Universiteit van Amsterdam

MSc Artificial Intelligence
Master Thesis

SAT and Exploratory Combinatorial Optimization

by

Fabrizio Galli

12681911

August 12, 2021

48 EC
Period 2-6

Supervisors:
Dr HC Herke van Hoof ,
Matthew Macfarlane

Assessor:
Niklas Höpner

Graduate School of Informatics

Contents

Abstract

With a new graph representation that encodes variable assignments, we present a Reinforcement Learning framework for the Satisfiability Problem, based on the technique of exploratory combinatorial optimization. The proposed method is successful in finding solutions for satisfiable instances of the SAT problem reliably. The introduction of this classical NP-Complete problem into the field of exploratory combinatorial optimization displays differentiated characteristics from other contemporary methods, which in turn provides new insights into the field and opens up interesting avenues for future research on the development of combinatorial optimization.

1 Introduction

NP-Hard problems have been conventionally solved using exact algorithms. This type of algorithms often employ a branch and search strategy for exploring the search space and arriving to the desired solutions. This strategy consists of simulating a decision tree, where each node of the tree is a possible state the input problem can take, and the edges of the tree represent the possible transitions between states. The solutions, or desired values, that the algorithm seeks to find are situated at the leaves of the tree. This approach has two, among others, important limitations. One is that these trees are very large, and they become exponentially larger with the number of input variables, so traversing the entire tree may take an exponential amount of time. The other is that the same state may be repeatedly found distributed among different branches of the decision tree, with the algorithm unable to enumerate previously visited states.

The fascinating property of combinatorial optimization (CO), is that each of the possible states that an input problem can take is directly mapped the value sought to optimize, through the power of function approximation, and a trajectory from state to state to the optimal solution can be obtained, or at least approximated, using modern techniques of Reinforcement Learning (RL). Furthermore, oftentimes the NP problems that are sought to be solved come from the same underlying distribution. This is of no concern for exact algorithms, but is something that machine learning is well known to successfully exploit.

Additionally, a common practice in CO is to represent the problems using graph neural networks (GNNs), which provides the advantage of mimicking the underlaying structure present in the data. This provides an inductive bias that further promotes the ability to *learn* from such problems. Experts argue that this is in fact more akin to human intelligence, where *"[a human mental model] has a similar relation-structure to that of the process it imitates."* (Craik, 1943), and further propose that CO should be a top priority in AI to achieve human-like intelligence (Battaglia et al., 2018).

One NP problem that has not been given its fair share of participation on CO research, considering its prominence in computer science, both historically and in terms of practical applications, is the Boolean Satisfiability Problem, or SAT. Most recent combinatorial optimization based on RL research has been focused on NP-Hard problems, a broader class of problems than NP-Complete (to which SAT belongs). There has been recent research in the application of GNNs for the SAT problem (Selsam et al., 2018), which has shown the capability of these networks to solve this problem, but to date there is no method that employs the combination of GNN and RL for the SAT problem.

This thesis seeks to investigate whether or not the SAT problem can be interpreted as Reinforcement Learning task based on graph neural networks, and if this is the case, resolve the following questions: what design considerations are important to be taken into account when building a RL SAT solving device? Can such RL models produce results that are on par with state-of-the-art SAT GNN methods? Is this problem any different than its NP-Hard counterparts in the context of CO?

In this thesis, the SAT problem is presented in a novel graph interpretation, that, apart from representing the problem itself, it allows the problem instance to take different states, encoding possible variable assignments, therefore making it suitable for RL algorithms. The RL framework is based on exploratory combinatorial optimization, a method proposed by Barrett et al., 2020 which considers NP-Hard problems.

In section 2 of this document, the theoretical foundations of the tools used for this research are presented in detail to the reader. Section 3 gives a broad overview of the current state of research in CO, as well as explaining the current standing of the Satisfiability Problem in this field. On section 4, a detailed explanation of the methodology, of how this method works, is provided. Section 5 discusses the experiments undertaken on this research, as well as the findings. Lastly, section 6 engages the reader into a interesting discussion of the implications of this research, as well as presenting an array of future research possibilities that this study begets.

2 Background

2.1 The SAT Problem

A formula in propositional logic is an expression composed of a set of boolean variables related to each other by a set of logical connectives ($\wedge, \vee, \neg, \rightarrow, \Leftrightarrow$). We say that a formula is *satisfiable* if there exists an assignment for the truth values of its variables, such that it makes the formula as a whole evaluate to *True*. Conversely, if no such assignment exists, the logical expression is said to be *unsatisfiable*. For any formula in propositional logic, there exists an equisatisfiable formula, i.e. a formula that is only satisfiable when the original formula is also satisfiable, given in *Conjunctive Normal Form* (CNF) (Selsam et al., 2018). A formula in CNF is a conjunction of clauses, e.g. $c_1 \wedge c_2 \wedge c_3$, each of which is a disjunction of (possibly negated) variables, e.g. $x_1 \vee x_2 \vee \neg x_3$. The *Boolean Satisfiability Problem* (SAT) is the task of determining if for a given propositional formula given in CNF, there exists an assignment of its variables such that the logical expression evaluates to *True*.

The SAT problem is central in many computer science and artificial intelligence domains, its wide range of applications include theorem proving, planning, non-monotonic reasoning, integrated circuit correctness checking and knowledge-based verification and validation (Audemard et al., 2008). The SAT problem is also of key importance in computational complexity theory and combinatorial optimization, for it is the first problem proven to be NP-Complete (Cook, 1971).

2.2 Reinforcement Learning

In reinforcement learning (RL), an *agent* interacts with an *environment* in a Markov Decision Process (MDP) with the goal of maximizing some reward generated by the environment. Through this MDP, the environment goes through a sequence of changing states, as a result of the *actions* taken by the agent, with the agent receiving a reward after taking each action. Assuming a finite MDP (which terminates after a finite number of steps), an *episode* is generated from the sequence of actions taken by the agent and changes in the environment. An episode initiated in time-step t is denoted by the sequence $s_t, a_t, r_{t+1}, \ldots, r_T, s_T$, where s_t is the state of the environment, a_t the action taken by the agent, r_t is the reward from the current state perceived by the agent, and T the time-step on which the environment reaches a terminating state.

For any episode, one can denote the total rewards perceived by the agent as $\sum_{t=0}^{T} r_t$, also known as the return. If the rewards are discounted through time, so as to give greater weight to immediate rewards, the return at time-step t is given by $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_t$, where γ is the *discount-factor*. The goal of the agent is to maximize R_t.

