Graph Neural Reasoning for 2-Quantified Boolean Formula Solvers

Zhanfu Yang [1] **Fei Wang** [1] **Ziliang Chen** [2] **Guannan Wei** [1] **Tiark Rompf** [1]

Abstract

In this paper, we investigate the feasibility of learning GNN (Graph Neural Network) based solvers and GNN-based heuristics for specified QBF (Quantified Boolean Formula) problems. We design and evaluate several GNN architectures for 2QBF formulae, and conjecture with empirical support that GNN has limitations in learning 2QBF solvers. Then we show how to learn heuristics for a CEGAR-based 2QBF solver. We further explore generalizing GNN-based heuristics to larger unseen instances and uncover some interesting challenges. In summary, this paper provides a comprehensive surveying view of applying GNN-embeddings to specified QBF solvers and aims to offer guidance in applying ML to more complicated symbolic reasoning problems.

1. Introduction

A propositional formula expression consists of Boolean constants ($\top$: true, $\bot$: false), Boolean variables (x_i), and propositional connectives such as $\wedge$, $\vee$, $\neg$, and etc. The SAT (Boolean Satisfiability) problem, which asks if a given formula can be satisfied (as $\top$) by assigning proper Boolean values to the variables, is the first proven NP-complete problem (Cook, 1971). As an extension of propositional formula, QBF (Quantified Boolean Formula) allows quantifiers ($\forall$ and $\exists$) over the Boolean variables. In general, a quantified Boolean formula can be expressed as such:

$$Q_i X_i Q_{i-1} X_{i-1} ... Q_0 X_0 \phi$$

Where Q_i denote quantifiers that differ from its neighboring quantifiers, X_i are disjoint sets of variables, and ϕ is propositional formulae with all Boolean variables bounded. The QBF problem is PSPACE-complete (Savitch, 1970). For this issue, researchers previously proposed incremental determination (Rabe & Seshia, 2016; Rabe et al., 2018) or CEGAR-based (Janota et al., 2016) solvers to solve it. They are non-deterministic, *e.g.*, employing heuristics guidance for search a solution. Recently, MaxSAT-based (Janota & Marques-Silva, 2011) and ML-based (Janota, 2018) heuristics have been proposed into CEGAR-based solvers. Without existing decision procedure, Selsam et al. (2018) presented a GNN architecture that embeds the propositional formulae. Amizadeh et al. (2019) adapt a RL-style explore-exploit mechanism in this problem, but considering circuit-SAT problems. However, these solvers didn't tackle unsatisfiable formulae. In terms of the above discussion, there is no desirable general solver towards a QBF problem in practice. Therefore, we focus on 2QBF formulae in this paper, a specified-QBF case with only one alternation of quantifiers.

Extended from SAT, 2QBF problems keep attracting a lot of attentions due to their practical usages (Mishchenko et al., 2015; Mneimneh & Sakallah, 2003; Remshagen & Truemper, 2005), yet remaining very challenging like QBF. Formally, $Q_1 X Q_2 Y.\phi$, where $Q_i \in \{\forall, \exists\}$, X and Y are sets of variables, and ϕ is quantifier-free formula. The quantifier-free formula ϕ can be in Conjunctive Normal Form (CNF), where ϕ is a conjunction of *clauses*, clauses are disjunctions of *literals*, and each literal is either a variable or its negation. For example, the following term is a well-formed 2QBF in CNF: $\forall x, y \exists z.(x \vee z) \wedge (y \vee \neg z)$. If ϕ is in CNF, it is required that the $\forall$ quantifier is on the outside, and the $\exists$ quantifier is on the inside. Briefly, the 2QBF problem is to ask whether the formula can be evaluated to $\top$ considering the $\forall$ and $\exists$ quantifications. It's presumably exponentially harder to solve 2QBF than SAT because it characterizes the second level of the polynomial hierarchy.

Our work explores several different 2QBF solvers by way of graph neural-symbolic reasoning. In Section 2, we investigate famous SAT GNN-based solvers (Selsam et al., 2018)(Amizadeh et al., 2019). We found these architectures hard to extend to 2QBF problems, due to that GNN is unable to reason about unsatisfiability. To solve this, we further make some effective reconfiguration to GNN. In Section 3, on behalf of a traditional CEGAR-based solver, three ways to learn the GNN-based heuristics are proposed: to rank the candidates, to rank the counterexamples, and their combination. They aim to avoid multiple GNN embeddings per formula, to reduce the GNN inference overhead. Relevant experiments showcase their superiorities in 2QBF.

[1]Department of Computer Science, Purdue University, USA [2]Sun Yat-Sen University, China. Correspondence to: Zhanfu Yang <yang1676@purdue.edu>, Tiark Rompf <tiark@purdue.edu>.

