Graph Neural Network as a Branching Heuristic in Solving Quantified Boolean Formulas

Senior Thesis

Ryan Brill

December 2019

Abstract

We explore a continuation of *Learning Heuristics for Quantified Boolean Formulas through Deep Reinforcement Learning* [1], which discusses using Conflict-Driven Clause Learning, with branching heuristic given by a Graph Neural Network (GNN), trained by Reinforcement Learning, to find satisfying assignments for 2-Quantified Boolean Formulas. In this project, we use Deep Graph Library (DGL), a Python package which supports deep learning on graphs, to re-implement the GNN from [1], which operates on the graph derived from the Conjunctive Normal Form (CNF) of a boolean formula. We also create novel graphs for the original versions of a boolean formula from which the CNF is derived, and use DGL to create GNNs operating on these novel graphs.

Acknowledgements

Thank you Professor Antonio Montalban for sponsoring this project and supporting my work. Thank you Professor Dana Scott for connecting me with researchers in the Berkeley and international math communities, such as Professor Sanjit Seshia, in order to help me find a fitting research topic. Thank you Professor Sanjit Seshia for giving me the opportunity to work in his research group, for suggesting that I explore the intersection of machine learning and automated reasoning, and for connecting me with UC Berkeley graduate student Gil Lederman. Thank you Gil Lederman for allowing me to pursue a continuation of your previous project [1], helping me learn the details of your work, being a pleasure to work with, and spending many hours with me on Skype.

Contents

1 Overview

1.1 The Quantified Boolean Formula Problem

The purpose of this project is to explore a method that could be used to solve a generalization of the Boolean Satisfiability Problem (SAT) [2] known as the Quantified Boolean Formula (QBF) Problem [3]. A **boolean formula** consists of boolean variables $x_1, x_2, ..., x_n$ (with values in $\{0, 1\}$), their negations $\overline{x}_1, \overline{x}_2, ..., \overline{x}_n$, and the logical operators $\wedge$ (and) and $\vee$ (or). For example,

$$F(x_1, x_2, x_3) = x_1 \vee (\overline{x}_2 \wedge x_3)$$

is a boolean formula. In SAT, the task is to find a **satisfying assignment** of a boolean formula, which is to find an assignment of variables such that the formula evaluates to 1. For example, a satisfying assignment of $F(x_1, x_2, x_3)$ is

$$x_1 = 0, x_2 = 0, x_3 = 1$$

since

$$F(0, 0, 1) = 0 \vee (\overline{0} \wedge 1) = 0 \vee (1 \wedge 1) = 0 \vee 1 = 1$$

In the QBF Problem, the task is to find a satisfying assignment of a **quantified boolean formula**, a boolean formula in which each variable is quantified by $\forall$ (the universal quantifier "for all") or $\exists$ (the existential quantifier "there exists"). Since there are only two quantifiers, we need only include the universal quantifier. For example,

$$G(x_2, x_3) = \forall x_1,\ x_1 \vee (\overline{x}_2 \wedge x_3)$$

is a quantified boolean formula, and a satisfying assignment of $G(x_2, x_3)$ is

$$x_2 = 0, x_3 = 1$$

since

$$G(0, 1) = \forall x_1,\ x_1 \vee (\overline{0} \wedge 1) = \forall x_1,\ x_1 \vee (1 \wedge 1) = \forall x_1,\ x_1 \vee 1 = \forall x_1, 1 = 1$$

In order to make the project more feasible, we restrict ourselves to **2-Quantified Boolean Formulas (2QBF)**, in which exactly 2 variables are universally quantified.

1.2 Conflict-Driven Clause Learning

Usually, **backtracking algorithms** [4] are used to decide the satisfiability of boolean formulas. Introduced in 1962, the **Davis–Putnam–Logemann–Loveland (DPLL) algorithm** [5] is an improvement on the basic backtracking algorithm, and it still forms the basis of most efficient solvers today. In our solver, we use the **Conflict-Driven Clause Learning (CDCL) algorithm** [6], an improvement of DPLL, because it is the best known form of backtracking for boolean satisfiability. Note that all of these backtracking based algorithms have exponential runtime $O(2^n)$, where n is the number of variables, which makes sense considering SAT is NP-Complete [7].

The basic backtracking algorithm [4] runs on a boolean formula F as follows:

```
Backtracking Algorithm(F):
    • While True:
        • choose a literal in F by some procedure
          (the "branching heuristic")
        • assign this literal a value 0 or 1
        • simplify F based on this assignment
        • if F is now satisfiable:
            • then the original formula is satisfiable,
              and we are done
        • if F is now not satisfiable:
            • then we must "backtrack" by undoing the
              current literal assignment
```

This backtracking algorithm creates a "search tree" in which nodes are literals and edges are the assignment of literals to either 0 or 1.

DPLL [5] enhances this basic backtracking algorithm by taking advantage of **Conjunctive Normal Form (CNF)**. A formula is in CNF if it is a conjunction of one or more **clauses**, where each clause is a disjunction of literals - in other words, it is an "AND of OR's". For example,

$$F(x_1, x_2, x_3) = (x_1 \vee \overline{x}_2) \wedge (x_1 \vee x_3)$$

is in CNF because it is the conjunction of 2 clauses: $x_1 \vee \overline{x}_2$ and $x_1 \vee x_3$. Also, note that this CNF formula is equivalent to our boolean formula from earlier:

$$F(x_1, x_2, x_3) = x_1 \vee (\overline{x}_2 \wedge x_3)$$

In fact, all boolean formulas can be written in CNF by manipulating the distributive properties of $\wedge$ and $\vee$.