For any combination of state and actions, or state-action pairs (s, a), the expectation of the return after taking action a on state s is given by the *state-action value function*, also known as Q-value,

$$Q(s, a) = \mathbb{E}[R_t | s_t = s, a_t = a, \pi], \tag{1}$$

where $\pi \sim P(a|s)$ is the policy, a conditional distribution used by the agent as a strategy to select the remaining actions in the episode. The main idea behind

RL is to find a policy π such that the state-action value function is optimized:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[R_t \;\middle|\; s_t = s, a_t = a, \pi\right]. \tag{2}$$

An important identity, known as the *Bellman Equation* (BE), tells us that if we know the optimal Q-value $Q^*(s', a')$ of the state s' on the next time-step, then the optimal strategy is to select action a' (Mnih et al., 2013),

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \;\middle|\; s, a\right]. \tag{3}$$

The Q-values then can be obtained by iteratively updating them until convergence, using this identity as an iterative update, $Q_{i+1}(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q_i(s',a')|s,a]$ (Mnih et al., 2013). Since it is impractical to compute optimal values for all possible state-action pairs, these are often approximated using deep neural networks. A well-known approach of policy function approximation is Deep Q-Network (DQN), where a *policy network* with parameters θ can be trained by minimizing the loss function at every iteration i,

$$L_i(\theta_i)^{\text{DQN}} = \mathbb{E}_{(s,a,r,s')}\left[\left(r + \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]. \tag{4}$$

Here, the term $Q^*(s', a'; \theta_{i-1}^-)$ is computed using a different *target network* with parameters θ^- for stability purposes. The target network parameters θ^- are updated with the policy network parameters θ every C steps, and kept fixed throughout (Mnih et al., 2015).

It has been shown that DQN overestimates the action values under certain conditions which can lead to sub-optimal policies (Van Hasselt et al., 2016). In order to reduce this overestimation, a modification to this algorithm has been proposed, called Double-DQN, where the term $\max_{a'} Q^*(s', a'; \theta_i^-)$ in equation 4 is rewritten as

$$L_i(\theta_i)^{\text{DoubleDQN}} = \mathbb{E}_{(s,a,r,s')}\left[\left(r + Q(s', \arg\max_{a'} Q(s',a';\theta_i);\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]. \tag{5}$$

This rewriting of the loss reduces overestimation and has been shown that it results on more stable and reliable training (Van Hasselt et al., 2016).

Another important class of methods in RL for finding optimal policies are *policy gradient methods*, for which stronger convergence guarantees are available than for action-value methods (Sutton and Barto, 2018). In policy gradient, instead of learning the values associated with the state-action pairs, the aim is to learn a parameterized policy $\pi(a|s,\theta)$ such that maximizes some performance measure, $J(\theta)$. The parameters θ are updated using gradient ascent,

$$\theta_{i+1} = \theta_t + \alpha \nabla J(\theta_t), \tag{6}$$

with some step size α. Via the *policy gradient theorem*, we have an expression for the gradient of performance with respect to the parameters,

$$\nabla J(\theta) \propto \mathbb{E}_{s_t, a_t \sim \pi}\left[R_t \frac{\nabla \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}\right], \tag{7}$$

where R_t is the return as defined above. This expression gives rise to the REINFORCE algorithm (Williams, 1992, Sutton and Barto, 2018) for updating policy parameters:

$$\theta_{t+1} = \theta_t + \alpha R_t \frac{\nabla \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta)}. \tag{8}$$

2.3 Graph Neural Networks and Message Passing

The most common type of neural network in deep learning is the feed-forward neural network, also called *Multi-Layer Perceptron*, where the network takes the as input the full raw signal, in an unstructured manner, and produces an output as the result of a series of linear transformations and point-wise non-linearities. However, some model architectures account for the underlaying structure of the data, thus introducing an *inductive bias* which fosters learning and allows for better results.

Convolutional Neural Networks (CNNs) have become the standard for image classification (Krizhevsky et al., 2012) by using an architecture that exploits the spatial structure of its input data. This architecture introduces location and translation invariance by convolving the input vectors with a sliding kernel of the same rank. Another model architecture, *Recurrent Neural Networks* (RNNs) (Elman, 1990), specializes on data of sequential nature and introduces a temporal inductive bias by using a recurrent block that passes through the input data in a sequential manner. RNNs have found many useful applications, especially in the field of natural language processing.

For the case of combinatorial optimization it would be desirable to account for a permutation invariance inductive bias. This is given by the fact that many problems, NP-hard problems in particular, may have an exponential number of permutations of its input units. For an MLP, each permutation would be considered fundamentally different. Neural networks that operate over graphs provide a strong relational inductive bias beyond that which CNNs and RNNs can provide, these are called *Graph Neural Networks* (GNNs) (Scarselli et al., 2008; Gilmer et al., 2017).

The main component of a GNN is a *graph-to-graph* module, that takes a graph as an input, performs computations over the graph, and returns a graph as an output. Furthermore, these graph-to-graph modules can be stacked on layers (similar to CNNs), or the same module can be used recursively over many iterations (similar to RNNs). We refer this propagation from module to module as *Message Passing*, as the information of node entities in the graph is sent and received by its neighbours on each iteration.

In the GNN framework, a *graph* is defined as the 3-tuple $(\mathbf{u}, V, E)$, where $\mathbf{u}$ are the global attributes of the graph, $V = \{\mathbf{v}\}_{i=1:|V|}$ is the set of vertices and $\mathbf{v}_i$ are the vertex attributes, and $E = \{(\mathbf{e}_i, r_i, s_i)\}_{i=1:|E|}$ is the set of edges connecting the vertices, with $\mathbf{e}_i$ the edge attributes and r_i, s_i the receiving and sending nodes.

On each pass of the graph-to-graph module, a message is produced on a per-edge basis $\mathbf{e}_i = \phi^e([\mathbf{e}_i, \mathbf{v}_{r_i}, \mathbf{v}_{s_i}, \mathbf{u}])$. These messages are further aggregated, with a permutation invariant function ρ, on a per-node basis $\hat{\mathbf{e}}_i = \rho^{e \to v}(M'_i)$, where M'_i is the set of messages for edges with receiving node of index i. After all messages are aggregated, each node updates its features via an update function,

$\mathbf{v}_i = \phi^v([\hat{\mathbf{e}}_i, \mathbf{v}_i, \mathbf{u}])$. In a similar fashion, the global attributes are updated by with the aggregation of all $\hat{\mathbf{e}}_i$, using $\mathbf{u} = \phi^u([\rho^{e\rightarrow u}(\cup_i \hat{\mathbf{e}}_i), \rho^{v\rightarrow u}(\cup_i \hat{\mathbf{v}}_i), \mathbf{u}])$. Note that the above methodology follows the formalization described by Battaglia et al., 2018, but many applications may omit some of these steps or apply a different scheme for graph computations.

