Санкт-Петербургский государственный университет

АНГЕНИ Георгий Эдуардович

Выпускная квалификационная работа

Поиск эвристических правил для решения задачи булевой выполнимости методами машинного обучения

Уровень образования: бакалавриат
Направление: 02.03.01 «Математика и компьютерные науки»
Основная образовательная программа: СВ.5152.2019

Научный руководитель:
Доцент,
Факультет Математики и Компьютерных Наук,
Санкт-Петербургский Государственный Университет,
К.ф-м.н.,
Александр Юрьевич Авдюшенко

Рецензент:
Доцент,
Департамент математики,
НИУ «Высшая Школа Экономики» (СПб),
К.ф-м.н.,
Александр Владимирович Сироткин

Санкт-Петербург
2023

Saint-Petersburg State University

Georgii Angeni

Graduation qualification thesis

Finding heuristic rules for solving the Boolean satisfiability problem using machine learning methods

Level of education: Bachelor's degree
Main field of study: 02.03.01 "Mathematics and Computer Science"
Main academic programme: SV.5152.2019

Thesis supervisor:
Assistant Professor,
Department of Mathematics and Computer Science,
Saint-Petersburg State University,
Candidate of Physics and Mathematics,
Alexander Iu. Avdiushenko

Thesis reviewer:
Assistant Professor,
Department of Mathematics,
"Higher School of Economics" University,
Ph.D.,
Alexander V. Sirotkin

Saint Petersburg
2023

Abstract

The present thesis is devoted to improving the performance of SAT solvers using machine learning methods, particularly, improving the branching heuristic. Current state-of-the-art SAT solvers use deterministic and easily interpretable heuristics, such as VSIDS, whereas one of the promising approaches is to use a deep neural network for this job. The aim of this research is to study the implementation of the popular deep reinforcement learning approach called Graph-Q-SAT and to improve its performance by making several changes to the original implementation. In particular, the main goal is to reduce the amount of branching decisions and the wall-clock time required to solve a SAT instance, which allows for more efficient implementation of SAT solvers in industrial-scale scenarios. It is achieved by embedding the neural network in the MiniSat solver code using the C++ PyTorch API, which was not done in the original research. It is shown that our implementation takes on average almost 4 times less to solve SAT instances from the Uniform Random-3-SAT distribution and outperforms Kissat, a solver more advanced than MiniSat. Furthermore, additional experiments are conducted which show the extent to which our modification improves the performance of the SAT solver.

Contents

1. Preliminaries & Subject Area Research

1.1. **Introduction.** The Boolean satisfiability problem (SAT) is a paramount problem of computer science that impacts various areas of knowledge, including but not limited to software and hardware verification, circuit design, automatic proof checking and cryptography. This is an NP-complete problem [9], which means that any problem from the NP class can be reduced to it in polynomial time, and the problem itself is assumed to be computationally hard and not having a straightforward algorithm that would solve it in a reasonable amount of time in general case. In spite of this, conflict-driven clause learning solving algorithms achieve very prominent results in reducing the time required to solve an instance of the SAT problem by employing various heuristics.The branching heuristic that decides which variable to assign a value during each iteration of the exhaustive search is one of the crucial ones.

In the Graph-Q-SAT [17] paper, a team of NVIDIA and the Oxford university researchers proposed to improve the widely used Variable State Independent Decaying Sum (VSIDS) heuristic [18] by using a deep neural network, particularly a graph neural network, to "communicate" with the MiniSat solver via a GYM [7] environment (implemented in Python) and predict the next variable to branch on during each step based on the current state of the problem. In many cases this approach decreases the number of branching decisions required to solve an instance of the SAT problem as compared to the original backbone solver MiniSat 2.2 [10]. However, as stated by the researchers themselves, it did not show any substantial results in improving wall-clock time, which is a key factor in industrial-scale settings.

In the proposed approach named Graph-Q-SAT++ the implementation of Graph-Q-SAT is studied and several changes are suggested in order to improve the results achieved in the original research. First of all, the deep neural network is embedded in the MiniSat 2.2 solver itself using the C++ PyTorch API and, secondly, experiments with the architecture of the network are conducted in order to shave off some computational time while preserving the quality of its predictions.

The structure of this work. In Section 1 of this thesis we state the Boolean satisfiability problem, provide an insight into CDCL solvers, a class of algorithms commonly used to solve the SAT problem, and study various approaches to increasing the SAT solving efficiency via machine learning methods. Particularly, in Section 1.4 the Graph-Q-SAT heuristic is studied and the relevant results obtained by the authors of the Graph-Q-SAT paper are described in detail. Then throughout Section 2 the architectures of Graph-Q-SAT and Graph-Q-SAT++ are compared and our work is described, which was done to implement Graph-Q-SAT++ and compare its efficiency in terms of wall-clock time with Graph-Q-SAT, plain MiniSat 2.2 and one of the state-of-the-art solvers Kissat. In Section 2.4 we discuss the applicability of our implementation. Section 3 concludes the paper.

1.2. **Boolean Satisfiability Problem & SAT Solvers.** The Boolean satisfiability problem (often abbreviated SAT) is a problem of determining whether there exists a set of assignments for variables in a Boolean formula that would satisfy it. In other words, it poses a task of assigning the variables in a Boolean formula values `TRUE` or `FALSE` in such a way that the formula evaluates to `TRUE`, or proving that there is no such assignment.

The Boolean formula in conjunctive normal form (CNF) is built from Boolean variables, conjunctions (also denoted by AND, $\wedge$), disjunctions (also denoted by OR, $\vee$) and negation (also denoted by NOT, $\neg$) operations. A Boolean formula in conjunctive normal form is essentially a conjunction of a number of clauses (or a single clause), each of which is a disjunction of Boolean literals, each literal being a variable or its negation (or a single literal). When the satisfying assignment exists we say that the formula is satisfiable, or SAT for short, and when it does not we say that it is unsatisfiable, or UNSAT. Example instances of the Boolean satisfiability problem can be seen in Figure (1).

$$(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_4)$$
$$\{\text{SAT}; x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0\}$$

$$(\neg x_1) \wedge (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$$
$$\{\text{UNSAT}\}$$

FIGURE 1. Examples of Boolean formulae with verdicts.

On computers, Boolean formulae in CNF are commonly stored in DIMACS format. The text file containing a formula in this format consists of several lines, each line being one of the following:

- `p cnf <NUMBER_OF_VARIABLES> <NUMBER_OF_CLAUSES>` — the problem line (only one per problem, other problem formats such as `sat` may be put instead of `cnf`, of which the parsing algorithm should take notice),
- `c <ANY ASCII TEXT>` — the comment line, which gives human-readable information and should not affect the behaviour of the parsing algorithm,
- `<VARIABLE_OR_ITS_NEGATION_1> ... <VARIABLE_OR_ITS_NEGATION_N> 0` — the clause line, each value in which (except the final `0` denoting the end of the clause) corresponds to a literal present in the clause. For arbitrary ordinal number i of the variable, such value equals i if the literal is equal to this variable without negation, and $-i$ otherwise

and possibly a line containing a single `%` or `0` sign, explicitly denoting where the formula ends. The DIMACS representations of formulae from Figure (1) are presented in Figure (2).

