r,j} \le b_{i,r} \quad \forall i, r, j \\
& p_{i,r,j} \le c_{r,j} \quad \forall i, r, j \\
& p_{i,r,j} \ge b_{i,r} + c_{r,j} - 1 \quad \forall i, r, j \\
& w_{i,j} \ge p_{i,r,j} \quad \forall i, r, j \\
& w_{i,j} \le \sum_{r} p_{i,r,j} \quad \forall i, j \\
& e_{i,j} \ge A_{i,j} - w_{i,j} \quad \forall i, j \\
& e_{i,j} \ge w_{i,j} - A_{i,j} \quad \forall i, j \\
& b_{i,r}, c_{r,j}, p_{i,r,j}, w_{i,j} \in \{0,1\}, e_{i,j} \in \mathbb{Z}_{\ge 0}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any choice of factor matrices induces the same Boolean product and Hamming error in the ILP. ($\Leftarrow$) Any feasible ILP assignment determines factor matrices B and C, and the linearization forces the objective to equal the Hamming distance between A and $B \cdot C$.

Solution extraction. Output the flattened bits of B followed by the flattened bits of C, discarding the reconstruction auxiliaries. □

Rule 3.140: (Consecutive Block Minimization → Integer Linear Programming) Permute the columns with a one-hot assignment and count row-wise block starts by detecting each 0-to-1 transition after permutation. *Overhead:* `num_vars = num_cols * num_cols + num_rows * num_cols + num_rows * num_cols`, `num_constraints = num_cols + num_cols + num_rows * num_cols + num_rows + num_rows * num_cols + 1`.

Proof: *Construction.* Variables: binary $x_{c,p}$ with $x_{c,p} = 1$ iff column c goes to position p, binary $a_{r,p}$ for the value seen by row r at position p, and binary block-start indicators $b_{r,p}$. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{p} x_{c,p} = 1 \quad \forall c \\
& \sum_{c} x_{c,p} = 1 \quad \forall p \\
& a_{r,p} = \sum_{c} A_{r,c} x_{c,p} \quad \forall r, p \\
& b_{r,0} = a_{r,0} \quad \forall r \\
& b_{r,p} \ge a_{r,p} - a_{r,p-1} \quad \forall r, p > 0 \\
& \sum_{r,p} b_{r,p} \le K \\
& x_{c,p}, a_{r,p}, b_{r,p} \in \{0,1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any column permutation determines exactly one block-start variable for each maximal run of 1s in every row. ($\Leftarrow$) A feasible ILP solution is a column permutation whose counted block starts sum to at most K, which is precisely the source criterion.

Solution extraction. Decode the column permutation from $x_{c,p}$. □

Rule 3.141: (Consecutive Ones Matrix Augmentation → Integer Linear Programming) Choose a column permutation and, for each row, choose the interval that will become its consecutive block of 1s; flips are needed only for zeros inside that interval.

Overhead: `num_vars = num_cols * num_cols + 5 * num_rows * num_cols`, `num_constraints = num_cols + num_cols + num_rows * num_cols + 2 * num_rows + num_rows + 3 * num_rows * num_cols + 4 * num_rows * num_cols + 1`.

Proof: *Construction.* Let the matrix have m rows and n columns, and let $A_{r,c} \in \{0,1\}$ be the given entry. For each row define the constant $\beta_r = 1$ if row r contains at least one 1, and $\beta_r = 0$ otherwise. Use `ILP<bool>` with variable order $(x_{c,p})_{c,p}, (a_{r,p})_{r,p}, (\ell_{r,p})_{r,p}, (u_{r,p})_{r,p}, (h_{r,p})_{r,p}, (f_{r,p})_{r,p}$. The indices are $\mathrm{idx}_{x(c,p)} = cn + p$, $\mathrm{idx}_{a(r,p)} = n^2 + rn + p$, $\mathrm{idx}_{\ell(r,p)} = n^2 + mn + rn + p$, $\mathrm{idx}_{u(r,p)} = n^2 + 2mn + rn + p$, $\mathrm{idx}_{h(r,p)} = n^2 + 3mn + rn + p$, and $\mathrm{idx}_{f(r,p)} = n^2 + 4mn + rn + p$. There are $n^2 + 5mn$ binary variables.

Here $x_{c,p} = 1$ means original column c is placed at position p of the permutation, $a_{r,p}$ is the value seen in row r at permuted position p, $\ell_{r,p}$ and $u_{r,p}$ choose the left and right interval boundaries of row r, $h_{r,p}$ indicates that position p lies inside that chosen interval, and $f_{r,p}$ indicates that row r flips a 0 to a 1 at position p.

The constraints are: $\sum_p x_{c,p} = 1$ for every column c; $\sum_c x_{c,p} = 1$ for every position p; $a_{r,p} = \sum_c A_{r,c} x_{c,p}$ for every row r and position p; $\sum_p \ell_{r,p} = \beta_r$ and $\sum_p u_{r,p} = \beta_r$ for every row r; $\sum_p p\ell_{r,p} \le \sum_p p u_{r,p} + (n-1)(1-\beta_r)$ for every row r, which forces the left boundary not to exceed the right boundary when the row is nonzero; for every row r and position p, $h_{r,p} \le \sum_{q=0}^{p} \ell_{r,q}$, $h_{r,p} \le \sum_{q=p}^{n-1} u_{r,q}$, and $h_{r,p} \ge \sum_{q=0}^{p} \ell_{r,q} + \sum_{q=p}^{n-1} u_{r,q} - 1$; $a_{r,p} \le h_{r,p}$ for every r, p, so every original 1 lies inside the chosen interval; $h_{r,p} \le a_{r,p} + f_{r,p}$, $f_{r,p} \le h_{r,p}$, and $f_{r,p} + a_{r,p} \le 1$ for every r, p, so $f_{r,p} = 1$ exactly when the position lies inside the interval but the original matrix has a 0 there; and the augmentation budget $\sum_{r=0}^{m-1} \sum_{p=0}^{n-1} f_{r,p} \le K$. These are the exact consecutive-ones constraints: after permutation, row r is 1 exactly on the positions with $h_{r,p} = 1$, and the only modifications charged are the zero-to-one flips recorded by f.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_p x_{c,p} = 1 \quad \forall c \\
& \sum_c x_{c,p} = 1 \quad \forall p \\
& a_{r,p} = \sum_c A_{r,c} x_{c,p} \quad \forall r, p \\
& \sum_p \ell_{r,p} = \beta_r, \sum_p u_{r,p} = \beta_r \quad \forall r \\
& \sum_p p\ell_{r,p} \le \sum_p p u_{r,p} + (n-1)(1-\beta_r) \quad \forall r \\
& h_{r,p} \le \sum_{q=0}^{p} \ell_{r,q}, h_{r,p} \le \sum_{q=p}^{n-1} u_{r,q} \quad \forall r, p \\
& h_{r,p} \ge \sum_{q=0}^{p} \ell_{r,q} + \sum_{q=p}^{n-1} u_{r,q} - 1 \quad \forall r, p \\
& a_{r,p} \le h_{r,p}; h_{r,p} \le a_{r,p} + f_{r,p}; f_{r,p} \le h_{r,p}; f_{r,p} + a_{r,p} \le 1 \quad \forall r, p \\
& \sum_{r=0}^{m-1} \sum_{p=0}^{n-1} f_{r,p} \le K \\
& x_{c,p}, a_{r,p}, \ell_{r,p}, u_{r,p}, h_{r,p}, f_{r,p} \in \{0,1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) A feasible augmentation chooses a permutation and flips exactly the zeros lying inside each row's final consecutive-ones interval. ($\Leftarrow$) Any feasible ILP solution yields a permuted matrix whose rows become consecutive-ones after the encoded zero-to-one augmentations, with total augmentation cost at most K.

Solution extraction. Decode the column permutation from $x_{c,p}$ and discard the auxiliary flip variables. □

Rule 3.142: (Consecutive Ones Submatrix → Integer Linear Programming) Select exactly K columns, permute only those selected columns, and require every row to have a single consecutive block within the chosen submatrix.
Overhead: `num_vars = num_cols + num_cols * bound + 5 * num_rows * bound`, `num_constraints = 1 + num_cols + bound + num_rows * bound + 2 * num_rows + num_rows + 3 * num_rows * bound + 4 * num_rows * bound`.

Proof: *Construction.* Variables: binary selection bits s_c, binary placement variables $x_{c,p}$ for selected columns, and row-interval auxiliaries as in ConsecutiveOnesMatrixAugmentation. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_c s_c = K \\ & \sum_p x_{c,p} = s_c \quad \forall c \\ & \sum_c x_{c,p} = 1 \quad \forall p \in \{1, \ldots, K\} \\ & \text{the selected rows satisfy the consecutive-ones interval constraints} \\ & s_c, x_{c,p} \in \{0, 1\} \end{aligned}$$

.

Correctness. (⇒) Any feasible column subset with a valid permutation satisfies the selection and interval constraints. (⇐) Any feasible ILP solution chooses exactly K columns whose induced submatrix admits a consecutive-ones permutation.

Solution extraction. Output the column-selection bits $(s_c)_{c=1}^{n}$ and ignore the permutation auxiliaries. □

Rule 3.143: (Sparse Matrix Compression → Integer Linear Programming) Assign each row one shift value and forbid any pair of shifted 1-entries from colliding in the storage vector.
Overhead: `num_vars = num_rows * bound_k`, `num_constraints = num_rows + num_rows * num_rows * bound_k * bound_k`.

Proof: *Construction.* Variables: binary $x_{r,g}$ with $x_{r,g} = 1$ iff row r uses shift $g \in \{0, \ldots, K-1\}$. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_g x_{r,g} = 1 \quad \forall r \\ & x_{r,g} + x_{s,h} \leq 1 \quad \text{whenever } A_{r,i} = A_{s,j} = 1 \text{ and } i + g = j + h \\ & x_{r,g} \in \{0, 1\} \end{aligned}$$

.

Correctness. (⇒) A valid compression chooses one shift per row and never overlays 1-entries from different rows in the same storage position. (⇐) Any feasible ILP solution gives exactly such a collision-free shift assignment, hence a valid storage vector of length $n + K$.

Solution extraction. For each row r, output the unique zero-based shift g with $x_{r,g} = 1$. □

Rule 3.144: (Shortest Common Supersequence → Integer Linear Programming) Fill the B positions of the supersequence with one-hot symbol variables and match each input string monotonically into those positions.
Overhead: `num_vars = max_length * (alphabet_size + 1) + total_length * max_length`, `num_constraints = max_length + total_length + total_length * max_length + total_length + max_length`.

Proof: *Construction.* Variables: binary $x_{p,a}$ with $x_{p,a} = 1$ iff position p carries symbol a, and binary matching variables $m_{s,j,p}$ saying that the j-th symbol of string s is matched to position p. The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{a} x_{p,a} = 1 \quad \forall p \\
& \sum_{p} m_{s,j,p} = 1 \quad \forall s, j \\
& m_{s,j,p} \le x_{p,a} \quad \forall s, j, p \ \text{ with symbol } \ a \\
& \text{matching positions are strictly increasing in j for every string s} \\
& x_{p,a}, m_{s,j,p} \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any common supersequence of length at most B induces a one-hot symbol assignment and a monotone match of every input string. ($\Leftarrow$) Any feasible ILP solution yields a length-B string into which every source string embeds as a subsequence.

Solution extraction. At each position p, output the unique symbol a with $x_{p,a} = 1$. □

Rule 3.145: (String-to-String Correction → Integer Linear Programming) A time-expanded ILP chooses one edit operation at each of the K stages and tracks the evolving string state until it matches the target. *Overhead:* `num_vars = (bound + 1) * source_length * source_length + (bound + 1) * source_length + 2 * bound * source_length`, `num_constraints = (bound + 1) * source_length * source_length`.

Proof: *Construction.* Let the source length be n, the target length be m, and the operation bound be K. If $m > n$ or $m < n - K$, return an infeasible empty ILP, because the model rejects such instances before any search. Otherwise use `ILP<bool>`. Track the evolving string by the identities of the original source positions, not only by their symbols: token $i \in \{0, ..., n-1\}$ carries symbol x_i from the source string.

The state variables are $z_{t,p,i}$ for $t \in \{0, ..., K\}$, $p \in \{0, ..., n-1\}$, $i \in \{0, ..., n-1\}$, where $z_{t,p,i} = 1$ iff token i occupies position p after step t. Their indices are $\text{idx}_{z(t,p,i)} = tn^2 + pn + i$. The emptiness bits are $e_{t,p}$ with index $\text{idx}_{e(t,p)} = (K+1)n^2 + tn + p$. The operation variables are delete bits $d_{t,j}$ for $t \in \{1, ..., K\}$ and $j \in \{0, ..., n-1\}$ with index $\text{idx}_{d(t,j)} = (K+1)(n^2+n) + (t-1)n + j$; swap bits $s_{t,j}$ for $j \in \{0, ..., n-2\}$ with index $\text{idx}_{s(t,j)} = (K+1)(n^2+n) + Kn + (t-1)(n-1) + j$; and no-op bits ν_t with index $\text{idx}_{\nu(t)} = (K+1)(n^2+n) + Kn + K(n-1) + (t-1)$. There are $(K+1)(n^2+n) + K(2n)$ variables.

The state validity constraints are: $e_{t,p} + \sum_i z_{t,p,i} = 1$ for every t, p; $\sum_p z_{t,p,i} \le 1$ for every t, i; and $e_{t,p} \le e_{t,p+1}$ for every t and $p < n-1$, forcing the active string to occupy a prefix and the deleted positions to form a suffix. The initial state is fixed by $z_{0,p,p} = 1$ for every p, $z_{0,p,i} = 0$ for every $i \neq p$, and $e_{0,p} = 0$ for every position p.

At each step t, choose exactly one operation: $\sum_{j=0}^{n-1} d_{t,j} + \sum_{j=0}^{n-2} s_{t,j} + \nu_t = 1$. Delete at position j is legal only when that current position exists, so $d_{t,j} \le 1 - e_{t-1,j}$. Swap at position j is legal only when both positions j and $j+1$ exist, so $s_{t,j} \le 1 - e_{t-1,j}$ and $s_{t,j} \le 1 - e_{t-1,j+1}$.

The state-update equations are conditioned by exact big-M bounds with $M = 1$, because every left-hand side and every referenced right-hand side is binary. For every token i, step t, and position p: if no-op is chosen, impose $z_{t,p,i} - z_{t-1,p,i} \le 1 - \nu_t$ and $z_{t-1,p,i} - z_{t,p,i} \le 1 - \nu_t$; for every delete position j, if $p < j$ impose $z_{t,p,i} - z_{t-1,p,i} \le 1 - d_{t,j}$ and $z_{t-1,p,i} - z_{t,p,i} \le 1 - d_{t,j}$; if $j \le p < n-1$, impose $z_{t,p,i} - z_{t-1,p+1,i} \le 1 - d_{t,j}$ and $z_{t-1,p+1,i} - z_{t,p,i} \le 1 - d_{t,j}$; and for the last position impose $z_{t,n-1,i} \le 1 - d_{t,j}$; for every swap position j, if $p \notin \{j, j+1\}$ impose $z_{t,p,i} - z_{t-1,p,i} \le 1 - s_{t,j}$ and $z_{t-1,p,i} - z_{t,p,i} \le 1 - s_{t,j}$; if $p = j$, impose $z_{t,j,i} - z_{t-1,j+1,i} \le 1 - s_{t,j}$ and $z_{t-1,j+1,i} - z_{t,j,i} \le 1 - s_{t,j}$; and if $p = j+1$, impose $z_{t,j+1,i} - z_{t-1,j,i} \le 1 - s_{t,j}$ and $z_{t-1,j,i} - z_{t,j+1,i} \le 1 - s_{t,j}$.

Finally force the step-K state to equal the target string: $\sum_{i:x_i=y_p} z_{K,p,i} = 1$ for every target position $p \in \{0, ..., m-1\}$, and $e_{K,p} = 1$ for every $p \in \{m, ..., n-1\}$. This exactly matches the model semantics, which compare the final working string to the target after K operations.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & e_{t,p} + \sum_i z_{t,p,i} = 1 \quad \forall t \in \{0, ..., K\}, p \in \{0, ..., n-1\} \\
& \sum_p z_{t,p,i} \leq 1 \quad \forall t \in \{0, ..., K\}, i \in \{0, ..., n-1\} \\
& e_{t,p} \leq e_{t,p+1} \quad \forall t \in \{0, ..., K\}, p \in \{0, ..., n-2\} \\
& z_{0,p,p} = 1, z_{0,p,i} = 0 \quad \forall i \neq p, e_{0,p} = 0 \quad \forall p \\
& \sum_{j=0}^{n-1} d_{t,j} + \sum_{j=0}^{n-2} s_{t,j} + \nu_t = 1 \quad \forall t \in \{1, ..., K\} \\
& d_{t,j} \leq 1 - e_{t-1,j} \quad \forall t, j \\
& s_{t,j} \leq 1 - e_{t-1,j}, s_{t,j} \leq 1 - e_{t-1,j+1} \quad \forall t, j \\
& \text{the exact M} = 1 \text{ state-update equations enforce no-op, delete, and adjacent-swap transitions} \\
& \sum_{i:x_i=y_p} z_{K,p,i} = 1 \quad \forall p \in \{0, ..., m-1\} \\
& e_{K,p} = 1 \quad \forall p \in \{m, ..., n-1\} \\
& z_{t,p,i}, e_{t,p}, d_{t,j}, s_{t,j}, \nu_t \in \{0, 1\}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any valid length-K edit script yields a feasible sequence of operations and states ending at the target. ($\Leftarrow$) Any feasible ILP solution traces a legal sequence of deletes, adjacent swaps, and no-ops whose final string is the target.

Solution extraction. For each step t, compute the current length $\ell_{t-1} = \sum_{p=0}^{n-1}(1 - e_{t-1,p})$. If $\nu_t = 1$, output the source code $2n$. If $d_{t,j} = 1$, output j. If $s_{t,j} = 1$, output $\ell_{t-1} + j$. This is exactly the encoding used by `evaluate()`: deletions use raw positions, swaps are offset by the current length, and no-op is the distinguished value $2n$. □

Rule 3.146: (Paint Shop → QUBO (real-weighted)) Each car's two occurrences must receive opposite colors, so a single binary variable per car determines the full coloring. Adjacent pairs in the sequence contribute quadratic terms to a QUBO matrix based on their parity (first vs.
second occurrence), and the QUBO minimum plus a constant offset equals the minimum number of color switches [155].
Overhead: `num_vars = num_cars`.

Proof: *Construction.* Given n cars, each appearing exactly twice in a sequence of length $2n$, introduce binary variables $x_1, ..., x_n$ (one per car). Initialize an $n \times n$ upper-triangular matrix Q of zeros. For each adjacent pair of positions $(j, j+1)$ with distinct cars a, b:

- **Same parity** (both first or both second occurrence): a color switch occurs iff $x_a \neq x_b$. The switch indicator is $x_a + x_b - 2x_a x_b$, so add $+1$ to Q_{aa}, $+1$ to Q_{bb}, and -2 to Q_{ab}.
- **Different parity** (one first, one second): a color switch occurs iff $x_a = x_b$. The switch indicator is $1 - x_a - x_b + 2x_a x_b$, so add -1 to Q_{aa}, -1 to Q_{bb}, and $+2$ to Q_{ab}, with a constant offset of $+1$.

Adjacent pairs where both positions are the same car always produce a switch (constant term), and are skipped. The QUBO objective is $\min \boldsymbol{x}^\top Q \boldsymbol{x}$; the minimum number of color switches equals the QUBO minimum plus the total constant offset (number of different-parity pairs plus number of same-car pairs).

Correctness. ($\Rightarrow$) Any PaintShop coloring corresponds to a binary assignment $\boldsymbol{x}$ with the same number of switches (up to the constant offset). ($\Leftarrow$) Any QUBO minimizer $\boldsymbol{x}$ defines a valid coloring (each car's two occurrences get opposite colors), and the offset-adjusted objective equals the switch count. Since the correspondence is bijective and value-preserving, optimality is preserved.

Solution extraction. The QUBO solution $(x_1, ..., x_n)$ maps directly back: car i's first occurrence gets color x_i, second gets $1 - x_i$. □

Example: 4 cars, sequence length 8

Source: PaintShop **Target:** QUBO

```
$ pred create --example PaintShop -o paintshop.json
$ pred reduce paintshop.json --to QUBO/f64 -o bundle.json
$ pred solve bundle.json
$ pred evaluate paintshop.json --config 1,0,0,0
```

Source: $n = 4$ cars, sequence (A, B, C, A, D, B, D, C).
Parity: 1st, 1st, 1st, 2nd, 1st, 2nd, 2nd, 2nd
Step 1 – One QUBO variable per car. Binary variable $x_i \in \{0, 1\}$ for each car i: $x_i = 0$ means "first occurrence gets color 0, second gets color 1"; $x_i = 1$ reverses.
Step 2 – Build the Q matrix from adjacent pairs. For each adjacent pair $(j, j+1)$ in the sequence with distinct cars a, b: if both positions have the *same* parity (both first or both second occurrence), a color switch occurs when $x_a \neq x_b$, contributing $+1$ to Q_{aa}, $+1$ to Q_{bb}, and -2 to Q_{ab}. If they have *different* parity, a switch occurs when $x_a = x_b$, contributing -1 to Q_{aa}, -1 to Q_{bb}, and $+2$ to Q_{ab}.
Step 3 – Verify. The QUBO solution $\boldsymbol{x} = (1, 0, 0, 0)$ yields coloring (1, 0, 0, 0, 0, 1, 1, 1) with 2 color switches ✓

Rule 3.147: (Paint Shop → Integer Linear Programming) One binary variable per car determines its first color, the second occurrence receives the opposite color automatically, and switch indicators count color changes along the sequence.
Overhead: `num_vars = num_cars + 2 * num_sequence`, `num_constraints = num_sequence + 2 * num_sequence`.

Proof: *Construction.* Variables: binary x_i for each car i, binary color variables k_p for sequence positions, and binary switch indicators c_p for positions $p > 0$. The ILP is:

$$\begin{aligned} \min \quad & \sum_p c_p \\ \text{subject to} \quad & k_p = x_i \quad \text{if p is the first occurrence of car i} \\ & k_p = 1 - x_i \quad \text{otherwise} \\ & c_p \geq k_p - k_{p-1} \quad \forall p > 0 \\ & c_p \geq k_{p-1} - k_p \quad \forall p > 0 \\ & x_i, k_p, c_p \in \{0, 1\} \end{aligned}$$

.

Correctness. ($\Rightarrow$) Any first-occurrence coloring determines the whole paint sequence and induces exactly the same number of switches in the ILP. ($\Leftarrow$) Any ILP assignment is already a valid source witness, and the switch variables are forced to count adjacent color changes.

Solution extraction. Output the first-occurrence color bits (x_i). □

Rule 3.148: (Isomorphic Spanning Tree → Integer Linear Programming) A bijection from the tree vertices to the graph vertices is enough: every tree edge must map to a graph edge, which then defines the desired spanning tree.

Overhead: `num_vars = num_vertices * num_vertices, num_constraints = 2 * num_vertices + 2 * (num_vertices + -1 * 1) * num_vertices * num_vertices`.

Proof: *Construction.* Variables: binary $x_{u,v}$ with $x_{u,v} = 1$ iff tree vertex u maps to graph vertex v. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{v} x_{u,v} = 1 \quad \forall u \\ & \sum_{u} x_{u,v} = 1 \quad \forall v \\ & x_{u,v} + x_{w,z} \leq 1 \quad \forall \{u,w\} \in E_{\text{tree}}, \{v,z\} \notin E_{\text{graph}} \\ & x_{u,z} + x_{w,v} \leq 1 \quad \forall \{u,w\} \in E_{\text{tree}}, \{v,z\} \notin E_{\text{graph}} \\ & x_{u,v} \in \{0,1\} \end{aligned}$$

.

Correctness. ($\Rightarrow$) Any isomorphism from the given tree to a spanning tree of the graph satisfies the bijection and non-edge constraints. ($\Leftarrow$) Any feasible ILP solution is a bijection that preserves every tree edge, so the image edges form a spanning tree of the graph isomorphic to the source tree.

Solution extraction. For each tree vertex u, output the unique graph vertex v with $x_{u,v} = 1$. □

Rule 3.149: (Rooted Tree Arrangement → Rooted Tree Storage Assignment) This $O(|E|)$ reduction [39] transforms a graph-embedding arrangement problem into a set-system path-cover problem. Each edge $\{u, v\}$ of the source graph becomes a 2-element required subset $\{u, v\}$ whose elements must lie on a directed path in a rooted tree. The bound adjusts by $K' = K - |E|$, since each edge contributes at least 1 to the arrangement cost but 0 to the extension cost when its endpoints are adjacent in the tree.
Overhead: `universe_size = num_vertices, num_subsets = num_edges`.

Proof: *Construction.* Given a Rooted Tree Arrangement instance with graph $G = (V, E)$ and bound K, construct a Rooted Tree Storage Assignment instance as follows. Set the universe $X = V$ with $|X| = |V|$ elements. For each edge $\{u, v\} \in E$, create a 2-element subset $X_e = \{u, v\}$, yielding a collection $\mathcal{C} = \{X_e : e \in E\}$ of $|E|$ subsets. Set the bound $K' = K - |E|$.

Correctness. ($\Rightarrow$) Suppose there exists a rooted tree T on $|V|$ nodes and a bijection $f : V \to U$ with $\sum_{\{u,v\} \in E} d_{T(f(u),f(v))} \leq K$, where every edge pair lies on a common root-to-leaf path. Using T as the storage tree and the identity embedding (since $X = V$), for each edge $e = \{u, v\}$ the extended subset X'_e consists of all nodes on the path from $f(u)$ to $f(v)$, costing $d_{T(f(u),f(v))} - 1$ additional elements. The total extension cost is $\sum_{e \in E} \left(d_{T(f(u),f(v))} - 1\right) = (\sum d_T) - |E| \leq K - |E| = K'$.

($\Leftarrow$) Suppose there exists a rooted tree $T = (X, A)$ and extended subsets forming directed paths with total extension cost $\leq K'$. The same tree T with the identity mapping $f(v) = v$ gives total arrangement stretch = extension cost + $|E| \leq K' + |E| = K$.

Solution extraction. The target solution is a parent array defining a rooted tree on $X = V$. The source solution is this same parent array concatenated with the identity mapping $f(v) = v$ for all $v \in V$. □

Example: Path graph P_4 ($n = 4$, $|E| = 3$, $K = 5$)

Source: RootedTreeArrangement **Target:** RootedTreeStorageAssignment

```
$ pred create --example RootedTreeArrangement/SimpleGraph -o rta.json
$ pred reduce rta.json --to RootedTreeStorageAssignment -o bundle.json
$ pred solve bundle.json
$ pred evaluate rta.json --config 0,0,1,2,0,1,2,3
```

> Source: path graph P_4 with vertices $\{0,1,2,3\}$, edges $\{0,1\},\{1,2\},\{2,3\}$, and bound $K=5$.
> Target: universe $X=\{0,1,2,3\}$, subsets $\{0,1\},\{1,2\},\{2,3\}$, bound $K'=5-3=2$.
> The chain tree $0\to1\to2\to3$ (parent array $(0,0,1,2)$) with identity mapping gives total stretch $1+1+1=3\le5$ in the source. In the target, every edge subset is already a parent-child pair, so extension cost is $0+0+0=0\le2$ ✓

Rule 3.150: (Rooted Tree Storage Assignment → Integer Linear Programming) Choose one parent for each non-root element, enforce acyclicity with depth variables, and linearize the path-extension cost of every subset by selecting its top and bottom vertices in the rooted tree.
Overhead: `num_vars = universe_size * universe_size * universe_size + 2 * universe_size * universe_size + universe_size + num_subsets * (universe_size * universe_size + 2 * universe_size + 3)`, `num_constraints = universe_size * universe_size * universe_size + universe_size * universe_size + universe_size * universe_size + num_subsets * universe_size * universe_size`.

Proof: *Construction.* Let $X=\{0,\dots,n-1\}$ and let the subset family be $\mathcal{C}=\{S_0,\dots,S_{m-1}\}$. For every subset of size 0 or 1 the model charges extension cost 0 automatically, so only the nontrivial subsets matter. Enumerate them as $I=\{k_0<\dots<k_{r-1}\}=\{k:|S_k|\ge2\}$. Use `ILP<i32>`. The variable blocks are: parent indicators $p_{v,u}\in\{0,1\}$ for all $v,u\in X$; depths $d_v\in\{0,\dots,n-1\}$; ancestor indicators $a_{u,v}\in\{0,1\}$, where $a_{u,v}=1$ means u is an ancestor of v (allowing $u=v$); auxiliary transitive-closure variables $h_{u,v,w}\in\{0,1\}$; and, for each nontrivial subset gadget $s\in\{0,\dots,r-1\}$ corresponding to original subset S_{k_s}, top selectors $t_{s,u}$, bottom selectors $b_{s,v}$, pair selectors $m_{s,u,v}$, endpoint depths T_s,B_s, and extension cost c_s.

The indices are $\mathrm{idx}_{p(v,u)}=vn+u$, $\mathrm{idx}_{d(v)}=n^2+v$, $\mathrm{idx}_{a(u,v)}=n^2+n+un+v$, $\mathrm{idx}_{h(u,v,w)}=2n^2+n+(un+v)n+w$, $\mathrm{idx}_{t(s,u)}=n^3+2n^2+n+sn+u$, $\mathrm{idx}_{b(s,v)}=n^3+2n^2+n+rn+sn+v$, $\mathrm{idx}_{m(s,u,v)}=n^3+2n^2+n+2rn+sn^2+un+v$, $\mathrm{idx}_{T(s)}=n^3+2n^2+n+2rn+rn^2+s$, $\mathrm{idx}_{B(s)}=n^3+2n^2+n+2rn+rn^2+r+s$, and $\mathrm{idx}_{c(s)}=n^3+2n^2+n+2rn+rn^2+2r+s$. The total number of variables is $n^3+2n^2+n+r(n^2+2n+3)$.

The rooted-tree constraints are: $\sum_{u=0}^{n-1}p_{v,u}=1$ for every vertex v; $\sum_v p_{v,v}=1$, so exactly one vertex chooses itself as parent and becomes the root; $d_v\le(n-1)\big(1-p_{v,v}\big)$ for every v, hence the unique root has depth 0; and for every ordered pair $u\ne v$, $d_v-d_u\ge1-n\big(1-p_{v,u}\big)$ and $d_v-d_u\le1+n\big(1-p_{v,u}\big)$. The exact big-$M$ here is $M=n$: the expression d_v-d_u-1 ranges from $-n$ to $n-2$ when both depths lie in $\{0,\dots,n-1\}$.

The ancestor relation is defined explicitly by $a_{v,v}=1$ for every v, $h_{u,v,v}=0$ for all u,v, and, for every $u\ne v$, $a_{u,v}=\sum_{w=0}^{n-1}h_{u,v,w}$. The helper variables linearize the recursion "u is an ancestor of v iff u is an ancestor of the unique parent of v": $h_{u,v,w}\le p_{v,w}$, $h_{u,v,w}\le a_{u,w}$, and $h_{u,v,w}\ge p_{v,w}+a_{u,w}-1$ for all u,v,w with $w\ne v$.

For each nontrivial subset gadget s corresponding to S_{k_s}, choose path endpoints only from the subset itself: $\sum_{u\in S_{k_s}}t_{s,u}=1$, $t_{s,u}=0$ for $u\notin S_{k_s}$, $\sum_{v\in S_{k_s}}b_{s,v}=1$, $b_{s,v}=0$ for $v\notin S_{k_s}$. Linearize the chosen ordered endpoint pair by $m_{s,u,v}\le t_{s,u}$, $m_{s,u,v}\le b_{s,v}$, $m_{s,u,v}\ge t_{s,u}+b_{s,v}-1$. Because exactly one top and one bottom are chosen, exactly one pair selector is 1.

The path condition for subset S_{k_s} is then fully explicit: $m_{s,u,v}\le a_{u,v}$ for every u,v, so the chosen top is an ancestor of the chosen bottom; and for every element $w\in S_{k_s}$, $m_{s,u,v}\le a_{u,w}$ and $m_{s,u,v}\le a_{w,v}$ for all u,v. Thus every subset element lies on the ancestor chain from the chosen top to the chosen bottom.

Bind the endpoint depths to the chosen selectors by exact big-M constraints with $M=n-1$: $T_s-d_u\le(n-1)\big(1-t_{s,u}\big)$ and $d_u-T_s\le(n-1)\big(1-t_{s,u}\big)$ for every u; $B_s-d_v\le(n-1)\big(1-b_{s,v}\big)$ and $d_v-B_s\le(n-1)\big(1-b_{s,v}\big)$ for every v. Finally set the extension cost of the subset to the exact path surplus $c_s=B_s-T_s+1-|S_{k_s}|$, require $c_s\ge0$, and bound the total cost by $\sum_{s=0}^{r-1}c_s\le K$. This matches the model's `subset_extension_cost()`: the top and bottom are the shallowest and deepest members of the subset on the chosen chain, and the path contributes exactly the interior vertices not already present in the subset.

The ILP is:

$$
\begin{aligned}
\text{find} \quad & \boldsymbol{x} \\
\text{subject to} \quad & \sum_{u=0}^{n-1} p_{v,u} = 1 \quad \forall v \in X \\
& \sum_{v} p_{v,v} = 1, d_v \le (n-1)\big(1 - p_{v,v}\big) \quad \forall v \in X \\
& d_v - d_u \ge 1 - n\big(1 - p_{v,u}\big), d_v - d_u \le 1 + n\big(1 - p_{v,u}\big) \quad \forall u \ne v \\
& a_{v,v} = 1, h_{u,v,v} = 0, a_{u,v} = \sum_{w=0}^{n-1} h_{u,v,w} \quad \forall u, v \in X \\
& h_{u,v,w} \le p_{v,w}, h_{u,v,w} \le a_{u,w}, h_{u,v,w} \ge p_{v,w} + a_{u,w} - 1 \quad \forall u, v, w \in X \text{ with } w \ne v \\
& \sum_{u \in S_{k_s}} t_{s,u} = 1, t_{s,u} = 0 \quad \forall u \notin S_{k_s}, s \\
& \sum_{v \in S_{k_s}} b_{s,v} = 1, b_{s,v} = 0 \quad \forall v \notin S_{k_s}, s \\
& m_{s,u,v} \le t_{s,u}, m_{s,u,v} \le b_{s,v}, m_{s,u,v} \ge t_{s,u} + b_{s,v} - 1 \quad \forall s, u, v \\
& m_{s,u,v} \le a_{u,v}, m_{s,u,v} \le a_{u,w}, m_{s,u,v} \le a_{w,v} \quad \forall s, u, v, w \in S_{k_s} \\
& T_s - d_u \le (n-1)\big(1 - t_{s,u}\big), d_u - T_s \le (n-1)\big(1 - t_{s,u}\big) \quad \forall s, u \\
& B_s - d_v \le (n-1)\big(1 - b_{s,v}\big), d_v - B_s \le (n-1)\big(1 - b_{s,v}\big) \quad \forall s, v \\
& c_s = B_s - T_s + 1 - |S_{k_s}|, c_s \ge 0 \quad \forall s; \sum_s c_s \le K \\
& p_{v,u}, a_{u,v}, h_{u,v,w}, t_{s,u}, b_{s,v}, m_{s,u,v} \in \{0, 1\} \\
& d_v, T_s, B_s, c_s \in \mathbb{Z}_{\ge 0}
\end{aligned}
$$

.

Correctness. ($\Rightarrow$) Any rooted tree satisfying all subset-path conditions induces parent, depth, and path-endpoint variables with the same total extension cost. ($\Leftarrow$) Any feasible ILP solution defines a rooted tree in which every subset lies on one ancestor chain, and the encoded path lengths keep the total extension cost within the bound.

Solution extraction. For each vertex v, output its unique parent u with $p_{v,u} = 1$. □

Rule 3.151: (Minimum Edge-Cost Flow → Integer Linear Programming) Introduce integer flow variables and binary arc-activation indicators, link them so that an indicator is forced to 1 whenever the corresponding arc carries positive flow, and minimize the total price of activated arcs.
Overhead: `num_vars = 2 * num_edges`, `num_constraints = 2 * num_edges + num_vertices + -1 * 1`.

Proof: *Construction.* Let $m = |A|$ and $n = |V|$. Use `ILP<i32>` with $2m$ variables: integer flow variables $f_a \in \{0, ..., c(a)\}$ for $a \in \{0, ..., m-1\}$ and binary activation indicators $y_a \in \{0, 1\}$ for $a \in \{m, ..., 2m-1\}$.

Constraints:
- *Linking:* $f_a - c(a) \cdot y_a \le 0$ for each arc a — forces $y_a = 1$ when $f_a > 0$ (m constraints).
- *Binary bound:* $y_a \le 1$ for each arc a (m constraints).
- *Flow conservation:* $\sum_{a \text{ entering } v} f_a - \sum_{a \text{ leaving } v} f_a = 0$ for each non-terminal vertex $v \in V \setminus \{s, t\}$ ($n - 2$ constraints).
- *Flow requirement:* $\sum_{a \text{ entering } t} f_a - \sum_{a \text{ leaving } t} f_a \ge R$ (1 constraint).

Objective: $\min \sum_{a=0}^{m-1} p(a) \cdot y_a$.

Total: $2m + n - 1$ constraints, $2m$ variables.

Correctness. ($\Rightarrow$) Any feasible flow of value $\ge R$ determines the flow variables directly, and the linking constraints force $y_a = 1$ for every arc with positive flow, so the ILP objective equals the edge cost. ($\Leftarrow$)

Any feasible ILP solution has $f_a \le c(a)y_a \le c(a)$ and satisfies conservation and the flow requirement. The objective $\sum p(a)y_a$ is at least the edge cost of the flow because $y_a \ge 1$ whenever $f_a > 0$.

Solution extraction. Output the first m variables $(f_0, ..., f_{m-1})$ as the flow assignment. □

Rule 3.152: (Minimum Feedback Arc Set (weighted) → Maximum Likelihood Ranking) This $O(n^2)$ reduction [5] applies to unit-weight feedback arc set instances. It keeps the same vertex set as ranking items and encodes each unordered pair by a skew-symmetric entry in $\{-1, 0, 1\}$ with comparison count $c = 0$.
Overhead: `num_items = num_vertices`.

Proof: *Construction.* Given a unit-weight Minimum Feedback Arc Set instance $(G = (V, A), \mathbf{1})$ with $V = \{0, ..., n-1\}$, construct the matrix $M \in \mathbb{Z}^{n \times n}$ by setting $M_{ii} = 0$ and, for every distinct pair i, j,

$$M_{ij} = \begin{cases} 1 & \text{if } (i \to j) \in A \wedge (j \to i) \neg \in A \\ -1 & \text{if } (j \to i) \in A \wedge (i \to j) \neg \in A. \\ 0 & \text{otherwise} \end{cases}$$

Then $M_{ij} + M_{ji} = 0$ for all $i \neq j$, so the target is a valid Maximum Likelihood Ranking instance with n items.