3 Related Work

There are three main approaches for solving NP-Hard problems: exact algorithms, approximation algorithms, and heuristics (Hochba, 1997, Goemans and Williamson, 1995). Exact algorithms are guaranteed to find optimal solutions, but may be prohibitive for large problem instances. Approximation algorithms that can find solutions in polynomial time are preferable in terms of efficiency, but these offer weak theoretical guarantees. Heuristics are often fast and efficient, but they also may not provide strong theoretical guarantees and they require large investment of expert knowledge. All of these approaches consider each instance of a problem in isolation and they do not exploit the fact that the problem instances often come from a common real world underlaying distribution. Considering this, a great body of research has been dedicated to the subject of applying machine learning to combinatorial problems (Bengio et al., 2020).

Research on the use of neural networks applied to combinatorial optimization (CO) dates back to Hopfield and Tank, 1985, where a neural model is deployed to solve the Travelling Salesman Problem (TSP). Reinforcement Learning (RL) methods were first applied to CO by Zhang and Dietterich, 1995, with a model designed to solve the NP-Hard job-shop problem (the general case of TSP). In recent years, many studies deploy the use of deep learning for CO, some of these with the use employment of pointer-networks for solving combinatorial problems, including the TSP (Vinyals et al., 2017, Bello et al., 2016, Kool et al., 2018), however, these architectures are generic in design and do not reflect the graph structure that the underlying problems may have.

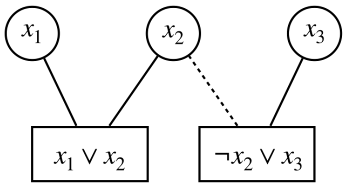
Dai et al., 2017, introduced a method that uses a combination graph neural networks (GNN) and Q-learning (Mnih et al., 2015), giving better results than pior works. In their approach, a graph encoded NP-Hard problem is given to a graph network, called *structure2vec* (Dai et al., 2016), which encodes embeddings for each node in the graph. Each of the node embeddings are subsequently mapped to a Q-value, which is used to incrementally construct a solution set, adding one node to the solution on every time-step of the episode. This method yields a greedy policy that produces a best-guess solution for each problem instance. Barrett et al., 2020 extend on this work, arguing that given the inherent complexity of many combinatorial problems, learning a policy that directly produces a single, best-guess solution is often impractical; They propose a method of *exploratory combinatorial optimization*, ECO-DQN. In this method, each episode begins with a randomly initialized solution set and, on each time-step, an agent adds or removes nodes from this set. This scheme lets the agent revise previous decisions taken during the episode and explore the search space during test time, something that the previous method did not allow, given its irreversible nature. This latter approach displayed significant improvements over its predecessor. For yet another approach on the use of graph neural networks on CO problems, see the tree-search based methods proposed by Abe et al., 2020 and Li et al., 2018.

All the work mentioned so far in this section aim to solve problems in their NP-Hard formulation, meaning that there is no known polynomial time algorithm that certificates that any proposed solution is the optimal one. This is fundamentally different to the Satisifiability Problem, which is a decision problem and known to be in NP. This is of theoretical relevance for the question of P

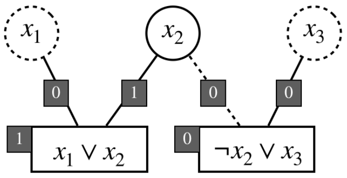
vs NP, given that NP-Hard problems need not to be in NP. Modern SAT solvers based on backtracking search are highly efficient, being able to solve problems with millions of variables (Biere et al., 2009). However, there are numerous studies focused the use of machine learning for tackling this problem. Some of these approaches aim to learn heuristics that may be used by the conventional solvers to increase their efficiency (Liang et al., 2018, Liang et al., 2016). Given that the SAT problem may also be given in a graph representation (Braunstein et al., 2005, Audemard et al., 2008), some approaches have been proposed for the use of GNN to solve the problem directly (Selsam et al., 2018, Bünz and Lamm, 2017). The NeuroSAT model, in particular, proposed by Selsam et al., 2018 has gained recent notoriety and is often cited as the leading example for the SAT problem in the context of CO and GNNs. This method is trained as a classifier (making *sat/unsat* predictions for each problem instance) and was able to correctly label problems with 85% accuracy. Furthermore, the activations of the nodes in the graph network are subsequently used to extrapolate truth values for the variable assignments. NeuroSAT was able to decode satisfiable assignment on 70% of the satisfiable instances. There was not found in literature a *non-supervised learning* method for the SAT problem, nor a method that uses reinforcement learning to directly construct ad-hoc solutions for this problem.

4 Methodology

4.1 SAT Graph Based Representation



(a) SAT factor graph representation.



(b) SAT factor graph with interpretation.

Figure 1: SAT graph representation for an input problem $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. (a) Factor graph representation: edges indicate the incidence of literals in clauses, dashed edges indicate the negation of a literal. (b) Factor graph with interpretation $I = \{\neg x_1, x_2, \neg x_3\}$: dashed outlines on variable nodes indicate the negation of a variable in I. Edges behave as $XNOR$ gates, clause nodes behave as OR gates.

As given by Braunstein et al., 2005, a SAT problem with N variables and M clauses can be represented as a bipartite *factor graph*, $G = (V \cup C; E = V \times C)$, where V, $|V| = N$, is set of *variable nodes* representing the variables of the problem, C, $|C| = M$, is the set of *clause nodes* representing the clauses of the problem, and E is the edge set. An edge exist between a clause node and a variable node if that variable (or its negation) is contained within the clause in the input problem. Additionally, all edges in the graph have a binary label indicating the negation of the literal in the clause.

An extension to the above defined factor graph is here proposed to account for the assignment of truth values for the variable nodes. A *SAT factor graph with interpretation* considers a factor graph G and an interpretation I which contains the truth value assignments of the variables of G. In this graph representation, each variable node has a binary label indicating its truth value according to I. Furthermore, these truth values are propagated across the graph to encode new features on edges and nodes.

First, each of the edges acts as an *XNOR* gate, producing as a result an *edge-sat-value*, by taking as input its own label and the label of its source variable node, this gives a value of 1 if and only if the edge label is the same as the variable node label. Following this, the clause nodes behave as *OR* gates, producing a value of 1 if at least one of its incident edges has a edge-sat-value of 1, and 0 otherwise. These values are referred to as *clause-sat-value* and denoted by $S(c; G, I)$, for some $c \in C$ with a graph G and interpretation I. With these new features encoded on the graph, a *global-sat-value* can be computed for the graph, given by $S(G, I) = \frac{1}{M}\sum_{c \in C} S(c; G, I)$. After a quick inspection, it is easy to note that the global-sat-value is equal to 1 if and only if G is satisfiable and I is a satisfying interpretation of the the variables,

$$S(G, I) = 1 \iff G \textit{ is satisfiable, } I \textit{ is a satisfiable assignment.} \tag{9}$$

.