One of the current standard methods of SAT solving is using conflict-driven clause learning algorithms, or CDCL solvers for short. During each of its iterations a CDCL solver decides on a variable to pick and assigns it a binary value (i.e. `TRUE` (1) or `FALSE` (0)). This step in usually referred to as branching. Then during *propagation* the solver tries to simplify the

```
1 p cnf 4 4
2 -1 -2 0
3 1 3 -4 0
4 2 -3 4 0
5 -4 0
```

```
1 p cnf 4 4
2 -1 0
3 1 2 0
4 -2 3 0
5 1 -3 0
```

FIGURE 2. Boolean formulae in DIMACS format.

formula and builds an implication graph that holds information about the causal relationships between assignments. If a conflict emerges during propagation, the implication graph is used to infer new clauses and backtrack to the decision after which the newly inferred clause becomes unit, i.e. having a single unassigned literal. This core heuristic is otherwise known as non-chronological backtracking. Various heuristics are also implemented in CDCL solvers, such as:

- pre-assigning literals that appear in the formula only in one polarity (always with or without the negation operator),
- introducing priority on learned clauses and forgetting clauses with lower priority after the amount of learned clauses becomes too large,
- using a separate algorithm dedicated to picking the variables to branch on

and so on with numerous modifications that improve the performance of the solver in some scenarios. For example, at the time of writing the VSIDS [18] heuristic is one of the most popular branching heuristics among its various modifications. This heuristic keeps an activity value for each variable and picks a variable with the maximum activity value during branching. The main idea of VSIDS is that the activity value of a variable is increased additively each time it is involved in a conflict and multiplicatively decreased at regular step intervals.

1.3. **Machine Learning for SAT.**

1.3.1. *Portfolio Algorithms.* One of the earlier ideas was to use machine learning to perform an algorithm selection task, i.e. train a model that would predict what SAT solver to choose for a SAT instance. This is the core idea of SATzilla [25]: a portfolio-based algorithm that employs machine learning models to predict the runtime of a solver on a given SAT instance based on the hand-crafted features extracted from the instance. The estimated runtime values are used to select a subset of solvers that should be ran to solve the problem. Despite heavily relying on the power of solvers it uses, SATzilla combines them into a considerably more effective algorithm than each of the solvers is by itself, and by doing so it won SAT Competition Awards e.g. in 2007 and 2009.

Loreggia et al. [19] went further and used a deep neural network to construct the features of a SAT instance not by hand, but by converting a SAT instance given in a text format into a grayscale square image, which is afterwards fed to a convolutional neural network that predicts which solver should be applied. Even though such approach is very unusual, the goal of this study was straightforward: to automate the feature selection process that, if performed manually, requires a tremendous amount of human expertise. Although their idea did not yield an algorithm which would set a new state of the art, their implementation did improve the set baseline of choosing the solver which on average performs best on the problems from the training dataset and executing it on all of the problems from the testing dataset.

1.3.2. *Graph Neural Networks: Architectures and Implementations.* To understand approaches to SAT solving featured in this section and further in the present thesis, it is necessary to determine the format for Boolean formulae to be presented in which is interpretable and keeps all the important information about it at the same time. First of all, the problem may be of arbitrary size, making the application of neural architectures which perform inference on input

of fixed size challenging from the very beginning. For example, convolutional neural networks which are very popular and have found their application in a myriad of tasks, are inapplicable to this problem. Moreover, the structure of the problem may change drastically in terms of the amount of clauses during the process of finding the solution, as we keep adding learned clauses after each conflict. Finally, the chosen structure should be permutation-invariable due to the basic logic of two instances being equivalent if one of them can be turned into another simply by rearranging clauses in the formula or literals within the same clauses and by permuting the variables. With this, recurrent neural networks, which are otherwise very commonly used to solve sequence processing tasks, may face a serious struggle, as well as other architectures working with data samples of complex structure, e. g. recursive neural networks [22].

Indeed, a common way to represent an instance of a SAT problem, which allows to address the specifics listed above, is a graph. More precisely, we may build an undirected bipartite graph having a node for every literal, a node for every clause and an edge between every literal and a clause that it can be found in. An approach to building a neural architecture for training and performing inference on graph-structured data is described below.

Graph Neural Network. We denote the set of nodes (variables) by V and the set of edges by E, then we denote the i-th graph node by v_i and the edge connecting the i-th and j-th nodes by e_{ij}. Now, suppose that we are given a graph with N types of nodes and M types of edges. The first step is to look up an embedding (for our purposes, it is enough to think of it as a learnable vector representation) of the corresponding type for each node v_i, which we denote by $\mathcal{V}_i$, and for each edge e_{ij}, which we denote by $\mathcal{E}_{ij}$. This can be simply performed by multiplying the matrix of all embeddings with a one-hot representation of the node or edge type (the matrix dimensions are $D_v \times N$ for nodes and $D_e \times M$ for edges, where D_v and D_e are the lengths of embeddings for nodes and edges correspondingly). Also, during training we learn a global embedding $\mathcal{U}$, the purpose of which will be explained later in this section. Then we apply a graph network layer as follows:

- update the embedding of each edge e_{ij} as follows

$$\mathcal{E}_{ij}^{new} = f_{\mathcal{E}}(\mathcal{U}, \mathcal{E}_{ij}^{old}, \mathcal{V}_i^{old}, \mathcal{V}_j^{old})$$

- update the embedding of each node v_i as follows

$$\mathcal{V}_i^{new} = f_{\mathcal{V}}\big(\mathcal{U}, \mathcal{V}_i^{old}, \rho_{\mathcal{E}\to\mathcal{V}}(\{\mathcal{E}_{ji}^{new} \mid e_{ji} \in E\})\big)$$

- update the global embedding $\mathcal{U}$ as follows

$$\mathcal{U}^{new} = f_{\mathcal{U}}\big(\mathcal{U}^{old}, \rho_{\mathcal{E}\to\mathcal{U}}(\{\mathcal{E}_{jk}^{new} \mid e_{jk} \in E\}), \quad \rho_{\mathcal{V}\to\mathcal{U}}(\{\mathcal{V}_j^{new} \mid v_j \in V\})\big)$$

The message passing algorithm is very flexible, because it leaves room for selection of update functions $f_{\mathcal{E}}, f_{\mathcal{V}}, f_{\mathcal{U}}$ and aggregation functions $\rho_{\mathcal{E}\to\mathcal{U}}, \rho_{\mathcal{E}\to\mathcal{V}}, \rho_{\mathcal{V}\to\mathcal{U}}$ with the only restriction of these functions having to be differentiable, however, usually multi-layer perceptrons (MLPs) are chosen as the update functions and the element-wise sum and the element-wise mean functions are used as the standard gather functions. Moreover, in order for it to be possible to reuse the same graph network layer multiple times, the original embedding often shares its length with the updated one. Now, from the formulae featured in the steps of the message passing

algorithm, the importance of adding the global embedding $\mathcal{U}$ can be observed. Particularly, if we remove $\mathcal{U}$ from the right-hand side of the edge update and the node update formulae, it will only be possible to pass information between neighbouring nodes and edges. Therefore, to carry the signal between every pair of nodes and every pair of edges an amount of layers more or equal to the width of the entire graph is required, which at least for some problems is unacceptable in terms of computation time. In a sense global embedding may be considered as an extra entity, which neighbours all nodes and edges of the graph, offering a shorter path to carry the signal through. Nevertheless, using this workaround is not imperative for every problem. The resulting node, edge and global embeddings can then be used for node-level, edge-level and global classification and regression tasks by applying a separate network that is dedicated for solving the specific target problem. The present architecture can be modified in a variety of ways, for example, the first two steps of the message passing algorithm may be swapped around, still yielding an algorithm with a permutation invariant result.

All in all, the flexibility and graph-like nature of message passing networks allows them to be applied to different tasks in various areas of knowledge including medicine [1], bioinformatics [29], social network analysis [11, 23] and even image processing (for instance, human-object interaction [20] and pose estimation [26, 28]).

Returning to the area of SAT solving, an implementation of a message passing neural network named NeuroSAT [21] is also among approaches to build an end-to-end ML-based solver. It is trained from a single bit of supervision, which encodes whether the instance is satisfiable or not. Such network trained on a distribution of SAT instances possesses multiple noteworthy qualities: the solution can be almost always decoded from the activation values of the network, it can generalize to larger and more difficult problems simply by increasing the number of message passing iterations, or even to entirely different domains, and it can also look for certain contradictions in the SAT problem to guess UNSAT with a similar property of allowing to find the variables involved in the contradiction from the activation values of the network, which allows for more efficient construction of a resolution proof, when the contradiction involves a small subset of variables. In Figure (3) it can be observed how the activation values for each variable change during the execution of message passing. After the required amount of message passing iterations is performed, the mean of the "votes" of the literals is calculated by applying a separate MLP to final embeddings of nodes that correspond to literals, and it is treated as a logit of the predicted satisfiability of the instance. The model parameters are trained by minimizing the binary cross-entropy loss between the predicted satisfiability and the true label.