By assuming a formula is in CNF, DPLL improves upon the basic backtracking algorithm with two enhancements, which are applied at the end of each iteration of the algorithm. The first is **unit propagation**: if a clause is a unit clause (i.e., contains only a single unassigned literal), we assign this literal a value to make it evaluate to 1. For example, if we have a unit clause $\overline{x}_5$, then we assign $x_5 = 0$ to satisfy this clause. The second improvement is **pure literal elimination**: if a variable occurs with only one polarity in a formula (i.e., only x_5 or $\overline{x}_5$ appears in a formula, but not both), we can assign a value to these pure literals to make them all evaluate to 1. For example, if $\overline{x}_5$ appears in a formula but x_5 never does, then we can assign $x_5 = 0$ to make all clauses containing $\overline{x}_5$ evaluate to 1.

However, when DPLL encounters a **conflict**, an assignment of variables which makes the formula unsatisfiable, DPLL learns nothing from the conflict, and it only backtracks one level in the search tree. CDCL [6] improves upon these shortcomings by learning something from a conflict and backtracking to a more appropriate level in the search tree. For example, suppose we run DPLL on a boolean formula containing clauses

$$\begin{gathered}\overline{x}_7 \vee \overline{x}_3 \vee x_9 \\ \overline{x}_7 \vee x_8 \vee \overline{x}_9\end{gathered}$$

and suppose DPLL has assigned $x_3 = 1, x_7 = 1, x_8 = 0$. Then we simplify these clauses by

$$\begin{aligned}\overline{x}_7 \vee \overline{x}_3 \vee x_9 &= 0 \vee 0 \vee x_9 = x_9 \\ \overline{x}_7 \vee x_8 \vee \overline{x}_9 &= 0 \vee 0 \vee \overline{x}_9 = \overline{x}_9\end{aligned}$$

The formula now has clauses x_9 and $\overline{x}_9$, which is a conflict because $x_9 \wedge \overline{x}_9$ always evaluates to 0. Furthermore, we know that this conflict arose by assigning $x_3 = 1, x_7 = 1, x_8 = 0$. DPLL would not use this information to improve our backtracking search, as it would merely unassign whichever variable was previously assigned. CDCL, however, will use the fact that the assignments $x_3 = 1, x_7 = 1, x_8 = 0$ led to our conflict. Specifically, we have

$$x_3 = 1 \wedge x_7 = 1 \wedge x_8 = 0 \implies \text{conflict}$$

The idea of CDCL is to take the contrapositive:

$$\text{not conflict} \implies \overline{(x_3 = 1 \wedge x_7 = 1 \wedge x_8 = 0)}$$

or equivalently,

$$\text{not conflict} \implies x_3 = 0 \vee x_7 = 0 \vee x_8 = 1$$

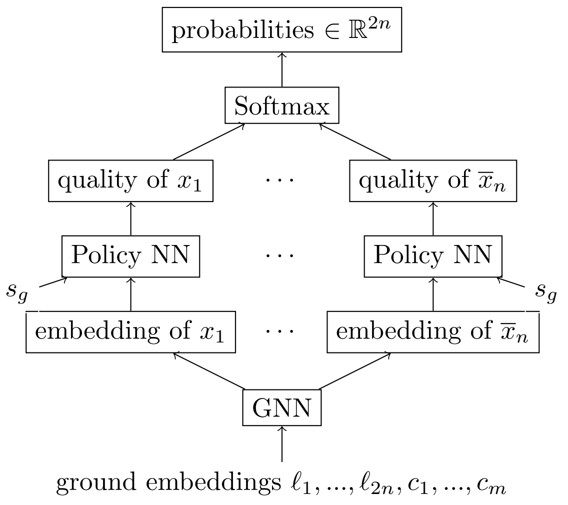
We have derived a new **conflict clause**, $x_3 = 0 \vee x_7 = 0 \vee x_8 = 1$, which we have learned from the conflict $x_9 \wedge \overline{x}_9$. In order for our formula to be satisfiable, the conflict clause must evaluate to 1. Hence, in CDCL, after we reach the conflict $x_9 \wedge \overline{x}_9$ during backtracking, we add the conflict clause $x_3 = 0 \vee x_7 = 0 \vee x_8 = 1$ to our CNF formula, and backtrack to the first assignment of a variable among x_3, x_7, x_8 in the search tree.

By backtracking in a non-chronological manner and learning from conflicts, CDCL is a significant improvement over DPLL. Therefore, in our 2QBF solver, we use the CDCL algorithm to traverse through the search tree and assign values to variables.

1.3 Graph Neural Network as a Branching Heuristic

In our 2QBF solver, we use CDCL to traverse through the search tree and assign values to variables. However, we still need to choose a **branching heuristic** - we need to specify *how* we should select a variable to assign a value of 0 or 1 at the beginning of each iteration of CDCL. The idea explored in *Learning*

Heuristics for Quantified Boolean Formulas through Deep Reinforcement Learning [1] is to use a **Graph Neural Network (GNN)** as part of the branching heuristic, since boolean formulas can be viewed as graphs. Specifically, given initial feature vectors for literals and clauses called **ground embeddings**, we use a GNN to compute **embeddings** (vectors) for each literal in a boolean formula. The literal embeddings, alongside the **global solver state** s_g (some statistics about the current state of the CDCL solver), are then piped through the **Policy Neural Network**, which produces a probability distribution that describes the quality of each literal. In other words, the greater the probability for a literal, the better it is as a choice for a branching decision in CDCL. We then sample from this probability distribution to select a literal as our next branching decision in CDCL. The entire branching heuristic is summarized by


probabilities $\in \mathbb{R}^{2n}$
Softmax
quality of x_1
$\cdots$
quality of $\overline{x}_n$
Policy NN
$\cdots$
Policy NN
s_g
s_g
embedding of x_1
$\cdots$
embedding of $\overline{x}_n$
GNN
ground embeddings $\ell_1, ..., \ell_{2n}, c_1, ..., c_m$

We detail our transformation of boolean formulas into graphs in Section 3. We detail the architecture of our GNNs in Section 4.