It is also possible to take this value propagation one step further by computing for each variable node a *variable-sat-value* that denotes the average of the clause-sat-values of its neighbours, given by $S(v; G, I) = \frac{1}{deg(v)} \sum_{c \in \mathcal{N}(v)} S(c; G, I)$, for some variable node $v \in V$ and some interpretation I. With this at hand, an alternative global-sat-value can be computed using $S'(G, I) = \frac{1}{N} \sum_{v \in V} S(v; G, I)$. This alternative global value may provide more variance than the previously defined and follows the property $S'(G, I) = 1 \iff S(G, I) = 1$.

4.2 Feature Encoding and Message Passing

As seen in subsection 4.1, there are a number of readily available features for any input graph G with an interpretation I. In addition to these, some other features are considered for the use in Reinforcement Learning. These extra features ensure that the graph representation describes a markovian state on an episodic setting, as well as (potentially) boosting learning. We refer to these as *primary features*, since they can be inferred directly from the current state of the graph at any point of an episode and they do not depend on any learnable parameters. These features are, at the local level of the graph:

- $\mathbf{v}_i$: primary features for variable node v_i, given by the concatenation of the following values:
  - *variable-sat-value* (see 4.1).
  - Steps in the episode since the variable was last changed.
  - Immediate change of *global-sat-value* (see 4.1) if variable is flipped.*
  - Boolean describing if graph reaches a satisfiable assignment if variable is flipped.*
- $\mathbf{c}_i$: primary features for clause node c_i, given by the *clause-sat-value* (see 4.1).
- $\mathbf{e}_{ij}$: primary features for edge connecting variable node v_i and clause node c_j, given by the *edge-sat-value* (see 4.1).

and at the global level of the graph:

- $\mathbf{u}$: context primary features, given by the concatenation of the following values:
  - Difference from current *global-sat-value* with the best observed so far in the episode.
  - Remaining steps in the episode.
  - Distance to best observed state, in terms of number of variable flips required to reach it.
  - Maximum possible increase of *global-sat-value*.
  - Number of actions that lead to an increase in the *global-sat-value*.*

Items denoted with an * indicate that these features require to look ahead the next states for each of the possible actions. Given that this is an expensive operation, the use of these features is omitted in some of the experiments, and their effect analysed (see section 5). It is worth mentioning that these features are not strictly primary, since they require forward simulation to be computed, but in an abuse of terminology they are considered as such, so as to separate them from the feature embeddings.

Aside from the primary features, each of the nodes of the graph, as well as the graph context, have their own *feature embeddings*, a vector of dimension d of real numbers, which is to be updated via message passing operations. The feature embeddings of the variable node with index i are denoted by $\bar{\mathbf{v}}_i \in \mathbb{R}^d$, those of the clause node with index i by $\bar{\mathbf{c}}_i \in \mathbb{R}^d$, and those of the graph context are denoted by $\bar{\mathbf{u}} \in \mathbb{R}^d$. The initial embeddings of these elements at time-step 0 (before any message passing operation is performed) are given by the functions

$$\bar{\mathbf{v}}_i^0 = \phi^v_{\text{INIT}}(\mathbf{v}), \tag{10}$$

$$\bar{\mathbf{c}}_i^0 = \phi^c_{\text{INIT}}(\mathbf{c}), \tag{11}$$

$$\bar{\mathbf{u}}^0 = \phi^u_{\text{INIT}}(\mathbf{u}). \tag{12}$$

These functions are implemented using three separate neural networks, each with their own set of parameters.

For a message passing sequence of length L, on each step l of the sequence, $0 < l \leq L$, each of the nodes encode a message, also of dimension d, using the functions $\phi^{v\to e}$ and $\phi^{c\to e}$, for variable and clause nodes respectively. Following this, each node aggregates the messages of its neighbours by taking their mean. Finally, each of the nodes is updated using the update functions

$$\bar{\mathbf{v}}_i^l = \phi^v\left(\left[\left(\frac{1}{\deg(v_i)} \sum_{j\in J(\mathcal{N}(v_i))} \phi^{c\to e}(\bar{\mathbf{c}}_j^{l-1}, \mathbf{e}_{ij})\right), \bar{\mathbf{u}}^{l-1}, \bar{\mathbf{v}}_i^{l-1}, \mathbf{v}_i\right]\right), \tag{13}$$

$$\bar{\mathbf{c}}_i^l = \phi^c\left(\left[\left(\frac{1}{\deg(c_i)} \sum_{j\in J(\mathcal{N}(c_i))} \phi^{v\to e}(\bar{\mathbf{v}}_j^{l-1}, \mathbf{e}_{ji})\right), \bar{\mathbf{u}}^{l-1}, \bar{\mathbf{c}}_i^{l-1}, \mathbf{c}_i\right]\right), \tag{14}$$

where $[\cdot,\cdot]$ is the concatenation function, $\deg(\cdot)$ is the degree of a node, and $J(\mathcal{N}(\cdot))$ is the index set of the neighbours. Additionally, the feature embeddings of the context of the graph are updated on each time-step l, $0 < l \leq L$, with a function ϕ^u that takes the aggregation of the clause nodes and variables nodes, and its own features:

$$\bar{\mathbf{u}}^l = \phi^u\left(\left[\left(\frac{1}{|V|}\sum_{i\in J(V)} \bar{\mathbf{v}}_i^{l-1}\right), \left(\frac{1}{|C|}\sum_{i\in J(C)} \bar{\mathbf{c}}_i^{l-1}\right), \bar{\mathbf{u}}^{l-1}, \mathbf{u}\right]\right). \tag{15}$$

The above formulas are based on the framework proposed by Battaglia et al., 2018 described in section 2.3. The main modification to this framework comes from the fact that the SAT graph representation is a heterogeneous graph, or heterograph, that is, the graph contains vertices of different types. Since the clause nodes and the variable nodes are entities of distinct nature, different embedding representations $(\bar{\mathbf{v}}, \bar{\mathbf{c}})$, obtained from different sets of parameters, are used, as seen in equations 13 and 14. Another deviation from their general proposed framework is that the edge attributes, i.e. messages, are not stored on every step of the message passing operation (and used for creating new messages on subsequent time steps). In order to account for this, each edge would be required to hold two embedding representations, one for each direction of the bipartite graph, or to have another function to combine these two into a resulting one. In favor of a less intricate implementation, this is omitted. This approach is in line with the ECO-DQN (Barrett et al., 2020) implementation, where the messages are discarded after each step of message passing. A difference with the ECO-DQN method, apart from the use of a heterograph, is that, here, the update of the context $\bar{\mathbf{u}}$ is performed on every step of message passing, whereas ECO-DQN computes the context on the final step only; This is opted for in order to incorporate more information during the message passing process, which may provide for a better generalization. The message aggregation function used is the simple mean (as opposed to e.g. sum) to account for invariability in number of variables, and degree of incidence of variable in clauses. The primary features of edges, vertices and context $(\mathbf{e}, \mathbf{v}, \mathbf{c}, \mathbf{u})$ are included in all relevant formulas, functioning as skip connections.

