Leveraging Reinforcement Learning on the Boolean Satisfiability Problem

Angelo Rajendram AA3RAJEN@UWATERLOO.CA
Faculty of Math
University of Waterloo

Abstract

This paper surveys the applications of reinforcement learning to enhance and automate learning heuristics for SAT Solvers. The mapping between various components of SAT solving and MDPs are presented and the connection between reinforcement learning and the SAT problem explored. Threads for future work and open problems relating to SAT solvers are indentified.

Keywords: Reinforcement Learning, Boolean SAT Problem, SAT Solver

1. Introduction

The Boolean satisfiability problem (SAT) is a fundamental problem in mathematics and computer science that seeks to determine if there exists an assignment that satisfies a given Boolean formula. Since it was proved to be NP-complete by Stephen Cook, the SAT problem has been extensively studied and has been shown to have a wide range of applications in fields such as constraint solving, software verification, and AI planning.

The success of modern SAT solvers can be attributed to the development and refinement of heuristics over several decades, which have allowed these solvers to efficiently solve problem instances with millions of variables. While these heuristics have been effective, there has been a lack of a satisfactory theoretical basis to explain their success.

Recent advances in SAT solvers have shown a trend towards using reinforcement learning (RL) frameworks to automate the learning of efficient heuristics. This survey aims to explore the connection between RL and SAT solvers by examining the application of RL techniques to the SAT problem. In particular, we will look at the use of RL in key heuristics such as branching, restarts, and clause deletion, as well as the parallelization of SAT solving. Finally, we will discuss open problems and future directions for research at the intersection of RL and SAT.

2. Preliminaries

2.1 The Boolean Satisfiability (SAT) Problem and SAT Solvers

Problem Statement 2.1 The Boolean Satisfiability Problem *Let X be a set of Boolean variables $\{x_1, x_2, .., x_n\}$. Given a Boolean formula $\phi(X)$, determine if there exists an assignment $I : X \to \{true, false\}$ to the variables such that the formula ϕ is satisfied.*

It is typically convenient to express Boolean formulas $\phi(X)$ in Conjunctive Normal Form (CNF) i.e. a conjunction (ANDs) of clauses consisting of a disjunction (ORs) of literals. A literal l is a variable $x \in X$ or its negation $\neg x$. An assignment I maps each variable to a Boolean value, and a clause is said to be satisfied by I if at least one of its literals

evaluates *true* under the assignment. Solving the SAT problem for a given instance ϕ amounts to finding a satisfying assignment I (SAT), or otherwise generating a proof that no satisfying assignments exist (UNSAT). The process of finding I is often abstracted as searching through a binary tree.

SAT Solvers generally fall into two categories: complete and incomplete solvers. The former can determine if a given instance is SAT or UNSAT whereas the latter is limited to only determining SAT. Incomplete solvers are typically based on stochastic local search (SLS) where a random initial candidate solution is iteratively updated by flipping a single variable's assignment at every step. State-of-the-art complete solvers predominantly use Davis-Putnam-Logemann-Loveland (DPLL) algorithm and its extension, conflict-driven clause learning (CDCL). Katebi et al. (2011) offer an empirical study of the various components of solvers and their contribution to its overall performance.

2.1.1 SAT Solver Heuristics

Key heuristics and processes used by SAT solvers that may be used as entry points for the application of RL techniques are presented below.

Branching Heuristic The branching heuristic, also called the decision heuristic, is responsible for variable selection and governs how the solver moves forward in the search space. Among the most prevalent is Variable State Independent Decaying Sum (VSIDS), introduced by Moskewicz et al. (2001). VSIDS is a counter-based heuristic that keeps track of a variable's *activities*, i.e. how often it is involved in conflicts.

Conflict-Driven Clause Learning (CDCL) CDCL is an enhancement over DPLL and consists of the following steps: When a conflict is encountered it is analyzed. A learnt clause is then derived and attached to the original set of clauses. The solver then backtracks to the decision level where the conflict is resolved. If VSIDS is used, variables that participate in generating learnt clauses will have their activities bumped.

Restarts Gomes et al. (2000) found that the runtime distributions of solvers show high variability for any particular problem instance and are characterized by heavy tails. Restarts are a mechanism to mitigate this by permitting solvers to escape from regions in the search space that have no solutions. Once a SAT solver encounter enough conflicts, it restarts at the root node of the search tree. Progress is preserved since learnt clauses are retained.

