CS 170 Lecture Notes

Alec Li

Fall 2021 — Professor Jelani Nelson

Contents

8/26/2021

Lecture 1

Introduction

Definition 1.1: Algorithm

An algorithm is a well defined procedure for carrying out some computational task.

The goals of an algorithm are:

1. *Correctness*: The algorithm halts with the right answer
2. *Efficiency*: Minimize consumption of computational resources

Today, we will consider arithmetic problems. For our purposes, we will be specifying the inputs with the Arabic numeral system.

1.1 Addition

We want to compute $x + y$, each is $\leq n$ digits (if they have different numbers of digits, we can pad with zeroes)

Algorithm 1: "Count on your fingers"; that is, we start at x, and increment y times.

How do we increment? We start at the right-most digit, and advance to the next digit, and propagate the carry if needed.

Runtime: $y \cdot (n+1) \approx y \cdot n$; incrementing takes in the worst case $n+1$ timesteps (one for each digit), and we increment y times. If we specify y with its number of digits, this runtime is bounded by $n \cdot 10^n$.

We could be a little bit more careful here with our carrying to get a smaller runtime estimate to approximately 10^n. This is still pretty bad though.

Algorithm 2: "Grade school algorithm"; write the first number on top of the second number, and go from right to left in digits and keep track of the carries.

Runtime: n, one timestep for each digit (with a potential additional digit), disregarding any constant factors. The runtime is actually $O(n)$ and $\Theta(n)$ as well, which we will more precisely define in the future. This is also assuming that we have hard-coded a lookup table for how to add one-digit numbers.

We can't do better than linear time here; one way to see this is that the output contains n digits, and it takes linear time to even write down this answer. Another way to see this is that if we had an algorithm that takes less than linear time, there was at least one digit in the input that we didn't even look at—we could just change this digit and get a wrong output.

1.2 Multiplication

We want to compute $x \cdot y$, each n digits long.

Algorithm 1: Start with 0, and add in x to a running sum, y times.

Runtime: $n \cdot y \leq O(n \cdot 10^n)$; we have to increment an n-digit number, y times.

Algorithm 2: "Grade school algorithm"

Runtime: $n^2 + n^2 \approx \theta(n^2)$; we take n^2 to get the numbers to add together, and n^2 to add the n numbers together.

The natural question that arises is can we do better than n^2? Kolmogorov conjectured that there does not exist any faster algorithm, but in 1960, Karatsuba gave an algorithm with $\Theta(n^{\log_2 3}) \approx \theta(n^{1.585})$, called the Karatsuba algorithm.

The approach of the Karatsuba algorithm is with "divide and conquer", which we will come back to later.

Algorithm 3: As an example, suppose $x = 5139$ and $y = 7082$. We first chop the numbers into two, giving us $x_h = 51$, $x_l = 39$, $y_h = 70$, $y_l = 82$; we have $x = x_h \cdot 10^{\frac{n}{2}} + x_l$ and $y = y_h \cdot 10^{\frac{n}{2}} + y_l$.

This means that the product

$$x \cdot y = x_h y_h 10^n + \left(x_h y_l + x_l y_h\right) 10^{\frac{n}{2}} + x_l y_l.$$

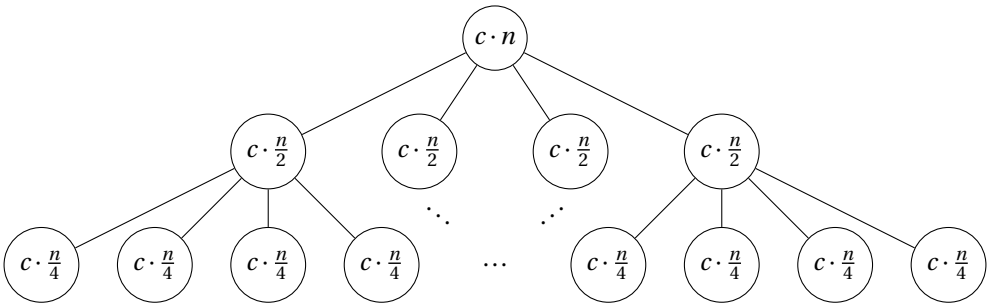
The recursion occurs when we're trying to multiply these smaller parts; we have 4 recursive calls to multiply two $\frac{n}{2}$ digit numbers. The other multiplication operations by 10^k are easier by padding zeroes.

To find the runtime, we can write a recurrence and solve it. The base case of the recursion is a lookup table (of perhaps multiplying 1-digit numbers), which is constant.

Let $T(n)$ be the worst-case runtime of this algorithm. We have

$$T(n) = \begin{cases} \overbrace{4 \cdot T\left(\frac{n}{2}\right)}^{\text{recursion}} + \overbrace{c \cdot n}^{\text{addition}} & n > 1 \\ \underbrace{c}_{\text{base case}} & n = 1 \end{cases}$$

We can draw out this recursion as a tree:



In the first level, we have one item, totaling cn. In the second level, we have 4 items, totaling $c \cdot 2n$. In the third level, we have 16 items, totaling $c \cdot 2^2 n$. Following this pattern, we have

$$c \cdot n \cdot \left(1 + 2 + 2^n + \cdots + 2^k\right) = c \cdot n \cdot \frac{2^{k+1} - 1}{2 - 1} \leq 2c \cdot n \cdot 2^k.$$

Here, k is the number of levels we recurse to. We also know that $k = \log_2 n$, because we're done when the number of items in the kth level is n. This means that our final runtime is $2c \cdot n \cdot 2^{\log_2 n} = 2c \cdot n \cdot n = \Theta(n^2)$.

What this means is that this algorithm is just as good (in practice, worse, because of a bigger constant factor with recursive overhead, etc.).

Algorithm 4: So what did Karatsuba do? He used a concept from Gauss to multiply complex numbers with fewer multiplications. Typically, to multiply $(a + b \cdot i)(c + d \cdot i)$, you'd need 4 multiplications, but Gauss came up with a way to only use 3 multiplications.

Looking back at our original equation:

$$x \cdot y = x_h y_h 10^n + \left(x_h y_l + x_l y_h\right) 10^{\frac{n}{2}} + x_l y_l,$$

we can compute $A = x_h \cdot y_h$, $B = x_l \cdot y_l$, and $D = (x_l + x_h) \cdot (y_l + y_h)$. Notice that $D = x_l y_l + x_h y_l + x_l y_h + x_h y_h$, so $D - A - B$ is our middle term, and we've saved one multiplication step:

$$x \cdot y = A \cdot 10^n + (D - A - B) \cdot 10^{\frac{n}{2}} + B.$$

This means that our recurrence is

$$T(n) = \begin{cases} 3 \cdot T\left(\frac{n}{2}\right) + c \cdot n & n > 1 \\ c & n = 1 \end{cases}$$

Our total sum would then be

$$c \cdot n \cdot \left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^k\right) = c \cdot n \cdot \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\frac{3}{2} - 1} \leq 2c \cdot \frac{3}{2} n \cdot \left(\frac{3}{2}\right)^k = 3cn\left(\frac{3}{2}\right)^k.$$

We got rid of the -1 in the numerator because it is insignificant, and we can pull out a $\frac{3}{2}$ as well.

Here, k is still $\log_2 n$ because we're still halving the number of digits at each level. This means

$$3cn\left(\frac{3}{2}\right)^{\log_2 n} = 3cn \cdot \frac{3^{\log_2 n}}{n} = 3c \cdot 3^{\log_2 n} = 3c\left(2^{\log_2 3}\right)^{\log_2 n} = 3c\left(2^{\log_2 n}\right)^{\log_2 3} = 3cn^{\log_2 3} \approx \Theta\left(n^{\log_2 3}\right).$$

Even though this algorithm is better asymptotically, the constant factor means that it is better to do the $\Theta(n^2)$ algorithm for smaller n. It turns out that Python uses the grade school algorithm for smaller n, and uses Karatsuba for larger values of n (as Python integers are arbitrary precision).

A follow-up to this is still—can we go even faster? Toom, Cook in the late 1960s did a very similar approach, splitting the number into 3 pieces (normally taking 9 multiplications) and needing only 5 multiplications, giving a runtime of $n^{\log_3 5}$. We can further generalize this to $c_k \cdot n^{\log_k (2k-1)}$ if we split up into k parts. The issue though is that c_k blows up as k grows.

In 1971, Schönhage-Strassen gave an algorithm with $n \log n \log\log n$ runtime. This was the record for a while, until Furer in 2007 gave an algorithm with $n \log n \cdot 8^{\log^* n}$, where $\log^*$ denotes the number of times we need to take a logarithm in order to get down to 1. For all intents and purposes, $\log^*$ is never bigger than 5.

Further, Harvey-Van der Hoeven in 2019 came up with an algorithm with $n \log n$ runtime, and it is conjectured that this is the fastest that we can go.

8/31/2021

Lecture 2

More arithmetic, Asymptotic Notation

2.1 Fibonacci Numbers

We have $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 1$.

Let us look at a few algorithms, and count the number of flops (floating point operations).

Algorithm 1: Recursive code:

```
1 def FibRec(n):
2     if n <= 1: return n
3     else: return FibRec(n - 1) + FibRec(n - 2)
```

Let us compute the number of operations that this algorithm needs. Suppose $T(n)$ is the number of flops needed to compute F_n with this algorithm. We have

$$T(n) = \begin{cases} 0 & n \leq 1 \\ T(n-1) + T(n-2) + 1 & n > 1 \end{cases}$$

We can approximate $T(n)$ by the Fibonacci numbers, and we have that

$$F_n = F_{n-1} + F_{n-2} \geq n \cdot F_{n-2} \geq 2 \cdot 2 \cdot F_{n-4} \geq \cdots.$$

This manes that we can generalize and say that $F_n \geq 2^{\frac{n}{2}}$; i.e. $T(n)$ is exponential.

We could also do a similar algorithm by saying $F_n \leq 2 \cdot F_{n-1}$ and generalize to $F_n \leq 2^n$. This means that $2^{\frac{n}{2}} \leq T(n) \leq 2^n$.

Can we come up with an algorithm that uses fewer flops?

Algorithm 2: Iteration

```python
def FibIter(n):
    if n <= 1: return n
    a, b = 0, 1
    for i in range(n-1):
        a, b = b, a + b
    return b
```

Similarly, the only floating point operation we do is with the + in the loop, and we do this $n-1$ times, so the number of flops that this algorithm takes $\Theta(n)$ flops.

Algorithm 3: Fast matrix powering

Suppose we define a matrix $\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. We have that $\mathbf{A}\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n + F_{n-1} \\ F_n \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$.

In other words, $\mathbf{A}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$.

To compute this matrix power, we can use repeated squaring.

Example 2.1: Repeated Squaring

Suppose we're trying to find 9^{71}. We can compute 9^1, 9^2, 9^4, etc. and use the binary expansion of 71 to write 9^{71} as the product of these powers.

Here, $71 = 64+4+2+1$, so $9^{71} = 9^{64} \cdot 9^4 \cdot 9^2 \cdot 9^1$.

In total, we need $\leq 2 \cdot \lceil \log_2 n \rceil$ flops to compute this repeated squaring.

We can expand this with matrices and do the same calculation; this means that we need $O(\log n)$ flops for matrix powering.

Algorithm 4: Closed form solution

The idea is that we can write $\mathbf{A}$ from the previous algorithm in the form as $\mathbf{Q}\Lambda\mathbf{Q}^T$, where each of these matrices are 2×2, and $\mathbf{Q}$ is an orthogonal matrix (i.e. $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$). We also have that $\Lambda = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$, the diagonal matrix of eigenvalues.

This is useful because if we look at $\mathbf{A}^n$, we have

$$\mathbf{A}^n = \mathbf{Q}\Lambda\mathbf{Q}^T\mathbf{Q}\Lambda\mathbf{Q}^T \cdots = \mathbf{Q}\Lambda^n\mathbf{Q}^T.$$

Computing Λ^n means that we just take each element in the diagonal and raise it to the power of n.

This means that we can compute a closed-form formula for this in terms of n. That is, if we have $\varphi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$, we have

$$\mathbf{Q} = \begin{bmatrix} \sqrt{\varphi} & -\sqrt{-\psi} \\ \sqrt{-\psi} & \sqrt{\varphi} \end{bmatrix}$$
$$\lambda_1 = \varphi$$
$$\lambda_2 = \psi$$

Our solution is then

$$F_n = \frac{1}{\sqrt{5}}(\varphi^n - \psi^n).$$

In summary, we have

Algorithm	Flops	Runtime
Recursion	$\exp(c \cdot n)$	$\exp(c \cdot n) \cdot n$
Iteration	n	n^2
Fast Matrix Power	$\log n$	n^2 or $M(n)$
Closed Form	$\log n$	—

Let us look at the runtime calculations for each of these.

The operations for the recursive algorithm are all additions. We can say that the nth Fibonacci number takes around n digits to write down. This means that each operation is linear with the number of digits, so in total we need another multiple of n.

The operations for the iterative case are also additions, and doing the same rough approximation for the number of digits gets us another multiple of n, meaning we have a runtime of n^2. This seems a little bit too pessimistic though; earlier loops should take less than n steps to do an addition (as not all numbers are n digits long).

However, the latter half of iterations (namely, when $i \geq \frac{n}{2}$) each take at least some $c' \cdot n$ time (because we know that each Fibonacci number is at least exponential in $\frac{n}{2}$ from Algorithm 1, and as such needs proportional to $\frac{n}{2}$ digits.), so the total time is at least $c' \cdot n \cdot \frac{n}{2} \geq c'' \cdot n^2$ (that is, we also need to multiply by the number of iterations in the second half).

The operations for the fast matrix power algorithm are all multiplications. We've established earlier that the nth Fibonacci number has proportional to n digits. This means that naive grade-school multiplication needs n^2 for multiplication. We have $\log n$ multiplication operations, so the runtime is $n^2 \log n$.

If we're a little bit more careful, we can get rid of the $\log n$; we can notice that squaring a^k takes around k^2 timesteps and we'd have $1^2 + 2^2 + 4^2 + \cdots \approx O(n^2)$. We can do even better by using something other than the grade school algorithm, giving us a runtime of $M(n)$, the runtime to multiply two n-digit integers.

The interesting thing here is that fast matrix power algorithm, even though it seems faster, isn't actually better than naive iteration unless we use a very sophisticated multiplication algorithm. It seemed like we were getting exponential improvements each time, but if we take the size of the numbers into account, we weren't actually improving by that much (after the improvement from recursive → iterative).

2.2 Asymptotic Notation

In this class, we're going to use notation that allows us to compare functions with each other. Suppose we have two functions $f, g : \mathbb{Z}^+ \to \mathbb{Z}^+$. That is, f and g are functions that map from the positive integers to the positive integers.

If we think about all the functions that map from the size of the input to the runtime needed, we have some measures of growth of these functions.

We have a notion of "Big-Oh". That is, $f = O(g)$ if $(\exists c > 0)(\forall n)(f(n) \leq c \cdot g(n))$. We can think of Big-Oh as a "less than or equal to" operator; f grows as most as fast as g.

We also have a notion of "Little-oh". That is, $f = o(g)$ means that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. We can think of Little-oh as a "strictly less than" operator.

We also have a notion of "Big-Omega". That is, $f = \Omega(g)$ if $g = O(f)$. We can think of Big-Omega as a "greater than or equal to" operator.

We also have a notion of "little-omega". That is, $f = \omega(g)$ if $g = o(f)$. We can think of little-omega as a "strictly greater than" operator.

Lastly, we have a notion of "Theta". That is, $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$. We can think of Theta as an "equals" operator.

In summary,

$f = O(g)$	$f \le g$
$f = \Omega(g)$	$f \ge g$
$f = o(g)$	$f < g$
$f = \omega(g)$	$f > g$
$f = \Theta(g)$	$f = g$

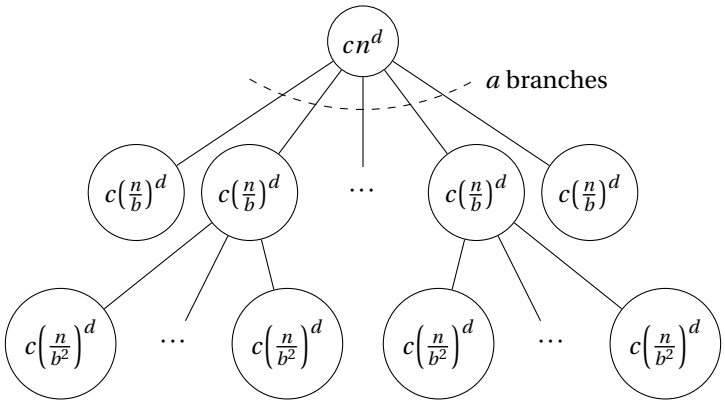
2.3 Master Theorem

The master theorem is a tool that allows you to solve recurrences of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d.$$

For example, Karatsuba had $a = 3$, $b = 2$, and $d = 1$.

We've looked at similar recursive problems in the past; drawing a tree, we have



Theorem 2.2: Master Theorem

If we have a recurrence of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d,$$

we have

1. $a = b^d$: Total work is $\Theta(n^d \cdot \log n)$.
2. $a < b^d$: Total work is $\Theta(n^d)$.

   Here, we have $cn^d \cdot \frac{1-p^{k-1}}{1-p}$ where $p = \frac{a}{b^d}$. The last part is constant, so we have $\Theta(n^d)$.

   In other words, the root of the tree dominates.
3. $a > b^d$: Total work is $\Theta(n^{\log_b a})$.

   Similarly, here we have $cn^d \cdot \frac{\left(\frac{a}{b^d}\right)^{\log_b n+1} - 1}{\frac{a}{b^d} - 1}$, and removing all constants, we have $\Theta(n^{\log_b a})$.

9/2/2021

Lecture 3

Divide and Conquer

3.1 Matrix Multiplication

The general idea is that we want to multiply **AB** to get **C**, where all three matrices have dimension $n \times n$ (for simplicity).

Algorithm 1: The natural implementation is 3 nested for loops, each with n iterations, meaning we have $\Theta(n^3)$ flops.

Algorithm 2: We can split up the matrices as follows:

$$\begin{array}{|c|c|}\hline \mathbf{A} & \mathbf{B} \\ \hline \mathbf{C} & \mathbf{D} \\ \hline \end{array} \times \begin{array}{|c|c|}\hline \mathbf{E} & \mathbf{F} \\ \hline \mathbf{G} & \mathbf{H} \\ \hline \end{array} = \begin{array}{|c|c|}\hline \mathbf{A}\times\mathbf{E}+\mathbf{B}\times\mathbf{G} & \mathbf{A}\times\mathbf{F}+\mathbf{B}\times\mathbf{H} \\ \hline \mathbf{C}\times\mathbf{E}+\mathbf{D}\times\mathbf{G} & \mathbf{C}\times\mathbf{F}+\mathbf{D}\times\mathbf{H} \\ \hline \end{array}.$$

The recurrence relation is then

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2).$$

This solves to $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$.

Algorithm 3: Strassen in 1969 gave an algorithm that only involves 7 multiplications, giving a faster algorithm (specific terms omitted for brevity). This needs only $\Theta(n^{\log_2 7})$ flops.

3.2 Sorting: Merge Sort

With merge sort, we want to sort an array A of length n. We divide the array into two, and recursively call merge sort on each half, then merge the two sorted halves together.

```
def mergesort(A[1...n]):
    B = mergesort(A[1...n/2])
    C = mergesort(A[n/2 + 1...n])
    return merge(B, C)
```

The recurrence is then

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n.$$

This solves to $T(n) = \Theta(n \log n)$.

We can also convert this algorithm into an iterative algorithm, by doing the work bottom-up.

```
def iterative_mergesort(A[1...n]):
    Q = queue_of_singletons(A)
    while len(Q) > 1:
        B, C = Q.pop_two()
        Q.push(merge(B, C))
```

This has the same runtime of $\Theta(n \log n)$; to reason this, we can think of the algorithm as consisting of phases, representing when the lists in the queue have the same size (that is, phase 1 has size 1, phase 2 has size 2, phase 3 has size 4, etc.).

This means that we have n iteration per phase (as each item is looked at once), and $\log n$ phases, giving us the $\Theta(n \log n)$ runtime.

Theorem 3.1: Comparison-based Lower Bound

$\Omega(n \log n)$ comparisons are needed even if we are promised that A is some permutation of $\{1, \ldots, n\}$.

Proof. Our algorithm can be thought of as a binary decision tree. Each node is some comparison, and we branch for whether the comparison is a yes or no.

Each leaf then corresponds to circumstances where the algorithm returns. WLOG, suppose that all leaves are at the same depth T. Another observation to make is that any two distinct permutations have two different sorts (that is, we need to do two different permutations to get back to the sorted array).

The number of leaves then must be at least the number of permutations on n elements. We have $n!$ permutations, with 2^T leaves (as we know the leaves are all at depth T). Solving for T, we have that $T \geq \log_2 n!$, which we claim is at most $n \log n$. □

3.3 Selection/Median

Our goal here is to select the kth smallest element in an array (ex. the median would be selecting $\frac{n}{2}$, the maximum would be selecting n, and the minimum would be selecting 0). We can also assume WLOG that all elements of A are distinct.

If we knew the median, we can first select a pivot p as the median, and create two arrays; L for the elements less than p, and R for the elements greater than p. Now, we recurse. If $k = |L| + 1$, then we return p. If $k \leq |L|$, then we return `QuickSelect(`l`, `k`)`. Otherwise, we return `QuickSelect(`R`, `$k - |L| - 1$`)`.

The recurrence here is

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n),$$

which solves to $T(n) = \Theta(n)$.

But we don't know the median. As such, we can then select a uniformly random element in the array, and we'll be expected to have a linear runtime. However, we can still get unlucky and get a slow runtime of $\Theta(n^2)$, but it's unlikely.

3.3.1 Deterministic Algorithm

We have an array A of n elements, and we break the array into sub-arrays of size 5. In each sub-array, we compute the median (which takes constant time because it has only 5 elements). We can then recursively find the median of this array of medians, and at the end we get a single element as a result, which we'll use as the pivot.

We can show that this pivot is at least between the $\frac{3n}{10}$th and $\frac{7n}{10}$th smallest elements.

The recurrence is then

$$T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + \Theta(n).$$

We can show that this recurrence solves to something at most $B \cdot n$, using strong induction (proof omitted for brevity).

9/7/2021

Lecture 4

Polynomial Multiplication, FFT, Cross-correlation

4.1 Polynomial Multiplication

Suppose we're given two polynomials of degree at most d:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{d-1} x^{d-1}$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{d-1} x^{d-1}$$

The output of the product is the coefficients of

$$A(x) \cdot B(x) = C(x) = c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}.$$

Suppose we define $N = 2d - 1$; we can then pad A and B with zero coefficients, and we have that the degree of $C(x)$ is less than N.

There's a relationship between polynomial and integer multiplication as well. In lecture 1, we covered Karatsuba's algorithm, which given integers α, β, we want $\gamma = \alpha \times \beta$. The digits of α would be $\alpha_{N-1}\alpha_{N-2}\cdots\alpha_0$, and similarly for β. Notice that we can define

$$A(x) := \alpha_0 + \alpha_1 x + \cdots + \alpha_{N-1} x^{N-1}$$
$$B(x) := \beta_0 + \beta_1 x + \cdots + \beta_{N-1} x^{N-1}$$

Notice that $\alpha = A(10)$ and $\beta = B(10)$. For polynomial multiplication, we want $\gamma = \alpha\beta = (A \cdot B)(10)$.

However, we aren't guaranteed that the coefficients of C are less than 10, but they are still small—the coefficient can be at most 81, meaning our terms are at most $81n$, which is an $\log n$ digit number.

This means that if we can multiply polynomials quickly, we can also multiply numbers quickly—the problems are very related in this way.

4.1.1 Algorithms

Algorithm 1: "Straightforward" algorithm. Let us take a look at what polynomial multiplication means. We've defined $C(x)$ before as

$$A(x) \cdot B(x) = C(x) = c_0 + c_1 x + \cdots + c_{2d-2} x^{2d-2}.$$

We have

$$\begin{aligned} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ &\vdots \\ c_k &= \sum_{i=0}^{k} a_i b_{k-1} \end{aligned}$$

In other words, c_k is the sum of all products of pairs whose indices sum to k. The straightforward algorithm is then

- Loop over $i = 0$ to $N-1$
- Compute c_i with a loop from $j = 0$ to i

We have two loops of at most N steps, so the total time is $O(N^2)$. We're a little bit sloppy here because the inner loops range from 1 to $N-1$ iterations. However, this algorithm is still $\Omega(N^2)$, as we're essentially adding $1 + 2 + \cdots + N - 1$; this means that this algorithm is $\Theta(N^2)$.

Algorithm 2:

$$\begin{aligned} A(x) &= \overbrace{a_0 + a_1 x + a_2 x^2 + \cdots}^{A_l(x)} + \overbrace{a_{\frac{N}{2}} x^{\frac{N}{2}} + \cdots + a_{d-1} x^{d-1}}^{A_h(x)} \\ B(x) &= \underbrace{b_0 + b_1 x + b_2 x^2 + \cdots}_{B_l(x)} + \underbrace{b_{\frac{N}{2}} x^{\frac{N}{2}} + \cdots + b_{d-1} x^{d-1}}_{B_h(x)} \end{aligned}$$

We have $A(x) = A_l(x) + x^{\frac{N}{2}} A_h(x)$ and $B(x) = B_l(x) + x^{\frac{N}{2}} B_h(x)$, and with the Karatsuba trick, we can get away with only three multiplications after expanding this out.

As such, we have $T(N) \le 3T\left(\frac{N}{2}\right) + \Theta(N)$, meaning $T(N) = \Theta(N^{\log_2 3})$

We can take a little detour on polynomial interpolation to get a much faster algorithm.

Theorem 4.1: Polynomial Interpolation

A degree $< N$ polynomial is fully determined by its evaluation on N distinct points.

Proof. Why does interpolation work? We have the following system of equations:

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & \cdots & x_0^{N-1} \\ 1 & x_1 & x_1^2 & x_1^3 & \cdots & x_1^{N-1} \\ 1 & x_2 & x_2^2 & x_2^3 & \cdots & x_2^{N-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N-1} & x_{N-1}^2 & x_{N-1}^3 & \cdots & x_{N-1}^{N-1} \end{bmatrix}}_{\mathbf{V}} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-1} \end{bmatrix}}_{\vec{c}} = \underbrace{\begin{bmatrix} y_0 = C(x_0) \\ y_1 = C(x_1) \\ y_2 = C(x_2) \\ \vdots \\ y_{N-1} = C(x_{N-1}) \end{bmatrix}}_{\vec{y}}.$$

We have $\mathbf{V}\vec{c} = \vec{y}$, so this means that $\vec{c} = \mathbf{V}^{-1}\vec{y}$ if $\mathbf{V}$ is invertible—that is, if $\det(\mathbf{V}) \neq 0$.

It turns out that $\mathbf{V}$ is called a "Vandermonde" matrix, and we know that $\det(\mathbf{V}) = \prod_{i<j}(x_i - x_j)$. □

An idea here is that rather than obtaining the coefficients of $C(x)$ directly by multiplying $A \cdot B$, we will determine $C(x_0), C(x_1), \ldots, C(x_{N-1})$ for distinct x_i. Since this is N evaluations of C, we know the polynomial.

How do we get these evaluations? Notice that $C(x) = A(x) \cdot B(x)$, so we can compute the evaluation of A and B on N distinct points each, multiply them, then interpolate to get back the coefficients of $C(x)$.

However, we can't use our matrix-vector solution we discussed before; just inverting a matrix takes $O(N^3)$ flops. All together, this runtime is $O(N^3)$; this looks like a terrible idea. It turns out that there are faster ways to invert a matrix, but it'll still be worse than $O(N^2)$.

Now, let's get into the Fast Fourier Transform. The key here is that we can choose our evaluation points $x_0, \ldots, y_{N-1}$.

4.2 Fourier Transforms

There are two types of Fourier Transforms:

- The *Discrete Fourier Transform* (DFT) is a *matrix*.
- The *Fast Fourier Transform* (FFT) is an *algorithm*

Definition 4.2: Discrete Fourier Transform

Suppose we define the complex number $\omega = e^{\frac{1}{N}2\pi i}$, or the "primitive Nth root of unity"

The DFT matrix $\mathbf{F}$ has elements $F_{ij} = \omega^{ij}$.

With this, we will evaluate the polynomial at points $1, \omega, \omega^2, \ldots, \omega^{N-1}$, meaning we just have the DFT matrix as our $\mathbf{V}$.

What is FFT? It is an algorithm for quickly computing $P(1), P(\omega), \ldots, P(\omega^{N-1})$ for some degree $< N$ polynomial P. In other words, it is applying DFT to the vector of P's coefficients $p_0, \ldots, p_{N-1}$. We further need to assume that N is a power of 2 (if it isn't, just pad a bunch of zero coefficients).

How does it work? If we look at $P(z) = p_0 + p_1 z + p_2 z^2 + \cdots + p_{N-1} z^{N-1}$, we can rewrite it as

$$P(z) = \left(p_0 + p_2 z^2 + p_4 z^4 + p_6 z^6 + \cdots + p_{N-2} z^{N-2}\right) + z \cdot \left(p_1 + p_3 z^2 + p_5 z^4 + \cdots + p_{N-1} z^{N-2}\right).$$

This means we have $P(z) = P_{even}(z^2) + z \cdot P_{odd}(z^2)$. There's a recursive structure here; we're evaluating a polynomial of degree $\frac{N}{2}$ on z^2.

The key here in using the roots of unity is that if we take the roots of unity and square them, we get the $\frac{N}{2}$th roots of unity—this means that we only need $\frac{N}{2}$ evaluations for $P_{even}(z^2)$ and $P_{odd}(z^2)$.

We'd like to write a recurrence relation; $T(N) = 2T(\frac{N}{2}) + \Theta(N)$. We have two recursive calls on degree $\frac{N}{2}$ polynomials and $\frac{N}{2}$ points, and we need $\Theta(N)$ time to combine them together. As such, with the Master Theorem, we have $T(N) = \Theta(n \log n)$.

Algorithm 3: FFT. Given as inputs the coefficients of $A(x)$ and $B(x)$ (in vectors $\vec{a}$ and $\vec{b}$), we use the following procedure:

1. Use FFT to compute $\hat{\vec{a}} := \mathbf{F}\vec{a}$.
2. Use FFT to compute $\hat{\vec{b}} := \mathbf{F}\vec{b}$.
3. For $i = 0$ to $N-1$, compute $\hat{c}_i = \hat{a}_i \times \hat{b}_i$; here, $\hat{\vec{c}}$ is the evaluation of C on $1, \omega, \ldots, \omega^{N-1}$.
4. We let $\vec{c} = \mathbf{F}^{-1}\hat{\vec{c}}$, and return $\vec{c}$, the coefficient vector of $C = A \cdot B$.

The only thing left is to compute the inverse of the DFT matrix, and how we can apply it to $\hat{\vec{c}}$ quickly.

We claim that $\mathbf{F}^{-1} = \frac{1}{N}\overline{\mathbf{F}}$. Here, $\overline{\mathbf{F}}$ is the entry-wise complex conjugate, and we can verify this claim by just computing the product.

Another claim is that for any matrix $\mathbf{M}$ and vector $\vec{x}$, we have $\overline{\mathbf{M}}\vec{x} = \overline{\mathbf{M}\bar{\vec{x}}}$.

This means that computing $\mathbf{F}^{-1}\hat{\vec{c}}$ requires doing another FFT (to compute $\mathbf{F}\bar{\hat{\vec{c}}}$).

Hence, we have 3 FFTs, with some extra linear time to do some combining. In total, we need $O(N \log N)$ time to compute the product, assuming we can multiply/add complex numbers in $O(1)$ time.

4.3 Cross-correlation

Definition 4.3: Cross-correlation

We have two inputs $\vec{x} = (x_0, x_1, \ldots, x_{m-1})$ and $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$, where $n \geq m$.