For a neural network implementation of the message functions, κ, and update functions, ϕ, there are two possible approaches. The first approach is to have one Neural Network layer for each of the functions, $\{\phi^{v\rightarrow e}, \phi^{c\rightarrow e}, \phi^{v}, \phi^{c}, \phi^{u}\}$, with independent parameters, which are used repeatedly in every time-step l of the message passing sequence of length L. These, thus, constitute a *recursive layer*, which shares parameters across time. The second approach is to have separate neural networks for each of the functions and for each of the time-steps, $\{\phi_l^{v\rightarrow e}, \phi_l^{c\rightarrow e}, \phi_l^{v}, \phi_l^{c}, \phi_l^{u}\}_{l=1:L}$, with independent parameters, which are used only once on each time-step l. This second approach does not share parameters across time and are more akin to a deep network, we refer to this approach as *stacked layers*.

The advantage of the *recursive layer* approach is that the message passing operation does not have to be fixed to a sequence length, which might be desirable for problems of differing sizes. Also networks with recursive layers are more memory efficient, since they require less parameters. A potential advantage of the *stacked layer* approach is that it may provide more expressive power given its deep architecture. Both approaches are to be analyzed within this study.

4.3 Reinforcement Learning Framework

So that a model can learn to find satisfiable assignments for the SAT problem, we define a Reinforcement Learning framework for this problem, closely related to the ECO-DQN method proposed by Barrett et al., 2020. In this framework, a state of the environment constitutes an input graph G, encoding a SAT problem instance, with some interpretation I, so a state at time-step t is denoted by $s_t = (G, I_t)$. The action taken by the agent at any time-step is to flip the value of a single variable in I, thus the number of available actions is equal to the number of variables in the input problem. The Q-values used to select these actions are obtained from stacking a fully connected layer, ϕ^q, on top of the message passing network described on section 4.2. This fully connected layer takes as input the feature embeddings of each node, as well as the feature embeddings of the graph context, and gives as output a real valued scalar, so that the Q-value for choosing node i is given by

$$Q_i = \phi^q([\bar{\mathbf{v}}_i^L, \bar{\mathbf{u}}^L]). \tag{16}$$

The same fully connected layer is used for all variable nodes in the graph, thus sharing parameters, so we have that the network can take problems with any number of input variables. As it is the case for ECO-DQN, the model is trained using the DQN algorithm (Mnih et al., 2015).

An episode always begins with a randomly initialized interpretation I_0 and it consists of a sequence of variable assignment changes, where the agent is allowed to revisit previous states and revise previous decisions. The episodes terminates when the agent has found a satisfiable assignment or when the number of actions taken equals double the number of the input variables. This is different from ECO-DQN, since for the problems this method considers, there is no known way to check if any solution is the optimal one (in polynomial time), in which case, all episodes must run for a fixed number of steps.

For every state s_t observed during an episode, there exists an associated *global-sat-value* (as defined in subsection 4.1), denoted by $S(s_t)$. Using this, a reward is computed for every step in the episode, given by

$$R_t = \max(0, S(s_t) - S(s^*)), \tag{17}$$

with s^* denoting the state that has given the highest global-sat-value so far in the episode. Under this scheme, the agent only observes positive rewards when taking steps that take the environment closer to a solution than it has been before. In their paper, Barrett et al., 2020 assign an additional reward to the ECO-DQN model when the environment reaches a local-maxima, so to encourage the model to visit local-maximas more often. This requires to look ahead one step forward, once for every action and every time-step, and propagate the graph values each time, which may impair efficiency. For the case of the SAT problem, however, any local-maxima that is not also a global-maxima, i.e. $S(s_t)$ equal to 1, is known not to be a solution for an input problem. Thus, the question arises of whether this local-maxima reward has an effective benefit in the case of the SAT problem, this will be further expanded on section 5. Another adjustment to the reward that may be considered for decidable problems is to give the agent an additional reward when the optimal solution is found, this is also discussed in section 5.

To observe whether or not over-estimations produced by the DQN method may result on sub optimal policies, the model is also trained using the Double DQN method (Van Hasselt et al., 2016) and the policy gradient method REINFORCE (Williams, 1992).

4.4 Experimental Data

The main dataset considered for this study is *Random K-SAT*, the most popular model for generating instances of the SAT problem (Boufkhad et al., 2005). The instances generated by this model are gotten by selecting m clauses from the set of all possible $2^k\binom{n}{k}$ clauses with k literals on a given set of n variables. As it is most commonly employed in research, k is set to a value of 3. The instances generated from this model display a *phase transition*, that is, a region on which the generated instances transition from being mostly satisfiable to mostly unsatisfiable, as a function of the ratio α between the number of clauses to the number of variables. This region, for the case of Random 3-SAT, is bounded by $3.52 \leq \alpha \leq 4.506$ (Boufkhad et al., 2005), so a value of 4 is assigned to α for the problems generated, in order to obtain sufficiently hard problems for the model to solve. Furthermore, only satisfiable instances are used to train and evaluate the models, this is obtained with *forced sat* instance generation, where an arbitrary assignment is first randomly sampled and, subsequently, all generated clauses that are not satisfied by this sampled assignment are rejected, thus guaranteeing that the generated instance will have at least one satisfiable assignment.

In order to compare the obtained results with existing related work, the same dataset as that used by Selsam et al., 2018 to train their NeuroSAT model is also generated. This data generation model, denoted by $SR(n)$, creates instances of the SAT problem with n input variables and with clauses of a varying number of literals. The number of literals on each clause is determined by some k, randomly sampled with mean just over 4. Lastly, for evaluation purposes, benchmark SAT problems encoding the Graph Coloring Problem are obtained from the online library SATLIB (flat30-60 dataset)[1].

[1]SATLIB

5 Experiments and Results

5.1 Notes on Implementation

The Graph Networks are implemented using the open source Python package *Deep Graph Library* (DGL)[2] and *PyTorch*. The source code for this project can be found online on GitHub[3]. All experiments are executed on GPU devices from the Lisa System, a cluster computer operated by SURFsara[4].

The training data , both for the generated Random K-SAT and SR datasets, contain 14000 problem instances, which are observed by the models exactly once during training. The validation set contain 1000 problem instances, sampled in batches of 200 for each validation run. The evaluation set contain 500 problem instances. All problem instances are of satisfiable problems only. All experiments are run using 3 random seeds. There has been low variation on results observed between different random seed runs.

The hyper-parameters, and other specifications, used to train the models can be found in appendix A. Given time constraints, hyper-parameter tuning was not performed for this study. The default hyper-parameters values of the discount factor γ, ϵ, ϵ decay rate, target update, and learning rate are based on those used for ECO-DQN (Barrett et al., 2020). The embedding dimension and the optimizer used are based on those utilized for NeuroSAT (Selsam et al., 2018).