Correctness. ($\Rightarrow$) Let π be any ranking and let $B(\pi) = \{(u \to v) \in A : \pi(u) > \pi(v)\}$ be its backward arcs. Removing $B(\pi)$ leaves only forward arcs, hence a DAG, so $B(\pi)$ is a feedback arc set. Partition unordered vertex pairs into one-directional pairs A_1 and bidirectional pairs A_2. Every one-directional backward arc contributes $+1$ to the MLR objective, every one-directional forward arc contributes -1, and bidirectional or absent pairs contribute 0. Therefore

$$\text{cost}(\pi) = 2\,|B(\pi)| - (|A_1| + 2|A_2|) = 2\,|B(\pi)| - |A|.$$

The target objective is thus the source objective shifted by the constant $-|A|$, so minimizing disagreement cost minimizes feedback arc set size. ($\Leftarrow$) Let $F \subseteq A$ be a minimum feedback arc set, and take a topological order π of the DAG $G - F$. Every arc in $A \setminus F$ is forward in π, hence every backward arc under π lies in F, so $B(\pi) \subseteq F$. Since $B(\pi)$ is itself a feedback arc set by the previous argument, minimality of F forces $|B(\pi)| = |F|$. Therefore an optimal source solution yields an optimal target ranking.

Solution extraction. Given the target rank vector, output one source bit per source arc $(u \to v)$ in source-arc order: set the bit to 1 iff item u is ranked after item v, and to 0 otherwise. □

Example: 5-vertex digraph ($n = 5$, $|A| = 7$, unit weights) mapped to a skew-symmetric ranking matrix

Source: MinimumFeedbackArcSet **Target:** MaximumLikelihoodRanking

```
$ pred create --example MinimumFeedbackArcSet/i32 -o mfas.json
$ pred reduce mfas.json --to MaximumLikelihoodRanking -o bundle.json
$ pred solve bundle.json
$ pred evaluate mfas.json --config 0,0,1,0,0,1,0
```

Step 1 – Source instance. The source digraph has vertices {0, 1, 2, 3, 4} and arcs $(0 \to 1)$, $(1 \to 2)$, $(2 \to 0)$, $(2 \to 3)$, $(3 \to 4)$, $(4 \to 2)$, $(0 \to 4)$, all with unit weight. The extracted optimal feedback arc set removes $(2 \to 0)$ and $(4 \to 2)$, so $|F| = 2$.

Step 2 – Build the comparison matrix. The reduction keeps the same item set and writes $M_{ij} = 1$ when only $i \to j$ exists, $M_{ij} = -1$ when only $j \to i$ exists, and $M_{ij} = 0$ otherwise. For this instance,

$$M = (0,\ 1,\ -1,\ 0,\ 1;\ -1,\ 0,\ 1,\ 0,\ 0;\ 1,\ -1,\ 0,\ 1,\ -1;\ 0,\ 0,\ -1,\ 0,\ 1;\ -1,\ 0,\ 1,\ -1,\ 0)$$

. Every off-diagonal pair sums to 0, so the target is a valid Maximum Likelihood Ranking instance with $c = 0$.

Step 3 – Verify a solution. The stored ranking vector is (0, 1, 2, 3, 4), interpreted as the map from items to ranks. The target disagreement cost is $-3 = 2 \cdot 2 - 7$, and the extracted source witness is exactly the backward-arc set $(2 \to 0)$ and $(4 \to 2)$ ✓

Multiplicity: The fixture stores one canonical optimum. Other optimal rankings exist because the DAG obtained after removing the two backward arcs has multiple valid topological orders.

Rule 3.153: (Maximum Likelihood Ranking → Integer Linear Programming) Each pair of items (i, j) with $i < j$ gets a binary variable x_{ij} indicating whether i is ranked before j. Transitivity constraints enforce a valid linear order, and the objective minimizes the total disagreement cost.
Overhead: `num_vars = num_items * (num_items + -1 * 1) * 2^-1`, `num_constraints = num_items * (num_items + -1 * 1) * (num_items + -1 * 2) * 3^-1`.

Proof: *Construction.* Given n items and comparison matrix A, introduce $\frac{n(n-1)}{2}$ binary variables x_{ij} for $i < j$. For each triple $\{a, b, c\}$ with $a < b < c$, add two transitivity constraints:

$$x_{ab} + x_{bc} - x_{ac} \leq 1$$
$$-x_{ab} - x_{bc} + x_{ac} \leq 0$$

The objective is:

$$\min \sum_{i<j} (a_{ji} - a_{ij}) \cdot x_{ij}$$

This yields $\frac{n(n-1)}{2}$ variables and $n(n-1)\frac{n-2}{3}$ constraints.

Correctness. ($\Rightarrow$) Any permutation π defines a consistent tournament: set $x_{ij} = 1$ iff i appears before j in π. The transitivity constraints are satisfied because a linear order has no directed cycles. The ILP objective equals the disagreement cost of π. ($\Leftarrow$) Any feasible binary solution defines a transitive tournament (an acyclic tournament), which corresponds to a unique linear order. The objective equals the disagreement cost of that order.

Solution extraction. For each item i, count the number of items ranked before it: $\text{rank}(i) = \sum_{j<i} x_{ji} + \sum_{j>i} (1 - x_{ij})$. The resulting rank vector is the permutation. □

Example: $n = 3$ items, 3 pairwise variables, 2 transitivity constraints

Source: MaximumLikelihoodRanking **Target:** ILP

```
$ pred create --example MaximumLikelihoodRanking -o mlr.json
$ pred reduce mlr.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate mlr.json --config 0,1,2
```

Step 1 – Source instance. A ranking instance with $n = 3$ items and comparison matrix A.

Step 2 – Build the ILP. Introduce $\binom{3}{2} = 3$ binary variables x_{ij} for each pair $i < j$. Add 2 transitivity constraints from all $\binom{3}{3}$ triples. The ILP has 3 variables and 2 constraints.

Step 3 – Verify. The ILP optimum extracts to ranking (0, 1, 2), which matches the source optimum ✓.

Rule 3.154: (Optimum Communication Spanning Tree → Integer Linear Programming) Binary edge selectors determine the spanning tree. Multi-commodity flow variables route one unit of flow for each vertex pair with positive requirement, and the objective minimizes the total weighted communication cost.
Overhead: `num_vars = num_edges + 2 * num_edges * num_vertices * (num_vertices + -1 * 1) * 2^-1`, `num_constraints = 1 + num_vertices * num_vertices * (num_vertices + -1 * 1) * 2^-1 + 2 * num_edges * num_vertices * (num_vertices + -1 * 1) * 2^-1`.

Proof: *Construction.* Given K_n with weights $w(e)$ and requirements $r(u,v)$, let $m = \frac{n(n-1)}{2}$. Introduce binary edge variables x_e for each edge. For each pair (s,t) with $r(s,t) > 0$, add directed flow variables $f^{st}_{e,\text{fwd}}$ and $f^{st}_{e,\text{bwd}}$ for each edge e.

Constraints:

- Tree size: $\sum_e x_e = n - 1$
- Flow conservation: for each commodity (s,t) and each vertex v, net inflow equals -1 at source s, $+1$ at sink t, and 0 elsewhere.
- Capacity linking: $f^{st}_{e,\text{dir}} \leq x_e$ for each commodity and direction.

Objective: $\min \sum_{(s,t):r>0} r(s,t) \cdot \sum_e w(e) \cdot \left(f^{st}_{e,\text{fwd}} + f^{st}_{e,\text{bwd}}\right)$

Correctness. ($\Rightarrow$) Any spanning tree defines edge selectors and unique path flows. The objective equals the communication cost. ($\Leftarrow$) Any feasible ILP solution with $n-1$ selected edges and valid flows corresponds to a connected spanning subgraph (tree), and the flow cost equals the path-weighted communication cost.

Solution extraction. Output the first m variables $(x_0, ..., x_{m-1})$ as the edge selection. □

Example: K_3, 21 variables, 28 constraints

Source: OptimumCommunicationSpanningTree **Target:** ILP

```
$ pred create --example OptimumCommunicationSpanningTree -o ocst.json
$ pred reduce ocst.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate ocst.json --config 1,1,0
```

Step 1 – Source instance. K_3 with edge weight and requirement matrices.

Step 2 – Build the ILP. Introduce edge selectors and multi-commodity flow variables. The ILP has 21 variables and 28 constraints.

Step 3 – Verify. The ILP optimum extracts to edge selection (1, 1, 0), which matches the source optimum ✓.

3.5 Unit Disk Mapping

Rule 3.155: (Maximum Independent Set → King's Subgraph MIS) [156] The key idea is to represent each vertex of a general graph as a chain of grid nodes (a "copy line") on a King's subgraph, where adjacency is limited to unit-distance neighbors. Edges between vertices in the original graph are encoded by crossing gadgets: when two copy lines cross, the gadget ensures that at most one can be fully selected, mimicking the independence constraint. The overhead from the copy-line structure is a known constant Δ, so $\text{MIS}(G_{\text{grid}}) = \text{MIS}(G) + \Delta$, and the reduction preserves optimality with at most quadratic blowup.

Proof: *Construction (Copy-Line Method).* Given $G = (V, E)$ with $n = |V|$:

1. *Vertex ordering:* Compute a path decomposition of G to obtain vertex order $(v_1, ..., v_n)$. The pathwidth determines the grid height.
2. *Copy lines:* For each vertex v_i, create an L-shaped "copy line" on the grid:

$$\text{CopyLine}(v_i) = \{(r, c_i) : r \in [r_{\text{start}}, r_{\text{stop}}]\} \cup \{(r_i, c) : c \in [c_i, c_{\text{stop}}]\}$$

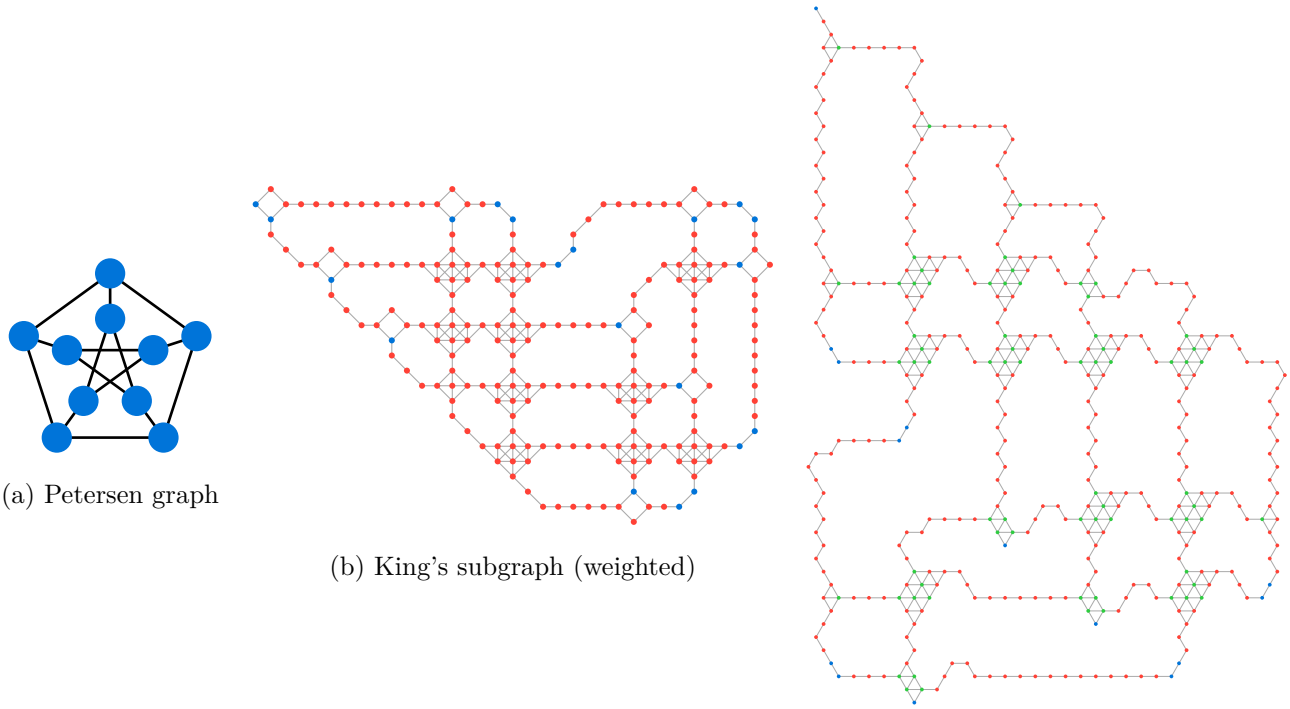
   where positions are determined by the vertex order and edge structure.
3. *Crossing gadgets:* When two copy lines cross (corresponding to an edge $(v_i, v_j) \in E$), insert a crossing gadget that enforces: at most one of the two lines can be "active" (all vertices selected).
4. *MIS correspondence:* Each copy line has MIS contribution $\approx |\text{line}|/2$. The gadgets add overhead Δ such that:

$$\mathrm{MIS}(G_{\text{grid}}) = \mathrm{MIS}(G) + \Delta$$

Correctness. ($\Rightarrow$) An IS S in G maps to a grid IS by activating copy lines for vertices in S (selecting alternating grid nodes) and deactivating lines for vertices not in S. At each crossing gadget between adjacent vertices $v_i, v_j \in S$, at most one line is active, but since v_i and v_j are not both in S (they are independent), no conflict arises. ($\Leftarrow$) A grid MIS determines which copy lines are active (majority of nodes selected). Active lines correspond to an IS in G: if two adjacent vertices v_i, v_j were both active, their crossing gadget would prevent both from contributing fully, contradicting optimality.

Solution extraction. For each copy line, check if the majority of its vertices are in the grid MIS. Map back: $v_i \in S$ iff copy line i is active. □

Example: Petersen Graph.[66] The Petersen graph ($n = 10$, MIS$= 4$) maps to a 30×42 King's subgraph with 219 nodes and overhead $\Delta = 89$. Solving MIS on the grid yields $\mathrm{MIS}(G_{\text{grid}}) = 4 + 89 = 93$. The weighted and unweighted KSG mappings share identical grid topology (same node positions and edges); only the vertex weights differ. With triangular lattice encoding [156], the same graph maps to a 42×60 grid with 395 nodes and overhead $\Delta = 375$, giving $\mathrm{MIS}(G_{\text{tri}}) = 4 + 375 = 379$.



(a) Petersen graph

(b) King's subgraph (weighted)

(c) Triangular lattice (weighted)

Figure 94: Unit disk mappings of the Petersen graph. Blue: weight 1, red: weight 2, green: weight 3.

Rule 3.156: (Maximum Independent Set → Triangular Subgraph MIS) [156] The same copy-line principle as the King's subgraph reduction applies, but on a triangular lattice. The triangular geometry offers a denser packing of neighbors (each node has 6 neighbors vs. 8 in the King's grid), which requires redesigned crossing and simplifier gadgets but preserves the same asymptotic overhead. The resulting graph is a unit disk graph under the triangular metric, suitable for hardware architectures based on triangular lattice connectivity.

Proof: *Construction.* Same copy-line method as the KSG mapping: vertex ordering via path decomposition, L-shaped copy lines, and crossing gadgets at edge intersections. The gadgets are adapted for the triangular lattice geometry, where adjacency is defined by unit distance under the triangular metric (6 neighbors per interior node instead of 8).

[66] Generated using `cargo run --example export_petersen_mapping` from the accompanying code repository.

Correctness. ($\Rightarrow$) An IS in G maps to an IS on the triangular grid by the same copy-line activation mechanism. ($\Leftarrow$) A grid MIS maps back to an IS by the copy-line activity rule, with the adapted crossing gadgets enforcing the same independence constraints.

Solution extraction. Same as the KSG mapping: determine copy-line activity by majority vote, then map back to the original graph.

Overhead. Both vertex and edge counts grow as $O(n^2)$ where $n = |V|$, matching the KSG mapping. □

Weighted Extension. For MWIS, copy lines use weighted vertices (weights 1, 2, or 3). Source weights < 1 are added to designated "pin" vertices.

QUBO Mapping. A QUBO problem $\min \boldsymbol{x}^\top Q \boldsymbol{x}$ maps to weighted MIS on a grid by:
1. Creating copy lines for each variable
2. Using XOR gadgets for couplings: $x_{\text{out}} = \neg(x_1 \oplus x_2)$
3. Adding weights for linear and quadratic terms

See `export_petersen_mapping.rs`.

Warning: Missing reduction rules:
- Clustering → ILP
- DecisionMinimumVertexCover → HamiltonianCircuit
- ExactCoverBy3Sets → AlgebraicEquationsOverGF2
- ExactCoverBy3Sets → MaximumSetPacking
- ExactCoverBy3Sets → MinimumAxiomSet
- ExactCoverBy3Sets → MinimumFaultDetectionTestSet
- ExactCoverBy3Sets → MinimumWeightSolutionToLinearEquations
- ExactCoverBy3Sets → StaffScheduling
- ExactCoverBy3Sets → SubsetProduct
- FeasibleRegisterAssignment → ILP
- GraphPartitioning → ILP
- GraphPartitioning → MaxCut
- GraphPartitioning → QUBO
- HamiltonianCircuit → BiconnectivityAugmentation
- HamiltonianCircuit → BottleneckTravelingSalesman

- HamiltonianCircuit → HamiltonianPath
- HamiltonianCircuit → QuadraticAssignment
- HamiltonianCircuit → RuralPostman
- HamiltonianCircuit → StackerCrane
- HamiltonianCircuit → StrongConnectivityAugmentation
- HamiltonianPath → ConsecutiveOnesSubmatrix
- HamiltonianPath → DegreeConstrainedSpanningTree
- HamiltonianPath → IsomorphicSpanningTree
- HamiltonianPathBetweenTwoVertices → LongestPath
- ILP → ILP
- KClique → ConjunctiveBooleanQuery
- KClique → SubgraphIsomorphism
- KColoring → Clustering
- KColoring → KColoring
- KColoring → PartitionIntoCliques
- KSatisfiability → KSatisfiability
- KSatisfiability → AcyclicPartition
- KSatisfiability → CyclicOrdering
- KSatisfiability → DecisionMinimumVertexCover
- KSatisfiability → DirectedTwoCommodityIntegralFlow
- KSatisfiability → FeasibleRegisterAssignment

- KSatisfiability → KClique
- KSatisfiability → Kernel
- KSatisfiability → MinimumVertexCover
- KSatisfiability → MonochromaticTriangle
- KSatisfiability → OneInThreeSatisfiability
- KSatisfiability → PreemptiveScheduling
- KSatisfiability → RegisterSufficiency
- KSatisfiability → SimultaneousIncongruences
- KSatisfiability → TimetableDesign
- MaxCut → MinimumCutIntoBoundedSets
- MaximumDomaticNumber → ILP
- MaximumIndependentSet → MaximumIndependentSet
- MaximumIndependentSet → IntegralFlowBundles
- MaximumSetPacking → MaximumSetPacking
- MinimumCapacitatedSpanningTree → ILP
- MinimumGraphBandwidth → ILP
- MinimumMatrixCover → ILP
- MinimumMetricDimension → ILP
- MinimumVertexCover → MinimumFeedbackArcSet
- NAESatisfiability → MaxCut
- NAESatisfiability → PartitionIntoPerfectMatchings
- NAESatisfiability → SetSplitting

- Partition → BinPacking
- Partition → MultiprocessorScheduling
- Partition → OpenShopScheduling
- Partition → ProductionPlanning
- Partition → SequencingToMinimizeTardyTaskWeight
- Partition → SequencingWithinIntervals
- Partition → ShortestWeightConstrainedPath
- PartitionIntoCliques → MinimumCoveringByCliques
- RegisterSufficiency → ILP
- Satisfiability → NAESatisfiability
- Satisfiability → NonTautology
- SpinGlass → SpinGlass
- SubsetSum → IntegerExpressionMembership
- SubsetSum → IntegerKnapsack
- SubsetSum → Partition
- ThreeDimensionalMatching → ILP
- ThreeDimensionalMatching → ThreePartition
- ThreePartition → FlowShopScheduling
- ThreePartition → JobShopScheduling
- ThreePartition → ResourceConstrainedScheduling
- ThreePartition → SequencingToMinimizeWeightedTardiness

- ThreePartition → SequencingWithReleaseTimesAndDeadlines

3.6 Resource Estimation from Examples

The following table shows concrete variable overhead for example instances, taken directly from the canonical fixture examples.

Definition 3.1 (Maximum Domatic Number): Given an undirected graph $G = (V, E)$, find the maximum k such that V can be partitioned into k disjoint dominating sets $V_1, ..., V_k$ where each V_i dominates all of V.

- Complexity: `2.695^num_vertices`.
- Reduces to: ILP.

MaximumDomaticNumber	
graph	The underlying graph G=(V,E)

Maximum Domatic Number (GT3) [5]. NP-complete for any fixed $k \geq 3$ (Garey, Johnson, Tarjan 1976). Polynomial for $k = 2$.

The best known exact algorithm runs in $O^*(2.695^n)$ (Riege, Rothe, Spakowski, Yamamoto 2007).

Rule 3.157: (Maximum Domatic Number → Integer Linear Programming) Binary assignment variables $x_{v,i} \in \{0, 1\}$ for each vertex v and set index $i \in \{1, ..., n\}$, plus binary usage indicators y_i. Partition constraints, domination constraints, and linking constraints enforce a valid domatic partition. Maximize $\sum_i y_i$.
Overhead: `num_vars = num_vertices * num_vertices + num_vertices`, `num_constraints = num_vertices + num_vertices * num_vertices + num_vertices * num_vertices`.

Proof: *Construction.* Introduce $n^2 + n$ binary variables. For each vertex v, $\sum_i x_{v,i} = 1$ (partition). For each v and i, $x_{v,i} + \sum_{u \in N(v)} x_{u,i} \geq y_i$ (domination). For each i, $y_i \leq \sum_v x_{v,i}$ (linking). Maximize $\sum_i y_i$.

Correctness. ($\Rightarrow$) Any valid domatic partition gives a feasible ILP assignment with the same number of non-empty sets. ($\Leftarrow$) Any feasible ILP solution encodes a valid domatic partition.

Solution extraction. For each vertex v, find i with $x_{v,i} = 1$; set $\text{config}[v] = i$. □

Definition 3.2 (Minimum Metric Dimension): Given an undirected graph $G = (V, E)$, find a minimum-size resolving set $V' \subseteq V$ such that for every pair of distinct vertices $u, v \in V$, there exists $w \in V'$ with $d(u, w) \neq d(v, w)$.

- Complexity: `2^num_vertices`.
- Reduces to: ILP.

MinimumMetricDimension	
graph	The underlying graph G=(V,E)

Minimum Metric Dimension (GT61) [5]. NP-complete via reduction from 3-Dimensional Matching. Polynomial for trees.

The best known exact algorithm is brute-force $O^*(2^n)$.

Rule 3.158: (Minimum Metric Dimension → Integer Linear Programming) Binary variables $z_v \in \{0, 1\}$ for each vertex. Precompute all-pairs shortest-path distances. For each pair (u, v) with $u < v$, add constraint $\sum_{w: d(u,w) \neq d(v,w)} z_w \geq 1$. Minimize $\sum_v z_v$.
Overhead: `num_vars = num_vertices`, `num_constraints = num_vertices * (num_vertices + -1 * 1) * 2^-1`.

Proof: *Construction.* n binary variables and $\frac{n(n-1)}{2}$ pair-distinguishing constraints from BFS distances.

Correctness. ($\Rightarrow$) A resolving set satisfies all pair constraints. ($\Leftarrow$) Any feasible solution is a resolving set since every pair is distinguished.

Solution extraction. $z_v = 1 \to \text{config}[v] = 1$. □

Definition 3.3 (Minimum Graph Bandwidth): Given an undirected graph $G = (V, E)$, find a one-to-one mapping $f : V \to \{0, 1, ..., |V| - 1\}$ that minimizes $\max_{\{u,v\} \in E} |f(u) - f(v)|$.
- Complexity: `factorial(num_vertices)`.
- Reduces to: ILP.

MinimumGraphBandwidth	
graph	The undirected graph G=(V,E)

Graph Bandwidth (GT40) [5]. NP-complete even for trees with maximum degree 3. The brute-force bound is $O^*(n!)$ over all permutations.

Rule 3.159: (Minimum Graph Bandwidth → Integer Linear Programming) Assignment variables $x_{v,p} \in \{0, 1\}$ for vertex v and position p, integer position variables, and bandwidth variable B. Bijection constraints enforce a valid permutation; edge-stretch constraints enforce $|\text{pos}(u) - \text{pos}(v)| \leq B$. Minimize B.
Overhead: `num_vars = num_vertices^2 + num_vertices + 1`, `num_constraints = 2 * num_vertices + num_vertices^2 + num_vertices + num_vertices + 1 + 2 * num_edges`.

Proof: *Construction.* $n^2 + n + 1$ variables (assignment + position + bandwidth). Bijection: $\sum_p x_{v,p} = 1$ and $\sum_v x_{v,p} = 1$. Position linking and edge-stretch constraints bound the bandwidth.

Correctness. ($\Rightarrow$) Any valid permutation with bandwidth B gives a feasible ILP with objective B. ($\Leftarrow$) Any feasible ILP solution encodes a valid permutation with bandwidth $\leq B$.

Solution extraction. For each vertex v, find p with $x_{v,p} = 1$; set $\text{config}[v] = p$. □

Definition 3.4 (Minimum Capacitated Spanning Tree): Given a weighted graph $G = (V, E)$ with root v_0, vertex requirements $r(v)$, and edge capacity c, find a spanning tree T rooted at v_0 minimizing $\sum_{e \in T} w(e)$ subject to: for each edge e in T, the sum of requirements in the subtree on the non-root side is at most c.
- Complexity: `2^num_edges`.
- Reduces to: ILP.

MinimumCapacitatedSpanningTree	
graph	The underlying graph G=(V,E)
weights	Edge weights w: E -> R
root	Root vertex
requirements	Vertex requirements r: V -> R (root has 0)
capacity	Subtree capacity bound

Minimum Capacitated Spanning Tree (ND5) [5]. NP-hard in the strong sense, even with unit requirements and capacity 3.

Rule 3.160: (Minimum Capacitated Spanning Tree (weighted) → Integer Linear Programming) Binary edge selectors y_e plus directed requirement-flow variables. Flow conservation routes each vertex's requirement to the root; capacity constraints bound flow per edge.
Overhead: `num_vars = 3 * num_edges`, `num_constraints = 5 * num_edges + num_vertices + 1`.

Proof: *Construction.* $3m$ variables: m edge selectors + $2m$ directed flow. Tree cardinality, binary bounds, flow conservation (each non-root vertex generates $r(v)$ flow toward root), flow-edge linking, and per-edge capacity bounds ($\leq c$). Minimize $\sum_e w(e) \cdot y_e$.

Correctness. ($\Rightarrow$) A feasible capacitated spanning tree induces a valid flow. ($\Leftarrow$) A feasible ILP solution encodes a spanning tree satisfying capacity constraints.

Solution extraction. First m variables (edge selectors) give the source configuration. □

Rule 3.161: (Minimum Matrix Cover $\rightarrow$ Integer Linear Programming) McCormick linearization of the quadratic form: n sign variables x_i plus $\frac{n(n-1)}{2}$ auxiliary variables y_{ij} linearizing products $x_i x_j$. Three McCormick constraints per pair. Minimize the linearized objective.
Overhead: `num_vars = num_rows + num_rows * (num_rows + -1 * 1) * 2^-1`, `num_constraints = 3 * num_rows * (num_rows + -1 * 1) * 2^-1`.

Proof: *Construction.* $n + \frac{n(n-1)}{2}$ binary variables and $3 \cdot \frac{n(n-1)}{2}$ constraints. For each pair $i < j$: $y_{ij} \leq x_i$, $y_{ij} \leq x_j$, $y_{ij} \geq x_i + x_j - 1$. The objective encodes $\sum_{i,j} a_{ij} \cdot f(i) \cdot f(j)$ via $f(i) = 2x_i - 1$.

Correctness. ($\Rightarrow$) Any sign assignment f maps to $x_i = \frac{f(i)+1}{2}$ with matching quadratic form value. ($\Leftarrow$) McCormick ensures $y_{ij} = x_i x_j$ at optimality.

Solution extraction. First n variables (sign variables) give the source configuration. □

Rule 3.162: (Integer Linear Programming $\rightarrow$ Integer Linear Programming) Integer variables with finite upper bounds are replaced by groups of binary variables using truncated binary encoding. Feasibility-Based Bound Tightening (FBBT) first infers per-variable upper bounds U_i from the constraint system; each integer variable $x_i \in \{0, ..., U_i\}$ is then encoded as $x_i = \sum_{j=0}^{K_i - 1} w_{ij} y_{ij}$ where $y_{ij} \in \{0, 1\}$ and the weights are powers of two with a truncated final weight ensuring $\sum w_{ij} = U_i$.
Overhead: `num_vars = num_vars`, `num_constraints = num_constraints + num_vars`.

Proof: *Construction.* Given an ILP over n non-negative integer variables with constraints $Ax \leq b$ (and/or $\geq, =$) and objective $c^\top x$: (1) Run FBBT to infer upper bounds $U_1, ..., U_n$. If infeasible, return a trivially infeasible binary ILP. If unbounded, fall back to $U_i = 2^{31} - 1$. (2) For each x_i, compute $K_i = \lceil \log_2(U_i + 1) \rceil$ and weights $w_{i0} = 1, w_{i1} = 2, ..., w_{i,K_i-2} = 2^{K_i-2}, w_{i,K_i-1} = U_i - (2^{K_i-1} - 1)$. (3) Substitute $x_i = \sum_j w_{ij} y_{ij}$ into every constraint and objective term, expanding each integer coefficient into a sum over binary coefficients.

Correctness. ($\Rightarrow$) Any integer assignment $x_i \in \{0, ..., U_i\}$ has a unique truncated binary representation y_{ij} with $\sum w_{ij} y_{ij} = x_i$, preserving all constraints and the objective value. ($\Leftarrow$) Any binary assignment $y_{ij} \in \{0, 1\}$ yields $x_i = \sum w_{ij} y_{ij} \in \{0, ..., U_i\}$, so the decoded integer assignment satisfies the original constraints.

Solution extraction. For each source variable x_i, compute $x_i = \sum_{j=0}^{K_i - 1} w_{ij} y_{ij}$ from the binary solution. □

Rule 3.163: (Hamiltonian Circuit $\rightarrow$ Hamiltonian Path) To decide whether $G = (V, E)$ contains a Hamiltonian circuit, we split an arbitrary vertex v_0 into two copies and attach a private pendant to each copy, forcing any Hamiltonian path in the expanded graph to enter through one pendant, traverse the original circuit, and exit through the other.
Overhead: `num_vertices = num_vertices + 3`, `num_edges = num_edges + num_vertices + 1`.

Proof: *Construction.* Let $G = (V, E)$ with $n = |V|$ and $m = |E|$. Fix vertex $v_0 = 0$. Introduce three new vertices: a duplicate v' of v_0, a pendant s, and a pendant t. Form $G' = (V', E')$ where $V' = V \cup \{v', s, t\}$, and E' contains (i) every original edge of E; (ii) an edge $\{v', u\}$ for each neighbor u of v_0 in G; (iii) the pendant edge $\{s, v_0\}$; and (iv) the pendant edge $\{t, v'\}$. Thus $|V'| = n + 3$ and $|E'| = m + \deg(v_0) + 2$.

Correctness. ($\Rightarrow$) Suppose G has a Hamiltonian circuit $C = (v_0, u_1, u_2, ..., u_{n-1}, v_0)$. The walk $s, v_0, u_1, u_2, ..., u_{n-1}, v', t$ visits every vertex of G' exactly once, so G' has a Hamiltonian path. ($\Leftarrow$) Suppose G' has a Hamiltonian path P. The pendants s and t have degree one, so P must start at s and end at t

(or vice versa), giving the form $s, v_0, \pi_1, ..., \pi_{n-1}, v', t$. Because $\{v', \pi_{n-1}\} \in E'$, vertex π_{n-1} is a neighbor of v_0 in G. Hence $v_0, \pi_1, ..., \pi_{n-1}, v_0$ is a Hamiltonian circuit in G.

Solution extraction. Orient the path so s is the start and t the end. Drop s and the last two elements v', t; the remaining sequence is the Hamiltonian circuit witness. □

Example: Cycle C_4 ($n = 4$): split v_0 into two copies with pendants

Source: HamiltonianCircuit **Target:** HamiltonianPath

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to HamiltonianPath/SimpleGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2,3
```

Step 1 – Source instance. The canonical source fixture is the cycle C_4 on vertices $\{0, ..., 3\}$ with 4 edges: $(0, 1)$, $(1, 2)$, $(2, 3)$, $(3, 0)$. The stored Hamiltonian-circuit witness is the permutation [0, 1, 2, 3].

Step 2 – Construction. Fix $v_0 = 0$. Its neighbors in the source are $\{1, 3\}$ ($\deg = 2$). Introduce $v' = 4$, $s = 5$, $t = 6$. The target graph G' has $7 = 4 + 3$ vertices and 8 edges: $(0, 1)$, $(1, 2)$, $(2, 3)$, $(3, 0)$, $(4, 1)$, $(4, 3)$, $(5, 0)$, $(6, 4)$. The 4 new edges are the duplicated adjacencies of v' plus the two pendant edges.

Step 3 – Verify a solution. The stored target Hamiltonian-path permutation is [5, 0, 1, 2, 3, 4, 6], visiting every vertex of G' exactly once. The path starts at pendant $s = 5$ and ends at pendant $t = 6$. Dropping s at the front and the last two vertices v', t at the back gives [0, 1, 2, 3], which is the source Hamiltonian circuit [0, 1, 2, 3].

Multiplicity: The fixture stores one canonical witness. For C_4 there are $4 \times 2 = 8$ directed Hamiltonian circuits (choice of start vertex and direction), each yielding a distinct target path.

Rule 3.164: (*k*-Clique → Subgraph Isomorphism) A k-clique in G is precisely a subgraph of G isomorphic to K_k. Constructing K_k as the pattern and passing G as the host reduces k-clique detection to a single subgraph-isomorphism query.
Overhead: `num_host_vertices = num_vertices`, `num_host_edges = num_edges`, `num_pattern_vertices = k`, `num_pattern_edges = k * (k + -1 * 1) * 2^-1`.

Proof: *Construction.* Given a k-clique instance (G, k) with $G = (V, E)$, $|V| = n$, $|E| = m$, build the subgraph-isomorphism instance (H, P) where $H := G$ (host) and $P := K_k$ (pattern, with k vertices and $\binom{k}{2}$ edges).

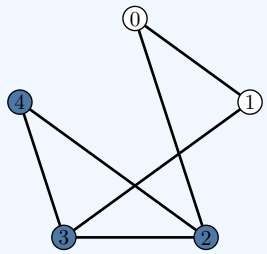
Correctness. ($\Rightarrow$) If G contains a k-clique $C = \{v_1, ..., v_k\}$, the injection $f(i) := v_i$ is a subgraph isomorphism from K_k into G. ($\Leftarrow$) If $f : V(K_k) \to V(G)$ is a subgraph isomorphism, the image $\{f(0), ..., f(k-1)\}$ is a set of k pairwise-adjacent vertices, i.e., a k-clique.

Solution extraction. The subgraph-isomorphism solution $c \in \{0, ..., n-1\}^k$ gives the host vertex assigned to each pattern vertex. The k-clique indicator $x \in \{0, 1\}^n$ sets $x[c[i]] := 1$ for each i. □

Example: 5-vertex graph with $k = 3$: clique detection via subgraph isomorphism

Source: KClique **Target:** SubgraphIsomorphism

```
$ pred create --example KClique/SimpleGraph -o kclique.json
$ pred reduce kclique.json --to SubgraphIsomorphism -o bundle.json
$ pred solve bundle.json
$ pred evaluate kclique.json --config 0,0,1,1,1
```



Step 1 – Source instance. The canonical KClique instance has $n = 5$ vertices, $|E| = 6$ edges, and clique size $k = 3$. The task is to decide whether G contains k pairwise-adjacent vertices.

Step 2 – Construct the pattern graph. The reduction sets the host graph $H := G$ (unchanged) and builds the pattern graph $P := K_k$, the complete graph on $k = 3$ vertices with $\binom{k}{2} = 3$ edges. The target instance has 5 host vertices and 3 pattern vertices.

Step 3 – Variable semantics. The source uses $n = 5$ binary indicator variables ($x_v = 1$ iff vertex v is in the clique). The target uses $k = 3$ variables, each ranging over $\{0, ..., n-1\}$, specifying which host vertex each pattern vertex maps to.

Step 4 – Verify a solution. The canonical witness has source config $\boldsymbol{x} = (0, 0, 1, 1, 1)$, selecting vertices $\{2, 3, 4\}$. The target config is $(2, 3, 4)$, mapping pattern vertex i to host vertex c_i. The image vertices $\{2, 3, 4\}$ are pairwise adjacent in G (they form a triangle), confirming the isomorphism ✓

Multiplicity: The fixture stores one canonical witness. The $k!$ permutations of the clique vertices each give a valid subgraph isomorphism, so the target side has $3! = 6$ witnesses for this single source clique.

Rule 3.165: (Partition → Multiprocessor Scheduling) Each element a_i becomes a task of length a_i on $m = 2$ processors with deadline $D = \lfloor \frac{S}{2} \rfloor$. A balanced partition exists iff a feasible schedule exists.
Overhead: `num_tasks = num_elements`.

Proof: *Construction.* Let $A = (a_1, ..., a_n)$ with total sum $S = \sum_{i=1}^{n} a_i$. Set task lengths $\ell_i = a_i$, number of processors $m = 2$, and deadline $D = \lfloor \frac{S}{2} \rfloor$.

Correctness. ($\Rightarrow$) If $A' \subseteq A$ has $\sum_{i \in A'} a_i = \frac{S}{2}$, assign tasks in A' to processor 0 and the rest to processor 1; both loads equal $\frac{S}{2} = D$. ($\Leftarrow$) If a feasible schedule exists with both loads $\leq D = \lfloor \frac{S}{2} \rfloor$, since both loads sum to S and each is at most $\lfloor \frac{S}{2} \rfloor$, equality holds, giving a balanced partition.

Solution extraction. The processor assignment $p_i \in \{0, 1\}$ is the partition assignment directly. □

Example: 4 elements, total sum $S = 10$, deadline $D = 5$

Source: Partition **Target:** MultiprocessorScheduling

```
$ pred create --example Partition -o partition.json
$ pred reduce partition.json --to MultiprocessorScheduling -o bundle.json
$ pred solve bundle.json
$ pred evaluate partition.json --config 0,1,1,0
```

Step 1 – Source instance. The canonical Partition instance has sizes $(1, 2, 3, 4)$ with total sum $S = 10$. A balanced partition requires each subset to sum to $\frac{S}{2} = 5$.

Step 2 – Construction. The reduction creates 4 tasks with lengths $(1, 2, 3, 4)$, sets the number of processors to $m = 2$, and the deadline to $D = \lfloor \frac{S}{2} \rfloor = 5$. No auxiliary variables are introduced: the target has the same 4 binary coordinates as the source.

Step 3 – Verify a solution. The canonical witness is $\boldsymbol{x} = (0, 1, 1, 0)$, which is the same binary vector on both sides. Processor 0 receives tasks at indices $\{0, 3\}$ with sizes $(1, 4)$, giving load $5 \leq 5 = D$.

Processor 1 receives tasks at indices $\{1, 2\}$ with sizes $(2, 3)$, giving load $5 \leq 5 = D$. Total: $5 + 5 = 10 = S$ ✓

Multiplicity: The example DB stores one canonical witness. This instance admits other balanced partitions (e.g., swapping the two subsets), but one witness suffices to demonstrate the reduction.

Rule 3.166: (Hamiltonian Circuit → Bottleneck Traveling Salesman) Construct a complete weighted graph with weight 1 on edges of G and weight 2 on non-edges. A Hamiltonian tour of bottleneck cost 1 exists iff G has a Hamiltonian circuit.
Overhead: `num_vertices = num_vertices, num_edges = num_vertices * (num_vertices + -1 * 1) * 2^-1`.

Proof: *Construction.* Let $G = (V, E)$ with $n = |V|$. Build K_n with $w(u, v) = 1$ if $\{u, v\} \in E$, else $w(u, v) = 2$.

Correctness. ($\Rightarrow$) A Hamiltonian circuit in G uses only weight-1 edges, giving bottleneck cost 1. ($\Leftarrow$) A tour with bottleneck ≤ 1 uses only weight-1 edges, which are exactly edges of G, so it is a Hamiltonian circuit.

Solution extraction. Recover the vertex cycle order from the tour edge set. □

Example: $n = 4$ vertices, $|E| = 4$ edges: HC → BTSP

Source: HamiltonianCircuit **Target:** BottleneckTravelingSalesman

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to BottleneckTravelingSalesman -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2,3
```

Step 1 – Source instance. The canonical HC instance is a graph with $n = 4$ vertices and $|E| = 4$ edges.