We want all the shifted dot products of x with y. That is, we want

$$x_0 y_0 + x_1 y_1 + \cdots + x_{m-1} y_{m-1}$$
$$x_0 y_1 + x_1 y_2 + \cdots + x_{m-1} y_m$$
$$\vdots$$

This is a collection of $n - m + 1$ numbers.

Here, notice that we can use polynomial multiplication to compute the cross-correlation; if we reverse the coefficients of $A(x)$, the coefficients of the product of $A(x)$ and $B(x)$ starting from term $m-1$ gives us the cross-correlation.

9/9/2021

Lecture 5

Graphs

5.1 Graphs

Definition 5.1: Graph

A graph G is a pair (V, E), consisting of the vertex set and the edge set.

If no self-loops are allowed, we call the graph *simple*.

There are two kinds of graphs we'll be talking about in this class.

Definition 5.2: Directed Graph

In a directed graph, the edge set $E \subseteq V \times V$, an ordered pair of vertices.

Definition 5.3: Undirected Graph

In an undirected graph, the edge set E is a set of unordered pairs from V.

Example 5.4

We can utilize graphs in many different ways.

- Road network: Vertices are intersections, and edges are road segments connecting intersections.

  We could compute things like shortest paths, the travelling salesman problem, etc.

  Should we represent these networks as directed or undirected graphs? Perhaps most roads are two-way roads, but there do exist one-way roads, so we should use a directed graph.

  We could also *weight* the graph. That is, each road could be associated with a distance or time.

- Social networks: Vertices are people, edges are friends.

  Twitter and Instagram would be directed (followers), whereas Facebook would be undirected (friends).

How do we represent a graph on a computer?

For us, we will always assume the set of vertices is $\{1, \dots, n\}$. We can represent edges in two main ways:

1. *Adjacency matrix* representation: We represent edges in a matrix $\mathbf{A}$, where

$$A_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & (i,j) \notin E \end{cases}.$$

   if the graph was weighted, we could represent elements as

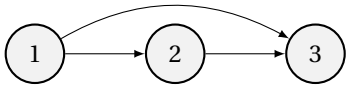
$$A_{ij} = \begin{cases} w_{ij} & (i,j) \in E \\ \infty & (i,j) \notin E \end{cases}.$$

   In a computer, this is just an $n \times n$ array of booleans or numbers.

2. *Adjacency list* representation: We represent edges as (for example) an array of linked lists.

   Here, $B[i]$ is a linked list containing all j such that $(i,j) \in E$.

   As an example, if we have



The differences between the two can be summarized below. Here, we define $n := |V|$ and $m := |E|$.

	Adjacency Matrix	Adjacency List
Space	n^2 bits	$\Theta(m+n)$ words
$(u,v) \in E$	$O(1)$ time	$O(\deg(u))$ time
enumerate neighbors of u	$\Theta(n)$ time	$\Theta(\deg(u))$ time

Some notes:

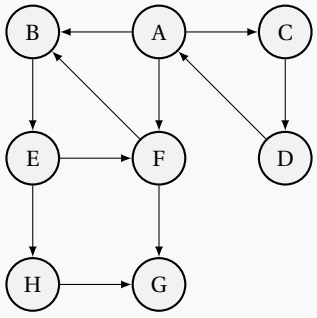
- Adjacency list space: We're always storing the array B of all vertices, of length n. In total, we're also storing $\Omega(m)$ linked list entries for the edges (in an undirected graph, we'd store each edge twice).
- Adjacency list edge existence: Here, $\deg(u)$ is the degree of vertex u; the number of vertices w such that $(u, w) \in E$. We could improve this runtime if we change the data structure to something like a hash set instead of a linked list.

In general, the adjacency list is better in these aspects we've touched on.

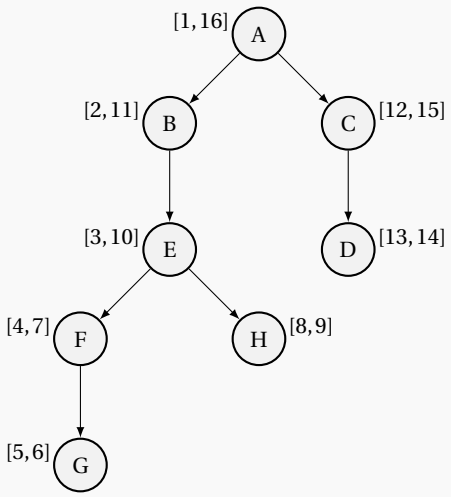
5.2 Depth First Search

```
def DFS(V, E):
    global V, E
    global clock = 1
    global visited = bool[n]  # initialize to all False
    global preorder, postorder = int[n]

    for v in V:  # for each vertex
        if not visited[v]:  # if it's not visited yet
            explore(v)  # we explore the vertex, along with other neighbors

def explore(v):
    visited[v] = True
    preorder[v] = clock++  # set preorder, then increment clock

    for w in neighbors(v):  # for each neighboring edge of v
        if not visited[w]:  # if it's not visited yet
            explore(w)  # recurse and explore the vertex

    postorder[v] = clock++  # set postorder, then increment clock
```

Example 5.5



If we run DFS on this graph starting from A (breaking ties alphabetically), we have



Here, the preorder and postorder traversal labels for vertex v are in the form `[pre(v), post(v)]`.

A DFS search from vertex u (that is, instead of looping over all vertices, we just start at one vertex and run `explore`) explores exactly the set of vertices V such that there exists a path from u to v.

Firstly, if `explore(v)` is called, there exists a path between u and v. The path is simply the chain of recursive calls that got us from u to v.

Next, if there is a path between u and v, then we will always call `explore(v)`. For contradiction, suppose there exists a reachable vertex v which doesn't get explored. If v is reachable, then there exists some path $(x_1 = u, x_2, \ldots, x_r = v)$ going from u to v in the graph. But we know that u is explored, so there must be some vertex on this path x_j which is not explored.

If we look at the vertex right before, then it must be the case that we're recursing over all neighboring vertices, and visiting them all. The only reason why x_j might not be explored is if it was already visited—but we've assumed that it was never visited, which is a contradiction. As such, we must visit v if there is a path from u to v.

The runtime of DFS depends on whether we're using adjacency matrix or adjacency list representation.

If we look at the code, we know that every vertex will get looked at, and will get looked at exactly once (as we don't recursively explore visited vertices). Inside of `explore`, we're enumerating the neighbors of v. That is,

$$\text{Total time} = \Theta(n) + \sum_{u \in V} (\text{time to enumerate neighbors of } u).$$

For an adjacency matrix, we have $\Theta(n^2)$ time. For an adjacency list, we have $\Theta(\sum_{u \in V} \deg(u)) = \Theta(n+m)$ because each edge gets looked at exactly once.

5.2.1 Applications of DFS

DFS has many different applications.

- Reachability; i.e. can we reach a vertex u?
- Identifying connected components (in undirected graphs)
- Articulation points; i.e. the set of vertices whose removal would disconnect the graph

  For example, the graph of a square, C_4, has no articulation points, but the graph of two triangles connected at a vertex has one articulation point.
- Finding biconnected/triconnected components (in an undirected graph, with a modified DFS)

A singly-connected components is what we typically call a connected component (i.e. every vertex in the same component can reach every other vertex in the set).

A biconnected component (or doubly-connected component) is where there exists at least two paths between any two vertices in a connected component, where no two paths share any edges.

A triconnected component (or triply-connected component) is very similar; each pair of vertices has at least three paths that do not share any edges.

- Strongly connected components

  Two vertices are strongly connected if you can get from u to v and from v to u. We'll cover this more next week.

- Planarity testing
- Isomorphism of planar graphs

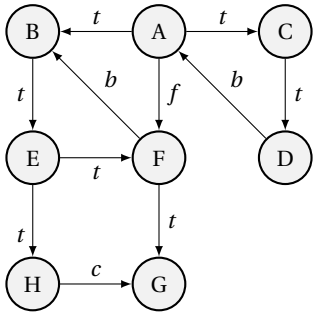
5.2.2 Preorder and Postorder Intervals

The set of intervals `[pre(u), post(u)]` are pairwise nested or disjoint. That is, if we have two vertices u and v, if we look at `[pre(u), post(u)]` and `[pre(v), post(v)]`, the two intervals are either disjoint, or the two are nested in one another.

5.2.3 Types of Edges in a Graph

- Tree edge: a traversed edge during DFS
- Forward edge: goes from an ancestor to a descendant in the DFS tree, and is not a tree edge
- Back edge: goes from a descendant to an ancestor in the DFS tree
- Cross edge: all other edges

As an example, t denotes a tree edge, b denotes a back edge, f denotes a forward edge, and c denotes a cross edge.



Why is this useful? Suppose (u, v) is an edge. We have the following two claims:

1. `postorder(u)` < `postorder(v)` if and only if (u, v) is a back edge
2. G has a cycle if and only if there exists a back edge

9/14/2021

Lecture 6

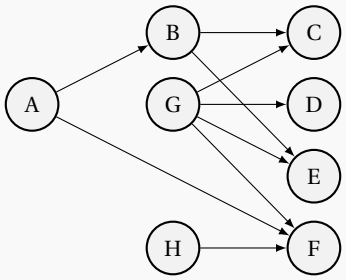
Topological Sort, Strongly Connected Components

6.1 Topological Sort

In a topological sort, the input is a DAG (directed acyclic graph), and the goal is to output an ordering of vertices such that if $(u, v) \in E$, then u must come before v in the ordering.

Note that this problem is not solvable if there is a cycle (because it's impossible to order a cycle like this), hence the restriction to a DAG.

Example 6.1



An example of a topological sort of this graph is $ABHGCFDE$.

To verify whether this is a valid topological sort is to go through every edge and ensure that if (u, v) is an edge, u comes before v.

The maximum number of topological sorts in a graph is $n!$, for the empty graph. It is also possible that there exists a unique topological sort, for a graph where $a \to b \to c \to \cdots$.

What's the use of a topological sort? We can envision each vertex as a function in code, and an edge between u and v tells us that u calls v as a subroutine. Suppose we were trying to test this code. One way we could do this is start with v, and ensure that they're correct, and then move on to u, etc., moving our way up the topological sort. This is because we want to ensure that any bugs we find is actually in the function we're currently testing, rather than a subroutine.

Algorithm 1: Brute force search. Try all permutations of V, and return the first one that is a topological sort.

The runtime is $O(n! \cdot mn)$. We try all $n!$ permutations, and check all m edges to make sure that it's respected (adding an additional factor of n if we don't know where the vertices are in the topological sort).

Algorithm 2: Suppose we define a "source" as a vertex with no incoming edges, and a "sink" as a vertex with no outgoing edges. We can show that every DAG has a source (as a lemma).

We can iteratively peel off source vertices from the graph one at a time, and the resulting order is a topological sort.

We could implement this algorithm in linear time, $O(n + m)$, with arrays and doubly linked lists (it's a little complex, but there's a better solution next).

Algorithm 3: Post-order numbers. Suppose we run a DFS through the DAG.

We claim that the vertex with the largest `post(v)` must be a source.

Proof. Suppose that this vertex (let's call it v) is not a source. This means that there exists a vertex u such that $(u, v) \in E$. We have only two possibilities; the intervals for u and v must either be nested or disjoint.

In the first case (nested), (u, v) is a back edge (as u has to be nested in v, because v has the highest post-order), and thus must contain a cycle, which is impossible. In the second case (disjoint) is impossible (because it would mean

that v was never visited within u's recursive sub-tree, even though there was an edge from u to v; further v can't have been visited before, because it has the largest post-order number). □

As such, we can do DFS in linear time and take the vertices as we set the post-orders (this would mean that we get vertices with sorted post-order numbers). This is because we can peel off the vertex with the largest post-order, and the new vertex with largest post-order is our next source (and we can just take the existing DFS graph because it's still valid for the remaining tree).

6.2 Strongly Connected Components

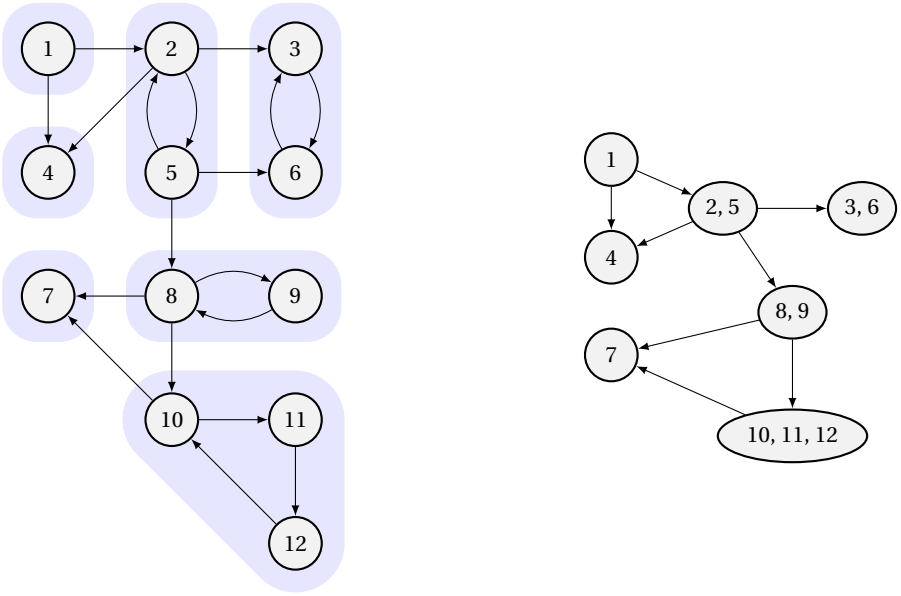
Definition 6.2: Strongly Connected

The vertices u and v are strongly connected if u has a path to v and vice versa.

Definition 6.3: Strongly Connected Component

A *strongly connected component* (SCC) is a maximal subset of strongly connected vertices.

Here is an example graph, with strongly connected components highlighted, along with a graph for the strongly connected components, called the "SCC Graph"



We claim that the SCC graph is always a DAG; if it had a cycle, then the components in the cycle could all reach each other, and should have been a single vertex in this SCC graph.

We could again do a brute force to find SCCs, but we'll skip over this.

Algorithm 1: We can do n DFS's (one from each vertex) to create a 2D array where (u, v) is 1 if u can reach v. This means that we must have (u, v) and (v, u) in order for u and v to be strongly connected.

Looking at the runtime, we need $O(n(n+m))$ in order to do the n DFS's, and an additional $O(n^2)$ for the last scan through the array, which in total is $O(n^2 + nm)$.

Algorithm 2: We can identify some vertex in a sink SCC, do a DFS, and peel it off. Each time we leave a component and go to a previously visited component, we've found an edge in the SCC graph. The crucial part here is that our DFS's are all proportional to the size of the component we're doing the DFS in.

This means that we have $O(n+m)$ runtime overall, because we're summing over some constant factor times $(n+m+o)$ for each component, where n is the number of vertices in the component, m is the number of edges within the component, o is the number of edges leaving the component.

One issue here is that we need to find a sink in the SCC graph. We can do this cleverly by reversing all vertices, and finding a source in this reversed graph (which would correspond to a sink in the original graph). How do we find a vertex in the source SCC of the reversed graph? The highest post-order number still works, by a similar case analysis as before.

9/16/2021

Lecture 7

Single Source Shortest Paths

We want an algorithm to compute all the shortest paths from one node to all others. That is, given an input digraph G and a starting node s, we want to compute `dist[v]`, the length of the shortest path between s and v, and `prev[v]`, the previous vertex to v on the shortest path from s to v.

There are many algorithms for SSSP; we'll be talking about four in CS170.

1. Breadth-First Search (BFS)

   With BFS, we must assume all edges have weight 1 (that is, the "unweighted" case)

2. Dijkstra's

   With Dijkstra's, we must have arbitrary edge weights but must be nonnegative.

3. Bellman-Ford

   With Bellman-Ford, the edge weights can be completely arbitrary.

4. DP on DAGs

   With DP, the edge weights are arbitrary but the graph must be a DAG.

7.1 Breadth-First Search

```
def BFS(G, s):
    init dist[1..n] to all inf
    init prev[1..n] to all None
    init vis[1..n] to all False

    init Q = empty queue
    dist[s] = 0
    vis[s] = True
    Q.push(s)

    while Q.size > 0:
        u = Q.pop()  # pop the front
        for (u, v) in E:
            if not vis[v]:
                vis[v] = True
                dist[v] = dist[u] + 1
                prev[v] = u
                Q.push(v)
    return dist, prev
```

Let's look at the runtime of this algorithm. All of the initializations are either linear or constant. In the main loop, each vertex is pushed onto the queue at most once (as once it's pushed to the queue, it's marked as visited, so it's

never visited again). When a vertex is popped off the queue, we loop over each outgoing edge and do constant work.

Hence, we have $\Theta(n) + \sum_{v \in V} C \cdot \big(1 + \text{outdeg}(v)\big)$, hence we have $O(m+n)$ runtime.

Looking at correctness, we have two lemmas. First, $\forall v \in V$, `dist[v]` $\geq d(s,v)$ where $d(s,v)$ is the shortest path length from s to v. Second, if we look at any point in time, suppose we have $Q = [v_1, \ldots, v_r]$. Then, (1) $\forall i$, `dist[`v_i`]` $\leq$ `dist[`v_{i+1}`]`, and (2) `dist[`v_r`]` $\leq$ `dist[`v_1`]` $+ 1$. Both these lemmas can be proven through induction.

Theorem 7.1: BFS Shortest Paths

BFS finds the shortest paths from s.

Proof. Suppose for contradiction that the `dist` array is incorrect. This means that $\exists v$ such that `dist[v]` $> d(s,v)$.

Let's take the v such that $d(s,v)$ is minimal (as there could be many such v). Note that we must have $v \neq s$, as we've already initialized `dist[s]` $= 0$.

If we take a shortest path from s to v, it must start at s and end at v; let's say this path goes through vertices $\{s = v_0, v_1, \ldots, v_{r-1}, v_r = v\}$. We claim that it can't be the case that v's distance is wrong.

If we look at the point in time that v_{r-1} was popped off Q, we know that v_{r-1} is set correctly (as we've assumed v is of the minimum distance), and as such when it's popped off, it sets `dist[v]` to be $1 +$ `dist[`v_{r-1}`]`, which would be correct.

The only way in which this doesn't happen is if v was already visited—this can only happen if v was visited by some other v' with `dist[`v'`]` $\leq$ `dist[v]`. However, this means that `dist[v]` will be set to `dist[v'] + 1`, which is at most `dist[`v_{r-1}`]` $+ 1 \leq d(s,v)$.

This is a contradiction, so BFS does actually give the shortest paths from s. □

7.2 Dijkstra's Algorithm

If all edge weights are all integers, then we can reduce this to the unweighted case by subdividing the edges. That is, replace an edge with weight t with $t-1$ connected vertices. Running BFS on this bigger graph gives us the shortest paths. However, this sacrifices runtime, as we're growing the graph by a factor of $L = \max(W)$, hence we have $O((n+m)L)$, or if you're a little more careful with isolated vertices, $O(n+mL)$.

We can do better with Dijkstra's, using heaps (i.e. priority queues). As a recap, in a heap, we want to maintain a set S of key-value pairs subject to three operations:

- `insert(k, v)`: inserts (k,v) into S
- `delMin()`: returns the (k,v) pair with the smallest k (breaking ties arbitrarily), and removes it from S.
- `decKey(*o, k')`: we replace the old key with $k' < k$

There are many different data structures that implement this, and we'll be using it as a black box.

```
def Dijkstra(G, s):
    init minheap H
    init dist[1..n] to all inf
    init prev[1..n] to all None

    for v in V:
        H.insert(inf, v)  # key=inf, val=v

    dist[s] = 0
    H.decKey(s, 0)

```

```
    # main loop
    while H.size > 0:
        u = H.delMin()
        for (u, v) in E:
            if dist[u] + w(u, v) < dist[v]:  # better to go through u
                H.decKey(v, dist[u] + w(u, v))
                dist[v] = dist[u] + w(u, v)
                prev[v] = u
    return dist, prev
```

Looking at the runtime, we do n calls to `delMin()`, and at most one `decKey` per inner loop. Suppose t_I is the time to insert, t_{dk} is the time to decrease key, and t_{dm} is the time to delete min.

We then have $O(n + m + n \cdot t_I + n \cdot t_{dm} + m \cdot t_{dk})$.

With a binary heap, we'd have $O((m+n)\log n)$, as each operation is $O(\log n)$. With a linked-list (one of the worse cases), we'd have $O(m+n^2)$, as insert/delete are constant, but `delMin` is $O(n)$. With a Fibonacci heap (the best case), we'd have $O(m + n\log n)$, as insert/delete are constant, and `delMin` is $O(\log n)$.

9/21/2021

Lecture 8

Minimum Spanning Trees

8.1 Dijkstra's Algorithm Correctness

Lemma 8.1

Any non-∞ `dist[u]` value corresponds to some `s` $\rightarrow$ `u` path.

Proof. The proof is omitted here, but we can prove it by induction on the number of steps in the algorithm. □

Lemma 8.2

The final `dist` and `prev` outputs are correct. That is, for all u, `dist[u]` is the shortest path distance from s to u, i.e. `dist[u]` $= \delta(s,u)$, and there exists a shortest path from s to u whose last edge is `(prev[u], u)`.

Proof. We'll focus on the distances here.

Let A be the set of all vertices that have been popped off of the heap so far. The claim here is that at all points in time, the `dist` values are correct in A. This would then imply that the `dist` values are correct everywhere, as every point will be popped off the heap at some point.

We will prove by induction on $|A|$ that `dist[u]` $= \delta(s,u)$, for all $u \in A$.

In our base case, if $|A| = 1$, then $A = \{s\}$, and we know that `dist[s]` $= 0 = \delta(s,s)$.

In the inductive step, suppose we have $|A| = k+1$, and we just added v to A.

If we look at the most recent time that v's key was set, there was some edge (u,v) and some path from s to v through u. Suppose there was another path P' from s to v, with a path Q from s to x, where x is the last vertex before we finally leave A.

Suppose we look at the length of P' (denoted as $L(P')$):

$$\begin{aligned} L(P') &\geq L(Q) + w(x,y) \\ &= \delta(s,x) + w(x,y) \\ &= \texttt{dist[x]} + w(x,y) && \text{(IH)} \\ &\geq \texttt{dist[y]} \\ &\geq \texttt{dist[v]} && \text{(since } v \text{ was popped off the heap)} \\ &= L(P) \end{aligned}$$

□

8.2 Minimum Spanning Trees

Definition 8.3: Tree

A tree is an undirected graph that is connected and acyclic.

Definition 8.4: Spanning Tree

A spanning tree of a graph G is a subgraph of G with $|V(G)|$ vertices which is a tree. That is, we take a subset of the edges and all vertices of G to create a tree.

If we have edge weights, minimum spanning tree is the spanning tree with minimum total edge weight.

Lemma 8.5

Any of the following two implies the third for a tree T with n vertices:

- $|E(T)| = n-1$
- T is connected
- T is acyclic

Algorithm 1: Brute force.

Try all subgraphs of G with exactly $n-1$ edges, and check connectivity with DFS; we take the subgraph with the minimum edge weight. Looking at runtime, we have $\binom{m}{n-1} \cdot (m+n)$ as we have $\binom{m}{n-1}$ possible subgraphs and we run DFS with $m+n$ time on each. Using the fact that $\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$, we have that

$$\binom{m}{n-1}(m+n) \leq \binom{n^2}{n-1} \cdot n \leq \left(\frac{en^2}{n-1}\right)^{n-1} \cdot n \leq (6n)^n.$$

Algorithm 2: We'll give a meta algorithm, and other fast algorithms are implementations of this algorithm.

```
def MST(G):
    X = ∅
    while |X| < n-1:
        pick S ⊊ V(G), S ≠ ∅ such that no edge crosses (S, V \ S) in X
        // ^ that is, we partition the graph G into two subsets, one of which is S
        let e = (a, b) be the minimum weight edge in G crossing the partition
        X = X ∪ {e}
    return X
```

Note that there's a lot of things that we have omitted here; we never specify how to pick the set S, and whether there even are any edges crossing this partition, etc.

Lemma 8.6

At all points in time, there exists an MST T containing X. That is, this will imply that X itself is an MST by the end of the algorithm.

Proof. We proceed by induction on $|X|$. Our base case where $|X| = 0$ is trivial; any MST contains the empty set.

For the inductive step, we assume that the claim is true for X so far (that is, $X \subseteq T$), and we're about to add one more edge $e = (a, b)$ where $a \in S$. In one case, e was in T, and we're done. In the other, e was not in T; this means that we still have some path from a to b in T. We can see that together, the path from a to b in T and the edge e is a cycle.

Suppose we delete another edge that has crossed the partition. We know that e is the edge of minimum weight, so this new path from a to b must have smaller weight (as we've removed a more expensive edge and added the cheaper edge e). □

If we look at one potential runtime, we can do this with $n-1$ DFS's, which in total is $O(mn)$ time.

8.3 Prim's Algorithm

Algorithm 3: Prim's Algorithm

```
def Prim(G, s):
    X = ∅
    S = {s}
    repeat n-1 times:
        e = minimum-weight edge (a, b) that crosses (S, V \ S), where b ∉ S
        S = S ∪ {b}
        X = X ∪ {e}
    return X
```

In summary, we're building X one edge at a time, and adding vertices to S one at a time, expanding it.

We can rewrite this algorithm using a heap to make this more efficient:

```
def Prim(G, s):
    X = {}
    H = heap()
    for i in range(1, n+1):
        H.insert(i, inf)
    from[1..n] = all null
    H.decKey(s, 0)

    while H.size() > 0:
        u = H.delMin()
        if u != s:
            X |= {(from[u], u)}
        for (u, v) in E:
            if v in heap and v.key > w(u, v):
                H.decKey(v, w(u, v))
                from[v] = u
    return X
```

Looking at the runtime, we have some initialization that takes $O(n)$ time, n insertions, n `delMin`'s, and m `decKey`'s.

With a binary heap, all the operations take $\log n$ time, so the total is $O(m \log n)$. With a Fibonacci heap, we have a constant time insertion, $\log n$ time `delMin`, and constant time `decKey`, which gives us $O(m + n \log n)$.

8.4 Union Find, Kruskal's Algorithm

The problem here is to maintain a partition of $\{1, \ldots, n\}$ subject to two operations:

- `find(a)`: returns the name of the partition that a is in.

  If we call `find` on two elements in the same set, they should return the same name.

- `union(a, b)`: merge the sets containing a and b.

For now, we'll pretend we have such a data structure, and describe Kruskal's algorithm.

Algorithm 4: Kruskal's Algorithm

```
def Kruskal(G):
    sort E(G) in increasing order of weight
    X = ∅
    initialize UnionFind data structure
    for e in E:
        if find(a) != find(b):
            X = X ∪ {e}
            union(a, b)
    return X
```

Looking at the runtime, we need $2m$ finds (twice for each edge), and $n-1$ unions (one for each edge actually added), with $m \log n$ to sort the edges. It turns out that there are union find data structures that are really fast, and that sorting is actually the bottleneck.

9/23/2021

Lecture 9

Greedy Algorithms

Suppose we have a search or optimization problem to find the best/some item in a collection. A greedy algorithm is one that builds the solution iteratively using a sequence of local choices.

It's a little bit hard to pinpoint mathematically and formally what we mean, but an example is with 2SAT. If we look at any given variable, we can set a variable to either true or false that satisfies as many clauses that we can at this moment. The meta algorithm for MSTs is also greedy; we choose the minimum weight edge across the partition at every iteration of the algorithm. A brute force algorithm is not greedy; a brute force algorithm would go through all possible final results and pick the best one—there are no iterative local choices.

We'll be looking at three problems today: scheduling, Huffman encoding, and set covers.

9.1 Scheduling

The input is a collection of jobs that need to be done. Each job has an associated time interval $[x_i, y_i]$ where $x_i < y_i$. We have one person that wants to do as many jobs as possible without any time conflicts (that is, no two jobs taken can overlap).

Attempt 1: Greedily keep doing the shortest possible remaining job that does not conflict with earlier ones.

This doesn't work. We could have two long jobs with one short job overlapping between the two.

Attempt 2: Greedily pick the next job to have the shortest y_i (end time) without conflicting with any of the jobs already done.

The runtime here would be $O(n \log n)$; we can sort the input by end time and iterate over the intervals to take the next job whose start time is after the previous taken job's end time.

We can prove that this works. Generally, with proofs of correctness for greedy algorithms, we go with a proof by contradiction.

Suppose the greedy algorithm takes jobs $G = \{j_1, j_2, \ldots, j_k\}$ in sorted order by end time. That is, we have the intervals $[x_{i_1}, y_{i_1}], [x_{i_2}, y_{i_2}], \ldots$ where $y_{i_1} < y_{i_2} < \cdots < y_{i_k}$.

Suppose this is not optimal. Let us then take the optimal solution S such that $|S \cap G|$ is maximal. We will show that we can actually find another solution that has more in common with G.

Suppose we have $S = \{s_1, \ldots\}$ is also sorted by endpoints. It is impossible that all $\{s_1, \ldots s_k\} = \{j_1, \ldots, j_k\}$ because greedy would have also chosen more at the end. This means that at some point $j_t \neq s_t$.

Then, suppose we construct another S' but just swap out s_t with j_t; there are no conflicts if we swap (as j_t ends earlier or at the same time as s_t—greedy will always choose the job that ends the earliest), and as such S' has more in common with G, which is a contradiction.

9.2 Huffman Problem

We have some alphabet Σ (ex. $\{A, B, \ldots, Z\}$ or $\{A, G, C, T\}$), and some text whose characters come from Σ.

We also have character frequencies; for $\sigma \in \Sigma$, $\text{freq}(\sigma)$ denotes the number of occurrences of σ in the text.

Example 9.1

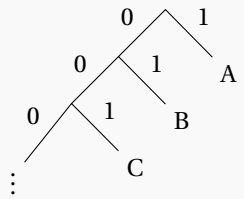
Suppose `A` is much more frequent than the other characters; i.e. `A` appears 10^6 times, while all the other letters appear once. A naive encoding would require 5 bits per character (as 32 is the smallest power of 2 larger than 26), requiring 5 million bits for just the A's. A better encoding would be to assign `A = 1`, `B = 01`, `C = 001`, etc., and we'd only need 1 million bits for just the A's.

One thing to beware though is that encodings may not always be uniquely decodable. For example, if `A = 0`, `T = 1`, `G = 01`, `C = 00`. If we had `001`, this is ambiguous.

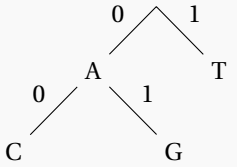
One way for an encoding to be unambiguous is through prefix-freeness. That is, no character's encoding is a prefix of any other character's encoding.

Example 9.2

Taking the encoding proposed in the previous example, we can construct a binary tree:



Compare the previous to this encoding that failed us earlier:



Prefix-free means that the characters are only at the leaves. In the second case, when we hit a character in an internal node, we aren't sure whether we should keep decoding or stop here.

Our goal is to find the optimal prefix-free encoding with the given information.

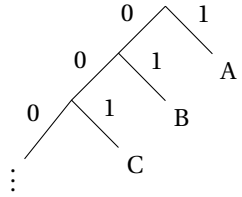
Algorithm 1: Brute force.

Prefix-free encodings form a bijection with binary trees; since we have n characters, we have n leaves. We'd search through all trees on n leaves and assignments of characters to leaves. Since all trees with n leaves have $2n-1$ nodes, we have a total of $\frac{1}{2n}\binom{2(2n-1)}{2n-1}$ possible trees. This is approximately 16^n trees, with $n!$ possible assignments.

This means that the brute force algorithm takes $O(16^n n!)$ time.

A really easy improvement to make is to assign characters to leaves greedily, getting rid of the $n!$ factor, leaving us with just $O(16^n)$ time.

Greedy Attempt 1: Let the most frequent character be the right child of the root, and recurse on the left subtree. This means that we'd have something that looks like this:



This cannot possibly be right. Each letter would use one additional bit, meaning in a 26 letter alphabet the average letter takes 13 bits. In the case of equally weighted letters, we'd want a perfectly balanced tree, using only 5 bits per letter.