Clause Deletion CDCL SAT solvers accumulate large sets of learnt clauses over the solver run and database reduction schemes are necessary to keep the clause database of practical size. These schemes typically use some measure of clause "quality", a function of size, age, activity, and literal block distance (LBD) [Audemard and Simon (2009)]. The LBD of a clause is the number of subsets of literals formed when the literals in the clause are partioned according to the decision level at which they were assigned. Intuitively, clauses with low LBD are more useful since they prune larger segments of the search tree.

2.1.2 Parallel SAT Solving

The increasing ubiquity of multi-core based parallel processing capabilities over the past few decades has prompted research into the parallelization of SAT solvers. However, the inherent

sequential nature of the SAT problem has made progress challenging. The key dilemma is the diversification/intensification trade-off where a balance must be struck to ensure an efficient division of labour between the parallel solvers. Popular modern methods focus on competitive parallelism where solvers on parallel cores with different restart, decision, and learning heuristics compete to solve the problem, while continuing to share learnt clauses over a cooperation network. This is called the *parallel portfolio* architecture.

Sharing clauses uniformly among a set of cores does not scale up with the number of cores n since having n or $n-1$ frequent communications over the search process slows down the performance of the entire system. This issue can be mitigated by controlling the amount information dynamically shared by the cores and is amenable to RL techniques [Hölldobler et al. (2011)].

2.2 Reinforcement Learning

2.2.1 Markov Decision Process (MDP) Formulation of RL

The RL problem is formulated based on a Markov decision process (MDP). An MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$ with the components described as as follows in the context of decisions for the SAT problem:

- $\mathcal{S}$ is the set of states. Each state $s(\phi, I)$ is characterized by the current sub-instance and partial assignment to the variables (full assignment in case of SLS), or otherwise by features extracted from them. Episodic RL terminates when an agent takes a predetermined number of steps or reaches terminal states denoted by $\mathcal{S}^+$.
- $\mathcal{A}(s)$ is the set of actions. For branching in state $s(\phi, I)$, an action $a \in \mathcal{A}$ corresponds to selecting an unassigned variable to branch on (flip a variable in SLS).
- $\mathcal{T} : \mathcal{S} \times \mathcal{A}(s) \to \mathcal{S}$ is the transition function which maps a state-action pair (s, a) to the next state s'. In general contexts apart from SAT, $\mathcal{T}$ may be stochastic.
- $\mathcal{R}(s, a, s')$ is the reward function that provides a feedback signal r_t to inform an agent of its performance at each time step t . Several reward schemes are possible for SAT, and they may or may not be stochastic in general contexts.
- $\gamma \in [0, 1)$ is a discount factor used to incentivize immediate or near term actions.

Given this formulation of an MDP, learning good heuristics corresponds to an RL agent improving its policy $\pi : \mathcal{S} \to \mathcal{A}$, a mapping from states to action, until an optimal policy is found. The expected return or *value* of a given state is the expected cumulative rewards achieved under policy π starting from that state. The optimal policy thus corresponds to maximizing the total expected return $R = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$ for infinite horizon trajectories. Sutton and Barto (2018) offers a comprehensive overview of established RL algorithms for obtaining such policies. Two main categories of RL include value based methods, such as Q-Learning, and policy based methods such the REINFORCE algorithm.

2.2.2 Multi-Armed Bandit (MAB)

Multi-armed bandits are a special case of reinforcement learning problem where there is only a single state $\mathcal{S} = s_0$, a finite set of actions $\mathcal{A}$, and a non-deterministic reward function $\mathcal{R}$. The multi-armed bandit problem highlights the exploitation/exploration dilemma. Common algorithms used to address the MAB problem include *upper confidence bound* (UCB), *Thompson sampling*, and ϵ-*greedy* algorithms.

If the probability distributions of rewards underlying the actions do not change over time, the problem is said to be *stationary*, otherwise it is *non-stationary*. Optimal actions in stationary problems can be determined by considering the mean of accumulated rewards for each action a. As the number of executions of a approaches infinity the mean converges to the expected reward for a. For non-stationary distributions on the other hand, convergence is undesirable and the estimate must track the underlying distribution. A common approach is to use *exponential recency weighted average* (ERWA) given in its incremental form by Equation 1.

$$\overline{r}^a_{n+1} = (1-\alpha)\overline{r}^a_n + \alpha r^a_{n+1} \tag{1}$$

Here r^a_i is the i-th reward attained upon executing action a and $\alpha \in (0,1)$ is the step-size that determines the rate of decay. As the search landscape of a SAT problem instance is explored, the distribution of rewards evolves and necessitates non-stationary approaches.