Step 2 – Construction. Build the complete graph K_4 with 6 edges. Each original edge gets weight 1 and each non-edge gets weight 2. The target edge weights are $(1, 2, 1, 1, 2, 1)$, where the 4 entries equal to 1 correspond to the $|E| = 4$ source edges and the 2 entries equal to 2 correspond to the non-edges.

Step 3 – Verify a solution. The source tour visits vertices in order $(0, 1, 2, 3)$. In the target, the selected edges are those with indicator 1 in $(1, 0, 1, 1, 0, 1)$, all of which have weight 1, giving bottleneck cost $= 1$ ✓.

Multiplicity: The fixture stores one canonical witness. The 4-cycle has 4 rotations × 2 reflections = 8 Hamiltonian circuits in total.

Rule 3.167: (k-Clique → Conjunctive Boolean Query) Introduce k existential variables over the vertex set and require the edge relation $R(y_i, y_j)$ for every pair $i < j$. The query is satisfiable iff G contains a k -clique.
Overhead: `domain_size = num_vertices, num_relations = 1, num_variables = k, num_conjuncts = k * (k + -1 * 1) * 2^-1`.

Proof: *Construction.* Given $G = (V, E)$ with $|V| = n$ and parameter k, set domain $D = V$, define binary relation $R = \{(u, v), (v, u) : \{u, v\} \in E\}$, and form the conjunction $\varphi = \bigwedge_{0 \leq i < j < k} R(y_i, y_j)$ with $\binom{k}{2}$ conjuncts.

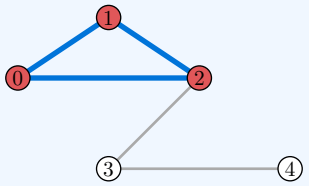
Correctness. ($\Rightarrow$) A k-clique $\{v_0, ..., v_{k-1}\}$ satisfies every conjunct since all pairs are adjacent. ($\Leftarrow$) A satisfying assignment gives k vertices where every pair is in R, hence pairwise-adjacent.

Solution extraction. The satisfying tuple $(d_0, ..., d_{k-1})$ gives vertex indices; set $x[d_i] = 1$ for the clique indicator. □

Example: 5-vertex graph ($n = 5$, $|E| = 5$, $k = 3$)

Source: KClique **Target:** ConjunctiveBooleanQuery

```
$ pred create --example KClique/SimpleGraph -o kclique.json
$ pred reduce kclique.json --to ConjunctiveBooleanQuery -o bundle.json
$ pred solve bundle.json
$ pred evaluate kclique.json --config 1,1,1,0,0
```



Step 1 – Source instance. The graph G has $n = 5$ vertices and $|E| = 5$ edges, with $k = 3$. Vertices $\{0, 1, 2\}$ form a triangle (the unique 3-clique).

Step 2 – Build the edge relation. Define the domain $D = \{0, ..., 4\}$ with $|D| = 5$. The single binary relation R contains both orientations of every edge: $|R| = 2\ |E| = 10$ tuples of arity 2.

Step 3 – Form the conjunctive query. Introduce $k = 3$ existential variables y_0, y_1, y_2 over D. The conjunction has $\binom{k}{2} = 3$ conjuncts: $R(y_0, y_1) \wedge R(y_0, y_2) \wedge R(y_1, y_2)$.

Step 4 – Verify a solution. The satisfying assignment $(y_0, y_1, y_2) = (0, 1, 2)$ maps to vertices $\{0, 1, 2\}$. Check each conjunct: $(0, 1) \in R$ ✓, $(0, 2) \in R$ ✓, $(1, 2) \in R$ ✓ — all pairs are adjacent, confirming the 3-clique. The source indicator is (1, 1, 1, 0, 0), marking exactly vertices $\{0, 1, 2\}$.

Multiplicity: The fixture stores one canonical witness. The triangle $\{0, 1, 2\}$ is the unique 3-clique in this graph, but the CBQ query has $3! = 6$ satisfying tuples (one per permutation of the three vertices); the canonical witness selects the sorted order $(0, 1, 2)$.

Rule 3.168: (Exact Cover by 3-Sets → Staff Scheduling) Each universe element becomes a time period requiring exactly one worker, each 3-element subset becomes a schedule active on those three periods, and the worker budget is $q = |X|/3$. A feasible assignment selects q schedules covering every period exactly once. *Overhead:* `num_periods = universe_size`, `num_schedules = num_subsets`, `num_workers = universe_size * 3^-1`.

Proof: *Construction.* Let $(X, \mathcal{C})$ with $|X| = 3q$. Set $m_p = 3q$ periods (one per element), requirement $r[i] = 1$ for all i, worker budget $W = q$. Each subset $S_j = \{a, b, c\}$ defines schedule σ_j with $\sigma_{j[i]} = 1$ iff $i \in S_j$.

Correctness. ($\Rightarrow$) An exact cover I with $|I| = q$ assigns one worker per selected schedule, covering each period exactly once. ($\Leftarrow$) A feasible assignment with $\sum_j w_j = q$ and each period covered exactly once gives pairwise-disjoint schedules covering all of X.

Solution extraction. Map $w \in \mathbb{N}^m$ to $c \in \{0, 1\}^m$ by $c[j] = 1$ if $w[j] > 0$. □

Example: $|X| = 6$, $|\mathcal{C}| = 4$ subsets

Source: ExactCoverBy3Sets **Target:** StaffScheduling

```
$ pred create --example ExactCoverBy3Sets -o x3c.json
$ pred reduce x3c.json --to StaffScheduling -o bundle.json
```

```
$ pred solve bundle.json
$ pred evaluate x3c.json --config 1,1,0,0
```

Step 1 – Source instance. The X3C instance has universe $X = \{0, ..., 5\}$ ($|X| = 6 = 3q$, so $q = 2$) and $|\mathcal{C}| = 4$ subsets: $S_0 = \{0, 1, 2\}$, $S_1 = \{3, 4, 5\}$, $S_2 = \{0, 3, 4\}$, $S_3 = \{1, 2, 5\}$.

Step 2 – Construct StaffScheduling instance. The reduction creates 6 periods (one per universe element), each with requirement $r[i] = 1$. Each subset S_j becomes a schedule σ_j with $\sigma_j[i] = 1$ iff $i \in S_j$, giving shifts_per_schedule $= 3$. The worker budget is $W = q = 2$.

Step 3 – Verify a solution. The canonical cover selects subsets via (1, 1, 0, 0): $S_0 = \{0, 1, 2\}$, $S_1 = \{3, 4, 5\}$. These 2 subsets are pairwise disjoint and cover all 6 elements ✓
The target config (1, 1, 0, 0) assigns $w[j]$ workers to schedule j: total workers $= 2 = q$ ✓. Each period is covered by exactly one worker ✓

Multiplicity: The fixture stores one canonical witness. The instance admits a second exact cover $\{S_2, S_3\}$, but the fixture records only the first found.

Rule 3.169: (*k*-SAT (*k*-ary) → Decision Minimum Vertex Cover (weighted)) Use the classical 3-SAT to Vertex Cover gadget construction, then ask whether the resulting graph has a vertex cover of size at most $k = n + 2m$ [5].
Overhead: `num_vertices = 2 * num_vars + 3 * num_clauses`, `num_edges = num_vars + 6 * num_clauses`, `k = num_vars + 2 * num_clauses`.

Proof: *Construction.* Given 3-CNF φ with n variables and m clauses, build the graph $G = (V, E)$ used in the classical 3-SAT → Minimum Vertex Cover reduction: for each variable x_i, add literal vertices u_i and $\overline{u}_i$ with one truth-setting edge between them; for each clause c_j, add triangle vertices t_0^j, t_1^j, t_2^j; for each literal position (j, r), add a communication edge from t_r^j to the literal vertex representing that literal. Assign unit weight to every vertex and set the decision threshold to $k = n + 2m$.

Correctness. ($\Rightarrow$) If φ is satisfiable, choose one literal vertex per variable according to a satisfying assignment and choose two triangle vertices per clause, omitting a vertex whose literal is true. This gives a cover of size $n + 2m$, so the decision instance is yes. ($\Leftarrow$) If the decision instance is yes, any cover of size at most $n + 2m$ must use exactly one literal vertex per variable edge and exactly two vertices per clause triangle. The omitted triangle vertex in each clause has its communication edge covered only by a selected literal vertex, so the corresponding literal is true. These selected literal vertices define a satisfying assignment for φ.

Solution extraction. For each variable x_i, inspect the truth-setting pair. Set $x_i = 1$ when the cover contains u_i, and set $x_i = 0$ otherwise. □

Example: 3-SAT with $n = 3$ variables, $m = 2$ clauses reduced to Decision Minimum Vertex Cover with bound $k = 7$

Source: KSatisfiability **Target:** DecisionMinimumVertexCover

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to DecisionMinimumVertexCover/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 0,0,1
```

Step 1 – Source instance. The 3-SAT formula has $n = 3$ variables and $m = 2$ clauses: $c_0 = (x_1 \vee x_2 \vee x_3)$, $c_1 = (\overline{x}_1 \vee \overline{x}_2 \vee x_3)$. A satisfying assignment is (0, 0, 1).

Step 2 – Build the vertex-cover graph. Create one truth-setting edge $(u_i, \overline{u}_i)$ per variable and one clause triangle per clause. Each triangle vertex is connected to the literal vertex of its clause position

by a communication edge. The resulting graph has $|V| = 2n + 3m = 12$ vertices and $|E| = n + 6m = 15$ edges.

Step 3 – Add the decision threshold. Wrap the constructed Minimum Vertex Cover instance in the decision predicate with bound $k = n + 2m = 7$. For this example, $k = 3 + 2 \cdot 2 = 7$.

Step 4 – Verify a witness. The target configuration (0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0) selects 7 vertices, so it meets the bound exactly. Each truth-setting edge has one selected endpoint ✓. Each clause triangle has two selected vertices ✓. Each communication edge is covered either by its literal endpoint or by its triangle endpoint ✓. Extracting the truth-setting choices returns (0, 0, 1), which satisfies every clause ✓

Multiplicity: The fixture stores one canonical decision witness. Different satisfying assignments may yield different size-k covers of the same graph.

Rule 3.170: (Decision Minimum Vertex Cover (weighted) → Hamiltonian Circuit) Garey and Johnson's Theorem 3.4 replaces each source edge by a 12-vertex cover-testing gadget and uses k selector vertices to choose k source vertices whose incident gadget-paths together cover every gadget [5]. In the unit-weight decision setting, the constructed graph is Hamiltonian iff the source graph has a vertex cover of size at most k.
Overhead: `num_vertices = 12 * num_edges + k`, `num_edges = 16 * num_edges + -1 * num_vertices + 2 * k * num_vertices`.

Proof: *Construction.* Let the source be a unit-weight Decision Minimum Vertex Cover instance $(G = (V, E), k)$ with G simple. For each edge $e = \{u, v\} \in E$, create a gadget with vertices (u, e, i) and (v, e, i) for $1 \le i \le 6$. Add the two 6-chains on the u-side and v-side together with the four cross edges $\{(u, e, 3), (v, e, 1)\}$, $\{(v, e, 3), (u, e, 1)\}$, $\{(u, e, 6), (v, e, 4)\}$, and $\{(v, e, 6), (u, e, 4)\}$. For every source vertex v, order its incident edges as $e_{v[1]}, ..., e_{v[\deg(v)]}$ and connect $\left\{\left(v, e_{v[i]}, 6\right), \left(v, e_{v[i+1]}, 1\right)\right\}$ for $1 \le i < \deg(v)$, forming one path that contains exactly the gadget copies labeled by v. Finally add selector vertices $a_1, ..., a_k$ and join each selector to both endpoints of every non-isolated vertex-path. Thus the theorem branch has $k + 12|E|$ vertices and $14|E| + \sum_{v \in V^+}(\deg(v) - 1) + 2k|V^+|$ edges, where $V^+ = \{v \in V : \deg(v) > 0\}$.

Correctness. ($\Rightarrow$) Suppose $C \subseteq V$ is a vertex cover with $|C| \le k$. Because all weights are 1, we may pad C with arbitrary additional non-isolated vertices until it has exactly k elements, say $v_1, ..., v_k$. For every edge gadget $e = \{u, v\}$, traverse it in one of the three gadget modes from [5]: if only $u \in C$, follow the unique Hamiltonian path from $(u, e, 1)$ to $(u, e, 6)$ through all 12 gadget vertices; if only $v \in C$, use the symmetric path from $(v, e, 1)$ to $(v, e, 6)$ through all 12 vertices; if both endpoints lie in C, use the two disjoint side paths from $(u, e, 1)$ to $(u, e, 6)$ and from $(v, e, 1)$ to $(v, e, 6)$. Chaining these gadget traversals along the paths for $v_1, ..., v_k$ and connecting consecutive paths through the selectors yields a Hamiltonian circuit of the target graph. ($\Leftarrow$) Suppose the target graph has a Hamiltonian circuit. Each selector has degree two inside the circuit and therefore cuts the circuit into k selector-to-selector segments. Inside any edge gadget, the circuit can appear only in the three modes above, so each segment must stay on the path corresponding to one source vertex. Mark a source vertex v selected exactly when both endpoints of its path are adjacent to selectors in the Hamiltonian circuit. This selects exactly k source vertices. Every edge gadget must be completely visited, and that is possible only if at least one of its endpoint paths is selected, so every source edge has a selected endpoint. Hence the extracted set is a vertex cover of size at most k.

Solution extraction. Given a Hamiltonian circuit witness, inspect the two endpoints of each source vertex-path. Set $x_v = 1$ iff both path endpoints are adjacent to selector vertices in the cycle; otherwise set $x_v = 0$. The resulting indicator vector is a valid source-side vertex cover. □

Example: 3-vertex path with bound $k = 1$: one selector threads two cover-testing gadgets

Source: DecisionMinimumVertexCover **Target:** HamiltonianCircuit

```
$ pred create --example DecisionMinimumVertexCover/SimpleGraph/i32 -o dmvc.json
$ pred reduce dmvc.json --to HamiltonianCircuit/SimpleGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate dmvc.json --config 0,1,0
```

Step 1 – Source instance. The canonical Decision Minimum Vertex Cover fixture has inner graph G on vertices $\{0, 1, 2\}$ with edges $(0, 1)$, $(1, 2)$ and unit weights. The bound is $k = 1$, and the stored cover witness is (0, 1, 0), i.e.
$C = \{1\}$.

Step 2 – Build the Hamiltonian graph. There is one selector vertex a_1 and one 12-vertex gadget for each source edge, so the target graph has $1 + 2 \cdot 12 = 25$ vertices. The path for source vertex 1 chains the two gadgets through the connector from $(1, e_0, 6)$ to $(1, e_1, 1)$. The completed target has 35 edges.

Step 3 – Verify a witness. The stored Hamiltonian circuit is (0, 7, 8, 9, 1, 2, 3, 4, 5, 6, 10, 11, 12, 13, 14, 15, 19, 20, 21, 22, 23, 24, 16, 17, 18). Reading the cycle between selector contacts shows that the unique selector traverses first the gadget for edge $(0, 1)$ in the "only vertex 1 chosen" mode and then the gadget for edge $(1, 2)$ in the same mode, visiting every one of the 25 target vertices exactly once. Extracting the selector-adjacent path endpoints returns the source cover (0, 1, 0) = $\{1\}$, which indeed covers both source edges ✓

Multiplicity: The fixture stores one canonical Hamiltonian circuit. Rotating or reversing that same cycle yields equivalent target witnesses with the same extracted cover.

Rule 3.171: (*k*-SAT (*k*-ary) → Minimum Vertex Cover (weighted)) Each variable contributes a truth-setting edge; each clause contributes a satisfaction-testing triangle. The formula is satisfiable iff the graph has a vertex cover of size $n + 2m$.
Overhead: `num_vertices = 2 * num_vars + 3 * num_clauses`, `num_edges = num_vars + 6 * num_clauses`.

Proof: *Construction.* Given 3-CNF φ with n variables and m clauses, construct $G = (V, E)$ with $|V| = 2n + 3m$. For each variable x_i: vertices u_i (index $2i$) and $\overline{u}_i$ (index $2i + 1$) with edge $(u_i, \overline{u}_i)$. For each clause c_j: triangle vertices t_0^j, t_1^j, t_2^j at indices $2n + 3j, 2n + 3j + 1, 2n + 3j + 2$. Communication edges connect each t_k^j to the literal vertex of its k-th literal.

Correctness. ($\Rightarrow$) A satisfying assignment selects literal vertices (n total) and two triangle vertices per clause ($2m$ total), covering all edges. ($\Leftarrow$) A cover of size $n + 2m$ must include exactly one literal vertex per variable and two triangle vertices per clause; the uncovered triangle vertex's communication edge forces the corresponding literal to be true.

Solution extraction. For variable x_i, set $x_i = 1$ if the cover indicator at position $2i$ is 1. □

Example: 3-SAT with $n = 3$ variables, $m = 2$ clauses

Source: KSatisfiability **Target:** MinimumVertexCover

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to MinimumVertexCover/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 0,0,1
```

Step 1 – Source instance. The 3-SAT formula has $n = 3$ variables and $m = 2$ clauses: $c_0 = (x_1 \vee x_2 \vee x_3)$, $c_1 = (\overline{x}_1 \vee \overline{x}_2 \vee x_3)$. A satisfying assignment is (0, 0, 1), i.e.
$x_1 = 0$, $x_2 = 0$, $x_3 = 1$.

Step 2 – Truth-setting edges. For each variable x_i, create vertices u_i (index $2(i - 1)$) and $\overline{u}_i$ (index $2(i - 1) + 1$) connected by a truth-setting edge. This gives $2n = 6$ literal vertices and $n = 3$ edges.

Step 3 – Clause triangles and communication edges. For each clause c_j, create a triangle of 3 vertices at indices $2n+3j, 2n+3j+1, 2n+3j+2$, connected by 3 internal edges. Each triangle vertex t_k^j is also connected to its literal vertex by a communication edge (3 per clause). Total: $3m = 6$ clause vertices, $3m = 6$ triangle edges, $3m = 6$ communication edges.

Step 4 – Target graph dimensions. The resulting graph has $|V| = 2n + 3m = 12$ vertices and $|E| = n + 6m = 15$ edges, with unit weights.

Step 5 – Verify a solution. The satisfying assignment (0, 0, 1) maps to a vertex cover of size $n + 2m = 7$. The target configuration is (0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0): the cover selects 7 vertices. For each truth-setting edge, exactly one endpoint is in the cover ✓. For each clause triangle, exactly two of three vertices are covered ✓. Each communication edge has at least one endpoint in the cover ✓.

Multiplicity: The fixture stores one canonical witness. Other valid covers correspond to different satisfying assignments of the formula.

Rule 3.172: (k-SAT (k-ary) → Monochromatic Triangle) This $O(n+m)$ reduction [5, GT6] uses one positive/negative literal pair per variable and one three-vertex clause gadget per clause. The target graph has $2n + 3m$ vertices and $n + 9m$ edges.
Overhead: `num_vertices = 2 * num_vars + 3 * num_clauses`, `num_edges = num_vars + 9 * num_clauses`.

Proof: *Construction.* Let $\varphi = \wedge_{j=1}^{m} \left(\ell_1^j \vee \ell_2^j \vee \ell_3^j\right)$ be a 3-CNF formula on variables $x_1, ..., x_n$. For each variable x_i, create literal vertices p_i and n_i and add the negation edge (p_i, n_i). For each clause C_j, map each literal ℓ_r^j to its literal vertex v_r^j ($x_i \mapsto p_i$, $\overline{x}_i \mapsto n_i$). Introduce fresh intermediate vertices m_{12}^j, m_{13}^j, and m_{23}^j. Add fan edges $\left(v_1^j, m_{12}^j\right)$, $\left(v_2^j, m_{12}^j\right)$, $\left(v_1^j, m_{13}^j\right)$, $\left(v_3^j, m_{13}^j\right)$, $\left(v_2^j, m_{23}^j\right)$, and $\left(v_3^j, m_{23}^j\right)$, then connect the intermediates into the clause triangle $\left(m_{12}^j, m_{13}^j\right)$, $\left(m_{12}^j, m_{23}^j\right)$, $\left(m_{13}^j, m_{23}^j\right)$. Each clause gadget therefore contributes exactly four triangles: the clause triangle itself and the three fan triangles rooted at v_1^j, v_2^j, and v_3^j.

Correctness. ($\Rightarrow$) Fix a satisfying assignment of φ. Color each negation edge (p_i, n_i) by the truth value of x_i, using color 0 for true and color 1 for false. For any clause, at least one of its literals is satisfied, and the four-triangle gadget has a 2-edge-coloring extending those literal choices with no monochromatic triangle; the implementation follows the verified local construction from issue 884 and colors each clause independently because the intermediates are clause-local. ($\Leftarrow$) Given a triangle-free coloring of the target graph, inspect the negation-edge colors. Either those colors, or their global complement after swapping the two colors, yields a satisfying assignment of the source formula under the same verified gadget analysis.

Solution extraction. Read the negation-edge colors in variable order, setting $x_i = 1$ when (p_i, n_i) has color 0 and $x_i = 0$ otherwise. If that assignment does not satisfy φ, complement all bits; this accounts for the global color-swap symmetry of the target witness. □

Example: Single-clause 3-SAT instance ($n = 3$, $m = 1$) reduced to a 4-triangle graph

Source: KSatisfiability **Target:** MonochromaticTriangle

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to MonochromaticTriangle/SimpleGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 1,1,1
```

Step 1 – Source instance. The fixture uses the single clause $c_1 = (x_1 \vee x_2 \vee x_3)$. The extracted satisfying assignment is (1, 1, 1).

Step 2 – Build the clause gadget. Create literal vertices $p_1, p_2, p_3, n_1, n_2, n_3$ together with negation edges (p_1, n_1), (p_2, n_2), and (p_3, n_3). For the clause, add intermediates m_{12}, m_{13}, m_{23} and the six

fan edges (p_1, m_{12}), (p_2, m_{12}), (p_1, m_{13}), (p_3, m_{13}), (p_2, m_{23}), (p_3, m_{23}), plus the clause triangle on (m_{12}, m_{13}, m_{23}). The target therefore has $|V| = 9$ vertices, $|E| = 12$ edges, and 4 triangles.

Step 3 – Verify a witness. The stored target coloring is (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0). Its first $n = 3$ entries color the negation edges; reading those colors and applying the global color-swap symmetry fix yields the source assignment (1, 1, 1), which satisfies c_1 ✓. The same target coloring makes all four target triangles non-monochromatic ✓.

Multiplicity: The fixture stores one canonical target coloring. Swapping the two edge colors gives another valid witness and may flip the decoded assignment to its complement.

Rule 3.173: (*k*-SAT (*k*-ary) → 1-in-3 SAT) This $O(n + m)$ reduction [5, LO4], [68] preserves the original variables, adds two global variables z_0 and z_T, and introduces six fresh auxiliaries per clause. Each 3-SAT clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ is replaced by a five-clause gadget of exact-one constraints. The target therefore has $n + 2 + 6m$ variables and $1 + 5m$ clauses.
Overhead: `num_vars = num_vars + 2 + 6 * num_clauses`, `num_clauses = 1 + 5 * num_clauses`.

Proof: *Construction.* Let $\varphi = \wedge_{j=1}^{m} \left(\ell_1^j \vee \ell_2^j \vee \ell_3^j\right)$ be a 3-CNF formula on variables $x_1, ..., x_n$. Introduce global variables z_0 and z_T, and add the clause $R(z_0, z_0, z_T)$, where $R(u, v, w)$ means that exactly one of the three literals is true. This forces $z_0 = 0$ and $z_T = 1$. For every source clause $C_j = \left(\ell_1^j \vee \ell_2^j \vee \ell_3^j\right)$, introduce six fresh auxiliaries $a_j, b_j, c_j, d_j, e_j, f_j$ and append the five target clauses

$$R\left(\ell_1^j, a_j, d_j\right), \quad R\left(\ell_2^j, b_j, d_j\right), \quad R\left(a_j, b_j, e_j\right), \quad R\left(c_j, d_j, f_j\right), \quad R\left(\ell_3^j, c_j, z_0\right).$$

Negated source literals are copied directly; no complement variables are needed because 1-in-3 SAT clauses in this codebase may contain negated literals.

Correctness. (⇒) Suppose φ is satisfiable. Set the first n target variables according to any satisfying source assignment, and set $(z_0, z_T) = (0, 1)$. Consider one clause gadget. Because $\ell_1^j \vee \ell_2^j \vee \ell_3^j = 1$, the truth triple $\left(\ell_1^j, \ell_2^j, \ell_3^j\right)$ is one of the seven nonzero 0/1 patterns. For each such pattern there is a choice of $\left(a_j, b_j, c_j, d_j, e_j, f_j\right)$ satisfying all five exact-one clauses; the implementation uses the gadget directly and the worked example shows one such extension. Doing this independently for every clause yields a satisfying 1-in-3 assignment.

(⇐) Suppose the target instance is satisfiable. The global clause forces $z_0 = 0$. Fix any clause gadget, and assume for contradiction that $\ell_1^j = \ell_2^j = \ell_3^j = 0$. Then $R\left(\ell_3^j, c_j, z_0\right)$ forces $c_j = 1$. Next $R\left(c_j, d_j, f_j\right)$ forces $d_j = f_j = 0$. Then $R\left(\ell_1^j, a_j, d_j\right)$ and $R\left(\ell_2^j, b_j, d_j\right)$ force $a_j = b_j = 1$. But now $R\left(a_j, b_j, e_j\right)$ has two true literals, impossible. Therefore every source clause has at least one true literal, so the restriction to the original variables satisfies φ.

Solution extraction. Return the first n target coordinates unchanged, discarding z_0, z_T, and all clause auxiliaries. □

Example: Single clause ($n = 3$, $m = 1$): 3-SAT → 1-in-3 SAT

Source: KSatisfiability **Target:** OneInThreeSatisfiability

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to OneInThreeSatisfiability -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 0,0,1
```

Step 1 – Source instance. The source formula has one clause $c_1 = (x_1 \vee x_2 \vee x_3)$. The canonical satisfying assignment is (0, 0, 1), i.e.
$x_1 = 0$, $x_2 = 0$, $x_3 = 1$.

Step 2 – Add the global false literal. Introduce z_0 (index 4) and z_T (index 5), then add the forcing clause $R(z_0, z_0, z_T)$, where $R(u, v, w)$ means "exactly one of u, v, w is true." In the stored witness, $(z_0, z_T) = (0, 1)$, which is the unique satisfying pattern of that clause.

Step 3 – Build the clause gadget. Introduce auxiliaries $a_1, b_1, c_1, d_1, e_1, f_1$ at indices 6 through 11 and replace c_1 by the five 1-in-3 clauses

$$R(x_1, a_1, d_1), \quad R(x_2, b_1, d_1), \quad R(a_1, b_1, e_1), \quad R(c_1, d_1, f_1), \quad R(x_3, c_1, z_0).$$

In the exported target instance these become exactly the six clauses $(4, 4, 5)$, $(1, 6, 9)$, $(2, 7, 9)$, $(6, 7, 10)$, $(8, 9, 11)$, $(3, 8, 4)$.

Step 4 – Verify a witness. The canonical target witness is (0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0). Reading off the auxiliaries gives $(a_1, b_1, c_1, d_1, e_1, f_1) = (0, 0, 0, 1, 1, 0)$. Every target clause has exactly one true literal ✓, and restricting to the first three coordinates recovers $(0, 0, 1)$, which satisfies the original clause ✓.

Multiplicity: The fixture stores one canonical witness. This single-clause source formula has 7 satisfying assignments, and each satisfying literal pattern extends to at least one target witness by the gadget construction.

Rule 3.174: (*k*-SAT (*k*-ary) → Directed Two-Commodity Integral Flow) This $O(n + m)$ reduction [5, ND38], [135] builds a unit-capacity directed network of variable lobes and clause sinks. Each literal occurrence contributes one clause-entry segment on the corresponding branch, and the repository pads every branch with one dummy segment so variables with missing polarity occurrences still have two realizable choices. For a 3-CNF formula with n variables and m clauses, the target has $6n + 7m + 4$ vertices and $7n + 13m + 1$ arcs.
Overhead: `num_vertices = 6 * num_vars + 2 * num_literals + num_clauses + 4`, `num_arcs = 7 * num_vars + 4 * num_literals + num_clauses + 1`.

Proof: *Construction.* Let $\varphi = \wedge_{j=1}^{m} C_j$ be a 3-CNF formula on variables $x_1, ..., x_n$. For each variable x_i, let p_i be its number of positive occurrences and q_i its number of negative occurrences. Create terminals s_1, t_1, s_2, t_2, one clause vertex d_j for each clause, and for each variable a lobe with entry a_i and exit b_i. The upper branch contains a dummy segment followed by one directed segment $u_{i,r}^{+} \to w_{i,r}^{+}$ for each positive occurrence $r = 1, ..., p_i$; the lower branch contains a dummy segment followed by one directed segment $u_{i,r}^{-} \to w_{i,r}^{-}$ for each negative occurrence $r = 1, ..., q_i$. Chain the lobes with arcs $s_1 \to a_1$, $b_i \to a_{i+1}$, and $b_n \to t_1$. For every positive occurrence of x_i in clause C_j, add $s_2 \to u_{i,r}^{+}$ and $w_{i,r}^{+} \to d_j$; for every negative occurrence, add $s_2 \to u_{i,r}^{-}$ and $w_{i,r}^{-} \to d_j$. Finally add $d_j \to t_2$ for each clause. All arcs have capacity 1, and the requirements are $R_1 = 1$ and $R_2 = m$.

Correctness. ($\Rightarrow$) Suppose φ is satisfiable. Route commodity 1 through the lower branch of lobe i when $x_i = 1$, and through the upper branch when $x_i = 0$. Consider any clause C_j. Choose one satisfied literal occurrence in that clause. If the chosen literal is x_i, route one unit of commodity 2 along $s_2 \to u_{i,r}^{+} \to w_{i,r}^{+} \to d_j \to t_2$; if it is $\overline{x}_i$, use the analogous lower-branch route. A true literal always lies on the branch opposite to commodity 1, so the shared occurrence segment is free. Because occurrence segments are distinct per clause position, the m commodity-2 units respect all capacities.

($\Leftarrow$) Suppose the target instance is feasible. Because commodity 1 has requirement $R_1 = 1$ and conservation holds for commodity 1 away from s_1 and t_1, its unit flow traverses exactly one branch in each lobe. Define $x_i = 1$ iff commodity 1 uses the lower branch of lobe i. Now fix any clause vertex d_j. Since $d_j \to t_2$ is the only outgoing arc from d_j, meeting $R_2 = m$ forces one unit of commodity 2 through every clause vertex. That unit must arrive from some occurrence segment $u \to w \to d_j$. The shared arc $u \to w$ cannot lie on commodity 1's chosen branch, so the corresponding literal is true under the assignment above. Hence every clause contains a true literal and φ is satisfiable.

Solution extraction. For each variable lobe, inspect the first lower-branch arc leaving its entry. Output $x_i = 1$ exactly when commodity 1 uses that arc. □

Example: Two-clause 3-SAT instance ($n = 3$, $m = 2$) reduced to Directed Two-Commodity Integral Flow

Source: KSatisfiability **Target:** DirectedTwoCommodityIntegralFlow

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to DirectedTwoCommodityIntegralFlow -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 1,1,0
```

Step 1 – Source instance. The source formula has clauses $c_1 = (x_1 \vee \overline{x}_2 \vee x_3)$ and $c_2 = (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$. The canonical satisfying assignment is (1, 1, 0), i.e.
$x_1 = 1$, $x_2 = 1$, $x_3 = 0$.

Step 2 – Build the lobes and clause sinks. Each variable appears once positively and once negatively, so each lobe contains an entry vertex, an exit vertex, one dummy segment on each branch, and one literal-occurrence segment on each branch. That is 10 vertices and 14 arcs per variable. Adding the 4 terminals and 2 clause vertices gives $|V| = 36 = 36$; adding the 4 commodity-1 chain arcs and 2 clause-to-sink arcs gives $|A| = 48 = 48$.

Step 3 – Verify a witness. Commodity 1 uses the lower branch in the lobes of x_1 and x_2 and the upper branch in the lobe of x_3, exactly matching the assignment $(1, 1, 0)$. Clause c_1 is satisfied by x_1, so commodity 2 routes one unit through the positive occurrence segment of x_1 into d_1. Clause c_2 is satisfied by x_2, so a second unit routes through the positive occurrence segment of x_2 into d_2. Both clause-to-sink arcs carry one unit, so the target meets $R_2 = 2 = 2$ ✓. Reading back which lower-branch entry arcs commodity 1 used recovers (1, 1, 0) ✓

Multiplicity: The fixture stores one canonical witness. This source formula has multiple satisfying assignments, and each satisfying clause can choose any satisfied literal occurrence when routing commodity 2.

Rule 3.175: (*k*-SAT (*k*-ary) → Feasible Register Assignment) Sethi's Reduction 3 [5, PO2], [86] builds a DAG with shared-register variable leaf pairs and $p/q/r/\overline{r}$ clause gadgets with cyclic links, plus a preassigned register allocation. The target has $2n + 12m$ vertices, $15m$ arcs, and $K = n + 9m$ registers.
Overhead: `num_vertices = 2 * num_vars + 12 * num_clauses`, `num_arcs = 15 * num_clauses`, `num_registers = num_vars + 9 * num_clauses`.

Proof: *Construction.* For each variable x_k, create two leaf nodes s_k^+, s_k^- sharing register S_k. For each literal occurrence $Y_{i,j}$ in clause C_i, create four nodes $p_{i,j}, q_{i,j}, r_{i,j}, \overline{r}_{i,j}$ with internal arcs $q_{i,j} \to p_{i,j} \to r_{i,j}$ and cyclic links $q_{i,1} \to \overline{r}_{i,2}$, $q_{i,2} \to \overline{r}_{i,3}$, $q_{i,3} \to \overline{r}_{i,1}$. Nodes $r_{i,j}$ and $\overline{r}_{i,j}$ share register $R_{i,j}$. If $Y_{i,j} = x_k$: $r_{i,j} \to s_k^+$ and $\overline{r}_{i,j} \to s_k^-$; if $Y_{i,j} = \overline{x}_k$: swap the attachments.

Correctness. ($\Rightarrow$) If φ is satisfiable, place the truth-selected leaf first for each variable, then unlock each clause gadget starting from a satisfied literal. ($\Leftarrow$) The cyclic links force at least one position j per clause where $r_{i,j}$ appears before $\overline{r}_{i,j}$; shared-register order transfer forces the corresponding literal leaf to appear first, encoding a true literal.

Solution extraction. For each variable x_k, set $\tau(x_k) = 1$ iff s_k^+ appears before s_k^- in the realization. □

Rule 3.176: (*k*-SAT (*k*-ary) → Register Sufficiency) Sethi's Reduction I / Theorem 3.11 [5, PO1], [86] builds a dependency DAG whose register pressure mirrors a literal-assignment phase followed by a clause-verification phase. For a 3-CNF formula with n variables, m clauses, and $b = \max(0, 2n - m)$, the target has $3n^2 + 9n + 4m + b + 4$ vertices, $6n^2 + 19n + 16m + 2b + 1$ arcs, and register bound $K = 3m + 4n +$

$1 + b$.
Overhead: `num_vertices = 3 * num_vars^2 + 9 * num_vars + 4 * num_clauses + register_sufficiency_padding + 4`, `num_arcs = 6 * num_vars^2 + 19 * num_vars + 16 * num_clauses + 2 * register_sufficiency_padding + 1`, `bound = 3 * num_clauses + 4 * num_vars + 1 + register_sufficiency_padding`.

Proof: *Construction.* Let $\varphi = \wedge_{i=1}^{m} C_i$ be a 3-CNF formula over variables $x_1, ..., x_n$, where $C_i = (Y_{i,1} \vee Y_{i,2} \vee Y_{i,3})$. Define $b = \max(0, 2n - m)$. Create node families $A = \{a_j : 1 \le j \le 2n + 1\}$, $B = \{b_j : 1 \le j \le b\}$, $C = \{c_i : 1 \le i \le m\}$, $F = \{f_{i,j} : 1 \le i \le m, 1 \le j \le 3\}$, $M = \{\text{initial}, d, \text{final}\}$, $R = \{r_{k,j} : 1 \le k \le n, 1 \le j \le 2n - 2k + 2\}$, $S = \{s_{k,j} : 1 \le k \le n, 1 \le j \le 2n - 2k + 1\}$, $T = \{t_{k,j} : 1 \le k \le n, 1 \le j \le 2n - 2k + 1\}$, $U = \{u_{k,1}, u_{k,2} : 1 \le k \le n\}$, $W = \{w_k : 1 \le k \le n\}$, $X = \{x_k^+, x_k^- : 1 \le k \le n\}$, and $Z = \{z_k : 1 \le k \le n\}$. Add the ten arc families from Sethi's theorem exactly as stated in the issue: initial depends on every node in $A \cup B \cup F \cup U$, every node in $C \cup R \cup S \cup T \cup W$ depends on initial, final depends on $W \cup X \cup Z \cup \{\text{initial}, d\}$, each variable gadget links x_k^+ and x_k^- to z_k, $u_{k,1}$, $u_{k,2}$, $s_{k,*}$, $t_{k,*}$, $r_{k,*}$, and each clause gadget links c_i to w_n, z_n, its three $f_{i,j}$ nodes, and the literal-lock edges determined by whether $Y_{i,j}$ is positive or negative.

Correctness. ($\Rightarrow$) Suppose φ is satisfiable under assignment τ. Execute Sethi's eight-stage schedule. In the variable phase, the chain gadgets force a choice between the positive and negative side of each variable, and the schedule can be arranged so that by the moment w_n is computed, x_k^+ has appeared iff $\tau(x_k) = 1$. Because every clause has a satisfied literal, the corresponding $f_{i,j}$ node is unlocked during the clause phase, and the lock edges from the opposite literals prevent incompatible clause traversals. Sethi proves that this entire computation uses at most K registers, so the target Register Sufficiency instance is feasible.

($\Leftarrow$) Suppose the target instance has a computation using at most K registers. Stop immediately after w_n is computed. At that snapshot, for each variable gadget, at most one of x_k^+ and x_k^- has been computed. Define $\tau(x_k) = 1$ iff x_k^+ has already been computed. Now assume some clause C_i is false under τ. Then every literal $Y_{i,j}$ is false, so the corresponding literal node has not yet been computed by the w_n snapshot. Consequently each $f_{i,j}$ still has an uncomputed literal predecessor. Sethi's clause-phase invariant leaves no free register between the computation of w_n and the later computation of d, so such a clause node c_i could not be discharged without exceeding K, contradiction. Therefore every clause contains a true literal under τ, and φ is satisfiable.

Solution extraction. Given a target computation ordering, let $t(w_n)$ be the position of w_n. Output $x_k = 1$ exactly when $t(x_k^+) < t(w_n)$. This is the corrected extraction rule: the snapshot is taken at w_n, not z_n, and the sign test uses x_k^+, not x_k^-. □

Example: Two-clause 3-SAT instance ($n = 3$, $m = 2$) reduced to Register Sufficiency

Source: KSatisfiability **Target:** RegisterSufficiency

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to RegisterSufficiency -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 1,1,1
```

Step 1 – Source instance. The canonical source formula is $\varphi = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3)$. The stored satisfying assignment is (1, 1, 1).