5.2 Design Experiments

There are many possible variants for the design of this method, including feature engineering, reward shaping and model architecture. To determine which model better fits the problem in question, different lines of experiments are executed, trained on a training set of problem instances, and their performances tested with a validation set (one random initialization given per problem instance). The following subsections describe each of these lines of experiments.

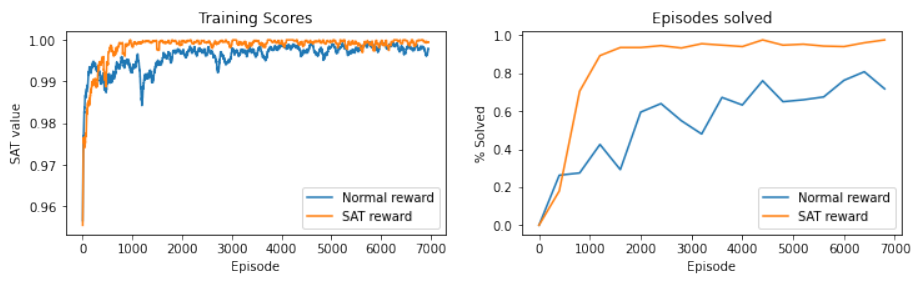
5.2.1 Lookahead behaviour

Lookahead consists of simulating all possible immediate future states s_{t+1}, to observe some features that may be helpful to the model for constructing better predictions. These lookahead features are denoted with an * in the description of features in section 4.2. The process of obtaining these features is requires many computations, which increases rapidly with the number of input variables.

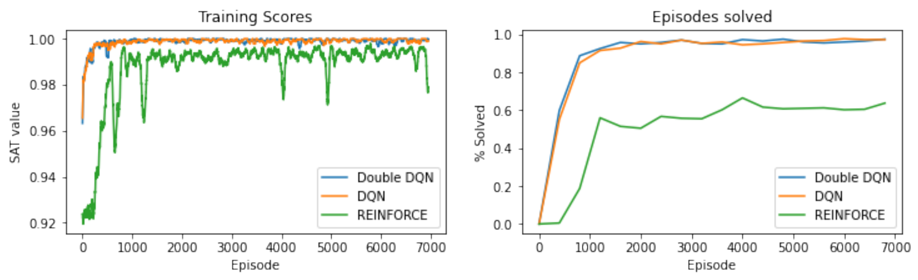
Models trained without lookahead behaviour solved 98.8% of the 3-SAT problem instances in the validation set, whereas models trained without these features solve 99.5% of the problem instances, a difference of less than 1%.

Although these features helped the models to achieve a slightly better performance, the difference is not sufficiently high to justify the cost of computing them. These features are excluded from the final models. This also means that the agent does not perceive an additional reward when it reaches a local maxima, which is the approach used by ECO-DQN (Barrett et al., 2020).

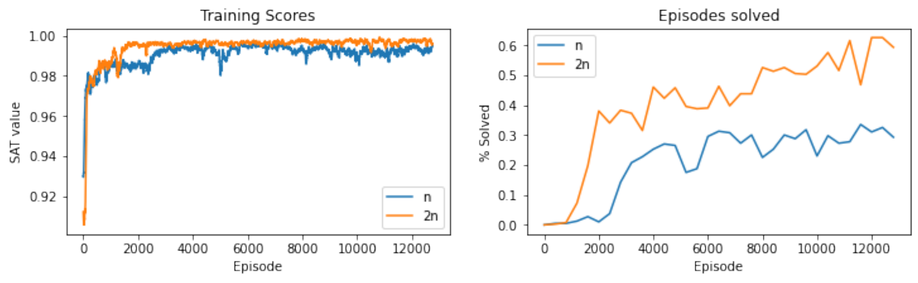
[2]dgl.ai
[3]Source Code repository
[4]Lisa System Description



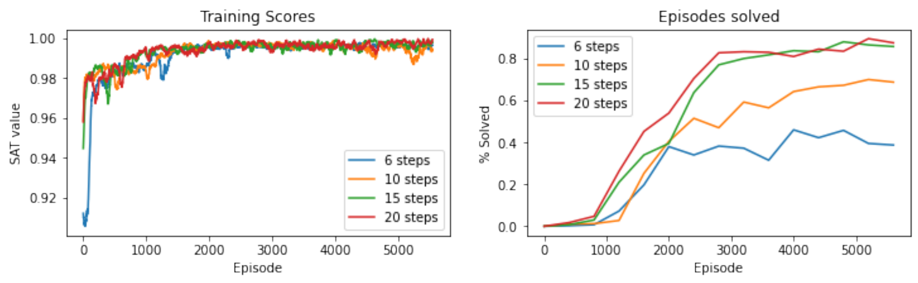
(a) Assigning an additional reward of 1 when an agent finds a solution for the problems increases performance substantially.



(b) DQN and Double DQN as training algorithms display similar behaviour, while they both outcompete REINFORCE.



(c) Models trained on episodes with a fixed maximum length of double the number of input variables, $2n$, have stronger performance.



(d) The performance of models increase with the number of steps in the message passing sequence, however, their time performance is impaired as this number grows.

Figure 2: Performance of models during train time on different experiment lines. Episode scores (left) on the training set. Percentage of problems solved (right) on the validation set (one random initialization per problem). Models in 2a and 2b are trained on the *random k-sat* dataset, those in 2c and 2d are trained on the *SR* dataset.

5.2.2 Stacked-layers vs recursive-layers

This experiment considers the effect of using a recursive architecture, versus one of stacked hidden layers. These two approaches are discussed in detail in section 4.2.

There was no significant difference observed between the two approaches, both solving roughly 99% of the validation problems. It is worth to consider, though, that in this experiment, the number of hidden layers in the stacked-layer models where equal to the sequence length of the recursive-layer models, which may explain the similarity in the results. The recursive-layer is opted for evaluation, given that it is more memory efficient and it allows to change the length of message passing sequence during test time.

5.2.3 Additional reward on solved problems

This experiment contemplates the effect of adding an additional reward of 1 to the normal reward defined on equation 17, when the agent finds a solution to the instance problem.

The models trained with this bonus reward solve 99.3% of the validation problems, versus 84.7% of problems solved without it. Figure 2a shows how the scores obtained during training are more stable when this reward is included, as well of giving a much stronger performance on the validation set.

Interestingly, this bonus reward has an important effect on the performance of the models, giving an increase of 17% of in performance. The reason for this effect may be given because when adding this additional reward, more information is provided to the model, a single bit of supervision (as expressed in the title of Selsam et al., 2018), which is only possible to assign on decision problems.

5.2.4 Type of reward

As discussed in section 4.1, the global SAT value on which the reward is to be shaped from, can be either derived from the variables nodes or the clause nodes. This experiment contemplates the effect of using either of these values as a basis for the reward function.