2.3 Representing SAT instances with Graph Neural Networks (GNN)

A Boolean formula ϕ can have an arbitrary size and clauses can be added or eliminated at different stages of the solver run. Furthermore, ϕ is invariant to permutations of the clauses, variables, and their renaming. These properties are respected by graph neural networks (GNN) which possess an inductive bias that preserves the relationships between variables and clauses. GNNs can thus be used in RL algorithms to learn Q-functions or stochastic policies over a given problem instance and its sub-instances to inform branching decisions. Figure 1 displays a bipartite representation of a simple boolean formula with solid and dashed lines indicating the polarity of a variable with respect to the clauses. x_n indicate variables and c_n indicate clauses.

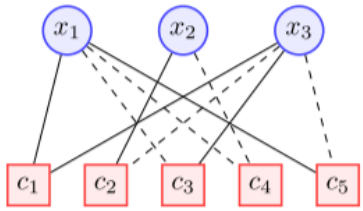

Figure 1: Graph for a CNF formula.[Yolcu and Poczos (2019)]

In essence, GNNs operate on directed graphs $\langle V, E, U\rangle$, where V is the set of vertices, E the set of directed edges, and U is a global attribute containing information globally relevant to the graph. Vertices, edges, and the global attribute are called *entities* and each entity has an associated annotation. The GNN updates the annotations of each entity based on their current annotation and an aggregation function that propagates information from adjacent entities. Battaglia et al. (2018) explores these properties and offers a formalism that unifies most existing GNN approaches.

3. Reinforcement Learning and SAT Solver Heuristics

3.1 RL for Decision Heuristics

3.1.1 Meta-Decision Heuristics

Meta-decisions at decision nodes Lagoudakis and Littman (2001) demonstrate a value based RL method that chooses from a set of branching algorithms with the objective of minimizing the overall execution time. The branching algorithms considered were a set

of seven well-known early heuristics, none of which were best over all instances of SAT problems. This process can be formulated as a variant of an MDP.

The states $s \in \mathcal{S}$ consist of a description of the current instance or sub-instance ϕ_t in terms of some features. Lagoudakis and Littman (2001) use the number of variables n in the instance (sub-instance) ϕ_t. The two terminating states $\mathcal{S}^+$are contradiction nodes, whereupon a conflict is encountered, and satisfaction nodes, where the current partial assignment I_p satisfies the formula ϕ_t.

The actions $a \in \mathcal{A}$ correspond to selecting one available branching heuristic when the solver arrives at a decision node in the search tree. This enables fine-grained decision making resulting in better branching decisions, albeit with higher costs at each decision point.

The reward function $\mathcal{R}$ assigns immediate costs associated with each decision. The cost r for each decision is the number of resultant decision nodes produced as a consequence of the current decision.

The meta-decision heuristic learns a parametrized Q-function $Q(s, a)$ to estimate the total node cost at every state s. The agent then acts in accordance with a greedy policy π with respect to the learnt Q-values. This approach results in a solver capable of making better decisions that translate to reducing the number of search nodes explored, however, this is only a surrogate for the wall-clock time. The recursive calls employed by this heuristic incurs higher overheads than classical methods.

Meta-decisions at restarts A more recent approach by Cherif et al. (2021) employs a meta-decision heuristic that selects a branching heuristic whenever a solver restarts rather than at each decision node during search. Cherif et al. (2021) recognize that meta-decisions at each branching node are costly and may cause interference between competing heuristics. Selecting between branching heuristics at restarts incurs less overhead and avoids interference. The approach presented adheres to the MAB framework and can be formulated as follows.

The actions $a \in \mathcal{A}$ correspond to selecting one among the available branching heuristics when the solver restarts and the trials are the solver runs between restarts.

The reward function $\mathcal{R}$ is given by Equation 2, where t denotes the current run, $decisions_t$ denotes the number of branching decisions taken, and $decidedVars_t$ denotes the explored depth of the search tree.

$$\mathcal{R}(a) = \frac{\log_2(decisions_t)}{decidedVars_t} \tag{2}$$

Variants of the UCB algorithm (UCB1 and minimax optimal strategy in the stochastic case (MOSS)) are then used to maximize rewards over a sequence of trials. Augmenting modern solvers with this approach improves solving time and yields highly competitive state-of-the art solvers.

