FINDING THE SMALLEST TREE IN THE FOREST: MONTE CARLO FOREST SEARCH FOR UNSAT SOLVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Monte Carlo Tree Search (MCTS) is an effective approach for finding low-cost paths through any large combinatorial space that can naturally be structured as a search tree. However, some combinatorial problems do not have a natural interpretation as searches for a good path. For example, solving a CSP can be represented as a path (assign variables sequentially and check the solution); however, proving that no solution exists (via existing methods) requires enumerating multiple paths to build out a "proof tree" demonstrating that every possible variable assignment leads to a conflict. Rather than finding a good path (solution) within a tree, the search problem becomes searching for a small proof tree within a forest of candidate trees. In this paper we develop Monte Carlo Forest Search (MCFS), an algorithm for finding small search trees. Our method leverages the benefits of the best MCTS approaches and further introduces two key ideas. First, we estimate tree size via the linear (i.e., path-based) and unbiased approximation from Knuth (1975). Second, we query a strong solver at a user-defined depth rather than learning a policy across the whole tree, in order to (1) reduce the variance of our tree-size estimates and (2) focus our policy search on early decisions, which offer the greatest potential for reducing tree size. We evaluated our approach on the Boolean satisfiability (SAT) problem, and found that it matched or improved performance over a strong baseline on two well-known distributions (`sgen`, `random`). Notably, we improved walltime by 9% on `sgen` over the `kcnfs` solver and even further over the strongest UNSAT solver from the 2021 SAT competition.

1 INTRODUCTION

Silver et al. (2017) took the world by storm when their AlphaGo system beat world champion Lee Sodol at Go, marking the first time a computer program had achieved superhuman performance on a game with such a large action space. Their key breakthrough was combining Monte Carlo Tree Search (MCTS) rollouts with a neural network-based policy to find increasingly strong paths through the game tree. This breakthrough demonstrated that, with good state-dependent policies, MCTS can asymmetrically explore a game tree to focus on high-reward regions despite massive state spaces.

The MCTS algorithm used in AlphaGo (hereafter simply referred to as MCTS) combines three powerful ideas: (1) Monte Carlo approximation of the value of a state by leveraging rollouts, thereby avoiding the exponential cost of enumerating all subsequent sequences of actions (Kearns et al., 2002; Coulom, 2006); (2) efficiently trading off exploration and exploitation in these rollouts by leveraging multi-arm bandit policies (Kocsis & Szepesvári, 2006); and (3) function approximation of the policies and values from previous problems to provide priors that further focus rollouts on promising paths (Coulom, 2007; Sutskever & Nair, 2008; Maddison et al., 2014).

MCTS is most useful in combinatorial spaces that have a natural hierarchical decomposition into a search tree. As a result, most of the original applications of MCTS were to search for good actions in game trees (Coulom, 2006; Baudiš & loup Gailly, 2012; Zook et al., 2019; Schrittwieser et al., 2020; Agostinelli et al., 2019). MCTS has also been used for searching for solutions to NP-hard optimization problems (e.g. Browne et al., 2012; Abe et al., 2019; Khalil et al., 2022) and is a promising approach for finding strong paths through any large combinatorial space.

However, many combinatorial problems cannot be expressed as searching for a good path. Consider constraint satisfaction problems (CSPs): while solving a satisfiable problem corresponds to finding a

path (assigning variables sequentially and checking the solution), existing methods for proving that no solution exists require enumerating multiple paths to build out a "proof tree" that demonstrates that all possible variable assignments lead to a conflict. The order in which variables are assigned—the *branching policy*—has a dramatic effect on the size of the proof tree and the corresponding time to solve the problem, so algorithm designers seek branching policies that lead to small proof trees. But this task is no longer a standard search problem: rather than finding a high-reward path within a single tree, our goal is to find a small tree within the forest of possible trees defined by every permutation of variable orderings. This difference matters because of the cost of evaluating each candidate policy: because each tree may be exponentially large and there are an exponential number of possible trees, the full search space becomes doubly exponential in the number of actions.