Despite the novelty of such approach, the authors saw "no obvious path" to improving the state of the art with this architecture. The main problem of using an end-to-end SAT solvers based entirely on machine learning methods is understandable: as any machine learning model, such solvers are uninterpretable, and in practice leave room for making mistakes, which means that they do not solve the problem per se.

However, instead of building a whole SAT solver from scratch, **one can modify an existing solver by implementing an ML-based heuristic**, which would only interfere with the solution finding process and not with the way the verdict is carried out. Such heuristic was

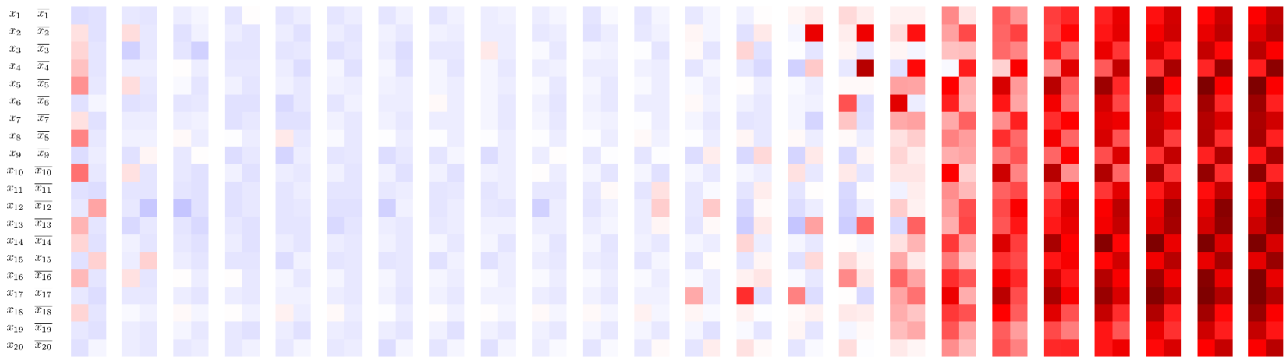


FIGURE 3. An image from the NeuroSAT paper, showing a sequence of the activation values of the message passing network as the iteration number increases from left to right. Each cell corresponds to a certain variable or its negation, colors correspond to the votes of literals on whether the instance is satisfiable (blue represents negative, red represents positive and white means zero), while higher saturation means higher confidence.

developed by the authors of the NeuroCore [6] paper. NeuroCore is a branching heuristic that uses a NeuroSAT-inspired model to predict how likely is each variable in the formula to be present in an unsatisfiable core, and periodically uses these values instead of the activity values of VSIDS. The main assumption of this approach is that it is beneficial to branch on variables that have a high chance of being present in an unsatisfiable core. Several widely used solvers substantially increased their performance on the SATCOMP-2018 [15] dataset after being modified to query NeuroCore. Moreover, if trained and tested on a dataset of formulae from a specific distribution, NeuroCore yields even better results. This quality makes it more likely for this heuristic to find an application in an industrial environment, since in numerous areas we indeed have to deal with specific Boolean formulae, which occur as a result of translating a certain combinatorial task, e.g. graph colouring, into SAT.

Authors of NeuroComb [24] expanded the idea of NeuroCore and devised an ML-based heuristic that uses a graph neural network to predict "important variables" and "important clauses" once before the solution process begins and use these predictions to guide the solver, the important variables being of two types: those that appear in unsatisfiable cores and those that have the same value among all satisfying assignments, and important clauses being those which participate in conflict resolutions more often than others. Finding important variables and important clauses are both treated as node classification tasks, one on variable nodes and the other on clause nodes respectively, and separate GNNs are trained and used to perform the two tasks.

Han in their research [13] introduced the glue variable prediction and glue level minimization heuristics, implemented via a trained graph neural network. The main idea of both heuristics revolves around a key feature of the Glucose [2] series of solvers, that introduces a value called "glue level" for every clause learnt during conflict analysis, which is used to reflect the quality of a clause. The lower the glue level of a clause is, the more important it is for the solver, and if the glue level is lower than 2, it is never removed from the learnt clauses database (such clauses are called "glue clauses"). Both heuristics were implemented into a state-of-the-art solver CaDiCaL [3, 4, 5], and the first heurstic was trained and tested on problems used in

the the SAT Competition[1] while the second was trained and tested on a dataset of SHA-1 preimage attacks in the form of Boolean formulae. The results of both tests show an increase of the efficiency of the solver.

Although the results shown in the studies of these heuristics are positive, their effectiveness relies on already made assumptions about variables or clauses of certain types being more valuable that others, and GNNs are just used as a means to implement them. Further in the thesis, we consider a broader approach that solves the problem of obtaining a branching heuristic more straightforwardly — by training a graph neural network that actually predicts the variable to branch on and the value that should be assigned to it.

1.4. **Graph-Q-SAT: The Original Approach.** As seen in the previous section, effective SAT solving is a very challenging task, the base solution of which relies on performing a time-consuming exhaustive search among all possible solutions, and many approaches have been tried to advance the current state-of-the-art for this problem, among which machine learning methods have shown themselves to be promising. However, machine learning models (when directly applied to the task) are known to be notorious for their "probabilistic" nature, in other words, one cannot guarantee that that an end-to-end SAT solver implemented via a deep neural network, for example, can yield a correct verdict in every single case despite possibly being more time efficient than deterministic models.

Therefore a question arises of the possibility of implementing ML-based heuristics in existing solvers that will optimize the solution process of a solver in some cases, at the same time keeping it complete, i.e. always yielding a correct verdict.

Looking for an answer to this question, in this section we look at one such approach and try to improve it (Section 2) in terms of wall-clock time, which is one of the most important industrial-scale criteria.

1.4.1. *Idea and Implementation Details.* In Section 1.3.2 we have taken a look at some examples of ML-based branching heuristics, which do not make an impact on the verdicts carried out by the solver they are implemented in. In spite of this, they help the solver during the solution process by hinting it which variables it should focus on during branching steps.

A natural approach for building a branching heuristic that utilizes machine learning methods is to consider the problem of choosing a branching heuristic as a reinforcement learning (RL) problem. An RL problem is often posed as follows: having a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a reward function $\mathcal{R}(s'|s,a)$ and the transition probability function $\mathcal{P}(s'|s,a)$, where $s, s' \in \mathcal{S}, a \in \mathcal{A}$, we wish to obtain an optimal policy $\pi(a|s)$ that would maximize the expected discounted cumulative reward $R = \sum_{t=0}^{\infty} \gamma^t r_t$, where $\gamma \in [0,1)$ is a discounting factor and $r_t = \mathcal{R}(s_{t+1}|s_t, a_t)$ is the instantaneous reward after performing the t-th action. Here and thereafter the triple (s, a, s'), just like the triple (s_t, a_t, s_{t+1}) denotes the current state, the action that was or is to be taken and the resulting state (the sequence may be continued by either a', s'', a'' or $s_{t+1}, s_{t+2}, a_{t+2}$ and so on).

In case of creating the environment for a branching heuristic a state $s \in \mathcal{S}$ represents the current state of the solver, i.e. currently assigned variables, given and learnt clauses, and other

[1] http://www.satcompetition.org/

information, some of which may be exclusive to a certain implementation of the solver. A set of actions $\mathcal{A}$ consists of all possible decisions the heuristic may guide the solver with, which are given the assignment **TRUE** or **FALSE** for any unassigned variable. Since we demand that the heuristic guides the solver to make its verdict in the least amount of decisions possible, we should "punish" it more for longer solving sessions, therefore our reward function $\mathcal{R}$ yields a constant negative reward for reaching any non-terminal state, i.e. the state from which a verdict cannot be deduced without performing more branchings. Otherwise it yields a reward of zero. When describing the transition probability function $\mathcal{P}$ it should be noted that SAT solvers akin to MiniSat [10] are deterministic algorithms, which after picking the same variable during branching in the same state will always lead to the exact same resulting state. Therefore in case of such solvers the distribution is degenerate (however it is not guaranteed in a broad sense).