Greedy Attempt 2: Suppose we look at the leaf at the deepest depth. This leaf must have a sibling (otherwise, we'd just be wasting a character); there must be an optimal solution that puts the least frequent characters at these deepest leaves.

For contradiction, if we had an optimal solution that put the least frequent at a higher depth, we could just swap them into a lower depth and the encoding will be better. This contradicts optimality.

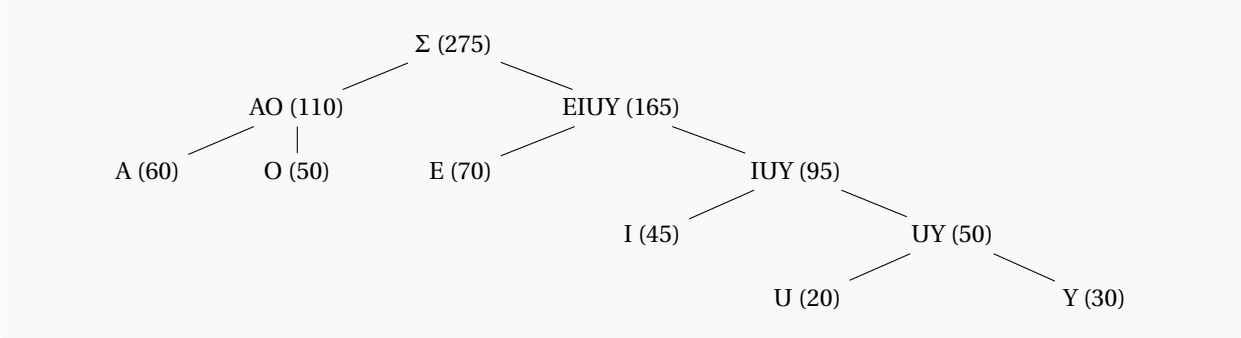
What does this mean for an algorithm? We provide the algorithm through an example.

Example 9.3

Suppose our alphabet are $\Sigma = \{A, E, I, O, U, Y\}$. The frequencies are

$$A = 60 \quad E = 70 \quad I = 45 \quad O = 50 \quad U = 20 \quad Y = 30.$$

We can take the two characters with the smallest frequencies; these will go at the bottom of our tree somewhere. In this case, it is U and Y. Then, we'll merge them together to form a new "character" $\{U, Y\}$ with frequency 50. Repeating this process, taking the next two least frequent "characters", merging them together, and continuing, we'd get the following tree:



What's the runtime of this algorithm? Naively, $O(n^2)$, because we need to look through all the characters to find the smallest two each iteration.

We can do better with a heap; we can insert all the characters into a min-heap, do two `delMin` operations to get the two characters with smallest frequency, and `insert` the merged character back into the heap with the summed frequency. This means that we need a total of $2n$ `delMin` operations and n insert operations (i.e. both linear).

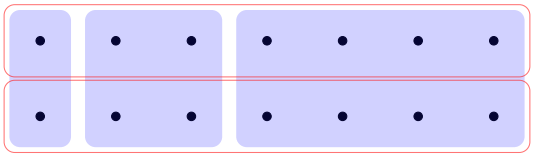
With a binary heap, we'd have $O(n \log n)$ runtime.

9.3 Set Cover

We have a universe of items $\{1, \ldots, n\}$, and a collection $S_1, \ldots, S_m$ of subsets of $[n]$. Our goal is to find a minimum size sub-collection which covers the universe $[n]$.

It turns out that this problem is NP-hard, i.e. we don't expect a fast solution to this problem.

We can see that a greedy algorithm is not optimal; suppose we have the following subsets:



The greedy algorithm that chooses the subsets that cover the most nodes will choose the blue items, whereas the two red subsets have fewer items each, but we only need two subsets to cover all items.

Notice that if we extend this, we can see that greedy can be off by at least a $\Theta(\log n)$ factor.

It turns out that if there exists a polynomial time algorithm for set cover which always uses $\leq \text{OPT} \cdot 0.9999 \ln n$ sets (where OPT is the optimal number of sets), then 3SAT can be solved in time $\leq n^{c \log\log n}$. We don't think the latter can be possible.

The greedy solution uses $\leq OPT \cdot \lceil \ln n \rceil$ sets. This means that the greedy solution is actually almost optimal.

9/28/2021

Lecture 10

Set Cover, Union Find

```
def greedySC(C):
    A = {1, ..., n}  # not covered yet
    B = {}  # sets taken
    while |A| > 0:
        # let j in [m] \ B such that A∩S_j is maximum
        A = A \ S_j
```

```
        B = B | {j}
    return B
```

Our claim is that if OPT (the optimal solution) uses k sets, then $|B| \le \lceil k \ln n \rceil$.

Proof. Let A_t be A after t times through the main loop ($|A_0| = n$). There must be some set S_t^* in OPT covering $\ge \frac{1}{k}|A_t|$ elements in A_t. The greedy algorithm definitely took the set that covered $\ge \frac{1}{k}|A_t|$, because it takes the best set; this means that $|A_{t+1}| \le (1 - \frac{1}{k})|A_t|$. By induction on t, this implies that $|A_L| \le \left(1 - \frac{1}{k}\right)^L |A_0| = n\left(1 - \frac{1}{k}\right)^L$. As soon as this last value is ≤ 1, then we're done (because sizes are integers).

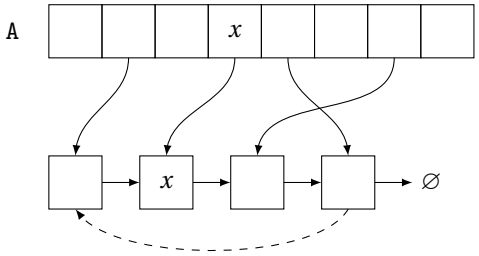
Since $1 + x < e^x$, taking $x = -\frac{1}{k}$ we have that $|A_l| < ne^{-\frac{L}{k}}$, which is ≤ 1 for $L = \lceil k \ln n \rceil$. This means that it takes at most $\lceil k \ln n \rceil$ steps to terminate. □

10.1 Union Find

Algorithm 1: Store an array `A[1 ...n]`, where `A[x]` is the name of x's set. Initially, `A[x] = x` for all x.

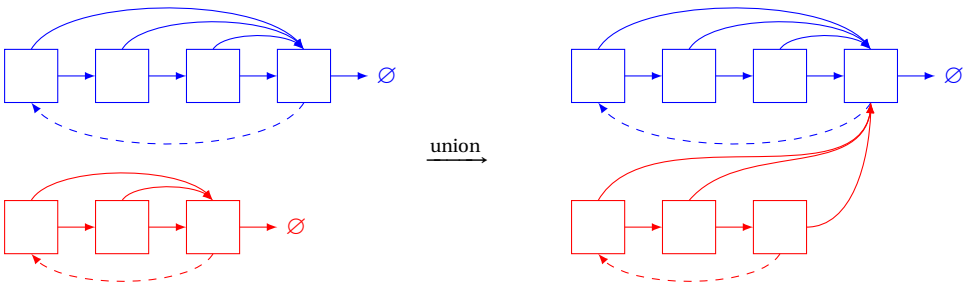
- `Find(x)`: `return A[x]`, which is constant time.
- `Union(x, y)`: let α = `A[x]` and β = `A[y]`; iterate through the array, and if `A[i]` $= \beta$, then we set `A[i]` $= \alpha$. This is linear time.

Algorithm 2: Represent each x by a linked list node. That is, we're representing sets as linked lists, and our data structure is a collection of linked lists.



- `Find(x)`: return the head of x's list; this takes linear time.
- `Union(x, y)`: if we store a pointer to the last element in the linked list from the head as an extra field, then all we need to do is travel to the head of both linked lists, and we have two options; either put x's list under y's list, or vice versa. This means that we take linear time (to traverse to the head, plus a constant number of pointer changes).

Algorithm 3: The same as algorithm 2, except each node contains a pointer to the head of the list.



- `Find(x)`: follow the head pointer and return the value; this takes constant time.
- `Union(x, y)`: do the same merge, but we also have to update all elements of one list (as they're now outdated); this takes linear time. We can put the smaller list under the larger list to reduce the amount of pointer changes we need to do.

Our claim is that when using both optimizations in algorithm 3, any sequence of $\leq m$ finds and $\leq$ unions takes total time $O(m + n \log n)$.

We can see that this is true, because find takes $O(1)$ time, and union takes $O(1) + O(\#$ head pointer changes$)$. The latter runtime over all operations is just the sum of the total number of times we changed any node's head pointer. Since whenever we change the head pointer, we at least double the size of the set (as the smaller set's head pointer gets changed), the number of times we change any fixed node's head pointer is at most $\log_2 n$ (as the max size of a set is n). This means that our total runtime is $O(m + n + n \log n) = O(m + n \log n)$.

Algorithm 4: Disjoint Forest

When finding the root, we can just change the intermediate nodes to point directly at the head. This means that each set is now a collection of rooted trees, and to union, we can just change the head of one tree to point to the head of the other tree (this is called "path compression").

Taking the other aforementioned optimization into account, we can also improve our amortized runtime by putting the shallower tree below the taller tree (this is a "union by height"). However, there is a problem if we use path compression with union by height; the height of a tree could change if we do a find in the middle. A solution is to just pretend that path compression doesn't exist, and maintain the (potentially incorrect) imaginary heights (this is usually called "rank" rather than "height" to avoid confusion, as this isn't really the height of the tree).

Here is some pseudocode for the disjoint forest algorithm:

```
p[1..n]  # parents array
r[1..n]  # rank array
def makeset(x):
    """Makes x into a singleton set"""
    p[x] = x
    r[x] = 0

def find(x):
    if x == p[x]: return x
    p[x] = find(p[x])  # path compression
    return p[x]  # p[x] is now root

def union(x, y):
    x = find(x)  # root of x
    y = find(x)  # root of y
    if x == y: return  # already in same set
    if r[x] > r[y]:
        x, y = y, x
    # at this point x has the smallest rank
    p[x] = y
    if r[x] == r[y]:
        r[y] += 1
```

It can be shown that with union by rank and path compression, all operations take amortized runtime equal to the inverse Ackerman function (something that grows really really slowly; it's essentially constant). We won't show the proof here, but we can show that any operation takes amortized $O(\log^* n)$ time, which is also essentially constant.

More specifically, any sequence of $\leq n$ finds and $\leq m$ unions takes amortized $O((n + m) \log^* n)$ time.

Proof. We'll first state some claims that we'll use in the proof.

Claim 1: any root node x has $\geq 2^{r[x]}$ elements in its tree.

Claim 2: ranks strictly increase as you follow parent pointers.

Claim 3: the number of nodes with rank exactly k is $\leq \frac{n}{2^k}$.

Look at our n items at any point in time; they all have some ranks from 0 to n. We'll now partition these items by rank; that is, we have sets with rank $[0,2^0)$, $[2^0,2^1)$, $[2,2^2)$, $[2^2,2^{2^2})$, etc.

How many groups are there? The number of groups is $\log^*$ of the largest possible rank (as we end when $2^{2^{\cdot^{\cdot}}} \le n$), or $\log^*(\log_2 n) = \log^* n - 1 \approx \log^* n$.

We know that `makeset` and `union` are both constant time (disregarding the use of `find` in `union`), with the exception of `find`, which takes time.

This means that the total time is $O(m+n+\#$ parent pointers I ever follow). When we follow a parent pointer from u to v, there are three cases:

1. v is the root of its tree

   This takes $O(1)$ per operation, as we only do this once per `find`.

2. u, v are in different groups (and v is not a root)

   This takes $O(\log^* n)$ per operation, as there are only $\log^* n$ groups; we can only change groups $\log^* n$ times as we go up, and v must have been in a later group than u since ranks strictly increase as we go up.

3. u, v are in the same group (and v is not a root)

   We'll keep these kinds of operations for later to figure out the total amortized time.

If we total up the "debt" from the third case, we have

$$\text{total debt} = \sum_{u=1}^{n} \#\text{ times } u \text{ was charged} = \sum_{u=1}^{n} \sum_{t=0}^{\texttt{g[u]}} \#\text{ times } u \text{ was charged while in group } t.$$

Here, we let `g[u]` is the final group that u was in at the end.

Suppose u was in group $[k,2^k)$. Each time u is charged because of case 3, its parent pointer continuously moves over to the right. This means that u can be charged in case 3 while it's still in this group at most $2^k - k \approx 2^k$ times.

This means that all charges combined must be $\le 2^k + 2^{k-1} + 2^{k-2} + \cdots = O(2^k)$. This means that the final sum of the total debt is $2 \cdot \sum_{u=1}^{n} 2^{k_u}$, where u's final group is $[k_u, 2^{k_u})$.

If we group the items by group, this is equal to $2 \cdot \sum_{\text{groups}} 2^k \cdot (\#$ items in group).

Here is where we use claim 3; the number of items in group $[k,2^k)$ is at most $\sum_{j=k}^{2^k-1} \frac{n}{2^j}$, and since this is a decreasing geometric series dominated by the first term, this is $\le 2 \cdot \frac{n}{2^k}$.

Hence, the final summation is equal to $4\sum_{\text{groups}} 2^k \cdot \frac{n}{2^k} = 4n \cdot (\#$ groups), which is $O(n \log^* n)$. □

10.2 Amortized Analysis

Suppose a data structure supports operations $O_1, O_2, \ldots, O_k$. Then we say the amortized cost of each operation is t_j if for any sequence of operations with N_1 of O_1 ops, N_2 of O_2 ops, then the total time is $\le \sum_{j=1}^{k} t_j N_j$. That is, if we look at the aggregate instead of individual operations, we could have a different *average* runtime over all operations.

9/30/2021

Lecture 11

Dynamic Programming I: Introduction, Single Source Shortest Paths

11.1 Fibonacci DP

Dynamic programming is really just recursion but with a lookup table, and we can improve it more by reversing it by going bottom-up.

The top-down version (memoization) is essentially just recursion with a lookup table, so that we don't have to recompute $f(\cdot)$ multiple times on the same arguments. The bottom-up version just fills up the lookup table iteratively instead of recursively.

Here's an example with Fibonacci. The naive version is

```
def fib(n):
    if n <= 1: return n
    else: return fib(n-1) + fib(n-2)
```

What's a memoized version of this implementation?

```
def fibFast(n):
    mem = [None] * (n + 1)
    return fibMemo(n, mem)

def fibMemo(n, mem):
    if n <= 1:
        return n
    elif mem[n] != None:
        return mem[n]
    else:
        mem[n] = fibMemo(n-1, mem) + fibMemo(n-2, mem)
        return mem[n]
```

Here, we utilize a lookup table `mem`, and in each recursive call, we first check if we've already computed `fib(n)`; if so, we don't need to do any more computation, and we just return the saved result. Otherwise, we recurse and fill up the lookup table for later, then return the computed result.

We can immediately see that this drastically improves the runtime of the algorithm. The runtime of `fibMemo` is just the number of possible inputs, multiplied by the time it takes per input. We have n possible inputs, with constant time per input, so this is $O(n)$ runtime. In contrast, `fib(n)` is some exponential runtime.

We can convert this into a bottom-up approach by filling up the lookup table iteratively:

```
def fibBottomUp(n):
    mem = [None] * (n+1)
    mem[0] = 0
    mem[1] = 1
    for i in range(2, n+1):
        mem[i] = mem[i-1] + mem[i-2]
    return mem[n]
```

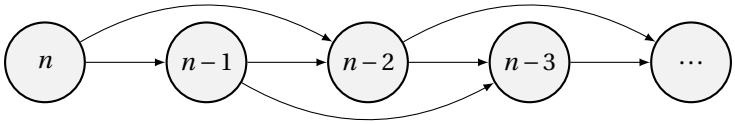
This is also fast, and take a linear number of flops. However, we can generally see that a bottom-up approach can save memory. In both of these current approaches, we need to store all n numbers, but we can save space because we really only need the last two entries of the table.

Doing this space optimization, we now have

```
def fibBottomUpSpaceSaving(n):
    mem = [0, 1]  # stores the mem[i-1] and mem[i-2]
    for i in range(2, n+1):
        x = mem[0] + mem[1]
        mem[0] = mem[1]
        mem[1] = x
    return mem[1]
```

Now, we've reduced the space complexity to $O(1)$; we only need space for two items.

If we look at the recursion tree of naive `fib(n)`, we have this graph dependency structure:



We can always say that this graph is acyclic (if it had a cycle, then we'd have an infinite loop). If you go from top-down to bottom-up, then we'd just topological sort this DAG, and fill in the table in reverse topological order. This is because if we have $u \to v$, then we must fill in v before we fill in u, because u depends on v.

11.2 DP for Single Source Shortest Paths

Speaking of DAGs and topological sort, let's do another DP problem: SSSP on a DAG. Remember that we've done this before with Dijkstra's, but this new DP algorithm will still work if the edge weights are negative.

The first thing that you should generally do is forget about DP; how can we solve this problem recursively? The first step is to think about a function $f(\cdot)$ that can be computed recursively such that I can extract the answer by looking at $f(x)$ for some x.

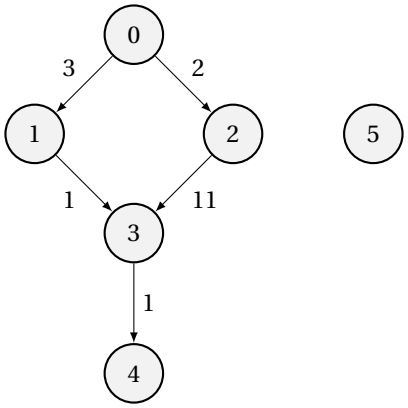
In our case, we have $f(t) :=$ length of shortest path from s to t, where s is our source. We want to know $f(t)$ for all t.

What is the recurrence relation for f?

$$f(t) = \begin{cases} 0 & \text{if } t = s \\ \infty & \text{if } \deg_{in}(t) = 0 \\ \min\limits_{a:(a,t)\in E} \{w(a,t) + f(a)\} & \text{otherwise} \end{cases}.$$

Here, the last recursive case will iterate through all possible edges from t and recurse on the other vertex a. This is just brute force.

Suppose we have the following graph:



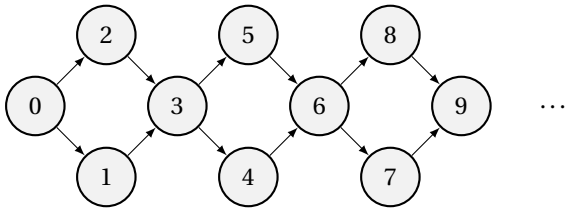
Let's represent this graph as an adjacency list (in Python, a list of lists, where an edge is a list of size 2). If we think about it, here we want to reverse the graph, because we want all of the edges that go into some vertex t.

We can code this naive recursive implementation in Python:

```
myG = [
    [[1, 3], [2, 1]],  # vertex 0, of form [target, weight]
    [[3, 1]],  # vertex 1
    [[3, 11]],  # vertex 2
    [[4, 1]],  # vertex 3
    [],  # vertex 4
    []  # vertex 5
```

```
]

def revGraph(G):
    revG = [[] for _ in range(len(G))]
    for i, edges in enumerate(G):  # iterate through vertices i, get their edges
        for edge in edges:
            v, weight = edge
            revG[v].append([i, weight])
    return revG

def shortestPathDAG(G, s, t):
    return spDAGHelper(revGraph(G), s, t)

def spDAGHelper(revG, s, t):
    if s == t:
        return 0
    elif len(revG[t]) == 0:
        return float('inf')
    else:
        best = float('inf')
        for e in revG[t]:
            v = e[0]
            weight = e[1]
            best=  min(best, sPDAGHelper(revG, s, v) + weight)
        return best
```

How fast or slow his this naive implementation? What if we had a graph like this?



The naive implementation will go through all possible paths, which in this graph of many diamonds will be something around $2^{\frac{n}{3}}$, which is exponential.

We can now look at the runtime of this algorithm if we added memoization. This runtime is

$$\sum_{\text{all inputs}} (\text{time spent on input}) \leq \sum_{u=1}^{n} C \cdot (1 + \deg_{in}(u)) = \Theta(m+n).$$

Here, for each possible input, we look through all of the edges pointing to the vertex (i.e. $\deg_{in}(u)$ other vertices). This means that we go through each vertex once and each edge once—memoization makes the algorithm runtime linear.

10/7/2021

Lecture 12

Dynamic Programming II: DP Examples

12.1 Recovering the Optimal Solution

In the previous lecture, we've shown how to compute the *cost* of the optimal solution, but not the optimal solution itself. We'll talk a little bit about how we can actually compute the optimal solution from the DP.

Here's the memoized code for the example from last time:

```python
def shortestPathDAGFast(G, s, t):
    mem = [None] * len(G)
    return spDAGHelperFast(revGraph(G), s, t, mem)

def spDAGHelperFast(revG, s, t, mem):
    if s == t:
        return 0
    elif len(revG[t]) == 0:
        return float('inf')
    elif mem[t] != None:
        return mem[t]
    else:
        mem[t] = float('inf')
        for edge in revG[t]:  # iterate through all edges going into t
            v, weight = edge
            mem[t] = min(
                mem[t],
                spDAGHelperFast(revG, s, v, mem) + weight
            )
        return mem[t]
```

Here, notice that the only thing that is changing between calls of this function is t. This means that we only need to memoize t.

How can we modify this so that we can return the optimal path? As with any DP problem discussed last time, we can model the problem as a DAG, where $x \to y$ if $f(x)$ depends on $f(y)$, and all we're doing is finding the shortest path in this DAG (coincidentally this example problem *is* finding the shortest path in a DAG, but any optimization problem can be modeled this way).

This means that all we need to keep track of is which path we took that was actually the minimum. This is similar to how we had a `prev[v]` array in BFS or Dijkstra's.

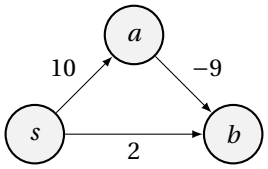
```python
def shortestPathDAGGetPath(G, s, t):
    mem = [None] * len(G)
    choices = [None] * len(G)
    cost = spDAGHelperGetPath(revGraph(G), s, t, mem, choices)

    if cost == float('inf'):  # no possible path from s to t
        return []

    # compute the actual path
    path = [t]
    at = t
    while at != s:
        at = choices[at]  # go backward in choices
    return path[::-1]  # we constructed the path in reverse, so return the reversed path

def spDAGHelperGetPath(revG, s, t, mem, choices):
    if s == t:
        return 0
    elif len(revG[t]) == 0:
        return float('inf')
    elif mem[t] != None:
        return mem[t]
    else:
        mem[t] = float('inf')
        for edge in revG[t]:  # iterate through all edges going into t
            v, weight = edge
```

```
            x = spDAGHelperGetPath(revG, s, v, mem, choices) + weight
            if x < mem[t]:  # we have a better solution
                choices[t] = v  # we got to t by going through v
                mem[t] = x
        return mem[t]
```

12.2 Bellman-Ford

The Bellman-Ford algorithm calculates SSSP on directed graphs. We already know Dijkstra's algorithm for this, but Dijkstra's does not work with negative edge weights.

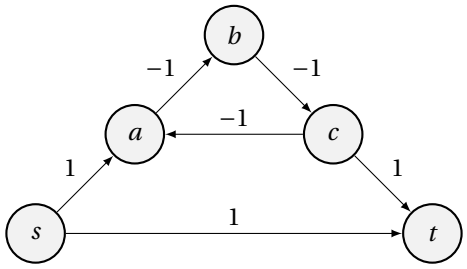
For example, suppose we have this graph.



Dijkstra's will take s as the starting vertex, call `decKey` on a and b, setting the cost to 10 for a and 2 for b. Next, it'll pop b off the heap, thinking that 2 is the shortest path to b. Seeing that there are no outgoing edges, it'll do nothing else, pop a off the heap, and see that b has already been processed, and halt.

Notice that this results in an incorrect shortest path to b; Dijkstra's thinks that the shortest path to b is 2 when it's actually 1.

However, we need to be careful; we could have a situation like this:



What's the shortest path from s to t? Notice that we can go $s \to a \to b \to c \to a \to b \to c \to \cdots$, and just keep decreasing in length as we loop around this cycle. This means that the shortest path is $-\infty$. We want to be able to detect this kind of cycle.

That is if the graph has no negative cycles, we should compute all SSSP distances from s to all other t, and if there exists a negative cycle, we flag it and return.

Let's try and derive Bellman-Ford. This algorithm will use a similar algorithm as the one we used for DAGs; there must exist some final edge $u \to v$, so we'll loop over all possible edges going into v. However, we can't use the same exact recurrence we used for DAGs; we could end up with an infinite recursion. This means that we need some way of breaking this.

Suppose we look at graphs with no negative cycles. We know that any shortest path from s to t must have at most $n-1$ edges; any path with more than $n-1$ edges will always have a cycle. We'll use this fact along with the previous observation to come up with an algorithm.

Our function is $f(t,k)$ is the length of the shortest path from s to t using $\leq k$ edges. The final result we want is

$f(t, n-1)$. The recurrence relation is

$$f(t,k) = \begin{cases} \infty & \text{if } k = 0,\ t \neq s \\ 0 & \text{if } k = 0,\ t = s \\ \min\{f(t,k-1), \min\limits_{(v,t)\in E}\{w(v,t) + f(v,k-1)\}\} & \text{otherwise} \end{cases}.$$

Here in the recursive case, we have two choices; we can either not use this current edge, or we can loop through all possible incoming edge and traverse that edge, adding the weight of the traveled edge. The minimum of all of these gives us the optimal cost.

Naively, the memory needed is $O(n^2)$ because we have t and k that vary, each having n possible values. We can reduce the memory required in a bottom-up implementation by observing that $f(\cdot, k)$ only depends on $f(\cdot, k-1)$ values. This means that we only need to store values for two consecutive values of k; we only need $O(n)$ space.

What about the runtime? There are n possibilities for k, and for each value of k, we need $\sum_{t\in V} C \cdot (1 + \deg_{in}(t))$. The sum over all in-degrees is just the number of edges, so in total we need $O(n^2 + mn)$ time.

There's still a little bit we can do to improve this algorithm.

Bottom up with space saving:

```
def BF(G, s):
    # T[t][0] = previous, T[t][1] = current
    T[1..n][0..1] = all inf
    T[s][0] = 0
    for k in range(1, n):  # from 1 to n-1
        for t in range(1, n+1):  # from 1 to n
            T[t][1] = T[t][0]
            for (v, t) in E:
                T[t][1] = min(T[t][1], T[v] +
                ↪ w(v, t))
        for t in range(1, n+1):  # from 1 to n
            T[t][0] = T[t][1]
    return T[1..n][1]

```

Textbook algorithm:

```
def BFBook(G, s):
    T[1..n] = all inf
    T[s] = 0
    for k in range(1, n):  # from 1 to n-1
        for (u, v) in E:
            T[v] = min(T[v], T[u] + w(u, v))
    return T

```

To understand what the textbook algorithm is doing and how we can get there, let us first establish two lemmas:

Lemma 12.1

At all points in time $\forall v$ `T[v]` is the length of *some* $s \to v$ path.

Lemma 12.2

$\forall k, \forall v$, after going through the outer loop k times, `T[v]` $\leq f(v,k)$.

The proofs are omitted for brevity, but both can be proven through induction. What do these two lemmas tell us? The first lemma tells us that `T[v]` is at least the length of some shortest path, while the second lemma tells us that `T[v]` is at most the length of some shortest path (at $k = n-1$). This means that `T[v]` must be exactly the length of a shortest path.

Lastly, how do we detect whether a negative cycle occurs? We claim that there exists a negative cycle if and only if there exists a vertex v such that $f(v,n) < f(v,n-1)$.

Proof. ($\exists$neg cycle $\impliedby \exists v$) There's no reason to ever use $n-1$ edges because we'd be repeating an edge—the only time we benefit from it is if we have a negative cycle.

($\exists$neg cycle $\implies \exists v$) We prove the contrapositive (that is, $\forall v\, f(v,n) \geq f(v,n-1) \implies \forall$cycle$C,\ w(C) \geq 0$).

Consider some cycle $C = v_1 \to v_2 \to \cdots \to v_r \to v_1$. We know that $\forall i$, $f(v_{i+1}, n-1) \le f(v_{i+1}, n)$ by our assumption.

We know that $f(v_{i+1}, n) \le f(v_i, n-1) + w(v_i, v_{i+1})$. This means that

$$\begin{aligned} f(v_{i+1}, n-1) - f(v_i, n-1) &\le w(v_i, v_{i+1}) \\ \sum_{i=1}^{r} f(v_{i+1}, n-1) - f(v_i, n-1) &\le \sum_{i=1}^{r} w(v_i, v_{i-1}) \\ 0 &\le w(C) \end{aligned}$$

Here, we notice that the LHS sum contains $f(v_i, n-1)$ exactly once added, and exactly once subtracted; these cancel out to give zero. The RHS sum is just the sum of all edges in the cycle, or $w(C)$. □

In other words, after we complete the nested for loop for k, we do an additional iteration of the for loop, and if `T[u]` + $w(u,v)$ < `T[v]`, a negative cycle is detected (this means that we've improved our shortest path with the nth edge). If this never happens, then we return the `T` array.

12.3 Floyd-Warshall

The APSP (all pairs shortest path) problem is as follows. Given a digraph G, we return a 2D array `d` such that for all pairs i, j, `d[i][j]` is the total length of the shortest $i \to j$ path.

A naive way to do this is to run SSSP once for each vertex, i.e. run Bellman-Ford n times. Since Bellman-Ford takes $O(mn)$ time, this naive algorithm would take $O(mn^2)$ time. Since m can be at most n^2, this can potentially take $O(n^4)$ time, which is pretty bad.

We can actually do this in $O(n^3)$ time, using the Floyd-Warshall algorithm. Without loss of generality, suppose G is the complete graph; if $e \notin E_i$, then we just pretend $w(e) = \infty$.

What is our function? Let us define $f(i, j, k)$ be the length of the shortest path from i to j when all intermediate vertices in the path must be in $\{1, \ldots, k\}$. In the end, we want $f(\cdot, \cdot, n)$ values—we don't care which vertices we use, but we want the values for all pairs of i and j.

What's the recurrence relation?

$$f(i,j,k) = \begin{cases} w(i,j) & \text{if } k = 0,\ i \neq j \\ 0 & \text{if } k = 0,\ i = j \\ \min\{f(i,j,k-1), f(i,k,k-1) + f(k,j,k-1)\} & \text{otherwise} \end{cases}.$$

If $k = 0$, we can only take the edge from i to j. The recursive case has two parts. We can either not use vertex k (and thus just decrease k, or we can use vertex k and split it into two parts (this is because our path must go $i \to k \to j$).

The memory naively would be $O(n^3)$, but with bottom-up, we can reduce this to $O(n^2)$, as $f(\cdot, \cdot, k)$ depends only on $f(\cdot, \cdot, k-1)$; we only need to store two levels of the DP array. We can actually still improve this more and only store one level, if we overwrite ourselves similar to the textbook version of Bellman-Ford.

The runtime of this algorithm is just the number of entries in the DP array, or $O(n^3)$; we have n^3 possible inputs, and for each input we do a constant time operation.

```
def FW(G):
    T[1..n][1..n] = all inf
    for (i, j) in G:
        T[i][j] = w(i, j)
    for k in range(1, n+1):  # from 1 to n
        for i in range(1, n+1):
            for j in range(1, n+1):
                T[i][j] = min(T[i][j], T[i][k] + T[k][j])
    return T
```

12.4 Longest Increasing Subsequence

We're given an input array of numbers, and we want to find a subsequence of this array that is strictly increasing, and we want one that is as long as possible.

For example, if we have `A = [2, 8, 3, 4]`, a greedy approach would be to take the next possible item. Here, we'd end up with `[2, 8]` and not be able to have anything else. The actual optimal solution is `[2, 3, 4]`.