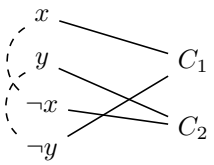
Preliminary work. Under review at the ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data. Do not distribute.

2. GNN-based QBF Solver Failed

Let's first revisit the existing GNN-based SAT solvers, and analyze why they fails to suit the 2QBF problem.

2.1. GNN for QBF

Embedding of SAT SAT formulae are translated into bipartite graphs Selsam et al. (2018), where literals (L) represent one kind of nodes, and clauses (C) represent the other kind. We denote EdgeMatrix ($\mathbb{E}$) as edges between literal and clause nodes with dimension $|C|$ x $|L|$. The graph of $(x \vee \neg y) \wedge (\neg x \vee y)$ is given below as an example.

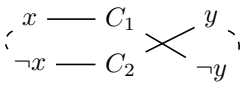


As below, Emb_L and Emb_C denote embedding matrices of literals and clauses respectively, $\text{Msg}_{X\to Y}$ denotes messages from X to Y, $\mathcal{M}_X$ denotes MLP of X for generating messages, $\mathcal{L}_X$ denotes LSTM of X for digesting incoming messages and updating embeddings, $X \cdot Y$ denotes matrix multiplication of X and Y, X^T denotes matrix transportation of X, $[X, Y]$ denotes matrix concatenation, and $\text{Emb}_{\neg L}$ denotes the embedding of L's negations.

$$\begin{aligned}\text{Msg}_{L\to C} &= \mathcal{M}_L(\text{Emb}_L)\\ \text{Emb}_C &= LSTM_C(\mathbb{E}\cdot\text{Msg}_{L\to C})\\ \text{Msg}_{C\to L} &= \mathcal{M}_C(\text{Emb}_C)\\ \text{Emb}_L &= LSTM_L([\mathbb{E}^T\cdot\text{Msg}_{C\to L}, \text{Emb}_{\neg L}])\end{aligned}$$

Iterations are fixed for train but can be unbounded for test.

Embedding of 2QBF We separate $\forall$-literals and $\exists$-literals in different groups, embed them via different NN modules. The graph representation of $\forall x \exists y.(x \vee \neg y) \wedge (\neg x \vee y)$ shows:



We use $\forall$ and $\exists$ to denote all $\forall$-literals and all $\exists$-literals respectively. We use $\mathbb{E}_X$ denote the EdgeMatrix between X and C, and $\mathcal{M}_{C\to X}$ denote MLPs that generate $\text{Msg}_{C\to X}$.

$$\begin{aligned}\text{Msg}_{\forall\to C} &= \mathcal{M}_\forall(\text{Emb}_\forall)\\ \text{Msg}_{\exists\to C} &= \mathcal{M}_\exists(\text{Emb}_\exists)\\ \text{Emb}_C &= LSTM_C([\mathbb{E}_\forall\cdot\text{Msg}_{\forall\to C}, \mathbb{E}_\exists\cdot\text{Msg}_{\exists\to C}])\\ \text{Msg}_{C\to\forall} &= \mathcal{M}_{C\to\forall}(\text{Emb}_C)\\ \text{Msg}_{C\to\exists} &= \mathcal{M}_{C\to\exists}(\text{Emb}_C)\\ \text{Emb}_\forall &= LSTM_\forall([\mathbb{E}_\forall^T\cdot\text{Msg}_{C\to\forall}, \text{Emb}_{\neg\forall}])\\ \text{Emb}_\exists &= LSTM_\exists([\mathbb{E}_\exists^T\cdot\text{Msg}_{C\to\exists}, \text{Emb}_{\neg\exists}])\end{aligned}$$

We designed multiple architectures (details in supplementary) and used the best one as above for the rest of the paper.

Data Preparation For training and testing, we follow Chen & Interian (2005), which generates QBFs in conjunctive normal form. Specifically, we generate problems of *specs* (2,3) and *sizes* (8,10). Each clause has five literals, 2 of them are randomly chosen from a set of 8 $\forall$-quantified variables, three are randomly chosen from a set of 10 $\exists$-quantified variables. We modify the generation procedure that it generates clauses until the formula becomes unsatisfiable. We then randomly negate a $\exists$-quantified literal per formula to make it satisfiable.