Models using the clause values as a basis for the reward solved 99.3% of the problem instances on the validation set, whereas those using the variable values solved 98.2%. The difference observed is small, less than 1%, which rather than showing that one approach is significantly better than the other, perhaps a better interpretation is to say that both of the approaches provide sufficient information to be an effective basis of the reward function. Nonetheless, the clause sat values are utilized on the final models for shaping the rewards, given the observed difference.

5.2.5 Learning Algorithms

Three types of RL algorithms are deployed for training the models, DQN (Mnih et al., 2013), Double DQN (Van Hasselt et al., 2016) and REINFORCE (Williams, 1992).

Both DQN and Double DQN displayed similar performances, but these, however, outperform REINFORCE by a significant margin. Figure 2b shows the training scores and validation results obtained during train time. The Double DQN approach is maintained for evaluation experiments.

5.2.6 Max length of episodes

As it is suggested by Barrett et al., 2020, the maximum length of an episode is set to double the number of input variables, $2n$. For speedup purposes, this max-length number is decreased to be equal to the number of variables, n, and the effect of this change evaluated.

It was observed observed that the suggested $2n$ max-length leads to stronger models (see figure 2c). This difference becomes even more apparent with models trained with the SR(40) dataset, which is harder than random 3-SAT, where the models trained with n steps solved up to 34% of the validation set problems and those with $2n$ steps solved 63% of them. The episode length of $2n$ is maintained for evaluation purposes.

5.2.7 Number of message passing steps

It was shown by Selsam et al., 2018 that there is a strong dependence on the number of iteration of message passing for a graph network to be capable of finding solutions for the SAT problem. In this method however, it is not possible to let the GNN iterate indefinitely, this has to be set to a fix number on each episode step in order for the Q-values to be computed. Long message passing sequences makes the network slower, while short sequences pose the risk of not generating meaningful embeddings to produce accurate Q-values. In this experiment, the effect of this trade-off is analysed. Models are trained on sequences, L, ranging between 6 and 20.

It was observed that models trained with a larger number of steps on the message passing sequence lead to better performances. Figure 2d display the learning curves of models trained on SR(40), where the effects observed were greater. A sequence length of $L = 15$ is used for evaluation on the SR models, given that the results were similar to those of models trained with $L = 20$. For random k-sat models, however, the difference observed was not great enough, so the minimum sequence length of $L = 6$ is maintained for evaluation.

5.3 Performance

During evaluation time, a model attempts to solve each problem instance with up to 50 random initializations. If a solution is found, the model moves to the next problem instance, without having to complete the 50 episodes quota. The evaluation results can be found on table 1.

The models trained on the random 3-SAT dataset can solve all problem instances from the evaluation set that have the same number of input variables as those on which they have been trained, roughly on their first attempt. Further, these models generalize to larger problems, of up to 200 variables, with fairly reliable performance. The model trained on the SR(40) dataset can also solve all problem instances of its own group, and generalises up to problems with 80 and 100 variables, solving 93% and 27% of them respectively. Problems generated with the SR model are generally harder to solve than those from random 3-SAT and the models where not able to generalise to problems with 200 variables on this harder dataset.

Model trained on 3-SAT with $n = 40$		
	% Solved	Avg. initializations
40 variables, 160 clauses	100%	1.0
80 variables, 320 clauses	100%	1.1
100 variables, 400 clauses	99.7%	1.6
200 variables, 800 clauses	15%	21.0
Model trained on 3-SAT with $n = 80$		
	% Solved	Avg. initializations
40 variables, 160 clauses	100%	1.1
80 variables, 320 clauses	100%	1.0
100 variables, 400 clauses	100%	1.0
200 variables, 800 clauses	77%	10.2
Model trained on $SR(n = 40)$		
	% Solved	Avg. initializations
40 variables, ∼230 clauses	100%	1.0
80 variables, ∼440 clauses	93%	8.4
100 variables, ∼520 clauses	27%	24.1

Table 1: Evaluation results. Third column indicates the average number of random initializations taken by the model before finding a solution. If no solution is found after 50 random initializations, the problem instance is considered unsolved.

The authors of the NeuroSAT method (Selsam et al., 2018) report solving 70% of the satisfiable problems during test time (tested after 26 iteration of message passing on the SR(40) dataset), but it was also pointed out by them that this percentage increases with longer sequences of message passing. Eventually, NeuroSAT approaches 100% solve rate when the number of iterations approaches 10^3. For the method presented in this thesis, on a model trained on SR(40) with fixed sequence length of $L = 15$ and maximum number of episode steps $2n$, the maximum number of message passing operations during one episode is of 1200. However, it is usually the case that a solution is found within the first half of the episode. Given that the model generally solves problems from SR(40) on the first episode (see table 1), the running times for the two methods seem not to be far apart. It is difficult, however, to make a one-to-one comparison between the two methods, given the different nature of the underlaying algorithms they are based upon.

Additionally, the model (trained on 3-SAT, $n = 80$) is also able to generalize to problems from different distributions. This is tested on a dataset of 100

SAT problem instances encoding the Graph Colouring Problem (3-colourable, 30 vertices, 60 edges, all satisfiable). The model is able to find solutions on 97% of these problem instances, however, there is no other GNN based SAT solver contemplating this dataset to compare results with.

6 Conclusions and Discussion

In this thesis, we developed a SAT solver for satisfiable instances based on exploratory combinatorial optimization, which was first introduced by Barrett et al., 2020, and closely related to the GNN based SAT solver NeuroSAT introduced by Selsam et al., 2018. This novel method gives the SAT problem, if not decision problems in general, a more solid standing in the ground of combinatorial optimization. Previous GNN methods considering the SAT problem were trained as classifiers primarily, and satisfiable assignments were obtained by some inference methods, such as clustering, from the internal activations of the graph (Selsam et al., 2018). This method, on the other hand, learns to solve satisfiable problems directly. Furthermore, most CO that use RL a learning method tend to focus on NP-Hard problems, such as the max cut problem and traveling salesman problem. These problems are fundamentally different to the satisfiable problem, given that the latter belongs in a more constrained class of problems, NP, and as such, presents different characteristics than the former.

One of these such different characteristics, observed during the course of this research, is that the models trained with this method are able to effectively learn the SAT problem, even if only a reward of 1 is given to the agent when it finds a solution, and 0 on all other states. This alternative reward may provide the benefit of not encouraging the agent to walk into sub-optimal local maximas of the gradient space. However, this approach suffers from the same defect as that of the Mountain Car environment of OpenAI gym[5]. That is, the agent can only start learning once it stumbles, by chance, on a solution. For this matter, models using this method were only able to learn problems with 10 input variables, given that the chances of randomly finding solutions diminishes with larger problems. It would be interesting to see if this problem can be resolved by training on a dataset with different numbers of input variables (large and small), or to have the model weights pre-trained with the current method.