1.4 Training the Graph Neural Network via Reinforcement Learning

We train our GNN by Reinforcement Learning (RL), using the same method provided in [1]. In RL, we consider an agent that interacts with an environment $\mathcal{E}$ over discrete time steps. The environment is a Markov decision process (MDP) with states $\mathcal{S}$, actions $\mathcal{A}$, and rewards per time step $r_t \in \mathbb{R}$. A **policy** is a mapping $\pi : \mathcal{S} \times \mathcal{A}$ such that $\sum_{a \in \mathcal{A}} \pi(s, a) = 1 \ \forall s$, defining the probability to take action a in state s. The goal of the agent is to maximize the expected reward accumulated over the episode, $\mathbb{E}[\sum_{t=0}^{\infty} r_t | \pi]$.

In our setting, the environment $\mathcal{E}$ is the solver CADET [8]. At each time step, the agent gets an observation, which consists of the current boolean formula as a graph, and the state of the CADET solver. Only the literals which have not yet been assigned a value of 0 or 1 are valid actions, and we assume that the observation includes the set of available actions. The agent then selects one action from the subset of the available literals, via a branching heuristic, which for us is a GNN.

An **episode** is the result of the interaction of the agent with the environment. We consider an episode to be **complete** if the solver reaches a terminating state in the last step. As there are arbitrarily long episodes, we want to abort them after some step limit, and consider these episodes **incomplete**.

We train the GNN using REINFORCE [9]. For each batch, we sample a formula from the training set, and generate b episodes by solving it multiple times. In each episode we run CADET for up to 250 steps using the latest policy. Then we assign rewards for each of these episodes and use them to estimate the gradient. We assign a small negative reward of -10^{-4} for each decision to encourage the heuristic to solve each formula in fewer steps. When a formula is solved successfully, we assign reward 1 to the last decision [1].

2 Generating Boolean Formulas

We need to generate boolean formulas in order to train the GNN and test our 2QBF solver. As done in [1], we generate 3 versions of a boolean formula - the Word-Level, AIG, and CNF formulas - which we describe in this section.

2.1 Word-Level Arithmetic Expressions

We generate boolean formulas by exploiting the correspondence between boolean formulas and arithmetic expressions like $x + y$.

The **full adder** is a prime example of how to convert arithmetic expressions to boolean formulas. A full adder is a digital circuit which adds binary numbers, and accounts for the values that are "carried in" and "carried out". Recall that standard base-10 addition involves "carrying the 1" if adding two digits exceeds 9. Similarly, binary addition involves "carrying the 1" if adding two digits exceeds 1. This bit-by-bit binary addition is given by the following truth table, where we add bits A and B with carry-input bit C_{in} to get the sum bit SUM and carry-output bit C_{out}:

Input			Output	
A	B	C_{in}	SUM	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Let's examine the fourth and fifth lines. In the fourth line, we want to add bits $A = 0$ and $B = 1$, and we have the carry input $C_{in} = 1$. Then $SUM = A + B + C_{in} = 0 + 1 + 1 = 2 = 0 \pmod 2$, and $C_{out} = 1$ since $2 > 1$. In the fifth line, we want to add bits $A = 1$ and $B = 0$, and we have the carry input $C_{in} = 0$. Then $SUM = A + B + C_{in} = 1 + 0 + 0 = 1$, and $C_{out} = 0$ since $1 \leq 1$.

The above truth table defines the addition of bits with a carry bit. We would like to add **words**, which for us is a variable with 8-bits. For example,

$$x + y$$

is a **Word-Level arithmetic expression** because it is the addition of the words x and y (so, x and y are 8-bit variables). To add words, we use a sequence of the above full adders, where C_{out} becomes C_{in} as the addition of bits progresses over the word.

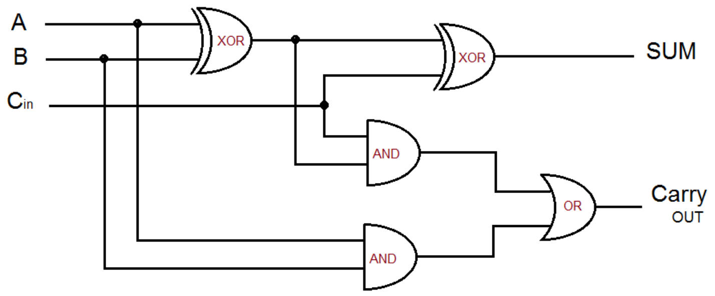
The Word-Level arithmetic expression $x + y$ relates to boolean formulas because we can implement the SUM and C_{out} components of the full adder via a boolean formula. Specifically, letting $\oplus$ denote XOR, we analyze the above truth table to notice that

$$SUM = C_{in} \oplus (A \oplus B)$$

and

$$C_{out} = (A \wedge B) \vee (C_{in} \wedge (A \oplus B))$$

Together, these boolean formulas form the following circuit:



Much like the 8-bit word expression $x + y$ corresponds to a chain of 8 of the above boolean circuits, other Word-Level arithmetic expressions correspond to other boolean circuits. It is easy to generate such arithmetic expressions, and from these we obtain many boolean formulas to train our GNN.

2.2 AIG Formulas

We use the Python package `py-aiger` [10] to transform Word-Level arithmetic expressions into boolean formulas. Specifically, `py-aiger` transforms arithmetic expressions into **Aiger (AIG) circuits**. AIG means "and-inverter gates", which is a descriptive name because AIG circuits consist entirely of literals and and-gates. For example, the following is an AIG circuit:

$$F = \begin{cases} \text{inputs } x_1, x_2 \\ x_3 = x_1 \wedge x_2 \\ x_4 = \overline{x}_1 \wedge x_3 \\ x_5 = x_2 \wedge \overline{x}_4 \\ \text{output } x_5 \end{cases}$$

F consists of input literals x_1, x_2, 3 and-gates x_3, x_4, x_5, and one output literal x_5.

Let's consider the link between arithmetic expressions like $x + y$ and AIG circuits. The full adder circuit consists of input bits A, B, C_{in} and output bits SUM, C_{out}. So, the input bits of an arithmetic circuit are linked to the input literals of an AIG circuit, and the output bits of an arithmetic circuit are linked to the output literals of an AIG circuit. The and-gates correspond to the innards of an arithmetic circuit.