Table 1. GNN Performance to Predict SAT/UNSAT

DATASET	40 PAIRS	80 PAIRS	160 PAIRS
8 ITERS	(0.98, 0.94)	(1.00, 0.92)	(0.84, 0.76)
TESTING	(0.40, 0.64)	(0.50, 0.48)	(0.50, 0.50)
16 ITERS	(1.00, 1.00)	(0.96, 0.96)	(0.88, 0.70)
TESTING	(0.54, 0.46)	(0.52, 0.52)	(0.54, 0.48)
32 ITERS	(1.00, 1.00)	(0.98, 0.98)	(0.84, 0.80)
TESTING	(0.32, 0.68)	(0.52, 0.50)	(0.52, 0.50)

Table 2. GNN Performance to Predict Witness of UNSAT

DATASET	160 UNSAT	320 UNSAT	640 UNSAT
8 ITERS	(1.00, 0.99)	(0.95, 0.72)	(0.82, 0.28)
TESTING	(0.64, 0.06)	(0.67, 0.05)	(0.69, 0.05)
16 ITERS	(1.00, 1.00)	(0.98, 0.87)	(0.95, 0.69)
TESTING	(0.64, 0.05)	(0.65, 0.05)	(0.65, 0.06)
32 ITERS	(1.00, 1.00)	(0.99, 0.96)	(0.91, 0.57)
TESTING	(0.63, 0.05)	(0.64, 0.05)	(0.63, 0.05)

SAT/UNSAT We vote MLPs from $\forall$-variables and use average votes as logits for SAT/UNSAT prediction:

$$\text{logits} = \text{mean}(\mathcal{M}_{vote}(\text{Emb}_\forall))$$

As in table 1, Each block of entries are accuracy rate of UNSAT and SAT formulae respectively. The models are tested on 600 pairs of formulae and we allow message-passing iterations up to 1000. GNNs fit well to smaller training dataset but has trouble for 160 pairs of formulae. Performance deteriorates when embedding iterations increase, and most GNNs become very biased at high iterations.

$\forall$-Witnesses of UNSAT Proving unsatisfiability of 2QBF needs a witness of unsatisfiability, which is an assignment to $\forall$-variables that eventually leads to UNSAT. We use logistic regression in this experiment. To be specific, the final embeddings of $\forall$-variables are transformed into logits via a MLP $\mathcal{M}_{asn}$ and used to compute the cross-entropy loss with the known witness unsatisfiability of the formulae.

$$\text{witness} = \text{softmax}(\mathcal{M}_{asn}(\text{Emb}_\forall))$$

This training task is very similar to Amizadeh et al. (2019), except our GNN has to reason about unsatisfiability of the simplified SAT formulae, which we believe infeasible. We summarize the results in Table 2. In each block of entries, we list the accuracy per variable and accuracy per formulae on the left and right separately. Entries in the upper half of each block are for training data and lower half for testing data. From the table, we see that GNNs fit well to the training data. More iterations of message-passing give better fitting. However, the performance on testing data is only

slightly better than random. More iterations in testing do not help with performance.

2.2. Why GNN-based QBF Solver Failed

We conjecture current GNN architectures and embedding processes are unlikely to prove unsatisfiability or reason about ∀-assignments. Even in SAT problem Selsam et al. (2018), GNNs are good at finding solutions for satisfiable formulae, while not for confidently proving unsatisfiability. Similarly, Amizadeh et al. (2019) had little success in proving unsatisfiability with DAG-embedding because showing SAT only needs a witness, but proving UNSAT needs complete reasoning about the search space. A DPLL-based approach would iterate all possible assignments and construct a proof of UNSAT. However, a GNN embedding process is neither following a strict order of assignments nor learning new knowledge that indicates some assignments should be avoided. The GNN embedding may be mostly similar to vanilla WalkSAT approaches, with randomly initialized assignments and stochastic local search, which can not prove unsatisfiability.

This conjecture may be a great obstacle for learning 2QBF solvers from GNN because proving either satisfiability or unsatisfiability of the 2QBF problem needs not only a witness. If the formula is satisfiable, proof needs to provide assignments to ∃-variables under all possible assignments of ∀-variables or in a CEGAR-based solver. If the formula is unsatisfiable, then the procedure should find an assignment for the ∀-variables.

3. Learn GNN-based Heuristics

In Section 2, we know that GNN-based 2QBF Solvers are unlikely to be learned; therefore, the success of learning SAT solvers (Selsam et al., 2018; Amizadeh et al., 2019) cannot simply extend to 2QBF or more expressive logic. We consider the CEGAR-based solving algorithm to reduce the GNN inference overhead. We first present the CEGAR-based solving procedure in Algorithm 1 (Janota & Marques-Silva, 2011).