Even having a certain tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$ the question still remains how do we find an algorithm to "train" an agent to act in the given environment. The authors of Graph-Q-SAT [17] decided to apply DQN, a Q-Learning algorithm which involves training a deep neural network to infer the action that would yield the largest expected reward in the current state. More formally, Q-Learning involves approximating a function Q^*, which estimates the discounted sum of rewards after performing action a in state s and acting in accordance with current optimal policy afterwards. The optimal policy itself is achieved by choosing actions which yield the largest expected reward, according to the Q^* function. This can be rewritten in the form of a Bellman equation

$$Q^*(s,a) = \mathbb{E}_{\mathcal{P}}\big(\mathcal{R}(s'|s,a) + \max_{a'} Q^*(s',a')\big)$$

and the optimal policy π^* being a degenerate distribution with support only in action a^*, which yields the maximum value of Q^*.

The neural network is trained by minimizing a temporal difference loss $L(\theta) = \big(Q_\theta(s,a) - (r + \gamma \max_{a'} Q_{\theta'}(s',a'))\big)^2$, where Q_θ is a Q-function approximated by a trained network and $Q_{\theta'}$ is approximated by a network with exactly the same architecture with fixed parameters, which are periodically updated by copying the parameters from Q_θ. In order to expose the trained model to more problems more effectively, the length of a training episode on each SAT instance is limited to a constant threshold, after which the training algorithm moves on to the next problem even if the current problem is still not solved.

The representation of SAT instances in graph form is very similar to one discussed in section 1.3.2 with some modifications, e.g. instead of a node for each literal there is a node for each variable, and the nodes representing variables and nodes representing clauses are connected by two types of edges, depending on whether the variable is negated in the clause or not. An example of such graph can be seen in Figure (4). It is important to note that the graph representation only includes currently unsatisfied clauses, in which only unassigned variables occur. This is done to rule out any possibility of the heuristic picking an already assigned variable. Also, the global embedding of an encoded SAT instance always holds a scalar zero value.

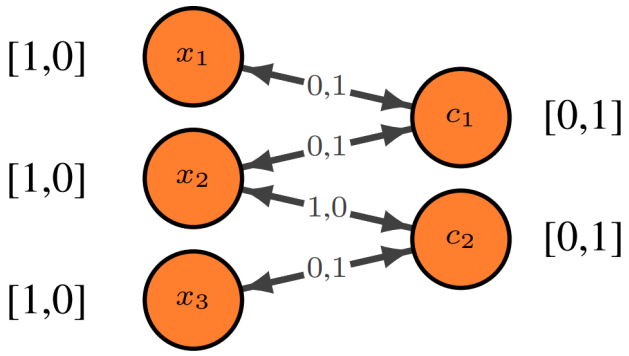


FIGURE 4. A variable-clause graph representation of the formula $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$ used in Graph-Q-SAT. For clearer understanding, the clauses correspond to the nodes named c_1 and c_2. Two numbers near each node and on every edge show the one-hot vector that is used to encode their types.

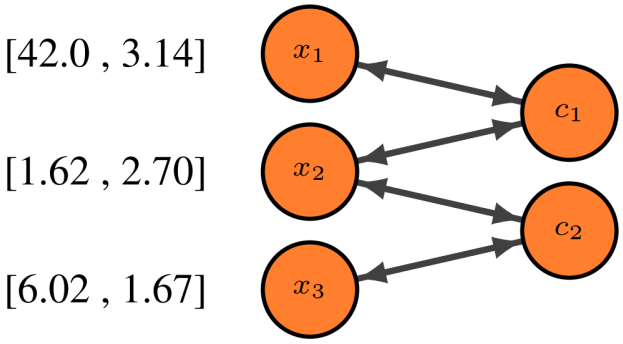


FIGURE 5. The values of the Q-function approximated by the deep neural network. There are two values that correspond to each variable node: the estimated rewards for choosing it to be `TRUE` or `FALSE`.

The architecture of the Q-function approximating model follows an encoder-core-decoder principle. The model is comprised of three graph neural networks, and its basic idea is somewhat similar to that of a recurrent neural network. The first model, which is the encoder, is an independent graph network, i.e. it performs inference on all embeddings as they are without message passing. The second model, or the core, performs multiple inferences with message passing and gathers the output of the encoder and its own current output before each message passing step: a trick very similar to adding residual connections in deep neural networks first featured in the ResNet [14] architecture. After a certain amount of message passing iterations, the final output of the core is fed to the decoder, which makes two predictions for every node in the graph. However, only nodes that correspond to variables are considered and the two predictions for each of them are interpreted as the estimated rewards for assigning the `TRUE` or `FALSE` Boolean value to the variable that the node represents. Figure (5) shows a graphical example of inferred Q-function values.

1.4.2. *Experimental Results.* Various experimental results can be found in the original Graph-Q-SAT paper, most of which utilize Median Relative Iteration Reduction (MRIR) as the main performance metric. **MRIR equals the median of ratios of the amount of branchings performed by plain MiniSat against that of the version of MiniSat which queries Graph-Q-SAT over all SAT instances featured in a certain dataset.**

According to the **MRIR** metric, the proposed approach does improve the efficiency of the solver upon the modification of VSIDS used in MiniSat in terms of the required amount of branching decisions. Moreover, the trained model performs well on datasets containing both SAT and unSAT problems and problems with more variables and clauses, albeit sampled from a similar distribution, as seen in Figure (6).

In the paper it is stated that decisions made by Graph-Q-SAT result in more propagations and it is shown that the exposure of the problem as a whole to Graph-Q-SAT plays a huge role in this as opposed to VSIDS having to warm up before its activity values actually begin to effectively guide the solver. The experiments conducted to show this involve varying the

dataset	mean	min	max
SAT 50-218	2.46	2.26	2.72
SAT 100-430	3.94	3.53	4.41
SAT 250-1065	3.91	2.88	5.22
unSAT 50-218	2.34	2.07	2.51
unSAT 100-430	2.24	1.85	2.66
unSAT 250-1065	1.54	1.30	1.64

FIGURE 6. A table from the Graph-Q-SAT paper showing the mean, minimum and maximum MRIR values over five models trained on problems from the SAT 50-218 dataset of SATLIB [16] benchmark. All the featured datasets belong to the same benchmark. For the evaluation on SAT 50-218 dataset, a separate subset of problems was used.

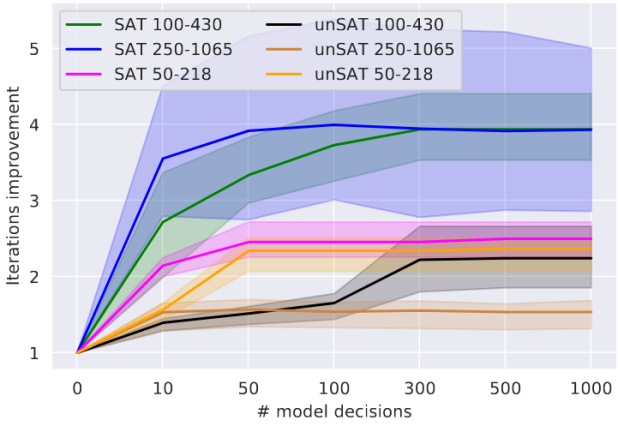


FIGURE 7. A plot from the Graph-Q-SAT paper depicting the relation between the amount of starting iterations and MRIR. Models trained on the SAT 50-218 dataset were used. The graph shows the mean MRIR value, whereas the shade represents the range between the minimum and maximum values of the metric.

amount of times Graph-Q-SAT is queried in the beginning of the solution process and reducing it to extremely low values. As seen in Figure (7) even if queried for as few as 10 times Graph-Q-SAT helps to achieve amounts of decisions several times smaller than the amount plain MiniSat makes.