We present a solution to this problem that we call Monte Carlo Forest Search (MCFS), an algorithm for finding small search trees while retaining the benefits of MCTS outlined above. The MCFS algorithm uses *Knuth samples* (Knuth, 1975) to obtain cheap and unbiased Monte Carlo approximations of tree sizes. A Knuth sample is drawn as follows: given an n-ary tree, we construct a path from the root to a leaf where an action is chosen at every node by rolling an n-sided unbiased die. If we denote the realized path length as X, then an unbiased estimate of the true tree size is n^X. For any variable ordering, we can leverage Knuth samples to obtain a linear-time stochastic approximation of the size of the proof tree that verifies all true and false assignments. We use MCTS to search over variable orderings while using Knuth samples to estimate the size of the resulting trees.

Of course, Knuth samples provide a linear-time approximation of an exponential function and thus will have very high variance, necessitating variance reduction techniques to reduce the sample complexity of MCFS. The most obvious such technique is to take many Knuth samples rather than only one, which we do. We also employ a further, more complex variance-reduction technique. When we arrive at a node at a user-defined threshold depth, we query a known strong subsolver, rather than continuing search all the way to a conflict. This limits sources of variance to the stochastic decisions up to that node, and it focuses the search on early decisions where the neural network has the highest potential to reduce tree size relative to inference cost.

We evaluated our approach on the Boolean satisfiability (SAT) problem, one of the most widely studied combinatorial problems. Leveraging the MiniSAT (Eén & Sörensson, 2003) framework, we integrated MCFS into the DPLL algorithm and learned a branching heuristic to efficiently find small proof trees of unsatisfiability. Proofs of UNSAT are important in practice, e.g., for system debugging (Suelflow et al., 2008) and formal verification (Bryant et al., 2009). As a result, the main track of the annual SAT competition typically has around 50% UNSAT instances and has often featured further tracks focused only on UNSAT instances (Berre et al., 2007).

We matched or improved over the performance of a strong baseline on two prominent SAT Competition distributions. First, we evaluated our method on uniform random 3-SAT at the solubility phase transition, perhaps the best-studied SAT distribution which has been featured in the SAT Competition from 2002 to 2018. We were able to match performance of `kcnfs07`, which was specifically designed to target this distribution. Second, we evaluated our method on the `sgen` distribution, which is notoriously difficult for its problem size and produced the smallest unsolved instance at the 2009 SAT competition. We improved running time on `sgen` by 9% over `kcnfs07`, which is 3.2× faster than the `hKis` solver, which solved the most UNSAT instances in the 2021 SAT competition.

In what follows, we discuss related work in Section 2. We introduce the SAT problem, the DPLL algorithm, and MCTS in Section 3 and describe our MCFS approach in Section 4. We present our experimental setup and results in Sections 5 and 6 respectively. Finally, we discuss future work in Section 7.

2 Related Work

Tree search is a fundamental algorithm for combinatorial optimization. It works by iteratively partitioning a search space, e.g., by choosing variables to branch on. At a high level, the efficiency of tree search is measured by the product of (1) the number of branches and (2) the time required to make each branching decision. In domains such as MIP solving, the "smart and expensive" paradigm has won out, where entire linear programs are solved to determine a branching decision. In other domains such as SAT solving, the "simple and cheap" paradigm dominates: high-performance SAT

solvers make extremely cheap branching decisions that do not condition on subproblem state. After 20 years, VSIDS (Moskewicz et al., 2001) has remained among the state of the art for QBF and SAT; it is a simple heuristic that only keeps track of how often a variable occurs in conflict analysis. Some interesting early work developed more expensive heuristics (Huang & Darwiche, 2003), but these generally could not compete with VSIDS in terms of running time. One specialized example in SAT where a more expensive structural heuristic is beneficial is the solver `kcnfs` (Dequen & Dubois, 2003) for random 3-SAT. It branches based on a proxy measure for how likely a variable is to be in the backbone, which is the set of literals that must be true in every model.

One might expect that machine learning could learn a more informative heuristic that is worth its cost, especially for shallow-depth branches where decisions are most consequential. Two approaches exist for learning models to make branching decisions: imitation learning and reinforcement learning.

Imitation learning works by learning a cheap approximation of (1) an expensive existing heuristic or (2) an expensive feature that is a good proxy for a good branching decision. Gasse et al. (2019) and Nair et al. (2020) exploited the fact that neural network inference time is faster than MIP heuristics to learn cheaper approximations that achieve state-of-the-art performance. Relevant to our work on UNSAT solving, Selsam & Bjørner (2019) learned to approximate small UNSAT core computation and then branched on variables predicted to belong to a core. They computed small UNSAT cores for 150,000 instances and trained a neural network to predict them, achieving a 6% speedup over a strong SAT solver when using their network for a limited number of top-level branching decisions. Note that if there exists a small UNSAT core, there exists a corresponding short proof, but there can exist a short proof if there doesn't exist a small UNSAT core.