Let's first look at a brute force algorithm for this problem. We can define $f(last, i)$ to be the length of the longest increasing subsequence of `A[i..n]` such that all values we use are strictly greater than `A[last]`. In the end, we want $\max_{1 \le i \le n} \big(f(i, i+1)+1\big)$.

Alternatively, we can first modify `A = [-∞] + A`, and pretend that we're always taking $-\infty$. This means that we'd want $f(1,2)$ as our final result (1-indexed); we've taken the first element, and want to use elements afterward (the original elements of `A`.

The recurrence relation here is

$$f(last, i) = \begin{cases} 0 & \text{if } i = n+1 \\ f(last, i+1) & \text{if } \texttt{A[i]} \le \texttt{A[last]} \\ \max\{f(last, i+1), 1 + f(i, i+1))\} & \text{otherwise} \end{cases}.$$

The naive space complexity is $O(n^2)$, but noticing that each k value depends only on $k-1$, we can reduce this to $O(n)$ space. The runtime is $O(n^2)$, as we have n^2 possible values for our two inputs.

10/12/2021

Lecture 13

Linear Programming

13.1 More DP Examples

13.1.1 Knapsack

We're given an input as an array `A[1..n]` of items, where each item is a pair of (weight, value). We also have a knapsack that can hold at most W weight. What is the maximum amount of value we can store in our knapsack?

Greedy: Suppose `A' = [(11, 15), (10, 10), (10, 10)]`, with $W = 20$.

The natural algorithm is to compute the ratio $\frac{v_i}{w_i}$ for item i, and take the items with the most ratio.

Here, notice that $\frac{15}{11}$ is the largest ratio, but doesn't allow for any other items. The alternative of taking both (10, 10) items perfectly fits within the knapsack. It can be shown that the greedy algorithm always does better than half of OPT.

DP: First, let's think about a recursive brute force approach.

Suppose we define $f(i, C)$ as the maximum value we can pack amongst `A[i..n]` with a knapsack capacity C. We want $f(1, W)$. The recurrence relation here is

$$f(i, C) = \begin{cases} f(i+1, C) & \texttt{w[i]} > C \\ \max\{f(i+1, C), \texttt{v[i]} + f(i+1, C - \texttt{w[i]})\} & \texttt{w[i]} \le C \\ 0 & i = n+1 \end{cases}.$$

What's the runtime? The initial call is $f(1, W)$. i can be any number from 1 to n, C can be any number from 0 to W. It takes constant time to calculate the maximum, so we have $\Theta(nW)$. Naively with memoized recursion, we need memory $\Theta(nW)$. With bottom-up space optimization, we only need $O(W)$ memory (we only need to keep the previous two rows, i.e. for $i-1$ and i).

This is a "pseudopolynomial" time algorithm. A polynomial-time algorithm means that the runtime is polynomial in the input size in bits. The number of bits it takes to write down an input to knapsack is roughly $O(n \log W)$, as we have n items, each of at most size W. This means a polynomial time algorithm is polynomial in n and $\log W$. However, our actual runtime is $O(nW)$, which is exponential with respect to $\log W$.

13.1.2 Traveling Salesman Problem

There are n locations with distances `D[i][j]` as the distance from i to j. Here, we're assuming that the distances are in a metric space; i.e. nonnegative, satisfies the triangle inequality, etc.

We want to visit all locations starting at location 1, while minimizing the total distance traveled.

The naive brute force solution is to try all $n!$ orderings, and taking linear time to calculate the cost for each ordering. This means that naive brute force takes $O(n!)$ time.

DP: Let $f(i, S)$ be the minimum travel time to visit all locations in $S \subset \{1, 2, \ldots, n\}$ when starting at location i. We want $f(1, \{2, \ldots, n-1\})$. We can already tell that this is exponential time, as there are a total of 2^n subsets. This is still better than the $O(n!) \sim O(n^n)$ naive brute force solution.

The recurrence relation is

$$f(i, S) = \begin{cases} 0 & S = \varnothing \\ \min\limits_{x \in S} f(x, S \setminus \{x\}) & \text{otherwise} \end{cases}.$$

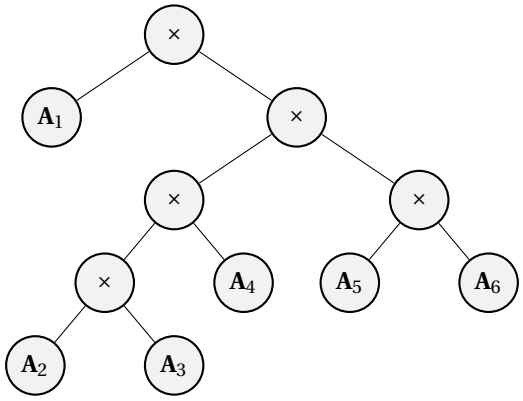
What is the runtime? We have 2^n choices for S, and n choices for i; there are $n2^n$ states. For each state, we need a linear loop over all states in S, so we have $O(n^2 2^n)$ runtime, with $O(n \cdot 2^n)$ space. We can in fact do better in memory with bottom-up DP, as sets of size k only depend on sets of size $k+1$; this means we only need $\binom{n}{k}$ space for each i, so we have $O(\sqrt{n} \cdot 2^n)$ space (using Stirling approximations and the fact that $\binom{n}{k}$ is maximized at $k = \frac{n}{2}$).

13.1.3 Matrix Chain Multiplication

Suppose we have a sequence of matrices we want to multiply. Given $s_1, s_2, \ldots, s_{n+1}$, where $\mathbf{A}_i$ is $s_i \times s_{i+1}$, we want to find the minimum number of f lops to compute $\mathbf{A}_1 \times \mathbf{A}_2 \times \cdots \times \mathbf{A}_n$.

For example, suppose we have $\mathbf{A}$ (3×3), $\mathbf{B}$ (3×2), and $\mathbf{C}$ (2×1). Computing $(\mathbf{AB})(\mathbf{C})$ takes $(3 \cdot 3 \cdot 2) + (3 \cdot 2 \cdot 1) = 24$ flops, as $\mathbf{AB}$ is now 3×2. Computing $(\mathbf{A})(\mathbf{BC})$ takes $(3 \cdot 2 \cdot 1) + (3 \cdot 3 \cdot 1) = 15$ flops.

In any full parenthesization, there is always a last multiplication we do, multiplying two groups of matrices together.



Here, our state will be $f(i, j)$, the optimal time it takes to multiply matrices $\mathbf{A}_i$ through $\mathbf{A}_j$, and we want $f(1, n)$. If we want to multiply matrices in some interval, we have a choice of what the last multiplication is (out of the total $j - i$ multiplications), i.e. how we split up this interval into two parts.

The recurrence relation is

$$f(i, j) = \begin{cases} 0 & i = j \\ \min\limits_{i \geq k \leq j-1} \left\{ f(i, k) + f(k+1, j) + s_i s_{k+1} s_{j+1} \right\} & \text{otherwise} \end{cases}.$$

Here, the recurrence has three parts; the time to recurse on the left, the time to recurse on the right, and the time to do the last multiplication.

What is the runtime? There are n^2 possible different arguments, with a linear loop at each state, meaning we have $O(n^3)$ runtime. The memory usage is $O(n^2)$, one for each state (there isn't any (clear) way to optimize memory here).

13.2 Linear Programming

With linear programming, we're trying to find the solution of the optimization problem in the form

$$\max \vec{c}^T \vec{x} : \mathbf{A}\vec{x} \le \vec{b}.$$

Here, the constraint inequality is entry-wise inequality.

To make this concrete, here's an example:

Example 13.1

I run a factory that sells foo for \$4/oz, bar for \$5/oz.

In order to make 1 oz of foo, it takes 2 oz water, 1 oz butter, and 4 oz goo.

In order to make 1 oz of bar, it takes no water, 3 oz butter, and 2 oz goo.

There are $W = 10$ oz water left, $B = 6$ oz butter left, $G = 6$ oz goo left.

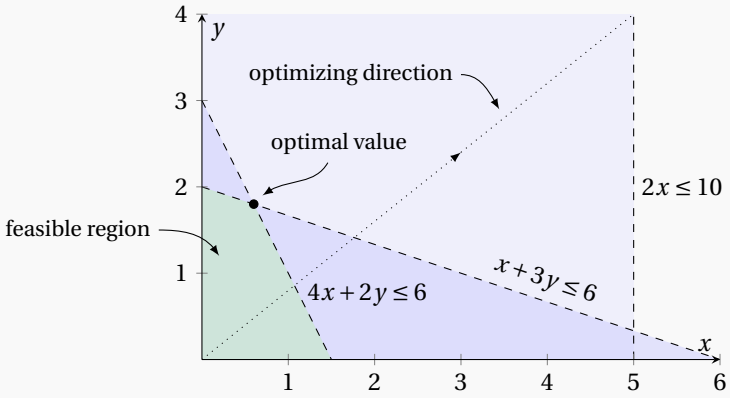
We want to maximize the amount of profit we make. If x and y denote the amount of foo and bar respectively, we have

$$\max 4x+5y : \begin{cases} 2x+0y \le 10 \\ 1x+3y \le 6 \\ 4x+2y \le 6 \\ -x \le 0 \\ -y \le 0 \end{cases}.$$

Specifically, we have

$$\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \\ 4 & 2 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} 10 \\ 6 \\ 6 \\ 0 \\ 0 \end{bmatrix} \qquad \vec{c} = \begin{bmatrix} 4 \\ 5 \end{bmatrix}.$$

Graphically, we have



Note that we can think of this as being rotated in the direction of the objective function, and letting gravity bring us to a minimum point.

We can see that the feasible region of an LP is always convex; the proof of this fact is omitted here.

Looking at just the first three inequalities (i.e. removing the nonnegative constraint), we can multiply both sides by z_1, z_2, and z_3, where $\vec{z}$ is the vector of z_i's:

$$\begin{aligned} z_1 \cdot 2x &\le z_1 \cdot 10 \\ z_2 \cdot (x + 3y) &\le z_2 \cdot 6 \\ z_3 \cdot (4x + 2y) &\le z_3 \cdot 6 \end{aligned}$$

Adding everything up, we have the single inequality

$$(2z_1 + z_2 + 4z_3) \cdot x + (3z_2 + 2z_3) \cdot y \le \vec{z}^T \vec{b}.$$

Recall that we're trying to maximize $4x + 5y$; as such, suppose we have $2z_1 + z_2 + 4z_3 = 4$ and $3z_2 + 2z_3 = 5$; we have that $\vec{c}^T \vec{x} \le \vec{z}^T \vec{b}$. Here, $\vec{c}^T \vec{x}$ is exactly the thing we want to maximize. This means that we can't ever make more than $\vec{z}^T \vec{b}$ dollars.

Notice that this inequality is an upper bound on the revenue. We want to minimize this upper bound:

$$\begin{aligned} \min \quad & \vec{b}^T \vec{z} \\ \text{s.t.} \quad & \mathbf{A}^T \vec{z} = \vec{c}, \\ & \vec{z} \ge \vec{0} \end{aligned}$$

This is the dual LP. The primal LP is the original optimization problem:

$$\begin{aligned} \min \quad & \vec{c}^T \vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} \le \vec{b} \end{aligned}$$

As such, from any primal LP, we can derive a dual LP.

There is also a notion of weak duality; that is, OPT(Primal) ≤ OPT(Dual) by construction.

The notion of strong duality means that if the primal is feasible and bounded (i.e. optimal is not infinity), then OPT(Primal) = OPT(Dual).

10/14/2021

Lecture 14

Network Flow

14.1 LP Algorithms

Finishing up from last time, let's talk a little bit about algorithms for solving LPs.

1. Simplex algorithm

   The simplex algorithm is a greedy algorithm; it starts at a vertex and keeps moving to better vertices (i.e. vertices of the geometric region, walking to neighboring vertices).

   This is a correct algorithm, but it has exponential runtime; there is no known sub-exponential time pivoting rule (i.e. the rule to use to choose the next vertex). However, inputs that force simplex to take exponential time are very fragile and unlikely to occur in real life; in most cases, it'll be quite fast.

   One note here is that even finding a feasible starting point is just as hard as solving the LP; if we could find the starting point easily, then we could essentially bound the optimal value by some constant Q, and use binary search.

The simplex algorithm takes in a minimization problem of the form

$$\begin{aligned} \min \quad & \vec{c}^T \vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} = b, \\ & \vec{x} \geq 0 \end{aligned}$$

and converts it into the equivalent problem

$$\begin{aligned} \min \quad & t \\ \text{s.t.} \quad & \mathbf{A}\vec{x} = (1-t) \cdot b, \\ & \vec{x} \geq 0, \\ & 0 \leq t \leq 1 \end{aligned}$$

It turns out that the starting vertex can be found if we start with $\vec{x} = \vec{0}$, $t = 1$. If simplex can find a solution where $t = 0$ in this equivalent problem, then we use this $\vec{x}$ point as the starting vertex in the original problem, and run simplex again.

One other point to make is that we can turn any inequality constraint into an equality constraint (as in the dual), all we need to do is add a slack variable, turning the inequality into an equality. We can do the reverse by doubling each inequality, i.e. $\mathbf{A}\vec{x} = \vec{b} \implies \mathbf{A}\vec{x} \leq \vec{b} \wedge -\mathbf{A}\vec{x} \leq -\vec{b}$.

2. Ellipsoid Algorithm

   The ellipsoid algorithm was the first polynomial time algorithm for LP; it takes $\text{poly}(m, n) \cdot L$, where m is the number of variables, n is the number of constraints, and L is the precision of the problem. Unfortunately even though this algorithm is polynomial time, it's a large polynomial and always has this large runtime—hence it's not used often.

3. Interior Point

   This is polynomial time but also usable, in contrast to the ellipsoid algorithm.

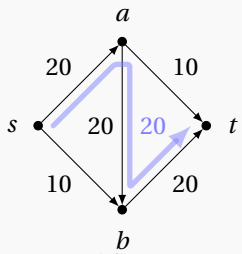
14.2 Network Flow

Imagine you have a pipe network, with pipes as edges in the graph. Each pipe has some capacity (ex. it can support some amount of water flowing through the pip), there is a source pipe, and a sink pipe.

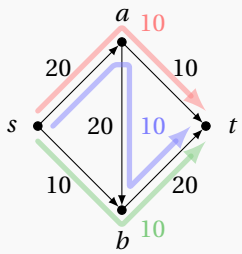
That is, the input is a capacitated directed graph with a source and sink vertex. We want to find the maximum rate of water flow from the source s to the sink t.

There are different families of flow problems, and this problem in particular is called s-t max flow.

Example 14.1



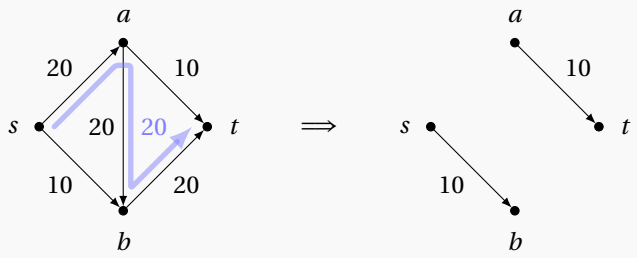
Total flow: 20



Total flow: 30

Algorithm 1: A natural algorithm to try is a greedy algorithm; we keep finding s-t paths iteratively, keeping track of the bottleneck edge. We then push the flow amount that saturates the bottleneck edge. At this point, we remove the bottleneck edge from the graph, subtracting the flow from the other edges, and continue.

Example 14.2



The issue is that this algorithm isn't correct. It'll find a flow, but it might not be optimal (i.e. it might not be the max flow).

Algorithm 2: The first pseudo-polynomial algorithm for max-flow was constructed by Ford-Fulkerson in the 1950s.

Before we talk about the details of the algorithm, we can first see that the max-flow problem is in fact an LP problem.

Each flow f is just a vector in $\mathbb{R}^m$ (i.e. one for each edge), where $f_e \geq 0$ for each edge e. There is also a capacity constraint $f_e \leq u_e$. Further, at each vertex, any water that goes in must go out, i.e. conservation of flow: $\sum_{e=(v,\cdot)} f_e - \sum_{e=(\cdot,v)} f_e = 0$.

Max flow is just maximizing $\sum_{e=(s,\cdot)} f_e$ such that $\vec{f}$ is a flow. That is,

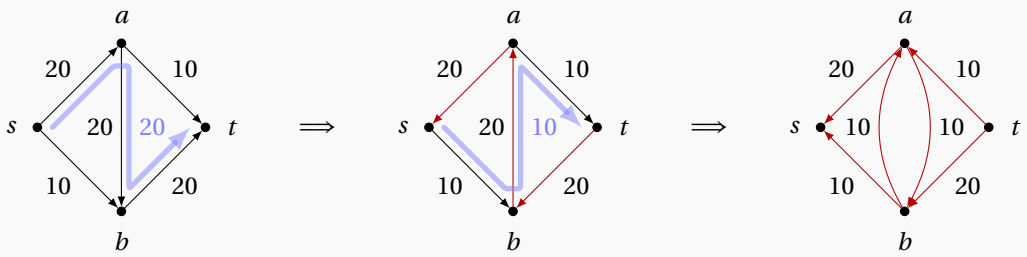
$$\begin{aligned}
\min \quad & \sum_{e=(s,\cdot)} f_e \\
\text{s.t.} \quad & \vec{f} \geq \vec{0}, && \text{(nonnegativity)}, \\
& \vec{f} \leq \vec{u}, && \text{(capacity)}, \\
& 0 = \sum_{e=(v,\cdot)} f_e - \sum_{e=(\cdot,v)} f_e, && \text{(conservation of flow)}
\end{aligned}$$

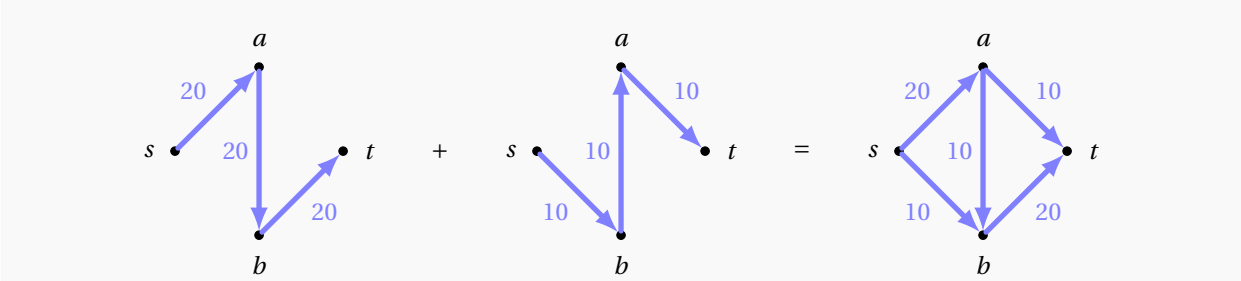
With Ford-Fulkerson, we will pretend that for each edge e, the reverse edge $\text{rev}(e)$ is also in the graph, with capacity 0. Then, if there is f_e flow on e, we will pretend that there is $-f_e$ flow on the reverse edge. Together, this allows us to reverse things we've done in the past.

As an example, in the previous example, instead of disconnecting the graph completely, there exist reverse edges with remaining flow of 20. If we had routed more flow down the reverse edges, we're essentially just reducing the amount of flow we had prior. We call the s-t paths in these residual graphs the "augmenting paths".

Hence, Ford-Fulkerson is just the greedy algorithm, but with these additional rules; we keep finding augmenting paths until we can't anymore.

Example 14.3: Running Ford-Fulkerson





What is the runtime of this algorithm?

In Ford Fulkerson's original paper in the '50s, they gave an example of a graph with 10 vertices and 48 edges that does not terminate. Zwick in the '90s gave an example with 6 vertices and 8 edges, and proved that this is the smallest example where Ford Fulkerson does not terminate.

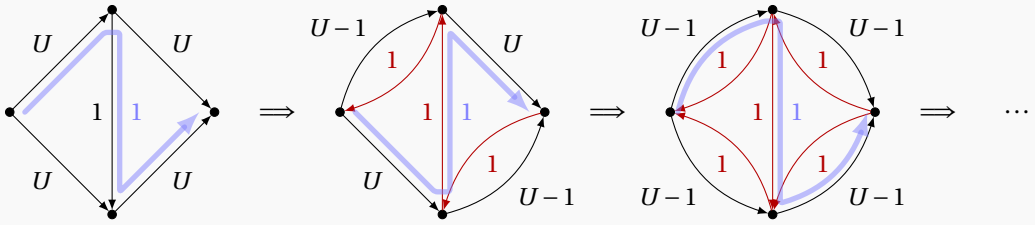
What's the catch? Both examples made use of irrational capacities. This means that if we want to prove that Ford Fulkerson terminates, we need to make use of the fact that the capacities are rational (and without loss of generality, we can assume they're integers).

If we know that all capacities are integers, all residual graphs will always have integer capacities (as bottleneck edges all have integer capacities). We further know that every single iteration, the bottleneck edge will have capacity at least one, so we must push at least one unit of flow every iteration. This means that the maximum number of iterations we need is at most the value of the max flow.

The runtime is thus $O((m+n)\cdot \text{val}(f^*))$, where $\text{val}(f^*)$ is the value of the maximum flow. Put another way, the largest that the flow could ever be is the total capacity leaving the source; if the edge with largest capacity leaving the source is U, then the flow out of the source is bounded by $(n-1)U$, making the runtime $O((m+n)nU)$.

Example 14.4

Here is an example where Ford Fulkerson could take a very long time to terminate:



Here, each iteration we remove one unit from the outer edges, so we'd be taking some multiple of U iterations to terminate.

One of the first improvements was by Edmonds-Karp in 1972 and Dinic in 1970 (independently), showing that Ford Fulkerson with BFS takes time $O(m^2 n)$. Dinic gave an entirely new algorithm with time $O(n^2 m)$ (this is better, as $m > n$).

Goldberg-Rao in 1998 (the Rao is Satish Rao) gave an algorithm that ran in time $O(m \cdot \min\left(\sqrt{m}, n^{\frac{2}{3}}\right) \cdot \log\left(\frac{n^2}{m}\right) \cdot \log U)$. This was the best algorithm for around 16 years, before Lee-Sidford in 2014 gave an algorithm that ran in $O(m\sqrt{n}\log U)$ time, which was very similar to the interior point algorithm.

With pseudopolynomial time algorithms, Kathuria-Liu-Sidfurd in 2020 gave an algorithm with time $O(m^{\frac{4}{3}} \cdot U^{\frac{1}{3}})$; this is pseudo-polynomial, but with graphs with small capacity, this may be preferred.

This isn't a complete history, and only highlights some of the algorithms discovered.

10/19/2021

Lecture 15

Max-flow Min-cut and Duality

15.1 Ford-Fulkerson Correctness

Lemma 15.1

Any $s \to t$ flow decomposes into at most m cycles and $s \to t$ paths.

That is, if given the vector of flows for each edge, we can extract at most m cycles and $s \to t$ paths.

Proof. We proceed by induction on m. The base case $m = 1$ has one edge $s \to t$, and the flow is a path consisting of a single edge from s to t.

We assume the claim holds for $m-1$ edges, and show the case for m.

Consider an s-t flow f. We will define a new graph H_f with the same vertices, but with only edges with strictly positive flow according to f (with value equal to the flows). What if we do a DFS from s in H_f?

If we reach t, then there exists an s-t path in H_f. We will then extract a path (i.e. delete it from H_f) with value equal to the minimum flow value along the path. This means that we have a flow with at most $m-1$ edges, and the inductive hypothesis says that the remaining flow can be decomposed into at most $m-1$ paths and cycles, and the original graph can be decomposed into at most m paths and cycles.

If we find a cycle, and we extract the cycle (i.e. delete it from H_f), and apply the inductive hypothesis. This gives us the same result, leaving a flow on at most $m-1$ edges, and the original graph can be decomposed into at most m paths and cycles.

There is a third case, where we get stuck at a vertex $v \neq t$. This is impossible due to conservation of flow—if we've entered a vertex, there must have been a flow into the vertex, and thus by conservation of flow, there must be a flow out of the vertex. The only way to get stuck is if there is no flow out of a vertex, which is impossible if we've entered it. □

Lemma 15.2

Suppose f is a flow in G and it is not the max flow. Then for any max flow f^*, $f^* - f$ is a feasible flow in the residual graph G_f.

Proof. The proof is omitted here, but is essentially a matter of checking the constraints of non-negativity, capacity constraints, and conservation of flow in the residual graph. □

Theorem 15.3: Max-flow Min-cut Theorem

Suppose we look at the value of a max-flow $\text{val}(f^*)$. This value is equal to the capacity of the min-cut, $u(S^*)$. Here, f^* is the maximum s-t flow, and S^* is a minimum s-t cut, where $\text{val}(\cdot)$ is the value of a flow, and $u(\cdot)$ is the capacity of a cut/edge.

Definition 15.4: Cut

A *cut* of a graph G is a partition of V into two non-empty sets S and T.

Definition 15.5: $s \to t$ Cut

An $s \to t$ cut is a cut where $s \in S$, and $t \in T$.

Definition 15.6: Capacity of a cut

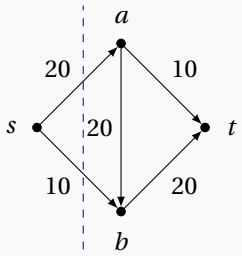
The *capacity of a cut* $u(S)$ is defined as

$$u(S) \coloneqq \sum_{\substack{e \in E \\ e \in S \times T}} u_e.$$

That is, we take the sum of all edges crossing the cut.

Example 15.7

Suppose we have the following graph, and the cut $S = \{s\}$, $T = \{a, b, t\}$.



The edges across the cut have capacities 20 and 10, meaning $u(S) = 30$.

Recall when we ran Ford-Fulkerson on the same graph before (Example 14.3); the max-flow was also 30.

Further, we will show that the min-cut can actually be extracted from the final residual graph;

$$S = \{v : s \text{ can reach } v\}.$$

This is always a valid s-t cut, because the final flow will never have a valid flow from s to t in the residual graph (otherwise we'd have added it to the flow).

Definition 15.8

Given an s-t cut S, we define the *net flow* $f(S)$ by

$$f(S) = \sum_{a \in S, b \in T} f_{(a,b)} - f_{b,a)}.$$

That is, we add the flows going from S to T, and subtract the flows going from T to S.

Recall that we defined $\text{val}(f) = f(\{s\})$. A key observation is as follows:

Lemma 15.9

For all s-t cuts S, $f(S) = \text{val}(f)$. (Here, f in $f(S)$ is a function, and f in $\text{val}(f)$ is a flow.)

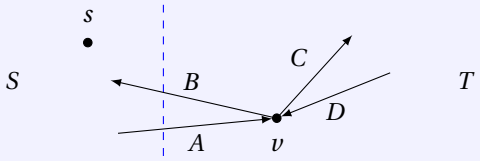
Proof. We proceed by induction on $|S|$.

The base case is where $|S| = 1$, and $S = \{s\}$. The net flow $f(\{s\}) = \text{f}$) by definition of $\text{val}(f)$.

In the inductive step, we now have $|S| > 1$. This means that there exists at least two vertices s and v in S. If we move v to T, then the resulting cut (by the inductive hypothesis) is equal to the value of the flow; further, we will show that if we move v back, then the value of the flow stays the same, and the net flow of S still stays the same.

If we look at the edges connected to v, we have four cases:

- Edges from S into v, with net flow A
- Edges from v into S, with net flow B
- Edges from v into T, with net flow C
- Edges from T into v, with net flow D



We know that by conservation of flow,

$$\underbrace{A+D}_{\text{flow in}} = \underbrace{B+C}_{\text{flow out}} \implies A - B = C - D.$$

Further, if we look at $f(S \setminus \{v\}) = \text{stuff} + A - B$, where "stuff" here is all the flow that does not involve v. Similarly, $f(S) = \text{stuff} + C - D$.

This means that the two flows must be equal; "stuff" is the same between the two expressions, and the leftover is also the same by conservation of flow. □

Now we are ready to prove the max-flow min-cut theorem (Theorem 15.3).

Proof. We will show two things; that $\text{val}(f^*) \le u(S^*)$, and that $\text{val}(f^*) \ge u(S^*)$.

Firstly, we will show that $\text{val}(f^*) \le u(S^*)$. We will show that for all f and S, $\text{val}(f) \le u(S)$; the claim will follow, as it's just a special case.

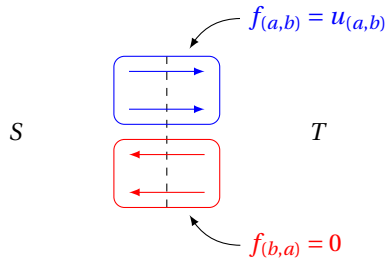
Intuitively, we know that any flow has a bottleneck edge, and the amount of flow can't possibly be greater than this bottleneck capacity. That is, any flow is bottlenecked by some cut.

Formally, we can first see that $\text{val}(f) = f(S)$ for any S (by Lemma 15.9 that we just proved). This leads to the following inequalities:

$$\begin{aligned}
\text{val}(f) &= f(S) \\
&= \sum_{a \in S, b \in T} \left(f_{(a,b)} - f_{(b,a)}\right) && \text{(def. net flow)} \\
&\le \sum_{a \in S, b \in T} f_{(a,b)} && \text{(flow} \ge 0\text{)} \\
&\le \sum_{a \in S, b \in T} u_{(a,b)} && \text{(capacity constraints)} \\
&= u(S) && \text{(def. cut capacity)}
\end{aligned}$$

Next, we will show that $\text{val}(f^*) \ge u(S^*)$. That is, we will show that there exists a flow f and a cut S such that $\text{val}(f) = u(S)$; since $\text{val}(f^*) \ge \text{val}(f) = u(S) \ge u(S^*)$, our claim follows.

Let f be a max-flow, and define $S = \{v : s \text{ has a path to } v \text{ in } G_f\}$. We claim that the value of this flow f is equal to the capacity of this cut. Below is a diagram of such an s-t cut of the final flow graph.



Here, we can see that if we take the s-t cut as described, there are going to be flows going from S to T and flows from T to S.

The flows from S to T must be exactly equal to their capacities; if not, then there would be some residual capacity leftover on an edge going from S to T, which contradicts the definition of S, as we can now reach a vertex in T from s.

Further, the flows from T to S must be zero; if not, then there would be some residual reverse edge with nonzero capacity, meaning its reverse edge would have made some vertex in T reachable from s.

By the same lemma as before, $\text{val}(f) = f(S)$. With the observations just described, we know that

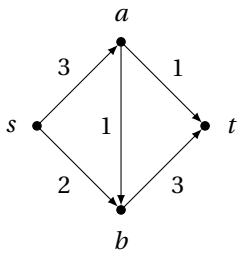
$$\text{val}(f) = f(S) = \sum_{a \in S, b \in T} (f_{(b,a)} - f_{(b,a)}) = \sum_{a \in S, b \in T} u_{(a,b)} - 0 = u(S).$$

□

15.2 Max-flow Min-cut Duality

We've talked before about the LP that the max-flow problem describes. It turns out that the minimum s-t cut is almost the dual of the max-flow LP.

To start, suppose we have the following graph:



Our objective is to maximize $1 \cdot f_{sa} + 1 \cdot f_{sb} + 1 \cdot f_{ab} + 1 \cdot f_{at} + 1 \cdot f_{bt}$, subject to capacity constraints, conservation of flow, and nonnegativity of flows.

That is,

$$\begin{aligned}
\max \quad & f_{sa} + f_{sb} \\
\text{s.t.} \quad & f_{sa} \le 3, && (m \text{ capacity constraints}), \\
& f_{sb} \le 2, \\
& \ldots \le \ldots, \\
& f_{sa} - f_{sb} + f_{at} = 0, && (n-2 \text{ conservation of flow constraints}), \\
& f_{sb} + f_{ab} - f_{bt} = 0, \\
& \vec{f} \ge 0, && (m \text{ nonnegativity constraints})
\end{aligned}$$

Recall that we can convert any primal LP into its dual LP:

Primal LP:

$$\begin{aligned}
\max \quad & \vec{c}^T \vec{x} \\
\text{s.t.} \quad & \mathbf{A}\vec{x} \le \vec{b}
\end{aligned}$$

Dual LP:

$$\begin{aligned}
\min \quad & \vec{b}^T \vec{y} \\
\text{s.t.} \quad & \mathbf{A}^T \vec{y} = \vec{c}, \\
& \vec{y} \ge \vec{0}
\end{aligned}$$

Let us go back to where the dual came from and look at what it would look like here.