Although these results seem very promising, it is worth bearing in mind that the MRIR metric is used in a proof-of-concept setting and does not reflect the actual demands of the industry. As the authors of the original paper state themselves, more work is required for Graph-Q-SAT to be applied in an industrial setting.

1.5. **Research Objectives.** Considering the results of Graph-Q-SAT shown in Section 1.4.2, the main goal of this study is to translate the reduction of the amount of branching decisions into a reduction in the wall-clock time taken for the SAT solver to prove (or disprove) satisfiability for SAT problems by making our own improved version of the solver which implements the Graph-Q-SAT heuristic, and study further possibilities of making it more time efficient. In order words, the present thesis is devoted to implementing the means necessary for it to be possible to utilize Graph-Q-SAT in an industrial-scale setting. In Section 2 a re-envisioned version of Graph-Q-SAT, named Graph-Q-SAT++, is presented, which is aimed at reducing the wall-clock time needed to solve the SAT problem.

Dataset	Mean MRIR	Lowest MRIR	Highest MRIR
SAT 50-218	2.59	2.31	2.7
SAT 100-430	3.67	2.92	4.75
SAT 250-1065	4.65	4.21	5.09
unSAT 50-218	2.26	1.99	2.48
unSAT 100-430	2.25	1.89	2.58
unSAT 250-1065	1.39	1.21	1.61

TABLE 1. Our reproduced values of the MRIR metric of Graph-Q-SAT. For evaluation, 100 problems were randomly selected from every dataset. The present results for the SAT 50-218 dataset are for a separated testing subset of problems.

2. Graph-Q-SAT++: Re-envisioning a Popular Approach

2.1. **Reproducing the Original Results.** Before moving on to describing our process of embedding the Graph-Q-SAT heuristic into an existing solver and conducting the experiments required to show the improvements in efficiency, a starting point should be set by reproducing those results from the Graph-Q-SAT paper, which are important to our research.

First, we obtain the datasets from the SATLIB benchmark[2] used in the original study and important to our studies, namely the Uniform Random-3-SAT datasets (which possess an important quality of having the ratio of the amount of clauses to the amount of variables close to 4.26 to 1, making it harder for the solver to make a verdict for problems from them [8]), and clone the Graph-Q-SAT repository[3]. We train the models the same way it was done in Graph-Q-SAT by using the `train.sh` script provided and only changing the arguments that correspond to the directories of the training and validation datasets, trained models, etc. Then we perform model selection on all the models that we saved during the execution of the training script by evaluating them with the `evaluate.sh` script and choosing the one that showed the best results, according to the MRIR metric (see Section 1.4.2). The best model is then evaluated on separate testing datasets using the same evaluation script and the MRIR value it returns is considered the overall metric of the training run. In the same fashion as in Graph-Q-SAT, five training runs are performed. The resulting MRIR values are presented in Table (1).

Afterwards, we obtain a distribution of the Minisat 2.2 solver from the MiniSat website[4], build it according to the instructions and launch execute it on problems from the same testing sets. We save the computational time required for MiniSat to solve every problem and compare it to the time measurements provided by the Graph-Q-SAT evaluation script. The time comparison graph is shown in Figure (8).

Since the authors of the original research did not provide any information about the technical specifications of the machine their tests are ran on, yet the MiniSat solver is fairly light and memory efficient and the trained model is light in terms of VRAM usage, we speculate that

[2] https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html
[3] https://github.com/NVIDIA/GraphQSat
[4] http://minisat.se/MiniSat.html

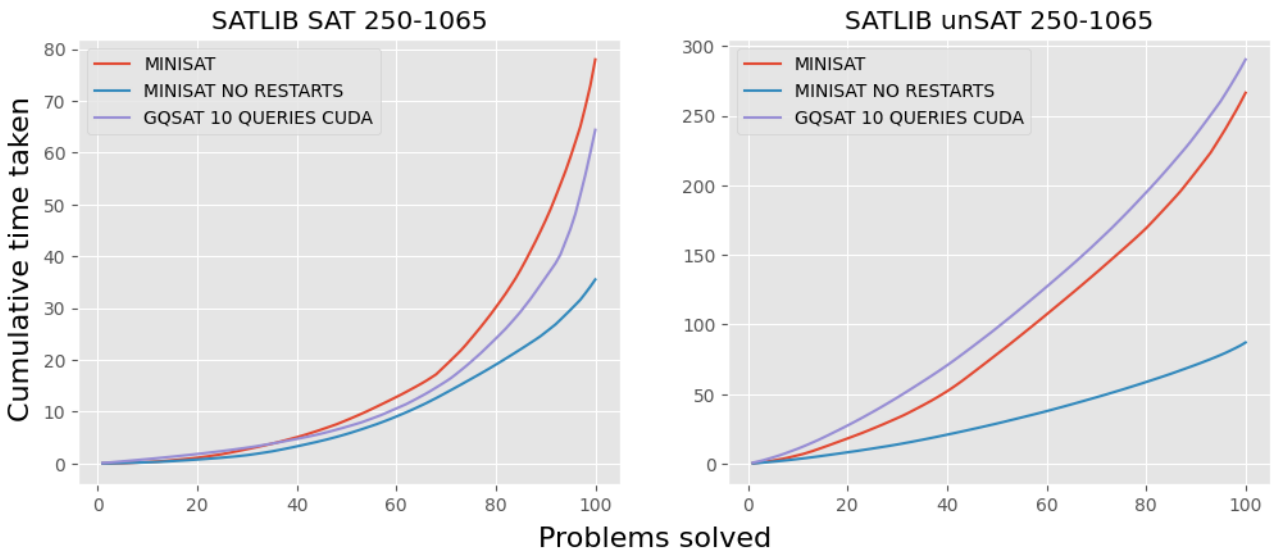


FIGURE 8. A graph similar to one featured in the Graph-Q-SAT paper showing the amount of time it takes for Graph-Q-SAT and MiniSat to solve problems from the SATLIB Uniform Random-3-SAT datasets. It can be observed that Graph-Q-SAT shows little to no improvement over MiniSat in terms of wall-clock time and shows a significant disadvantage in comparison with MiniSat with disabled restarts.

it would be enough for their tests to be ran on a personal computer with a somewhat modern GPU. In our case the total time taken by Graph-Q-SAT is about 20% less than the total time featured in the original research. Because we ran the tests on a personal computer with rather high performance (Intel Core i5-9600K CPU & NVIDIA RTX 3070 GPU), we consider that our measurements are congruent with the measurements provided in the original paper (see Figure 5 in Appendix A[5]). However, the computational times for MiniSat measured by us is very dissimilar to what is shown in the Graph-Q-SAT paper. In fact, the comparison of our MiniSat measurements and Graph-Q-SAT measurements shows almost zero improvement of Graph-Q-SAT over plain MiniSat, whereas in the graph provided in the original paper, the improvements are somewhat noticeable.

We also analyze the values of the MRIR metrics of the selected models on the Uniform Random-3-SAT testing datasets, and although they slightly differ from those featured in the original paper (see Figure (6)), they share the same tendencies: as the problems become larger, for the datasets containing satisfiable instances the MRIR metric grows, and for datasets with unsatisfiable instances MRIR reduces. Our measurements are shown in Table (1).

2.2. **Direct Implementation Into a SAT Solver.** In this section of the thesis the technical side of the work is described, i.e. what has been done in order to build a SAT solver which utilizes the Graph-Q-SAT heuristic.