Reinforcement learning directly searches for policies that can produce such short proofs instead of seeking to imitate an existing heuristic. Seminal early work on SAT by Lagoudakis & Littman (2001) used TD-learning to train a policy to select between seven predefined branch heuristics based on simple hand-crafted features; Yolcu & Póczos (2019) learned a local search heuristic; and Tönshoff et al. (2022) built a generic graph neural network-based method for iteratively changing assignments in CSPs and demonstrated good results on 100 variable SAT problems. For QBF, Lederman et al. (2019) learned an alternative to VSIDS using the REINFORCE algorithm that roughly minimizes decisions. They improved over VSIDS in CPU time on hard problems from a distribution within the framework of CADET, an open-source competitive solver. For model counting, Vaezipoor et al. (2021) used an evolutionary strategy to learn a state-of-the-art branching heuristic, demonstrating improvements in walltime performance over the competitive `SharpSAT` solver on instances with more than 10,000 variables. Finally, Kurin et al. (2019)'s work is closest to our own. They used reinforcement learning for branching in the CDCL solver Minisat. Training on random-SAT problems with 50 variables and generalizing online to 250 variables for both SAT and UNSAT instances, they were able to improve running time over a generic solver. However, they had difficultly scaling due to high variance returns and credit assignment, two issues that our MCFS approach addresses. They resolved the path / tree distinction by treating a traversal through a tree as a path and allowing backtracking state transitions, but they only trained on SAT problems where the optimal SAT policy was a path rather than a traversal.

MCTS has several existing applications to solving NP-hard problems. Browne et al. (2012) used the UCT algorithm for solving MIPs, taking paths from the root to a leaf and propagating up the maximum over child LP values; Abe et al. (2019) searched over assignment paths in graph problems such as choosing edges to cut in a graph; and Khalil et al. (2022) searched over paths of variables to find MIP backdoors. The work of Scavuzzo et al. (2022) is the closest to our own in distinguishing between searching for paths and trees. They introduced "TreeMDPs" in which each state can transition to many states, which are a natural representation for UNSAT proof trees since after branching on a variable, we must explore both the paths for the true and false literal of that variable.

We approximate the tree size of a DPLL policy via Knuth's estimate (Knuth, 1975). There are other approaches for approximating tree size (e.g., Kilby et al., 2006), but they are not based on finding paths, and we saw no clear-cut way to integrate them into MCFS.

3 PRELIMINARIES

This section includes the required technical background on both Boolean saisfiability problems as well as Monte Carlo Tree Search.

3.1 BOOLEAN SATISFIABILITY

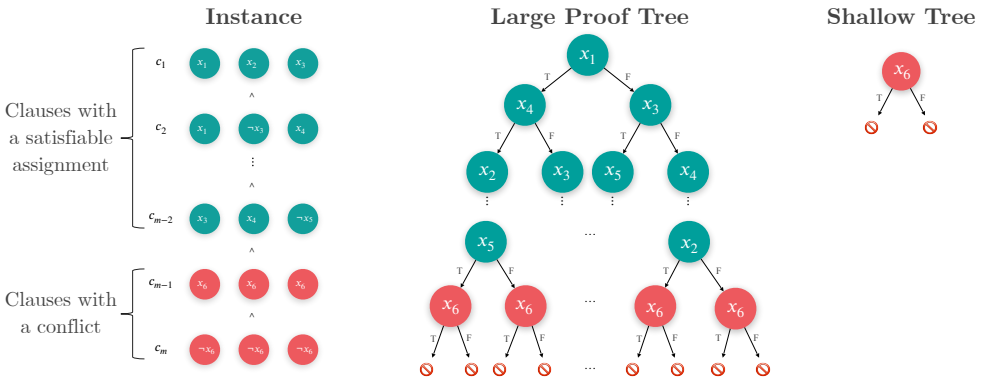

Figure 1: Example illustrating the importance of branching policies to the computational cost of solving SAT instances. a) The problem contains a large number of satisfiable clauses and a small unsatisfiable "core", which do not share any variables. b) Branching first on variables from the satisfiable clauses leads to a large search tree, where the same conflict must be identified many times. c) Branching on the single variable x_6 from the unsatisfiable clauses leads to a small search tree where the contradiction is immediately found.