Remember that the dual came from multiplying both sides of $\mathbf{A}\vec{x} \le \vec{b}$ by some $\vec{y}$, but we didn't want to flip the inequality (hence the nonnegativity constraint).

That is, we started with a system of constraints $\sum a_{1j}x_j \le b_1$, $\sum a_{2j}x_j \le b_2$, ..., and we multiplied both sides by y_i's to get $y_1\left(\sum a_{1j}x_j\right) \le y_1 b_1$, $y_2\left(\sum a_{2j}x_j\right) \le y_2 b_2$, ...

Adding everything up, we have that

$$x_1 \cdot (a_{11}y_1 + a_{21}y_2 + \cdots) + x_2 \cdot (a_{12}y_1 + a_{22}y_2 + \cdots) + \cdots \le \vec{b}^T \vec{y}.$$

If the ith constraint in the primal LP was actually an equality, then we do not need an $y_i \ge 0$ constraint; that is, we only need nonnegativity constraints for the dual variables y_i corresponding to inequality constraints in the primal LP.

Further, in our max-flow LP, we already had nonnegativity constraints on all the variables. This means that we have $x_i \ge 0$, which turns into $w_i \cdot (-x_i) \le 0$ after negating and multiplying by some dual variable. Note here that w_i's must all be nonnegative, as these constraints are inequalities. Also, note that the w_i's do not participate at all in the RHS.

If we include these constraints and group them together with the x_i's as before, now we have terms of the form

$$x_1 \cdot (a_{11}y_1 + a_{21}y_2 + \cdots - w_1) + x_2 \cdot (a_{12}y_1 + a_{22}y_2 + \cdots - w_2) + \cdots \le \vec{b}^T \vec{y}.$$

Recall that the point of doing this summation was to set each of the inner linear combinations of a_{ij} and y_j to be some constant c_i from the primal objective function (this allows us to minimize $\vec{b}^T \vec{y}$ subject to those constraints to then find the maximum of our primal objective, as $\vec{b}^T \vec{y}$ is an upper bound).

Here, notice that we have an extra free variable w_i for each of these terms, meaning that we now have expressions like $a_{11}y_1 + a_{21}y_2 + \cdots - w_1 = c_1$, which means $a_{11}y_1 + a_{21}y_2 + \cdots \ge c_1$ if we get rid of the w_1.

Intuitively, this means that we don't actually care if the linear combinations of a_{ij}'s and y_j's don't exactly add up to c_j; we can adjust w_j to bring us down to c_j. In terms of the dual LP, this means that we do not actually have a $\mathbf{A}^T \vec{y} = \vec{c}$ constraint; we instead have $\mathbf{A}^T \vec{y} \ge \vec{c}$.

In summary, with the constraints that we have in our max-flow problem, we have two differences. Firstly, equality constraints do not require the associated dual variables to be nonnegative. Secondly, the nonnegativity constraints on our flows (i.e. our $\vec{x}$) means that the $\mathbf{A}^T \vec{y} = \vec{c}$ in the dual becomes the inequality $\mathbf{A}^T \vec{y} \ge \vec{c}$.

How many dual variables do we have? The m capacity constraints give us m dual variables that must be nonnegative. The $n-2$ conservation of flow constraints give us $n-2$ dual variables that are unconstrained (as these are equalities

in the primal). The m nonnegativity constraints do not add any additional dual variables, as they give us our w_i's to implicitly convert the equality in $\mathbf{A}^T \vec{\boldsymbol{y}} = \vec{\boldsymbol{c}}$ into the inequality $\mathbf{A}^T \vec{\boldsymbol{y}} \geq \vec{\boldsymbol{c}}$.

As an example, we can write out the dual of the example graph from before.

Remember that $\vec{\boldsymbol{b}}$ contains the scalars on the RHS of the constraints, and $\vec{\boldsymbol{y}}$ contains the dual variables; the new objective in the dual multiplies the dual variables with each of the RHS scalars in the respective constraints.

This means that the duals for the capacity constraints are multiplied by their respective capacities, and the duals for the conservation of flow constraints are multiplied by zero. Hence, our dual objective function only consists of our y_i's corresponding to each edge, multiplied by the respective edge's capacity:

$$\min 3 \cdot y_{sa} + 2 \cdot y_{sb} + 1 \cdot y_{ab} + 1 \cdot y_{at} + 3 \cdot y_{bt}.$$

Looking at the constraints, remember that constraints in the primal give us variables in the dual, and variables in the primal give us constraints in the dual. So how many constraints are there? There are m variables in the primal, and as such there are m constraints in the dual.

We can just do the computation and see what these constraints are. We'll also be calling the dual variables corresponding to the conservation of flow constraints z_i's, for vertex i.

$$\begin{aligned}
\min \quad & 3 \cdot y_{sa} + 2 \cdot y_{sb} + 1 \cdot y_{ab} + 1 \cdot y_{at} + 3 \cdot y_{bt} \\
\text{s.t.} \quad & y_{sa} + z_a \geq 1, && \text{(edges leaving } s\text{)}, \\
& y_{sb} + z_b \geq 1, \\
& y_{ab} - z_a + z_b \geq 0, && \text{(edges not leaving } s\text{)}, \\
& \dots \geq \dots, \\
& \vec{\boldsymbol{y}} \geq \vec{\mathbf{0}}, && \text{(nonnegativity)}
\end{aligned}$$

What does all of this mean?

Remember that typically an LP problem is over the reals; all of the variables could take on values in $\mathbb{R}$. However, if we had a promise that all of the variables can only take on integer values, then we claim that this dual is exactly the min-cut problem.

We claim that if S is an s-t cut, then we can get a dual solution out of it (that is, any s-t cut will satisfy the dual constraints). Further, we will see that our dual LP is minimizing over the capacity of the cut, giving us the optimal solution of the min-cut problem.

Here, we define our dual variables accordingly:

$$y_{ab} = \begin{cases} 1 & \text{if } a \in S, b \in T \\ 0 & \text{otherwise} \end{cases} \qquad z_a = \begin{cases} 1 & \text{if } a \in S \\ 0 & \text{otherwise} \end{cases}$$

We can see that the objective makes sense; the nonzero terms in the objective function correspond to capacities of edges that cross the cut. Other edges that do not cross the cut are zeroed out because of the definition of the y_i's. As such, we're minimizing over the capacities of the edges that cross the cut—this is the goal of the min-cut problem.

What about the constraints? They follow a similar story.

Suppose we look at the constraints corresponding to the edges leaving s; note that all of these constraints are of the form $y_{sa} + z_a \geq 1$. That is, we have the y variable corresponding to the edge leaving s, and the z variable corresponding to the vertex that we go to.

If (s, a) crossed the cut, then we know that $y_{sa} = 1$ and $z_a = 0$, because a must not be in S (as s is in S). If (s, a) did not cross the cut, then we know that $y_{sa} = 0$ and $z_a = 1$, because both s and a must be on the same side (the S side) of the cut.

Suppose we now look at the constraints corresponding to the edges not leaving s; note that all of these constraints are of the form $y_{ab} - z_a + z_b$. That is, we have the y variable corresponding to the edge between two vertices a and b, and we subtract the z variable corresponding to the vertex a and add the z variable corresponding to the vertex b.

If (a, b) crossed the cut, then we know that $y_{ab} = 1$, and exactly one of z_a and z_b must be 1 (while the other is 0); this is because exactly one of the vertices must be in S, and the other must be in T. This satisfies the inequality, as $-z_a + z_b$ can either be 1 or -1, both of which give a result ≥ 0.

If (a, b) did not cross the cut, then we know that $y_{ab} = 0$, and that $z_a = z_b$; this is because both a and b must be on the same side of the cut (either both in S or both in T). This satisfies the inequality, as $-z_a + z_b$ will always be 0.

From this, we can see that these definitions are consistent with our dual LP formulation, meaning that this dual LP does indeed describe the min-cut problem. This establishes the duality between the max-flow and min-cut problems.

10/21/2021

Lecture 16

Games

16.1 Zero-sum Games

Definition 16.1: Game

In a game, for the sake of CS 170, there are $p > 2$ players, each of which must take one action from a set of possible actions. Once all of the actions are taken, the results are revealed and each player has some utility (score) for each possible outcome.

Example 16.2: Prisoner's Dilemma

There are two criminals that got caught and arrested. The defense attorney is considering two different charges; lesser charges and greater charges.

The DA gives each criminal the same deal: if you stay silent, then you get 1 year. If you snitch on your partner, then you get 0 years and your partner gets 3 years. If both of you snitch, then both of you get 2 years.

If both criminals trusted each other, then both staying silent will result in a sentence of 1 year. However, if we're trying to maximize our own utility, and we know that the other is going to commit in staying silent, then the best course of action is to snitch. The drawback is that the same logic could be applied the other way around, leading both criminals to snitch and get 2 years.

As a game, each player has two choices: snitch or stay silent. An outcome is a cell of the following matrix, each with a utility.

	snitch	silent
snitch	$(-2,-2)$	$(0,-3)$
silent	$(-3,0)$	$(-1,-1)$

.

Here, we assume that we want to maximize utility, so negative utilities are bad. We'll come back to this kind of game later.

Definition 16.3: Zero-sum Game

One special kind of game is a *zero sum game*; that is, games where the payoffs for the two players sum to zero for any outcome.

Example 16.4: Rock, Paper, Scissors

Here is the game of rock, paper, scissors; we'll represent a win as a 1, a tie as 0, and a loss as a -1.

	R	P	S
R	$(0,0)$	$(-1,1)$	$(1,-1)$
P	$(1,-1)$	$(0,0)$	$(-1,1)$
S	$(-1,1)$	$(1,-1)$	$(0,0)$

.

This is the "payoff matrix" of this game, and if we look at any cell in the matrix, the sum of the utilities are zero.

Since these values sum to zero, we can just write the first number in the cell, and the second number is just the negation:

	R	P	S
R	0	-1	1
P	1	0	-1
S	-1	1	0

.

In zero sum games, the row player wants to maximize the score, and the column player wants to minimize the score (as the score is the utility for the row player).

There are a few strategies:

1. *Pure strategies*: always pick some particular action

   For example, one could always play rock in rock, paper, scissors.

2. *Mixed strategies*: a randomized strategy where there is some probability distribution over actions.

   For example, in a mixed strategy, we can have $\vec{x}$ be a vector of probabilities over actions for the row player, and $\vec{y}$ be the same for the column player.

   We could have $\vec{x} = (\frac{1}{5}, \frac{2}{5}, \frac{2}{5})$. In this case, the optimal strategy is $\vec{y} = (0,0,1)$ for the opponent. In this case, the column player wins $\frac{2}{5}$ of the time, loses $\frac{1}{5}$ of the time, and ties $\frac{2}{5}$ of the time.

   Here, the expected payoff is

   $$\mathbb{E}[\text{payoff}] = \frac{2}{5} \cdot (-1) + \frac{2}{5} \cdot 0 + \frac{1}{5} \cdot 1 = -\frac{1}{5}.$$

   Here, we have an expected loss for the row player and an expected win for the column player.

   If we instead had $\vec{x} = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, and kept $\vec{y} = (0,0,1)$, let us calculate the expected payoff:

   $$\mathbb{E}[\text{payoff}] = \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 0 + \frac{1}{3} \cdot (-1) = 0.$$

   What else could the column player do? Suppose we had $\vec{y} = (y_1, y_2, y_3)$. (Here, for ease suppose each action is A_i.)

   $$\begin{aligned}
   \mathbb{E}[\text{payoff}] &= \sum_{j=1}^{3} \mathbb{E}[\text{payoff} \mid A_j] \cdot y_j \\
   &= \sum_{j=1}^{3} \left(y_j \cdot \Big(\underbrace{\sum_{i=1}^{3} \frac{1}{3} \cdot U(i,1)}_{0} \Big) \right) \\
   &= 0
   \end{aligned}$$

   This means that no matter what the column player does, the expected payoff is zero.

We can see that there exists a strategy $\vec{x}$ for the row player which guarantees $\mathbb{E}[\text{payoff}] \geq 0$. Similarly (symmetrically), we can see that there exists a strategy $\vec{y}$ for the column player which guarantees $\mathbb{E}[\text{payoff}] \leq 0$.

If both players play optimally, then the expected payoff will be zero. In this situation, we say that this pair of strategies is in *equilibrium*, and the value of the RPS game is equal to zero.

The main thing that we'll be concerning ourselves with is to look at algorithms which given the payoff matrix, gives us a pair of strategies that is in equilibrium, and finds the value of the game.

Example 16.5

Let's look at a different game:

	L	R
T	5	-3
B	-1	1

.

What if the row player tries $\vec{x} = (\frac{1}{2}, \frac{1}{2})$? Let us look at the pure strategies in response to this.

If the column player commits to play L,

$$\mathbb{E}[\text{payoff}] = 5 \cdot \frac{1}{2} - 1 \cdot \frac{1}{2} = 2.$$

If the column player commits to play R,

$$\mathbb{E}[\text{payoff}] = -3 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = -1.$$

Remember that the column player wants to minimize the payoff; this means that the pure strategy of playing R is best for the column player.

What if the column player plays the uniform strategy $\vec{y} = (\frac{1}{2}, \frac{1}{2})$?

If the row player commits to play T,

$$\mathbb{E}[\text{payoff}] = 5 \cdot \frac{1}{2} - 3 \cdot \frac{1}{2} = 1.$$

If the row player commits to play B,

$$\mathbb{E}[\text{payoff}] = -1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = 0.$$

Since the row player wants to maximize the payoff, this means that the pure strategy of playing T is best for the row player.

However, we can already see that these numbers don't match up; the pure strategy is also probably not the best way.

A better strategy (for now we can let this be magic, but we will be able to tell later how we can get these numbers) is $\vec{x} = (\frac{1}{5}, \frac{4}{5})$.

If the column player commits to play L,

$$\mathbb{E}[\text{payoff}] = 5 \cdot \frac{1}{5} - 1 \cdot \frac{4}{5} = \frac{1}{5}.$$

If the column player commits to play R,

$$\mathbb{E}[\text{payoff}] = -3 \cdot \frac{1}{5} + 1 \cdot \frac{4}{5} = \frac{1}{5}.$$

As such, no matter what the column player does, we can guarantee that $\mathbb{E}[\text{payoff}] \geq \frac{1}{5}$ (specifically, equal to $\frac{1}{5}$).

Similarly, if $\vec{y} = (\frac{2}{5}, \frac{3}{5})$, then we can do the same calculations.

If the row player commits to play T,

$$\mathbb{E}[\text{payoff}] = 5 \cdot \frac{2}{5} - 3 \cdot \frac{3}{5} = \frac{1}{5}.$$

If the row player commits to play B,

$$\mathbb{E}[\text{payoff}] = -1 \cdot \frac{3}{5} + 1 \cdot \frac{2}{5} = -\frac{1}{5}.$$

As such, we can see that no matter what the row player does, we can guarantee that $\mathbb{E}[\text{payoff}] \leq \frac{1}{5}$.

This means that these two strategies are in equilibrium.

16.2 Finding Equilibria of Zero-sum Games

Now let us turn to algorithms; given a game, how can we find these strategies?

Example 16.6

More generally, if $\vec{x} = (x_1, x_2)$, then the pure strategy responses are

$$\begin{aligned} L &\implies 5 \cdot x_1 - 1 \cdot x_2 \\ R &\implies -3 \cdot x_1 + 1 \cdot x_2 \end{aligned}$$

The row player wants (if we know the best response is a pure strategy)

$$\max_{\vec{x}} \min\{5x_1 - x_2, -3x_1 + x_2\}.$$

We will eventually also show that the best response can always be a pure response.

We can convert this problem into an LP. Suppose we have a slack variable z to represent the inner minimum; that is, we ensure that $z \le \min(5x_1 - x_2, -3x_1 + x_2)$ such that we can maximize z to get the minimum.

This means we have

$$\begin{aligned} \max_{x_1, x_2, z} \quad & z \\ \text{s.t.} \quad & 5x_1 - x_2 \ge z, \\ & -3x_1 + x_2 \ge z, \\ & x_1 + x_2 = 1, \\ & x_1 \ge 0, \\ & x_2 \ge 0 \end{aligned}$$

The first two constraints ensure that z is the minimum of the two payoffs, and the last three constraints ensure that $\vec{x}$ is a valid probability distribution.

If we solve this LP, we'd get that $z = \frac{1}{5}$, $x_1 = \frac{1}{5}$, $x_2 = \frac{4}{5}$.

We can similarly write an LP for the column player; if $\vec{y} = (y_1, y_2)$, then the pure strategy responses are

$$\begin{aligned} T &\implies 5 \cdot y_1 - 3 \cdot y_2 \\ B &\implies -1 \cdot y_1 + 1 \cdot y_2 \end{aligned}$$

The column player wants

$$\min_{\vec{y}} \max\{5y_1 - 3y_2, -y_1 + y_2\}.$$

This gives us our LP:

$$\begin{aligned} \min_{x_1, x_2, z} \quad & w \\ \text{s.t.} \quad & 5y_1 - 3y_2 \le w, \\ & -y_1 + y_2 \le w, \\ & y_1 + y_2 = 1, \\ & y_1 \ge 0, \\ & y_2 \ge 0 \end{aligned}$$

We aren't going to go through the entire conversion, but we can see that these two LPs that we've found are actually duals!

It turns out that y_1 and y_2 are the dual variables corresponding to the first two constraints in the row LP, and w is the dual variable corresponding to the third constraint in the row LP. Here, we can see that the third constraint is an equality, meaning the corresponding dual variable w is unconstrained, while the first two constraints are inequalities, meaning the corresponding dual variables y_1 and y_2 are constrained to be nonnegative—this is consistent to what we expect.

Further, we have the nonnegativity constraints on x_1 and x_2, which corresponds to the two inequalities in the dual rather than the typical equalities. The equality constraint in the column LP is the constraint corresponding to the z variable, which has no nonnegativity constraint.

As such, by strong duality we know that the two LPs have equal optimums.

One thing that we skipped over is why a pure strategy will always be best; why can't there be a mixed strategy that does better than any pure strategy?

Lemma 16.7

In response to some strategy $\vec{x}$, the best response is always a pure strategy.

Proof. Suppose we look at general zero-sum games between two players. WLOG, let us look at the row player and what they want. They want to maximize payoff assuming the column player plays optimally in response (i.e. knowing that the column player will minimize payoff in response):

$$\max_{\vec{x}} \min_{\vec{y}} \sum_{j=1}^{m} \left(y_j \left(\sum_{i=1}^{m} x_i U(i,j) \right) \right).$$

(One tidbit to notice here is that the summations turn out to be equal to $\vec{y}^T \mathbf{A} \vec{x}$.)

Notice that the inner summation $\sum_{i=1}^{m} x_i U(i,j)$ depends only on j; we're summing over all i. This means that we can just set this inner sum to be some value Q_j depending only on j (and $\vec{x}$, but within the minimum this is fixed). This substitution gives us

$$\max_{\vec{x}} \min_{\vec{y}} \sum_{j=1}^{m} y_j Q_j.$$

Further, notice that Q_j itself is an expectation; it's the expected payoff with some fixed strategy $\vec{x}$ given that we've already chosen what action to take as the column player (via y_j). That is,

$$Q_j = \mathbb{E}[\text{payoff} \mid \text{column takes action } j].$$

With this in mind, the inner minimization objective is some weighted average of these expected values. To find the minimum value of this weighted average (this is what the column player is trying to do), note that there must be some smallest Q_j among these possibilities. By a greedy argument, we can see that putting all of our probability mass on this smallest payoff expectation gives us the optimal result—putting any probability mass on anything greater would result in some larger weighted average, which is undesirable.

Hence, what we've just found out is that the best response to any strategy $\vec{x}$ is a pure strategy—there could be mixed strategies that do equally well, but one best strategy is always to greedily put all our probability mass on the action that gives the smallest expected payoff. □

The fact that the row/column LPs are dual, and thus there always exists an equilibrium, was found by Von Neumann in 1928; we can find these equilibrium strategies in polynomial time via an LP.

Note that we could try to use the same idea in the prisoner's dilemma that we introduced at the beginning of this lecture (Example 16.2). However, we run into a problem—the row player wants to optimize $\vec{y}^T \mathbf{A} \vec{x}$, while the column player wants to optimize $\vec{x}^T \mathbf{B} \vec{y}$, where $\mathbf{A}$ and $\mathbf{B}$ are completely unrelated. This is because the pair of numbers in each cell are also completely unrelated—there is no nice relation that zero-sum games guarantee. This means that

we don't have the same notion of a dual between these LPs in non-zero-sum games.

In 1951, John Nash showed that equilibria always exist, even with > 2 players, and with games that are not zero sum. This utilizes something called Brewer's fixed point theorem, which is a theorem in topology that is beyond the scope of this course.

In 2006, Daskalakis, Goldberg, and Papadimitriou showed that finding equilibria in even 3-player games is hard. It's not NP-hard (we haven't covered what this exactly means in the class yet), but it's a class of problems called PPAD-hard. Another group of researchers showed the very next year that it's even hard in 2-player games in general.

This means that even though Nash showed that equilibria always exist, it's computationally hard to actually solve for that equilibrium.

10/26/2021

Lecture 17

Multiplicative Weights

17.1 Online Decision-making

Today, we'll talk about an algorithm called Multiplicative Weight Updates, or sometimes also called the "Hedge" algorithm; this relates to online decision-making. Later on, we'll also talk about its relation to zero-sum games.

What is online decision making? Suppose we have a decision that we're trying to make, and we have n experts that have conflicting opinions. Who do we trust, and which decision should we make? Here, in each successive trial, we know the history up until the current trial (that is, we know nothing in trial 1, but in trial 2, we know the results from trial 1, etc.).

Example 17.1

As an example, suppose we have to make a decision on whether the weather will be rainy on a given day. We have n experts, and each day we keep track of which experts were wrong, and keep a running "loss tally" for each expert (where 0 is a correct prediction and 1 is an incorrect prediction):

expert	day 1	loss tally	day 2	loss tally
1	1	1	1	2
2	1	1	0	1
3	0	0	1	1
4	0	0	0	0
5	1	1	1	2
⋮	⋮	⋮	⋮	⋮

Here, each day, we want to make an "online" decision; that is, we only know the results from previous days and the loss tallies up until now. After each day, the losses are revealed and we update the tallies.

Our goal is to find an algorithm that can make these decisions for us, utilizing this history that we have up until the current point in time.

Let us first introduce some notation that we'll use for the remainder of the lecture.

There are n experts, and we follow expert $i_t \in \{1, \dots, n\}$ on day $t \in \{1, \dots, T\}$, where T is the total number of days that we'll be doing this process on.

Each expert incurs a loss $\ell_i^{(t)}$ on day t; this means that *our* total loss is $L := \sum_{t=1}^{T} \ell_{i_t}^{(t)}$. This sums over all days, taking the losses of the experts that we followed on that day. We want to minimize L.

Additionally, in general, there could be multiple actions that we could take (not just binary as we have in the previous example), and the losses can be some number $\ell_i^{(t)} \in [0, 1]$. Here, all that is needed is that the loss is bounded; we can

just rescale to $[0,1]$ anyways.

Ideally, we want to compare $L = \sum_t \ell_{i_t}^{(t)}$ and $\sum_{t=1}^{T} \min_i \ell_i^{(t)}$; the latter case is if we magically know

This is not a realistic goal. Going back to the rain example, suppose we have two experts, where one expert always predicts that it does not rain, and one expert always predicts that it does rain. In the optimal case, we magically know whether it rains or not, and our loss is 0. However, more realistically, if it rains 50% of the time, no matter what we do, we incur a loss of approximately $\frac{T}{2}$; we expect to be wrong half of the time, as we can't possibly predict a uniformly random outcome.

As such, we will settle for something slightly weaker, i.e. compare our L to $\min_i \sum_{t=1}^{T} \ell_i^{(t)}$. That is, we compare our actual loss to the case where we look back and only follow one expert throughout. We further define

$$R := L - \min_i \sum_{t=1}^{T} \ell_i^{(t)} = L - L^*$$

as our "regret". Here, L^* is what we're comparing against. It turns out that we can have an algorithm that gives us very little regret for this definition.

17.2 Algorithms for Online Decision-making

Strategy 1: For all days t, we always follow expert 1; i.e. $i_t = 1$.

The claim is that our regret $R \le T$; that is, we can always be wrong in every day. It isn't hard to come up with an example where $R = T$ (here we only give the losses on each day):

expert	day 1	day 2	day 3	$\cdots$
1	1	1	1	$\cdots$
2	0	0	0	$\cdots$

In this case, we have $L = T$ and $L^* = 0$, meaning $R = L - L^* = T$.

Strategy 2: Follow the majority opinion.

The claim is that our regret $R \le T$. Again, it isn't hard to come up with an example where $R = T$; we can have a bunch of experts that are wrong, and the very last expert that is always right:

expert	day 1	day 2	day 3	$\cdots$
1	1	1	1	$\cdots$
2	1	1	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$n-1$	1	1	1	$\cdots$
n	0	0	0	$\cdots$

In this case, we have $L = T$ and $L^* = 0$, meaning $R = L - L^* = T$.

Strategy 3: Each day, listen to a uniformly random expert.

(Just as a side-note, remember that in our worst-case analysis, we pretend that we have an adversary that can screw us over depending on what strategy we use. That is, the adversary knows everything about the algorithm and strategy, and can act accordingly to give us a worst-case outcome, which we're trying to analyze. However, one crucial limitation is that the adversary cannot possibly know what the output of randomness is—that is, random choices cannot be predicted by the adversary.)

The claim is that $\mathbb{E}[R] \ge (1 - \frac{1}{n})T$.

We will prove an upper bound on $\mathbb{E}[R]$; that is, $\mathbb{E}[R] \le (1 - \frac{1}{n})T$. If we pick a uniformly random expert to follow, each day we have an expected loss of $\frac{1}{n}\sum_{i=1}^{n} \ell_i^{(t)}$. This means that our total expected loss is

$$\mathbb{E}[L] = \sum_{t=1}^{T} \left(\frac{1}{n} \sum_{i=1}^{n} \ell_i^{(t)} \right).$$

We further know that one of these i's is actually the best person of the day, so we can split this up:

$$\mathbb{E}[L] = \sum_{t=1}^{T} \left(\min_i \ell_i^{(t)} + \frac{1}{n} \sum_{j \neq i} \ell_j^{(t)} \right).$$

Firstly, we can see that $\sum_{t=1}^{T} \min_i \ell_i^{(t)} \leq L^*$; this will usually be better than our target L^*, as we have the choice of which best expert to choose on each day. Further, since each loss is upper bounded by 1, and we have $n-1$ other experts in the inner summation, the second term $\frac{1}{n} \sum_{j \neq i} \ell_j^{(t)} \leq \frac{n-1}{n}$.

$$\mathbb{E}[L] \leq L^* + \sum_{t=1}^{T} \frac{n-1}{n} = L^* + \left(1 - \frac{1}{n}\right) T.$$

As such, we have the following upper bound for our expected regret $\mathbb{E}[R]$:

$$\mathbb{E}[R] = \mathbb{E}[L] - L^* \leq \left(1 - \frac{1}{n}\right) T.$$

The lower bound as claimed can also be shown through a similar probabilistic analysis, but what we can conclude is that this strategy can do at best a little bit less than T.

Strategy 4: Follow the expert who's been the best so far.

The claim is that $\mathbb{E}[R] \geq (1 - \frac{1}{n})\, T$, up to some small additive term.

We will give an example with three experts, but we can generalize this as well.

expert	$\ell^{(1)}$	tally	$\ell^{(2)}$	tally	$\ell^{(3)}$	tally	$\ell^{(4)}$	tally
1	0	0	1	1	0	1	0	1
2	$\frac{1}{3}$	$\frac{1}{3}$	0	$\frac{1}{3}$	1	$\frac{4}{3}$	0	$\frac{4}{3}$
3	$\frac{2}{3}$	$\frac{2}{3}$	0	$\frac{2}{3}$	0	$\frac{2}{3}$	1	$\frac{5}{3}$

Here, in the first day, we have no information, so WLOG suppose we choose expert 1. Then, on the next day, the adversary knows we will pick expert 1 to follow, so it will purposefully make expert 1 wrong, while all other experts are correct. The next day, the adversary knows we will pick expert 2 to follow, so it will adjust accordingly to give us the most loss. This can continue for all days, meaning that we're losing on every single day except for the first.

As such, our loss in this worst-case scenario is $T-1$, with some small additive term to account for the first day that is insignificant with large T. This is because we have a loss of one every day except for the first.

One observation is that our first strategy would actually be better for this exact scenario; notice that each expert is mostly correct; any individual expert is only incorrect on $\frac{1}{n}$ of the days. This means that if we had used the first strategy, our expected loss would have been $\frac{T}{n}$, with some additive factor to account for the first day.

This means that we have an actual loss of $L \approx T$, with an optimal loss of $L^* \approx \frac{T}{n}$ with our first strategy. Our regret is thus $R = L - L^* \approx (1 - \frac{1}{n})\, T$.

Strategy 5: (Multiplicative weights/Hedge algorithm) Use both history *and* randomness.

In this strategy, on each day we have a probability distribution $\vec{\boldsymbol{x}}^{(t)}$ on the experts, and on each day, we update this probability based on their performance.

Here, our expected loss on day t is

$$L_t = \left\langle \vec{\boldsymbol{x}}^{(t)}, \vec{\boldsymbol{\ell}}^{(t)} \right\rangle = \sum_{i=1}^{n} x_i^{(t)} \cdot \ell_i^{(t)}.$$

As such, our total loss is $L = \sum_{t=1}^{T} L_t$; note that this takes the expectation into account (i.e. this is our expected loss).

Definition 17.2: Multiplicative Weights/Hedge Algorithm

This algorithm takes in an input parameter $\varepsilon \in (0, \frac{1}{2}]$. We'll define weights $w_i^{(t)}$, such that their initial values $w_i^{(1)} = 1$ for all $i \in \{1, \dots, n\}$. This means that our probability distribution is

$$x_i^{(t)} = \frac{w_i^{(t)}}{\sum_{j=1}^n w_j^{(t)}}.$$

Here, we'll define $W^{(t)} = \sum_{j=1}^n w_j^{(t)}$ as the total weight on day t. To update these weights, we have

$$w_i^{(t+1)} = w_i^{(t)} \cdot (1-\varepsilon)^{\ell_i^{(t)}}.$$

Intuitively, what does this mean? In the extreme case of a binary loss, the people that are right will not have their weights updated. However, the people that are wrong will have their weights exponentially dampened at a rate of $1-\varepsilon$ per day; each day we'd be multiply the weight by a factor of $1-\varepsilon$.

With non-binary losses, the dampening varies based on how bad the loss is, but the same idea holds; we exponentially penalize the people that are wrong, and do nothing to the people that are correct.

To analyze this algorithm, we will use two main lemmas:

Lemma 17.3

The weight at the end of day T, $W^{(T+1)}$, is at most $(1-\varepsilon)^{L^*}$. (Here we use $T+1$ because $W^{(t)}$ is the weight at the beginning of day t.)

Proof. We know that

$$W^{(T+1)} = \sum_{i=1}^n w_i^{(T+1)} \geq w_{i^*}^{(T+1)},$$

where here i^* is the best expert, i.e. it's the expert that defines L^*.

We further know that

$$w_{i^*}^{(T+1)} = \prod_{t=1}^T (1-\varepsilon)^{\ell_{i^*}^{(t)}} = (1-\varepsilon)^{\sum_{t=1}^T \ell_{i^*}^{(t)}} = (1-\varepsilon)^{L^*}.$$

□

Lemma 17.4

$W^{(T+1)} \leq n \cdot \prod_{t=1}^T (1-\varepsilon L_t)$.