Step 2 – Build Sethi's DAG. Here $n = 3$, $m = 2$, and $b = \max(0, 2n - m) = 4$. The construction creates $|A| = 2n + 1 = 7$, $|B| = b = 4$, $|C| = m = 2$, $|F| = 3m = 6$, $|M| = 3$, $|R| = n(n + 1) = 12$, $|S| = |T| = n^2 = 9$, $|U| = 2n = 6$, $|W| = |Z| = n = 3$, and $|X| = 2n = 6$, for a total of $|V| = 70 = 70$ vertices. The target bound is $K = 23 = 23$, and the construction emits $|A| = 152 = 152$ arcs. For clause C_1, the six literal-lock arcs are $(x_1^+, f_{1,1})$, $(x_2^-, f_{1,2})$, $(x_3^+, f_{1,3})$, $(x_1^-, f_{1,2})$, $(x_1^-, f_{1,3})$, and $(x_2^+, f_{1,3})$.

Step 3 – Verify extraction at w_n. The target witness is a computation ordering on 70 vertices. Reading the prefix ending at w_3 marks exactly those variables whose x_k^+ node has already been

computed, which reconstructs (1, 1, 1) ✓. Evaluating the original 3-SAT formula under that assignment returns true ✓

Multiplicity: The fixture stores one canonical satisfying assignment. Different satisfying assignments can induce different valid computation orders in the target DAG.

Rule 3.177: (Feasible Register Assignment → Integer Linear Programming) Direct ILP formulation of the feasible register assignment problem: binary permutation matrix variables, topological ordering constraints, and register-conflict constraints via shared-register ordering indicators.
Overhead: `num_vars = 2 * num_vertices + num_vertices * (num_vertices + -1 * 1) * 2^-1`, `num_constraints = 3 * num_vertices * (num_vertices + -1 * 1) * 2^-1 + 3 * num_vertices + 2 * num_arcs + 2 * num_same_register_pairs`.

Proof: *Construction.* Binary variables $x_{v,t} \in \{0,1\}$ (vertex v at position t). Permutation: each row and column sums to 1. Topological: for arc (u,v), $\sum_t t \cdot x_{v,t} < \sum_t t \cdot x_{u,t}$. Register conflict: for vertices v, w sharing a register, an ordering indicator $b_{v,w}$ with big-M constraints ensures all dependents of the first-computed vertex complete before the second uses the register. Feasibility objective (Value = Or).

Correctness. The ILP is feasible iff a valid evaluation ordering respecting the register assignment exists.

Solution extraction. Read vertex positions from the permutation matrix. □

Rule 3.178: (Register Sufficiency → Integer Linear Programming) Direct ILP formulation of Register Sufficiency: integer evaluation times, latest-use times, binary pair-order selectors, and per-step live-value indicators. For a DAG with n vertices, m arcs, and s sinks, the ILP has $\frac{7n^2+3n}{2}$ variables and $\frac{21n^2+3n}{2} + 2m + s$ constraints.
Overhead: `num_vars = 3 * num_vertices^2 + num_vertices * (num_vertices + -1 * 1) * 2^-1 + 2 * num_vertices`, `num_constraints = 9 * num_vertices^2 + 3 * num_vertices * (num_vertices + -1 * 1) * 2^-1 + 3 * num_vertices + 2 * num_arcs + num_sinks`.

Proof: *Construction.* Let the source DAG use the repository convention that an arc (v,u) means vertex v depends on vertex u. Introduce integer variables $t_v \in \{0,...,n-1\}$ for evaluation positions and $l_v \in \{0,...,n\}$ for latest-use positions. For every unordered vertex pair $\{u,v\}$, add a binary selector $b_{u,v}$ with big-M constraints forcing either $t_u < t_v$ or $t_v < t_u$; since all t_v lie in the interval $\{0,...,n-1\}$, the positions form a permutation. For every dependency arc (v,u), enforce $t_v \geq t_u + 1$ and $l_u \geq t_v$. For every sink vertex (no dependents), set $l_u = n$. For each vertex-step pair (u,s) with $s \in \{0,...,n-1\}$, add binary threshold variables $p_{u,s}$ and $q_{u,s}$ satisfying $p_{u,s} = 1$ iff $t_u \leq s$ and $q_{u,s} = 1$ iff $l_u > s$, plus a binary live indicator $h_{u,s} = p_{u,s} \wedge q_{u,s}$. Finally impose $\sum_u h_{u,s} \leq K$ for every step s.

Correctness. ($\Rightarrow$) Any valid computation ordering of the source DAG yields a feasible ILP solution: assign each t_v to the vertex position in the ordering, each l_v to the latest dependent position (or n for sinks), and derive the binary threshold/live variables from those integers. The dependency constraints hold by topological validity, and the live-count inequalities hold because the source witness uses at most K registers. ($\Leftarrow$) Any feasible ILP solution gives distinct positions t_v, hence a permutation of the vertices, and the arc constraints make that permutation topological. The live indicators $h_{u,s}$ certify exactly which values remain live after step s, so the step constraints prove that no more than K values are simultaneously live. Therefore the extracted ordering is a valid Register Sufficiency witness.

Solution extraction. Return the first n ILP coordinates $(t_0,...,t_{n-1})$ as the vertex evaluation positions. □

Rule 3.179: (Partition → Sequencing Within Intervals) A unit-length enforcer task pinned at $\left[\lfloor \frac{S}{2} \rfloor, \lfloor \frac{S}{2} \rfloor + 1\right)$ splits the timeline into two blocks. A valid schedule exists iff the elements partition into two equal-sum subsets.
Overhead: `num_tasks = num_elements + 1`.

Proof: *Construction.* Let $A = \{a_1, ..., a_n\}$ with $S = \sum a_i$ and $h = \lfloor \frac{S}{2} \rfloor$. Create $n + 1$ tasks: element tasks with $r_i = 0$, $d_i = S + 1$, $p_i = a_i$; enforcer task with $r = h$, $d = h + 1$, $p = 1$.

Correctness. ($\Rightarrow$) A balanced partition places one subset's tasks in $[0, h)$ and the other in $[h + 1, S + 1)$. ($\Leftarrow$) The enforcer at $[h, h + 1)$ splits usable time into two blocks of size $h = \frac{S}{2}$; since element tasks fill both blocks exactly, the assignment gives a balanced partition.

Solution extraction. Task t_i starting at time s_i: assign to subset 0 if $s_i \leq h$, else subset 1. □

Example: $n = 4$ **elements,** $S = 10$

Source: Partition **Target:** SequencingWithinIntervals

```
$ pred create --example Partition -o partition.json
$ pred reduce partition.json --to SequencingWithinIntervals -o bundle.json
$ pred solve bundle.json
$ pred evaluate partition.json --config 1,0,0,1
```

Step 1 – Source instance. The Partition instance has $n = 4$ elements with sizes (1, 2, 3, 4) and total sum $S = 10$. The half-sum is $h = \lfloor S/2 \rfloor = 5$.

Step 2 – Construct tasks. The reduction creates $n + 1 = 5$ tasks. Each element a_i becomes a task with release time $r_i = 0$, deadline $d_i = S + 1 = 11$, and length $p_i = a_i$. An enforcer task is pinned at the midpoint: $r = 5$, $d = 6$, $p = 1$. This enforcer occupies $[h, h + 1)$, splitting the timeline into two blocks of size $h = 5$ each.

Step 3 – Verify a solution. The canonical partition assigns elements to subsets via (1, 0, 0, 1): subset $0 = \{2, 3\}$ (sum 5), subset $1 = \{1, 4\}$ (sum 5). Both subsets sum to $S/2 = 5$ ✓
The target schedule has start-time offsets (6, 0, 2, 7, 0): subset-0 tasks fill $[0, h)$ and subset-1 tasks fill $[h + 1, S + 1)$, with the enforcer at $[h, h + 1)$ ✓

Multiplicity: The fixture stores one canonical witness. Other valid partitions (e.g. swapping the two subsets) exist but are symmetric.

Rule 3.180: (Minimum Vertex Cover (weighted) → Minimum Feedback Arc Set (weighted)) Each vertex v splits into v^{in} and v^{out} joined by an internal arc weighted $w(v)$. Each edge becomes two crossing arcs weighted $M = 1 + \sum_v w(v)$. The optimal FAS never includes crossing arcs; selecting internal arcs for cover vertices breaks every cycle.
Overhead: `num_vertices = 2 * num_vertices`, `num_arcs = num_vertices + 2 * num_edges`.

Proof: *Construction.* Given (G, w) with $G = (V, E)$, $n = |V|$. Build directed graph H on $2n$ nodes. Internal arcs $(v^{\text{in}}, v^{\text{out}})$ with weight $w(v)$. For each $\{u, v\} \in E$: crossing arcs $(u^{\text{out}}, v^{\text{in}})$ and $(v^{\text{out}}, u^{\text{in}})$ with weight $M = 1 + \sum_{v \in V} w(v)$.

Correctness. ($\Rightarrow$) A vertex cover S gives FAS $F = \{(v^{\text{in}}, v^{\text{out}}) : v \in S\}$; every cycle through a crossing arc has at least one internal arc in F. ($\Leftarrow$) Since M exceeds total internal weight, no crossing arc is in the optimal FAS. For each edge $\{u, v\}$, the 4-cycle through both internal and crossing arcs forces at least one internal arc into F.

Solution extraction. Internal arcs at positions $0, ..., n - 1$; the cover is $c[0 : n]$. □

Example: Triangle graph ($n = 3$, $|E| = 3$): VC → FAS via vertex splitting

Source: MinimumVertexCover **Target:** MinimumFeedbackArcSet

```
$ pred create --example MinimumVertexCover/SimpleGraph/i32 -o mvc.json
$ pred reduce mvc.json --to MinimumFeedbackArcSet/i32 -o bundle.json
```

```
$ pred solve bundle.json
$ pred evaluate mvc.json --config 0,1,1
```

Step 1 – Source instance. The source graph G has $n = 3$ vertices and $|E| = 3$ edges: $E = \{(0, 1), (1, 2), (2, 0)\}$ with weights $\boldsymbol{w} = (1, 1, 1)$.

Step 2 – Construction. Each vertex v splits into v^{in} and v^{out}, yielding $2n = 6$ nodes in the target digraph H. Internal arcs $(v^{\text{in}}, v^{\text{out}})$ carry weight $w(v)$: (0, 3) w=1, (1, 4) w=1, (2, 5) w=1. Each undirected edge produces two crossing arcs with weight $M = 1 + \sum w(v) = 4$: (3, 1) w=4, (4, 0) w=4, (4, 2) w=4, (5, 1) w=4, (5, 0) w=4, (3, 2) w=4. Total: 9 arcs (3 internal + 6 crossing).

Step 3 – Verify a solution. The source cover is $C = \{1, 2\}$ (size 2). The target FAS selects arcs at indices $\{1, 2\}$, which are the internal arcs (1, 4), (2, 5) – exactly the internal arcs of cover vertices $v^{\text{in}} \to v^{\text{out}}$ for $v \in C$. No crossing arc (weight $M = 4$) is selected, confirming the optimal FAS uses only internal arcs ✓

Multiplicity: The fixture stores one canonical witness. By symmetry of the triangle, any two-vertex cover is optimal.

Rule 3.181: (*k*-SAT (*k*-ary) → *k*-Clique) Assign one vertex per literal position (j, p); connect vertices from different clauses whose literals are not contradictory. A k-clique selects one consistent true literal per clause.
Overhead: `num_vertices = 3 * num_clauses`, `num_edges = 9 * num_clauses * (num_clauses + -1 * 1) * 2^-1`, `k = num_clauses`.

Proof: *Construction.* Given 3-CNF $\varphi = C_1 \wedge \cdots \wedge C_m$ over n variables, construct $G = (V, E)$ with $V = \{(j, p) : 1 \le j \le m, 1 \le p \le 3\}$, $|V| = 3m$. Edge between (j_1, p_1) and (j_2, p_2) iff $j_1 \neq j_2$ and $\ell_{j_1,p_1} \neq \neg\ell_{j_2,p_2}$. Set $k = m$.

Correctness. ($\Rightarrow$) A satisfying assignment picks one true literal per clause; these vertices form a clique since they span all clauses without contradiction. ($\Leftarrow$) A k-clique has exactly one vertex per clause group; the selected literals are pairwise non-contradictory, defining a satisfying assignment.

Solution extraction. For selected vertex v: clause $j = \lfloor \frac{v}{3} \rfloor$, position $p = v \bmod 3$. If literal $\ell_{j,p} = x_i$ set $x_i = 1$; if $\ell_{j,p} = \neg x_i$ set $x_i = 0$. □

Example: 3-SAT with $m = 2$ clauses, $n = 3$ variables → k-clique on 6 vertices

Source: KSatisfiability **Target:** KClique

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to KClique/SimpleGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 0,0,1
```

Step 1 – Source instance. The 3-CNF formula φ has $m = 2$ clauses over $n = 3$ variables:

$$\varphi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

Step 2 – Construct the conflict graph. Create one vertex per literal position: vertex $3j + p$ represents position p (0-indexed) in clause j, giving $|V| = 3 \cdot 2 = 6$ vertices. Connect (j_1, p_1) and (j_2, p_2) whenever $j_1 \neq j_2$ and the two literals are not contradictory. The resulting graph has $|E| = 7$ edges: $E = \{(0, 4), (0, 5), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5)\}$. Set $k = m = 2$.

Step 3 – Verify a solution. The satisfying assignment $(x_1, x_2, x_3) = (0, 0, 1)$ makes literal x_3 true in clause 0 (position 2, vertex 2) and literal $\overline{x_1}$ true in clause 1 (position 0, vertex 3). The target

configuration $\boldsymbol{x} = (0, 0, 1, 1, 0, 0)$ selects vertices 2 and 3, which form a clique (edge $(2, 3) \in E$ ✓) of size $k = 2$ spanning both clause groups ✓

Multiplicity: The fixture stores one canonical witness. The formula has multiple satisfying assignments; each induces at least one k-clique by choosing one true literal per clause.

Rule 3.182: (*k*-SAT (*k*-ary) → Cyclic Ordering) This $O(n + m)$ reduction [5], [157] represents each variable by a three-element orientation gadget and each clause by five auxiliary elements linked through ten cyclic-ordering triples. For n variables and m clauses it produces $3n + 5m$ target elements and $10m$ triples. *Overhead:* `num_elements = 3 * num_vars + 5 * num_clauses`, `num_triples = 10 * num_clauses`.

Proof: *Construction.* Let φ be a 3-CNF formula with variables $x_1, ..., x_n$ and clauses $C_1, ..., C_m$. For each variable x_i, create three target elements $\alpha_i, \beta_i, \gamma_i$. For a positive literal x_i, define its associated cyclically ordered triple as $(\alpha_i, \beta_i, \gamma_i)$; for a negative literal $\neg x_i$, define it as $(\alpha_i, \gamma_i, \beta_i)$. For each clause $C_\nu = (\ell_1 \vee \ell_2 \vee \ell_3)$, write the three associated literal triples as (a, b, c), (d, e, f), and (g, h, i). Add five fresh auxiliary elements $j_\nu, k_\nu, l_\nu, m_\nu, n_\nu$ and the ten cyclic-ordering triples

$$(a, c, j_\nu), (b, j_\nu, k_\nu), (c, k_\nu, l_\nu), (d, f, j_\nu), (e, j_\nu, l_\nu), (f, l_\nu, m_\nu), (g, i, k_\nu), (h, k_\nu, m_\nu), (i, m_\nu, n_\nu), (n_\nu, m_\nu, l_\nu).$$

Correctness. ($\Rightarrow$) Let S be a satisfying assignment of φ. For each variable, exactly one of the two opposite orientations $(\alpha_i, \beta_i, \gamma_i)$ and $(\alpha_i, \gamma_i, \beta_i)$ is derived by any cyclic order; interpret the literal made true by S as the one whose associated orientation is *not* derived. In every clause at least one literal is true, and Galil–Megiddo's clause gadget lemma shows that the ten triples above are then consistent with the three literal orientations for that clause [157]. Because different clauses use disjoint auxiliary element sets, the per-clause cyclic orders combine into one global cyclic order satisfying all target triples. ($\Leftarrow$) Conversely, let a cyclic ordering satisfy every target triple. For each variable x_i, put $x_i = 1$ iff $(\alpha_i, \beta_i, \gamma_i)$ is *not* derived. If some clause had all three literals false under this rule, then all three associated literal orientations would be derived. The same clause gadget lemma implies that the ten triples for that clause would then be inconsistent, contradicting feasibility. Hence every clause contains a true literal, so the extracted assignment satisfies φ.

Variable mapping. Positive literal x_i is read through the orientation $(\alpha_i, \beta_i, \gamma_i)$, while negative literal $\neg x_i$ is read through the reversed orientation $(\alpha_i, \gamma_i, \beta_i)$.

Solution extraction. Given a target permutation f, set $x_i = 1$ iff $(f(\alpha_i), f(\beta_i), f(\gamma_i))$ is *not* in cyclic order; otherwise set $x_i = 0$. □

Example: Single-clause 3-SAT reduced to 14 cyclic-order elements

Source: KSatisfiability **Target:** CyclicOrdering

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to CyclicOrdering -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 1,1,1
```

Step 1 – Source instance. The canonical formula is $\varphi = (x_1 \vee x_2 \vee x_3)$ with $n = 3$ variables and $m = 1$ clause. The stored satisfying assignment is (1, 1, 1), so every literal in the clause is true.

Step 2 – Create variable and clause elements. The reduction introduces three variable elements per Boolean variable, giving variable triples $(\alpha_1, \beta_1, \gamma_1) = (0, 1, 2)$, $(\alpha_2, \beta_2, \gamma_2) = (3, 4, 5)$, and $(\alpha_3, \beta_3, \gamma_3) = (6, 7, 8)$. The single clause adds five auxiliary elements $(j, k, l, m, n) = (9, 10, 11, 12, 13)$, so the target has $3n + 5m = 14$ elements total.

Step 3 – Emit the ten cyclic-ordering triples. Because the clause literals are (x_1, x_2, x_3), the literal-orientation triples are the forward variable triples. The target constraints are exactly (0, 2, 9),

(1, 9, 10), (2, 10, 11), (3, 5, 9), (4, 9, 11), (5, 11, 12), (6, 8, 10), (7, 10, 12), (8, 12, 13), (13, 12, 11), so $|\Delta| = 10 = 10m$.

Step 4 – Verify a solution. The target witness permutation is (0, 11, 1, 9, 12, 10, 6, 13, 7, 2, 3, 4, 8, 5). It satisfies all ten cyclic-order constraints ✓. For each variable triple, the forward orientation $(\alpha_i, \beta_i, \gamma_i)$ is *not* derived by this permutation, so extraction returns (1, 1, 1), which satisfies φ ✓

Multiplicity: The fixture stores one canonical witness. Any cyclic permutation of the target order is also valid, and other satisfying assignments of φ induce additional valid cyclic orders.

Rule 3.183: (*k*-SAT (*k*-ary) → Preemptive Scheduling) Ullman's reduction first builds a variable-capacity unit-task scheduling instance for 3-SAT, then pads each time slot with chained filler jobs so a fixed number of processors simulates the desired capacity profile. Because every task has length 1, preemption is irrelevant: the resulting instance is already a valid preemptive scheduling instance whose optimal makespan is at most $T = n + 3$ iff the formula is satisfiable [5], [118].
Overhead: `num_tasks = (2 * num_vars + 2 + 6 * num_clauses + sqrt((2 * num_vars + 2 + -1 * 6 * num_clauses)^2)) * 2^-1 * (num_vars + 3)`, `num_processors = (2 * num_vars + 2 + 6 * num_clauses + sqrt((2 * num_vars + 2 + -1 * 6 * num_clauses)^2)) * 2^-1`, `d_max = (2 * num_vars + 2 + 6 * num_clauses + sqrt((2 * num_vars + 2 + -1 * 6 * num_clauses)^2)) * 2^-1 * (num_vars + 3)`.

Proof: *Construction.* Let φ be a 3-CNF formula with variables $x_1, ..., x_n$ and clauses $C_1, ..., C_m$. Create unit jobs $x_{i,j}$ and $\overline{x}_{i,j}$ for $1 \le i \le n$ and $0 \le j \le n$, plus forcing jobs $y_i, \overline{y}_i$, and clause jobs $D_{r,s}$ for $1 \le r \le m$, $1 \le s \le 7$. Add chain precedences $x_{i,j} \prec x_{i,j+1}$ and $\overline{x}_{i,j} \prec \overline{x}_{i,j+1}$, and branching precedences $x_{i,i-1} \prec y_i$, $\overline{x}_{i,i-1} \prec \overline{y}_i$. Set $T = n + 3$ and slot capacities $c_0 = n$, $c_1 = 2n + 1$, $c_t = 2n + 2$ for $2 \le t \le n$, $c_{n+1} = m + n + 1$, and $c_{n+2} = 6m$. For each clause $C_r = (\ell_1 \vee \ell_2 \vee \ell_3)$ and each nonzero bit pattern $b \in \{1, ..., 7\}$, create clause job $D_{r,b}$. Its predecessors are the three chain endpoints chosen according to the bits of b: for literal position k, use the endpoint of ℓ_k when bit k is 1 and of $\neg\ell_k$ when bit k is 0. This makes exactly one clause job per clause ready one slot earlier when the clause is satisfied.

To convert the variable-capacity instance to fixed processors, let $p = \max(2n + 2, 6m)$. For every slot t, add $p - c_t$ filler jobs and impose complete-bipartite precedences from every filler at slot t to every filler at slot $t + 1$. Keep every task length equal to 1 and use p processors. The total work is exactly pT, so any schedule of makespan at most T must saturate every slot and therefore realizes the intended capacities.

Correctness. ($\Rightarrow$) Given a satisfying assignment, place exactly one of $x_{i,0}, \overline{x}_{i,0}$ at slot 0 for each variable, propagate the two chains forward one step at a time, schedule the forcing jobs immediately after their branch points, and place the unique matching clause job for each clause at slot $n + 1$ (all other clause jobs at slot $n + 2$). The filler jobs occupy the remaining $p - c_t$ processor positions in slot t, so the schedule finishes by time $T = n + 3$. ($\Leftarrow$) Conversely, if the constructed instance has makespan at most T, then every slot is full and the filler chains force exactly $p - c_t$ filler jobs into slot t, leaving precisely c_t non-filler positions. Ullman's capacity argument then applies: at slot 0 exactly one of $x_{i,0}, \overline{x}_{i,0}$ is chosen per variable, this choice propagates consistently through the chains, and the availability of one clause job per clause at slot $n + 1$ implies each clause has a satisfied literal. Hence the extracted assignment satisfies φ.

Solution extraction. In the binary schedule encoding, inspect the row for each starter job $x_{i,0}$. Set $x_i = 1$ iff that row has its single 1 in column 0; otherwise set $x_i = 0$. □

Example: 3-SAT with $n = 3$ variables and $m = 1$ clause → unit-task preemptive schedule on 48 jobs and 8 processors

Source: KSatisfiability **Target:** PreemptiveScheduling

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to PreemptiveScheduling -o bundle.json
```

```
$ pred solve bundle.json
$ pred evaluate ksat.json --config 0,0,1
```

Step 1 – Source instance. The formula is $\varphi = (x_1 \vee x_2 \vee x_3)$ with satisfying assignment $(x_1, x_2, x_3) = (0, 0, 1)$.

Step 2 – Build Ullman's unit-task gadgets. For $n = 3$, the reduction creates $2n(n+1) = 24$ chain jobs $x_{i,j}, \overline{x}_{i,j}$, $2n = 6$ forcing jobs $y_i, \overline{y}_i$, and $7m = 7$ clause jobs $D_{r,s}$. The slot capacities are $(3, 7, 8, 8, 5, 6) = (3, 7, 8, 8, 5, 6)$. We realize these capacities with $p = \max(2n + 2, 6m) = 8$ processors and $F = 11$ filler jobs, giving 48 total unit jobs. In this example the filler counts are $(5, 1, 0, 0, 3, 2)$.

Step 3 – Verify a schedule. The witness schedule has exactly $p = 8$ jobs in each of the $T = 6$ slots: (8, 8, 8, 8, 8, 8). The positive chain starters $x_{1,0}, x_{2,0}, x_{3,0}$ are jobs $0, 8, 16$, placed at slots $(1, 1, 0) = (1, 1, 0)$, so extraction reads $(0, 0, 1)$ back from slot 0. The clause-pattern jobs are indices $30, ..., 36$; their slots are (4, 5, 5, 5, 5, 5, 5), so exactly one clause job is promoted to slot $n + 1 = 4$ and the remaining six sit at slot $n + 2 = 5$.

Multiplicity: The fixture stores one canonical witness. Other satisfying assignments induce different slot-0 choices for the variable chains and therefore different valid schedules meeting the same threshold.

Rule 3.184: (*k*-SAT (*k*-ary) → Timetable Design) This polynomial-time implementation[67] realizes the NP-hardness of Timetable Design [21] by normalizing the formula and compiling a constrained edge-coloring instance into a timetable. It uses $4L$ periods in the worst case, where L is the source literal count, and creates $O(L^2)$ craftsmen and tasks with binary requirements.
Overhead: `num_periods = 4 * num_literals`, `num_craftsmen = 64 * num_literals^2 + 32 * num_literals + 1`, `num_tasks = 64 * num_literals^2 + 32 * num_literals + 1`.

Proof: *Construction.* Given a 3-CNF formula φ, first eliminate pure literals and fix their truth values. For every remaining variable with more than three literal occurrences, apply Tovey's cloning trick: replace each occurrence by a fresh variable and add a cycle of 2-clauses $(y_i \vee \overline{y}_{i+1})$ so that all clones must agree. Let the transformed variables be $y_1, ..., y_q$; by construction each y_i appears at most twice positively and at most twice negatively.

Reserve four colors for each transformed variable: $N_{i,2}, N_{i,1}, P_{i,2}, P_{i,1}$. Build a bipartite core graph with one central vertex. For every y_i, add the six 2-list edges used by the implementation, with allowed color pairs $(P_{i,1}, N_{i,2})$, $(P_{i,2}, P_{i,1})$, $(P_{i,1}, P_{i,2})$, $(P_{i,2}, N_{i,1})$, $(N_{i,2}, P_{i,2})$, and $(N_{i,1}, P_{i,1})$. This variable gadget has exactly two proper colorings, which represent $y_i = 1$ and $y_i = 0$. For every transformed clause, add one clause edge from the center to a fresh clause vertex whose allowed colors are exactly the colors assigned to that clause's literal occurrences.

Next compile the list-edge-coloring instance to an ordinary edge-coloring instance with blocked colors on vertices. Every 2-list edge is replaced by a three-edge path whose two internal vertices block all colors except the two allowed ones. A 1-list or 3-list clause edge stays direct, and the clause endpoint blocks every disallowed color. Finally encode colors as timetable periods, left-side vertices as craftsmen, right-side vertices as tasks, and every graph edge as one unit requirement between its endpoints. Whenever color h is blocked at a vertex u, add a one-period blocker craftsman or task that is available only in period h and is forced to match u once. This dummy assignment consumes the unique slot of u in period h, so no adjacent core edge can use that color.

Correctness. (⇒) A satisfying assignment extends to the cloned variables and chooses one of the two color patterns in each variable gadget. Because every clause has a satisfied literal, its clause edge can take the corresponding literal color. The path gadgets propagate the chosen colors on every 2-list edge,

[67] The repository encodes the reduction via an explicit bounded-occurrence list-edge-coloring gadget chain rather than reproducing the original three-period presentation of [21] verbatim. The implementation is still a many-one reduction to the general Timetable Design model.

and the blocker pairs occupy exactly the forbidden periods. Translating each edge color into its period yields a feasible timetable satisfying availability, exclusivity, and exact requirements. ($\Leftarrow$) In any feasible timetable, each required pair is scheduled in exactly one period, so every core edge receives a unique color. The blocker pairs forbid exactly the blocked colors, and every expanded path forces its represented edge to use one of the two colors in its list. Hence each variable gadget collapses to one of the two mirror patterns, defining a Boolean value for each transformed variable. A clause edge can then be colored only by one of its literal colors, so some literal in every clause matches the extracted variable pattern. Projecting clone values back to the original variables and reinstating the fixed pure-literal assignments yields a satisfying assignment of φ.

Variable mapping. Pure literals removed during preprocessing are stored as fixed source assignments. Every remaining source variable is represented either by one transformed variable or by a cycle of clone variables introduced by the bounded-occurrence step; the added 2-clauses appear as ordinary clause gadgets and force all clones of one source variable to agree. Periods are indexed by the four colors attached to each transformed variable.

Solution extraction. The implementation reads the period of the distinguished `vb` edge in each variable gadget. Color $N_{i,2}$ is interpreted as truth value 1, and color $P_{i,2}$ as truth value 0. The recovered transformed assignment is then projected back to the original variables, with the fixed pure-literal values reinserted.□

Example: Two-clause satisfiable formula reduced to a timetable gadget instance

Source: KSatisfiability **Target:** TimetableDesign

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to TimetableDesign -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 1,0
```

Step 1 – Source instance. The canonical formula has $n = 2$ variables and $m = 2$ clauses:

$$\varphi = (x_1 \vee x_2 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_2)$$

The stored satisfying assignment is $(x_1, x_2) = (1, 0)$.

Step 2 – Normalize and assign periods. The source has $L = 6$ literal occurrences. Because x_2 appears four times, the bounded-occurrence preprocessing replaces it by a cycle of occurrence variables, so the normalized instance uses 5 transformed variables and therefore $4q = 20$ timetable periods/colors.

Step 3 – Compile the gadget graph. The list-edge-coloring gadget becomes a timetable with 696 craftsmen, 662 tasks, and 1362 binary requirements. Among these rows, 646 craftsmen and 612 tasks are one-period blockers used only to forbid colors on internal gadget vertices.

Step 4 – Verify a solution. The target witness has 1362 scheduled pairs, exactly matching the 1362 nonzero requirements. Hence each required craftsman-task pair is scheduled once, every blocker occupies its designated period, and `pred evaluate` accepts the timetable. Reading back the distinguished variable-gadget periods recovers (1, 0), which satisfies both clauses ✓

Multiplicity: The fixture stores one canonical witness. Different satisfying assignments, or different satisfying choices for the clause-edge colors, can induce distinct feasible timetables for the same formula.

Rule 3.185: (*k*-SAT (*k*-ary) → Acyclic Partition (weighted)) This composed reduction realizes 3-SAT → Acyclic Partition via the witness-preserving chain 3-SAT → Subset Sum → Partition → Acyclic Partition. The first two stages are the classical Sipser digit encoding [145] and Garey–Johnson padding [5]; the final stage embeds the balanced partition into a bipartite source–sink digraph whose two-block quotient is automatically acyclic. For n variables and m clauses, the target has $2n + 2m + 3$ vertices and $4n + 4m + 2$ arcs.

Overhead: `num_vertices = 2 * num_vars + 2 * num_clauses + 3`, `num_arcs = 4 * num_vars + 4 * num_clauses + 2`.

Proof: *Construction.* Given a 3-CNF formula φ with n variables and m clauses:

(i) Apply the Sipser digit-encoding reduction (see KSatisfiability → SubsetSum) to obtain a Subset Sum instance with $2n + 2m$ elements and target T.

(ii) Apply the Subset Sum → Partition reduction (see SubsetSum → Partition) to obtain a Partition instance with at most $2n + 2m + 1$ elements and total Σ.

(iii) For each Partition element a_i, create one item vertex with weight $2a_i$. Add a source vertex s and a sink vertex t, each with weight $\Sigma + 1$. Create arcs (s, v_i) and (v_i, t) for every item vertex v_i, all with unit cost. Set the weight bound $B = \Sigma + 1 + \Sigma - (\Sigma \bmod 2)$ and the arc-cost bound $K =$ number of items.

Correctness. ($\Rightarrow$) A satisfying assignment of φ yields a Subset Sum solution, which yields a balanced Partition, which splits the item vertices into two groups of equal total size. Placing one group with s and the other with t gives a two-block partition. Each block's weight is $\Sigma + 1 + \Sigma \leq B$. The arcs from s to the opposite group and from the opposite group to t form the cut, and the quotient digraph s-block → t -block is acyclic. ($\Leftarrow$) Any feasible acyclic partition must separate s and t (otherwise the digon $s \to v \to t$ and $s \to v' \to t$ would create a quotient cycle). The weight bound forces the doubled item sizes in each block to be nearly balanced, recovering a balanced Partition. Reversing the Subset Sum and 3-SAT stages then yields a satisfying assignment.

Solution extraction. Identify the block containing s and the block containing t. Assign each item vertex to Partition side 0 or 1 according to whether it shares a block with t. Reverse the Subset Sum → Partition extraction, then reverse the 3-SAT → Subset Sum extraction. □

Example: 3-SAT with $n = 1$ variables, $m = 1$ clause → acyclic partition on 7 vertices

Source: KSatisfiability **Target:** AcyclicPartition

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to AcyclicPartition/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 1
```

Step 1 — Source instance. The canonical formula has $n = 1$ variable and $m = 1$ clause. The stored satisfying assignment is (1).

Step 2 — Compose the three-stage chain. The reduction composes 3-SAT → Subset Sum → Partition → Acyclic Partition. First, the Sipser digit-encoding produces a Subset Sum instance with $2n + 2m$ elements. Second, the Subset Sum → Partition padding appends at most one element. Third, the Partition → Acyclic Partition gadget builds a bipartite digraph: for each of the Partition elements, create one item vertex; add a source vertex and a sink vertex, with arcs from source to every item vertex and from every item vertex to sink. The resulting digraph has 7 vertices and 10 arcs. Vertex weights are doubled element sizes for items and $(\Sigma + 1)$ for the two endpoints; the weight bound is $\Sigma + 1 + \Sigma - (\Sigma \bmod 2)$ where Σ is the Partition total, and the arc-cost bound equals the number of items.

Step 3 — Verify a solution. The target witness (1, 0, 1, 1, 0, 0, 1) partitions the 7 vertices into two blocks. The source and sink land in different blocks, ensuring the quotient digraph is acyclic. The item vertices split so that the doubled sizes on each side, together with the endpoint weight, respect the weight cap. Extracting back through Partition and Subset Sum recovers (1), which satisfies the formula ✓

Multiplicity: The fixture stores one canonical witness. Other satisfying assignments of the source formula induce different balanced partitions of the item vertices.

Rule 3.186: (Hamiltonian Circuit → Biconnectivity Augmentation (weighted)) Start with the edgeless graph on n vertices. Price original edges at cost 1 and non-edges at cost 2. A budget-n augmentation is achievable iff G has a Hamiltonian circuit (the only way to biconnect with n weight-1 edges is a Hamiltonian cycle). *Overhead:* `num_vertices = num_vertices, num_edges = 0, num_potential_edges = num_vertices * (num_vertices + -1 * 1) * 2^-1`.

Proof: *Construction.* Given $G = (V, E)$ with $n = |V|$, let $H = (V, \emptyset)$. For every pair $\{u, v\}$: potential edge with weight 1 if $\{u, v\} \in E$, else weight 2. Budget $B = n$.

Correctness. ($\Rightarrow$) A Hamiltonian circuit selects n weight-1 edges forming a 2-connected cycle, cost $= n$. ($\Leftarrow$) Budget n with $\geq n$ edges required (degree ≥ 2) forces exactly n weight-1 edges (all from E), which must form a Hamiltonian cycle.

Solution extraction. Walk the unique cycle from vertex 0 through selected edges to recover the circuit order. □

Example: 4-cycle graph ($n = 4$, $|E| = 4$): HC → biconnectivity augmentation

Source: HamiltonianCircuit **Target:** BiconnectivityAugmentation

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to BiconnectivityAugmentation/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2,3
```

Step 1 – Source instance. The source graph G has $n = 4$ vertices and $|E| = 4$ edges: $E = \{(0, 1), (1, 2), (2, 3), (3, 0)\}$. This is a 4-cycle, which admits a Hamiltonian circuit.

Step 2 – Construction. Start with the edgeless graph $H = (V, \emptyset)$ on $n = 4$ vertices. For each pair $\{u, v\}$, create a potential edge with weight 1 if $\{u, v\} \in E$ and weight 2 otherwise. This yields 6 potential edges: {0, 1} (w=1), {0, 2} (w=2), {0, 3} (w=1), {1, 2} (w=1), {1, 3} (w=2), {2, 3} (w=1). The budget is $B = 4 = n$.

Step 3 – Verify a solution. The canonical Hamiltonian circuit visits vertices in order (0, 1, 2, 3). The target configuration $\boldsymbol{x} = (1, 0, 1, 1, 0, 1)$ selects potential edges {0, 1}, {0, 3}, {1, 2}, {2, 3} — exactly the $n = 4$ cycle edges, each of weight 1, for a total cost of $4 = n = B$ ✓

Multiplicity: The fixture stores one canonical witness. The 4-cycle has 3 distinct Hamiltonian circuits ($(n-1)!/2$ directed cycles up to reversal).

Rule 3.187: (Hamiltonian Circuit → Strong Connectivity Augmentation (weighted)) Start with the empty digraph on n vertices. Weight-1 candidate arcs correspond to edges of G; weight-2 arcs for non-edges. A budget-n augmentation that achieves strong connectivity must select exactly n weight-1 arcs forming a directed Hamiltonian cycle.
Overhead: `num_vertices = num_vertices, num_arcs = 0, num_potential_arcs = num_vertices * (num_vertices + -1 * 1)`.

Proof: *Construction.* Given $G = (V, E)$ with $n = |V|$. Build $D = (V, \emptyset)$. For every ordered pair (u, v) with $u \neq v$: candidate arc with weight 1 if $\{u, v\} \in E$, else weight 2. Budget $B = n$.

Correctness. ($\Rightarrow$) A Hamiltonian circuit gives n directed arcs of weight 1 forming a strongly-connected cycle. ($\Leftarrow$) Strong connectivity needs $\geq n$ arcs; budget n forces all weight 1, hence all from E, forming a single n-cycle.