Boolean satisfiability (SAT) problems are defined as follows. A SAT instance S is defined by a set of clauses $C = \{c_1, \ldots, c_m\}$ over a set of variables $X = \{x_1, \ldots, x_n\}$. Each clause consists of a set of Boolean *literals*, defined as either a variable x_i or its negation $\neg x_i$. Each clause is evaluated as *True* iff at least one of its literals is true (i.e., the literals in a clause are joined by OR operators). For example, a clause $c_i = x_j \vee \neg x_k$ evaluates to *True* iff either x_j is set to *True* or x_k is set to *False*. S is *True* if there exists an assignment of values to variables for which all the clauses simultaneously evaluate to *True* (i.e., the clauses are joined by AND operators). If such an assignment exists, the instance is called *satisfiable*; it is called *unsatisfiable* otherwise.

SAT solvers try to find a way to set the variables to demonstrate that the problem is satisfiable (SAT), or to construct a proof tree that shows that no setting of the literals can satisfy the formula (UNSAT).

In this paper, we consider only UNSAT problems, for which such proof trees must be found. Many SAT solvers rely on the Davis-Putnam-Logemann-Loveland algorithm (DPLL), which assigns variables in an order given by some (potentially state-dependent) variable selection policy.

Definition 1. *Let S be a SAT instance. A policy ϕ is a mapping $\phi : S \to (v, \mathbb{1})$ that determines which variable v to assign in DPLL and what value $\mathbb{1} \in \{0, 1\}$ to assign it to first.*[1]

Given a policy, the DPLL algorithm is straightforward: it iterates over the selected variables recursively, checking both the *True* and *False* assignments, and performing "unit propagation" at each step (setting variable values forced by single-variable ("unit") clauses; propagating these values to other clauses in which the same variables appear (possibly in negated form); repeating until no unit clauses remain) until either a conflict or satisfying assignment is found. The full pseudocode of the DPLL algorithm is included in Appendix A.2.

There can be massive gaps in performance between different policies for choosing branching variables in DPLL. For example, Figure 1 shows a formula that leads to a three-node proof tree if x_6 is selected

[1]When looking for a SAT assignment one must occasionally backtrack and try a different assignment for a literal. This is handled by the DPLL algorithm itself rather than the policy.

first by the policy, but a tree that could have as many as $2^{|X|} - 1$ nodes if x_6 is selected last. Another way that policies can affect tree size is through DPLL's unit propagation step: policies that cause more unit propagation earlier in the search have fewer decisions to make overall and therefore yield smaller search trees. Overall, for an instance S that is solved using variable selection policy ϕ, we denote the size of the resulting proof tree as $T_\phi(S)$. For a given distribution over problems $\mathbb{P}$, our goal is to find a policy ϕ^* that minimizes the average proof tree size $\mathcal{L}(\phi; \mathbb{P}) = \mathbf{E}_{S' \sim \mathbb{P}}[T_\phi(S')]$. Finding such policies is computationally challenging; for a problem with n variables, there are $O(2^n n!)$ possible variable selection policies, and exactly evaluating $T_\phi(S)$ entails $O(2^n)$ operations.

If we are prepared to assume that the optimal variable selection policy, $\phi^*_{\mathbb{P}} = \arg\min_{\phi'} \mathcal{L}(\phi'; \mathbb{P})$ is learnable given an appropriate model family, we could in principle learn an approximation to the optimal policy, $\hat{\phi}^*$, to use within our solver. The challenge is designing a procedure to efficiently minimize $\mathcal{L}(\phi; \mathbb{P})$ so that we can collect labeled training examples. This is a challenging reinforcement learning problem. As mentioned above, given any particular variable selection policy, even evaluating the loss function on a single instance requires time exponential in the problem size, and the space of variable selection policies is massive.

3.2 MCTS

MCTS is an algorithm that has been successfully used to find good policies in large search spaces. Modern instantiations of MCTS, like the one used by AlphaGo, follow four key steps.