Before beginning to modify the SAT solver to implement the heuristic in question, the original Graph-Q-SAT code had to be modified, specifically the file containing the code for the neural modules which comprise the graph neural network for inferring the branching decision. The reason for having to modify the code in the first place was to make it convertible in a format

[5] https://web.archive.org/web/20220320153903/https://arxiv.org/pdf/1909.11830.pdf

that could be directly embedded into the C++ code of the solver. The TorchScript[6] framework, created by the PyTorch team, provides two means for converting a model into such format (namely the script module format): *tracing* and *scripting*.

Tracing method is very easy to use, as it creates a script module simply by taking a sample input from the user, performing inference on the values provided and then storing the computational path, which these values underwent. This method can be run on practically any neural module, as long as it does not rely on any unsupported third party libraries, therefore this method requires little to no changes. However, this method in **inapplicable to any neural architecture without a fixed computational path, in particular, to graph neural networks**, for the reason that the traced version of such architecture is only able to perform inference on data of the exact same structure.

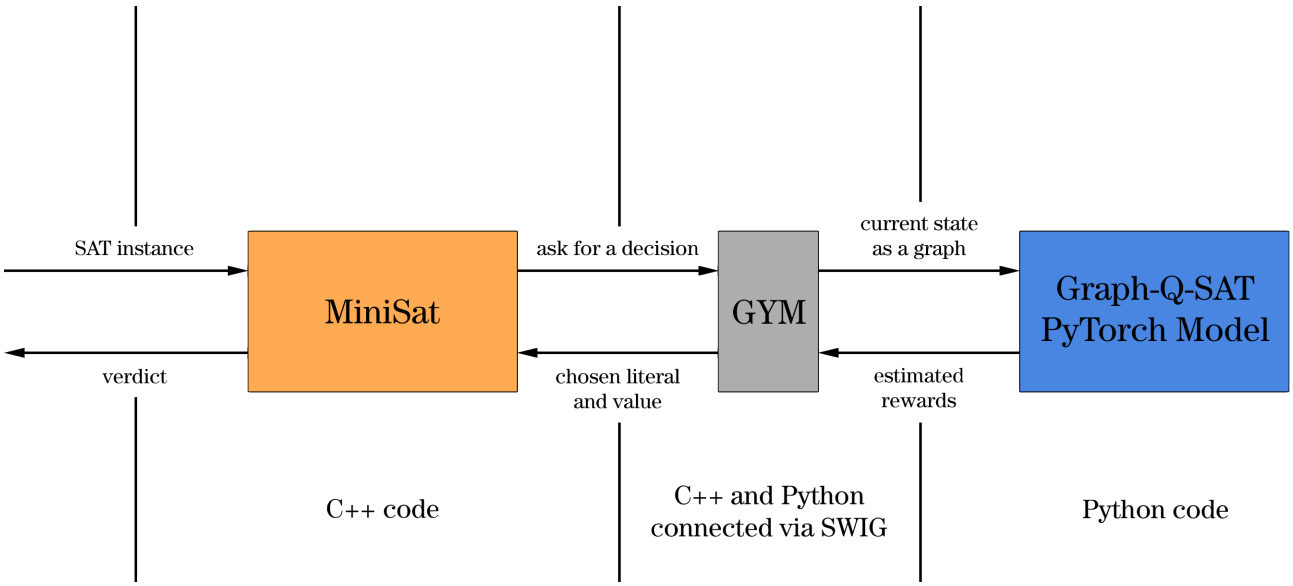
Therefore the original modules had to be scripted. The scripting technique recursively analyzes the Python code of the PyTorch module and transforms it into code written in language used by TorchScript. The main difficulty of applying this method resides in this inner TorchScript language being strongly typed, unlike Python language which is dynamically typed and offers much more freedom for the programmer. In the case of Graph-Q-SAT to ensure this **a considerable portion of the code of the neural modules had to be rewritten and expanded** during the process of constantly dealing with newly encountered errors returned by the scripting algorithm, many of which are practically impossible to find a readily available solution for.

The next step, after modifying the code of the Graph-Q-SAT framework in a way that made it possible to save the branching model in the required format, was to implement the code necessary for the solver to make use of the model by querying it during the branching step.

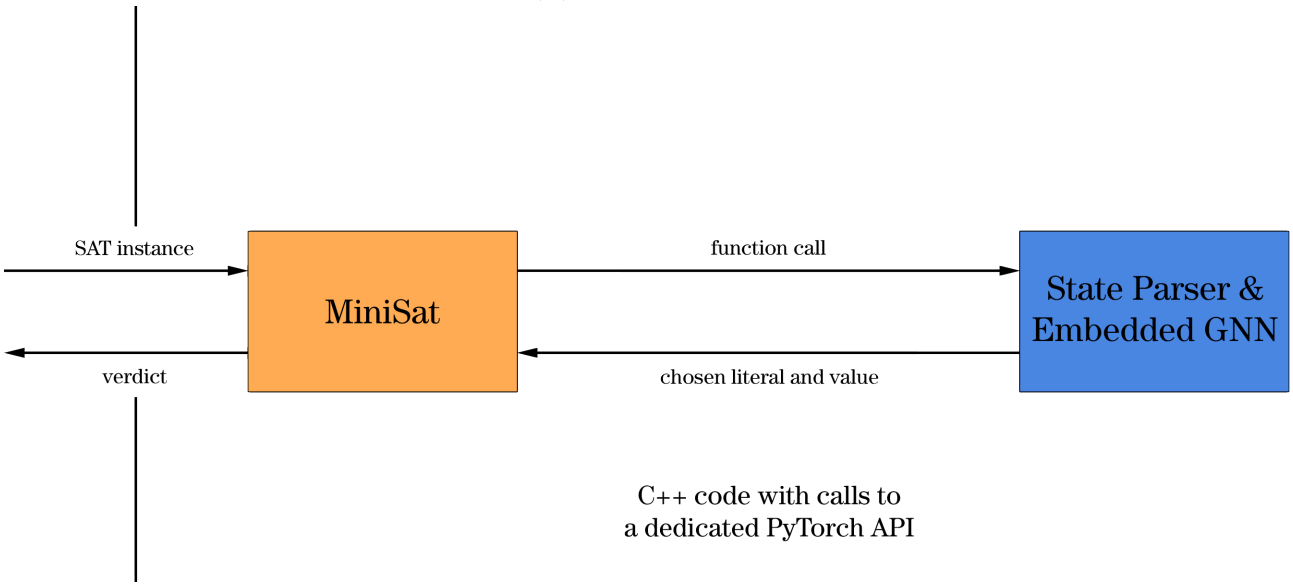
In spite of the obvious fact that there was an option to solely rely on the code of the training environment in Graph-Q-SAT, please note that the structure of this training environment is very different from how Graph-Q-SAT++ was envisioned (see Figure (9)). Most of the code added to the solver in Graph-Q-SAT is written is Python, and instead of being directly called in the solver this code "communicates" with the solver via SWIG[7] — a framework which generates code to allow such connections between the C/C++ code and code written in modern high-level programming languages. Because of this, a large portion of Python code, including the code which parses the current state of the solver and prepares input for the GNN based on it, had to be rewritten in C++ as additional functionality of the solver. Moreover, **each of the additions** made to the actual code of the solver by the authors of Graph-Q-SAT **had to be considered and analyzed** in order for us to understand whether they had to be carried over to Graph-Q-SAT++ and modified beforehand.

Also, the essential albeit not the hardest part of enhancing the solver with the ability to query the script module was to study the C++ PyTorch API used for this purpose. Even though a minor portion of the code uses it, calls to this API are **much less intuitive** than the code usually written in Python which utilizes PyTorch, therefore modifying so that the solver could make the required calls still posed a challenge.

[6] https://pytorch.org/docs/stable/jit.html
[7] https://www.swig.org/



(A) Graph-Q-SAT



(B) Graph-Q-SAT++

FIGURE 9. Layouts of architectures of Graph-Q-SAT [17] (A) and Graph-Q-SAT++ (B) (ours). Before making a verdict, MiniSat may query the network multiple times, each time having its current state read via SWIG and transformed by a parser written in Python and sent as input to the model implemented with PyTorch. In our implementation the main goal is to drastically simplify the architecture by embedding the neural network into the SAT solver and enhancing the solver itself with the functionality required to provide it with the ability to query the network directly, e.g. the state parser.