Proof. We will show that in general, $W^{(t+1)} \leq W^{(t)}(1-\varepsilon L_t)$. If we had this, then by induction $W^{(T+1)} \leq W^{(1)} \cdot \prod_{t=1}^T (1-\varepsilon L_t)$, where $W^{(1)} = n$ on the first day.

So, to show the first inequality, we know that

$$W^{(t+1)} = \sum_{i=1}^n w_i^{(t+1)} = \sum_{i=1}^n w_i^{(t)} \cdot (1-\varepsilon)^{\ell_i^{(t)}}.$$

Here, we can bound $(1-\varepsilon)^{\ell_i^{(t)}} \leq \left(1-\varepsilon \ell_i^{(t)}\right)$, meaning we have

$$W^{(t+1)} \leq \sum_{i=1}^n w_i^{(t)} \cdot (1-\varepsilon \ell_i^{(t)}).$$

Knowing that $w_i^{(t)} = \sum_i x_i^{(t)} W^{(t)}$, we have that

$$W^{(T+1)} \leq \sum_{i=1}^{n} x_i^{(t)} W^{(t)} \cdot (1 - \varepsilon \ell_i^{(t)}) = W^{(t)} \Bigg(\underbrace{\sum_{i=1}^{n} x_i^{(t)}}_{1} - \varepsilon \underbrace{\sum_{i=1}^{n} x_i^{(t)} \ell_i^{(t)}}_{L_t} \Bigg) = W^{(t)}(1 - \varepsilon L_t).$$

The last simplification makes use of the fact that $\vec{\boldsymbol{x}}^{(t)}$ is a probability distribution (and hence its elements sum to 1), and the definition of the expected loss L_t.

Since we've now proved our initial claim, the original lemma is proven as well. □

Theorem 17.5

`Hedge`(ε) achieves $\mathbb{E}[R] \leq \varepsilon \cdot T + \frac{\ln n}{\varepsilon}$. If we choose $\varepsilon = \sqrt{\frac{\ln n}{T}}$ to equate these terms, we have $\mathbb{E}[R] \leq 2\sqrt{T \ln n}$.

Proof. We will use the two prior lemmas to prove this theorem.

The lemmas give us an upper and a lower bound for W_T, so we can compare the two sides; now taking the natural log of both sides, we have

$$(1-\varepsilon)^{L^*} \leq n \cdot \prod_{t=1}^{T} (1 - \varepsilon L_t)$$

$$L^* \ln(1-\varepsilon) \leq \ln n + \sum_{t=1}^{T} \ln(1 - \varepsilon L_t)$$

We will now make use of the fact that $\forall z \in [0, \frac{1}{2}]$, $-z - z^2 \leq \ln(1-z) \ln -z$; this can be derived by taking the first few terms of the Taylor expansion.

This means that we have

$$L^*(-\varepsilon - \varepsilon^2) \leq \ln n - \varepsilon \sum_{t=1}^{T} L_t$$

$$L^*(-1 - \varepsilon) \leq \frac{\ln n}{\varepsilon} - \sum_{t=1}^{T} L_t$$

$$\sum_{t=1}^{T} L_t - L^* \leq \frac{\ln n}{\varepsilon} + \varepsilon L^*$$

We know that our loss $L = \sum_{t=1}^{T} L_t$, and we can bound $L^* \leq T$, so this means that we have

$$R = L - L^* \leq \frac{\ln n}{\varepsilon} + \varepsilon T,$$

as desired. □

17.3 Connection to Zero Sum Games

We used strong duality last lecture to prove that if we have two players in a zero sum game, the expected payoff is the same for each optimization problem (one for each player), which were duals.

It turns out that we can just use weak duality combined with the main theorem of the Hedge algorithm; this also proves that the expected payoff is the same for both players. Further, this actually gives us an algorithm to find this pair of strategies.

The row player and the column player both start with a uniform mixed strategy. On day t, the row player is revealed a loss vector $\ell^{(t)} = -\mathbf{A}\vec{y}^{(t)}$, and the column player is revealed a loss vector $\ell^{(t)} = \mathbf{A}^T\vec{x}^{(t)}$. That is, we rig the loss vectors such that the resulting payoffs are exactly the same as the Hedge algorithm.

If we run multiple iterations of this game and modify the probability distributions for each player, we'd end up converging to the same expected payoff (by weak duality).

10/28/2021

Lecture 18

Reductions, Bipartite Matching

18.1 Reductions

Definition 18.1: Reduction

Suppose we have two computational problems A and B. We say that "A reduces to B", i.e. $A \to B$, if an efficient algorithm for B means that there is an efficient algorithm for A.

The good thing here is that if we know how to solve B efficiently, then we can solve A efficiently.

The bad thing is that if we can prove that A has no efficient algorithm, then B must not have an efficient algorithm either (because then we could just use the algorithm for B to solve A).

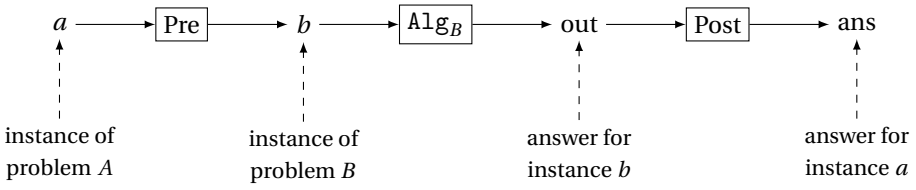
Hence, we can use reductions to get efficient algorithms and also to prove no efficient algorithms exist.

There are a few types of reductions. For example,

- Turing reductions
- Cook reductions
- Many-to-one reductions
- Log-space reductions
- etc.

For today, we will mostly be talking about Cook reductions. Specifically, we're dealing with reductions that take polynomial time. That is, if we know a polynomial time algorithm $\texttt{Alg}_B$ that solves B, then we can call it (possibly multiple times) inside a solution to A. This means that A can be solved in polynomial time as well.

A possible Cook reduction could be as follows:



A more general reduction could make multiple calls to $\texttt{Alg}_B$, but this is the general process. We take an instance of problem A (i.e. a specific case of the problem A), do some pre-processing, and feed it in as an input for an instance of problem B. Utilizing $\texttt{Alg}_B$ to get the answer for this instance, we can then do some post-processing to turn that into an answer for A.

We'll look at a few examples today:

- Maximal Bipartite Matching → Max Flow
- Any problem $\in P$ → Circuit Value Problem → Linear Programming

- Matrix Multiplication → Matrix Inversion, Matrix Inversion → Matrix Multiplication

Here, P is the class of boolean problems (decision problem) that can be solved in polynomial time in the input size.

One thing to note here; it may seem trivial that the second point is true—if we were talking about Cook reductions, if we already *know* that there is a polynomial time solution for a problem in P (this is true by definition of being *in* P), then we could just run that algorithm at pre-processing, and we're done; we don't even need the reductions.

In a similar vein, we can claim that any problem $\in P$ can be reduced to some non-polynomial-time problem, trivially. That is, we can just run our polynomial time algorithm to find an answer, and run the non-polynomial time problem completely separately and independently.

However, the important thing to note here is that the reductions in the second point are *log-space reductions*. That is, the pre- and post-processing takes logarithmic extra working memory; it's a very space-efficient conversion. This is what makes this reduction non-trivial and useful. We won't really be talking too much about this kind of reduction in CS170 though.

One more note is that usually we only care about the conversions being in some polynomial time; today, we'll really care about finding very efficient pre- and post- algorithms (ex. linear time), making the reductions themselves efficient as well.

18.2 Maximum Bipartite Matching

Definition 18.2: Matching

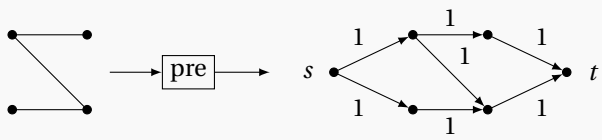
A matching M on a graph $G = (V, E)$ is a subset of E such that no two edges in M share a common vertex.

The input is an undirected bipartite graph. Our goal is to find a matching M where $|M|$ is as large as possible. We'd like to come up with an efficient algorithm for this problem, and we'll use a reduction to max-flow to accomplish this.

Specifically, we'll take in an undirected bipartite graph, and pre-process it to become a directed graph with weights, and special vertices s and t. Running any max-flow algorithm, we'll get a flow as an output, which we then post-process to create a matching.

What is the pre-processing algorithm? For an input graph, we will make all edges directed going from left to right (i.e. from one group to the other in the bipartite graph), all with capacity 1. We will then add a vertex s to the left of everything, with edges going to the left group, all with capacity 1, and we will add a vertex t to the right of everything, with edges coming from the right group, all with capacity 1.

Example 18.3



One thing to notice here is that we aren't actually reducing to the typical max-flow problem; we'd have issues if we have fractional flows along any edges (which is very possible as it is now). Hence, we'll add the additional constraint that all flows must be integers—we know by construction that this is fine, because all our capacities are integers. (For example, Ford-Fulkerson would give us an integral max-flow.)

We further claim that matchings and integral flows are in bijective correspondence. In particular, a matching M maps to f_M such that $\text{val}(f_M) = |M|$, and an integral flow f maps to M_f such that $\text{val}(f) = |M_f|$.

That is, given a flow, we can extract a matching with the same size of the value of the flow, and if there exists any matching of greater size, there is always a corresponding valid flow. This means that these two problems are essentially the same.

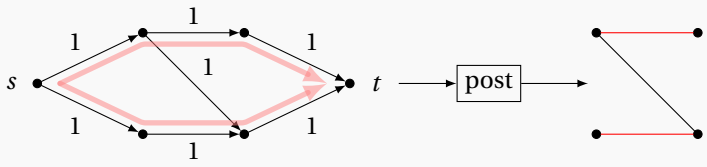
To prove this claim, let us consider these two cases. How do we go from a matching M to a flow f_M? Given a matching $M = \{e_1, \ldots, e_R\}$, we can see that the flow along the edge e in the flow f_M is

$$(f_M)_e = \begin{cases} 1 & \text{if } e = (s,b) \text{ and } b \text{ is matched in } M \\ 0 & \text{if } e = (s,b) \text{ and } b \text{ is not matched in } M \\ 1 & \text{if } e = (a,t) \text{ and } a \text{ is matched in } M \\ 0 & \text{if } e = (a,t) \text{ and } a \text{ is not matched in } M \\ 1 & \text{otherwise and } e \in M \\ 0 & \text{otherwise and } e \notin M \end{cases}$$

Given an integral flow f, we know that f decomposes into a collection of cycles and s-t paths. However, the graph has no cycles by the way that we're creating the graph (it's a DAG). This means that f will always be a collection of s-t paths. As such, each path has an integer value along the path—and each value must be 1, because all the capacities are 1.

Each one of these paths is of the form $s \to a \to b \to t$, and all we'll do is put (a,b) in the matching M_f.

Example 18.4



What's the runtime of this algorithm? Ford-Fulkerson gives a runtime of $O((m+n)\cdot \text{val}(f^*))$. In our case, $m > n$, and at further the maximum flow value is at most $\frac{n}{2}$, because each vertex could be matched to one other, giving us a match size of $\frac{n}{2}$. This gives us a runtime of $O(mn)$.

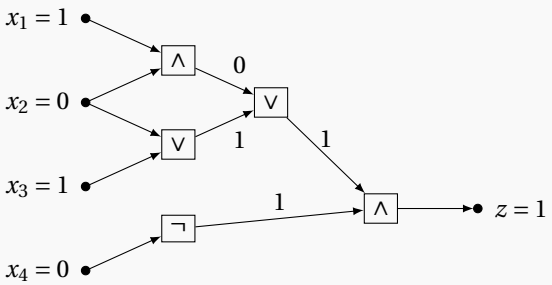
The current records are $\tilde{O}(m+n^{1.5})$ by Van den Brand, Lee Nanongkai, and Peng in 2020, and also $\tilde{O}(m^{\frac{4}{3}})$ by Kathuria, Liu, Sidford in 2020. (Here, $\tilde{O}$ just hides some polynomial factors.)

18.3 Circuit Value Problem

In the circuit value problem, we're given a boolean circuit C with n input bits $x_1, \ldots, x_n \in \{0,1\}$ and one output bit z. We want to compute o. Here, we will consider circuits containing AND, OR, and NOT gates.

Example 18.5

Suppose we're given the following circuit and inputs:



Following the circuit, the output value here is $z = 1$.

As we can see, there isn't too much going on here; all we need to do is push things forward through the circuit. Since the circuit C is given as a DAG, we can topologically sort C, then simulate the circuit from left to right to compute values of the intermediary gates. This means that there is a linear time algorithm solving this problem.

The claim is that any polynomial-time algorithm has a log-space reduction to the circuit value problem. We do not cover this proof in CS170, but you can take CS172 to learn more. A sketch of the proof is as follows.

Since we have a problem A in P, we know that there exists a polynomial-time algorithm $\texttt{Alg}_A$ for A. Given an instance a for A, running $\texttt{Alg}_A(a)$ uses at most $T = \text{poly}(n)$ memory and time.

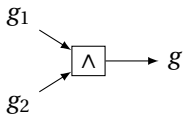
We can think of any algorithm contains some code (instructions), and a memory state. Based on this information, the algorithm transitions to a new memory state based on the instruction. This means that we can make T circuits, one for each time step, and the outputs for the circuit at time t, i.e. the outputs of the circuit C_t, are the inputs to circuit C_{t+1}.

This means that all we're doing is just simulating the program, using these circuits to compute the next memory state.

We will now show that the circuit value problem reduces to linear programming (and thus by transitivity, all problems in P reduces to linear programming).

We will have one variable per gate and input bit. Let z_g be the variable in the LP that stores the value of gate g. We have the following constraints:

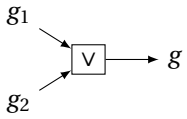
- $\forall g, 0 \le z_g \le 1$
- $\forall$ input gates $x_j = b$, add the constraint $z_j = b$
- If we have an AND gate:



We can have the following constraints:

$$z_g \le z_{g_1}$$
$$z_g \le z_{g_2}$$
$$z_g \ge z_{g_1} + z_{g_2}$$

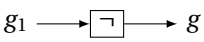
- If we have an OR gate:



We can have the following constraints:

$$z_g \ge z_{g_1}$$
$$z_g \ge z_{g_2}$$
$$z_g \le z_{g_1} + z_{g_2}$$

- If we have a NOT gate:



We can have the following constraint:

$$z_g = 1 - z_{g_1}.$$

We don't have anything that we're maximizing or minimizing, so we can just put a constant as the objective; we only care about whether these constraints are feasible or not.

18.4 Matrix Inversion

Our claim is that $(n \times n)$ matrix multiplication reduces to matrix inversion, and vice versa. The opposite direction is perhaps more useful for us, because we already know that there exist fast algorithms for matrix multiplication. On the other hand, we shouldn't expect that matrix inversion is easier than matrix multiplication, because they're the same problem).

First, let us prove that matrix multiplication can be reduced to matrix inversion. We want $\mathbf{A} \times \mathbf{B}$; suppose we

pre-process the inputs $\mathbf{A}$, $\mathbf{B}$ to get

$$\mathbf{M} = \begin{bmatrix} \mathbf{I} & -\mathbf{A} & 0 \\ 0 & \mathbf{I} & -\mathbf{B} \\ 0 & 0 & \mathbf{I} \end{bmatrix}.$$

It turns out that if we invert this matrix, we get

$$\mathbf{M}^{-1} = \begin{bmatrix} \mathbf{I} & \mathbf{A} & \mathbf{A}\times\mathbf{B} \\ 0 & \mathbf{I} & \mathbf{B} \\ 0 & 0 & \mathbf{I} \end{bmatrix}.$$

The post-processing algorithm then spits out the top-right block.

Both the pre- and post-processing algorithms take $O(n^2)$ to run, because all we're doing is writing matrices down. Put another way, the reduction is linear in terms of the input size (as the input is an n^2 size matrix).

Next, let us prove that matrix inversion reduces to matrix multiplication (in linear $O(n^2)$ time, i.e. linear to the input size).

As an example, let us look at 2×2 gaussian elimination; we essentially have

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{C}\mathbf{A}^{-1} & \mathbf{I} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ 0 & \mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B} \end{bmatrix}.$$

Breaking this down, all we're saying is that we want to divide the first row by $\mathbf{A}$ (i.e. multiply by the inverse), multiply it by $\mathbf{C}$, and subtract it from the second row. This gives us a zero (matrix) in the lower-left corner.

Further, from linear algebra, we know that if $\mathbf{X} = \mathbf{Y}\mathbf{Z}$, then $\mathbf{X}^{-1} = \mathbf{Z}^{-1}\mathbf{Y}^{-1}$. Letting $\mathbf{S} = \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}$ and $\mathbf{Y} = \mathbf{C}\mathbf{A}^{-1}$ for ease, this means that we have

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1} \\ 0 & \mathbf{S}^{-1} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I} & 0 \\ -\mathbf{Y} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^{-1} - \mathbf{Z}\mathbf{Y} & \mathbf{Z} \\ -\mathbf{S}^{-1}\mathbf{Y} & \mathbf{S}^{-1} \end{bmatrix}.$$

In this last matrix, we let $\mathbf{Z} = -\mathbf{A}^{-1}\mathbf{B}\mathbf{S}^{-1}$ for ease. We now can feed this idea into a divide-and-conquer algorithm.

That is, in order to invert a $n\times n$ matrix, it suffices to invert 2 $\frac{n}{2}\times\frac{n}{2}$ matrices (i.e. we invert $\mathbf{A}$ and $\mathbf{S}$), and some constant number of matrix multiplications, additions, and subtractions. This gives us the recurrence

$$T(n) \le 2\cdot T\left(\frac{n}{2}\right) + O(n^\omega) \implies T(n) = O(n^\omega).$$

11/4/2021

Lecture 19

Search Problems

Recall that we talked about reductions last time. That is, suppose we know that A reduces to B (i.e. $A \to B$). This means that if B is easy, then A is easy, and if A is hard, then B is hard.

Today, we'll build up some of these concepts in talking about P/NP, NP-hardness, and NP-completeness.

Firstly, we'll talk about *binary relations*. That is, $R \subseteq \{0,1\}^* \times \{0,1\}^*$, where $(x,w) \in R$. Here, we have $(x,w)\in R$, where we call x an "instance", and w a "witness".

We'll be looking at two kinds of problems:

- `Decide(`R`)`: given x, does there exist w such that $(x,w)\in R$?
- `Search(`R`)`: given x, return some w such that $(x,w)\in R$, or detect if no such w exists.

We can see that `Decide(`R`)` $\to$ `Search(`R`)`, because we can use the result of `Search` to determine whether there exists an w or not.

It turns out that `Search(`R`)` $\to$ `Decide(`R`)` as well, by slightly modifying R. Specifically, to search for a specific w, we'd first ask "does there exist a w at all?", and we can stop early if not.

Then, we ask `Decide(`R`)` to determine whether there exists a w, bit by bit. That is, we ask "does there exist a w that starts with 0?", then based off of that (suppose there does), we ask "does there exist a w that starts with 00?", etc. As such, we can slowly determine what w is, bit by bit, through `Decide(`R`)`.

Example 19.1: Max Flow

We can think of the instance x as a version of the problem, and the witness w is something that convinces us of a solution to the problem.

In the case of a max-flow problem, we may have $x = G, s, t$ (i.e. the graph, the source, and the sink), and $w = f^*$ (i.e. the max flow). Note that these values will all actually just be binary strings in essence, because we're storing them on a computer and manipulating them on a computer.

Here, we can see that for any x, `Decide(`R`)` should always return "yes" (a max-flow always exists). We also have `Search(`R`)` as an algorithm to find f^*.

To make `Decide(`R`)` a little bit more interesting (and helpful), we could let x also take in a value v, and modify w to find a flow f such that $\text{val}(f) \geq v$. Now what we're asking in `Decide(`R`)` is "does there exist a flow with value at least v?" In `Search(`R`)` now, we'd still compute the max flow.

Here is a natural question that follows: does there always exist an algorithm to solve `Decide(`R`)` for any R?

It turns out that the answer is no! The Halting Problem is an example.

Here, we have $x = P$, the description of a program, where $(x, w) \in R_{Halt}$ if and only if P halts when it is run. The proof of the undecidability of this problem is left out here, as it is covered in CS70.

Today, we will focus on R's with an efficient verifier. Here, a verifier V_R is just an algorithm, which given (x, w), determines

$$V_R(x, w) = \begin{cases} 1 & \text{if } (x, w) \in R \\ 0 & \text{if } (x, w) \notin R \end{cases}.$$

We will look at problems R such that there exist a verifier V_R running in poly(|x|), that is, polynomial in the size of x.

Note that any such R can be decided (or even searched) in $\leq \exp\big(\text{poly}(|x|)\big)$. Why? If such a V_R exists, we can always brute force search over all w and try to find one such that $V_R(x, w) = 1$.

19.1 P and NP

P and NP are two "complexity classes" that we'll be looking at today.

Definition 19.2: Complexity Class

A *complexity class* is a set of problems that is defined by some resource constraints.

Definition 19.3: P

P stands for "polynomial time"; it is the set of all $L \subseteq \{0,1\}^*$ such that there exists an algorithm to determine whether or not $x \in L$ in time poly(|x|).

That is, if L was the set of all inputs that would give us an answer of "yes" to a problem, we can determine efficiently whether or not an input $x \in L$; i.e. whether x would give us "yes" to the problem.

Definition 19.4: NP

NP stands for "nondeterministic polynomial time" (it does *not* stand for "not polynomial").

This name comes from a "nondeterministic Turing machine"—this is a program model that has unbounded parallelism. (For example, in a brute force search with a nondeterministic program, we could try all the different branches of the search in parallel, and we can have as many parallel branches as we'd like.)

As such, nondeterministic polynomial time means that there exists a nondeterministic program which runs in polynomial time—that is, the time it takes the longest branch to run is polynomial. Of course, in reality we can never have such a program, but this is how this complexity class is defined.

For us, an equivalent definition is that NP is the class of problems R such that there exists a poly($|x|$) time V_R such that for all x, if there exists a w where $(x, w) \in R$, then there exists a poly($|x|$)-sized w such that $(x, w) \in R$.

Put another way, NP is the class of problems R with instances where "yes" answers can be verified in polynomial time (where we're given a "hint" w to help us verify). The verifier must also be correct if the instance should give a "no" answer, but we don't care about its time complexity if the answer is "no".

One thing to note is that $\mathsf{P} \subseteq \mathsf{NP}$. That is, all problems in P are also in NP. If a problem $R \in \mathsf{P}$, then there exists a polynomial-time algorithm A with solves `Decide(`R`)`, by definition of P. This means that V_R can just run A and return A's output in polynomial time.

The question of whether or not $\mathsf{P} = \mathsf{NP}$ is still unsolved.

Definition 19.5: NP-hard

A problem R is NP-hard if every $B \in \mathsf{NP}$ has a reduction to R, i.e. $(\forall B \in \mathsf{NP})\ B \to R$. Note that R might not itself be in NP.

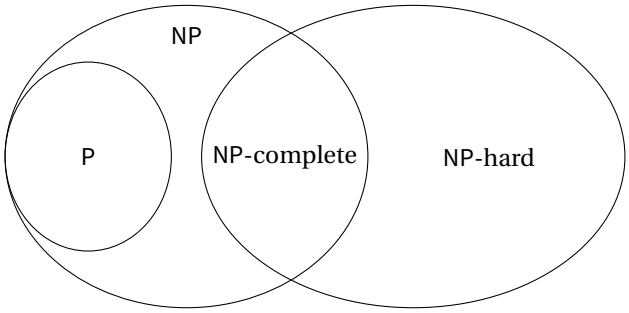
It is possible to show that the halting problem R_{Halt} is NP-hard (but the halting problem is definitely not in NP).

Definition 19.6

A problem R is NP-complete if both (1) $R \in \mathsf{NP}$ and (2) R is NP-hard.

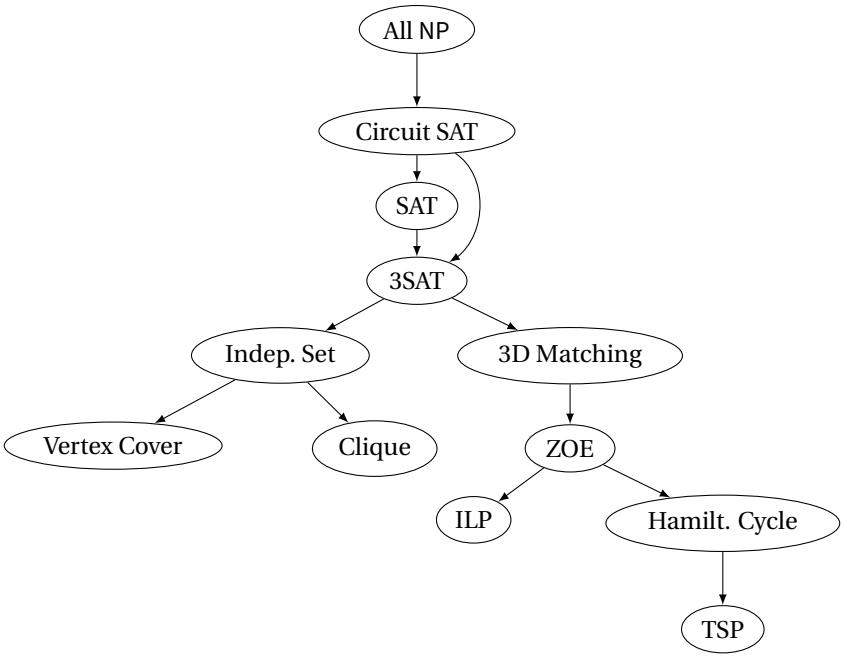
As such, NP-complete problems are the hardest problems in NP.

Under the assumption that $\mathsf{P} \neq \mathsf{NP}$, then we have the following diagram of the space of problems in P, NP, NP-hard, and NP-complete (if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{P} = \mathsf{NP} = \mathsf{NP}$-complete would be contained entirely within NP-hard):



How do we show that a problem is NP-hard? It seems like we have to show an infinite number of reductions; how would we do this? The hard part is finding just one problem in NP-hard; once we have one, we no longer need to show an infinite number of reductions; we only need one.

In the next few lectures, we will go through the following chain of reductions:



19.2 Circuit SAT

The Circuit SAT problem R_{CSAT} contains tuples (x, w), where $x = C$ is a circuit containing AND, OR, and NOT gates, and $(x, w) \in R_{CSAT}$ if and only if $C(w) = 1$. That is, given a circuit C, we want to check whether there is an input w that gives us an output of 1.

Theorem 19.7: Circuit SAT is NP-complete

We claim that Circuit SAT is NP-complete.

Proof. Firstly, we need to check whether CSAT is in NP. We can see pretty easily that there is an efficient V_{CSAT}. Given a circuit C and an input w, V_{CSAT} simply evaluates C on w. That is, we topologically sort the gates, and compute the value of each gate in reverse topological order.

Next, we need to show that CSAT is NP-hard. This proof will be hand-wavy; see CS172 for a more rigorous treatment.

The idea is as follows. Suppose $B \in$ NP; we need to show that it reduces to CSAT. Since $B \in$ NP, there must exist an efficient verifier V_B such that $V_B(x, w) = 1$ if and only if $(x, w) \in B$.

At the end of the day, V_B is a program; we can look at the source code and preprocess it (along with x) to create a circuit C_B such that $C_B(w) = V_B(x, w)$. That is, this circuit C_B models the computation of V_B with x on w. (Here, we skip over the derivation of why we can create this circuit and how we create the circuit.)

We then feed C_B to the CSAT algorithm; the result will be our desired answer. □

Next, we will look at how Circuit SAT reduces to SAT and 3SAT.

Firstly, in the SAT problem, the input is a formula in "Conjunctive Normal Form" (CNF). That is, we have some variables and their negations, in a finite number of clauses. Each clause is a series of ORs, and the clauses are put together with ANDs. For example,

$$\phi = (x_1 \vee \bar{x}_7) \wedge (x_8) \wedge (x_2 \vee x_1 \vee x_3 \vee x_5 \vee \bar{x}_7) \wedge \cdots.$$

The question is "does there exist a setting of x that makes $\phi(x)$ true?" The setting of x is our w to our problem.

Theorem 19.8: SAT is NP-complete

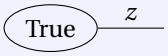
We claim that SAT is NP-complete.

Proof. Firstly, we need to check whether SAT is in NP. Given a solution w, we can verify this solution easily by evaluating the formula on w, in linear time with respect to the size of ϕ.

Next, we need to show that SAT is NP-hard. Here, we'll use the fact that reductions are transitive; that is, if $A \to B$ and $B \to C$, then $A \to C$. Since we've already shown that CSAT is NP-hard, every problem in NP reduces to CSAT. As such, it suffices to show that CSAT reduces to SAT; that is, given an instance of SAT, we need to use SAT to solve CSAT.

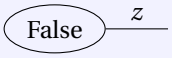
If we have some circuit C, we have some inputs, some gates, and some wires connecting gates and/or inputs. We'll assign each wire connecting gates/inputs to be a variable, and we'll create clauses using these variables to describe this circuit.

- If we have the following gate: We have the corresponding clause:



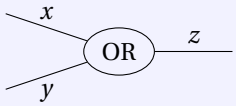
$$(z)$$

- If we have the following gate: We have the corresponding clause:



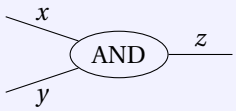
$$(\bar{z})$$

- If we have the following gate: We have the corresponding clauses:



$$(z \vee \bar{x}) \wedge (z \vee \bar{y}) \wedge (\bar{z} \vee x \vee y)$$

- If we have the following gate: We have the corresponding clauses:



$$(\bar{z} \vee x) \wedge (\bar{z} \vee y) \wedge (z \vee \bar{x} \vee \bar{y})$$

- If we have the following gate: We have the corresponding clauses:

$$(z \vee x) \wedge (\bar{z} \vee \bar{x})$$

In some of the more complex gates (i.e. OR, AND, NOT), all we're doing is constructing a series of clauses (called a "gadget") that forces z to be a certain value for any given pair of values we assign to x and y. That is, the truth table for z matches up with the truth table for the gate. (Remember that we are looking for a setting that satisfies all of the clauses, since they're joined by ANDs.)

Now that we have all of these gate conversions, we can AND all of these corresponding clauses together, along with the clause $\wedge(o)$, to also force the output gate o to be true.

We can then feed this CNF formula to our SAT algorithm, which will tell us if there is a satisfying assignment that makes all the clauses true. In such a satisfying assignment, there must have been some values set to the input gates (and in particular sets the output gate to be true) which were the inputs that make the circuit output a value of 1, verifying CSAT. As such, we've just reduced CSAT to SAT, and thus SAT is NP-hard as well (and therefore NP-complete). □

Corollary 19.9: 3SAT is NP-complete

We claim that 3SAT is NP-complete.

Proof. The proof follows directly from the reduction of CSAT to SAT; since our conversions only involved clauses of size 1, 2, or 3, the formula satisfies the constraints of 3SAT (namely, that each clause contains at most 3 variables). This means that our earlier reduction also shows that 3SAT is NP-complete. □

19.3 Independent Set

An independent set is a subset $S \subseteq V$ such that no two elements in S are neighbors in G.

In the independent set problem, we're given an undirected graph G and an integer k; we want to determine whether there exists an independent set of size at least k. (The search version of this problem would be to find the largest possible independent set in the graph G.)

Theorem 19.10: Independent Set is NP-complete

We claim that the independent set problem is NP-complete.

Proof. Firstly, we can see that this problem is in NP. Given a solution w, we can easily verify it in polynomial time by checking if any two vertices in the set are neighbors.

The hard part is the reduction to show that this problem is NP-hard. We'll be reducing 3SAT to independent set; that is, we want to use independent set to solve 3SAT.