Solution extraction. Follow unique successors from vertex 0 to recover the Hamiltonian permutation. □

Example: 4-cycle ($n = 4$): HC to budget-4 SCA

Source: HamiltonianCircuit **Target:** StrongConnectivityAugmentation

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to StrongConnectivityAugmentation/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2,3
```

Step 1 – Source instance. The source graph is the cycle on 4 vertices with edges $(0,1)$, $(1,2)$, $(2,3)$, $(3,0)$. The canonical Hamiltonian-circuit witness is the vertex permutation [0, 1, 2, 3].

Step 2 – Construction. Start with the empty digraph $D = (V, \emptyset)$ on 4 vertices. Generate all 12 ordered pairs as candidate arcs: 8 weight-1 arcs $(0 \to 1)$, $(0 \to 3)$, $(1 \to 0)$, $(1 \to 2)$, $(2 \to 1)$, $(2 \to 3)$, $(3 \to 0)$, $(3 \to 2)$ corresponding to source edges (both orientations), and 4 weight-2 arcs $(0 \to 2)$, $(1 \to 3)$, $(2 \to 0)$, $(3 \to 1)$ for non-edges. Budget $B = 4$.

Step 3 – Verify a solution. The target configuration [1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0] selects arcs $(0 \to 1)$, $(1 \to 2)$, $(2 \to 3)$, $(3 \to 0)$, all weight 1. Total cost $= 4 \times 1 = 4 = B$ ✓. These 4 arcs form a single directed cycle visiting every vertex, so the augmented digraph is strongly connected. Extracting the circuit: follow successors from vertex 0 to recover [0, 1, 2, 3] ✓

Multiplicity: The fixture stores one canonical witness. For the 4-cycle there are $4 \times 2 = 8$ Hamiltonian-circuit permutations (choice of start vertex and direction), each yielding a distinct set of directed arcs.

Rule 3.188: (Hamiltonian Circuit → Stacker Crane) Each vertex v_i splits into $(v_i^{\text{in}}, v_i^{\text{out}})$ with a mandatory directed arc of length 1. Undirected connector edges of length 1 encode original graph edges. A tour of cost $2n$ exists iff a Hamiltonian circuit exists.
Overhead: `num_vertices = 2 * num_vertices`, `num_arcs = num_vertices`, `num_edges = 2 * num_edges`.

Proof: *Construction.* Given $G = (V, E)$ with $n = |V|$. Create $2n$ vertices: $v_i^{\text{in}} = 2i$, $v_i^{\text{out}} = 2i + 1$. Add n mandatory arcs $(v_i^{\text{in}}, v_i^{\text{out}})$ of length 1. For each $\{v_i, v_j\} \in E$: connector edges $\left(v_i^{\text{out}}, v_j^{\text{in}}\right)$ and $\left(v_j^{\text{out}}, v_i^{\text{in}}\right)$ of length 1.

Correctness. ($\Rightarrow$) A Hamiltonian circuit gives a tour serving arcs in circuit order, each inter-arc hop using one connector edge, total cost $2n$. ($\Leftarrow$) Cost $2n$ with n arcs of cost 1 leaves exactly n connector hops of cost 1 each; single-hop connections correspond to edges of G.

Solution extraction. The arc service permutation directly encodes the Hamiltonian circuit vertex order.□

Example: Cycle C_4 ($n = 4$): vertex splitting to Stacker Crane

Source: HamiltonianCircuit **Target:** StackerCrane

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to StackerCrane -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2,3
```

Step 1 – Source instance. The canonical source fixture is the cycle C_4 on vertices $\{0, \ldots, 3\}$ with 4 edges: $(0,1)$, $(1,2)$, $(2,3)$, $(3,0)$. The stored Hamiltonian-circuit witness is the permutation [0, 1, 2, 3].

Step 2 – Construction. Each vertex v_i splits into $v_i^{\text{in}} = 2i$ and $v_i^{\text{out}} = 2i + 1$, giving $2 \cdot 4 = 8$ vertices. The reduction creates 4 mandatory arcs: $(0 \to 1)$, $(2 \to 3)$, $(4 \to 5)$, $(6 \to 7)$, each of length 1. For each source edge, two undirected connector edges of length 1 are added, giving $2 \cdot 4 = 8$ connector edges: $\{1,2\}$, $\{3,0\}$, $\{3,4\}$, $\{5,2\}$, $\{5,6\}$, $\{7,4\}$, $\{7,0\}$, $\{1,6\}$.

Step 3 – Verify a solution. The stored target configuration [0, 1, 2, 3] is a permutation of arcs. Following this order: arc 0 serves $(0 \to 1)$, then a connector edge leads to the next arc, and so on. The

tour traverses 4 arcs (cost 4) and 4 connector edges (cost 4), for total cost $2 \cdot 4 = 8$. Recovering the source witness: arc i corresponds to vertex i, so the permutation [0, 1, 2, 3] is the Hamiltonian circuit ✓

Multiplicity: The fixture stores one canonical witness. For C_4 there are $4 \times 2 = 8$ directed Hamiltonian circuits (choice of start vertex and direction), each yielding a distinct arc-service permutation.

Rule 3.189: (Hamiltonian Circuit → Rural Postman (weighted)) Each vertex splits into two copies connected by a required edge. Original graph edges become connector edges. A minimum-cost tour of cost $2n$ traversing all required edges exists iff G has a Hamiltonian circuit.
Overhead: `num_vertices = 2 * num_vertices, num_edges = num_vertices + 2 * num_edges, num_required_edges = num_vertices`.

Proof: *Construction.* Given $G = (V, E)$ with $n = |V|$. Build H with $2n$ vertices: $v_i^a = 2i$, $v_i^b = 2i + 1$. Required edges $\{v_i^a, v_i^b\}$ of weight 1 (n total). For each $\{v_i, v_j\} \in E$: connector edges $\{v_i^b, v_j^a\}$ and $\{v_j^b, v_i^a\}$ of weight 1.

Correctness. ($\Rightarrow$) A Hamiltonian circuit $(v_{p_0}, ..., v_{p_{n-1}})$ gives a tour traversing required edges and single connector edges, total $2n$. ($\Leftarrow$) Each required edge costs 1 (n total); remaining budget n for connectors forces single-hop connections corresponding to edges of G.

Solution extraction. Identify used connector edges and follow the successor map from vertex 0. □

Example: Cycle C_3 ($n = 3$): vertex splitting to Rural Postman

Source: HamiltonianCircuit **Target:** RuralPostman

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to RuralPostman/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2
```

Step 1 – Source instance. The canonical HC instance is a cycle C_3 with $n = 3$ vertices and $|E| = 3$ edges. The stored witness is the permutation (0, 1, 2).

Step 2 – Construction. Each vertex splits into (v_i^a, v_i^b), producing $2n = 6$ vertices. The target graph has 9 edges: 3 required edges (one per source vertex) and 6 connector edges (two per source edge). All edge lengths are 1.

Step 3 – Verify a solution. The target solution assigns edge multiplicities (1, 1, 1, 1, 0, 1, 0, 0, 1). The tour traverses all 3 required edges plus 3 connector edges, for total cost $= 6 = 2n$ ✓.

Multiplicity: The fixture stores one canonical witness. The 3-cycle has 3 rotations × 2 reflections = 6 directed Hamiltonian circuits.

Rule 3.190: (Maximum Independent Set (weighted) → Integral Flow with Bundles) Each vertex v_i maps to a flow unit through an intermediate node; edge-bundle constraints cap combined outflow of adjacent pairs at 1, so feasible flow of value k exists iff an independent set of size $\geq k$ exists.
Overhead: `num_vertices = num_vertices + 2, num_arcs = 2 * num_vertices, num_bundles = num_edges + num_vertices`.

Proof: *Construction.* Directed graph on $n + 2$ nodes: source s, intermediates $w_0, ..., w_{n-1}$, sink t. Arcs $a_i^{\text{in}} = (s, w_i)$ (index $2i$) and $a_i^{\text{out}} = (w_i, t)$ (index $2i + 1$). Edge bundles $\{a_i^{\text{out}}, a_j^{\text{out}}\}$ with capacity 1 for each $\{v_i, v_j\} \in E$. Vertex bundles $\{a_i^{\text{in}}, a_i^{\text{out}}\}$ with capacity 2. Flow requirement $R = 1$.

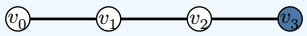
Correctness. ($\Rightarrow$) An independent set S gives flow $f_i = 1$ for $v_i \in S$; edge bundles satisfied since S is independent. ($\Leftarrow$) Feasible flow gives $f_i \in \{0, 1\}$; edge bundles force $\{v_i : f_i = 1\}$ to be independent.

Solution extraction. Vertex v_i in independent set iff target solution at arc index $2i+1$ is positive. □

Example: Path graph P_4 ($n = 4$, $|E| = 3$)

Source: MaximumIndependentSet **Target:** IntegralFlowBundles

```
$ pred create --example MaximumIndependentSet/SimpleGraph/i32 -o mis.json
$ pred reduce mis.json --to IntegralFlowBundles -o bundle.json
$ pred solve bundle.json
$ pred evaluate mis.json --config 0,0,0,1
```



Step 1 – Source instance. The path graph P_4 has $n = 4$ vertices and edges {(0,1), (1,2), (2,3)}, with unit weights (1, 1, 1, 1).

Step 2 – Build the flow network. The reduction creates a directed graph with 6 nodes: source $s = 0$, intermediates $w_0, ..., w_3$, and sink $t = 5$. There are 8 arcs ($2n = 8$): for each vertex v_i, arc $a_i^{\text{in}} = (s, w_i)$ at index $2i$ and $a_i^{\text{out}} = (w_i, t)$ at index $2i + 1$.

Step 3 – Create bundles. There are 7 bundles total. For each original edge $\{v_i, v_j\} \in E$, an edge bundle $\{a_i^{\text{out}}, a_j^{\text{out}}\}$ with capacity 1 enforces that at most one endpoint is selected (3 edge bundles). For each vertex v_i, a vertex bundle $\{a_i^{\text{in}}, a_i^{\text{out}}\}$ with capacity 2 links the in/out arcs (4 vertex bundles). Bundle capacities: (1, 1, 1, 2, 2, 2, 2). Flow requirement $R = 1$.

Step 4 – Verify a solution. The canonical IS selects vertices $\{v_3\}$ (config (0, 0, 0, 1)). Each selected vertex v_i contributes flow 1 on arcs a_i^{in} and a_i^{out}, giving target config (0, 0, 0, 0, 0, 0, 1, 1). The total flow equals the IS size (1). Every edge bundle is satisfied because no two adjacent vertices are both selected, and vertex bundles are satisfied with capacity 2 ≥ individual flow of 1.

Multiplicity: The fixture stores one canonical witness. The path P_4 admits larger independent sets (e.g., $\{v_0, v_2\}$ or $\{v_0, v_3\}$), but the canonical witness suffices to demonstrate the reduction.

Rule 3.191: (Hamiltonian Circuit → Quadratic Assignment) Position-adjacency encoded in cost matrix C (directed cycle on positions), graph-adjacency in distance matrix D (1 for edges, $\omega = n + 1$ for non-edges). QAP optimum equals n iff a Hamiltonian circuit exists.
Overhead: `num_facilities = num_vertices`, `num_locations = num_vertices`.

Proof: *Construction.* Let $G = (V, E)$ with $n = |V|$ and $\omega = n + 1$. Cost matrix: $c[i][j] = 1$ if $j \equiv i + 1 \pmod n$, else 0. Distance matrix: $d[k][l] = 0$ if $k = l$; 1 if $\{k, l\} \in E$; ω otherwise.

Correctness. ($\Rightarrow$) A Hamiltonian circuit $v_0, ..., v_{n-1}$ with $\gamma(i) = v_i$ gives cost n (each consecutive pair is an edge). ($\Leftarrow$) Cost n forces $d[\gamma(i), \gamma((i+1) \bmod n)] = 1$ for all i, meaning all consecutive pairs are edges.

Solution extraction. The QAP permutation γ is the Hamiltonian circuit visit order directly. □

Example: Cycle graph C_4 ($n = 4$, $|E| = 4$)

Source: HamiltonianCircuit **Target:** QuadraticAssignment

```
$ pred create --example HamiltonianCircuit/SimpleGraph -o hc.json
$ pred reduce hc.json --to QuadraticAssignment -o bundle.json
$ pred solve bundle.json
$ pred evaluate hc.json --config 0,1,2,3
```

Step 1 – Source instance. The graph G has $n = 4$ vertices and edges {(0,1), (1,2), (2,3), (3,0)}, forming a cycle C_4. The penalty weight is $\omega = n + 1 = 5$.

Step 2 – Construction. The cost matrix C encodes a directed cycle on positions: $c[i][(i+1) \bmod 4] = 1$, all other entries 0. The distance matrix D encodes graph adjacency: $d[k][l] = 1$ if $\{k, l\} \in E$, $d[k][l] = 5$ for non-edges, $d[k][k] = 0$. Both matrices are 4×4, so the QAP has $n = 4$ facilities and $n = 4$ locations.

Step 3 – Verify a solution. The canonical Hamiltonian circuit visits vertices in order $\gamma = (0, 1, 2, 3)$. The QAP permutation is the same: (0, 1, 2, 3). The QAP cost is $\sum_{i=0}^{n-1} c[i][(i+1) \bmod n] \cdot d[\gamma(i)][\gamma((i+1) \bmod n)]$. Since γ maps each position i to vertex i, each consecutive pair $(\gamma(i), \gamma(i+1 \bmod n))$ is an edge in G, contributing $1 \cdot 1 = 1$. Total cost $= 4 = n$ ✓

Multiplicity: The fixture stores one canonical witness. The cycle C_4 has 4 rotations and 2 reflections, giving $2n = 8$ distinct Hamiltonian circuits; the canonical one is the identity permutation.

Rule 3.192: (Hamiltonian Path → Consecutive Ones Submatrix) The vertex-edge incidence matrix has the consecutive-ones property on a selected subset of $n-1$ columns iff those columns correspond to a Hamiltonian path.
Overhead: `num_rows = num_vertices`, `num_cols = num_edges`.

Proof: *Construction.* Given $G = (V, E)$ with $|V| = n$, $|E| = m$, build $n \times m$ Boolean matrix A with $A_{i,j} = 1$ iff vertex i is an endpoint of edge j. Set bound $K = n - 1$.

Correctness. ($\Rightarrow$) A Hamiltonian path's $n-1$ edges, ordered by path position, give each row at most two consecutive ones. ($\Leftarrow$) $n-1$ columns with the C1P property form a subgraph where each vertex has degree ≤ 2; with $n-1$ edges spanning n vertices, this is a Hamiltonian path.

Solution extraction. Selected columns identify edges; walk from a degree-1 vertex to recover the path ordering. □

Example: Path graph P_4 ($n = 4$, $|E| = 3$)

Source: HamiltonianPath **Target:** ConsecutiveOnesSubmatrix

```
$ pred create --example HamiltonianPath/SimpleGraph -o hp.json
$ pred reduce hp.json --to ConsecutiveOnesSubmatrix -o bundle.json
$ pred solve bundle.json
$ pred evaluate hp.json --config 0,1,2,3
```

Step 1 – Source instance. The graph G has $n = 4$ vertices and $m = 3$ edges: {(0,1), (1,2), (2,3)}.

Step 2 – Build the incidence matrix. Construct the $n \times m = 4 \times 3$ vertex-edge incidence matrix A where $A_{i,j} = 1$ iff vertex i is an endpoint of edge j. Set bound $K = n - 1 = 3$. The matrix is:

$$A = \left(\begin{array}{c} 1,\ 0,\ 0; \\ 1,\ 1,\ 0; \\ 0,\ 1,\ 1; \\ 0,\ 0,\ 1 \end{array} \right)$$

Step 3 – Verify a solution. The canonical Hamiltonian path visits vertices in order (0, 1, 2, 3). The target configuration (1, 1, 1) selects 3 columns (all $K = 3$ edges). Ordering the selected columns by path position, each row has at most two consecutive ones ✓

Multiplicity: The fixture stores one canonical witness. The path P_4 has exactly 2 Hamiltonian paths (forward and reverse); the canonical one is (0, 1, 2, 3).

Rule 3.193: (Partition → Bin Packing (weighted)) Items with sizes a_i packed into 2 bins of capacity $\lfloor \frac{S}{2} \rfloor$. A valid 2-bin packing exists iff a balanced partition exists.
Overhead: `num_items = num_elements`.

Proof: *Construction.* Set item sizes $s_i = a_i$, bin capacity $C = \lfloor \frac{S}{2} \rfloor$, number of bins $k = 2$.

Correctness. ($\Rightarrow$) A balanced partition with both subsets summing to $\frac{S}{2}$ gives a valid 2-bin packing. ($\Leftarrow$) Both bins summing to S with capacity $\lfloor \frac{S}{2} \rfloor$ forces S even and each bin holding exactly $\frac{S}{2}$.

Solution extraction. Normalize bin labels: element i in subset 0 if $b_i = b_0$, else subset 1. □

Example: 6 elements, total sum $S = 10$

Source: Partition **Target:** BinPacking

```
$ pred create --example Partition -o partition.json
$ pred reduce partition.json --to BinPacking/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate partition.json --config 0,1,1,0,1,1
```

Step 1 – Source instance. The canonical Partition instance has sizes (3, 1, 1, 2, 2, 1) with total sum $S = 10$, so a balanced partition requires each half to sum to $\frac{S}{2} = 5$.

Step 2 – Build the bin-packing instance. The reduction copies each size into the item-size list and sets the bin capacity to $C = \lfloor \frac{S}{2} \rfloor = 5$ with $k = 2$ bins. The target instance has sizes (3, 1, 1, 2, 2, 1) and capacity 5. No auxiliary variables are introduced, so the target has the same 6 assignment coordinates as the source.

Step 3 – Verify the canonical witness. The witness assigns each element to bin 0 or bin 1 via the binary vector $\boldsymbol{b} = (0, 1, 1, 0, 1, 1)$, which equals the target config (0, 1, 1, 0, 1, 1). Bin 0 receives elements $\{0, 3\}$ with sizes (3, 2) summing to $5 \leq 5$ ✓. Bin 1 receives elements $\{1, 2, 4, 5\}$ with sizes (1, 1, 2, 1) summing to $5 \leq 5$ ✓. Both bins fit within the capacity, and the total $5 + 5 = 10$ accounts for all items.

Multiplicity: The fixture stores one canonical witness. This instance may admit other balanced partitions, but one witness suffices to demonstrate the reduction.

Rule 3.194: (Exact Cover by 3-Sets → Maximum Set Packing) The identity map embeds exact cover as set packing: unit-weight 3-element subsets with a $3q$-element universe. A maximum packing of q disjoint sets is an exact cover.
Overhead: `num_sets = num_subsets`.

Proof: *Construction.* Given $(X, \mathcal{C})$ with $|X| = 3q$, the MaximumSetPacking instance is $(X, \mathcal{C})$ with unit weights.

Correctness. ($\Rightarrow$) An exact cover selects q pairwise-disjoint sets, which is a maximum packing (no packing can exceed q sets of size 3 over $3q$ elements). ($\Leftarrow$) A maximum packing of q disjoint size-3 sets covers all $3q$ elements, hence is an exact cover.

Solution extraction. The selection vector is unchanged. □

Example: 5 subsets over $3q = 6$ elements

Source: ExactCoverBy3Sets **Target:** MaximumSetPacking

```
$ pred create --example ExactCoverBy3Sets -o x3c.json
$ pred reduce x3c.json --to MaximumSetPacking/One -o bundle.json
$ pred solve bundle.json
$ pred evaluate x3c.json --config 1,0,1,0,0
```

Step 1 – Source instance. The X3C instance has universe $X = \{0, ..., 5\}$ with $q = 2$ and 5 candidate triples: $S_0 = \{0, 1, 2\}$, $S_1 = \{0, 1, 3\}$, $S_2 = \{3, 4, 5\}$, $S_3 = \{2, 4, 5\}$, $S_4 = \{1, 3, 5\}$.

Step 2 – Construct the target. The identity map copies each triple as a unit-weight set: 5 sets with weights (1, 1, 1, 1, 1). The target asks for a maximum packing of pairwise-disjoint sets.

Step 3 – Verify the canonical witness. Source config (1, 0, 1, 0, 0) selects subsets {0, 2}:
- $S_0 = \{0, 1, 2\}$
- $S_2 = \{3, 4, 5\}$

These 2 triples are pairwise disjoint and cover all $6 = 3 \cdot 2$ elements ✓ \ Target config is identical: (1, 0, 1, 0, 0) — packing value $= 2 = q$ ✓

Multiplicity: The fixture stores one canonical witness. For this instance there are no other exact covers since every pair of triples that covers all 6 elements is unique.

Rule 3.195: (Exact Cover by 3-Sets → Minimum Fault Detection Test Set) This $O(m + q)$ reduction adapts the classical X3C gadget behind Fault Detection in Directed Graphs [5, MS18] to the repository's internal-vertex coverage semantics. It creates one input per source triple, one internal vertex per universe element, and one shared output, so the target has $m + 3q + 1$ vertices and $3m + 3q$ arcs. The X3C instance is a YES-instance if and only if the Minimum Fault Detection Test Set optimum is at most $q = |U|/3$.
Overhead: `num_vertices = num_subsets + universe_size + 1`, `num_arcs = 3 * num_subsets + universe_size`, `num_inputs = num_subsets`, `num_outputs = 1`.

Proof: *Construction.* Let the X3C instance be $(U, \mathcal{C})$ with $|U| = 3q$ and $\mathcal{C} = \{C_0, ..., C_{m-1}\}$, where each $C_j \subseteq U$ has size 3. Create input vertices $i_0, ..., i_{m-1}$, internal vertices e_u for every $u \in U$, and one shared output vertex o. Add arcs (i_j, e_u) exactly when $u \in C_j$, and add (e_u, o) for every $u \in U$. The implemented target counts only internal vertices, so the coverage requirement applies precisely to the $3q$ vertices e_u.

Correctness. ($\Rightarrow$) If $\mathcal{C}' \subseteq \mathcal{C}$ is an exact cover, select the corresponding input-output pairs (i_j, o). Each chosen pair covers exactly the three internal vertices e_u with $u \in C_j$, and the q chosen triples partition U, so all $3q$ internal vertices are covered using q pairs. ($\Leftarrow$) Suppose the target admits a witness of value at most q. Every selected pair covers at most three internal vertices, while there are $3q$ internal vertices to cover, so an optimal witness of value at most q must in fact have value exactly q and each selected pair must cover three previously uncovered internal vertices. Hence the corresponding source triples are pairwise disjoint and together cover all of U, yielding an exact cover.

Solution extraction. The target configuration has one coordinate per source triple (there is only one output), so the extraction map is the identity. □

Example: 3 triples over $3q = 6$ elements, with one shared output

Source: ExactCoverBy3Sets **Target:** MinimumFaultDetectionTestSet

```
$ pred create --example ExactCoverBy3Sets -o x3c.json
$ pred reduce x3c.json --to MinimumFaultDetectionTestSet -o bundle.json
$ pred solve bundle.json
$ pred evaluate x3c.json --config 1,1,0
```

Step 1 – Source instance. The X3C fixture has universe $U = \{0, ..., 5\}$ with $q = 2$ and triples $C_0 = \{0, 1, 2\}$, $C_1 = \{3, 4, 5\}$, $C_2 = \{0, 3, 4\}$.

Step 2 – Build the fault-detection DAG. Create one input vertex for each triple, one internal vertex for each universe element, and one shared output. The target therefore has 10 vertices, 15 arcs, inputs {0, 1, 2}, and output {9}. Input i_j connects to exactly the three internal vertices for elements in C_j, and every internal vertex connects to the shared output.

Step 3 – Verify the canonical witness. The stored source configuration (1, 1, 0) selects $C_0 = \{0, 1, 2\}$ and $C_1 = \{3, 4, 5\}$, which are disjoint and cover all six universe elements. The target config-

uration is identical: (1, 1, 0). Pair $(0, 9)$ covers internal vertices $\{0, 1, 2\}$, pair $(1, 9)$ covers $\{3, 4, 5\}$, and together they cover every internal vertex with value 2 ✓.

Multiplicity: The fixture stores one canonical witness. Any target witness of value q selects exactly q inputs, and since each selected pair covers only 3 internal vertices while there are $3q$ internal vertices overall, those q neighborhoods must be pairwise disjoint and form an exact cover.

Rule 3.196: (Exact Cover by 3-Sets → Minimum Axiom Set) This $O(m)$ reduction [5, LO17] encodes each source triple as a set-sentence with three forward implications to its elements and one backward implication from those three element-sentences back to the set-sentence. The target has $3q + m$ sentences and $4m$ implications, and the X3C instance is a YES-instance if and only if the Minimum Axiom Set optimum is at most $q = |U|/3$.
Overhead: `num_sentences = universe_size + num_subsets`, `num_true_sentences = universe_size + num_subsets`, `num_implications = 4 * num_subsets`.

Proof: *Construction.* Let the X3C instance be $(U, \mathcal{C})$ with $|U| = 3q$ and $\mathcal{C} = \{C_0, ..., C_{m-1}\}$, where each $C_j = \{a_j, b_j, c_j\}$ has size 3. Create one element-sentence e_u for each $u \in U$ and one set-sentence z_j for each C_j, and let $T = S$ be the full sentence set. For every triple $C_j = \{a_j, b_j, c_j\}$ add the four implications

$$\big(\{z_j\}, e_{a_j}\big), \quad \big(\{z_j\}, e_{b_j}\big), \quad \big(\{z_j\}, e_{c_j}\big), \quad \big(\{e_{a_j}, e_{b_j}, e_{c_j}\}, z_j\big).$$

Variable mapping. Source coordinate j corresponds to set-sentence z_j. The target configuration lists the $3q$ element-sentence coordinates first and the m set-sentence coordinates second.

Correctness. ($\Rightarrow$) If $\mathcal{C}' \subseteq \mathcal{C}$ is an exact cover, then $|\mathcal{C}'| = q$. Choose the corresponding set-sentences as the only axioms. Their forward implications derive all $3q$ element-sentences in one round because the cover spans U. Once every element-sentence is true, every backward implication fires, so every set-sentence becomes true and the full closure equals T. Hence the target optimum is at most q.

($\Leftarrow$) Suppose some axiom set A of size at most q derives all of T. Write $A = E \cup Z$, where E contains the chosen element-sentences and Z the chosen set-sentences. Backward implications never create a new element-sentence: if a backward rule derives z_j, its antecedent already contains the three element-sentences of C_j, and the forward rules from z_j only repeat those same elements. Therefore the closure's element-sentences are exactly the chosen element axioms together with the elements that lie in triples indexed by Z. Since the closure contains all $3q$ element-sentences,

$$3q \le |E| + 3\,|Z| \le |E| + 3(q - |E|) = 3q - 2\,|E|.$$

Thus $|E| = 0$ and then $|Z| = q$. Equality also forces the chosen triples to be pairwise disjoint and to cover all $3q$ elements, so the corresponding source subsets form an exact cover.

Solution extraction. Given a target axiom vector, keep only the coordinates of the set-sentences $z_0, ..., z_{m-1}$. On every optimal target witness of value q, these coordinates select exactly q disjoint triples covering U, so they are an X3C witness. □

Example: 5 triples over $3q = 6$ elements, with decision bound $q = 2$

Source: ExactCoverBy3Sets **Target:** MinimumAxiomSet

```
$ pred create --example ExactCoverBy3Sets -o x3c.json
$ pred reduce x3c.json --to MinimumAxiomSet -o bundle.json
$ pred solve bundle.json
$ pred evaluate x3c.json --config 0,0,0,1,1
```

Step 1 – Source instance. The X3C fixture has universe $U = \{0, ..., 5\}$ with $q = 2$ and candidate triples $C_0 = \{0, 1, 2\}$, $C_1 = \{0, 3, 4\}$, $C_2 = \{2, 4, 5\}$, $C_3 = \{1, 3, 5\}$, $C_4 = \{0, 2, 4\}$.

Step 2 – Build the axiom system. Create one element-sentence for each universe element and one set-sentence for each triple, so the target has 11 sentences and 11 true sentences. Each triple contributes three forward implications and one backward implication, giving 20 implications total. The optimization instance itself stores only the axiom-system data; the X3C bound $q = 2$ is checked externally against the optimum target value.

Step 3 – Verify the canonical witness. The stored source config (0, 0, 0, 1, 1) selects $C_3 = \{1, 3, 5\}$ and $C_4 = \{0, 2, 4\}$. These two triples are disjoint and cover all six universe elements ✓. The target axiom vector (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1) selects exactly the set-sentence coordinates 9 and 10, namely z_3 and z_4. One closure round derives every element-sentence $e_0, ..., e_5$; then the backward rules derive the remaining set-sentences z_0, z_1, z_2, so the closure equals all 11 true sentences. This witness therefore attains value 2 ✓, and extracting the chosen set-sentence coordinates recovers the exact cover.

Multiplicity: The fixture stores one canonical witness. Any target witness of value q must select exactly q set-sentences and no element-sentences, because each direct element axiom lowers the maximum possible element coverage by two.

Rule 3.197: (Subset Sum → Partition) This $O(n)$ reduction [5] computes padding $d = |\Sigma - 2T|$ and appends at most one element. A subset of S sums to T if and only if the padded multiset admits a balanced partition.
Overhead: `num_elements = num_elements + 1`.

Proof: *Construction.* Let (S, T) be a Subset Sum instance with n elements and $\Sigma = \sum_{i=1}^{n} s_i$. Compute $d = |\Sigma - 2T|$. If $d = 0$, output Partition instance S. If $d > 0$, output $S \cup \{d\}$ (one extra element). Let Σ' denote the total of the Partition instance and $H = \Sigma'/2$.

Correctness. There are three cases.

Case 1 ($\Sigma = 2T$, $d = 0$): $H = T$. ($\Rightarrow$) A subset summing to $T = H$ is one half of a balanced partition. ($\Leftarrow$) A balanced partition yields a subset summing to $H = T$.

Case 2 ($\Sigma > 2T$, $d = \Sigma - 2T$): $\Sigma' = 2(\Sigma - T)$, $H = \Sigma - T$. ($\Rightarrow$) If $A' \subseteq S$ sums to T, then $A' \cup \{d\}$ sums to $T + (\Sigma - 2T) = H$. ($\Leftarrow$) In any balanced partition, the side containing d has S-elements summing to $H - d = T$.

Case 3 ($\Sigma < 2T$, $d = 2T - \Sigma$): $\Sigma' = 2T$, $H = T$. ($\Rightarrow$) A subset summing to $T = H$ gives one side directly. ($\Leftarrow$) The side opposite d has S-elements summing to $H = T$.

If $T > \Sigma$, then $d > \Sigma'/2$, so a single element exceeds the half-sum and the Partition instance is infeasible.

Solution extraction. Given a Partition solution $c \in \{0, 1\}^m$: if $d = 0$, return $c[0..n]$. If $\Sigma > 2T$, the S-elements on the same side as the padding form the subset summing to T. If $\Sigma < 2T$, the S-elements on the opposite side from the padding form the subset summing to T. □

Example: 4 elements, target $T = 11$

Source: SubsetSum **Target:** Partition

```
$ pred create --example SubsetSum -o subsetsum.json
$ pred reduce subsetsum.json --to Partition -o bundle.json
$ pred solve bundle.json
$ pred evaluate subsetsum.json --config 0,1,1,0
```

Step 1 – Source instance. Subset Sum with sizes (1, 5, 6, 8) and target $T = 11$. Total $\Sigma = 20$.

Step 2 – Compute padding. $\Sigma = 20$, $2T = 22$. Since $\Sigma < 2T$, we have $d = 2T - \Sigma = 2$. The Partition instance is $S \cup \{d\} = (1, 5, 6, 8, 2)$ with 5 elements.

Step 3 – Verify a solution. Source config (0, 1, 1, 0): selected elements = {5, 6} sum to $11 = T$ ✓. Target config (0, 1, 1, 0, 0): side-0 sum = 11, side-1 sum = 11 – balanced ✓.

Multiplicity: The fixture stores one canonical witness. Other valid subsets summing to T may exist.

Rule 3.198: (Subset Sum → Integer Knapsack) This size-preserving embedding from Garey and Johnson's Integer Knapsack entry [5, MP10] copies each Subset Sum number into both the size and value of a knapsack item and sets the capacity to the target sum. Any exact subset-sum witness becomes a feasible Integer Knapsack witness of value B. The converse fails for the implemented unbounded model because target witnesses may use multiplicities greater than 1, so the edge is documented but intentionally proof-only.
Overhead: `num_items = num_elements`, `capacity = target`.

Proof: *Construction.* Given Subset Sum instance $(S = \{a_1, ..., a_n\}, B)$, create n Integer Knapsack items. For each i, set the item size and value to the same number:

$$s_i = a_i, \quad v_i = a_i.$$

Set the knapsack capacity to B. The target therefore has the same number of items as the source has elements.

Correctness. ($\Rightarrow$) If $I \subseteq \{1, ..., n\}$ satisfies $\sum_{i \in I} a_i = B$, define multiplicities $c_i = 1$ for $i \in I$ and $c_i = 0$ otherwise. Then

$$\sum_i c_i s_i = \sum_{i \in I} a_i = B \leq B$$

and, because $v_i = s_i$, also

$$\sum_i c_i v_i = B.$$

So every YES instance of Subset Sum maps to an Integer Knapsack witness achieving value B.

($\Leftarrow$) The backward implication is false for the implemented target model. Integer Knapsack allows arbitrary non-negative multiplicities, while Subset Sum is 0-1. For example, with $S = \{3\}$ and $B = 6$, the target witness $c_0 = 2$ is feasible and attains value 6, but the source has no subset summing to 6. Hence neither exact witness recovery nor exact optimum-value recovery is available from the target side.

Solution extraction. No runtime extractor is registered. The forward map is enough for the NP-hardness proof, but unbounded multiplicities prevent an exact inverse map back to Subset Sum. □

Example: 5 elements, target $B = 16$: exact forward witness, but multiplicities create a backward gap

Source: SubsetSum **Target:** IntegerKnapsack

```
$ pred create --example SubsetSum -o subsetsum.json
$ pred solve subsetsum.json
$ pred create --example IntegerKnapsack -o integer-knapsack.json
$ pred solve integer-knapsack.json
```

Step 1 – Source instance. The canonical Subset Sum instance has sizes (3, 7, 1, 8, 5) and target $B = 16$. The stored witness (1, 0, 0, 1, 1) selects elements {0, 3, 4}, whose values sum to $16 = B$ ✓.

Step 2 – Build the target. Copy each source size into both the size and value lists. The Integer Knapsack instance therefore has sizes (3, 7, 1, 8, 5), values (3, 7, 1, 8, 5), and the same capacity $B = 16$.

Step 3 – Verify the forward witness. Reuse the same 0-1 vector as multiplicities: (1, 0, 0, 1, 1). Its total size is $16 \leq 16$, and because size equals value coordinate-wise, its total value is also $16 = B$ ✓.

Step 4 – Backward gap. For the source instance $A = \{3\}$ with target $B = 6$, Subset Sum is NO, but Integer Knapsack can set multiplicity $c_0 = 2$ and achieve total size/value 6. This is why the catalog records the edge for proof topology only and disables all runtime reduction modes.

Rule 3.199: (SAT $\rightarrow$ Non-Tautology) This $O(n + m)$ reduction [5] negates a CNF formula via De Morgan's laws to obtain a DNF formula $E = \neg\varphi$. The formula φ is satisfiable if and only if E is not a tautology. Variables, their count, and polarity are preserved; each clause becomes a disjunct of negated literals.
Overhead: `num_vars = num_vars`, `num_disjuncts = num_clauses`.

Proof: *Construction.* Let $\varphi = C_1 \wedge ... \wedge C_m$ be a CNF formula over n variables. For each clause $C_j = (l_1 \vee ... \vee l_k)$, form the disjunct $D_j = \left(\overline{l_1} \wedge ... \wedge \overline{l_k}\right)$ where $\bar{l}$ is the complement of literal l. The Non-Tautology instance is the DNF formula $E = D_1 \vee ... \vee D_m$.

Correctness. ($\Rightarrow$) If $\alpha \vDash \varphi$, then α makes every clause true, so $\alpha \vDash \neg E$ (since $E = \neg\varphi$), and E has a falsifying assignment. ($\Leftarrow$) If β falsifies E, then $\beta \nvDash \neg\varphi$, so $\beta \vDash \varphi$ and φ is satisfiable.

Solution extraction. The falsifying assignment for E is directly the satisfying assignment for φ – no transformation needed since the variables are identical. □

Example: $n = 3$ variables, $m = 3$ clauses

Source: Satisfiability **Target:** NonTautology

```
$ pred create --example Satisfiability -o sat.json
$ pred reduce sat.json --to NonTautology -o bundle.json
$ pred solve bundle.json
$ pred evaluate sat.json --config 1,0,1
```

Step 1 – Source instance. CNF with $n = 3$ variables and $m = 3$ clauses.

Step 2 – Apply De Morgan. Each clause $C_j = (l_1 \vee ... \vee l_k)$ becomes disjunct $D_j = \left(\overline{l_1} \wedge ... \wedge \overline{l_k}\right)$. The Non-Tautology instance has 3 disjuncts over 3 variables.

Step 3 – Verify a solution. Source config (1, 0, 1) satisfies the CNF. Target config (1, 0, 1) falsifies the DNF (same assignment). Variables are identical ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.200: (k-Coloring (k-ary) $\rightarrow$ Partition into Cliques) This $O(n^2)$ reduction [1] computes the complement graph $\overline{G}$ and sets the clique bound $K' = K$. A proper K-coloring of G exists if and only if $\overline{G}$ can be partitioned into at most K' cliques, because color classes (independent sets in G) correspond to cliques in $\overline{G}$.
Overhead: `num_vertices = num_vertices`, `num_edges = num_vertices * (num_vertices + -1 * 1) * 2^-1 + -1 * num_edges`.

Proof: *Construction.* Given K-Coloring instance $(G = (V, E), K)$, compute $\overline{G} = \left(V, \overline{E}\right)$ where $\overline{E} = \{\{u, v\} : u \neq v, \{u, v\} \notin E\}$. Set $K' = K$. Output Partition Into Cliques instance $\left(\overline{G}, K'\right)$.

Correctness. ($\Rightarrow$) If $c : V \rightarrow \{0, ..., K-1\}$ is a proper coloring, define $V_i = \{v : c(v) = i\}$. For $u, v \in V_i$, $c(u) = c(v)$ implies $\{u, v\} \notin E$, so $\{u, v\} \in \overline{E}$, making V_i a clique in $\overline{G}$. The K color classes partition V into at most $K' = K$ cliques. ($\Leftarrow$) If $V_0, ..., V_{k-1}$ is a partition of $\overline{G}$ into $k \leq K'$ cliques, then each V_i is an independent set in G. Assigning color i to all vertices in V_i gives a proper k-coloring of G.

Solution extraction. Given a partition $V_0, ..., V_{k-1}$ into cliques, assign color i to every vertex in V_i. □

Example: $n = 5$ vertices, $k = 3$ colors

Source: KColoring **Target:** PartitionIntoCliques