2.3. **Experimental Results & Improvements.** We conducted multiple experiments in order to confirm that the modified solver allows to solve the SAT problem more efficiently in terms of walltime, and to see if its performance may be increased further.

In the first experiment, we measure time required by the modified solver to find solutions for problems from various datasets, both publicly available and generated by ourselves. The testing was performed as follows: to estimate the time required for Graph-Q-SAT to solve problems from a certain dataset, we provide a trained model to the standard evaluation script

Approach	SAT 250-1065	unSAT 250-1065	SAT 300-1278	unSAT 300-1278
Graph-Q-SAT CPU	77.77	304.0	683	2477
Graph-Q-SAT CUDA	64.38	290.3	667	2461
MiniSat 2.2	35.49	87.1	260	655
Kissat 3.0.0	29.87	273.7	186	2611
Kissat 3.0.0 OPT	27.77	501.1	130	>3600
Graph-Q-SAT++ CPU (ours)	31.44	70.6	140	418
Graph-Q-SAT++ CUDA (ours)	**22.47**	**62.4**	**123**	**403**

TABLE 2. Overall time taken (in seconds) by algorithms to solve all problems from various testing datasets. For Graph-Q-SAT and Graph-Q-SAT++, time measurements for launches with CPU inference are present as well. Values for the Kissat 3.0.0 OPT row show the time measurements of total time taken by the solver built with the `--sat` flag for satisfiable problems and `--unsat` for unsatisfiable problems.

`evaluate.py` from the original repository and record the times required to solve each problem it returns. To measure the wall-clock time required by plain MiniSat and Graph-Q-SAT++ we introduce miscellaneous changes to the solver which have nothing to do with the implementation of the heuristic, but are essential to provide fair testing grounds for Graph-Q-SAT++ (without affecting the performance of plain MiniSat). These changes are listed and explained in Appendix A. For this experiment we also obtain a distribution of the Kissat solver[8], a modification of which was considered the state of the art as of 2020 [12], and build it with the `--competition` flag.

For testing Graph-Q-SAT, a model was trained on 800 problems from the SATLIB SAT 50-218 benchmark using the exact same code and hyperparameters as in the original research conducted by the authors of the paper. After each 1000 model updates, the model was evaluated on a separate validation set of 100 problems from the same benchmark and the model with the best MRIR (see Section 1.4.2 for definition) was chosen. For Graph-Q-SAT++, we launched a modified version of Graph-Q-SAT, and followed the same training and model choosing algorithm.

The final solving time for MiniSat, Graph-Q-SAT and Graph-Q-SAT++ was then measured on all 100 problems from the SATLIB SAT 250-1065 and unSAT 250-1065 benchmarks and on custom datasets featuring 100 satisfiable and 100 unsatisfiable problems from a similar distribution, but with each problem containing 300 variables and 1278 clauses, named CUSTOM SAT 300-1278 and CUSTOM unSAT 300-1278 respectively. The CUSTOM datasets were created using a SAT instance generator for various problem distributions, similar to the one used by Yolcu and Poczos in their research [27]. The results for each problem are shown in Figure (10) while the overall results are shows in Table (2).

[8] https://github.com/arminbiere/kissat

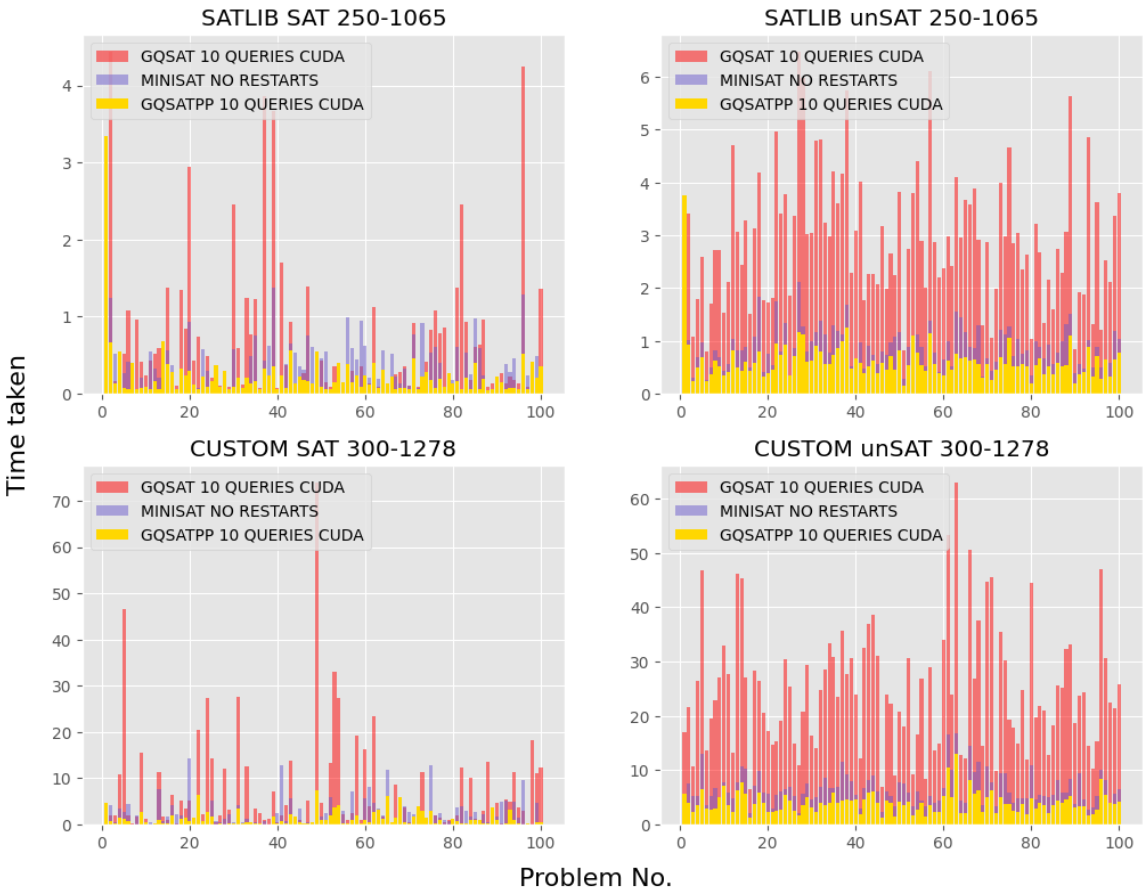


FIGURE 10. Time taken (in seconds) by selected algorithms to solve each of the problems in various testing datasets. The "10 QUERIES" suffix means that the graph neural network was queried for the first 10 branching decisions of the solution process for each problem (instead of the original 500). All solving algorithms were launched without restarts, since this way they yielded better results. For readability purposes, results for Graph-Q-SAT and Graph-Q-SAT++ with inference on CPU are not present since they are slightly worse than these for inference on CUDA.

We then investigate whether the time efficiency of the modified solver may be increased even further by changing the number of branching decisions the ML-based heuristic is queried for at the start of the solution process. Finding the best performing amount of queries is essential in improving performance due to the MRIR to inference time trade-off it introduces. To do so we launch Graph-Q-SAT++ with inference on CUDA, since this method yielded the best results according to the previous experiment, and keep track of MRIR, ARIR (which is the same as MRIR, but equals the mean of all ratios instead of the median) and wall-clock time required (in seconds) to solve all problems from a dataset. The results of this experiment are shown in Table (3).

Although ARIR continues to grow as we make more starting decisions with accordance to the heuristic, performing 10 queries to the neural model at the start of the solution process seems to offer well-balanced trade-off between the decision amount reduction and time consumption and provides the best overall results.