Given some formula ϕ, we'll have some finite number of clauses. For each clause, we'll create a clique of vertices, where each vertex is labeled by a variable in the clause. Note that we can possibly end up with multiple vertices with the same label, if the same variable appears in multiple clauses.

After creating these cliques, we then add additional edges connecting each pair of complementary variables; that is, we connect all vertices labeled x to all vertices labeled $\bar{x}$, etc.

If we run independent set on this final graph, we can see that an independent set of size n (where n is the number of clauses in ϕ) exists if and only if ϕ is satisfiable.

The reason why this is true comes from how the edges are placed; the independent set must contain one vertex in each clique, and this chosen vertex represents the expression we choose to be true in the clause. If there are multiple possible solutions, we could just choose any one of them. The edges connecting the variable complements ensures that we never set both x and $\bar{x}$ to true at the same time; they're always neighbors in the graph.

As such, we've reduced 3SAT to independent set, and proved that independent set is NP-complete. □

11/30/2021

Lecture 20

NP-Completeness

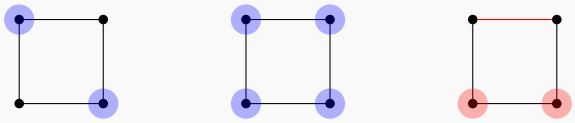
20.1 Vertex Cover

Definition 20.1: Vertex Cover

For a graph $G = (V, E)$, a set $S \subseteq V$ is a vertex cover if $\forall e \in E \; \exists v \in S$ such that e touches v.

Example 20.2

The first two sets (in blue) are vertex covers of the graphs, but the last set (in red) is not a vertex cover.



The vertex cover problem takes an input (G, k), where $G = (V, E)$ is a graph, and we want to determine whether G has a vertex cover of size at most k.

Theorem 20.3: Vertex Cover is NP-complete

We claim that the vertex cover problem is NP-complete.

Proof. Firstly, we can see that this problem is in NP. Given a solution set S, we can easily verify it in polynomial time by checking whether all edges touch some vertex in S.

The reduction to prove that this problem is NP-hard is quite easy—we will reduce independent set to vertex cover, and we claim that a set S is a vertex cover if and only if $V \setminus S$ is an independent set.

($\implies$) Suppose S is a vertex cover, and suppose for contradiction that $V \setminus S$ is not an independent set. This means that there exist two vertices $a, b \in V \setminus S$ connected by an edge such that both a and b are in $V \setminus S$, meaning neither a nor b are in S. However, we know that since S is a vertex cover, at least one of these vertices must be in S, since they're connected by an edge—this is a contradiction, and thus $V \setminus S$ must be an independent set.

($\impliedby$) Suppose $V \setminus S$ is an independent set, and suppose for contradiction that S is not a vertex cover. This means that there exists an edge with two adjacent vertices a and b such that neither vertex is in S. However, this means that both vertices must be in $V \setminus S$, which would violate the condition that $V \setminus S$ is an independent set—this is a contradiction, and thus S is a vertex cover.

As such, vertex covers and independent sets are in one-to-one correspondence. If we're given an input (G, k) to independent set, we will reduce this to vertex cover with the input $(G, n - k)$, where $n = |V|$. That is, finding a vertex cover of size $n - k$ gives us directly the independent set of size k. □

20.2 Clique

Definition 20.4: Clique

In a graph $G = (V, E)$, a set $S \subseteq V$ is a clique if there exist edges between all pairs of vertices in S.

The clique problem takes an input (G, k), where $G = (V, E)$ is a graph, and we want to determine whether G has a clique of size at least k.

Theorem 20.5: Clique is NP-complete

We claim that the clique problem is NP-complete.

Proof. Firstly, we can see that this problem is in NP. Given a solution set S, we can easily verify it in polynomial time by checking whether there exist edges between all pairs of vertices in S.

The reduction to prove that this problem is NP-hard is also quite straightforward—we will reduce independent set to clique, and we claim that a set S is a clique if and only if it is an independent set in G', the complement of G (that is, if an edge was in G, it is not in G', and vice versa).

This can easily be shown as well—in any independent set S, no two vertices are connected by an edge. In the complement G', every pair of vertices in S is connected by an edge, and as such S is a clique in G'.

As such, cliques and independent sets are in one-to-one correspondence. If we're given an input (G, k) to independent set, we will reduce this to clique with the input (G', k). That is, finding a clique of size at least k in the complement G' is equivalent to finding an independent set of size at least k in the original graph G. □

20.3 3D Matching

3D matching is a generalization of bipartite matching. That is, instead of trying to match the most pairs, we want to match the most triples.

That is, we are given n each of three categories (ex. boxes, cars, drivers); let us label these as $B = (b_1, \ldots, b_n)$, $C = (c_1, \ldots, c_n)$, and $D = (d_1, \ldots, d_n)$. We are also given a set T of triples $T \subseteq B \times C \times D$, and we want to determine whether there exists a subset S of T which covers each element of B, C, and D exactly once (i.e. each element is in some triple, and no element is used more than once).

We can prove that 3D matching is NP-hard through a reduction from 3SAT to a special case of 3SAT called $3\text{SAT}_{L\leq 2}$, and reduce from $3\text{SAT}_{L\leq 2}$ to 3D matching.

Definition 20.6: $3\text{SAT}_{L\leq 2}$

A $3\text{SAT}_{L\leq 2}$ instance has an answer of True if the formula ϕ is a satisfiable 3SAT instance, and each literal in ϕ occurs at most 2 times.

Here, x and $\bar{x}$ are two distinct literals, meaning each variable can occur at most 4 times, twice in its positive form and twice in its negative form.

Theorem 20.7: $3\text{SAT}_{L\leq 2}$ is NP-complete

We claim that $3\text{SAT}_{L\leq 2}$ is NP-complete.

Proof. It should be clear that $3\text{SAT}_{L\leq 2}$ is in NP; we will focus on the reduction from 3SAT.

Suppose ϕ is a 3SAT instance on n variables. We can first simplify ϕ by removing any variable $x_1, \ldots, x_n$ that only appears positively or negatively. That is, if x only appears as x, and not $\bar{x}$, then we can just set it True. Equivalently, if x only appears as $\bar{x}$, and not x, then we can just set it to False.

Now, we have simplified our formula down to a form where each variable appears positively and negatively.

If each of the literals occur at most twice, then we're done. If some literal appears more than twice, then the

corresponding variable must also appear at least 3 times.

Suppose that the variable x appears t times. We will then create t new variables $x_1, \dots, x_t$ and replace the ith occurrence of x with x_i.

However, this does not preserve satisfiability by itself; we need to add consistency to ensure that $x_1 = x_2 = \cdots = x_t$ as well. It turns out that this isn't too hard; all we need to do is add a few more clauses:

$$(x_1 \vee \bar{x}_2) \wedge (x_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_4) \wedge \cdots \wedge (x_t \vee \bar{x}_1).$$

We can see that this ensures that all of these variables have the same value. If x_1 is False, then this forces $\bar{x}_2$ to be True, and hence x_2 must be False; this in turn forces x_3 to be False, etc. If x_1 is True, then this forces x_t to be True, and in turn forces x_{t-1} to be True, etc.

As such, no matter whether x_1 is True or False, all of these variables are consistent.

Further, we can see that after this transformation, no literal appears more than twice; we use it once (either positively or negatively) in the original formula, and we use a positive and a negative literal in the additional clauses.

Thus, we've shown that 3SAT and $3\text{SAT}_{L\leq 2}$ are equivalent problems, and as such $3\text{SAT}_{L\leq 2}$ is NP-complete. □

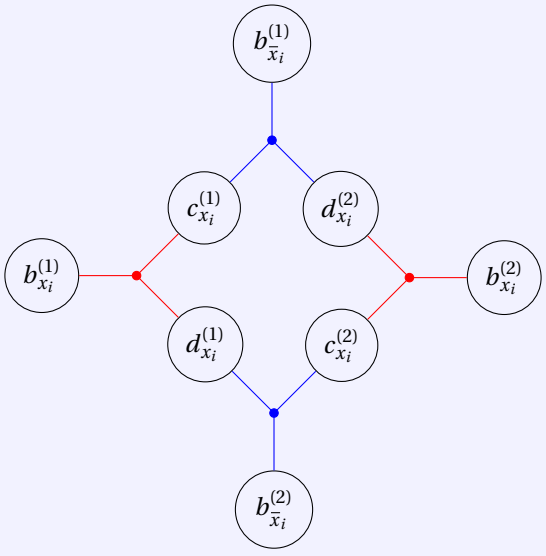
Theorem 20.8: 3D Matching is NP-complete

We claim that 3D matching is NP-complete.

Proof. It should be clear that 3D matching is in NP; we will focus on the reduction from $3\text{SAT}_{L\leq 2}$.

Here, we are given some formula ϕ and we want to create an instance of 3D matching to solve 3SAT for ϕ.

To start out, let us introduce a "gadget":



Remember that in 3D matching, we would like to match each element in B, C, and D. Looking only at c's and d's, there are only two ways to match all c's and d's; we can either select the blue triples, or the red triples.

As such, this binary choice can be used for a variable x_i, enforcing the fact that it must either be True or False. If we choose the blue triples, we'll assign x_i to be True; if we choose the red triples, we'll assign x_i to be False (making $\bar{x}_i$ True). This means that we'll have n gadgets, one for each variable in ϕ.

Now, let us look at the clauses. Suppose we have a clause R; we will introduce c_R and d_R to represent this clause—we want these two elements to be matched in the final matching in order for the clause to be satisfied. To enforce this, suppose we have x_i as one of the variables in the clause.

We can represent the choice of x_i to be True as selecting the triple $(b_{x_i}^{(1)}, c_R, d_R)$ or the triple $(b_{x_i}^{(2)}, c_R, d_R)$; both would be included in our set of triples T. We have a similar logic if we had $\bar{x}_i$ instead; we'd have the triple $(b_{\bar{x}_i}^{(1)}, c_R, d_R)$ or the triple $(b_{\bar{x}_i}^{(2)}, c_R, d_R)$.

Notice that we have two choices here; we can't repeat any b_{x_i}, otherwise we may run into the issue of choosing b_{x_i} more than once—this is where the extra condition in $3\text{SAT}_{L\leq 2}$ comes in.

We're guaranteed that any literal appears in the 3SAT formula at most twice; this means that we'll never run out of b_{x_i}'s to assign to literals in clauses. The first time we see x_i in a clause, we can use $(b_{x_i}^{(1)}, c_R, d_R)$, and the second time we see x_i in a clause, we can use $(b_{x_i}^{(2)}, c_R, d_R)$.

As such, these new triples for each clause enforce the idea that some literal in a clause must be True; together with the first gadget to enforce the binary choice of any variable, we now have a way to enforce each clause in the 3SAT instance.

However, we still have one last problem—if any literal appears less than 2 times (i.e. only once), we'd have leftover b_{x_i}'s not matched by our satisfying assignment. How many b_{x_i}'s would we have left?

We know that in total, we have $4n$ different b_{x_i}'s; two each for both x_i and $\bar{x}_i$. We would have selected $2n$ of these from each variable (as each variable must be assigned either True or False, and each assignment selects two b_{x_i}'s), and we would have selected m of these from each clause (as each clause has exactly one selected triple, used to satisfy the clause).

This means that we have $4n - (2n + m) = 2n - m$ leftover b_{x_i}'s. To remedy this, we can introduce $2n - m$ new c and d elements, each of which appears in a triple with every single b_{x_i}. That is, we'd have c'_j and d'_j for $j = 1, \ldots, 2n - m$, and for each j we'd have $4n$ triples,

$$(b_{x_i}^{(1)}, c'_j, d'_j),\ (b_{x_i}^{(2)}, c'_j, d'_j),\ (b_{\bar{x}_i}^{(1)}, c'_j, d'_j),\ (b_{\bar{x}_i}^{(2)}, c'_j, d'_j),$$

for each of the n variables.

This allows for the 3D matching instance to clean up all of the leftover b_{x_i}'s after matching for each clause and variable, ensuring that we do have a valid 3D matching for any given $3\text{SAT}_{L\leq 2}$ instance.

As such, we've just shown that any $3\text{SAT}_{L\leq 2}$ formula ϕ can be transformed into a 3D matching problem through the use of gadgets and additional triples to enforce binary variables, all of the clauses, and any leftovers needed. With any $3\text{SAT}_{L\leq 2}$ instance as an input, we can create our set of triples T and our sets of elements B, C, and D to feed into 3D matching; this will produce a 3D matching that directly corresponds to a valid assignment to the variables in the original $3\text{SAT}_{L\leq 2}$ formula ϕ.

All together, we've shown that 3D matching is indeed NP-complete. □

20.4 Zero-One Equations

Given as input a binary matrix $\mathbf{A} \in \{0,1\}^{m\times n}$, we want to determine whether there exists an $\vec{\boldsymbol{x}} \in \{0,1\}^n$ such that $\mathbf{A}\vec{\boldsymbol{x}} = \vec{\mathbf{1}}$?

Example 20.9

If we are given as input

$$\begin{cases} x_1 \quad\quad + x_3 = 1 \\ x_1 + x_2 \quad\quad = 1 \\ x_1 + x_2 + x_3 = 1 \\ \quad\quad x_2 + x_3 = 1 \end{cases},$$

we'd have the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

The answer in this case would be no, because the last equation forces exactly one of x_2 and x_3 to be 1, and the first equation forces exactly one of x_1 and x_3 to be 1. If we take x_3 to be 1, then neither x_1 nor x_2 would be 1, and the second equation is not satisfied. If we take both x_1 and x_2 to be 1, then the second equation would still not be satisfied, as their sum is 2.

Theorem 20.10: ZOE is NP-complete

We claim that ZOE is NP-complete.

Proof. It should be clear that ZOE is in NP; we will focus on the reduction from 3D matching.

Here, we are given three groups B, C, and D of size n, and a set of triples of size m; we want to use ZOE to create a subset of triples where each element in every group is in exactly one triple.

The reduction is quite straightforward; we define a matrix $\mathbf{A}$ with $3n$ rows (one for each element of B, C, and D) and m columns (one for each triple).

An element A_{ij} is defined as

$$A_{ij} = \begin{cases} 1 & \text{if } i \text{ participates in } j\text{th triple} \\ 0 & \text{otherwise} \end{cases}.$$

Let's think about what this means.

Each row has the form $\sum_j x_{T_j} = 1$, which means that element x must appear in exactly one triple. This is exactly the condition on our output set, so if we can find a vector $\vec{v}$ where $\mathbf{A}\vec{v} = \vec{\mathbf{1}}$, then our solution gives us which triples we need.

As such, we've just shown that we can create an instance of ZOE to solve any 3D matching instance, and hence ZOE is NP-complete. □

20.5 Integer Linear Programming

Integer linear programming (ILP) is just like linear programming, but we are also allowed integrality constraints.

ILPs are of the following form:

$$\begin{aligned} \min / \max \quad & \vec{c}^T \vec{x} \\ \text{s.t.} \quad & \mathbf{A}\vec{x} \le \vec{b}, \\ & x_i \in \mathbb{Z}, \qquad \forall i \in S \end{aligned}$$

In our case, we can show that just a subset of instances of ILPs are NP-hard, showing the general problem of computing solutions to ILPs is NP-complete.

Theorem 20.11: ILP is NP-complete

We claim that ILP is NP-complete.

Proof. It should be clear that ILP is in NP; we will focus on the reduction from ZOE.

We'll only look at a subset of ILP problems, namely where $\vec{b} = \vec{1}$, S is the set of all variables (i.e. all variables must be integers), and we have the additional constraints $0 \le x_i \le 1$.

It is straightforward to see that this is almost exactly a ZOE problem. In ZOE, we're trying to find an $\vec{x}$ such that $A\vec{x} = \vec{1}$, and we know that we can split the equality constraint into two inequality constraints, fitting the form of the ILP as mentioned prior.

As such, since this subset of ILPs is NP-hard to compute feasible solutions for, the general problem of computing any solution for any ILP is also NP-hard, and ILP is NP-complete. □

11/16/2021

Lecture 21

Coping with NP-hardness

We've talked about how many problems are NP-hard and NP-complete, but today we'll be talking about how we cope with NP-hardness. There are problems that are NP-hard but are still necessary in day-to-day life.

For example, most scheduling problems and resource allocation problems are NP-hard; airports need to determine which planes should land where and when, and this scheduling is crucial to the functionality of airports and airlines—so what are these airlines doing?

There are a few different ways to deal with NP-hardness.

21.1 Dealing with NP-hardness

21.1.1 Exact Methods

The first group consists of exact methods.

Faster exponential-time algorithms There could be algorithms that improve on the exponential-time naive brute-force search; improving runtime from $O(2^n)$ to $O(1.5^n)$ or $O(1.1^n)$ is an improvement to how fast algorithms can run, and can make algorithms fast enough for the instances that we'll actually see.

Backtracking Brute-force search naively tries all possibilities; with backtracking, we can end a recursive brute-force search early if it is already going to fail.

For example, with 3SAT, if at some point in the recursion we find that one of the clauses is not satisfied, there's no point in continuing to recurse in this recursive branch. Ending this recursive case early can eliminate the need to check possibilities we already know are useless.

Branch and bound For optimization problems (ex. TSP), we can cutoff subtrees of recursive brute-force search if we can show that any completion of the partial solution will not beat our current best solution.

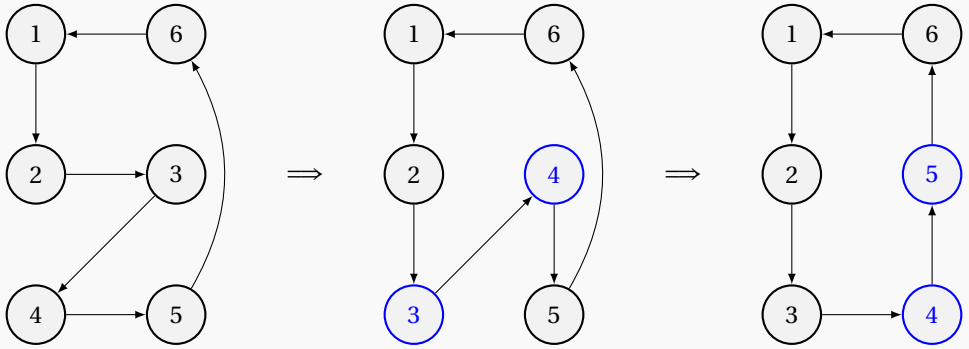
It turns out that branch and bound can be used to make an approximation algorithm; instead of proving that the partial solution will not beat our current best solution, we can instead determine whether our solution will be within some percentage of the optimal solution.

21.1.2 Heuristics

Local Search In a local search, we start with a random solution, then iteratively make "local" adjustments greedily to make it better until we can't anymore.

Example 21.1

With TSP, we could make local improvements as follows:



There is no guarantee that there is a sequence of local swaps that gets us to the global optimum. Sometimes there are (ex. guarantees that we'd be within some margin from the optimal solution), but we'd call those approximation algorithms.

Simulated Annealing In local search, we only make a local move if we get a better solution, but in simulated annealing, we have a small probability of accepting a local adjustment that increases cost. The purpose of this is to prevent us from being stuck at a local solution.

There isn't too much that one can prove about these kinds of random searches, but there are ways in which the search can be optimized and potentially give more accurate results.

For example, we can adjust this probability we accept a worse solution as the iterative algorithm progresses. In the beginning, we may allow more worse solutions, but as the algorithm progresses, we can decrease this probability. This is called an annealing schedule, and how one decreases this probability is one of the ways one can adjust the simulated annealing algorithm.

21.1.3 Approximation Algorithms

Approximation algorithms are essentially just heuristics with provable guarantees of closeness to OPT. This will only apply to optimization problems, as we want to find a solution as close to some optimal solution as possible.

For minimization problems, we want to find a solution of cost $\leq \alpha \cdot \mathsf{OPT}$. This $\alpha \in [1, \infty)$ is called an "approximation ratio".

For maximization problems, we want to find a solution of cost $\geq \mathsf{OPT}/\alpha$, where $\alpha \in [1, \infty)$ is still called the approximation ratio. Sometimes in literature $\beta \in (0, 1]$ is used instead, and we have $\geq \beta \cdot \mathsf{OPT}$.

21.2 Vertex Cover Approximations

Just as a reminder, the goal of vertex cover is to find a subset of vertices S such that each edge $e \in E$ is incident to at least one vertex $v \in S$, minimizing $|S|$.

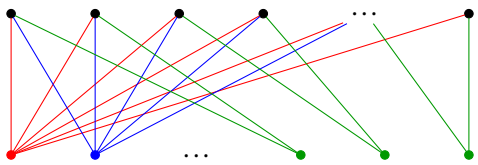
21.2.1 With Set Cover

One observation we can make is that vertex cover is a special case of set cover; the sets would just be singleton sets of vertices, and the edges become the universe elements.

As such, running the greedy set cover approximation algorithm on vertex cover, we have an $(\ln m)$-approximation, or a $(2\ln n)$-approximation. That is, we greedily take the vertex with the highest degree.

However, just because the greedy algorithm for set cover is an $(\ln m)$-approximation does not mean that this same algorithm is an $(\ln m)$-approximation for vertex cover; what if the cases that force greedy to have a $\ln m$ error cannot be realized as a vertex cover instance?

It turns out that this is not the case. Suppose we have the following graph, where we have one row of n vertices. We have a second row of vertices, where $\lfloor \frac{n}{n} \rfloor$ of them have degree n, $\lfloor \frac{n}{n-1} \rfloor$ of them have degree $n-1$, etc., and $\lfloor \frac{n}{2} \rfloor$ have degree 2.



We can see that OPT is of size n, since we have n vertices in the first row that we can select.

We can further see that the greedy algorithm will take all of the bottom vertices, in order, since it's going from largest degree to smallest degree. The sum is on the order of $n\sum \frac{1}{i} \approx n\ln n$, if we ignore the floor functions. Hence, the greedy solution is an $\ln n$-approximation of vertex cover.

21.2.2 With a Greedy Algorithm

The following algorithm is a better algorithm for vertex cover; we'll just be greedier.

```
S ← ∅
while ∃e ∈ E not covered by S:
    add both endpoints of e to S
return S
```

We claim that this algorithm is a 2-approximation of vertex cover; that is, $|S| \le 2 \cdot \text{OPT}$ for this algorithm.

Let's look at the edges e not covered in the while loop iterations that forced us to add to S; let's call this set of edges M.

Our claim is that M is a maximal matching.

Recall that a matching is a set of edges with no vertices in common. No two edges that trigger the while loop can have a vertex in common; if we took one of the edges, then the common vertex would already have been taken, and the second edge would not have triggered the while loop.

A maximal matching is a matching such that we cannot add to it to make it bigger. This differs from a *maximum* matching in that a maximum matching is the globally largest possible matching, whereas a maximal matching is local. It should also be straightforward to see that M is maximal, because if there were an edge we can add, then we'd already have added it, as it would have triggered the while loop.

Further, we claim that for all vertex covers S and all matchings M, $|S| \ge |M|$; this implies that $\text{OPT} \ge |M|$. This claim should be straightforward to see; all it's saying is that we need to take at least one incident vertex for each edge in M. There is no overcounting, since the edges have no vertices in common, and we're forced to take one incident vertex for each edge.

Putting this together, we can see that our algorithm produces an S and M such that $|S| \le 2|M|$ (namely, we have $|S| = 2|M|$). This means that we have

$$\text{OPT} \le |S| \le 2|M| \le 2 \cdot \text{OPT}.$$

As such, the vertex cover S we find from this algorithm is $|S| \le 2 \cdot \text{OPT}$, as desired.

21.2.3 With Linear Programming

Here's another 2-approximation of vertex cover; the idea behind this algorithm is the basis behind most other approximation algorithms—we can model this as an ILP.

Given a graph $G = (V, E)$, We can introduce variables x_v for each vertex $v \in V$, which are 0 if we do not take vertex v, and are 1 if we do take vertex v. Since we must take at least one vertex incident to each edge, we have the following ILP:

$$\begin{aligned} \min \quad & \sum_{v \in V} x_v \\ \text{s.t.} \quad & x_u + x_v \geq 1, \quad \forall (u, v) \in E, \\ & 0 \leq x_v \leq 1, \quad \forall v \in V, \\ & x_v \in \mathbb{Z}, \quad \forall v \in V \end{aligned}$$

The issue is that ILPs are NP-hard, as we've proven last lecture, so we shouldn't expect polynomial-time algorithms to solve this problem. However, we can relax these constraints to make it an LP rather than an ILP—we just remove the integrality constraint.

This is called "LP relaxation", and makes it so that we can get a solution in polynomial-time from an LP solver. However, the optimal solution that is returned could be (will be) fractional. As such, we will round the fractional solution to an integral solution—how we do the rounding is problem-specific. For vertex cover, we round normally to the nearest integer; that is, all $x_v \geq \frac{1}{2}$ will round up to 1, and all $x_v < \frac{1}{2}$ will round down to 0.

We claim that this rounding will always be a valid vertex cover. For each constraint, we have something like $x_u + x_v \geq 1$. This means that at least one of these variables must be $\geq \frac{1}{2}$, so we must choose at least one of the two vertices in our final solution.

Looking at cost/vertex cover size, we can first see that $\mathsf{OPT}(\text{LP}) \leq \mathsf{OPT}(\text{ILP})$. This is because the LP has less constraints; any solution to the ILP would also be a solution to the LP, and as such we can only get a smaller cost from the relaxation.

Further, the vertex cover S outputted by our LP relaxation and rounding has size $|S| \leq 2 \cdot \mathsf{OPT}(\text{LP})$. This is because we've at most doubled each x_v by rounding; some x_v's would be less than doubled (ex. going from 0.9 to 1), but this is definitely an upper bound to how much the cost could have changed.

Putting everything together, we have

$$|S| \leq 2 \cdot \mathsf{OPT}(\text{LP}) \leq 2 \cdot \mathsf{OPT}(\text{ILP}).$$

Hence, this algorithm is also a 2-approximation of vertex cover.

In general, this idea of LP rounding is a very common way to get approximations to optimization problems. The rounding itself can get a lot more complicated, and sometimes need to be non-deterministic.

21.3 Traveling Salesman Problem

Recall that the traveling salesman problem (TSP) takes an input of n cities, with distances d_{ij}. The goal is to start at city 1, visit every other city once, then return to city 1, where this tour is as short as possible.

We didn't show this in lecture, but TSP is NP-complete. For the longest time, the best known polynomial-time approximation was a 1.5-approximation (Held, Karp). In fact, only very recently a $(1.5 - 10^{-36})$-approximation was discovered, meaning that there is actually still room for improvement.

We won't go through these algorithm in class, but we'll show a 2-approximation.

The idea is to realize that any tour is a cycle C, and any cycle with the last edge removed is a path (which is a spanning tree).

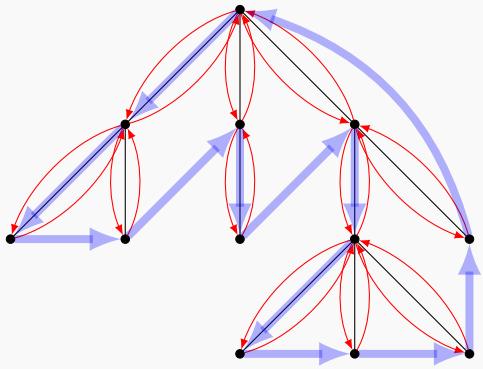
As such, OPT is always at least the cost of the minimum spanning tree (since it contains a spanning tree inside it).

The algorithm is thus:

- Compute the MST T^*
- Shortcut DFS tour of T^* (rooted arbitrarily)

Example 21.2

For example, suppose we have the following (arbitrarily rooted) MST:



The red path is the DFS tour of the MST, but in typical TSP formulations, we are not allowed to repeat vertices. As such, we take the blue path as our final path—this is the shortcut DFS tour. In the shortcut tour, we only visit vertices that have not been visited before as we go through the DFS.

We can see that the raw DFS tour equal to $\text{cost(DFS)} = 2 \cdot \text{cost}(T^*) \leq 2 \cdot \text{OPT}$.

At this point, we need to utilize the triangle inequality—which is only true in the specific case of *metric* TSP; that is, our distances are consistent with real distances in a metric space.

Utilizing the triangle inequality, taking shortcuts will only ever decrease the total distance, and as such

$$\text{cost(shortcut)} \leq \text{cost(DFS)} = 2 \cdot \text{cost}(T^*) \leq 2 \cdot \text{OPT}.$$

Hence, this algorithm is a 2-approximation of metric TSP.

This algorithm may not be a 2-approximation for *general* TSP though—that is, in cases where distances do not satisfy the triangle inequality.

However, in the general case, we can actually show that no efficient approximation is possible:

Theorem 21.3

For any polynomial-time computable function $f(n)$, there is no $f(n)$-approximation of TSP unless $\mathsf{P} = \mathsf{NP}$.

11/18/2021

Lecture 22

Randomized Algorithms

Today we'll be talking about randomized algorithms. With randomized algorithms, the inputs are not random, but the algorithm itself makes random decisions.

One such common example of a randomized algorithm is with polling. It's oftentimes infeasible to poll the entire population to get a measure of overall preference, so instead it's common to take random samples of the population instead. In this case, our "algorithm" is trying to solve the problem of counting, and randomized sampling allows us to get an approximate count in less time.

There are two main categories of randomized algorithms:

1. Las Vegas

   In a Las Vegas randomized algorithm, the algorithm always produces a correct answer, but the runtime is a random variable. One such example is Quick Sort; we can only say that the *expected* runtime is fast (i.e. $O(n \log n)$), but with some small probability it could actually take $O(n^2)$ time.

2. Monte Carlo

   In a Monte Carlo randomized algorithm, the runtime is always fixed, but the correctness is a random variable. Polling is an example of a Monte Carlo randomized algorithm; we're always taking some fixed number of samples, but we aren't always going to be exactly correct.

In this sense, we can express Las Vegas random algorithms as saying

$$(\forall x)(\mathbb{E}_r[\text{Runtime}(\mathcal{A}(x,r))] \leq \varepsilon),$$

where $\mathcal{A}$ is our algorithm, taking in an input x and a random string r (to be used as our source of randomness; that is, we can think of it as a preprocessed sequence of calls to some `rand()` function). This essentially says that for all inputs x, the expected runtime for our algorithm is always less than some small ε.

We can express Monte Carlo algorithms in a similar fashion:

$$(\forall x)(\mathbb{P}_r(\mathcal{A}(x,r) \text{ is correct}) \geq 1-\varepsilon).$$

This essentially says that for all inputs x, the probability that the algorithm produces a correct result is very high.

22.1 Probability Recap

Here's some basic review of probability that we'll need for the rest of lecture.

We'll be looking at discrete random variables today, taking on integer values.

We define the expectation of a random variable X to be

$$\mathbb{E}[X] = \sum_{k=-\infty}^{\infty} k \cdot \mathbb{P}(X=k) = \sum_{k=1}^{\infty} \mathbb{P}(X \geq k),$$

where the last equality holds if X is nonnegative.

We also have linearity of expectation; for all scalars $\alpha, \beta \in \mathbb{R}$ and random variables X and Y, we have

$$\mathbb{E}\big[\alpha X + \beta Y\big] = \alpha \mathbb{E}[X] + \beta \mathbb{E}[Y].$$

Lastly, we have Markov's inequality, which says that if X is nonnegative, then

$$\mathbb{P}(X > \lambda) < \frac{\mathbb{E}[X]}{\lambda}.$$

22.2 Quick sort

Quick sort is a Las Vegas randomized algorithm, which does comparison-based sorting. WLOG, we will assume that all array elements are distinct.

In quick sort, we have an input array of size n. We pick a uniformly random element of A, called the "pivot", and compare with every other element in the array, and partition based on which elements are less than or greater than this pivot. We then recursively sort the left and right arrays, and concatenate them at the end.

Intuitively, if we choose a random pivot, we'd expect this pivot to be the median. This means that the partition would end up splitting the array into two subarrays of size $\frac{n}{2}$, and the runtime recurrence turns into $T(n) = 2T(\frac{n}{2}) + O(n)$, which is $T(n) = O(n \log n)$.