Note that ω is constraints for candidates. Initially, ω is $\emptyset$, and any assignment of ∀-variables can be proposed as a candidate which may reduce the problem to a smaller propositional formula. If we can find an assignment to ∃-variables that satisfy the propositional formula, this assignment is called a counterexample to the candidate. We denote $\phi_{counter}$ as all clauses in ϕ that are satisfied by the counterexample. The counterexample can be transformed into a constraint, stating that next candidates cannot simultaneously satisfy clauses ($\phi \setminus \phi_{counter}$) since those candidates are already rejected by the current counterexample. This constraint can be added to ω as a propositional term,

Algorithm 1 CEGAR 2QBF solver

Input: $\forall X \exists Y \phi$
Output: (sat, -) or (unsat, witness)
Initialize constraints ω as empty set.
while true **do**
  (has-candidate, candidate) = SAT-solver(ω)
  if not has-candidate **then**
    return (sat, -)
  end if
  (has-counter, counter) = SAT-solver($\phi[X \rightarrow$ candidate$]$)
  if not has-counter **then**
    return (unsat, candidate)
  end if
  add counter to constraints ω
end while

thus finding new candidates is done by solving constraints-derived propositional term ω.

3.1. Ranking the Candidates

To decide which candidate to use from SAT-solver (ω), we can rank solutions in MaxSAT-style by simplifying the formula with candidates and ranking them based on the number of clauses they satisfy. We use it as a benchmark comparison. Besides, the hardness can be evaluated as the number of solutions of the simplified propositional formula. Thus the training data of our ranking GNN is all possible assignments of ∀-variables and the ranking scores that negatively relate to the number of solutions of each assignment-propagated propositional formula (Details shown in supplementary).

We extend the GNN embedding architecture so that the final embedding of the ∀-variables are transformed into a scoring matrix ($\text{Sm}_\forall$) for candidates via a MLP ($\mathcal{M}_{\forall,scoring}$). A batch of candidates ($\mathbb{C}$) are ranked by passing through a two-layer MLP without biases, where the weights of the first layer are the scoring matrix ($\text{Sm}_\forall$), and the weights of the second layer is a weight vector ($\text{Wv}_\forall$).

$$\text{Sm}_\forall = \mathcal{M}_{\forall,\text{scoring}}(\text{Emb}_\forall)$$
$$\text{Score}_\forall = \text{ReLU}(\mathbb{C} \cdot \text{Sm}_\forall) \cdot \text{Wv}_\forall$$

We make use of the TensorFlow ranking library (Pasumarthi et al., 2018) to compute the pairwise-logistic-loss with NDCG-lambda-weight for supervised training. What's more, we evaluate our ranking heuristics by adding them to CEGAR cycle and measure the average steps needed to solve the problems. It requires us to change the $SAT(\omega)$ subroutine to a $nSAT(\omega)$ subroutine, where once a solution is found, it is added back to the formula as constraint, and search for a different solution, until no solutions can be found, or maximal number of solutions is reached. Then the heuristics ranks the solutions and proposes the best one as a candidate. We use four datasets: (1)TrainU: 1000 unsatisfiable formulae used for training; (2) TrainS: 1000 satisfiable formulae used for training; (3) TestU: 600 unsatisfiable formulae used for testing; (4) TestS: 600 satisfiable formulae

Table 3. Performance of CEGAR Candidate Ranking

DATASET	TRAINU	TRAINS	TESTU	TESTS
-	21.976	34.783	21.945	33.885
MAXSAT	13.144	30.057	12.453	28.863
GNN1	13.843	31.704	13.988	30.573
GNN2	15.287	32.0	14.473	30.788

used for testing); and four ranking heuristics: (1) -: no ranking; (2) MaxSAT: ranking by the number of satisfied clauses via on-the-fly formula simplification; (3) GNN1: ranking by hardness via GNN model inference; (4) GNN2: ranking by the number of satisfied clauses via GNN model inference.

As shown in Table 3, all 3 ranking heuristics improve the solving process of all 4 datasets. Unsatisfiable formulae benefit more from the heuristics, and the heuristics generalizes very well from training formulae to testing formulae. Machine learning results are repeated twice with different random seeds, and numbers shown are from models with best performance on training data.

3.2. Ranking the Counterexamples

We consider a GNN-based heuristics for ranking counterexamples. Each counterexample contributes to a constraint in ω, which either shrinks the search space of the witnesses of unsatisfiability or be added to the constraints indicating that no candidates are witnesses of unsatisfiability.

As follows, we compute ranking scores for our training data. For satisfiable 2QBF instances in the training data, we list all possible assignments of $\exists$-variables and collect all constraining clauses in ω. Then we solve ω with `hmucSAT` (Nadel et al., 2013), seeking for unsatisfiability cores. Initially we plan to give a high ranking score (10) for $\exists$-assignments corresponding to clauses in unsatisfiability cores, and a low ranking score (1) for all other $\exists$-assignments. Later, we choose to give other $\exists$-assignments ranking scores based on the number of satisfied clauses, in range of $[1, 8]$ because unsatisfiability cores are often small.