Queries	MRIR	ARIR	Time taken, sec.
1	1.50	1.53	210.5
3	1.99	2.88	159.0
5	2.35	4.44	130.8
10	2.36	10.65	**123.3**
20	2.15	11.80	127.6
40	**2.49**	**11.90**	136.3

(A) CUSTOM SAT 300-1278

Queries	MRIR	ARIR	Time taken, sec.
1	1.21	1.24	539.9
3	1.51	1.57	422.5
5	1.64	1.69	396.8
10	1.69	1.76	402.8
20	1.73	**1.78**	**386.9**
40	**1.74**	1.77	400.9

(B) CUSTOM unSAT 300-1278

TABLE 3. Measurements for varying amounts of Graph-Q-SAT++ starting branching decisions for CUSTOM SAT 300-1278 (A) and CUSTOM unSAT 300-1278 (B) datasets. It can be observed, especially in (A), that the reduction of the amount of decisions is not always followed by the reduction of time taken, and (B) shows that overkill amounts of queries do not produce more considerable decision reduction.

2.4. **Discussion.** Even though our primary goal of increasing the time efficiency of one of the standard and popular SAT solvers was achieved, one should take note that machine learning methods are not some mechanism that is guaranteed to improve any solution, and should be applied with care and awareness. Reflecting upon the experimental results of our research, one should not speculate that a model trained on a set of random SAT instances will have an overall decent performance in guiding a SAT solver while finding solutions for random SAT instances sampled from a different distribution.

Moreover, it is important to choose the right amount of starting queries for each distribution. Our basic recommendation would be to pick a larger number of queries, when trying to utilize a trained GNN on an unfamiliar dataset, as the only disadvantage it brings is the extra overall time required for inference, while picking a number of queries that is too slow results in hardly noticeable reduction in decisions made, as seen in Table (3).

However, we consider our approach to be applicable in an industrial-scale setting, for the reason that there are numerous combinatorial tasks that are important for the industry which are usually solved by being transformed into SAT problems and sent as input to a solver. For a certain task, we may assume that there is a distribution of SAT instances that corresponds with it, and train a model to guide the solver to make verdicts for the problems from this distribution more efficiently.

3. Conclusion

In the present thesis, the domain of SAT solving was explored, and various machine learning approaches to improving the performance of SAT solvers were studied and reviewed, focusing on those which utilize graph neural networks, since they were showing very promising results at the moment of writing. Out of all of the studied heuristics a popular and straightforward approach called Graph-Q-SAT was picked. The approach and its implementation were thoroughly studied and the reproduction of results of the original work of interest was carried out. Graph-Q-SAT was then re-envisioned as an algorithm embedded into one of the standard solvers, MiniSat, instead of being a separate algorithm which communicates with the solver re-implemented as a reinforcement learning environment. The code of Graph-Q-SAT was modified to make it possible for the algorithm to save the trained graph neural network, which predicts the next beneficial decision for the solver, in TorchScript format, a format embeddable into C++ code, and the MiniSat solver was modified to query this network.

The experiments, in which MiniSat, Graph-Q-SAT and Graph-Q-SAT++ are compared in terms of wall-clock time, were conducted. They show that the latter approach is more efficient in this setting. In particular, we observe that Graph-Q-SAT++ outperforms Kissat and shows an average increase in speed of almost 4 times over Graph-Q-SAT on datasets containing problems from the Uniform Random-3-SAT family.

Moreover, additional experiments were conducted which show the extent to which our modification improves the performance of the SAT solver.

In the future, the proposed approach can be released as an open source tool with readily available models pre-trained on larger formulae sets, which could be embedded into various SAT solvers (since a huge portion of them is written in C++), including the state-of-the-art ones, in order to further improve their performance on certain pools of SAT problems.

References

[1] David Ahmedt-Aristizabal, Mohammad Ali Armin, Simon Denman, Clinton Fookes, and Lars Petersson. Graph-based deep learning for medical diagnosis and analysis: Past, present and future. *CoRR*, abs/2105.13137, 2021.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[3] Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2017. In *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*, 2017.

[4] Armin Biere. Cadical, lingeling, plingeling, treengeling and yalsat entering the sat competition 2018. In *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, pages 13–14, 2018.

[5] Armin Biere. Cadical at the sat race 2019. In *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, pages 8–9, 2019.

[6] Nikolaj S Bjørner. 3.3 guiding high-performance SAT solvers with unsat-core predictions. *Deduction Beyond Satisfiability*, page 29, 2019.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[8] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, page 331–337, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[9] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[10] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. *CoRR*, abs/1902.07243, 2019.

[12] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Sat competition 2020. *Artificial Intelligence*, 301:103572, 2021.

[13] Jesse Michael Han. Enhancing SAT solvers with glue variable predictions. *CoRR*, abs/2007.02559, 2020.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[15] Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. SAT competition 2018. *J. Satisf. Boolean Model. Comput.*, 11(1):133–154, 2019.

[16] Holger H Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. *Sat*, 2000:283–292, 2000.

[17] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Can Q-learning with graph networks learn a generalizable branching heuristic for a SAT solver? *Advances in Neural Information Processing Systems*, 33:9608–9621, 2020.

[18] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In *Haifa Verification Conference*, pages 225–241. Springer, 2015.

[19] Andrea Loreggia, Yuri Malitsky, Horst Samulowitz, and Vijay Saraswat. Deep learning for algorithm portfolios. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[20] Siyuan Qi, Wenguan Wang, Baoxiong Jia, Jianbing Shen, and Song-Chun Zhu. Learning human-object interactions by graph parsing neural networks. *CoRR*, abs/1808.07962, 2018.

[21] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[22] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[23] Qiaoyu Tan, Ninghao Liu, and Xia Hu. Deep representation learning for social network analysis. *CoRR*, abs/1904.08547, 2019.

[24] Wenxi Wang, Yang Hu, Mohit Tiwari, Sarfraz Khurshid, Kenneth L. McMillan, and Risto Miikkulainen. Neurocomb: Improving SAT solving with graph neural networks. *CoRR*, abs/2110.14053, 2021.

[25] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Int. Res.*, 32(1):565–606, jun 2008.

[26] Yiding Yang, Zhou Ren, Haoxiang Li, Chunluan Zhou, Xinchao Wang, and Gang Hua. Learning dynamics via graph neural networks for human pose estimation and tracking. *CoRR*, abs/2106.03772, 2021.

[27] Emre Yolcu and Barnabás Póczos. Learning local search heuristics for Boolean satisfiability. *Advances in Neural Information Processing Systems*, 32, 2019.

[28] Ailing Zeng, Xiao Sun, Lei Yang, Nanxuan Zhao, Minhao Liu, and Qiang Xu. Learning skeletal graph neural networks for hard 3d pose estimation. *CoRR*, abs/2108.07181, 2021.

[29] Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. Graph neural networks and their current applications in bioinformatics. *Frontiers in Genetics*, 12, 2021.

Appendix A. Non-Heuristic-Related Changes to the Solver

A couple of changes was made to the original code of MiniSat, which do not affect neither the solver nor the implementation the Graph-Q-SAT heuristic, but maintain an environment similar to one on which the time required by Graph-Q-SAT is measured.

Firstly, the input of the modified solver is a path to a text file containing paths to text files with problems to be solved written in DIMACS format (explained in section 1.2), and the solver is ran on all of them successively in a single execution. The reason for such modification is that PyTorch models are known to perform inference much slower during the warm-up period, and the implementation of Graph-Q-SAT deals with this problem by instantiating an environment with a branching model only once before beginning to find solutions for all the SAT instances provided. By doing so the high inference time during warm-up only affects the efficiency of the algorithm on the first problem.

Secondly, instead of CPU time, the solver measures the real time taken for itself to make the verdict, which is an interval from right before the instance of the solver is initialized in the code and right after the solving function returns its value. This is primarily important for Graph-Q-SAT++, as the inference of the graph network which performs branching decisions on a CPU almost always runs on multiple cores. This affects the CPU time provided by the operating system and makes it substantially larger than the actual time the solving algorithm runs for.