3.1.2 Directly Integrating RL into the Branching Heuristic

Conflict History Based Branching (CHB) Liang et al. (2016a) attempts to map learning branching heuristics to a MAB framework rather than choosing from handcrafted branching heuristics. This approach does not quite conform to the MAB framework since

rewards are known a priori and therefore there is no metric being optimized. This pseudo-MAB problem is formulated as follows.

The actions $a \in \mathcal{A}$ correspond to selecting a variable $x \in X_u$, where X_u is the set of currently unassigned variables at a decision node. The agent selects these actions greedily with respect to a score $Q(x)$.

The reward function $\mathcal{R}$ is given by Equation 3, and generates an immediate reward whenever a variable is branched on, propagated, or asserted. Variables that were involved in conflicts are incentivized further by setting *multiplier* 1.0. The score $Q(x)$ is initialized to zero at the start of the search and is updated incrementally with rewards using an exponential recency weighted average (ERWA).

$$\mathcal{R}(x) = \frac{multiplier}{numConflicts - lastConflict[x] + 1}, \quad multiplier \in \{1.0, 0.9\} \tag{3}$$

CHB is based on the general principle that branching on variables which lead to more conflicts allows the solver to prune the search tree quickly by accumulating learnt clauses. Furthermore, CHB is capable of learning online during the search process by using feedback from conflict analysis. Empirical evaluation on SAT competition benchmarks from 2013 and 2014 show CHB solving significantly more instances that VSIDS, the dominant branching heuristic at the time.

Learning Rate Based Branching (LRB) Liang et al. (2016b) develops CHB further by redefining the rewards to provide an interface between the MAB framework and the branching heuristic. The formulation is as follows.

The actions $a \in \mathcal{A}$ correspond to selecting a variable $x \in X_u$, where X_u is the set of currently unassigned variables at a decision node. The LRB agent selects these actions actions greedily with respect to the ERWA of the rewards $Q(x)$.

The reward function $\mathcal{R}$ generates a reward r consisting of the *learning rate* augmented with the *reason side rate*. This reward is computed whenever a variable x transitions from assigned to unassigned. Finally, *locality* is a decay factor applied to $Q(x)$ after every conflict to discourage the algorithm from exploring inactive variables and instead allow it to exploiting the community structure of the CNF instance.

The introduction of a learning rate as a metric of a variable's propensity to generate learnt clauses allows branching to be framed as an optimization problem. LRB thus adheres to the MAB framework and attempts to maximize the learning rate, improving upon both VSIDS and CHB with respect to instances solved versus wall-clock time.

3.1.3 Policy-Gradient Methods with GNNs for Stochastic Local Search

Yolcu and Poczos (2019) present an approach to learn solver heuristics from scratch using a policy-gradient based deep reinforcement learning method. Although the end result is an incomplete solver that is not competitive with the state-of-the-art, it showcases an approach towards automated algorithm design that distinguishes it from the previous approaches.

SAT solvers that use SLS start with an initial assignment I to the set of Boolean variables X. This assignment is iteratively refined by visiting neighbouring assignments until a satisfying assignment is found. The neighbouring assignments differ by the assignment of a single variable. This process may be formulated as an MDP as follows.

A state $s \in \mathcal{S}$ consists of the Boolean CNF formula and an assignment to is variables (ϕ, I). Episodes terminate if a satisfying assignment is found or after a predetermined number of steps are taken. At the start of each episode ϕ is sampled from a particular class of SAT problems D, and the variables X receive a uniformly random assignment I.

An action $a \in \mathcal{A}$ corresponds to selecting the variable $x \in X$ to be flipped given the state s of the solver. A GNN is used as a policy network $\pi_\theta(\phi, I)$ to learn a distribution over X and the action is sampled from this policy $a \sim \pi_\theta(\phi, I)$. Furthermore, with a probability of $1/2$ the agent randomly selects a variable from a randomly selected unsatisfied clause to encourage exploration.

The reward function $\mathcal{R} : \mathcal{S} \to \{0, 1\}$ is defined as $\mathcal{R}(s) = \phi(I)$ i.e. 1 if a satisfying assignment (at which point the episode terminates) is found and 0 otherwise. Finally, a discount factor $\gamma \in [0, 1)$ is used to incentivize finding satisfying assignments sooner.

The REINFORCE algorithm is employed to iteratively improve the policy network π_θ over a particular class of SAT instances, and specialized heuristics are learnt for each problem class. Learning is accomplished with little prior knowledge and with a simple rewards scheme differentiating it from approaches that require reward shaping and feature engineering. Furthermore, the learnt heuristic requires fewer although costlier steps to find a solution relative to generic heuristics.