```
$ pred create --example KColoring/SimpleGraph/KN -o kcoloring.json
$ pred reduce kcoloring.json --to PartitionIntoCliques/SimpleGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate kcoloring.json --config 0,1,1,0,2
```

Step 1 – Source instance. Graph G with $n = 5$ vertices, $|E| = 6$ edges, $k = 3$ colors.

Step 2 – Complement graph. $\overline{G}$ has the same $n = 5$ vertices and $|\overline{E}| = 4$ edges. Clique bound $K' = 3$.

Step 3 – Verify a solution. Source coloring (0, 1, 1, 0, 2). Target partition (0, 1, 1, 0, 2) – each color class is a clique in $\overline{G}$ ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.201: (*k*-Coloring (*k*-ary) → Clustering) This $O(n^2)$ reduction [5, MS9], [158] keeps the vertex set as the ground set, assigns distance 1 to adjacent pairs and distance 0 to nonadjacent pairs, and fixes $K = 3$ and $B = 0$. A feasible clustering is therefore exactly a partition of the graph into at most three independent sets.
Overhead: `num_elements = num_vertices`.

Proof: *Construction.* Given a 3-Coloring instance $(G = (V, E), 3)$, build a Clustering instance on the same ground set $X = V$. Set $d(u, v) = 1$ if $\{u, v\} \in E$, set $d(u, v) = 0$ if $\{u, v\} \notin E$, and set $d(u, u) = 0$ for every vertex u. Set the cluster bound to $K = 3$ and the diameter bound to $B = 0$.

Correctness. ($\Rightarrow$) A proper 3-coloring partitions V into at most three color classes, and each color class is an independent set. Because independent sets contain no adjacent pair, every two vertices inside one color class have distance 0, so these classes form a feasible clustering. ($\Leftarrow$) Conversely, a feasible clustering with $B = 0$ puts only distance-0 pairs in the same cluster. By construction distance 0 means nonadjacent, so every cluster is an independent set. Assigning one color per cluster yields a proper 3-coloring.

Solution extraction. Read the cluster label of each source vertex as its color label. □

Rule 3.202: (Clustering → Integer Linear Programming) This direct $O(n^2 K)$ ILP encoding[68] introduces one binary variable $x_{i,c}$ for each element i and cluster c, forces every element into exactly one cluster, and forbids every pair with $d(i, j) > B$ from sharing a cluster (nK variables and at most $n + n^2 K$ constraints).
Overhead: `num_vars = num_elements * num_clusters`, `num_constraints = num_elements + num_elements * (num_elements + -1 * 1) * 2^-1 * num_clusters`.

Proof: *Construction.* Given a Clustering instance on elements $X = \{0, ..., n-1\}$ with cluster bound K and diameter bound B, create binary variables $x_{i,c} \in \{0, 1\}$ for every element $i \in X$ and cluster $c \in \{0, ..., K-1\}$. Interpret $x_{i,c} = 1$ iff element i is assigned to cluster c.

For every element i, add the assignment constraint $\sum_{c=0}^{K-1} x_{i,c} = 1$. For every pair $i < j$ with $d(i, j) > B$ and every cluster c, add the conflict constraint $x_{i,c} + x_{j,c} \leq 1$. Use the zero objective and minimize it, so the target is a pure feasibility ILP.

Correctness. ($\Rightarrow$) If the source instance has a feasible clustering, set $x_{i,c} = 1$ exactly for the cluster assigned to element i. Every element is assigned once, so all assignment equalities hold. Whenever $d(i, j) > B$, the source clustering never puts i and j together, so for every cluster c at least one of $x_{i,c}$ or $x_{j,c}$ is 0 and every conflict inequality holds. ($\Leftarrow$) If the ILP is feasible, the assignment equalities choose exactly one cluster for each element. If some extracted cluster contained a pair i, j with $d(i, j) > B$, then for that cluster

[68] Standard ILP formulation of diameter-bounded clustering; no specific literature key is currently registered in `references.bib`.

c we would have $x_{i,c} = x_{j,c} = 1$, contradicting the conflict inequality. Hence every extracted cluster has diameter at most B, so the extracted assignment is a feasible clustering using at most K clusters.

Variable mapping. The implementation stores variable $x_{i,c}$ at index $i \cdot K + c$, i.e. consecutive blocks of K variables per element.

Solution extraction. For each element i, scan the block $(x_{i,0}, ..., x_{i,K-1})$ and return the unique cluster c with value 1. □

Example: 4 elements, $K = 2$, $B = 1$ → ILP with 8 variables and 12 constraints

Source: Clustering **Target:** ILP

```
$ pred create --example Clustering -o clustering.json
$ pred reduce clustering.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate clustering.json --config 0,0,1,1
```

Step 1 – Source instance. The canonical source has $n = 4$ elements, cluster bound $K = 2$, and diameter bound $B = 1$. Its distance rows are (0, 1, 3, 3); (1, 0, 3, 3); (3, 3, 0, 1); (3, 3, 1, 0), so the only pairs above the bound are $(0, 2)$, $(0, 3)$, $(1, 2)$, and $(1, 3)$.

Step 2 – Build the ILP. Introduce one binary variable $x_{i,c}$ for each element-cluster pair, giving $nK = 8$ variables. The $n = 4$ assignment equalities $\sum_c x_{i,c} = 1$ force every element into exactly one cluster, and the four violating pairs contribute $4 \cdot K = 8$ conflict inequalities. The stored target therefore has 8 variables and 12 constraints.

Step 3 – Verify the canonical witness. The stored ILP vector is (1, 0, 1, 0, 0, 1, 0, 1). Reading each block of $K = 2$ variables yields the clustering (0, 0, 1, 1), so cluster 0 contains elements {0, 1} and cluster 1 contains elements {2, 3}. The only within-cluster distances are $d(0, 1) = 1$ and $d(2, 3) = 1$, both at most B ✓.

Multiplicity: The fixture stores one canonical witness. Swapping the two cluster labels gives an equivalent second witness because the ILP distinguishes clusters only by index.

Rule 3.203: (Partition into Cliques → Minimum Covering by Cliques) This $O((n + 2m)^2)$ reduction [5, GT17], [159], [160] inlines Orlin's vertex-clique-cover to edge-clique-cover construction. Each source vertex v_i becomes left/right copies x_i and y_i, each directed source edge contributes a 4-vertex gadget, and two side cliques account for the additive $2m + 2$ slack. The target graph has an edge-clique cover of size at most $K + 2m + 2$ if and only if the source graph admits a partition into at most K cliques.
Overhead: `num_vertices = 2 * num_vertices + 4 * num_edges + 2`, `num_edges = (num_vertices + 2 * num_edges)^2 + 2 * num_vertices + 10 * num_edges`.

Proof: *Construction.* Let $(G = (V, E), K)$ be a Partition Into Cliques instance with $V = \{v_1, ..., v_n\}$ and $m = |E|$. Define the directed-edge index set $A = \{(i, j) : i \neq j \wedge \{v_i, v_j\} \in E\}$, so $|A| = 2m$. Create vertices x_i, y_i for each source vertex v_i, gadget vertices $a_{i,j}, b_{i,j}$ for each $(i, j) \in A$, and two special vertices z_L, z_R. Let $L = \{x_i : 1 \leq i \leq n\} \cup \{a_{i,j} : (i, j) \in A\}$ and $R = \{y_i : 1 \leq i \leq n\} \cup \{b_{i,j} : (i, j) \in A\}$. Make L a clique and R a clique, join z_L to every vertex of L and z_R to every vertex of R, add each matching edge $x_i y_i$, and for every $(i, j) \in A$ add the four cross edges $x_i y_j$, $x_i b_{i,j}$, $a_{i,j} y_j$, and $a_{i,j} b_{i,j}$. Output the resulting Minimum Covering by Cliques instance. It has

$$|V(H)| = 2n + 4m + 2, \quad |E(H)| = (n + 2m)^2 + 2n + 10m,$$

and threshold $K' = K + 2m + 2$.

Correctness. (⇒) Suppose G is partitioned into cliques $C_1, ..., C_t$ with $t \leq K$. For each source clique C_r, define $D_r = \{x_i, y_i : v_i \in C_r\}$. For each $(i, j) \in A$, define $Q_{i,j} = \{x_i, a_{i,j}, b_{i,j}, y_j\}$. Also define $L^* = L \cup \{z_L\}$ and $R^* = R \cup \{z_R\}$. Every D_r is a clique: if $v_i, v_j \in C_r$ with $i \neq j$, then $\{v_i, v_j\} \in E$, so the construction

includes both cross edges $x_i y_j$ and $x_j y_i$. The family $\{D_1, ..., D_t\} \cup \{Q_{i,j} : (i,j) \in A\} \cup \{L^*, R^*\}$ therefore covers every target edge, using at most $K + 2m + 2$ cliques.

($\Leftarrow$) Conversely, suppose H has an edge-clique cover with at most $K + 2m + 2$ cliques. Each gadget edge $a_{i,j} b_{i,j}$ belongs to the unique maximal clique $Q_{i,j}$, so covering all $2m$ such edges requires at least $2m$ distinct cliques. Likewise, some clique must contain z_L and some clique must contain z_R, and neither of those cliques can contain a matching edge $x_i y_i$. Hence at most K cliques remain available for the matching edges. If two matching edges $x_i y_i$ and $x_j y_j$ lie in the same target clique, that clique contains the four vertices x_i, y_i, x_j, y_j, so in particular $x_i y_j$ is an edge of H; by construction this implies $\{v_i, v_j\} \in E$. Therefore the source vertices whose matching edges share one target-clique label form a clique of G. Grouping each v_i by the label used on $x_i y_i$ yields a partition of V into at most K cliques.

Variable mapping. The source witness labels source vertices by clique. The target witness labels target edges by the covering clique that contains them.

Solution extraction. Inspect the label assigned to each matching edge $x_i y_i$. Compress the distinct matching-edge labels to $0, ..., k-1$ and assign source vertex v_i to the compressed label of its matching edge. The previous paragraph proves that these label classes are source cliques, and the forced gadget/side cliques guarantee $k \leq K$ whenever the target cover has size at most $K + 2m + 2$. □

Example: $n = 3$ **vertices,** $m = 1$ **edges,** $K = 2$

Source: PartitionIntoCliques **Target:** MinimumCoveringByCliques

```
$ pred create --example PartitionIntoCliques/SimpleGraph -o partition-into-cliques.json
$ pred reduce partition-into-cliques.json --to MinimumCoveringByCliques/SimpleGraph -o
bundle.json
$ pred solve bundle.json
$ pred evaluate partition-into-cliques.json --config 0,0,1
```

Step 1 – Source instance. Graph G with $n = 3$ vertices, $m = 1$ edge, and clique bound $K = 2$. The stored partition witness is (0, 0, 1), namely the cliques $\{0, 1\}$ and $\{2\}$.

Step 2 – Orlin construction. The target graph has 12 vertices and 41 edges. Because the source has two directed edge copies, the construction adds the gadgets $Q_{0,1}$ and $Q_{1,0}$, plus the side cliques L^* and R^*. The threshold is $K' = K + 2m + 2 = 6$.

Step 3 – Verify the witness. The target witness labels 41 target edges with 6 clique IDs, corresponding to $D_1 = \{x_0, x_1, y_0, y_1\}$, $D_2 = \{x_2, y_2\}$, $Q_{0,1}$, $Q_{1,0}$, L^*, and R^*. Reading only the labels on the matching edges $x_i y_i$ recovers the source partition (0, 0, 1) ✓.

Multiplicity: The fixture stores one canonical witness. Any permutation of the six target clique labels is equivalent.

Rule 3.204: (*k*-SAT (*k*-ary) → Kernel) This $O(n + m)$ reduction [5] constructs a directed graph with $2n + 3m$ vertices and $2n + 6m$ arcs. Each variable contributes a digon on its positive and negative literal vertices, and each 3-clause contributes a directed 3-cycle whose three clause vertices point to the corresponding literal vertices. The implemented kernels may contain clause vertices as well as literal vertices.
Overhead: `num_vertices = 2 * num_vars + 3 * num_clauses`, `num_arcs = 2 * num_vars + 6 * num_clauses`.

Proof: *Construction.* Let $\varphi = C_1 \wedge ... \wedge C_m$ be a 3-SAT instance on variables $x_1, ..., x_n$. For each variable x_i, create two literal vertices p_i and n_i with arcs (p_i, n_i) and (n_i, p_i). For each clause $C_j = (\ell_{j,0} \vee \ell_{j,1} \vee \ell_{j,2})$, create clause vertices $c_{j,0}, c_{j,1}, c_{j,2}$ with cycle arcs $(c_{j,0}, c_{j,1})$, $(c_{j,1}, c_{j,2})$, and $(c_{j,2}, c_{j,0})$. Add one literal arc $(c_{j,t}, v(\ell_{j,t}))$ from each clause vertex to the literal vertex representing its own literal. Thus each clause contributes 3 cycle arcs and 3 literal arcs, for a total of $2n + 6m$ arcs.

Correctness. ($\Rightarrow$) Let α be a satisfying assignment. Put into K exactly one literal vertex from each digon: p_i if $\alpha(x_i) = \text{true}$ and n_i otherwise. For each clause C_j, additionally put $c_{j,t}$ into K exactly when $\ell_{j,t}$ is

false and $\ell_{j,(t+1) \bmod 3}$ is true. This never creates an arc inside K: each selected clause vertex points to a false literal vertex, and two adjacent clause vertices cannot both satisfy the selection rule. Every unselected literal vertex is absorbed by its digon partner. For an unselected clause vertex $c_{j,t}$, either $\ell_{j,t}$ is true, so its literal arc hits the selected literal vertex, or $\ell_{j,t}$ is false. In the latter case $c_{j,t}$ was not selected, so $\ell_{j,(t+1) \bmod 3}$ is also false; because the clause is satisfied, $\ell_{j,(t+2) \bmod 3}$ is true, hence $c_{j,(t+1) \bmod 3}$ was selected and absorbs $c_{j,t}$ along the cycle.

($\Leftarrow$) Let K be a kernel of the constructed digraph. In each variable digon, at most one literal vertex can lie in K by independence, and at least one must lie in K to absorb the other endpoint; so each digon contributes exactly one selected literal vertex. Set $\alpha(x_i) = \text{true}$ iff $p_i \in K$. Now fix a clause gadget with cycle $(c_{j,0}, c_{j,1}, c_{j,2})$. If none of its three literal vertices were selected, then the only possible absorbers for $c_{j,0}, c_{j,1}, c_{j,2}$ would be the cycle successors. Independence allows at most one clause vertex in K, but one selected vertex on a directed 3-cycle cannot absorb the other two, contradiction. Therefore every clause has at least one selected literal vertex, so α satisfies every clause.

Solution extraction. Read only the positive literal vertices: $\alpha(x_i) = 1$ iff the even-indexed vertex for x_i is in the kernel. Any selected clause vertices are ignored during extraction. □

Example: $n = 3$ variables, $m = 2$ clauses

Source: KSatisfiability **Target:** Kernel

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to Kernel -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 1,1,1
```

Step 1 – Source instance. 3-SAT with $n = 3$ variables and $m = 2$ clauses. The canonical satisfying assignment is (1, 1, 1).

Step 2 – Construct the digraph. Variable gadgets contribute $2n = 6$ literal vertices and $2n = 6$ digon arcs. Clause gadgets contribute $3m = 6$ clause vertices, $3m = 6$ cycle arcs, and $3m = 6$ literal arcs. Total: 12 vertices and 18 arcs $= 2n + 6m$.

Step 3 – Verify the canonical witness. The target kernel selects literal vertices {0, 2, 4} and clause vertices {10}. Here {0, 2, 4} encode $(x_1, x_2, x_3) = (1, 1, 1)$, and the extra clause vertex 10 is needed in the second clause gadget: vertex 9 is absorbed by arc $(9, 10)$, while vertex 11 is absorbed by its literal arc to vertex 4 ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.205: (Hamiltonian Path → Degree-Constrained Spanning Tree) This $O(n + m)$ reduction [5] passes the graph through unchanged and sets the degree bound $K = 2$. A Hamiltonian path is a spanning tree with maximum degree 2 (a path), and conversely any degree-2 spanning tree is a Hamiltonian path. The reduction is size-preserving.
Overhead: `num_vertices = num_vertices`, `num_edges = num_edges`.

Proof: *Construction.* Given Hamiltonian Path instance $G = (V, E)$, output Degree-Constrained Spanning Tree instance $(G, K = 2)$.

Correctness. ($\Rightarrow$) A Hamiltonian path $v_0, v_1, ..., v_{n-1}$ uses $n - 1$ edges spanning all vertices. Interior vertices have degree 2, endpoints have degree 1, so max degree $\leq 2 = K$. ($\Leftarrow$) A spanning tree with max degree ≤ 2 has no branching (a branch point requires degree ≥ 3). A connected acyclic graph without branching is a simple path. Since the tree spans all n vertices, it is a Hamiltonian path.

Solution extraction. Collect the selected edges, find an endpoint (degree 1 vertex), walk the path to produce the vertex permutation. □

Example: $n = 6$ **vertices,** $K = 2$

Source: HamiltonianPath **Target:** DegreeConstrainedSpanningTree

```
$ pred create --example HamiltonianPath/SimpleGraph -o hampath.json
$ pred reduce hampath.json --to DegreeConstrainedSpanningTree/SimpleGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate hampath.json --config 0,2,4,3,1,5
```

Step 1 – Source instance. Graph G with $n = 6$ vertices and $|E| = 8$ edges.

Step 2 – Identity reduction. Target graph is identical: $n = 6$ vertices, $|E| = 8$ edges, degree bound $K = 2$.

Step 3 – Verify a solution. Hamiltonian path visits vertices in order (0, 2, 4, 3, 1, 5). The corresponding spanning tree selects 5 edges (all with max degree ≤ 2) ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.206: (NAE-SAT $\to$ Set Splitting) This $O(n + m)$ reduction [5] maps each variable x_{i+1} to two universe elements ($2i$ for positive, $2i + 1$ for negative literal). Complementarity subsets $\{2i, 2i + 1\}$ force opposite colors, and each clause becomes a subset of the corresponding literal elements. A NAE-satisfying assignment exists if and only if the Set Splitting instance admits a valid 2-coloring.
Overhead: `universe_size = 2 * num_vars`, `num_subsets = num_vars + num_clauses`.

Proof: *Construction.* Given NAE-SAT with n variables and m clauses, define universe $U = \{0, 1, ..., 2n - 1\}$. For each variable x_{i+1}, create complementarity subset $R_i = \{2i, 2i + 1\}$. For each clause C_j, create subset T_j containing element $2(k - 1)$ for positive literal x_k and $2(k - 1) + 1$ for negative literal $\overline{x}_k$. Total: $|U| = 2n$, $n + m$ subsets.

Correctness. ($\Rightarrow$) A NAE-satisfying assignment α induces 2-coloring $\chi(2i) = \alpha(x_{i+1})$, $\chi(2i + 1) = 1 - \alpha(x_{i+1})$. Complementarity subsets are non-monochromatic by construction. Clause subsets are non-monochromatic because NAE ensures both true and false literals. ($\Leftarrow$) A valid 2-coloring with $\chi(2i) \neq \chi(2i + 1)$ (forced by R_i) defines $\alpha(x_{i+1}) = \chi(2i)$. Non-monochromaticity of clause subsets ensures both true and false literals in each clause.

Solution extraction. Set $\alpha(x_{i+1}) = \chi(2i)$ for $i = 0, ..., n - 1$. □

Example: $n = 3$ **variables,** $m = 2$ **clauses**

Source: NAESatisfiability **Target:** SetSplitting

```
$ pred create --example NAESatisfiability -o naesat.json
$ pred reduce naesat.json --to SetSplitting -o bundle.json
$ pred solve bundle.json
$ pred evaluate naesat.json --config 1,1,1
```

Step 1 – Source instance. The fixture has clauses $C_1 = (x_1 \vee \overline{x_2} \vee x_3)$ and $C_2 = (\overline{x_1} \vee x_2 \vee \overline{x_3})$. The canonical NAE assignment is (1, 1, 1), so the two clauses evaluate to $(1, 0, 1)$ and $(0, 1, 0)$ respectively.

Step 2 – Build the universe and complementarity subsets. The reduction creates $U = \{0, ..., 5\}$ with positive literals on $\{0, 1, 2\}$ and negative literals on $\{3, 4, 5\}$. The first three target subsets are $R_1 = \{0, 3\}$, $R_2 = \{1, 4\}$, and $R_3 = \{2, 5\}$.

Step 3 – Encode the clauses as set-splitting constraints. Clause C_1 becomes $T_1 = \{0, 4, 2\}$, and clause C_2 becomes $T_2 = \{3, 1, 5\}$. Under the target coloring (1, 1, 1, 0, 0, 0), T_1 receives colors $(1, 0, 1)$ and T_2 receives $(0, 1, 0)$, so both subsets are non-monochromatic.

Step 4 – Verify the witness pair. Every complementarity pair has opposite colors: $(0,3)$ gives $(1,0)$, $(1,4)$ gives $(1,0)$, and $(2,5)$ gives $(1,0)$. Reading the positive-literal colors $(1,1,1)$ recovers the source assignment (1, 1, 1) ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.207: (NAE-SAT → Partition into Perfect Matchings) This $O(n+m)$ reduction [5, GT16], [68] normalizes each 2-literal clause (ℓ_1, ℓ_2) to (ℓ_1, ℓ_1, ℓ_2), then builds 4-vertex variable gadgets, 2-vertex signal pairs, 4-vertex K_4 clause gadgets, and 2-vertex equality-chain links. For m normalized clauses it produces $4n + 16m$ vertices, $3n + 21m$ edges, and fixes $K = 2$.
Overhead: `num_vertices = 4 * num_vars + 16 * num_clauses`, `num_edges = 3 * num_vars + 21 * num_clauses`, `num_matchings = 2`.

Proof: *Construction.* Let φ be a NAE-SAT instance on variables $x_1, ..., x_n$ whose clauses have size 2 or 3, matching the implemented rule. Replace every 2-literal clause (ℓ_1, ℓ_2) by (ℓ_1, ℓ_1, ℓ_2), yielding normalized 3-literal clauses $C_j = (\ell_{j,0}, \ell_{j,1}, \ell_{j,2})$ for $j = 0, ..., m-1$. For each variable x_i, create vertices t_i, t_i', f_i, f_i' with edges (t_i, t_i'), (f_i, f_i'), and (t_i, f_i). For each clause position (j,k), create a signal pair $s_{j,k}, s'_{j,k}$ with edge $(s_{j,k}, s'_{j,k})$. For each clause C_j, create vertices $w_{j,0}, w_{j,1}, w_{j,2}, w_{j,3}$ forming a K_4, and add connection edges $(s_{j,k}, w_{j,k})$ for $k \in \{0,1,2\}$.

For each variable, chain its positive occurrences starting from t_i and its negative occurrences starting from f_i. If (j,k) is the next occurrence in the chosen sign-order and src is the current chain source, create fresh vertices μ, μ' with edges (μ, μ'), (src, μ), and $(s_{j,k}, \mu)$, then update src $:= s_{j,k}$. Output the Partition Into Perfect Matchings instance $(G, 2)$.

Correctness. ($\Rightarrow$) Let α be a NAE-satisfying assignment. Put t_i, t_i' in group 0 and f_i, f_i' in group 1 when $\alpha(x_i) = 1$; swap the two groups when $\alpha(x_i) = 0$. Every equality-chain pair forces its signal vertex to share the group of the current chain source, so positive occurrences inherit the group of t_i and negative occurrences inherit the group of f_i. In each normalized clause, the three signals are not all equal because α satisfies the NAE condition. Assign $w_{j,k}$ to the opposite group from $s_{j,k}$ for $k = 0, 1, 2$, and assign $w_{j,3}$ to the minority group among $w_{j,0}, w_{j,1}, w_{j,2}$. Then every variable gadget, signal pair, and equality-chain pair contributes exactly one same-group edge, and each K_4 splits $2 + 2$, so every vertex has exactly one same-group neighbor.

($\Leftarrow$) Suppose $(G, 2)$ admits a partition into two perfect matchings. In each variable gadget, the edges (t_i, t_i') and (f_i, f_i') force those pairs to share a group, while the edge (t_i, f_i) forces t_i and f_i to lie in opposite groups. Each equality-chain pair forces its signal vertex to share the group of the chain source, so positive signals copy t_i and negative signals copy f_i. In a clause gadget, each signal vertex is opposite its corresponding $w_{j,k}$, and the K_4 must split $2 + 2$; therefore $w_{j,0}, w_{j,1}, w_{j,2}$ cannot all share one group, so neither can the three signal vertices. Defining $\alpha(x_i) = 1$ iff t_i lies in group 0 makes every normalized clause NAE-satisfied, hence every original clause is NAE-satisfied as well.

Solution extraction. Read the variable gadgets: set $\alpha(x_i) = 1$ iff t_i lies in group 0. □

Example: $n = 3$ variables, $m = 2$ clauses, target $K = 2$

Source: NAESatisfiability **Target:** PartitionIntoPerfectMatchings

```
$ pred create --example NAESatisfiability -o naesat.json
$ pred reduce naesat.json --to PartitionIntoPerfectMatchings/SimpleGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate naesat.json --config 1,1,0
```

Step 1 – Source instance. The canonical NAE-SAT fixture has clauses $C_1 = (x_1 \vee x_2 \vee x_3)$ and $C_2 = (\overline{x_1} \vee x_2 \vee \overline{x_3})$. The stored source witness is (1, 1, 0), so the clause truth patterns are $(1, 1, 0)$ and $(0, 1, 1)$, hence both clauses satisfy the NAE condition.

Step 2 – Lay out the gadgets. Each variable contributes 4 vertices, each clause contributes 6 signal vertices and 4 clause-gadget vertices, and each literal occurrence contributes one 2-vertex equality-chain pair. Concretely the target has 44 vertices and 51 edges: variable gadgets occupy vertices $0, ..., 11$, signal pairs occupy $12, ..., 23$, the two K_4 clause gadgets occupy $24, ..., 31$, and the equality-chain pairs occupy $32, ..., 43$.

Step 3 – Propagate the literal values. Because $x_1 = x_2 = 1$ and $x_3 = 0$, the three signal vertices for clause C_1 are in groups $(0, 0, 1) = (0, 0, 1)$, while the three signal vertices for clause C_2 are in groups $(1, 0, 0) = (1, 0, 0)$. The equality-chain pairs at $(32, 33), ..., (42, 43)$ carry the complementary groups needed to keep each copied signal synchronized with the appropriate t_i or f_i.

Step 4 – Verify the clause gadgets and extraction. The first K_4 gadget uses groups $(1, 1, 0, 0) = (1, 1, 0, 0)$, and the second uses $(0, 1, 1, 0) = (0, 1, 1, 0)$. Each gadget therefore splits $2 + 2$, so every clause gadget induces a perfect matching inside each group. Reading the truth assignment back from the variable vertices $(0, 4, 8)$ gives groups $(0, 0, 1) = (0, 0, 1)$, which extracts to (1, 1, 0) ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.208: (Exact Cover by 3-Sets → Subset Product) This $O(q^2 \log q)$ reduction [5] assigns the first $3q$ primes to universe elements. Each 3-element subset $C_j = \{a, b, c\}$ maps to $s_j = p_a \cdot p_b \cdot p_c$. The target product is $B = \prod_{i=0}^{3q-1} p_i$. By unique factorization, a sub-product equals B if and only if the selected subsets form an exact cover.
Overhead: `num_elements = num_sets`.

Proof: *Construction.* Let $(X, \mathcal{C})$ be an X3C instance with $|X| = 3q$ and $\mathcal{C} = \{C_1, ..., C_n\}$. Assign primes $p_0 = 2, p_1 = 3, ..., p_{3q-1}$. For each $C_j = \{a, b, c\}$, set $s_j = p_a \cdot p_b \cdot p_c$. Set target $B = \prod_{i=0}^{3q-1} p_i$.

Correctness. ($\Rightarrow$) An exact cover $\{C_{j_1}, ..., C_{j_q}\}$ partitions X, so $\prod s_{j_\ell} = \prod_{i=0}^{3q-1} p_i = B$. ($\Leftarrow$) If $\prod_{j:x_j=1} s_j = B$, unique factorization forces each prime to appear exactly once, so the selected subsets are pairwise disjoint and cover all elements.

Solution extraction. The X3C configuration equals the Subset Product configuration: select subset j iff $x_j = 1$. □

Example: $|U| = 6$, $|\mathcal{C}| = 3$ subsets

Source: ExactCoverBy3Sets **Target:** SubsetProduct

```
$ pred create --example ExactCoverBy3Sets -o x3c.json
$ pred reduce x3c.json --to SubsetProduct -o bundle.json
$ pred solve bundle.json
$ pred evaluate x3c.json --config 1,1,0
```

Step 1 – Source instance. The fixture has $U = \{0, ..., 5\}$ and three 3-sets: $C_0 = \{0, 1, 2\}$, $C_1 = \{3, 4, 5\}$, and $C_2 = \{0, 3, 4\}$. The witness (1, 1, 0) selects C_0 and C_1.

Step 2 – Recover the prime assignment from the concrete products. The target numbers are $s_0 = 30 = 2 \cdot 3 \cdot 5$, $s_1 = 1001 = 7 \cdot 11 \cdot 13$, and $s_2 = 154 = 2 \cdot 7 \cdot 11$. Thus the six universe elements are concretely labeled by the primes $(2, 3, 5, 7, 11, 13)$.

Step 3 – Form the Subset Product instance. The target product is $B = 30030 = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$. Selecting the first two source subsets therefore means selecting target numbers $(30, 1001)$.

Step 4 – Verify the witness pair. The selected sets C_0 and C_1 are disjoint and cover all six elements exactly once, and on the target side $30 \cdot 1001 = 30030$ while 154 is omitted. Because the configuration is unchanged, the target witness (1, 1, 0) extracts back to the same exact cover ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.209: (Subset Sum → Integer Expression Membership) This $O(n)$ reduction [75] builds for each element s_i a union node $(1 \cup (s_i + 1))$, then chains all unions via Minkowski sum. Set target $K = B + n$. Selecting $s_i + 1$ (right branch) encodes including element i in the subset. The expression tree has $4n - 1$ nodes.
Overhead: `num_union_nodes = num_elements`.

Proof: *Construction.* Given Subset Sum instance $(S = \{s_1, ..., s_n\}, B)$, for each s_i construct choice expression $c_i = (1 \cup (s_i + 1))$. Build the overall expression $e = c_1 + c_2 + ... + c_n$ (Minkowski sum chain). Set target $K = B + n$.

Correctness. ($\Rightarrow$) If $A' \subseteq S$ sums to B, choose $d_i = s_i + 1$ for $s_i \in A'$ and $d_i = 1$ otherwise. Then $\sum d_i = B + |A'| + (n - |A'|) = B + n = K$. ($\Leftarrow$) If $\sum d_i = K$ with $d_i \in \{1, s_i + 1\}$, let $A' = \{s_i : d_i = s_i + 1\}$. Then $\sum d_i = \sum_{s_i \in A'} s_i + n$, so $\sum_{s_i \in A'} s_i = B$.

Solution extraction. The Subset Sum configuration is the Integer Expression Membership configuration: $x_i = 1$ (right branch) means element i is selected. □

Example: 4 elements, target $B = 11$

Source: SubsetSum **Target:** IntegerExpressionMembership

```
$ pred create --example SubsetSum -o subsetsum.json
$ pred reduce subsetsum.json --to IntegerExpressionMembership -o bundle.json
$ pred solve bundle.json
$ pred evaluate subsetsum.json --config 0,1,1,0
```

Step 1 – Source instance. The Subset Sum fixture has sizes (1, 5, 6, 8) and target $B = 11$. The canonical source configuration (0, 1, 1, 0) selects the second and third items, so the source sum is $5 + 6 = 11$.

Step 2 – Build the choice sets inside the expression. Each source item contributes one union node $(1 \cup (s_i + 1))$, so the concrete choices are $(1 \cup 2)$, $(1 \cup 6)$, $(1 \cup 7)$, and $(1 \cup 9)$. With $n = 4$ union nodes, the target is shifted to $K = B + n = 15$.

Step 3 – Follow the canonical branch choices. The target configuration (0, 1, 1, 0) means left, right, right, left, so the chosen branch values are 1, 6, 7, and 1.

Step 4 – Verify the equality. The target-side sum is $1 + 6 + 7 + 1 = 15$, exactly matching K. The right branches occur in the same two positions as the chosen source elements, so extracting the target witness returns the original Subset Sum solution ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.210: (Exact Cover by 3-Sets → Minimum Weight Solution to Linear Equations) This $O(qn)$ reduction [5] builds the $3q \times n$ incidence matrix A where $A_{i,j} = 1$ iff element $u_i \in C_j$. Set right-hand side $b = (1, ..., 1)^\top$ and weight bound $K = q = |X|/3$. An exact cover corresponds to a binary solution of weight exactly q.
Overhead: `num_variables = num_sets`, `num_equations = universe_size`.

Proof: *Construction.* Let $(X, \mathcal{C})$ be an X3C instance with $|X| = 3q$ and $\mathcal{C} = \{C_1, ..., C_n\}$. Define n variables $y_1, ..., y_n$. Build matrix $A \in \{0,1\}^{3q \times n}$ with $A_{i,j} = 1$ iff $u_i \in C_j$. Each column has exactly 3 ones. Set $b = \mathbf{1} \in \mathbb{Z}^{3q}$ and $K = q$.

Correctness. ($\Rightarrow$) An exact cover selects q sets, each covering 3 elements with no overlap, giving $Ay = b$ with $y \in \{0,1\}^n$ and weight $q = K$. ($\Leftarrow$) If y has at most $K = q$ nonzero entries and $Ay = b$, summing all equations gives $3\sum_j y_j = 3q$, so $\sum y_j = q$. With $|\text{support}| \leq q$ and each column contributing 3 incidences, every row is hit by exactly one selected column, forcing each nonzero $y_j = 1$. The selected sets form an exact cover.

Solution extraction. Select subset j iff $y_j \neq 0$. □

Example: $|U| = 6$, $|\mathcal{C}| = 3$ subsets

Source: ExactCoverBy3Sets **Target:** MinimumWeightSolutionToLinearEquations

```
$ pred create --example ExactCoverBy3Sets -o x3c.json
$ pred reduce x3c.json --to MinimumWeightSolutionToLinearEquations -o bundle.json
$ pred solve bundle.json
$ pred evaluate x3c.json --config 1,1,0
```

Step 1 – Source instance. The X3C fixture has subsets $C_0 = \{0, 1, 2\}$, $C_1 = \{3, 4, 5\}$, and $C_2 = \{0, 3, 4\}$ over a universe of size 6, so $q = 2$.

Step 2 – Build the incidence matrix. The three columns correspond to C_0, C_1, and C_2. The six rows are $r_0 = (1, 0, 1)$, $r_1 = (1, 0, 0)$, $r_2 = (1, 0, 0)$, $r_3 = (0, 1, 1)$, $r_4 = (0, 1, 1)$, and $r_5 = (0, 1, 0)$, with right-hand side $b = (1, 1, 1, 1, 1, 1)$.

Step 3 – Check the linear equations on the witness. With $y = (1, 1, 0)$, the row products are $r_0 \cdot y = 1$, $r_1 \cdot y = 1$, $r_2 \cdot y = 1$, $r_3 \cdot y = 1$, $r_4 \cdot y = 1$, and $r_5 \cdot y = 1$. Hence $Ay = b$ for the stored target witness.

Step 4 – Check weight and extraction. The vector y has 2 nonzero entries, exactly the required $q = 2$. Those two nonzero positions select C_0 and C_1, so the target witness encodes the same exact cover as the source configuration $(1, 1, 0)$ ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.211: (k-SAT (k-ary) → Simultaneous Incongruences) This $O(n^2 + m)$ reduction [75] assigns each variable x_i a distinct prime $p_i \geq 5$, encoding TRUE as residue 1 and FALSE as residue 2 modulo p_i. All other residues are forbidden. Each clause is encoded via CRT as a single forbidden residue class modulo the product of its variables' primes. A satisfying assignment exists iff some integer avoids all forbidden classes. *Overhead:* `num_pairs = simultaneous_incongruences_num_incongruences`.

Proof: *Construction.* Given 3-SAT with n variables and m clauses, assign primes $p_1, ..., p_n \geq 5$. For each variable x_i, forbid residues $\{0, 3, 4, ..., p_i - 1\}$ modulo p_i, leaving only $\{1, 2\}$. For each clause C_j over variables $x_{i_1}, x_{i_2}, x_{i_3}$, compute the falsifying residue $r_k \in \{1, 2\}$ for each literal and use CRT to find R_j with $R_j \equiv r_k \bmod p_{i_k}$ for $k = 1, 2, 3$. Forbid R_j modulo $M_j = p_{i_1} p_{i_2} p_{i_3}$.

Correctness. ($\Rightarrow$) A satisfying assignment τ defines residues $r_i \in \{1, 2\}$ per variable. By CRT, some integer x has these residues. It avoids all variable-forbidden classes and all clause-forbidden classes (since at least one literal is true, the residue triple differs from the falsifying triple). ($\Leftarrow$) Any feasible x has $x \bmod p_i \in \{1, 2\}$ for all i. Define $\tau(x_i) =$ TRUE if residue 1, FALSE if 2. If a clause were false, x would match its forbidden CRT class – contradiction.

Solution extraction. Set $\tau(x_i) =$ TRUE if $x \bmod p_i = 1$, FALSE if $x \bmod p_i = 2$. □

Example: $n = 2$ variables, $m = 2$ clauses

Source: KSatisfiability **Target:** SimultaneousIncongruences

```
$ pred create --example KSatisfiability/K3 -o ksat.json
$ pred reduce ksat.json --to SimultaneousIncongruences -o bundle.json
$ pred solve bundle.json
$ pred evaluate ksat.json --config 1,1
```

Step 1 – Source instance. The two clauses are $C_1 = (x_1 \vee x_2 \vee x_2)$ and $C_2 = (\overline{x_1} \vee x_2 \vee x_2)$. The canonical satisfying assignment is (1, 1).

Step 2 – Assign primes and variable residue constraints. With two variables, the reduction uses primes 3 and 5. The variable-generated forbidden pairs are $(3, 3)$, $(5, 5)$, $(3, 5)$, and $(4, 5)$, leaving only residues 1 and 2 modulo 3 and modulo 5.

Step 3 – Encode the clauses by CRT. Clause C_1 is false only when $(x_1, x_2) = (0, 0)$, i.e.
residues $(2 \bmod 3, 2 \bmod 5)$, which yields the forbidden pair $(2, 15)$. Clause C_2 is false only when $(x_1, x_2) = (1, 0)$, i.e.
residues $(1 \bmod 3, 2 \bmod 5)$, which yields $(7, 15)$.

Step 4 – Verify the target witness. The stored integer is $x = 1$. It satisfies $x \equiv 1 \bmod 3$ and $x \equiv 1 \bmod 5$, so it decodes to the source assignment $(1, 1)$. It also avoids all six forbidden classes: $1\neg \equiv 0 \bmod 3$, $1\neg \equiv 0, 3, 4 \bmod 5$, and $1\neg \equiv 2, 7 \bmod 15$ ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.212: (Partition → Sequencing to Minimize Tardy Task Weight) This $O(n)$ reduction [1] maps each element a_i to a task with length $l(t_i) = a_i$, weight $w(t_i) = a_i$, and common deadline $T = B/2$. The tardiness bound is $K = T$. Tasks scheduled before T are on-time; those after are tardy. A balanced partition exists iff total tardy weight can be at most K.
Overhead: `num_tasks = num_elements`.

Proof: *Construction.* Given Partition instance $A = \{a_1, ..., a_n\}$ with total B. If B is odd, output a trivially infeasible instance (all deadlines 0, $K = 0$). If B is even, set $T = B/2$. For each a_i, create task t_i with $l(t_i) = w(t_i) = a_i$ and deadline $d(t_i) = T$. Set bound $K = T$.