For unsatisfiable 2QBF instances, we collect all constraining clauses in ω. As ω is satisfiable and solutions are witnesses of unsatisfiability, we add solutions to ω as extra constraints until the ω becomes unsatisfiable. We then compute the ranking scores. We use another dataset of which rankings scores are based on the number of clauses satisfied for comparison. Notations include $\mathrm{Sm}_\exists$ for the scoring matrix, $\mathcal{M}_{\exists,\mathrm{scoring}}$ for a MLP to get scoring matrix from the final embedding of $\exists$-variables, $\mathbb{CE}$ for a batch of counterexamples, and $\mathrm{Wv}_\exists$ for the weight vector.

$$\mathrm{Sm}_\exists = \mathcal{M}_{\exists,\mathrm{scoring}}(\mathrm{Emb}_\exists)$$
$$\mathrm{Score}_\exists = \mathrm{ReLU}(\mathbb{CE} \cdot \mathrm{Sm}_\exists) \cdot \mathrm{Wv}_\exists$$

After supervised training, we evaluate the trained GNN-based ranking heuristics in a CEGAR-based solver. The results are shown in Table 4. Based on the MaxSAT heuristics, ranking counterexamples benefits solving satisfiable formulae more than unsatisfiable formulae. However, GNN1 performs worse than GNN2. The likely explanation is that predicting unsatisfiability cores is far too complicated for GNN. Moreover, knowledge of unsatisfiability cores cannot be obtained from each counterexample alone, but needs analysis of all counterexamples collectively. It may go back to the limitation of GNN in reasoning about "all possible solutions", and the added score information behaves like an interference rather than knowledge for GNN-based ranking heuristics. Machine learning results are repeated twice, reporting models with best training data performance.

Table 4. Performance of CEGAR-COUNTER-RANKING

DATASET	TRAINU	TRAINS	TESTU	TESTS
-	21.976	34.783	21.945	33.885
MAXSAT	14.754	22.265	14.748	21.638
GNN1	17.492	26.962	17.198	26.598
GNN2	16.95	26.717	16.743	26.325

Table 5. Performance of CEGAR-BOTH-RANKING

DATASET	TRAINU	TRAINS	TESTU	TESTS
-	21.976	34.783	21.945	33.885
MAXSAT	9.671	20.777	9.425	19.883
GNN1	11.686	25.021	11.605	24.518
GNN2	12.505	25.505	12.22	24.938
GNN3	11.25	24.76	12.008	24.295

3.3. Combination of the Heuristics

To combine ranking heuristics and counterexamples in a single solver, we extend the GNN-embedding architecture with ranking data of candidates and counterexamples. We have GNN1 trained by ranking scores from hardness and unsatisfiability cores, GNN2 trained by ranking scores from the number of satisfied clauses for both candidates and counterexamples, and GNN3 trained by ranking scores from hardness for candidates, and number of satisfied clauses for counterexamples. As shown in Table 5, GNN3 is arguably the best model we obtained from supervised learning via this ranking method. All machine learning results are repeated twice with different random seeds, and models with the best performance in training data are reported.

4. Conclusion

In this paper, we show learning GNN-based 2QBF solvers is hard by current GNN architectures due to its inability to reason about unsatisfiability. Our work extends the previous GNN-based 2QBF solver in terms of CEGAR-based heuristic. A suite of GNN-based techniques has been made to improve the GNN embedding for reasoning 2QBF solutions. Their superiorities are witnessed in our experiments.

References

Amizadeh, S., Matusevych, S., and Weimer, M. Learning to solve circuit-SAT: An unsupervised differentiable approach. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=BJxgz2R9t7.

Chen, H. and Interian, Y. A model for generating random quantified boolean formulas. In *IJCAI*, pp. 66–71. Professional Book Center, 2005.

Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pp. 151–158, New York, NY, USA, 1971. ACM. doi: 10.1145/800157.805047. URL http://doi.acm.org/10.1145/800157.805047.

Janota, M. Towards generalization in QBF solving via machine learning. In *AAAI*, pp. 6607–6614. AAAI Press, 2018.

Janota, M. and Marques-Silva, J. P. Abstraction-based algorithm for 2qbf. In *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pp. 230–244. Springer, 2011.

Janota, M., Klieber, W., Marques-Silva, J., and Clarke, E. M. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234:1–25, 2016.

Mishchenko, A., Brayton, R. K., Feng, W., and Greene, J. W. Technology mapping into general programmable cells. In *FPGA*, pp. 70–73. ACM, 2015.