Action Selection At every state s, MCTS chooses an action $a \in \arg\min_{a'}(Q(s, a') - U(s, a'))$. $Q(s, a)$ is the cost estimate of a at state s and $U(s, a)$ is the corresponding confidence interval. These confidence bounds are computed via the PUCT algorithm (Rosin, 2011; Silver et al., 2016):

$$U(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_{a'} N(s, a')}}{1 + N(s, a)} \tag{1}$$

where $N(s, a)$ is the number of lookaheads that branch on action a at state s and c_{puct} is a constant that controls exploration. $P(s, a)$ represents a prior distribution over the actions for a given state. This prior comes from the key innovation of AlphaZero, combining MCTS with a neural-network-based policy. This policy network is trained to output a prior $P(s, a)$ when given the state s as input. It is trained to match the count of how many times each action was taken once MCTS terminates. This creates a loop where MCTS finds strong paths through the tree to supervise the neural network, and the neural network guides MCTS in future iterations to discover even stronger paths.

Expansion When MCTS reaches a state that has actions which have never been played, it can choose to take one of these actions and create a new child node. This causes the tree to progressively widen in areas that are sampled frequently.

Simulation The MCTS traversal is cut short when it reaches one of these newly-constructed nodes. However, rewards are only stored at the terminal nodes of the tree. To deal with this issue, MCTS uses a value network, which is trained to predict the value of a path originating from this state.

Backpropagation The reward (predicted by the value network or achieved at the leaf) is passed back through the traversal path. The Q-values and visit counts of each node are updated accordingly.

After a fixed number of iterations, it commits to the action at the root having the largest number of samples, and the resulting state becomes the new root node for MCTS. This outer loop repeats until it has found a path to a leaf of the tree. Finally each node along this path is used as a training example for the policy network and the reward of the path is used to train the value network. The pseudocode for the inner loop of MCTS is given in Appendix A.2.

AlphaZero's neural network architecture has two important feature that make it well suited for its task. Firstly, they use convolutional neural networks which work well for games like Go which are translationally invariant and rotationally symmetric. Secondly, they jointly train the policy network and value network. This allows them to share a single representation and allows the value network to leverage the more powerful training signal the policy network provides.

AlphaZero stores nodes in a tree structure where a node is reached by a single sequence of decisions. There may be many nodes with identical state if the nodes are reached with different decision order.

4 METHODS

We introduce Monte Carlo Forest Search, a method for adapting MCTS to the setting where policies produce trees rather than paths, which is precisely the problem of searching for a DPLL variable selection policy for solving UNSAT. The pseudocodes for both the inner and outer loop of MCFS are given in Appendix A.2 and the differences with MCTS highlighted in red. We now describe how MCFS turns trees into paths and how MCFS alters expansion, selection and value modeling of MCTS.

4.1 TURNING TREES INTO PATHS

To evaluate a given policy, ϕ', we need to know the associated tree size, $T_{\phi'}(S)$. The simulation step of MCTS would require visiting $O(2^n)$ states before reaching a terminal node. This is computationally expensive, both because the absolute number of states may be prohibitively large, and because at each node in the tree we need to query ϕ' to decide which variable is assigned next, which is non-trivial when ϕ' is parameterized by a neural network. We address this by stochastically approximating $T_{\phi'}(S)$ using Knuth samples (Knuth, 1975).

Theorem 1 (due to Knuth (1975))**.** *Let ℓ_P be the length of a path P sampled uniformly at random from a binary tree T with size s_T. Then, $s_T = \mathbf{E}_P\left[2^{\ell_P}\right]$.*

In the context of a DPLL solver with variable selection policy ϕ', this amounts to replacing a complete traversal of the binary tree of all *True* / *False* assignments with a set of paths through the tree where assignments are chosen uniformly at random. We can take the length of the resulting paths, ℓ, and give each node at depth d a cost of $2^{\ell-d}$ for its corresponding policy decision. The average of these costs across a set of paths approximates the proof tree size.