2.3 CNF Formulas from AIG Formulas

We know how to convert Word-Level arithmetic expressions to AIG formulas, but we need to convert these AIG formulas to CNF in order to use CDCL.

Suppose we have the following Aiger formula

$$F_{AIG} = \begin{cases} \text{inputs } x_1, x_2 \\ x_3 = x_1 \wedge x_2 \\ x_4 = \overline{x}_1 \wedge x_3 \\ x_5 = x_2 \wedge \overline{x}_4 \\ \text{output } x_5 \end{cases}$$

An Aiger formula consists only of literals and and-gates. So, to convert an Aiger formula to CNF, we need only transform each and gate into a disjunction, which we accomplish via the **Tseytin transformation** [11]. Letting $C = A \wedge B$ denote an arbitrary and-gate on literals $A, \overline{A}, B, \overline{B}, C, \overline{C}$, the Tseytin transformation is

$$C = A \wedge B \quad \mapsto \quad (\overline{A} \vee \overline{B} \vee C) \wedge (A \vee \overline{C}) \wedge (B \vee \overline{C})$$

Applying the Tseytin transformation to the 3 and-gates in the above AIG formula derives the following CNF clauses:

$$\begin{aligned} x_3 = x_1 \wedge x_2 \quad &\mapsto \quad (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_3) \\ x_4 = \overline{x}_1 \wedge x_3 \quad &\mapsto \quad (x_1 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_1 \vee \overline{x}_4) \wedge (x_3 \vee \overline{x}_4) \\ x_5 = x_2 \wedge \overline{x}_4 \quad &\mapsto \quad (\overline{x}_2 \vee x_4 \vee x_5) \wedge (x_2 \vee \overline{x}_5) \wedge (\overline{x}_4 \vee \overline{x}_5) \end{aligned}$$

Also, for the output x_5 of the above AIG formula, we add a singleton clause

$$x_5$$

Hence the corresponding CNF formula is the conjunction of the above 10 clauses:

$$F_{CNF} = \begin{cases} \ (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \\ \wedge\ (x_1 \vee \overline{x}_3) \\ \wedge\ (x_2 \vee \overline{x}_3) \\ \wedge\ (x_1 \vee \overline{x}_3 \vee x_4) \\ \wedge\ (\overline{x}_1 \vee \overline{x}_4) \\ \wedge\ (x_3 \vee \overline{x}_4) \\ \wedge\ (\overline{x}_2 \vee x_4 \vee x_5) \\ \wedge\ (x_2 \vee \overline{x}_5) \\ \wedge\ (\overline{x}_4 \vee \overline{x}_5) \\ \wedge\ x_5 \end{cases}$$

3 Boolean Formulas as Graphs

We want to use a GNN as part of the branching heuristic in the CDCL algorithm to solve 2-quantified boolean formulas. To do so, we need to view boolean formulas as graphs. In this section, we describe how to construct a graph for CNF, AIG, and Word-Level formulas.

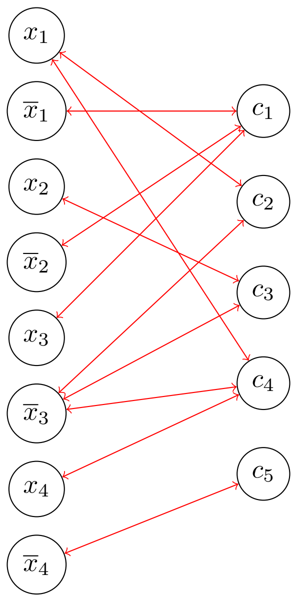
3.1 CNF Graph

The **CNF graph** is a representation of the CNF of a boolean formula. The nodes are the $2n$ literals $x_1, \overline{x}_1, ..., x_n, \overline{x}_n$ and the m clauses denoted $c_1, ..., c_m$. For each clause $c_k = x_{i_1} \vee x_{i_2} \vee \cdots \vee x_{i_\ell}$ of the CNF formula, we have 2 directed "CNF edges" $x_{i_j} \leftarrow c_k$ and $x_{i_j} \rightarrow c_k$ for each x_{i_j} in the clause (the edges are bi-directional and not undirected because the GNN is designed for directed graphs). Hence there are 2 edge types: **CNF forward edges** and **CNF backward edges** (where forward edges refer to $\rightarrow$ and backward edges refer to $\leftarrow$).

For example, consider the CNF of the boolean formula H:

$$H_{CNF} = \begin{cases} \quad (\overline{x}_1 \vee \overline{x}_2 \vee x_3) \\ \wedge\ (x_1 \vee \overline{x}_3) \\ \wedge\ (x_2 \vee \overline{x}_3) \\ \wedge\ (x_1 \vee \overline{x}_3 \vee x_4) \\ \wedge\ \overline{x}_4 \end{cases}$$

The nodes are the 8 literals $x_1, \overline{x}_1, ..., x_4, \overline{x}_4$ and the 5 clauses denoted $c_1, ..., c_5$. Coloring the CNF edges red, the CNF graph of H looks as follows:



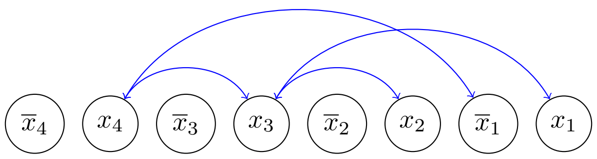
3.2 AIG Graph

The **AIG graph** is a representation of an AIG formula. The nodes are the $2n$ literals $x_1, \overline{x}_1, ..., x_n, \overline{x}_n$. For each and-gate $x_k = x_i \wedge x_j$ in the AIG formula, we have 4 directed "AIG edges" $x_k \leftarrow x_i, x_k \rightarrow x_i, x_k \leftarrow x_j, x_k \rightarrow x_j$, corresponding to 2 edge types: **AIG forward edges** and **AIG backward edges** (where forward edges refer to $\rightarrow$ and backward edges refer to $\leftarrow$).