Mneimneh, M. N. and Sakallah, K. A. Computing vertex eccentricity in exponentially large graphs: QBF formulation and solution. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pp. 411–425. Springer, 2003.

Nadel, A., Ryvchin, V., and Strichman, O. Efficient MUS extraction with resolution. In *FMCAD*, pp. 197–200. IEEE, 2013.

Pasumarthi, R. K., Wang, X., Li, C., Bruch, S., Bendersky, M., Najork, M., Pfeifer, J., Golbandi, N., Anil, R., and Wolf, S. Tf-ranking: Scalable tensorflow library for learning-to-rank. *CoRR*, abs/1812.00073, 2018.

Rabe, M. N. and Seshia, S. A. Incremental determinization. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pp. 375–392. Springer, 2016.

Rabe, M. N., Tentrup, L., Rasmussen, C., and Seshia, S. A. Understanding and extending incremental determinization for 2qbf. In Chockler, H. and Weissenbacher, G. (eds.), *Computer Aided Verification*, pp. 256–274, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96142-2.

Remshagen, A. and Truemper, K. An effective algorithm for the futile questioning problem. *J. Autom. Reasoning*, 34(1):31–47, 2005.

Savitch, W. J. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177 – 192, 1970. ISSN 0022-0000. doi: https://doi.org/10.1016/S0022-0000(70)80006-X. URL http://www.sciencedirect.com/science/article/pii/S002200007080006X.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018.

Graph Neural Reasoning for 2-Quantified Boolean Formula Solver (Supplementary Manuscript)

Anonymous Authors[1]

1. All GNN-embedding Architectures

We use subscript symbols $\forall$ to denote all $\forall$-quantified literals, $\exists$ to denote all $\exists$-quantified literals, L to denote all literals, and C to denote all clauses. We use notations Emb_X to denote embeddings of X, where X can be subscript $\forall$, $\exists$, L, or C. We use notations $\text{Emb}_{\neg X}$ to denote embedding of the negations of X ($\forall$, $\exists$, or L), which is part of Emb_X but at different indices. We use notations $\text{Msg}_{X\to Y}$ to denote messages from X to Y. We also use notations $\mathcal{M}_X$ to denote MLPs that generate messages from the embeddings of X, notations $\mathcal{M}_{X\to Y}$ to denote MLPs that generate messages from the embeddings of X for Y, notations $\mathcal{L}_X$ to denote LSTMs that update embeddings of X given incoming messages, and notations $\mathcal{L}_{X\leftarrow Y}$ to denote LSTMs that update embeddings of X given incoming messages from Y. We also use notations $\mathbb{E}_X$ to denote adjacency matrix of X ($\forall$, $\exists$, or C) and clauses, notations $X \cdot Y$ to denote matrix multiplication of X and Y, notations $[X, Y]$ to denote matrix concatenation of X and Y, and notations X^T to denote matrix transportation of X.

The simplest form of embedding of QBF (Model 1) is given below.

Model 1:

$$\begin{gathered}
\text{Msg}_{\forall\to C} = \mathcal{M}_\forall(\text{Emb}_\forall)\\
\text{Msg}_{\exists\to C} = \mathcal{M}_\exists(\text{Emb}_\exists)\\
\text{Emb}_C = \mathcal{L}_C(\mathbb{E}_\forall \cdot \text{Msg}_{\forall\to C} + \mathbb{E}_\exists \cdot \text{Msg}_{\exists\to C})\\
\text{Msg}_{C\to L} = \mathcal{M}_C(\text{Emb}_C)\\
\text{Emb}_\forall = \mathcal{L}_\forall([\mathbb{E}_\forall^T \cdot \text{Msg}_{C\to L}, \text{Emb}_{\neg\forall}])\\
\text{Emb}_\exists = \mathcal{L}_\exists([\mathbb{E}_\exists^T \cdot \text{Msg}_{C\to L}, \text{Emb}_{\neg\exists}])
\end{gathered}$$

In Model 2, we update the clause embedding by 2 LSTMs, each of them take the messages from $\forall$ and $\exists$ literals separately.