4.2 SIMULATION FROM A PRE-EXISTING POLICY

Rather than querying a value network at the first time a node is encountered and backpropagating, we callout to either (1) a pre-existing variable selection policy or (2) a value network that approximates the tree size of this fixed policy after a fixed number of decisions ℓ[2]. In practice, the policies we learn are implemented with deep neural networks, which are far more expensive to evaluate than standard heuristics. When applying this approach to large UNSAT problems, the sheer amount of policy network calls required to decide a policy for an entire problem will outweigh the time savings even if the decisions result in smaller policy trees. This is especially true for smaller subproblems that take a fraction of a second to solve with existing methods. To circumvent this bottleneck, we leverage existing SAT solvers at both training time and testing time. After ℓ decisions, the problem is small enough to be solved quickly and reliably using a pre-existing solver. This number of decisions is held constant across training and testing time so that the policy network can learn to leverage this downstream solver. A given node at depth d along the path updates its Q value with $2^{\ell-d}T_{\phi_{sub}}(S^*) + 2^{\ell-d} - 1$, where ℓ is the fixed number of decisions before the subsolver; S^* is the SAT instance that is given to the subsolver; and ϕ_{sub} is the policy of the subsolver being used. An added benefit of this approach is that the subsolver used within the model is easily replaceable, allowing us to use the best subsolver available for whichever distribution we are targeting.

4.2.1 VALUE NETWORK FOR PRE-EXISTING POLICY

During training, rather than explicitly solving the subproblem at depth ℓ, we only need to know the number of decisions that the subsolver would have had to make. We therefore train an additional neural network to predict how many decisions a subsolver would require for a given SAT instance, giving us access to a much cheaper (albeit potentially unreliable) reward signal during training. Unlike MCTS, this value network is estimating the cost of a fixed policy. This network is trained from an initial batch of subsolver calls and then retrained as we collect new, large batches of data, so its performance may vary. When the output of the network is too unreliable, it may not carry enough

[2]Another reason to use node expansion is that it can lower the variance of the rewards below a given state. However, we noticed empirically that the variance does not grow significantly with sample length within the range of number of decisions before we call a subsolver. Therefore, the variance reduction does not outweigh the benefit of removing expensive subsolver calls.

signal about the true number of decisions for good policies to be learned. To address this challenge, we add functionality to use a mix of calls to the subsolver and calls to our value network, proportional to an online estimate of the accuracy of the value network. We track the mean multiplicative error ϵ of our value network over time by querying the value network with every subsolver call. Given a user-defined accuracy threshold parameter t (we use $t = 0.5$), we sample the value network with probability $1 - \min(1, \epsilon/t)$, so the probability of calling the subsolver halves as the error halves.

4.3 Selection

We make three deviations from Equation 1. First, we add a Q_d coefficient to calibrate confidence intervals across different depths in the tree. We select variable $v = \arg\min_{v'}(Q(s, v') - Q_d U(s, v')$. In AlphaZero, rewards preserve scale with depth, where -1 represents a loss and 1 represents a win. Because costs in our setting scale exponentially with depth, we calibrate each depth with Q_d, the average tree size of nodes at depth d, which can be thought of as a per-depth constant. We compute Q_d online as the running average of tree sizes encountered from depth d. Second, we initialized Q values with the first lookahead, as this represents an unbiased measure of performance of our incumbent policy (i.e., the first sample is exactly the neural network policy), which we would like to improve upon. This is similar to AlphaZero's choice of initializing Q to 0, which is the baseline they aim to improve upon. Third, we only compute $P(S, a)$ once at each committed decision and passed down the network prediction to its child nodes. This saves substantial computation since our calls to the policy network tends to dominate the running time.

We also change our data structure for state transitions from a tree to a graph similar to Czech et al. (2020) so that we are invariant to the order of past decisions. This lets us share information across different paths to the same state which is common with variables with similar scores.

4.4 Policy and Value Modelling

Unlike Go, SAT problems vary in size and are invariant to permutations of its rows (clauses) and columns (variables within a clause). We capture the invariances of our problem by using an equivariant architecture where permutations of the input guarantee a corresponding permutation in the output and that can handle any size input matrix. We also add a third head to our network beyond the policy and value head to predict Q-values. The purpose of the Q-value head was as an auxiliary task to help train a better shared representation for the policy head; we observed a 5% reduction in tree size after adding the Q-value head.

5 Implementation and Experimental Details

We integrated our MCFS algorithm into the CDCL solver `Maple_LCM_Dist_ChronoBT` (Ryvchin & Nadel, 2018), which won the 2018 SAT competition and is based on the MiniSAT framework (Eén & Sörensson, 2003). We removed all clause-learning components so that the solver was running strictly DPLL search. We trained our neural networks with PyTorch (Paszke et al., 2019) in Python and ported over to our solver in C++ using tracing. We include our code in the supplementary material and will link to a public repository at publication time. Below, we describe implementation details and bechmarks. See Appendix B for a description of our model training.