However, this analysis isn't accurate/correct; the $T(\frac{n}{2})$ is actually a random variable, so we'd need to do some probabilistic analysis, which is what we'll do.

A first observation is that our runtime is proportional to the number of comparisons. As such, it's sufficient to count the number of comparisons in order to get a measure of runtime.

A second observation is that we will never compare the same two elements twice. This is because one of the elements in the comparison will always be the pivot, and after the partition, the pivot no longer exists as a part of the subproblem.

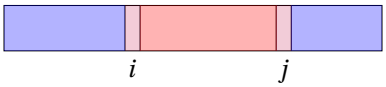
These two observations combined imply that the total runtime is proportional to $C \cdot \sum_{i<j} X_{ij}$, where X_{ij} is an indicator RV for when the ith smallest element is ever compared with the jth smallest.

This then means that the expectation of runtime

$$\mathbb{E}[\text{runtime}] \approx \mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}[X_{ij}].$$

Since $\mathbb{E}[X_{ij}] = \mathbb{P}(X_{ij} = 1) = \mathbb{P}(i\text{th smallest compared with } j\text{th smallest})$, we can replace this in the sum.

As such, the only thing we need to calculate is the probability that the ith smallest element is compared with the jth smallest element. If we visualize our array of elements (pictured in sorted order here for clarity; the array isn't actually in sorted order), we have



If we choose our pivot to be anywhere in the blue region (i.e. $p < i < j$ or $i < j < p$), we're essentially just delaying the choice of whether we compare i and j. If we choose our pivot to be anywhere in the red region (i.e. $i < p < j$), we will never compare i and j; we'd compare i with the pivot, j with the pivot, and then split them up into two separate partitions.

The only time in which we will compare i and j is if our pivot is either i or j. Given that we did not choose our pivot in the blue region (that is, at this moment in time, we either compare i and j, or we do not ever compare i and j), the probability of comparing i and j now is $\frac{2}{j-i+1}$.

This means that

$$\begin{aligned}
\mathbb{E}[\text{Runtime}] &\le C \cdot \sum_{i<j} \frac{2}{j-i+1} \\
&= 2C \cdot \sum_{i<j} \frac{1}{j-i+1} \\
&= 2C \cdot \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j-i+1}
\end{aligned}$$

To understand what this sum actually evaluates to, we can look at each case:

$$\begin{aligned}
i = 1 &: \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \\
i = 2 &: \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} \\
&\vdots \\
i = n-1 &: \frac{1}{2}
\end{aligned}$$

If we add all of these together, we have that

$$\begin{aligned}
\mathbb{E}[\text{Runtime}] &\le 2C \cdot \left((n-1)\frac{1}{2} + (n-2)\frac{1}{3} + \cdots + 1 \cdot \frac{1}{n}\right) \\
&\le 2C \cdot n\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) \\
&\le 2C \cdot n \ln n
\end{aligned}$$

The last inequality is due to the fact that the harmonic series is bounded by $\ln n$.

22.3 Freivald's Algorithm

Freivald's algorithm is an algorithm to verify matrix multiplication. We're given two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$, and a matrix $\mathbf{C}$ that we want to verify whether it is actually $\mathbf{C} = \mathbf{AB}$, in faster time than multiplying $\mathbf{AB}$.

We further want to ensure that we can catch one-element errors; this means that we can't just sample entries and check, as we won't be able to find these small errors.

In this algorithm, we pick vectors $\vec{x}_1, \ldots, \vec{x}_T \in \{0,1\}^n$ independently and uniformly at random. We then iterate through all these vectors and check whether $\mathbf{AB}\vec{x}_i \stackrel{?}{=} \mathbf{C}\vec{x}_i$. If this is not equal, then we return false, and otherwise once we've checked all T equalities, we return true.

Matrix-vector multiplication only takes $O(n^2)$ time, and we can compute the RHS with one matrix-vector multiplication, and we can compute the LHS with two matrix-vector multiplications (i.e. $\mathbf{AB}\vec{x}_i = \mathbf{A}(\mathbf{B}\vec{x}_i)$). This means that our runtime is $O(n^2 T)$.

To analyze this algorithm, let us denote $\mathbf{AB} - \mathbf{C} = \mathbf{D}$. We want to know whether $\mathbf{D} = 0$.

Our claim is that if $\mathbf{D} \neq 0$, then $\mathbb{P}\left(\mathbf{D}\vec{x} = \vec{\mathbf{0}}\right) \leq \frac{1}{2}$ for each $\vec{x} \in \{0,1\}^n$. This implies that the probability that our algorithm is incorrect is at most $\frac{1}{2^T} \leq p$, with the RHS if $T := \left\lceil \log_2 \frac{1}{p} \right\rceil$.

We will now prove this claim. If $\mathbf{D} \neq 0$, then there exists some i, j such that $D_{ij} \neq 0$. For notation, let us denote $\vec{\boldsymbol{d}}_j$ as the jth column of $\mathbf{D}$.

Note that in this case, if we found an $\vec{x}$ such that $\mathbf{D}\vec{x} = \vec{\mathbf{0}}$, then $\mathbf{D}\left(\vec{x}|_j\right) \neq \vec{\mathbf{0}}$, where $\vec{x}|_j$ means that we flip the jth entry of $\vec{x}$.

To see why, notice that we can write

$$\mathbf{D}\vec{x} = \sum_{k=1}^{n} x_k d_k = \left(\sum_{k \neq j} x_k d_k\right) + x_j d_j.$$

Suppose that x_j was originally 0. This means that $x_j \vec{\boldsymbol{d}}_j$ was zero in $\mathbf{D}\vec{x}$, but after flipping x_j, we now have a nonzero value of $x_j \vec{\boldsymbol{d}}_j$, since $\vec{\boldsymbol{d}}_j$ is nonzero by our assumptions. As such, we now have $\mathbf{D}\left(\vec{x}|_j\right) = \vec{\boldsymbol{d}}_j \neq \vec{\mathbf{0}}$.

Suppose that x_j was originally 1. This means that the entire sum, including $x_j \vec{\boldsymbol{d}}_j \neq \vec{\mathbf{0}}$ was zero; after flipping x_j, we are now *excluding* some nonzero term, meaning that we're subtracting some value from our sum. As such, we now have $\mathbf{D}(\vec{x}|_j) = -\vec{\boldsymbol{d}}_j \neq \vec{\mathbf{0}}$.

Now, suppose we pair $\{0,1\}^n$ into 2^{n-1} pairs, where each $\vec{x}$ is paired with $\vec{x}|_j$. For each $\vec{x}$ in the pair, evaluating $\mathbf{D}\vec{x}_1$ and $\mathbf{D}\vec{x}_1$, at most one of these results will evaluate to $\vec{\mathbf{0}}$, by our previous claim; if it evaluated to zero for $\vec{x}_1$, then it must evaluate to a nonzero vector for $\vec{x}_2$, and vice versa.

Let us denote E as the set of vectors $\vec{x}$ that cause $\mathbf{D}\vec{x} = \vec{\mathbf{0}}$. We know from our previous results that $|E|$ is at most the number of pairs, or 2^{n-1}. This means that the probability that we choose one of these vectors is equal to

$$\mathbb{P}\left(\mathbf{D}\vec{x} = \vec{\mathbf{0}}\right) = \frac{|E|}{2^n} \leq \frac{2^{n-1}}{2^n} = \frac{1}{2}.$$

(As a side note, since we don't know j, we can't just pick one of these pairs and check both vectors in the pair; there are n possible j's, and this would turn into an n^3 algorithm if we checked all of them.)

22.4 Karger's Contraction Algorithm

Karger's Contraction Algorithm solves the global min-cut problem. For the global min-cut problem, we are given a weighted, undirected graph $G = (V, E)$, and our goal is to find the global min-cut. That is, we want to find a cut of the graph such that the total edge weight across the cut is minimized. (This differs from min s-t cut in that we don't care which vertices are on which side).

Note that we can reduce this to s-t min-cut by calculating the min-cut for $s = 1$ and t ranging from 2 through n. This means that we need $n-1$ calls to s-t min-cut to calculate the global min-cut.

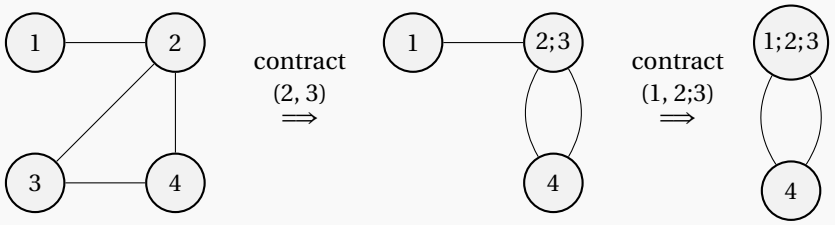
Karger's contraction algorithm applies for the unweighted case (i.e. all cuts are of the same weight); it's very simple to say and implement, though it isn't the most efficient.

In this algorithm, we take a random edge and contract it. That is, we "glue" the two incident vertices together, but keep all of the other edges incident to these vertices (we'll likely have duplicate edges between a pair of vertices now, but we'll keep those—in that sense, this is now a multi-graph).

When we have two (multi-)vertices left (i.e. after $n-2$ contractions), we output the resulting cut.

Example 22.1

Here's an example:



Here, our cut has partitions $\{1,2,3\}$ and $\{4\}$.

Our claim is that if we fix a particular min-cut S, $V \setminus S$, then the probability that the contraction returns S is at least $\frac{1}{\binom{n}{2}}$.

This isn't that good. To fix this, we will run the algorithm R times independently, and return the best one. The probability that we fail to find the min-cut is then

$$\mathbb{P}(\text{fail to find min-cut}) \le \left(1 - \frac{1}{\binom{n}{2}}\right)^R \le e^{-R/\binom{n}{2}}.$$

Choosing $R = \left\lceil \binom{n}{2} \log_2 \frac{1}{p} \right\rceil$, then we guarantee that this probability is $\le p$.

This algorithm runs in approximately $O(n^4)$ time, since we take $O(n^2)$ time to do one run of the algorithm (linear time contractions, n times), and we need to do $O(n^2)$ runs.

One consequence of the previous claim (that the probability that the contraction returns a fixed min-cut S is $\frac{1}{\binom{n}{2}}$) is that no graph can have more than $\binom{n}{2}$ min-cuts.

This bound is also tight with the n-cycle; every single min-cut is of size two, so we can just choose 2 of the vertices to be the boundary of the min-cut.

We will now prove the previous claim.

Theorem 22.2

If we fix a particular min-cut S, $V \setminus S$,

$$\mathbb{P}\big(\text{output } S, V \setminus S\big) \ge \frac{1}{\binom{n}{2}}.$$

Proof. Let k denote the number of edges crossing $(S, V \setminus S)$.

We can observe that the algorithm will return the desired cut if and only if it never contracts any edge

crossing the cut.

If we look at the first round, the probability that we contract an edge crossing the cut is $\frac{k}{m}$, because we have k edges crossing the cut, and m total edges.

We claim that $m \geq \frac{nk}{2}$. If we look at cuts of the form $\{u\}$, $V \setminus \{u\}$ for all $u \in V$, we can see that the number of edges crossing this cut is $\deg(u)$. By definition, the size of the min-cut must be at most as large as $\deg(u)$. That is $\deg(u) \geq k$.

As such, if we sum over the degrees of all vertices, we have

$$2m = \sum_{u \in V} \deg(u) \geq nk.$$

The left equality is due to the handshaking lemma; each edge is counted exactly twice if we sum over all degrees of all vertices. The right inequality is simply what we derived prior; we have n vertices, and each of them must have a degree at least as large as the min-cut. This directly implies that $m \geq \frac{nk}{2}$.

Using this fact, we can see that the probability that we contract an edge crossing the cut is

$$\mathbb{P}\big(\text{contract edge across min-cut}\big) = \frac{k}{m} \leq \frac{k}{\left(\frac{nk}{2}\right)} = \frac{2}{n}.$$

This means that the probability that we *don't* contract an edge across the cut is at least $1 - \frac{2}{n}$.

Given that we do not contract any edge across the min-cut in the first round, this means that in the second round, we now only have $n-1$ vertices.

If we generalize, the probability that we don't contract any edge across the cut in the ith round, given that we don't contract any such edge in any of the previous rounds is $1 - \frac{2}{n-i+1}$.

This means that the probability that we never contract any edges across the cut in any round is

$$\left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right) \cdot \left(1 - \frac{2}{n-2}\right) \cdots = \frac{\cancel{n-2}}{n} \cdot \frac{\cancel{n-3}}{n-1} \cdot \frac{\cancel{n-4}}{\cancel{n-2}} \cdot \frac{\cancel{n-5}}{\cancel{n-3}} \cdots \frac{\cancel{3}}{\cancel{5}} \cdot \frac{2}{\cancel{4}} \cdot \frac{1}{\cancel{3}} = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}.$$

□

Some brief points of history:

- Karger: $O(n^4)$ time (for success 99%)
- Karger-Stein: $O(n^2 \log^2 n)$
- Karger: $O(m \log^2 n)$
- Li (2021): $O(m^{1+o(1)})$ deterministically

11/23/2021

Lecture 23

Hashing

Today, we will look at the dictionary problem (i.e. `dict` in Python).

The problem is that we want to maintain a database of (key, value) pairs, where the keys are in the universe of $\{0, \ldots, U-1\}$, and we have a database of size n (the number of entries in the database).

There are two versions of this problem. In the static version, the entire database is given up front (no insertions/deletions), and we must support `query(k)`, which returns `null` if the key is not in the database, and return the

associated value if the key appears in the database. The dynamic version also supports insertion and deletion, as well as the previous query operation.

There are a few solutions.

- (static/dynamic) Store the database in an array of size U.

  We have constant $O(1)$ time insert, delete, and query. The downside is that this takes a lot of memory; we need to reserve space for U elements in the array.

- (dynamic) Store the database in a balanced binary search tree.

  We have $O(n)$ memory, and $O(\log n)$ time per operation, and is deterministic.

- (static) Store the data in an array sorted by key.

  Our query here is just binary search, which takes $O(\log n)$ time.

A major open question is whether there exists a solution for a dynamic dictionary that gives $O(1)$ time per operation, $O(n)$ memory, all deterministically.

We'll start with the dynamic case.

23.1 Hashing with Chaining

We will pick a random function $h : \{0, 1, \ldots, U-1\} \to \{0, 1, \ldots, m-1\}$, and we will have a hash table with m entries. Each entry in the table will be a pointer to the head of a linked-list. That is, `A[i]` is a doubly-linked list storing all pairs (k, v) such that $h(k) = i$.

For insertion of a pair (k, v), we compute $h(k)$, scan the corresponding linked list at `A[`$h(k)$`]`, and add it to the end if it is not present. For deletion, we do the same hash, and splice the element out of the linked list if it is found. For query, we do the same hash, and just return the corresponding value in the linked list if the key is found.

(One side note that we will address later: let's think about what this h function is actually doing. We want to be able to know these mappings later on in the future in order to recover the same data, so we'd need to store the h function somewhere. However, this function would need to know where each of the U elements in the universe should go—wouldn't this require U memory, making our solution just an over-complicated version of the first solution with an array? It turns out that we can actually store h efficiently!)

Let's analyze this data structure. Our claim is that the expected time for an operation is $O(T_h + \frac{n}{m})$, where T_h is the time it takes to evaluate the hash function (which is constant if we just store everything in an array), where m is the size of the hash table, and n is the size of the database. This implies that to get optimal runtime, we'd want the size of the hash table and the size of the database to be roughly equal (or $m > n$), and we'd get approximately constant time.

Remember that to query an item, we need to hash the key, and then traverse the linked list. This means that

$$\mathbb{E}\big[\text{Time to query } x\big] \le \mathbb{E}\big[T_h + C \cdot (\text{Size of LL containing } x)\big] = T_h + C \cdot \mathbb{E}\big[\text{Size of LL containing } x\big].$$

Suppose the keys in the database are $j_1, j_2, \ldots, j_n$, and let Z_i be the indicator for whether $h(j_i) = h(x)$. This means that the size of the linked list containing x is just $\sum_{i=1}^{n} Z_i$. By linearity, we have

$$\begin{aligned}
\mathbb{E}\big[\text{Size of LL containing } x\big] &= \mathbb{E}\left[\sum_{i=1}^{n} Z_i\right] \\
&= \sum_{i=1}^{n} \mathbb{E}[Z_i] \\
&= \sum_{i=1}^{n} \mathbb{P}\big(h(j_i) = h(x)\big) \\
&= \sum_{i=1}^{n} \sum_{t=1}^{m} \mathbb{P}\big((h(j_i) = t) \wedge (h(x) = t)\big)
\end{aligned}$$

Each of these probabilities evaluates to $\frac{1}{m^2}$, since we have a $\frac{1}{m}$ chance that any key gets assigned to a specific t hash value. As such, the expected size of the linked list containing x is $nm \cdot \frac{1}{m^2} = \frac{n}{m}$.

Putting this together, the expected query time for an element x is $T_h + C \cdot \frac{n}{m}$, or $O(T_h + \frac{n}{m})$, as desired.

Notice in the previous proof that there's only one place that we used the fact that h is random; it's the part where we simplified $\mathbb{P}\big((h(j_i) = t) \wedge (h(x) = i)\big) = \frac{1}{m^2}$. This means that if we have a function h that isn't actually truly random, but does satisfy this probability, then it's good enough.

This is the observation that will allow us to store the hash function h efficiently in way less than U memory, which we will talk about next.

23.2 Universal/k-wise Independent Hash Families

Definition 23.1: k-wise Independent Hash Family

A set $\mathcal{H}$ of functions mapping $\{0,\dots,U-1\}$ into $\{0,\dots,m-1\}$ is *k-wise independent* if

$$(\forall x_1 \neq x_1 \neq \cdots \neq x_k)\big(\forall y_1, y_2, \dots, y_k\big)\left(\mathbb{P}\big((h(x_1) = y_1) \wedge \big(h(x_2) = y_2\big) \wedge \cdots \wedge \big(h(x_k) = y_k\big)\big) = \frac{1}{m^k}\right).$$

Here, note that this probability is over a uniform choice of h from the set $\mathcal{H}$; each function is deterministic, but the choice of the function is uniformly random.

Definition 23.2: Universal Hash Family

$\mathcal{H}$ is *universal* if $\forall x_1 \neq x_2, \mathbb{P}(h(x_1) = h(x_2)) \leq \frac{1}{m}$.

Note that 2-wise independence $\implies$ universal, and our earlier proof only needed our hash function to be universal (i.e. drawn from a universal hash family).

Example 23.3

Suppose we have $\mathcal{H}$ be the set of all functions mapping $\{0,\dots,U-1\}$ to $\{0,\dots,m-1\}$.

We can see that this family of functions is k-wise independent for all k; any hash function would be truly random.

Specifying some $h \in \mathcal{H}$ takes $\lceil \log_2 |H| \rceil$ bits, and we have $|\mathcal{H}| = m^U$. This means that we need $O(U \log m)$ bits to specify such a function. These $\lceil \log_2 |H| \rceil$ bits are called the *seed*.

This should make sense, since it's essentially just an array, mapping each of the U inputs to one of the m outputs.

Example 23.4

We can actually specify another hash family that has a smaller seed length.

Suppose $U = m = p$, where p is a prime. Our earlier solution requires $p \log p$ bits, but we claim that this hash family can get away with $\log p$ bits.

Let $\mathcal{H} = \{ax + b \mod p \mid a, b \in \{0,\dots,p-1\}\}$. We have $|\mathcal{H}| = p^2$, which means that $\log|\mathcal{H}| = O(\log p)$.

The claim is that the $\mathcal{H}$ here is 2-wise independent, which we will not prove here, as it involves topics in abstract algebra. The general gist is that since elements in $\mathcal{H}$ are described by a linear function, any two points will define the line, meaning a third point will be completely determined; two or less points chosen

from this set will be random.

The main takeaway here is that we can create a hash family that is 2-wise independent, but only requires a small amount of space to store and specify.

23.3 Static Dictionaries

In the static case, we can still use hashing with chaining, giving us an expected query time of $O(1)$. However, our goal is to have $O(1)$ query guaranteed.

To do this, we utilize *perfect hashing*: that is, an h such that $\forall k_1 \neq k_2$ in the database, then $h(k_1) \neq h(k_2)$.

The idea here is to use the "inverse" birthday paradox (an unofficial term); if there are very few people in a room, there probably are no two people that share the same birthday.

For our purposes, if the number of keys we have in our database is much smaller than the size of our database, then we're good.

Formally, let us look at the probability that there exists a collision for a random h from our hash family. We have

$$\mathbb{P}(\exists \text{collision}) = \mathbb{P}(\#\text{ collisions} \geq 1) \leq \frac{\mathbb{E}[\#\text{ collisions}]}{1}.$$

If we let Z_{ab} be the indicator for whether $h(j_a) = h(j_b)$, then the number of collisions is $\sum_{a<b} Z_{ab}$. As such, by linearity, we have

$$\mathbb{E}[\#\text{ collisions}] = \mathbb{E}\left[\sum_{a<b} Z_{ab}\right] = \sum_{a<b} \mathbb{E}[Z_{ab}] = \sum_{a<b} \mathbb{P}\big(h(j_a) = h(j_b)\big).$$

Since our hash family is 2-wise independent, this probability is just $\frac{1}{m}$. Further, there are $\binom{n}{2}$ elements in the set, so our final probability is $\frac{\binom{n}{2}}{m}$, and we know that this is $\leq \frac{1}{2}$ if we pick $m = n^2$.

This means that if we pick our hash table size to be at least n^2, then we'll probably not have any collisions. If we do have collisions, then we'll just keep trying other hash functions until no collisions happen (since the probability of no collision is at least $\frac{1}{2}$, this is just a Geom($\frac{1}{2}$) distribution, meaning we only need to try 2 times in expectation before we have no collisions—more importantly, this is a constant number of tries in expectation). After we find an h in pre-processing, we'd use this h to get our deterministic $O(1)$-time query.

12/1/2021

Lecture 24

Streaming Algorithms

There are two ways to view what streaming algorithms are.

- Algorithms that make one pass over data

  For example, with a search engine, we process the data as they come in.

- Dynamic data structures

  A data structural problem is one where we have some data and we want to efficiently layout the data to allow efficient queries. Dynamic data structures just mean that you can support insertion/deletion, etc.

In essence, when we refer to streaming algorithms, we care about minimizing memory usage. That is, we want $o(n)$ memory, or asymptotically less than linear memory.

As motivation, suppose we're looking at a network, and the packets sent over the network. TO analyze this data, we would like to not store every single packet that comes over the network; we'd like to have some algorithm that processes packets as they come through, and analyze and answer queries on the spot.

There are several types of streaming problems:

- Counting problems (which we'll cover today)
- Graph problems

  In a social network, the graph is continuously changing, and we'd like to query things like finding a path from one vertex to another, or finding a spanning forest, etc. It turns out that we can do things like create a spanning forest in $O(n\log^3 n)$ bits (which we won't be looking at today).
- Linear algebraic problems (ex. low-rank approximation, regression)
- Geometric problems
- Set maintenance

24.1 Counting

We want to maintain a counter N subject to three operations:

- `init()`: $N \leftarrow 0$
- `incr()`: $N \leftarrow N+1$
- `query()`: return N

We want to minimize memory usage.

The trivial algorithm is to maintain a single counter with N represented in binary (for example). This needs $\Theta(\log N)$ bits.

What if we know that our counter will never exceed t? Can we beat $\log t$ bits? No—it's impossible. Since we have t different numbers, we need at least t different counter values, and we need $\log t$ bits to store each of these counter values.

24.2 Approximate Counting

Suppose our counter value got to a really big number, i.e. 500 digits. Naively, this would require 500 digits of memory ($\approx 500\lceil\log_2 10\rceil$ bits).

As an approximation, suppose we just store $\lceil\log_{10} N\rceil$. If we want the answer to 1% error, we can store $\lceil\log_{1.01} N\rceil$.

Generalized, if we want to know the value up to $1+\varepsilon$ accuracy, the stored value will be $\Theta\left(\frac{\log N}{\varepsilon}\right)$, and we'd need $O(\log\log N - \log\varepsilon)$ bits, which is much better than the naive $\log N$ bits to store the exact number (which gets better for larger numbers).

However, this doesn't actually solve the problem; we don't know the number, so we can't increment anything. We don't know when we should go into the next number in the counter.

The idea here is to use randomness, and the algorithm we'll look at is Morris's algorithm.

(As an aside: why do we even care about these kinds of approximation counting algorithms, if we have lots of memory in modern-day computers? Morris, back in the 70's, wanted to keep track of occurrences of trigrams in a document (for spell-checking). Since there are approximately 26^3 different trigrams (more, including other characters), Morris found that he could only devote 8 bits per counter, but he needed to keep track of numbers more than 255; this is what led him to develop his algorithm, which we'll talk about today. However, a more modern application is with Redis, which is an in-memory database software. To implement caches (specifically, for the eviction policy), there needs to be a counter for how many times any given key was accessed, and this is proportional to the number of keys in the database, which could be incredibly large. Redis uses the Morris counter to keep track of those counters efficiently.)

A first idea is to not store N in memory, but instead store X. We'd have

- `init()`: $X \leftarrow 0$
- `incr()`: if `rand() % 2 == 0`, $X \leftarrow X+1$
- `query()`: return $2X$.

There are two problems here; we have a big error for small N (if we only increment once, then we either return 0 or 2, neither of which is within 1% of N), and we only save 1 bit of memory.

To save more memory, suppose we have

- `init()`: $X \leftarrow 0$
- `incr()`: with probability $\frac{1}{2^X}$, $X \leftarrow X+1$, otherwise do nothing
- `query()`: return $2^X - 1$

Here, it's very unlikely to increment X. To see why we return 2^X-1, we claim that $\mathbb{E}\left[2^{X_n}\right] = n+1$, where X_n is the value of X after the first n increments. If this is true, then we'd have $\mathbb{E}\left[2^X-1\right] = n+1-1 = n$, so this is an unbiased estimator for n.

To prove this claim, we proceed by induction on n. Our base case for $n=0$, we have $X_0 = 0$, and $2^{X_0} = 1 = n+1$.

In our induction step, we want to show the case for $n+1$. We have by total expectation

$$\begin{aligned}
\mathbb{E}\left[2^{X_{n+1}}\right] &= \sum_{j=0}^{\infty} \mathbb{E}\left[2^{X_{n+1}} \mid X_n = j\right] \mathbb{P}\left(X_n = j\right) \\
&= \sum_{j=0}^{\infty} \left(\frac{1}{2^j} \cdot 2^{j+1} + \left(1 - \frac{1}{2^j}\cdot 2^j\right)\right) \mathbb{P}\left(X_n = j\right) && \text{(total probability)} \\
&= \sum_{j=0}^{\infty} \left(2 + 2^j - 1\right) \mathbb{P}\left(X_n = j\right) \\
&= \sum_{j=0}^{\infty} \mathbb{P}\left(X_n = j\right) + \sum_{j=0}^{\infty} 2^j \, \mathbb{P}\left(X_n = j\right) \\
&= 1 + \mathbb{E}\left[2^{X_n}\right] && \text{(first term is summing over all possibilities of } X_n\text{)} \\
&= 1 + n + 1 = n+2
\end{aligned}$$

This then proves our claim for $n+1$.

To continue, we'd need to use Chebyshev's inequality, i.e. for all $\lambda > 0$,

$$\mathbb{P}(|X - \mathbb{E}[X]| > \lambda) < \frac{\text{Var}(X)}{\lambda}.$$

Another thing to keep in mind is that $\text{Var}(X) = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2$.

Coming back to our problem, we have an estimator Z for N (for us, $Z = 2^X - 1$). We've already shown that $\mathbb{E}[Z] = N$.

As a next step, we'd like to determine

$$\mathbb{P}(|Z - N| > \varepsilon N) < \frac{\text{Var}(Z)}{\varepsilon^2 N^2}.$$

Our hope here is to have a small $\text{Var}(Z)$, i.e. we want $\frac{\text{Var}(Z)}{\varepsilon^2 N^2} < p$ for any p. If we did the calculation for the variance, we'd have that $\text{Var}(Z) = \frac{1}{2}N^2 - \frac{1}{2}N - 1$.

However, we have a problem here; our algorithm doesn't even take ε into account, so there's nothing to cancel out the ε^2. This means that we'd need to modify our algorithm to take ε into account, and get this variance down.

There are two ways to decrease the variance:

- We can run R independent copies of the algorithm in parallel, then use the average of all of the counters $Z = \frac{1}{R}\sum_{i=1}^{R} Z_i$ as our estimator.

  The expectation of this average is still N, but we have a lower variance, i.e. $\text{Var}\left(\frac{1}{R}\sum_{i=1}^{R} Z_i\right) = \frac{1}{R}\text{Var}(Z_i)$. If we choose $R = \frac{1}{\varepsilon^2 p}$, then we'd ensure that

  $$\mathbb{P}(|Z - N| > \varepsilon N) < \frac{\text{Var}(Z)}{\varepsilon^2 N^2} = \frac{\text{Var}(Z_i)}{R\varepsilon^2 N^2} \approx \frac{N^2}{\frac{1}{\varepsilon^2 p}\cdot \varepsilon^2 N^2} = p.$$

  Here, we used the fact that $\text{Var}(Z_i) \approx N^2$, as before.

- There are two observations we can make use of here. If we increment with probability 0.5^X, we have low memory, but high variance. On the other hand, if we increment with probability 1^X, then we have high memory, but low variance.

  Intuitively, there should be some continuous exchange between memory and variance as we change the probability. This means that if we increment with probability $\frac{1}{(1+a)^X}$, we can show that

  $$\mathbb{E}\left[\frac{(1+a)^X - 1}{a}\right] = N.$$

  This means that $Z = \frac{(1+a)^X - 1}{a}$ is our unbiased estimator. Further, we can also show that

  $$\text{Var}(Z) \le \frac{aN(N-1)}{2}.$$

  Letting $a \approx 2\varepsilon^2 p$, we'd get from Chebyshev's that $\mathbb{P}(|Z - N| > \varepsilon N) < p$.

  We can also show that the memory consumption is close to $O(\log\log N + \log\frac{1}{a})$, and with our choice of a, we need $O(\log\log N + \log\frac{1}{\varepsilon} + \log\frac{1}{p})$ bits.

24.3 Unique Counting

The distinct/unique counting problem is similar to the generic count problem, except we want to count the number of distinct items we see, in a stream of m items (possibly with repeats) coming from a universe of size $\{1, \ldots, n\}$.

There exists an algorithm using $O(\frac{1}{\varepsilon^2} + \log n)$ bits of memory with success probability of 99%. We can also prove that there is no better algorithm; this is optimal.

We won't be looking at this specific algorithm today, but we'll briefly go over an idealized version of the algorithm, and explain its connection to hashing.

We have `init()` that sets $X \leftarrow 1$, and we pick $h : [n] \to [0, 1]$, i.e. h maps $[n]$ to a real number between 0 and 1.

We have `update(i)` as $X \leftarrow \min(X, h(i))$. That is, we'll always be storing the smallest hash value seen so far. It can be shown that the expected value of X is equal to $\frac{1}{N+1}$ (since in expectation the hashes should be equally spaced in $[0, 1]$).

This means that our `query()` would need to be $\frac{1}{X} - 1$ in order for $\mathbb{E}[Z] = \mathbb{E}\left[\frac{1}{X} - 1\right] = N$, our true unique count.

There are a few problems with this. Firstly, we can't store exact real numbers, so we'd need to store them up to some finite precision. This can be done in a straightforward manner, but the more pressing problem is our hash function itself.

We'd need to store this hash function in order to maintain consistency, but this hash function is just a mapping for all n possible inputs, i.e. we need $O(n)$ memory. This is really bad, since the naive solution already takes $O(n)$ memory. As such, we can replace h with a hash function from a 2-wise independent family, and reduce the memory needed to store the hash function.

This is another example of how we can use 2-wise independent families to reduce the memory consumption of storing the hash function in memory.