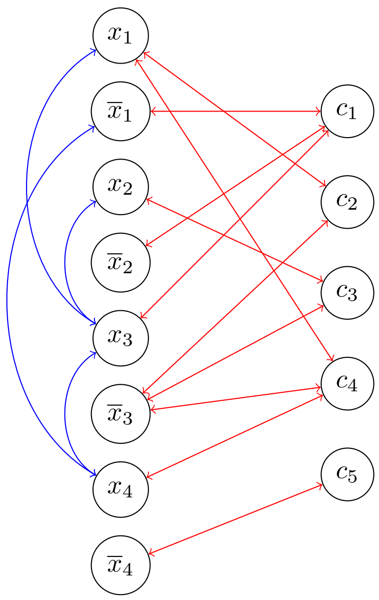
For example, consider the H from the CNF graph. The AIG form of H is

$$H_{AIG} = \begin{cases} \text{inputs } x_1, x_2 \\ x_3 = x_1 \wedge x_2 \\ x_4 = \overline{x}_1 \wedge x_3 \\ \text{output } \overline{x}_4 \end{cases}$$

The nodes are the 8 literals $x_1, \overline{x}_1, ..., x_4, \overline{x}_4$. Coloring the AIG edges blue, the AIG graph of H looks as follows:



Because the CNF and AIG graphs of H share nodes (the literals), we can combine the CNF and AIG graph of H to produce the **CNF-AIG graph** of H, which looks as follows:



3.3 Word-Level Graph

The **Word-Level graph** is a representation of a Word-Level arithmetic expression. The Word-Level graph has a node for each variable, constant, and operation, and an edge for each operation (more precisely, forward and backward edges for each operation). Furthermore, for non-commutative operations like $-$, we split the operation in two: $-_L$ and $-_R$ (meaning $-$ left and $-$ right).

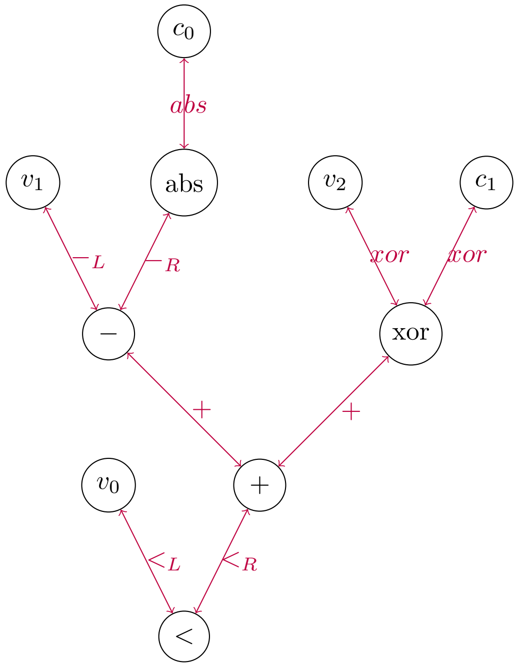
Here, we do not consider the example H from the CNF/AIG graphs, because even a very small arithmetic expression yields a AIG/CNF circuit that is much bigger than H. Hence we consider an arithmetic expression on its own, and imagine connecting it to the CNF/AIG graphs. For example, consider the arithmetic expression

$$v_0 < (v_1 - abs(c_0)) + \mathrm{xor}(v_2, c_1)$$

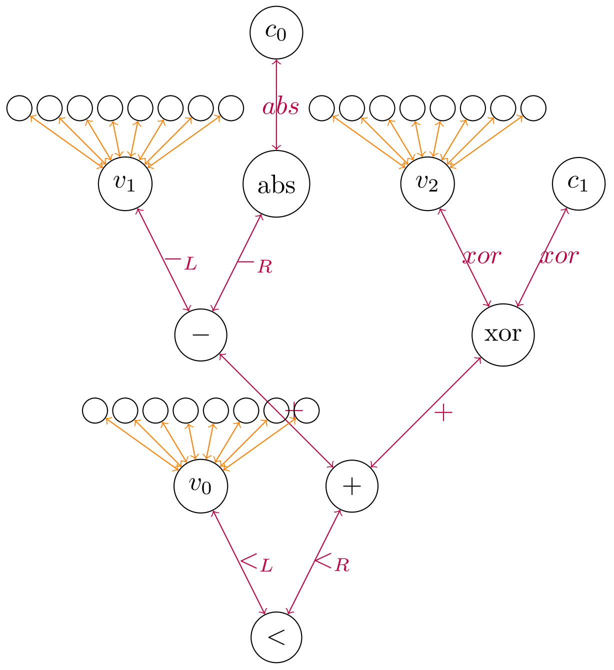
where v_0, v_1, and v_2 are variables (which are connected to the input variables of the AIG circuit, as discussed in Section 2.1), and c_0, c_1 are constants in $\mathbb{R}$. To represent this arithmetic expression as a graph, we write it in prefix notation as

$$< (v_0, +(-(v_1, \mathrm{abs}(c_0)), \mathrm{xor}(v_2, c_1))$$

which has a natural binary tree structure. The nodes are the variables v_0, v_1, v_2, the constants c_0, c_1, and the operations abs,, xor, $+$, $<$. We have an edge for each operation $+, \mathrm{xor}, \mathrm{abs}, <_L, <_R, -_L, -_R$, colored in purple. Then the Word-Level graph looks as follows:



Because the Word-Level and AIG graphs share nodes - the Word-Level variables correspond to the input variables of an AIG circuit - we can combine the Word-Level and AIG graphs. We depict the input variables of the AIG circuit by the small unlabelled circles, and color the edges connecting the Word-Level graph to the AIG graph in orange:



4 The Graph Neural Network (GNN)

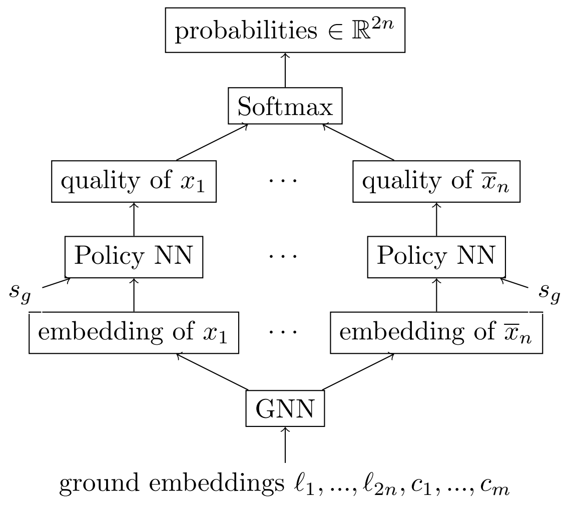
4.1 Graph Neural Network as a Branching Heuristic

We want to use a GNN as part of the branching heuristic in the CDCL algorithm to solve 2-quantified boolean formulas. In other words, during the current iteration of CDCL, the GNN helps us decide which literal to assign a value to.

Given initial feature vectors for each node in our graph (for instance, initial feature vectors for each literal and clause node in the CNF graph), our GNN computes an **embedding** (vector) for each literal. The idea of a GNN is to "pass messages" through the graph. In other words, for each node v in the graph, we compute the embedding of v by aggregating the messages of v's neighboring nodes, where a node's message is some manipulation of its initial feature vector. The details of this GNN computation are given in the remainder of this section.

Then, the $2n$ literal embeddings computed by our GNN, alongside the **global solver state** s_g (some statistics about the current state of the CDCL solver), are fed through another neural network called the **Policy Neural Network** (a multilayer perceptron ending in a softmax), which produces a probability distribution that describes the quality of each literal. In other words, the greater the probability for a literal, the better it is as a choice for a branching decision in the CDCL algorithm. We then sample from this probability distribution to select a literal as our next branching decision in CDCL.

The entire network is summarized by


probabilities $\in \mathbb{R}^{2n}$
Softmax
quality of x_1
$\cdots$
quality of $\overline{x}_n$
Policy NN
$\cdots$
Policy NN
s_g
s_g
embedding of x_1
$\cdots$
embedding of $\overline{x}_n$
GNN
ground embeddings $\ell_1, ..., \ell_{2n}, c_1, ..., c_m$

4.2 CNF Graph Neural Network Layer

Recall that the CNF graph consists of nodes for each clause and each literal, with an edge (x_i, c_j) iff literal x_i is in clause c_j. The idea of the CNF Layer is to learn embeddings for the literal nodes by passing messages (embeddings) across edges (i.e, from the literals to the clauses and then back to the literals). We compute these embeddings over T iterations, where T is a hyperparameter. For iterations $0 \leq t \leq T$, let $c_v^{(t)}$ denote the embedding for the clause corresponding to node v at iteration t, and let $\ell_u^{(t)}$ denote the embedding for the literal corresponding to node u at iteration t. Let $\mathcal{N}(v)$ be the set of neighbors of node v. Note that the parameters of our network (which we learn via Reinforcement Learning) are the matrices $\mathbf{W}_L, \mathbf{W}_C$ and vectors $\mathbf{b}_L, \mathbf{b}_C$, which are written in bold. Then we update the embeddings in iteration $t+1$ by

$$c_v^{(t+1)} = \text{ReLU}\Bigg(\frac{1}{|\mathcal{N}(v)|} \sum_{u \in \mathcal{N}(v)} \big(\mathbf{W}_L \cdot \ell_u^{(t)} + \mathbf{b}_L\big)\Bigg) \tag{4.2.1}$$

$$\ell_u^{(t+1)} = \text{ReLU}\Bigg(\frac{1}{|\mathcal{N}(u)|} \sum_{v \in \mathcal{N}(u)} \big(\mathbf{W}_C \cdot c_v^{(t+1)} + \mathbf{b}_C\big)\Bigg) \tag{4.2.2}$$

Let's analyze formula 4.2.1. Its purpose is to compute the updated embedding $c_v^{(t+1)}$ for clause node v at iteration $t+1$. The high-level idea is to aggregate the messages of the literal nodes corresponding to the literals that are in clause v. So, for each $u \in \mathcal{N}(v)$ (i.e., for each literal u which is in clause v), we take the embedding $\ell_u^{(t)}$, and compute its message by piping it through a Linear Layer (i.e., $\mathbf{W}_L \cdot \ell_u^{(t)} + \mathbf{b}_L$). Then, we aggregate these messages by taking their average. Finally, we pipe this averaged-message through a non-linearity (specifically, a ReLU) to produce the updated embedding for clause node v.

Formula 4.2.2 is similar. Its purpose is to compute the updated embedding $\ell_u^{(t+1)}$ for literal node u at iteration $t+1$. The high-level idea is to aggregate the messages of the clause nodes corresponding to the clauses which contain literal u. So, for each $v \in \mathcal{N}(u)$ (i.e., for each clause v which contains literal u), we take the embedding $c_v^{(t+1)}$, and compute its message by piping it through a linear layer (i.e., $\mathbf{W}_C \cdot c_v^{(t+1)} + \mathbf{b}_C$). We then aggregate these messages by taking their average. Next, we pipe this averaged-message through a non-linearity (specifically, a ReLU) to produce the updated embedding for literal node u.

We have recursive updates for each $c_v^{(t+1)}$ and $\ell_u^{(t+1)}$, so we need to define initial embeddings for each clause and literal node, which we call **ground embeddings**. For each literal u, its ground embedding $\ell_u^{(0)} \in \mathbb{R}^8$ indicates whether the literal is universally or existentially quantified, whether it currently has a value assigned, whether it was selected as a decision variable already on the current search branch, and its VSIDS activity score (a popular heuristic for SAT solving with CDCL, which reflects how often a variable recently occurred in conflict analysis). For each clause v, its ground embedding $c_v^{(0)} \in \mathbb{R}$ is a single scalar in $\{0, 1\}$ indicating whether the clause is original or derived in conflict analysis.