One of the core distinction between the method presented in this thesis and NeuroSAT is that the latter can repeat the message passing iteration up to any point desired, without re-initializing the feature embeddings throughout. Here may lay the reason why NeuroSAT is capable of generalizing to even larger problems than those it has been trained on, up to 200 variables on the SR dataset, as compared to the method from this project, which can only generalize to problems of up to 100 variables on that same dataset. Figure 2d well illustrates that there is a strong dependency on the length of the message passing operation for attaining good performance. The authors of the NeuroSAT algorithm point out that there is a phase transition, on which the network passes from a state of low confidence to one of high confidence that the input problem is satisfiable, after a certain number of message passing iterations. This seems to indicate that there is some depth required for the network to produce enough information in order to attain satisfiability features. The methodology used in this thesis was based on the work of Barrett et al., 2020, ECO-DQN, on which the feature embeddings are re-initialized after taking each action, setting the message passing iteration to a fixed length of L. It can be argued, though, that this approach is carelessly throwing away meaningful information, contained in the feature embeddings computed by the network, after taking each action. A possible way

[5]MountainCar Env

to circumvent this problem is to only process one message passing iteration on each time-step of the episodes and not re-initialize the feature embeddings after an action is taken. This, however, poses serious challenges to the DQN algorithm, given that at the optimization step, these feature embeddings cannot be re-computed, unless the entire sequence of the episode actions are stored in the memory replay. This problem is not presented in the REINFORCE algorithm, where each episode is optimized exactly once right after its termination. It was observed that models trained with REINFORCE using this alternative interpretation were able to learn the task. In fact, these models, displayed even better performance compared to the REINFORCE models discussed in section 5.2.5. Unfortunately, given time constraints imposed by the thesis deadline, this point was not further expanded upon. Further research on this topic is an object of interest.

Given that one of the main limitations of this method is the performance in terms of time, it would be interesting to see if this method could be interpreted as a multi-agent system (Lauer and Riedmiller, 2000), on which multiple actions may be taken on each time-step of the episode. This approach may be possible, given that there exists a distributed value, *variable-sat-value* (see section 4.1), that may be used as independent rewards on a cooperation game. On this multi-agent system, each variable is its own distinct agent, with the choice of actions of flipping its own value or to pass.

Lastly, another consideration to be had is that this method is not an end-to-end SAT solver, given it only considers satisfiable problems. It remains to be seen if a graph network could learn to construct proofs for unsatisfiable problems.

References

Craik, K. (1943). The nature of exploration cambridge university press. *Cited in online encyclopaedia of human-computer interaction.*

Cook, S. A. (1971). The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, 151–158.

Hopfield, J. J., & Tank, D. W. (1985).
neural" computation of decisions in optimization problems. *Biological cybernetics*, *52*(3), 141–152.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, *14*(2), 179–211.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3), 229–256.

Goemans, M. X., & Williamson, D. P. (1995). Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)*, *42*(6), 1115–1145.

Zhang, W., & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. *IJCAI*, *95*, 1114–1120.

Hochba, D. S. (1997). Approximation algorithms for np-hard problems. *ACM Sigact News*, *28*(2), 40–52.

Lauer, M., & Riedmiller, M. (2000). An algorithm for distributed reinforcement learning in cooperative multi-agent systems. *In Proceedings of the Seventeenth International Conference on Machine Learning.*

Boufkhad, Y., Dubois, O., Interian, Y., & Selman, B. (2005). Regular random k-sat: Properties of balanced formulas. *Journal of Automated Reasoning*, *35*(1), 181–200.

Braunstein, A., Mézard, M., & Zecchina, R. (2005). Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, *27*(2), 201–226.

Audemard, G., Jabbour, S., & Sais, L. (2008). Sat graph-based representation: A new perspective. *Journal of Algorithms*, *63*(1-3), 17–33.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2008). The graph neural network model. *IEEE transactions on neural networks*, *20*(1), 61–80.

Biere, A., Heule, M., van Maaren, H., & Walsh, T. (2009). Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, 131–153.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*, 1097–1105.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602.*

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529–533.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., & Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. *CoRR*, *abs/1611.09940.* http://arxiv.org/abs/1611.09940

Dai, H., Dai, B., & Song, L. (2016). Discriminative embeddings of latent variable models for structured data. *International conference on machine learning*, 2702–2711.

Liang, J. H., Ganesh, V., Poupart, P., & Czarnecki, K. (2016). Learning rate based branching heuristic for sat solvers. *International Conference on Theory and Applications of Satisfiability Testing*, 123–140.

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI conference on artificial intelligence*, *30*(1).

Bünz, B., & Lamm, M. (2017). Graph neural networks and boolean satisfiability. *arXiv preprint arXiv:1702.03592*.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., & Song, L. (2017). Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. *International conference on machine learning*, 1263–1272.

Vinyals, O., Fortunato, M., & Jaitly, N. (2017). Pointer networks.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gulcehre, C., Song, F., Ballard, A., Gilmer, J., Dahl, G., Vaswani, A., Allen, K., Nash, C., Langston, V., . . . Pascanu, R. (2018). Relational inductive biases, deep learning, and graph networks.

Kool, W., Van Hoof, H., & Welling, M. (2018). Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*.

Li, Z., Chen, Q., & Koltun, V. (2018). Combinatorial optimization with graph convolutional networks and guided tree search.

Liang, J. H., Oh, C., Mathew, M., Thomas, C., Li, C., & Ganesh, V. (2018). Machine learning-based restart policy for cdcl sat solvers. In O. Beyersdorff & C. M. Wintersteiger (Eds.), *Theory and applications of satisfiability testing – sat 2018* (pp. 94–110). Springer International Publishing.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., & Dill, D. L. (2018). Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Abe, K., Xu, Z., Sato, I., & Sugiyama, M. (2020). Solving np-hard problems on graphs with extended alphago zero.

Barrett, T., Clements, W., Foerster, J., & Lvovsky, A. (2020). Exploratory combinatorial optimization with reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *34*(04), 3243–3250.

Bengio, Y., Lodi, A., & Prouvost, A. (2020). Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*.

A Appendix

Random K-SAT hyper-parameters	
α (clauses to variables ratio)	4
K (variables per clause)	3
GNN hyper-parameters	
Embedding Dimension (d)	128
Activation Function	LeakyReLU(0.1)
Training hyper-parameters	
Learning rate	1E-04
γ (discount factor)	0.95
Total number of episodes	14000
Optimizer	Adam
Device	CUDA
Random seeds	1,2,3
DQN hyper-parameters	
Batch size	32
Replay memory size	5000
Target network update (in episodes)	10
Epsilon decay (% of episodes)	10%
Epsilon after decay	0.05

Table 2: Hyper-Parameters used.