Correctness. ($\Rightarrow$) A balanced partition A', A'' with sums T each: schedule A' first (on-time, total time T), then A'' (tardy, weight $T = K$). ($\Leftarrow$) If tardy weight $\leq K = T$, then on-time tasks fit before T and sum to $\leq T$, while tardy tasks have weight $B - \sum_{\text{on-time}} \leq T$, forcing on-time sum $= T$. This yields a balanced partition.

Solution extraction. On-time tasks (completing by T) form one partition half ($x_i = 0$), tardy tasks the other ($x_i = 1$). □

Example: 6 elements, total $= 10$

Source: Partition **Target:** SequencingToMinimizeTardyTaskWeight

```
$ pred create --example Partition -o partition.json
$ pred reduce partition.json --to SequencingToMinimizeTardyTaskWeight -o bundle.json
$ pred solve bundle.json
$ pred evaluate partition.json --config 1,0,0,1,0,0
```

Step 1 – Source instance. The Partition fixture has sizes (3, 1, 1, 2, 2, 1) with total 10, so the target deadline is $T = 5$. The canonical source vector (1, 0, 0, 1, 0, 0) splits the multiset into sums 5 and 5.

Step 2 – Build the task table.

task	l_j	w_j	d_j
t_0	3	3	5
t_1	1	1	5
t_2	1	1	5
t_3	2	2	5
t_4	2	2	5
t_5	1	1	5

Step 3 – Follow the canonical schedule. The target permutation (1, 2, 4, 5, 0, 3) schedules tasks in the order $t_1, t_2, t_4, t_5, t_0, t_3$. The completion times are $1, 2, 4, 5, 8, 10$, so t_1, t_2, t_4, t_5 are on time and t_0, t_3 are tardy.

Step 4 – Compute tardy weight and recover the partition. Because weights equal lengths here, the tardy weight is $w_0 + w_3 = 3 + 2 = 5$, and the on-time tasks have total size 5 while the tardy tasks have total size 5. Extracting the schedule therefore returns the balanced partition (1, 0, 0, 1, 0, 0) ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.213: (Partition → Open Shop Scheduling) This $O(k)$ reduction [114] creates $k + 1$ jobs on 3 machines: k element jobs with $p_{j,i} = a_j$ on all machines, plus one special job with $p = Q = S/2$. The target makespan is $3Q$. A balanced partition exists iff the open shop can achieve makespan $\leq 3Q$.
Overhead: `num_jobs = num_elements + 1`, `num_machines = 3`.

Proof: *Construction.* Given Partition instance $A = \{a_1, ..., a_k\}$ with total S and $Q = S/2$. Set $m = 3$ machines. For each a_j, create element job J_j with $p_{j,1} = p_{j,2} = p_{j,3} = a_j$. Create special job J_{k+1} with $p_{k+1,i} = Q$ on all machines. Deadline $D = 3Q$.

Correctness. ($\Rightarrow$) With a balanced partition I_1, I_2, schedule the special job consecutively on machines 1, 2, 3 during $[0, Q), [Q, 2Q), [2Q, 3Q)$. Use a rotated assignment for I_1 and I_2 jobs to fill the remaining idle blocks, each of length Q. ($\Leftarrow$) With makespan $\leq 3Q$, the special job alone needs $3Q$ elapsed time, so it tiles $[0, 3Q)$ exactly. On each machine, element jobs fill two idle blocks of length Q each. The jobs in one block sum to Q, giving a balanced partition.

Solution extraction. Identify the special job's position on machine 1. Element jobs in one idle block form a subset summing to Q. □

Example: 3 elements, $m = 3$ machines

Source: Partition **Target:** OpenShopScheduling

```
$ pred create --example Partition -o partition.json
$ pred reduce partition.json --to OpenShopScheduling -o bundle.json
$ pred solve bundle.json
$ pred evaluate partition.json --config 0,0,1
```

Step 1 – Source instance. The Partition fixture has sizes (1, 2, 3), total 6, and half-sum $Q = 3$. The canonical source vector (0, 0, 1) gives subset sums 3 and 3.

Step 2 – Build the open-shop job table.

job	$p_{\{j,1\}}$	$p_{\{j,2\}}$	$p_{\{j,3\}}$
J_0	1	1	1
J_1	2	2	2
J_2	3	3	3

job	$p_{\{j,1\}}$	$p_{\{j,2\}}$	$p_{\{j,3\}}$
J_3	3	3	3

The first three jobs come from the partition elements, and the special job J_3 has processing time $Q = 3$ on every machine.

Step 3 – Decode the canonical machine orders. The target configuration (0, 1, 2, 3, 0, 1, 2, 3, 2, 3, 0, 1) splits into $M_1 = (0, 1, 2, 3)$, $M_2 = (0, 1, 2, 3)$, and $M_3 = (2, 3, 0, 1)$. On machine M_3, job J_2 occupies $[0, 3)$ and the special job J_3 starts exactly at time $Q = 3$, so the prefix before the special job contains precisely job J_2.

Step 4 – Verify extraction and makespan. Because only J_2 finishes on M_3 by time Q, the extracted source vector is (0, 0, 1), i.e.
subset sum 3 versus 3. Evaluating the stored machine orders gives a concrete makespan of 12, so the `load-example()` fixture shows both the machine assignment and the middle-machine split used for extraction ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.214: (NAE-SAT → Max-Cut (weighted)) This implemented reduction sets $M = m + 1$, creates two literal vertices per variable, and adds one unit-weight edge for every literal pair inside a clause. When every clause has 3 literals, each clause gadget is a triangle, so the construction is the usual NAE-SAT-to-Max-Cut graph on $2n$ vertices with $n + 3m$ edges.
Overhead: `num_vertices = 2 * num_vars`, `num_edges = num_vars + num_literal_pairs`.

Proof: *Construction.* Let φ have n variables and m clauses, and set $M = m + 1$. For each variable x_i, create a positive literal vertex $p_i = 2i$ and a negative literal vertex $n_i = 2i + 1$, joined by one weight-M edge. For each clause C_j, add a unit-weight edge between every pair of literal vertices appearing in C_j. In particular, if every clause has three literals, each clause becomes a unit-weight triangle.

Correctness. Assume from here on that each clause has exactly three literals, matching the canonical fixture. Then every clause gadget is a triangle.

($\Rightarrow$) Let α be a NAE-satisfying assignment. Put p_i and n_i on opposite sides of the cut according to α, so every variable edge is cut and contributes M. In each clause triangle, at least one literal is true and at least one is false, so the three vertices split 1-2 across the cut and contribute exactly 2. Therefore the cut weight is $nM + 2m = n(m + 1) + 2m$.

($\Leftarrow$) Suppose a cut has weight at least $n(m + 1) + 2m$. The m clause triangles contribute at most $2m$ in total, so the variable edges must contribute at least $n(m + 1)$. Since each variable edge contributes at most $M = m + 1$, all n variable edges are cut. Thus p_i and n_i lie on opposite sides for every variable, and the cut defines a consistent Boolean assignment by reading the side of p_i. The remaining $2m$ weight must come from the clause triangles, so each triangle contributes exactly 2 and therefore has vertices on both sides of the cut. Hence every clause contains both a true and a false literal, and the extracted assignment NAE-satisfies φ. Because a satisfying instance attains $n(m + 1) + 2m$, every optimal cut of the target has this form.

Solution extraction. Read the positive literal vertices: $x_i = 1$ iff vertex $2i$ lies on side 1 of the cut. □

Example: $n = 3$ variables, $m = 2$ clauses, $M = 3$

Source: NAESatisfiability **Target:** MaxCut

```
$ pred create --example NAESatisfiability -o naesat.json
$ pred reduce naesat.json --to MaxCut/SimpleGraph/i32 -o bundle.json
```

```
$ pred solve bundle.json
$ pred evaluate naesat.json --config 1,0,1
```

Step 1 – Source instance. NAE-SAT with $n = 3$ variables and $m = 2$ clauses. The implementation uses forcing weight $M = m + 1 = 3$.

Step 2 – Construct the weighted graph. Variable gadgets contribute 3 heavy edges of weight M. Because the canonical fixture has 3 literals per clause, each clause contributes one unit-weight triangle, so the target has 6 unit-weight clause edges and 9 edges total on 6 vertices.

Step 3 – Verify the canonical witness. Source assignment (1, 0, 1) induces target cut (1, 0, 0, 1, 1, 0). All 3 heavy edges are cut, and each of the 2 clause triangles has a 1-2 split contributing 2, so the total cut weight is 13 ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.215: (Three-Dimensional Matching → 3-Partition) This $O(t^2)$ reduction [5] first checks whether every coordinate of W, X, and Y appears in some triple; uncovered coordinates yield a fixed infeasible 3-Partition instance. Otherwise it composes the classical 3DM → ABCD-Partition, ABCD-Partition → 4-Partition, and 4-Partition → 3-Partition constructions, producing $24t^2 - 3t$ integers arranged into $8t^2 - t$ triples.
Overhead: `num_elements = 24 * num_triples * num_triples + -1 * 3 * num_triples`, `num_groups = 8 * num_triples * num_triples + -1 * num_triples`.

Proof: *Construction.* Let the source instance have universe size q and triples $m_l = \left(w_{a_l}, x_{b_l}, y_{c_l}\right)$ for $l = 0, ..., t-1$. If some coordinate of $W \cup X \cup Y$ is absent from all triples, the source instance is trivially NO, so the implementation returns a fixed infeasible 3-Partition instance with sizes $(6, 6, 6, 6, 7, 9)$ and bound 20.

Otherwise set $r = 32q$ and $T_1 = 40r^4$. For each triple create

$$u_l = 10r^4 - c_l r^3 - b_l r^2 - a_l r$$

, one B-item $10r^4 + a_l r$ on the first occurrence of coordinate a_l and $11r^4 + a_l r$ on later occurrences, one C-item $10r^4 + b_l r^2$ on the first occurrence of b_l and $11r^4 + b_l r^2$ later, and one D-item $10r^4 + c_l r^3$ on the first occurrence of c_l and $8r^4 + c_l r^3$ later. This is the ABCD-Partition instance.

Tag the four classes modulo 16:

$$a_i' = 16a_i + 1, b_i' = 16b_i + 2, c_i' = 16c_i + 4, d_i' = 16d_i + 8$$

with target $T_2 = 16T_1 + 15$. Enumerate the resulting $4t$ tagged numbers as $a_0, ..., a_{4t-1}$.

For every tagged number create one regular element $w_i = 4(5T_2 + a_i) + 1$. For every unordered pair $i < j$, create pairing elements

$$u_{ij} = 4\big(6T_2 - a_i - a_j\big) + 2$$

and

$$u_{ij}' = 4\big(5T_2 + a_i + a_j\big) + 2$$

. Finally add $8t^2 - 3t$ filler elements of size $20T_2$ and set the 3-Partition bound to $B = 64T_2 + 4$.

Correctness. The fixed preprocessing case is immediate: an uncovered coordinate makes the 3DM instance infeasible, and $(6, 6, 6, 6, 7, 9; 20)$ is a valid infeasible 3-Partition instance.

Assume now that every coordinate appears at least once.

($\Rightarrow$) Given a perfect matching M', form one ABCD group for every source triple. If $m_l \in M'$, combine u_l with the unique first-occurrence B, C, and D items of coordinates (a_l, b_l, c_l); otherwise combine u_l with the corresponding later-occurrence dummy items. Because $r = 32q$ prevents carries between the r, r^2, r^3, and r^4 digits, every such group sums to T_1, so the tagged instance has a 4-partition. For each tagged 4-set choose any two members a_i, a_j and let the other two be a_k, a_l. Then $\{w_i, w_j, u_{ij}\}$ and $\{w_k, w_l, u'_{ij}\}$ both sum to B. Every pairing gadget not used this way joins one filler in a triple $\{u_{ij}, u'_{ij}, 20T_2\}$. Hence the produced 3-Partition instance is feasible.

($\Leftarrow$) In any feasible target solution every number lies strictly between $\frac{B}{4}$ and $\frac{B}{2}$, so the partition really is into triples. Modulo 4, regular numbers are congruent to 1, pairing numbers to 2, and fillers to 0. Therefore every triple is either of type $(1, 1, 2)$ or $(0, 2, 2)$. The $(0, 2, 2)$ triples identify the unused pairing gadgets, leaving a family of $(1, 1, 2)$ triples that reconstructs a 4-partition of the tagged numbers. Since $1 + 2 + 4 + 8 \equiv 15 \bmod 16$, every recovered tagged 4-set contains exactly one former A-, B-, C-, and D -item. The carry-free base-r encoding then forces each ABCD group to be either a real group (all first occurrences) or a dummy group (all later occurrences). The real groups pick exactly q source triples, one for each coordinate of W, X, and Y, so they form a perfect 3-dimensional matching.

Solution extraction. Reverse the 4-Partition $\rightarrow$ 3-Partition gadget by pairing each triple containing some u_{ij} with the unique triple containing the matching u'_{ij}. This recovers the tagged 4-set. Undo the mod-16 tags to obtain one ABCD group, discard every dummy group whose B, C, and D items are not first occurrences, and read the selected source triple from the surviving A-item. □

Example: $q = 1$, $t = 1$, **target has 21 numbers**

Source: ThreeDimensionalMatching **Target:** ThreePartition

```
$ pred create --example ThreeDimensionalMatching -o three-dimensional-matching.json
$ pred reduce three-dimensional-matching.json --to ThreePartition -o bundle.json
$ pred solve bundle.json
$ pred evaluate three-dimensional-matching.json --config 1
```

Step 1 – Source instance. The canonical source has $q = 1$ and a single triple $M = \{(0, 0, 0)\}$, so the witness (1) selects the only available triple and is therefore a perfect 3-dimensional matching ✓.

Step 2 – Encode the ABCD and tagged 4-partition numbers. Here $r = 32q = 32$ and $T_1 = 40r^4 = 41943040$. Because every coordinate is 0 and occurs once, the ABCD numbers are all $10r^4 = 10485760$. After the mod-16 tags, the 4-partition numbers become $(167772161, 167772162, 167772164, 167772168)$ and sum to $T_2 = 16T_1 + 15 = 671088655$.

Step 3 – Build the 3-partition gadget. From the 4 tagged numbers the construction creates 21 target numbers: 4 regular numbers, 12 pairing numbers, and 5 fillers. The target bound is $B = 64T_2 + 4 = 42949673924$, matching the exported instance's bound 42949673924. The canonical target witness is (0, 0, 1, 1, 0, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 2, 3, 4, 5, 6): groups 0 and 1 are the non-filler triples, and the remaining 5 groups each contain one filler together with one unused pairing pair.

Step 4 – Verify the witness. The target configuration partitions all 21 numbers into 7 triples summing to $B = 42949673924$ ✓. Reversing the gadget recovers the unique tagged 4-set, whose B, C, and D members are all first occurrences, so the extracted source witness is again (1) ✓.

Multiplicity: The fixture stores one canonical witness. In this $q = 1$ instance there is only one source matching, but the target still admits multiple equivalent 3-partition witnesses because any choice of one pairing gadget to split the unique 4-set yields a valid solution.

Rule 3.216: (Three-Dimensional Matching $\rightarrow$ Integer Linear Programming) Direct ILP formulation: one binary variable per triple, with $3q$ equality constraints ensuring each element of W, X, and Y is covered

exactly once. The ILP has t variables and $3q$ constraints.
Overhead: `num_vars = num_triples`, `num_constraints = 3 * universe_size`.

Proof: *Construction.* Given a 3DM instance with universe size q and triples $m_0, ..., m_{t-1}$ where $m_l = \left(w_{a_l}, x_{b_l}, y_{c_l}\right)$. Create binary variables $x_0, ..., x_{t-1}$. For each element $e \in \{0, ..., q-1\}$ in W, add constraint $\sum_{l:a_l=e} x_l = 1$. Similarly for each element in X and Y. The ILP is:

$$\begin{aligned} \text{find} \quad & \boldsymbol{x} \\ \text{subject to} \quad & \sum_{l:a_l=e} x_l = 1 \quad \forall e \in \{0, ..., q-1\} \quad \text{(W-coverage)} \\ & \sum_{l:b_l=e} x_l = 1 \quad \forall e \in \{0, ..., q-1\} \quad \text{(X-coverage)} \\ & \sum_{l:c_l=e} x_l = 1 \quad \forall e \in \{0, ..., q-1\} \quad \text{(Y-coverage)} \\ & x_l \in \{0, 1\} \quad \forall l \in \{0, ..., t-1\} \end{aligned}$$

.

Correctness. ($\Rightarrow$) A perfect 3-dimensional matching selects q triples covering every element exactly once. Setting $x_l = 1$ for selected triples satisfies all $3q$ equality constraints. ($\Leftarrow$) Any binary feasible solution selects a set of triples in which every element of W, X, and Y appears exactly once, which is a perfect 3-dimensional matching.

Solution extraction. Return the ILP solution vector directly: $x_l = 1$ iff triple m_l is selected. □

Example: $q = 3$, $t = 5$ triples → ILP with 5 variables and 9 constraints

Source: ThreeDimensionalMatching **Target:** ILP

```
$ pred create --example ThreeDimensionalMatching -o three-dimensional-matching.json
$ pred reduce three-dimensional-matching.json --to ILP/bool -o bundle.json
$ pred solve bundle.json
$ pred evaluate three-dimensional-matching.json --config 1,1,1,0,0
```

Step 1 – Source instance. The canonical source has universe size $q = 3$ and $t = 5$ triples: (0, 1, 2), (1, 0, 1), (2, 2, 0), (0, 0, 0), (1, 2, 2). The stored witness (1, 1, 1, 0, 0) selects triples 0, 1, 2, which cover every element of W, X, and Y exactly once ✓.

Step 2 – Build the ILP. One binary variable x_j per triple ($j = 0, ..., 4$). For each of the $3q = 9$ elements across the three sets W, X, Y, add an equality constraint requiring that the sum of x_j over all triples containing that element equals 1. This yields 9 constraints and 5 variables.

Step 3 – Verify a solution. The target configuration (1, 1, 1, 0, 0) is identical to the source configuration because the mapping is one variable per triple with identity extraction. Each element-coverage constraint sums to exactly 1 ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.217: (3-Partition → Resource Constrained Scheduling) Each element becomes a unit-length task requiring a_i units of a shared resource with bound B. With 3 processors and deadline m, every slot receives exactly 3 tasks summing to B.
Overhead: `num_tasks = num_elements`.

Proof: *Construction.* Given (S, B) with $|S| = 3m$ and $\frac{B}{4} < a_i < \frac{B}{2}$. Create $3m$ unit-length tasks with resource requirement $r_i = a_i$, $p = 3$ processors, resource bound B, deadline $D = m$.

Correctness. ($\Rightarrow$) A valid 3-partition assigns each triple to a time slot; each slot uses exactly B resource units. ($\Leftarrow$) $3m$ tasks in m slots with $p = 3$: every slot has exactly 3 tasks. Resource bound B with total mB: each slot sums to exactly B. Size constraints prevent fewer or more than 3 elements per slot.

Solution extraction. Task t_i assigned to slot k means element a_i belongs to group k. □

Example: 6 elements, $B = 15$

Source: ThreePartition **Target:** ResourceConstrainedScheduling

```
$ pred create --example ThreePartition -o threepartition.json
$ pred reduce threepartition.json --to ResourceConstrainedScheduling -o bundle.json
$ pred solve bundle.json
$ pred evaluate threepartition.json --config 0,0,0,1,1,1
```

Step 1 – Source instance. The canonical Three-Partition instance has $3m = 6$ elements with sizes (4, 5, 6, 4, 6, 5) and bound $B = 15$, so $m = 2$ groups are required.

Step 2 – Build the Resource-Constrained Scheduling instance. Each element a_i becomes a unit-length task with resource requirement $r_i = a_i$. The reduction sets $p = 3$ processors, a single resource with bound 15, and deadline $D = 2$. The target has 6 tasks with resource requirements (4, 5, 6, 4, 6, 5).

Step 3 – Verify the canonical witness. The source config (0, 0, 0, 1, 1, 1) assigns elements to groups:
- Slot 0: elements {0, 1, 2} with sizes $4 + 5 + 6 = 15 = B$ ✓
- Slot 1: elements {3, 4, 5} with sizes $4 + 6 + 5 = 15 = B$ ✓

Each slot has exactly 3 tasks and each slot's resource usage sums to B. The target config is (0, 0, 0, 1, 1, 1), matching the source config since task t_i is assigned to the same slot as element a_i.

Multiplicity: The fixture stores one canonical witness. Other valid 3-partitions (if any) would yield equally valid schedules.

Rule 3.218: (3-Partition $\rightarrow$ Sequencing with Release Times and Deadlines) $m - 1$ filler tasks with tight release windows partition the timeline into m slots of width B. Element tasks must fill these slots, and size constraints force exactly 3 elements per slot.
Overhead: `num_tasks = num_elements + num_groups + -1 * 1`.

Proof: *Construction.* Given (S, B) with $|S| = 3m$. Create $3m$ element tasks with $p_i = a_i$, $r_i = 0$, $d_i = H$ where $H = m(B + 1) - 1$. Add $m - 1$ filler tasks with $p = 1$, $r_j = (j + 1)B + j$, $d_j = (j + 1)B + j + 1$.

Correctness. ($\Rightarrow$) A valid 3-partition fills each slot of width B with exactly 3 elements; fillers are placed in their tight windows. ($\Leftarrow$) Fillers pinned to unit windows create m slots of width B; total element work mB fills all slots exactly; size constraints force 3 elements per slot.

Solution extraction. Decode the Lehmer code, simulate the schedule tracking start times; assign elements to groups by $\lfloor \frac{\text{start}}{B+1} \rfloor$. □

Example: 3-Partition with $3m = 6$ elements and $B = 15$ mapped to 7 sequencing tasks

Source: ThreePartition **Target:** SequencingWithReleaseTimesAndDeadlines

```
$ pred create --example ThreePartition -o tp.json
$ pred reduce tp.json --to SequencingWithReleaseTimesAndDeadlines -o bundle.json
$ pred solve bundle.json
$ pred evaluate tp.json --config 0,0,0,1,1,1
```

Step 1 – Source instance. The canonical 3-Partition instance has $3m = 6$ elements with sizes (4, 5, 6, 4, 6, 5) and bound $B = 15$. Since $m = 2$, we must partition the elements into 2 groups, each summing to B.

Step 2 – Construct element tasks. Each element a_i becomes a task with processing time $p_i = a_i$, release time $r_i = 0$, and deadline $d_i = H$ where $H = m(B+1) - 1 = 2 \cdot (15+1) - 1 = 31$. This gives 6 element tasks with lengths (4, 5, 6, 4, 6, 5).

Step 3 – Construct filler tasks. Add $m - 1 = 1$ filler task(s). Filler j has length 1, release time $r_j = (j+1)B + j = 15$, and deadline $d_j = r_j + 1 = 16$. This tight window pins each filler to a single time unit, splitting the timeline into m slots of width $B = 15$.

Step 4 – Verify a solution. The source witness assigns elements to groups: [0, 0, 0, 1, 1, 1]. Group 0 contains elements with sizes (4, 5, 6), summing to $15 = B$ ✓. Group 1 contains sizes (4, 6, 5), summing to $15 = B$ ✓. The target Lehmer code is [0, 0, 0, 3, 0, 0, 0]: element tasks fill slot $[0, B)$, the filler occupies its tight window $[B, B+1)$, and remaining elements fill slot $[B+1, 2B+1)$.

Multiplicity: The fixture stores one canonical witness. A second valid partition (swapping groups) exists, but both map to distinct Lehmer codes.

Rule 3.219: (3-Partition → Sequencing to Minimize Weighted Tardiness) High-weight filler tasks with tight deadlines force zero-tardiness schedules to leave exactly m slots of width B for element tasks. Size constraints ensure 3 elements per slot.
Overhead: `num_tasks = num_elements + num_groups + -1 * 1`.

Proof: *Construction.* Given (S, B) with $|S| = 3m$. Horizon $H = mB + (m-1)$, filler weight $W_f = mB + 1$. Element tasks: $p_i = a_i$, $w_i = 1$, $d_i = H$. Filler tasks $(m-1)$: $p = 1$, $w = W_f$, $d_j = (j+1)B + (j+1)$. Tardiness bound $K = 0$.

Correctness. ($\Rightarrow$) A valid 3-partition schedules each triple in a slot between fillers, achieving zero tardiness. ($\Leftarrow$) Zero tardiness forces fillers to their tight deadlines, creating m slots of width B; element tasks fill them exactly with 3 per slot.

Solution extraction. Decode Lehmer code; scan left to right incrementing group index at each filler. □

Example: $m = 2$ groups, $B = 20$, $3m = 6$ elements

Source: ThreePartition **Target:** SequencingToMinimizeWeightedTardiness

```
$ pred create --example ThreePartition -o threepartition.json
$ pred reduce threepartition.json --to SequencingToMinimizeWeightedTardiness -o bundle.json
$ pred solve bundle.json
$ pred evaluate threepartition.json --config 0,0,0,1,1,1
```

Step 1 – Source instance. The canonical ThreePartition instance has $3m = 6$ elements with sizes (7, 7, 6, 7, 7, 6) and bound $B = 20$. The total sum is $40 = mB = 2 \times 20$. Each element satisfies $B/4 < a_i < B/2$, i.e.
$6 \leq a_i \leq 9$.**Step 2 – Construct target tasks.** The horizon is $H = mB + (m-1) = 41$, and the filler weight is $W_f = mB + 1 = 41$. The reduction creates 7 tasks:

- **6 element tasks** with lengths (7, 7, 6, 7, 7, 6), weights (1, 1, 1, 1, 1, 1), and deadlines (41, 41, 41, 41, 41, 41) (all equal to H).
- **1 filler task** with length 1, weight $W_f = 41$, and deadline 21 (tight: $(1) \cdot B + 1 = 21$).

The tardiness bound is $K = 0$.**Step 3 – Verify a solution.** The source assignment (0, 0, 0, 1, 1, 1) places elements $\{0, 1, 2\}$ (sizes $7, 7, 6$, sum $= 20$) in group 0 and elements $\{3, 4, 5\}$ (sizes $7, 7, 6$, sum $= 20$) in group 1. Both groups sum to $B = 20$ ✓. The target Lehmer code is (0, 0, 0, 3, 0, 0, 0), which

decodes to the schedule: tasks $0, 1, 2$ (slot 0, total length 20), then filler (length 1, completes at $21 \leq 21$ ✓), then tasks $3, 4, 5$ (slot 1, completes at 41). All element deadlines are 41 ✓, and the filler meets its tight deadline. Zero tardiness achieved ✓.**Multiplicity:** The fixture stores one canonical witness. This instance admits other valid orderings within each slot (e.g.
permuting elements 0, 1, 2 within slot 0), but the group assignment is unique up to slot relabeling.

Rule 3.220: (3-Partition → Job-Shop Scheduling) On 2 machines, $m - 1$ long separator jobs on machine 0 force element jobs into m windows of width B. Size constraints ensure 3 elements per window.
Overhead: `num_jobs = num_elements + num_groups + -1 * 1`, `num_tasks = 2 * num_elements + num_groups + -1 * 1`.

Proof: *Construction.* Given (S, B) with $|S| = 3m$, $L = mB + 1$, $D = mB + (m - 1)L$. Element jobs $(3m)$: two tasks (machine $0, a_i$) then (machine $1, a_i$). Separator jobs $(m - 1)$: single task (machine $0, L$).

Correctness. ($\Rightarrow$) A valid 3-partition interleaves element groups with separators on machine 0, achieving makespan D. ($\Leftarrow$) Separators of length $L > mB$ create impassable barriers; remaining budget mB split into m windows; size constraints force 3 elements per window.

Solution extraction. Decode machine 0 Lehmer code; walk permutation incrementing group at each separator. □

Example: $3m = 3$ **elements,** $B = 15$**, 2 machines**

Source: ThreePartition **Target:** JobShopScheduling

```
$ pred create --example ThreePartition -o threepartition.json
$ pred reduce threepartition.json --to JobShopScheduling -o bundle.json
$ pred solve bundle.json
$ pred evaluate threepartition.json --config 0,0,0
```

Step 1 – Source instance. The canonical 3-Partition instance has $3m = 3$ elements with sizes $S = (4, 5, 6)$ and bound $B = 15$. Since $m = 1$, there are $m - 1 = 0$ separators, separator length $L = mB + 1 = 16$, and deadline $D = mB + (m - 1)L = 15$.

Step 2 – Construct jobs. Each element a_i becomes an element job with two tasks: (machine $0, a_i$) then (machine $1, a_i$). This gives 3 jobs on 2 processors. The target JSS instance has jobs: [(0,4), (1,4)]; [(0,5), (1,5)]; [(0,6), (1,6)].

Step 3 – Verify a solution. The source config (0, 0, 0) assigns all 3 elements to group 0. With $m = 1$ and no separators, any ordering of the 3 element tasks on each machine is valid. The target Lehmer code (0, 0, 0, 0, 0, 0) encodes identity orderings on both machines. The resulting makespan is $\sum a_i = 15 = B = 15 \leq D$ ✓, and all 3 elements land in a single processor slot containing exactly 3 elements ✓.

Multiplicity: The fixture stores one canonical witness. With $m = 1$ there is only one valid group assignment (all elements in group 0); the $3! \times 3!$ task orderings on the two machines yield multiple target configs, but only one source partition.

Rule 3.221: (Max-Cut (weighted) → Minimum Cut Into Bounded Sets (weighted)) Invert edge weights relative to $w_{\max}$ on a complete graph K_N with $N = 2n'$. A minimum balanced bisection in the inverted graph corresponds to a maximum cut in the original.
Overhead: `num_vertices = 2 * num_vertices + 2`, `num_edges = (num_vertices + 1) * (2 * num_vertices + 1)`.

Proof: *Construction.* Given $G = (V, E, w)$ with $n = |V|$. Set $n' = n + (n \bmod 2)$, $N = 2n'$, $w_{\max} = 1 + \max_{e \in E} w(e)$. Build K_N with $\tilde{w}(i, j) = w_{\max} - w(i, j)$ for edges in E, else $w_{\max}$. Designate $s = n'$, $t = n' + 1$, bound $b = n'$.

Correctness. ($\Rightarrow$) A max-cut extended to a balanced bisection gives a feasible target instance. ($\Leftarrow$) Minimizing $\tilde{w}$-cut cost is equivalent to maximizing original weight crossing the cut, since $\tilde{w} = w_{\max} - w$.

Solution extraction. Return the first n entries of the target assignment. □

Example: Triangle graph ($n = 3$, $|E| = 3$, unit weights) mapped to K_8

Source: MaxCut **Target:** MinimumCutIntoBoundedSets

```
$ pred create --example MaxCut/SimpleGraph/i32 -o maxcut.json
$ pred reduce maxcut.json --to MinimumCutIntoBoundedSets/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate maxcut.json --config 0,0,1
```

Step 1 – Source instance. The source MaxCut instance is a triangle $G = (V, E)$ with $n = 3$ vertices, $|E| = 3$ edges, and unit weights $w = (1, 1, 1)$. A maximum cut partitions vertices into two sides to maximize crossing-edge weight; here the optimum is 2 (any single vertex versus the other two).

Step 2 – Pad to even vertex count. Since $n = 3$ is odd, set $n' = n + 1 = 4$. The target complete graph has $N = 2n' = 8$ vertices, giving $8 \cdot (8-1)/2 = 28$ edges.

Step 3 – Invert weights on K_8. Compute $w_{\max} = 1 + \max w(e) = 2$. For each original edge $(i, j) \in E$, the inverted weight is $\tilde{w}(i, j) = w_{\max} - w(i, j) = 2 - 1 = 1$. All other edges (including those to padding vertices) receive weight $w_{\max} = 2$. Designate source $s = 4$, sink $t = 5$, size bound $b = 4$. The target edge weights are (1, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2).

Step 4 – Verify a solution. The canonical source witness is (0, 0, 1): vertices 0, 1 on side 0 and vertex 2 on side 1, cutting 2 of 3 edges (max cut value = 2). The target witness is (0, 0, 1, 0, 0, 1, 1, 1). Check: (1) the first $n = 3$ entries match the source partition ✓ (2) source vertex $s = 4$ and sink vertex $t = 5$ are on opposite sides ✓ (3) each side has exactly $b = 4$ vertices (balanced bisection) ✓.

Multiplicity: The fixture stores one canonical witness. The triangle has 3 maximum cuts of value 2 (isolate any one vertex); padding vertices can be assigned to balance both sides, yielding multiple valid target configurations.

Rule 3.222: (Hamiltonian Path → Isomorphic Spanning Tree) This $O(n)$ reduction [5] passes the graph through unchanged and sets the target tree $T = P_n$ (the path on n vertices). A Hamiltonian path in G is exactly a spanning tree isomorphic to P_n: both are connected, acyclic, span all vertices, and have maximum degree 2. The reduction is size-preserving.
Overhead: `num_vertices = num_vertices`, `num_graph_edges = num_edges`, `num_tree_edges = num_vertices + -1 * 1`.

Proof: *Construction.* Given $G = (V, E)$ with $|V| = n$, set the host graph to G and the target tree to $T = P_n = (\{t_0, ..., t_{n-1}\}, \{\{t_i, t_{i+1}\} : 0 \le i \le n-2\})$.

Correctness. ($\Rightarrow$) A Hamiltonian path $v_{\pi(0)}, ..., v_{\pi(n-1)}$ gives edges $\{v_{\pi(i)}, v_{\pi(i+1)}\}$ forming a spanning subgraph isomorphic to P_n via $\varphi(t_i) = v_{\pi(i)}$. ($\Leftarrow$) A spanning tree of G isomorphic to P_n is a path on all n vertices (since P_n has max degree 2 and is connected). The isomorphism φ gives the Hamiltonian path $\varphi(t_0), ..., \varphi(t_{n-1})$.

Solution extraction. The isomorphism φ directly yields the Hamiltonian path as the sequence $\varphi(t_0), \varphi(t_1), ..., \varphi(t_{n-1})$. □

Example: $n = 5$ vertices, target tree P_n

Source: HamiltonianPath **Target:** IsomorphicSpanningTree

```
$ pred create --example HamiltonianPath/SimpleGraph -o hampath.json
$ pred reduce hampath.json --to IsomorphicSpanningTree/SimpleGraph -o bundle.json
$ pred solve bundle.json
$ pred evaluate hampath.json --config 0,1,2,3,4
```

Step 1 – Source instance. Graph G with $n = 5$ vertices and $|E| = 4$ edges.

Step 2 – Identity reduction. Target host graph is identical. Target tree $T = P_5$ with 4 edges.

Step 3 – Verify a solution. Hamiltonian path visits vertices in order (0, 1, 2, 3, 4). The isomorphism maps P_n to this path in G ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.223: (Exact Cover by 3-Sets → Algebraic Equations over GF(2)) This $O(qn^2)$ reduction [5] assigns one binary variable x_j per subset. For each universe element u_i, a linear equation $\sum_{j \in S_i} x_j + 1 = 0 \pmod 2$ enforces odd coverage, and pairwise products $x_j x_k = 0 \pmod 2$ forbid double coverage. Together these force exactly-once covering.
Overhead: `num_vars = num_sets`.

Proof: *Construction.* Let $(X, \mathcal{C})$ be an X3C instance with $|X| = 3q$ and $\mathcal{C} = \{C_1, ..., C_n\}$. Define n variables over GF(2). For each element u_i, let $S_i = \{j : u_i \in C_j\}$. Add linear constraint $\sum_{j \in S_i} x_j + 1 = 0 \pmod 2$ and pairwise constraints $x_j \cdot x_k = 0 \pmod 2$ for all $j < k \in S_i$. Total: at most $3q + \sum_i \binom{|S_i|}{2}$ equations.

Correctness. ($\Rightarrow$) An exact cover sets $x_j = 1$ for exactly q selected sets. Each element has exactly one covering set, so $\sum_{j \in S_i} x_j = 1 \equiv 1 \pmod 2$, satisfying the linear constraint. All pairwise products vanish since at most one $x_j = 1$ per S_i. ($\Leftarrow$) The linear constraint forces an odd number of selected sets per element. The pairwise constraints forbid selecting two sets covering the same element. Together: exactly one set per element. Since each set has 3 elements and all $3q$ are covered, exactly q sets are selected.

Solution extraction. The X3C configuration equals the GF(2) configuration: select subset j iff $x_j = 1$. □

Example: $|U| = 6$, $|\mathcal{C}| = 3$ subsets

Source: ExactCoverBy3Sets **Target:** AlgebraicEquationsOverGF2

```
$ pred create --example ExactCoverBy3Sets -o x3c.json
$ pred reduce x3c.json --to AlgebraicEquationsOverGF2 -o bundle.json
$ pred solve bundle.json
$ pred evaluate x3c.json --config 1,1,0
```

Step 1 – Source instance. The X3C fixture uses subsets $C_0 = \{0, 1, 2\}$, $C_1 = \{3, 4, 5\}$, and $C_2 = \{0, 3, 4\}$ over a universe of size 6.

Step 2 – Build the GF(2) system. The target has 3 variables and 9 equations. Grouping the JSON equations by element gives: for element 0, $x_0 + x_2 + 1 = 0$ and $x_0 x_2 = 0$; for elements 1 and 2, $x_0 + 1 = 0$; for elements 3 and 4, $x_1 + x_2 + 1 = 0$ and $x_1 x_2 = 0$; for element 5, $x_1 + 1 = 0$.

Step 3 – Evaluate the canonical target witness. The target assignment is $x = (1, 1, 0) = (1, 1, 0)$. Substituting gives $1 + 0 + 1 = 0 \bmod 2$, $1 \cdot 0 = 0$, and $1 + 1 = 0 \bmod 2$, which are exactly the three polynomial patterns appearing in the fixture.

Step 4 – Verify the witness pair. The two 1-entries in x select C_0 and C_1, while $x_2 = 0$ omits C_2. Thus the target witness encodes the same exact cover as the source configuration (1, 1, 0) ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.224: (Partition → Production Planning) This $O(n)$ reduction [116] maps each element a_i to a production period with capacity $c_i = a_i$ and setup cost $b_i = a_i$ (zero production and inventory costs). One demand period requires $Q = S/2$ units with no production capacity. The cost bound is $B = Q$. Activating a subset summing to Q exactly meets demand at cost $Q = B$.
Overhead: `num_periods = num_elements + 1`.

Proof: *Construction.* Given Partition instance $A = \{a_1, ..., a_n\}$ with total S and $Q = S/2$. If S is odd, output a trivially infeasible instance. Otherwise create $n + 1$ periods: for each a_i, period i has $r_i = 0, c_i = a_i, b_i = a_i, p_i = h_i = 0$; demand period $n + 1$ has $r_{n+1} = Q, c_{n+1} = 0, b_{n+1} = p_{n+1} = h_{n+1} = 0$. Cost bound $B = Q$.

Correctness. ($\Rightarrow$) A balanced partition A' with sum Q activates those periods: total production $= Q$ meets demand, inventory levels stay non-negative, and total cost $= \sum_{i \in A'} a_i = Q = B$. ($\Leftarrow$) Any feasible plan has setup cost $\sum_{i \in J} a_i \leq Q$ (where J is the set of active periods) and must produce at least Q units. Since $x_i \leq c_i = a_i$, total production $\leq \sum_{i \in J} a_i$. These force $\sum_{i \in J} a_i = Q$, yielding a balanced partition.