3.1.4 Q-Learning with GNNs to learn Branching Heuristics

Also using a GNN, Kurin et al. (2020) use a value based deep reinforcement learning scheme instead to learning a branching heuristic for a backtracking solver. The result is a complete solver called Graph-Q-SAT. While still not competitive with state-of-the-art SAT solvers, Graph-Q-SAT exhibits high data efficiency and good generalization properties.

The states $s \in \mathcal{S}$ are tuples (X_u, C_u) where X_u and C_u are the set of currently unassigned variables $x \in X_u$ and unsatisfied clauses $c \in C_u$ respectively. The MDP is episodic and terminates when a satisfying assignment is found (SAT), or the search space has been exhausted (UNSAT), or if predetermined number of steps has been reached.

The actions $a \in \mathcal{A}$ select an unassigned variable x and assigns it to $true$ or $false$. Graph-Q-SAT uses a GNN to learn Q-function values over both $true$ and $false$ assignments to variables in X_u and follows a greedy policy π with respect to the learnt Q-function values.

The reward function $\mathcal{R} : \mathcal{S} \to \{p, 0\}$ provides a negative reward p for every step the agent takes, and a reward of 0 in the terminal state. This simple reward scheme encourages the agent to complete episodes as quickly as possible without elaborate reward shaping.

The branching heuristic learnt using Graph-Q-SAT makes decisions that result in more propagations relative to VSIDS and so prunes the search tree quicker. This improvement is attributed to the fact that early on in the solution procedure VSIDS makes poor uninformed decisions. Graph-Q-SAT considers the problem structure in order to make better decisions early on. Despite reducing the number of iterations by a factor 2-3X, frequent calls to the learnt heuristic incurs a large overhead and is not yet competitive based on wall-clock time.

3.2 RL for Clause Deletion Heuristics

Clause deletion heuristics are integral components of CDCL solvers that prevent slowdown during Boolean constraint propagation (BCP). Vaezipoor et al. (2020) use a policy gradient

algorithm to optimize a stochastic policy for an RL agent performing garbage collection, formulated as an MDP.

An observation $s \in \mathcal{S}$ consists of state features that include the ratio of learnt to original clauses, a histogram of LBD scores of recently learned clauses along with their average, and moving averages of both the recent trail size and recent decision levels. The episode terminates when the solver finds a satisfying or unsatisfying assignment, or if the solver run is aborted upon reaching some number of iterations.

An action $a \in \mathcal{A}$ is a decision whether to keep or drop each clause out of N learnt clauses. The policy of the agent outputs an LBD threshold, and all clauses with an LBD higher than this threshold are subsequently deleted.

The reward function $\mathcal{R}$ uses the number of logical operations *op* performed by the solver as a more stable and deterministic surrogate of the wall-clock solving time. Episodes continue until the instance is solved or while $op < 10^9$. The reward of an aborted episode is 0, and that of a successful episode is $200 - op \times 10^{-7}$ received at the end of the episode.

Vaezipoor et al. (2020) show that their approach can be trained to improve its reward on the Cellular Automata benchmark of SATCOMP-2018. However, further development is needed to improve upon the state-of-the-art.

3.3 RL for Parallel SAT Solving

Hölldobler et al. (2011) uses the MAB framework to dynamically control clause sharing between parallel cores with CDCL solvers to overcome the scalability problem with more cores. Their approach, called bandit ensemble for parallel SAT solving (BESS), treat the different cores in parallel portfolio architecture as individual MAB agents.

Controlling clause sharing in a cooperation network of interconnected cores can be viewed as deciding which *emitter* cores are permitted to send clauses to any given *receiver* core. BESS controls the cooperation network by attaching a MAB agent to each receiver core. The MAB problem can be formulated as follows.

The actions $a \in \mathcal{A}$ constitute setting a reward threshold used to decide which emitter cores are permitted to transmit clauses during a given time period. Cores permitted to transmit to the receiver are *alive*, otherwise they are *sleeping*. If any current emitter core produce average cumulative rewards lower than a it is sent to sleep and the emitter asleep for the longest duration is awakened. The reward threshold is updated at every K conflicts encountered by the solver on the receiver core.

The reward function $\mathcal{R}$ is designed to be some metric of the quality of clauses transmitted by each emitter core. These metrics include clause size based rewards, LBD based rewards, variable/literal activity based rewards, and combinations of these. As the search yields non-stationary rewards, a fixed-size windowed simple moving average is employed.