4.3 AIG Graph Neural Network Layer

Recall that the AIG graph consists of nodes for each literal, with edges (x_i, x_j) and (x_j, x_i) iff literal x_j is in the and-gate which defines x_i. The AIG GNN Layer is similar to the CNF GNN Layer. However, whereas edges in the CNF graph connect literals to clauses, edges in the AIG graph connect literals to literals. So, in the AIG Layer, we pass messages from literals to literals. Note that the parameters of our network (which we learn via Reinforcement Learning) are the matrices $\mathbf{W}_f, \mathbf{W}_b$ and vectors $\mathbf{b}_f, \mathbf{b}_b$, which are written in bold, where f refers to "forward" AIG edges and b refers to "backward" AIG edges. Then we update the embeddings for literal nodes in iteration $t + 1$ by

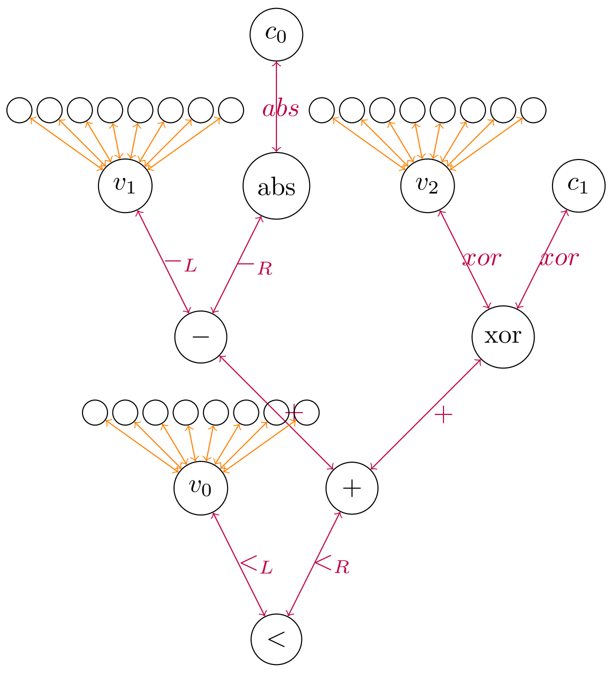
$$\begin{aligned} N_v &= \texttt{In Degree}(v) + \texttt{Out Degree}(v) \\ \ell_v^{(t+1)} &= \mathrm{ReLU}\Bigg(\frac{1}{N_v}\Bigg[\sum_{\substack{\text{edges} \\ (v,u)}} \big(\mathbf{W}_f \cdot \ell_u^{(t)} + \mathbf{b}_f\big) + \sum_{\substack{\text{edges} \\ (w,v)}} \big(\mathbf{W}_b \cdot \ell_w^{(t)} + \mathbf{b}_b\big)\Bigg]\Bigg) \end{aligned}$$

Let's analyze this formula. Its purpose is to compute the updated embedding $\ell_v^{(t+1)}$ for literal node v at iteration $t+1$. The high-level idea is to aggregate the messages of the literal nodes that are connected to v via an edge. So, for each forward edge (v, u), we take the embedding $\ell_u^{(t)}$, and compute its message by piping it through a Linear Layer (i.e., $\mathbf{W}_f \cdot \ell_u^{(t)} + \mathbf{b}_f$). Also, for each backward edge (w, v), we take the embedding $\ell_w^{(t)}$, and compute its message by piping it through a Linear Layer (i.e., $\mathbf{W}_b \cdot \ell_w^{(t)} + \mathbf{b}_b$). Then, we aggregate these messages by taking their average. Finally, we pipe this averaged-message through a non-linearity (specifically, a ReLU) to produce the updated embedding for literal node v.

We have a recursive update for each $\ell_v^{(t+1)}$, so we need to define the literal **ground embeddings** as we did with the CNF Layer. We use the same literal ground embeddings as with the CNF Layer, except we add an extra feature, indicating whether the literal is an input/output in the AIG circuit. So, for the AIG Layer, each literal has a ground embedding $\ell^{(0)} \in \mathbb{R}^9$.

4.4 Word-Level Graph Neural Network Layer

Recall Section 3.3, which discusses the Word-Level graph. For convenience, we display our example Word-Level graph here:



Recall that the Word-Level variables are input variables in an AIG circuit, which is depicted by the orange edges in the above graph. The high-level idea of the Word-Level Layer is to pass messages from these AIG literals into the Word-Level variables, down through the tree, and then back up through the tree to update the AIG literal embeddings.

Note that for each operation (which corresponds to 2 edges in the Word-Level graph, a “forward” and “backward” edge), we have 4 parameters, which we learn via Reinforcement Learning, and these parameters are denoted in bold. For example, the operation + corresponds to + forward edges and + backward edges, so + has 4 corresponding parameters, the matrices $\mathbf{W}_{\underset{f}{+}}, \mathbf{W}_{\underset{b}{+}}$ and vectors $\mathbf{b}_{\underset{f}{+}}, \mathbf{b}_{\underset{b}{+}}$. So, the biggest drawback of this Word-Level propagation is that we have many parameters.