Model 2:

$$\begin{gathered}
\text{Msg}_{\forall\to C} = \mathcal{M}_\forall(\text{Emb}_\forall)\\
\text{Msg}_{\exists\to C} = \mathcal{M}_\exists(\text{Emb}_\exists)\\
\text{Emb}_C = \mathcal{L}_{C\leftarrow\forall}(\mathbb{E}_\forall \cdot \text{Msg}_{\forall\to C})\\
\text{Emb}_C = \mathcal{L}_{C\leftarrow\exists}(\mathbb{E}_\exists \cdot \text{Msg}_{\exists\to C})\\
\text{Msg}_{C\to L} = \mathcal{M}_C(\text{Emb}_C)\\
\text{Emb}_\forall = \mathcal{L}_\forall([\mathbb{E}_\forall^T \cdot \text{Msg}_{C\to L}, \text{Emb}_{\neg\forall}])\\
\text{Emb}_\exists = \mathcal{L}_\exists([\mathbb{E}_\exists^T \cdot \text{Msg}_{C\to L}, \text{Emb}_{\neg\exists}])
\end{gathered}$$

We switch the order of these 2 LSTMs in Model 3.

Model 3:

$$\begin{gathered}
\text{Msg}_{\forall\to C} = \mathcal{M}_\forall(\text{Emb}_\forall)\\
\text{Msg}_{\exists\to C} = \mathcal{M}_\exists(\text{Emb}_\exists)\\
\text{Emb}_C = \mathcal{L}_{C\leftarrow\exists}(\mathbb{E}_\exists \cdot \text{Msg}_{\exists\to C})\\
\text{Emb}_C = \mathcal{L}_{C\leftarrow\forall}(\mathbb{E}_\forall \cdot \text{Msg}_{\forall\to C})\\
\text{Msg}_{C\to L} = \mathcal{M}_C(\text{Emb}_C)\\
\text{Emb}_\forall = \mathcal{L}_\forall([\mathbb{E}_\forall^T \cdot \text{Msg}_{C\to L}, \text{Emb}_{\neg\forall}])\\
\text{Emb}_\exists = \mathcal{L}_\exists([\mathbb{E}_\exists^T \cdot \text{Msg}_{C\to L}, \text{Emb}_{\neg\exists}])
\end{gathered}$$

In Model 4 we concatenate the messages from $\forall$ and $\exists$ literals.

Model 4:

$$\begin{gathered}
\text{Msg}_{\forall\to C} = \mathcal{M}_\forall(\text{Emb}_\forall)\\
\text{Msg}_{\exists\to C} = \mathcal{M}_\exists(\text{Emb}_\exists)\\
\text{Emb}_C = \mathcal{L}_C([\mathbb{E}_\forall \cdot \text{Msg}_{\forall\to C}, \mathbb{E}_\exists \cdot \text{Msg}_{\exists\to C}])\\
\text{Msg}_{C\to L} = \mathcal{M}_C(\text{Emb}_C)\\
\text{Emb}_\forall = \mathcal{L}_\forall([\mathbb{E}_\forall^T \cdot \text{Msg}_{C\to L}, \text{Emb}_{\neg\forall}])\\
\text{Emb}_\exists = \mathcal{L}_\exists([\mathbb{E}_\exists^T \cdot \text{Msg}_{C\to L}, \text{Emb}_{\neg\exists}])
\end{gathered}$$

The performance of our GNN architectures improve greatly after we realize that (in Model 5) we may also need to use different MLP modules to generate messages from clauses to $\forall$ and $\exists$ literals. Note that this is also the model we reported in the main paper, and the model we decided to use for all results reported in main paper.

Model 5:

$$\begin{gathered}
\text{Msg}_{\forall\to C} = \mathcal{M}_\forall(\text{Emb}_\forall)\\
\text{Msg}_{\exists\to C} = \mathcal{M}_\exists(\text{Emb}_\exists)\\
\text{Emb}_C = \mathcal{L}_C([\mathbb{E}_\forall \cdot \text{Msg}_{\forall\to C}, \mathbb{E}_\exists \cdot \text{Msg}_{\exists\to C}])\\
\text{Msg}_{C\to\forall} = \mathcal{M}_{C\to\forall}(\text{Emb}_C)\\
\text{Msg}_{C\to\exists} = \mathcal{M}_{C\to\exists}(\text{Emb}_C)\\
\text{Emb}_\forall = \mathcal{L}_\forall([\mathbb{E}_\forall^T \cdot \text{Msg}_{C\to\forall}, \text{Emb}_{\neg\forall}])\\
\text{Emb}_\exists = \mathcal{L}_\exists([\mathbb{E}_\exists^T \cdot \text{Msg}_{C\to\exists}, \text{Emb}_{\neg\exists}])
\end{gathered}$$

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review at the ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data. Do not distribute.

We also explore the possibility (in Model 6) of having two embeddings for each clause, one serving the $\forall$ literals and one serving the $\exists$ literals. We need extra notations: $\text{Emb}_{X\to Y}$ denotes embeddings of X that serves Y. $\mathcal{L}_{X\to Y}$ denotes LSTMs that updates embedding of X that serves Y.