BESS leverages the UCB algorithm for each MAB agent implemented on top of the ManySAT parallel SAT solver, state-of-the-art at the time. As the number of cores in the communication topology were increased so did the comparative performance margin in favour of BESS over the standard implementation of ManySAT.

4. Future Horizons and Conclusion

Leveraging RL for decision making by solvers has proven fruitful with the development of performant branching heuristics like CHB and LRB. The integration of RL into SAT solver permits abstracting processes in the solver as optimization problems. Furthermore, a hierarchy of decision-making can be established by leveraging meta-decision heuristics. The key pitfall to avoid is higher computational cost associated with more complex decision making although this can be mitigated by choosing suitable entry points for each decision process, such as restarts.

The approaches surveyed appear to fall into two clusters; implementations with little overhead albeit with significant reward shaping reliant on domain knowledge (CHB and LRB) and those that leverage ML to learn heuristics with little to no reward shaping or feature engineering (GNN based methods). While the latter are far from competitive, they nevertheless serve to showcase approaches that automate algorithm design and may lead to theoretical advances with the SAT problem. These algorithms are also likely to improve with the incorporation of sophisticated RL stabilization and temporal credit assignment techniques. Algorithms such as CHB and LRB are methods based on MAB and developing informative state feature representations are natural extensions of this work and may lead to the development of techniques that fit in the gap between these two clusters.

The use of RL for other key solver heuristics such as restarts and clause deletion has received less attention than for branching, but may nevertheless result in incremental gains in the competitive landscape of modern SAT Solvers.

Dynamical control of the communication topology over parallel solvers using approaches such as BESS have been essential towards implementing solvers on multi-core architectures that are now the norm, which has historically been a tricky problem due to the inherently sequential nature of the SAT problem. As BESS uses a simple MAB formulation that sorts and filters emitter cores, future work may consider incorporating more spohisticated MAB/RL or other fine-grained methods capable of controlling which clauses are transmitted over the cooperative network.

To conclude, it is worth mentioning a connection between machine learning and solvers identified by Ganesh et al. that is reminiscent of reinforcement learning:

"Solvers implement proof systems via a combination of symbolic reasoning rule-based methods and ML heuristics aimed at optimally sequencing, selecting and proof rules in order to construct optimal proofs for a given input formula. A particularly effective way to do this is to view a solver's sequencing and selection heuristics as reinforcement learning agents that sequence/select rules based on the rewards they obtain by interacting with a deductive environment."

References

G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. pages 399–404, 07 2009.

P. Battaglia, J. B. C. Hamrick, V. Bapst, A. Sanchez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. Allen, C. Nash, V. J. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv*, 2018. URL `https://arxiv.org/pdf/1806.01261.pdf`.

M. S. Cherif, D. Habet, and C. Terrioux. Combining vsids and chb using restarts in sat. In *CP*, 2021.

V. Ganesh, S. Seshia, and S. Jha. Machine learning and logic: A new frontier in artificial intelligence. unpublished.

C. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. 01 2000.

S. Hölldobler, N. Manthey, V.-H. Nguyen, J. Stecklina, and P. Steinke. A short overview on modern parallel sat-solvers. 01 2011.

H. Katebi, K. Sakallah, and J. Silva. Empirical study of the anatomy of modern sat solvers. pages 343–356, 06 2011. ISBN 978-3-642-21580-3. doi: 10.1007/978-3-642-21581-0_27.

V. Kurin, S. Godil, S. Whiteson, and B. Catanzaro. Can q-learning with graph networks learn a generalizable branching heuristic for a sat solver? In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the dpll procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001. ISSN 1571-0653. doi: https://doi.org/10.1016/S1571-0653(04)00332-4. URL `https://www.sciencedirect.com/science/article/pii/S1571065304003324`. LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001).

J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Exponential recency weighted average branching heuristic for sat solvers. In *AAAI Conference on Artificial Intelligence*, 2016a.

J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for sat solvers. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 123–140, Cham, 2016b. Springer International Publishing. ISBN 978-3-319-40970-2.

M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001. doi: 10.1145/378239.379017.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

P. Vaezipoor, G. Lederman, Y. Wu, R. B. Grosse, and F. Bacchus. Learning clause deletion heuristics with reinforcement learning. 2020.

E. Yolcu and B. Poczos. Learning local search heuristics for boolean satisfiability. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/12e59a33dea1bf0630f46edfe13d6ea2-Paper.pdf.