Then, for this graph, the message propagation is as follows, where $emb[n]$ is the embedding of node n:

Forward Propagation:

$$\begin{aligned}
emb[\text{abs}] &= \text{ReLU}\Big(\mathbf{W}_{\underset{f}{abs}} \cdot emb[c_0] + \mathbf{b}_{\underset{f}{abs}}\Big)\\
emb[-] &= \text{ReLU}\Big(\text{mean}\Big[\big(\mathbf{W}_{\underset{f}{-^L}} \cdot emb[v_1] + \mathbf{b}_{\underset{f}{-^L}}\big)\ ,\ \big(\mathbf{W}_{\underset{f}{-^R}} \cdot emb[\text{abs}] + \mathbf{b}_{\underset{f}{-^R}}\big)\Big]\Big)\\
emb[\text{xor}] &= \text{ReLU}\Big(\text{mean}\Big[\big(\mathbf{W}_{\underset{f}{xor}} \cdot emb[v_2] + \mathbf{b}_{\underset{f}{xor}}\big)\ ,\ \big(\mathbf{W}_{\underset{f}{xor}} \cdot emb[c_1] + \mathbf{b}_{\underset{f}{xor}}\big)\Big]\Big)\\
&\vdots\\
emb[<] &= \text{ReLU}\Big(\text{mean}\Big[\big(\mathbf{W}_{\underset{f}{<^L}} \cdot emb[v_0] + \mathbf{b}_{\underset{f}{<^L}}\big)\ ,\ \big(\mathbf{W}_{\underset{f}{<^R}} \cdot emb[+] + \mathbf{b}_{\underset{f}{<^R}}\big)\Big]\Big)
\end{aligned}$$

Backward Propagation:

$$\begin{aligned}
emb[v_0] &= \text{ReLU}\Big(\mathbf{W}_{\underset{b}{<^L}} \cdot emb[<] + \mathbf{b}_{\underset{b}{<^L}}\Big)\\
emb[+] &= \text{ReLU}\Big(\mathbf{W}_{\underset{b}{<^R}} \cdot emb[<] + \mathbf{b}_{\underset{b}{<^R}}\Big)\\
&\vdots\\
emb[c_0] &= \text{ReLU}\Big(\mathbf{W}_{\underset{b}{abs}} \cdot emb[\text{abs}] + \mathbf{b}_{\underset{b}{abs}}\Big)
\end{aligned}$$

Let's analyze the forward propagation. We begin with initial embeddings of v_0, v_1, v_2, c_0, c_1 (we get the initial embeddings of v_0, v_1, v_2 from the AIG graph, since these are inputs to the AIG circuit, and c_0, c_1 are constants in $\mathbb{R}$). We want to propagate messages down the tree, so we create an ordering of the non-initialized nodes and update their embeddings in that order. For this graph, the forwards-propagation ordering is $(\text{abs}, -, \text{xor}, +, <)$. This is because we need the embedding of abs to compute the embedding of $-$, we need the embeddings of $-$ and xor to compute the embedding of $+$, and we need the embedding of $+$ to compute the embedding of $<$. Now, let's examine the embedding computation for $-$, given by

$$emb[-] = \text{ReLU}\Big(\text{mean}\Big[\big(\mathbf{W}_{\underset{f}{-^L}} \cdot emb[v_1] + \mathbf{b}_{\underset{f}{-^L}}\big)\ ,\ \big(\mathbf{W}_{\underset{f}{-^R}} \cdot emb[\text{abs}] + \mathbf{b}_{\underset{f}{-^R}}\big)\Big]\Big)$$

Looking at the graph, we see that the $-$ node should aggregate messages from its upstream neighbors v_1 and abs. Because v_1 is connected to $-$ via a $-L$ edge, and because this is a downstream computation, we use a $-_L$ forward edge. Thus we compute the message of v_1 by piping its embedding through the $\underset{f}{-^L}$ Linear Layer, where $\underset{f}{-^L}$ refers to the $-_L$ forward edge. In other words, the message of

v_1 is

$$\mathbf{W}_{-_f^L} \cdot emb[v_1] + \mathbf{b}_{-_f^L}$$

Similarly, the message of abs is

$$\mathbf{W}_{-_f^R} \cdot emb[\text{abs}] + \mathbf{b}_{-_f^R}$$

Then, we aggregate the messages of v_1 and abs by taking their mean. Next, we pipe this aggregated message through a non-linearity (specifically, a ReLU) to produce the embedding $emb[-]$ for the $-$ node.

The backward propagation is computed similarly to the forwards propagation.

4.5 Testing

We have two testing objectives. First, we want we want to test the efficacy of the CNF GNN as implemented in DGL compared to the CNF GNN from [1]. Also, we want to test whether message propagation on the AIG graph and/or the Word-Level graph, in addition to the CNF graph, enhances the GNN's performance as a branching heuristic in CDCL.

We are currently testing our models. Preliminary results suggest that the CNF GNN implemented in DGL is much slower than the CNF GNN from [1], due to some back-end implementation issues. However, we have contacted DGL, and they are working on fixing the issues. Nonetheless, we expect the addition of the AIG and Word-Level graphs to improve the GNN because these graphs provide significant extra information.

References

[1] Gil Lederman, Markus N. Rabe, Edward A. Lee, and Sanjit A. Seshia. Learning Heuristics for Quantified Boolean Formulas through Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1807.08058, Jul 2018.

[2] Boolean satisfiability problem. `https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`, Oct 2019. Accessed: 2019-09-30.

[3] True quantified boolean formula. `https://en.wikipedia.org/wiki/True_quantified_Boolean_formula`, Apr 2019. Accessed: 2019-09-21.

[4] Backtracking. `https://en.wikipedia.org/wiki/Backtracking`, Sep 2019. Accessed: 2019-09-30.

[5] Dpll algorithm. `https://en.wikipedia.org/wiki/DPLL_algorithm`, Sep 2019. Accessed: 2019-09-30.

[6] Conflict-driven clause learning. `https://en.wikipedia.org/wiki/Conflict-driven_clause_learning`, May 2019. Accessed: 2019-09-30.

[7] Cook-levin theorem. `https://en.wikipedia.org/wiki/Cook\OT1\textendashLevin_theorem`, Sep 2019. Accessed: 2019-09-30.

[8] Markus N. Rabe and Sanjit A. Seshia. *Incremental Determinization.* Springer International Publishing, 2016. pp. 375–392.

[9] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992.

[10] Marcell Vazquez-Chanlatte. mvcisback/py-aiger. `https://doi.org/10.5281/zenodo.1326224`, August 2018.

[11] Tseytin transformation. `https://en.wikipedia.org/wiki/Tseytin_transformation`, Sep 2019. Accessed: 2019-09-30.