Learning Heuristics of SAT by Graph Neural Networks

Guannan Wei
Purdue University

Abstract

In this document, we summarize the discussion and provide some formal description of the proposed model.

1 Problem Statement

In this paper, we report the progress and result on the project that aims to learn branching heuristics for SAT problem based on DPLL algorithm. The DPLL algorithm [1] and its variants are widely used in many modern SAT solvers. The core ideas behind DPLL are searching and unit propagation. The searching involves choosing a variable and assigning it to either $\top$ or $\bot$, which is the part of our concern in this paper.

The DPLL algorithm works on Conjunction Normal Form. A CNF formula is a conjunction of clauses. A clause is a disjunction of literals. A literal is either a variable (symbol) or a negation of a variable. Formulas in other forms can be translated to CNF [3]. For example, $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ is a valid CNF formula.

The pseudocode of DPLL procedure is listed as follows:

```
def dpll(f: Formula): Boolean = {
  if (f.hasUnsatClause) return false
  if (f.isSat) return true
  if (f.hasUnitClause) return dpll(f.elimUnitClause)
  val v = f.pickVariable
  val tryTrue = dpll(f.addUnitClause(v))
  if (tryTrue) true
  else dpll(f.addUnitClause(neg(v)))
}
```

Particularly, we would like to learn the branching heuristics for `f.pickVariable`. In this paper, we use graph neural network to encode a CNF formula and use reinforcement learning to learn the best branching heuristics. We use the policy gradient method for the reinforcement learning part.

2 Proposed Model

2.1 General Description

Graph Representation The graph embedding captures the permutation invariance and the structure of a CNF formula. We first review NeuroSAT's graph embedding of a CNF formula [2].

Given a formula F of m clauses and n variables in CNF, one can construct a graph $G = \langle V, E\rangle$ such that $V = V_{lit} \cup V_{cls}$ where V_{lit} is the set of vertices embedding literals, V_{cls} is the set of vertices embedding clauses. And E contains two different types of edge, the first one is the edegs connecting literals and its containing clauses, the second one is the edges connecting a variable and its negation, formally, $E = \{(l, c)|l \in c\} \cup \{(l, \neg l)|l\}$ for all literals l and clauses c. Note that in the NeuroSAT's encoding, there is a vertex for every variable and its negation no matter it appears or not, so the size of vertices set $|V| = 2n + m$.

Message-Passing Neural Network After the graph is constructed, NeuroSAT uses message-passing mechanism to refine the vector space embeddings. One iteration of a message-passing consists of two stages: 1) first every clause receives messages from its neighboring literals and update the clause embedding; 2) then every literal receives its message from its neighboring clauses and its negated literal vertices, and update its embedding.

Suppose the size of embedding vectors is d. The initial embedding vectors of literals and clauses are $L^0 \in \mathbb{R}^{2n\times d}$ and $C^0 \in \mathbb{R}^{m\times d}$ which are randomly initialized. NeuroSAT uses two MLPs $\mathcal{L}_{msg}$ and $\mathcal{C}_{msg}$ to produce the messages from embeddings, and two LSTMs $\mathcal{L}_u$ and $\mathcal{C}_u$ to update the embeddings every iteration (the hidden states are elided). As mentioned above, for the time t there are two stages in a iteration:

$$C^{t+1} \leftarrow \mathcal{C}_u(\mathcal{L}_{msg}(L^t))$$
$$L^{t+1} \leftarrow \mathcal{L}_u(\mathcal{C}_{msg}(C^{t+1}))$$

After T iterations, with the embedding produced, another MLP $\mathcal{P}$ which served as the policy network takes the embeddings as input and produces the vector $\mathcal{P}(L^T) \in \mathbb{R}^{2n}$ that represents the probabilities of choosing which literal.

2.2 Embedding Problem

However, naively adapting NeuroSAT's graph embedding is not going to work. Because NeuroSAT models SAT as a supervised learning problem without using traditional algorithms such as DPLL, which then means the graph structure is static. But in the DPLL and reinforcement learning setting, the formula is changed every time when we make a decision – some variables and clauses may completely be eliminated by unit propagation over the time.

This further raises two entangled problems: 1) How to embed the assignments as well as the decision history into

Purdue CS590-DLS, Dec 2019, West Lafayette, Indiana
2019.

the graph. 2) How to reflect the structural changes in the graph representation. Ideally, if the assignments are properly embedded, the graph should be able to automatically reflect the resulting formula after an assignment. Otherwise, adding annotations on nodes to identify live literals and clauses or manually pruning the graph (i.e., removing the edges and nodes) are two options.

Solutions It seems a perfect self-adapting graph neural network architecture that reflects the assignments does not exist yet.

So we discuss the second pruning schema that changes the graph structure every time. Note that there are conceptually two neural networks, $\mathcal{G}$ transforms a formula (graph represented) to embeddings, and $\mathcal{P}$ predicts how to choose from the live variables according to the embeddings. Also note that if conflict-driven clauses learning is not introduced, the graph is monotonically shrinking every time.

Consider a complete decision trace that consists of t decisions:

$$(F_0, G_0, d_0) \to (F_1, G_1, d_1) \cdots \to (F_t, G_t, d_t)$$

where F_i is the formula, G_i is the corresponding graph embedding, and d_i is the decision at time i. A decision d is a pair that consists of a variable and a assigned value (either $\top$ or $\bot$).

If eventually we have SAT, then we assign a reward 1 for every decision on the this path, and backprop the two neural networks $\mathcal{G}$ and $\mathcal{P}$ at each level of decisions.

If eventually we have UNSAT or have to backtrack, then we assign reward $-\frac{1}{2^{t-i+1}}$ and backprop the $\mathcal{G}$ and $\mathcal{P}$ at the level i. For example, we assign the most recent decision a reward $-\frac{1}{2}$ and backprop the $\mathcal{G}$ and $\mathcal{P}$ at this level; then assign a reward $-\frac{1}{4}$ and backprop the $\mathcal{G}$ and $\mathcal{P}$ at level $t-1$, and etc.

References

[1] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. https://doi.org/10.1145/368273.368557

[2] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L Dill. 2018. Learning a SAT Solver from Single-Bit Supervision. *arXiv preprint arXiv:1802.03685* (2018).

[3] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning*. Springer, 466–483.