Solution extraction. Active element periods ($x_i > 0$) form one partition half. □

Example: 5 elements, total $= 20$

Source: Partition **Target:** ProductionPlanning

```
$ pred create --example Partition -o partition.json
$ pred reduce partition.json --to ProductionPlanning -o bundle.json
$ pred solve bundle.json
$ pred evaluate partition.json --config 0,0,0,1,1
```

Step 1 – Source instance. The Partition fixture has sizes (3, 5, 2, 4, 6) with total 20, so $Q = 10$. The canonical source vector (0, 0, 0, 1, 1) splits the instance into sums 10 and 10.

Step 2 – Build the period table.

period	c_t	b_t	r_t	x_t
P_0	3	3	0	0
P_1	5	5	0	0
P_2	2	2	0	0
P_3	4	4	0	4
P_4	6	6	0	6
P_5	0	0	10	0

The first five periods encode the partition elements, and the last period carries the demand of 10 units.

Step 3 – Track cumulative production and inventory. The stored plan (0, 0, 0, 4, 6, 0) gives cumulative production $0, 0, 0, 4, 10, 10$ against cumulative demand $0, 0, 0, 0, 0, 10$. Hence the inventory levels are $0, 0, 0, 4, 10, 0$, so every prefix remains feasible.

Step 4 – Check the cost and recover the partition. Only periods P_3 and P_4 are active, so the total cost is just the setup cost $4 + 6 = 10$; production and inventory costs are all zero in the fixture. The active periods therefore recover the source vector (0, 0, 0, 1, 1), selecting the subset of size 10 ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.225: (Hamiltonian Path Between Two Vertices → Longest Path) This $O(n + m)$ reduction [5] passes the graph through with unit edge lengths, same source s and target t, and bound $K = n - 1$. A Hamiltonian s-t path uses exactly $n - 1$ edges, which is the maximum possible for a simple path on n vertices. The reduction is size-preserving.
Overhead: `num_vertices = num_vertices`, `num_edges = num_edges`.

Proof: *Construction.* Given $(G = (V, E), s, t)$ with $n = |V|$, set $G' = G$, $\ell(e) = 1$ for all $e \in E$, $s' = s$, $t' = t$, and $K = n - 1$.

Correctness. ($\Rightarrow$) A Hamiltonian s-t path has $n - 1$ edges of length 1 each, giving total length $n - 1 = K$. ($\Leftarrow$) A simple s'-t' path of length $\geq K = n - 1$ has $\geq n - 1$ edges. Since a simple path on n vertices can have at most $n - 1$ edges, it has exactly $n - 1$ edges and visits all vertices – it is a Hamiltonian s-t path.

Solution extraction. From the edge-selection vector, trace the path from s following selected edges to reconstruct the vertex permutation. □

Example: $n = 5$ **vertices,** $s = 0$, $t = 4$

Source: HamiltonianPathBetweenTwoVertices **Target:** LongestPath

```
$ pred create --example HamiltonianPathBetweenTwoVertices/SimpleGraph -o hampath2v.json
$ pred reduce hampath2v.json --to LongestPath/SimpleGraph/One -o bundle.json
$ pred solve bundle.json
$ pred evaluate hampath2v.json --config 0,1,2,3,4
```

Step 1 – Source instance. Graph G with $n = 5$ vertices, $|E| = 4$ edges, $s = 0$, $t = 4$.

Step 2 – Identity reduction. Target graph is identical with unit edge lengths. $K = n - 1 = 4$, same s and t.

Step 3 – Verify a solution. Source Hamiltonian path visits vertices (0, 1, 2, 3, 4) from $s = 0$ to $t = 4$. Target selects 4 edges, total length $= n - 1 = K$ ✓.

Multiplicity: The fixture stores one canonical witness.

Rule 3.226: (Graph Partitioning → Max-Cut (weighted)) This $O(n^2)$ reduction [5] builds a weighted complete graph on the same vertices, with weight $P - 1$ on original edges and weight P on non-edges, where $P = |E| + 1$. For any partition (A, B), the target cut weight is $P\ |A|\ |B| - |E(A, B)|$, so the factor P first forces balance and then the $-\ |E(A, B)|$ term minimizes the number of source edges crossing the bisection.
Overhead: `num_vertices = num_vertices`, `num_edges = num_vertices * (num_vertices + -1 * 1) * 2^-1`.

Proof: *Construction.* Let $G = (V, E)$ be a Graph Partitioning instance with $|V| = n$ even, and let $P = |E| + 1$. Build the complete graph K_n on the same vertex set. For each pair $u \neq v$, set $w(u, v) = P - 1$ if $\{u, v\} \in E$ and $w(u, v) = P$ otherwise. The target Max-Cut instance therefore has n vertices and $n(n - 1)/2$ weighted edges.

Correctness. For any partition (A, B) of V, let $c(A, B) = |E(A, B)|$ be the number of source edges crossing from A to B. Among the $|A|\ |B|$ crossing pairs, exactly $c(A, B)$ are edges of G and the remaining $|A|\ |B| - c(A, B)$ are non-edges. Hence the target cut weight is

$$c(A, B)(P - 1) + (|A|\ |B| - c(A, B))P = P\ |A|\ |B| - c(A, B).$$

($\Rightarrow$) If (A, B) is a balanced partition of G with $|A| = |B| = n/2$ and cut size $c(A, B)$, then the same partition in the target graph has weight $Pn^2/4 - c(A, B)$.

($\Leftarrow$) Any unbalanced partition of an even-sized vertex set satisfies $|A|\ |B| \leq n^2/4 - 1$, so its target weight is at most $P(n^2/4 - 1)$. On the other hand, any balanced partition has weight at least $Pn^2/4 - |E| > P(n^2/4 - 1)$ because $P = |E| + 1$. Therefore no optimal Max-Cut can be unbalanced: every optimum must satisfy $|A| = |B| = n/2$. Once balance is forced, the term $P\ |A|\ |B| = Pn^2/4$ is constant, so maximizing target weight is exactly the same as minimizing $c(A, B)$. Thus optimal Max-Cut solutions are precisely minimum bisections of G.

Solution extraction. The Max-Cut partition vector is already a valid Graph Partitioning witness, so extraction is the identity map. □

Example: $n = 6$ **vertices,** $|E| = 9$

Source: GraphPartitioning **Target:** MaxCut

```
$ pred create --example GraphPartitioning/SimpleGraph -o graphpart.json
$ pred reduce graphpart.json --to MaxCut/SimpleGraph/i32 -o bundle.json
$ pred solve bundle.json
$ pred evaluate graphpart.json --config 0,0,0,1,1,1
```

Step 1 – Source instance. Graph G has $n = 6$ vertices and $|E| = 9$ edges, so the code uses penalty $P = |E| + 1 = 10$.

Step 2 – Build the weighted complete graph. The target has 6 vertices and 15 edges. Original edges receive weight $P - 1 = 9$, while non-edges receive weight $P = 10$.

Step 3 – Verify the canonical witness. The balanced partition (0, 0, 0, 1, 1, 1) gives sides $A = \{0, 1, 2\}$ and $B = \{3, 4, 5\}$ with 9 crossing pairs. It cuts 3 source edges, so the identical Max-Cut partition has weight $87 = 10 \cdot 9 - 3$. Any unbalanced 2-4 split has at most 8 crossing pairs and therefore weight at most $80 < 87$, so the optimum is forced to be balanced ✓.

Multiplicity: The fixture stores one canonical witness.

Bibliography

[1] R. M. Karp, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations*, Plenum Press, 1972, pp. 85–103.

[2] C. E. Shannon, "The zero error capacity of a noisy channel," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 8–19, 1956, doi: 10.1109/TIT.1956.1056798.

[3] M. Xiao and H. Nagamochi, "Exact Algorithms for Maximum Independent Set," *Information and Computation*, vol. 255, pp. 126–146, 2017, doi: 10.1016/j.ic.2017.06.001.

[4] J. Alber and J. Fiala, "Geometric separation and exact solutions for the parameterized independent set problem on disk graphs," *Journal of Algorithms*, vol. 52, no. 2, pp. 134–151, 2004, doi: 10.1016/j.jalgor.2003.10.001.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[6] F. Barahona, "On the computational complexity of Ising spin glass models," *Journal of Physics A: Mathematical and General*, vol. 15, no. 10, pp. 3241–3253, 1982.

[7] M. X. Goemans and D. P. Williamson, "Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming," *Journal of the ACM*, vol. 42, no. 6, pp. 1115–1145, 1995, doi: 10.1145/227683.227684.

[8] R. Williams, "A new algorithm for optimal 2-constraint satisfaction and its implications," *Theoretical Computer Science*, vol. 348, no. 2–3, pp. 357–365, 2005, doi: 10.1016/j.tcs.2005.09.023.

[9] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some Simplified NP-Complete Graph Problems," *Theoretical Computer Science*, vol. 1, no. 3, pp. 237–267, 1976, doi: 10.1016/0304-3975(76)90059-1.

[10] M. Cygan, D. Lokshtanov, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, "Minimum Bisection Is Fixed Parameter Tractable," *SIAM Journal on Computing*, vol. 48, no. 2, pp. 417–450, 2019, doi: 10.1137/140990255.

[11] S. Arora, S. Rao, and U. Vazirani, "Expander Flows, Geometric Embeddings and Graph Partitioning," *Journal of the ACM*, vol. 56, no. 2, pp. 1–37, 2009, doi: 10.1145/1502793.1502794.

[12] M. Held and R. M. Karp, "A Dynamic Programming Approach to Sequencing Problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196–210, 1962, doi: 10.1137/0110015.

[13] A. Björklund, "Determinant Sums for Undirected Hamiltonicity," *SIAM Journal on Computing*, vol. 43, no. 1, pp. 280–299, 2014, doi: 10.1137/110839229.

[14] A. Björklund, T. Husfeldt, and M. Koivisto, "Set Partitioning via Inclusion-Exclusion," *SIAM Journal on Computing*, vol. 39, no. 2, pp. 546–563, 2009, doi: 10.1137/070683933.

[15] N. Robertson and P. D. Seymour, "Graph Minors. XIII. The Disjoint Paths Problem," *Journal of Combinatorial Theory, Series B*, vol. 63, no. 1, pp. 65–110, 1995, doi: 10.1006/jctb.1995.1006.

[16] K.-i. Kawarabayashi, Y. Kobayashi, and B. Reed, "The disjoint paths problem in quadratic time," *Journal of Combinatorial Theory, Series B*, vol. 102, no. 2, pp. 424–435, 2012, doi: 10.1016/j.jctb.2011.07.004.

[17] S. Even and R. E. Tarjan, "A Combinatorial Problem Which Is Complete in Polynomial Space," *Journal of the ACM*, vol. 23, no. 4, pp. 710–719, 1976.

[18] J. Bruno and L. Weinberg, "A Constructive Graph-Theoretic Solution of the Shannon Switching Game," *IEEE Transactions on Circuit Theory*, vol. 17, no. 1, pp. 74–81, 1970.

[19] N. Alon, R. Yuster, and U. Zwick, "Color-coding," *Journal of the ACM*, vol. 42, no. 4, pp. 844–856, 1995, doi: 10.1145/210332.210337.

[20] A. Itai, "Two-Commodity Flow," *Journal of the ACM*, vol. 25, no. 4, pp. 596–611, 1978, doi: 10.1137/0203021.

[21] S. Even, A. Itai, and A. Shamir, "On the Complexity of Timetable and Multicommodity Flow Problems," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 691–703, 1976, doi: 10.1137/0205048.

[22] C. Büsing and S. Stiller, "Line planning, path constrained network flow and inapproximability," *Networks*, vol. 57, no. 1, pp. 106–113, 2011, doi: 10.1002/net.20386.

[23] C. H. Papadimitriou and M. Yannakakis, "The Complexity of Restricted Spanning Tree Problems," *Journal of the ACM*, vol. 29, no. 2, pp. 285–309, 1982, doi: 10.1145/322307.322309.

[24] H. C. Joksch, "The Shortest Route Problem with Constraints," *Journal of Mathematical Analysis and Applications*, vol. 14, no. 2, pp. 191–197, 1966, doi: 10.1016/0022-247X(66)90002-6.

[25] R. Hassin, "Approximation Schemes for the Restricted Shortest Path Problem," *Mathematics of Operations Research*, vol. 17, no. 1, pp. 36–42, 1992, doi: 10.1287/moor.17.1.36.

[26] D. H. Lorenz and D. Raz, "A Simple Efficient Approximation Scheme for the Restricted Shortest Path Problem," *Operations Research Letters*, vol. 28, no. 5, pp. 213–219, 2001, doi: 10.1016/S0167-6377(01)00079-6.

[27] R. Beigel and D. Eppstein, "3-Coloring in Time $O(1.3289^n)$", *Journal of Algorithms*, vol. 54, no. 2, pp. 168–204, 2005, doi: 10.1016/j.jalgor.2004.06.008.

[28] P. Wu, H. Gu, H. Jiang, Z. Shao, and J. Xu, "A Faster Algorithm for the 4-Coloring Problem," in *European Symposium on Algorithms (ESA)*, in LIPIcs, vol. 308. 2024, pp. 103:1–103:16. doi: 10.4230/LIPIcs.ESA.2024.103.

[29] O. Zamir, "Breaking the 2^n Barrier for 5-Coloring and 6-Coloring", in *International Colloquium on Automata, Languages, and Programming (ICALP)*, in LIPIcs, vol. 198. 2021, pp. 113:1–113:20. doi: 10.4230/LIPIcs.ICALP.2021.113.

[30] J. M. M. van Rooij and H. L. Bodlaender, "Exact algorithms for dominating set," *Discrete Applied Mathematics*, vol. 159, no. 17, pp. 2147–2164, 2011, doi: 10.1016/j.dam.2011.07.001.

[31] J. Edmonds, "Paths, Trees, and Flowers," *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, 1965.

[32] P. C. Gilmore and R. E. Gomory, "Sequencing a One State-Variable Machine: A Solvable Case of the Traveling Salesman Problem," *Operations Research*, vol. 12, no. 5, pp. 655–679, 1964, doi: 10.1287/opre.12.5.655.

[33] S. E. Dreyfus and R. A. Wagner, "The Steiner Problem in Graphs," *Networks*, vol. 1, no. 3, pp. 195–207, 1971, doi: 10.1002/net.3230010302.

[34] J. Byrka, F. Grandoni, T. Rothvoß, and L. Sanità, "Steiner Tree Approximation via Iterative Randomized Rounding," *Journal of the ACM*, vol. 60, no. 1, pp. 1–33, 2013, doi: 10.1145/2432622.2432628.

[35] K. P. Eswaran and R. E. Tarjan, "Augmentation Problems," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 653–665, 1976, doi: 10.1137/0205044.

[36] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, "The Complexity of Multiterminal Cuts," *SIAM Journal on Computing*, vol. 23, no. 4, pp. 864–894, 1994, doi: 10.1137/S0097539292225297.

[37] Y. Cao, J. Chen, and J. Wang, "An Improved Fixed-Parameter Algorithm for the Minimum Weight Multiway Cut Problem," in *Fundamentals of Computation Theory (FCT 2013)*, 2013, pp. 96–107. doi: 10.1007/978-3-642-40164-0_11.

[38] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some Simplified NP-Complete Graph Problems," *Theoretical Computer Science*, vol. 1, no. 3, pp. 237–267, 1976.

[39] F. Gavril, "Some NP-Complete Problems on Graphs," in *Proceedings of the 11th Conference on Information Sciences and Systems*, 1977, pp. 91–95.

[40] D. Adolphson and T. C. Hu, "Optimal Linear Ordering," *SIAM Journal on Applied Mathematics*, vol. 25, no. 3, pp. 403–423, 1973, doi: 10.1137/0125040.

[41] J. M. Robson, "Finding a Maximum Independent Set in Time $O\left(2^{\{\frac{n}{4}\}}\right)$", 2001.

[42] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theoretical Computer Science*, vol. 363, no. 1, pp. 28–42, 2006, doi: 10.1016/j.tcs.2006.06.015.

[43] J. W. Moon and L. Moser, "On cliques in graphs," *Israel Journal of Mathematics*, vol. 3, no. 1, pp. 23–28, 1965, doi: 10.1007/BF02760024.

[44] M. Yannakakis and F. Gavril, "Edge Dominating Sets in Graphs," *SIAM Journal on Applied Mathematics*, vol. 38, no. 3, pp. 364–372, 1980, doi: 10.1137/0138030.

[45] I. Razgon, "Computing Minimum Directed Feedback Vertex Set in $O^*(1.9977^n)$", in *Proceedings of the 10th Italian Conference on Theoretical Computer Science (ICTCS)*, 2007, pp. 70–81.

[46] G. Even, J. Naor, B. Schieber, and M. Sudan, "Approximating Minimum Feedback Sets and Multicuts in Directed Graphs," *Algorithmica*, vol. 20, no. 2, pp. 151–174, 1998, doi: 10.1007/PL00009191.

[47] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto, "Fourier Meets Möbius: Fast Subset Convolution," in *Proceedings of the 39th ACM Symposium on Theory of Computing (STOC)*, 2007, pp. 67–74. doi: 10.1145/1250790.1250801.

[48] J. Nederlof, "Fast Polynomial-Space Algorithms Using Möbius Inversion: Improving on Steiner Tree and Related Problems," in *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP)*, in LNCS, vol. 5555. 2009, pp. 713–725. doi: 10.1007/978-3-642-02927-1_59.

[49] R. A. Cody and J. E. G. Coffman, "Record Allocation for Minimizing Expected Retrieval Costs on Drum-Like Storage Devices," *Journal of the ACM*, vol. 23, no. 1, pp. 103–115, 1976, doi: 10.1145/321921.321933.

[50] L. T. Kou, "Polynomial Complete Consecutive Information Retrieval Problems," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 67–75, 1977, doi: 10.1137/0206005.

[51] K. S. Booth and G. S. Lueker, "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms," *Journal of Computer and System Sciences*, vol. 13, no. 3, pp. 335–379, 1976, doi: 10.1016/S0022-0000(76)80045-1.

[52] J. Edmonds, "Submodular functions, matroids, and certain polyhedra," in *Combinatorial Structures and Their Applications*, Gordon, Breach, 1970, pp. 69–87.

[53] I. Doron-Arad, A. Kulik, and H. Shachnai, "You (Almost) Can't Beat Brute Force for 3-Matroid Intersection," *arXiv preprint arXiv:2412.02217*, 2024.

[54] F. V. Fomin, D. Lokshtanov, F. Panolan, and S. Saurabh, "Exact Algorithms via Monotone Local Search," *Journal of the ACM*, vol. 66, no. 2, pp. 1–23, 2019, doi: 10.1145/3277568.

[55] L. J. Stockmeyer, "The Set Basis Problem Is NP-Complete," Yorktown Heights, New York, technical report RC5431, 1975.

[56] C. L. Lucchesi and S. L. Osborn, "Candidate Keys for Relations," *Journal of Computer and System Sciences*, vol. 17, no. 2, pp. 270–279, 1978, doi: 10.1016/0022-0000(78)90009-0.

[57] W. L. Jr., "Two NP-Complete Problems Related to Information Retrieval," in *Fundamentals of Computation Theory (FCT 1977)*, in Lecture Notes in Computer Science, vol. 56. Springer, 1977, pp. 452–458. doi: 10.1007/3-540-08442-8_115.

[58] F. Glover, G. Kochenberger, and Y. Du, "Quantum Bridge Analytics I: a tutorial on formulating and using QUBO models," *4OR*, vol. 17, pp. 335–371, 2019, doi: 10.1007/s10288-019-00424-y.

[59] H. W. Lenstra, "Integer Programming with a Fixed Number of Variables," in *Mathematics of Operations Research*, 1983, pp. 538–548. doi: 10.1287/moor.8.4.538.

[60] D. Dadush, "Integer Programming, Lattice Algorithms, and Deterministic Volume Estimation," Doctoral dissertation, 2012.

[61] S. Sahni, "Computationally Related Problems," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 262–279, 1974.

[62] P. van Emde Boas, "Another NP-complete Problem and the Complexity of Computing Short Vectors in a Lattice," technical report 81–4, 1981.

[63] R. Kannan, "Minkowski's Convex Body Theorem and Integer Programming," *Mathematics of Operations Research*, vol. 12, no. 3, pp. 415–440, 1987, doi: 10.1287/moor.12.3.415.

[64] D. Micciancio and P. Voulgaris, "A Deterministic Single Exponential Time Algorithm for Most Lattice Problems Based on Voronoi Cell Computations," in *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*, 2010, pp. 351–358. doi: 10.1145/1806689.1806739.

[65] D. Aggarwal, D. Dadush, and N. Stephens-Davidowitz, "Solving the Closest Vector Problem in 2^n Time – The Discrete Gaussian Strikes Again!", in *Proceedings of the 56th IEEE Symposium on Foundations of Computer Science (FOCS)*, 2015, pp. 563–580. doi: 10.1109/FOCS.2015.41.

[66] S. A. Cook, "The Complexity of Theorem-Proving Procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[67] R. Impagliazzo, R. Paturi, and F. Zane, "Which Problems Have Strongly Exponential Complexity?," in *Journal of Computer and System Sciences*, 2001, pp. 512–530. doi: 10.1006/jcss.2001.1774.

[68] T. J. Schaefer, "The Complexity of Satisfiability Problems," *Conference Record of the 10th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 216–226, 1978, doi: 10.1145/800133.804350.

[69] B. Aspvall, M. F. Plass, and R. E. Tarjan, "A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas," *Information Processing Letters*, vol. 8, no. 3, pp. 121–123, 1979, doi: 10.1016/0020-0190(79)90002-4.

[70] T. D. Hansen, H. Kaplan, O. Zamir, and U. Zwick, "Faster k-SAT Algorithms Using Biased-PPSZ", in *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, 2019, pp. 578–589. doi: 10.1145/3313276.3316359.

[71] D. Lichtenstein, "Planar Formulae and Their Uses," *SIAM Journal on Computing*, vol. 11, no. 2, pp. 329–343, 1982, doi: 10.1137/0211025.

[72] M. Järvisalo, P. Kaski, M. Koivisto, and J. H. Korhonen, "Finding Efficient Circuits for Ensemble Computation," *Theory and Applications of Satisfiability Testing – SAT 2012*, vol. 7317. in Lecture Notes in Computer Science, vol. 7317. Springer, pp. 369–382, 2012. doi: 10.1007/978-3-642-31612-8_28.

[73] A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and J. M. Pollard, "The Number Field Sieve," *The Development of the Number Field Sieve*, vol. 1554. in Lecture Notes in Mathematics, vol. 1554. Springer, 1993. doi: 10.1007/BFb0091539.

[74] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994, pp. 124–134. doi: 10.1109/SFCS.1994.365700.

[75] L. J. Stockmeyer and A. R. Meyer, "Word Problems Requiring Exponential Time: Preliminary Report," in *Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC)*, 1973, pp. 1–9. doi: 10.1145/800125.804029.

[76] R. Williams, "Algorithms for Quantified Boolean Formulas," in *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002, pp. 299–307.

[77] K. S. Booth, "PQ-Tree Algorithms," Doctoral dissertation, 1975.

[78] S. Haddadi and Z. Layouni, "Consecutive block minimization is 1.5-approximable," *Information Processing Letters*, vol. 108, no. 3, pp. 161–163, 2008, doi: 10.1016/j.ipl.2008.05.003.

[79] T. Epping, W. Hochstättler, and P. Oertel, "Complexity results on a paint shop problem," *Discrete Applied Mathematics*, vol. 136, no. 2–3, pp. 217–226, 2004, doi: 10.1016/S0166-218X(03)00442-6.

[80] J. M. M. van Rooij, M. van Kooten Niekerk, and H. L. Bodlaender, "Partition Into Triangles on Bounded Degree Graphs," *Theory of Computing Systems*, vol. 52, no. 4, pp. 687–718, 2013, doi: 10.1007/s00224-012-9412-5.

[81] O. H. Ibarra and C. E. Kim, "Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems," *Journal of the ACM*, vol. 22, no. 4, pp. 463–468, 1975, doi: 10.1145/321906.321909.

[82] E. Horowitz and S. Sahni, "Computing Partitions with Applications to the Knapsack Problem," *Journal of the ACM*, vol. 21, no. 2, pp. 277–292, 1974, doi: 10.1145/321812.321823.

[83] D. S. Johnson and K. A. Niemi, "On Knapsacks, Partitions, and a New Dynamic Programming Technique for Trees," *Mathematics of Operations Research*, vol. 8, no. 1, pp. 1–14, 1983, doi: 10.1287/moor.8.1.1.

[84] S. G. Kolliopoulos and G. Steiner, "Partially Ordered Knapsack and Applications to Scheduling," *Discrete Applied Mathematics*, vol. 155, no. 8, pp. 889–897, 2007, doi: 10.1016/j.dam.2006.09.003.

[85] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove?," in *Languages and Compilers for Parallel Computing (LCPC)*, in LNCS, vol. 4382. Springer, 2006, pp. 283–298. doi: 10.1007/978-3-540-72521-3_21.

[86] R. Sethi, "Complete Register Allocation Problems," *SIAM Journal on Computing*, vol. 4, no. 3, pp. 226–248, 1975, doi: 10.1137/0204020.

[87] R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *Journal of the ACM*, vol. 17, no. 4, pp. 715–728, 1970, doi: 10.1145/321607.321620.

[88] C. W. Kessler, "Scheduling Expression DAGs for Minimal Register Need," Doctoral dissertation, 1998.

[89] J. Bruno and R. Sethi, "Code Generation for a One-Register Machine," *Journal of the ACM*, vol. 23, no. 3, pp. 502–510, 1976, doi: 10.1145/321958.321974.

[90] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code Generation for Machines with Multiregister Operations," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ACM, 1977, pp. 21–28. doi: 10.1145/512950.512953.

[91] J. K. Lenstra and A. H. G. R. Kan, "On General Routing Problems," *Networks*, vol. 6, no. 3, pp. 273–280, 1976, doi: 10.1002/net.3230060305.

[92] G. N. Frederickson, "Approximation Algorithms for Some Postman Problems," *Journal of the ACM*, vol. 26, no. 3, pp. 538–554, 1979, doi: 10.1145/322139.322150.

[93] C. H. Papadimitriou, "On the Complexity of Edge Traversing," *Journal of the ACM*, vol. 23, no. 3, pp. 544–554, 1976.

[94] J. Edmonds and E. L. Johnson, "Matching, Euler Tours and the Chinese Postman," *Mathematical Programming*, vol. 5, pp. 88–124, 1973.

[95] G. N. Frederickson, M. S. Hecht, and C. E. Kim, "Approximation Algorithms for Some Routing Problems," *SIAM Journal on Computing*, vol. 7, no. 2, pp. 178–193, 1978, doi: 10.1137/0207017.

[96] G. N. Frederickson and D.-W. Guan, "Nonpreemptive Ensemble Motion Planning on a Tree," *Journal of Algorithms*, vol. 15, no. 1, pp. 29–60, 1993.

[97] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2004, pp. 1367–1372. doi: 10.1109/TPAMI.2004.75.

[98] A. Jüttner and P. Madarasi, "VF2++ — An improved subgraph isomorphism algorithm," *Discrete Applied Mathematics*, vol. 242, pp. 69–81, 2018, doi: 10.1016/j.dam.2018.02.018.

[99] D. Maier, "The Complexity of Some Problems on Subsequences and Supersequences," *Journal of the ACM*, vol. 25, no. 2, pp. 322–336, 1978, doi: 10.1145/322063.322075.

[100] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974, doi: 10.1145/321796.321811.

[101] C. Haase and S. Kiefer, "The Complexity of the Kth Largest Subset Problem and Related Problems," *Information Processing Letters*, vol. 116, no. 2, pp. 111–115, 2016.

[102] D. A. Plaisted, "Some Polynomial and Integer Divisibility Problems Are NP-Hard," technical report STAN-CS-76-583, 1976.

[103] K.-J. Räihä and E. Ukkonen, "The Shortest Common Supersequence Problem over Binary Alphabet is NP-Complete," *Theoretical Computer Science*, vol. 16, no. 2, pp. 187–198, 1981, doi: 10.1016/0304-3975(81)90075-X.

[104] R. A. Wagner, "On the Complexity of the Extended String-to-String Correction Problem," in *Proceedings of the 7th Annual ACM Symposium on Theory of Computing (STOC)*, 1975, pp. 218–223. doi: 10.1145/800116.803771.

[105] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974, doi: 10.1145/321796.321811.

[106] J. A. Storer, "NP-Completeness Results Concerning Data Compression," technical report 234, 1977.

[107] J. A. Storer and T. G. Szymanski, "Data Compression via Textual Substitution," *Journal of the ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[108] M. Charikar *et al.*, "The Smallest Grammar Problem," in *IEEE Transactions on Information Theory*, 2005, pp. 2554–2576.

[109] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos, "A Note on Exact Algorithms for Vertex Ordering Problems on Graphs," *Theory of Computing Systems*, vol. 50, no. 3, pp. 420–432, 2012, doi: 10.1007/s00224-011-9312-0.

[110] J. Chen, Y. Liu, S. Lu, B. O'Sullivan, and I. Razgon, "A Fixed-Parameter Algorithm for the Directed Feedback Vertex Set Problem," *Journal of the ACM*, vol. 55, no. 5, pp. 1–19, 2008, doi: 10.1145/1411509.1411511.

[111] C. L. Lucchesi and D. H. Younger, "A Minimax Theorem for Directed Graphs," *Journal of the London Mathematical Society*, no. 3, pp. 369–374, 1978, doi: 10.1112/jlms/s2-17.3.369.

[112] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Research Logistics Quarterly*, vol. 1, no. 1, pp. 61–68, 1954, doi: 10.1002/nav.3800010110.

[113] L. Shang, C. Wan, and J. Wang, "An exact algorithm for the three-machine flow shop problem," *Computers & Operations Research*, vol. 91, pp. 79–89, 2018, doi: 10.1016/j.cor.2017.10.015.

[114] T. Gonzalez and S. Sahni, "Open Shop Scheduling to Minimize Finish Time," *Journal of the ACM*, vol. 23, no. 4, pp. 665–679, 1976, doi: 10.1145/321978.321985.

[115] J. J. Bartholdi, J. B. Orlin, and H. D. Ratliff, "Cyclic Scheduling via Integer Programs with Circular Ones," *Operations Research*, vol. 28, no. 5, pp. 1074–1085, 1980, doi: 10.1287/opre.28.5.1074.

[116] M. Florian, J. K. Lenstra, and A. H. G. R. Kan, "Deterministic Production Planning: Algorithms and Complexity," *Management Science*, vol. 26, no. 7, pp. 669–679, 1980.

[117] L. V. Sickle and K. M. Chandy, "Computational Complexity of Network Design Algorithms," in *IFIP Congress 77*, 1977, pp. 235–239.

[118] J. D. Ullman, "NP-Complete Scheduling Problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384–393, 1975, doi: 10.1016/S0022-0000(75)80008-0.

[119] J. K. Lenstra and A. H. G. R. Kan, "Complexity of Scheduling under Precedence Constraints," *Operations Research*, vol. 26, no. 1, pp. 22–35, 1978, doi: 10.1287/opre.26.1.22.

[120] E. G. Coffman and R. L. Graham, "Optimal Scheduling for Two-Processor Systems," *Acta Informatica*, vol. 1, no. 3, pp. 200–213, 1972, doi: 10.1007/BF00288685.

[121] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961, doi: 10.1287/opre.9.6.841.

[122] C. H. Papadimitriou and M. Yannakakis, "Scheduling Interval-Ordered Tasks," in *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC)*, 1979, pp. 1–7. doi: 10.1145/800135.804393.

[123] P. Brucker, M. R. Garey, and D. S. Johnson, "Scheduling equal-length tasks under tree-like precedence constraints to minimize maximum lateness," *Mathematics of Operations Research*, vol. 2, no. 3, pp. 275–284, 1977.

[124] J. K. Lenstra, A. H. G. R. Kan, and P. Brucker, "Complexity of Machine Scheduling Problems," *Annals of Discrete Mathematics*, vol. 1, pp. 343–362, 1977, doi: 10.1016/S0167-5060(08)70743-X.

[125] W. E. Smith, "Various Optimizers for Single-Stage Production," *Naval Research Logistics Quarterly*, vol. 3, no. 1–2, pp. 59–66, 1956, doi: 10.1002/nav.3800030106.

[126] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*. Addison-Wesley, 1967.

[127] W. A. Horn, "Minimizing Average Flow Time with Parallel Machines," *Operations Research*, vol. 21, no. 3, pp. 846–847, 1973.

[128] J. M. Moore, "An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs", *Management Science*, vol. 15, no. 1, pp. 102–109, 1968, doi: 10.1287/mnsc.15.1.102.

[129] E. L. Lawler, "Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints," *Annals of Discrete Mathematics*, vol. 2, pp. 75–90, 1978, doi: 10.1016/S0167-5060(08)70356-7.

[130] E. L. Lawler, "A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness," *Annals of Discrete Mathematics*, vol. 1, pp. 331–342, 1977, doi: 10.1016/S0167-5060(08)70742-8.

[131] C. N. Potts and L. N. V. Wassenhove, "A Branch and Bound Algorithm for the Total Weighted Tardiness Problem," *Operations Research*, vol. 33, no. 2, pp. 363–377, 1985, doi: 10.1287/opre.33.2.363.

[132] S. Tanaka, S. Fujikuma, and M. Araki, "An exact algorithm for single-machine scheduling without machine idle time," *Journal of Scheduling*, vol. 12, no. 6, pp. 575–593, 2009, doi: 10.1007/s10951-008-0093-5.

[133] H. M. Abdel-Wahab and T. Kameda, "Scheduling to Minimize Maximum Cumulative Cost Subject to Series-Parallel Precedence Constraints," *Operations Research*, vol. 26, no. 1, pp. 141–158, 1978, doi: 10.1287/opre.26.1.141.

[134] C. L. Monma and J. B. Sidney, "Sequencing with Series-Parallel Precedence Constraints," *Mathematics of Operations Research*, vol. 4, no. 3, pp. 215–224, 1979, doi: 10.1287/moor.4.3.215.

[135] S. Even, A. Itai, and A. Shamir, "On the Complexity of Timetable and Multicommodity Flow Problems," *SIAM Journal on Computing*, vol. 5, no. 4, pp. 691–703, 1976, doi: 10.1137/0205048.

[136] W. S. Jewell, "Optimal Flow Through Networks with Gains," *Operations Research*, vol. 10, no. 4, pp. 476–499, 1962, doi: 10.1287/opre.10.4.476.

[137] C. Beeri and P. A. Bernstein, "Computational Problems Related to the Design of Normal Form Relational Schemas," *ACM Transactions on Database Systems*, vol. 4, no. 1, pp. 30–59, 1979, doi: 10.1145/320064.320066.

[138] A. K. Chandra and P. M. Merlin, "Optimal Implementation of Conjunctive Queries in Relational Data Bases," in *Proceedings of the 9th Annual ACM Symposium on Theory of Computing (STOC)*, 1977, pp. 77–90. doi: 10.1145/800105.803397.

[139] P. G. Kolaitis and M. Y. Vardi, "Conjunctive-Query Containment and Constraint Satisfaction," in *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1998, pp. 205–213. doi: 10.1145/275487.275511.

[140] G. Gottlob, N. Leone, and F. Scarcello, "Hypertree Decompositions and Tractable Queries," *Journal of Computer and System Sciences*, vol. 64, no. 3, pp. 579–627, 2002, doi: 10.1006/jcss.2001.1809.

[141] K. S. Booth and G. S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms," *Journal of Computer and System Sciences*, vol. 13, no. 3, pp. 335–379, 1976.

[142] K. G. Murty, "A fundamental problem in linear inequalities with applications to the travelling salesman problem," *Mathematical Programming*, vol. 3, pp. 326–370, 1972, doi: 10.1007/BF01584550.

[143] E. Canale, C. Qureshi, and A. Viola, "Qubo model for the Closest Vector Problem," *arXiv preprint*, 2023.

[144] A. Lucas, "Ising formulations of many NP problems," *Frontiers in Physics*, vol. 2, no. 5, 2014.

[145] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Cengage Learning, 2012.

[146] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. MIT Press, 2022.

[147] J. C. Lagarias and A. M. Odlyzko, "Solving Low-Density Subset Sum Problems," *Journal of the ACM*, vol. 32, no. 1, pp. 229–246, 1985, doi: 10.1145/2455.2461.

[148] M. J. Coster, A. Joux, B. A. LaMacchia, A. M. Odlyzko, C.-P. Schnorr, and J. Stern, "Improved Low-Density Subset Sum Algorithms," *Computational Complexity*, vol. 2, no. 2, pp. 111–128, 1992, doi: 10.1007/BF01201999.

[149] S. Heidari, M. J. Dinneen, and P. Delmas, "An Equivalent QUBO Model to the Minimum Multi-Way Cut Problem," technical report CDMTCS-565, 2022.

[150] J. D. Whitfield, M. Faccin, and J. D. Biamonte, "Ground-state spin logic," *EPL (Europhysics Letters)*, vol. 99, no. 5, p. 57004, 2012.

[151] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[152] S. Chopra and J. H. Owen, "Extended formulations for the A-cut problem," *Mathematical Programming*, vol. 73, pp. 7–30, 1996, doi: 10.1007/BF02592096.

[153] R. T. Wong, "A Dual Ascent Approach for Steiner Tree Problems on a Directed Graph," *Mathematical Programming*, vol. 28, no. 3, pp. 271–287, 1984, doi: 10.1007/BF02612335.

[154] T. Koch and A. Martin, "Solving Steiner Tree Problems in Graphs to Optimality," *Networks*, vol. 32, no. 3, pp. 207–232, 1998, doi: 10.1002/(SICI)1097-0037(199810)32:3<207::AID-NET5>3.0.CO;2-O.

[155] M. Streif, S. Yarkoni, A. Skolik, F. Neukart, and M. Leib, "Beating classical heuristics for the binary paint shop problem with the quantum approximate optimization algorithm," *Physical Review A*, vol. 104, p. 12403, 2021, doi: 10.1103/PhysRevA.104.012403.

[156] M.-T. Nguyen, J.-G. Liu, J. Wurtz, M. D. Lukin, S.-T. Wang, and H. Pichler, "Quantum Optimization with Arbitrary Connectivity Using Rydberg Atom Arrays," *PRX Quantum*, vol. 4, p. 10316, 2023, doi: 10.1103/PRXQuantum.4.010316.

[157] Z. Galil and N. Megiddo, "Cyclic Ordering is NP-Complete," *Theoretical Computer Science*, vol. 5, no. 2, pp. 179–182, 1977, doi: 10.1016/0304-3975(77)90005-1.

[158] P. Brucker, "On the Complexity of Clustering Problems," *Optimization and Operations Research*, vol. 157. in Lecture Notes in Economics and Mathematical Systems, vol. 157. Springer, pp. 45–54, 1978.

[159] J. B. Orlin, "Contentment in Graph Theory: Covering Graphs with Cliques," *Indagationes Mathematicae (Proceedings)*, vol. 80, no. 5, pp. 406–424, 1977, doi: 10.1016/1385-7258(77)90055-5.

[160] L. T. Kou, L. J. Stockmeyer, and C. K. Wong, "Covering Edges by Cliques with Regard to Keyword Conflicts and Intersection Graphs," *Communications of the ACM*, vol. 21, no. 2, pp. 135–139, 1978, doi: 10.1145/359340.359346.