Model 6:

$$\begin{aligned}
\text{Msg}_{\forall\to C} &= \mathcal{M}_\forall(\text{Emb}_\forall)\\
\text{Msg}_{\exists\to C} &= \mathcal{M}_\exists(\text{Emb}_\exists)\\
\text{Emb}_{C\to\forall} &= \mathcal{L}_{C\to\forall}([\mathbb{E}_\forall\cdot\text{Msg}_{\forall\to C}, \mathbb{E}_\exists\cdot\text{Msg}_{\exists\to C}])\\
\text{Emb}_{C\to\exists} &= \mathcal{L}_{C\to\exists}([\mathbb{E}_\forall\cdot\text{Msg}_{\forall\to C}, \mathbb{E}_\exists\cdot\text{Msg}_{\exists\to C}])\\
\text{Msg}_{C\to\forall} &= \mathcal{M}_{C\to\forall}(\text{Emb}_{C\to\forall})\\
\text{Msg}_{C\to\exists} &= \mathcal{M}_{C\to\exists}(\text{Emb}_{C\to\exists})\\
\text{Emb}_\forall &= \mathcal{L}_\forall([\mathbb{E}_\forall^T\cdot\text{Msg}_{C\to\forall}, \text{Emb}_{\neg\forall}])\\
\text{Emb}_\exists &= \mathcal{L}_\exists([\mathbb{E}_\exists^T\cdot\text{Msg}_{C\to\exists}, \text{Emb}_{\neg\exists}])
\end{aligned}$$

We further explore possibility (in Model 7) that our embedding scheme should reflect a CEGAR cycle, which starts from $\forall$ variables (proposing candidates), to clauses, to $\exists$ variables (finding counterexamples), back to clauses, then back to $\forall$ variables.

Model 7:

$$\begin{aligned}
\text{Msg}_{\forall\to C} &= \mathcal{M}_\forall(\text{Emb}_\forall)\\
\text{Emb}_{C\to\exists} &= \mathcal{L}_{C\to\exists}(\mathbb{E}_\forall\cdot\text{Msg}_{\forall\to C})\\
\text{Msg}_{C\to\exists} &= \mathcal{M}_{C\to\exists}(\text{Emb}_{C\to\exists})\\
\text{Emb}_\exists &= \mathcal{L}_\exists([\mathbb{E}_\exists^T\cdot\text{Msg}_{C\to\exists}, \text{Emb}_{\neg\exists}])
\end{aligned}$$

$$\begin{aligned}
\text{Msg}_{\exists\to C} &= \mathcal{M}_\exists(\text{Emb}_\exists)\\
\text{Emb}_{C\to\forall} &= \mathcal{L}_{C\to\forall}(\mathbb{E}_\exists\cdot\text{Msg}_{\exists\to C})\\
\text{Msg}_{C\to\forall} &= \mathcal{M}_{C\to\forall}(\text{Emb}_{C\to\forall})\\
\text{Emb}_\forall &= \mathcal{L}_\forall([\mathbb{E}_\forall^T\cdot\text{Msg}_{C\to\forall}, \text{Emb}_{\neg\forall}])
\end{aligned}$$

2. Functions for Ranking Scores

Function for candidate ranking scores based on hardness, i.e. the number of models of reduced SAT formula.

```
def n_model_2_ranking_score(n_models):
    if n_models <= 3: return 10.0 - n_models
    if n_models <= 5: return 6.0
    if n_models <= 8: return 5.0
    if n_models <= 12: return 4.0
    if n_models <= 16: return 3.0
    if n_models <= 21: return 2.0
    else: return 1.0
```

Function for candidate ranking scores in maxSAT-style, i.e. based on the number of satisfied clauses.

```
def n_clauses_2_ranking_score(n_clauses_list):
    n_clauses_min = min(n_clauses_list)
    return [max(1, 10 - n_clauses + n_clauses_min)
            for n_clauses in n_clauses_list]
```

Function for counterexample ranking scores based on unsatisfiability cores and number of satisfied clauses.

```
def unsat_core_2_ranking_score(core_index,
                               n_clauses_list):
    n_clauses_max = max(n_clauses_list)
    scores = [max(1, 8 - n_clauses_max + n_clauses)
              for n_clauses in n_clauses_list]
    scores = numpy.array(scores)
    scores[core_index] = 10
    return scores.tolist
```

Function for counterexample ranking scores in maxSAT-style, i.e. based on the number of satisfied clauses.

```
def n_clauses_2_ranking_score_counter(n_clauses_list):
    n_clauses_max = max(n_clauses_list)
    return [max(1, 10 - n_clauses_max + n_clauses)
            for n_clauses in n_clauses_list]
```