Parameterization of MCFS Two important hyperparameters were (1) the constant for choosing the level of exploration c_{puct} and (2) the number of lookaheads k. Through a coarse grid search, we selected $c_{puct} = 0.5$ and $k = 100,000$, which found the best policies within an approximate 48-hour time window. After k lookaheads, we either committed to the action with the highest counts $v = \arg\max_{v'} N(S, v')$ or sampled the best action from our current neural network policy. We set the probability of these alternatives to 0.5 to balance on-policy and off-policy learning. Each MCFS decision yields a training point with a (state, policy vector, Q value) triple, where the policy vector consists of normalized counts from MCFS.

We chose $\ell = 6$ and $\ell = 8$ for the `random` and `sgen` distributions respectively, since that is where MCFS tended to find the strongest policies under reasonable time constraints. We made decisions with MCFS until depth ℓ, summing to 2^ℓ decisions for every run

Prior to a good policy network being learned, MCFS is less efficient. We pretrained our policy network and value network by running MCFS with 10,000 lookaheads on 1,000 instances for `random` and `sgen`. We ran one iteration of MCFS with 1,000 instances and 10,000 lookaheads to improve the policy and value network further and then a final iteration with 2,000 instances and 100,000 lookaheads, which we used to train our final model. Each pretraining run of MCFS took approximately 24 hours and were run on 300-variable and 55-variable problems from `random` and `sgen`. The two iterations using the prior took approximately 48 hours and was respectively run on 300-variable and 65-variable problems from `random` and `sgen`.[3]

Benchmarks and Baselines We targeted instance distributions that are well known and difficult for modern SAT solvers, with similar-sized action spaces to Go. We do not consider industrial SAT instances, as they often contain millions of variables with state spaces significantly larger than any application of MCTS we are aware of. We evaluated our approach on a random distribution (uniform random 3-SAT at the solubility phase transition, or `R3SAT`) and a crafted distribution (`sgen`).

For `R3SAT`, we followed Crawford & Auton (1996) to estimate the location of the phase transition with a number of clauses-to-number of variables (n/m) ratio of $n = 4.258 \cdot m + 58.26 \cdot m^{-2/3}$. We trained on 300 variable instances where calling a subsolver was quick ($\approx$ 1 second solving time), and filtered out SAT instances for training and testing. To evaluate upward-size generalization, we set aside 100 test instances at our training size of 300 variables as well as at 350, 400, and 450 variables. `R3SAT` is a well-studied distribution where there has been a focused algorithmic effort, making it an especially challenging benchmark. We compare to `kcnfs07` (Dequen & Dubois, 2003), which is specifically designed for this distribution and among the strongest solvers. We used the version submitted to the 2007 SAT competition (Berre et al., 2007), where it won the silver medal in the Random UNSAT track, the last time it was submitted to a competition. It previously won the gold medal at the 2005 Random UNSAT track submission.

`sgen` (Spence, 2010) is a hand-crafted generator that is notoriously difficult for its size. An `sgen` generated instance was the smallest to be unsolved at the 2009 SAT competition. The principle behind the generator is to ensure that many assignments must be made before reaching a conflict. At a high level, the variables are partitioned so that variables across partitions seldom appear in clauses together but contradictions occur across partitions so it is difficult to find a contradiction without assigning many variables. We set the `-unsat` option on the generator, which guarantees instances that contain contradictions. For similar reasons as `R3SAT`, we trained with 65 variables, and to evaluate upward-size generalization we set aside 100 test instances at our training size of 65 variables as well as at 75, 85, and 95 variables. Unlike `R3SAT`, there are no solvers known to perform well on this distribution. We likewise selected `kcnfs07` for our `sgen` benchmark since it was consistently faster than the best UNSAT solver from the most recent (2021) SAT competition, `hKis`, but we also included `hKis` in our evaluation.

We also evaluated a `uniform+kcnfs` baseline for each dataset, where we replaced neural network calls with uniform-random decisions and called the `kcnfs07` solver for subproblems at the same user-defined depth. We tried purely random decisions without calling out to the `kcnfs` subsolver on 65 variable sgen problems and average running time was a factor of 40 times slower than `kcnfs`. Given the poor performance, we did not run pure random beyond this dataset.

Computing Resources We ran our model training and solver benchmarking experiments on a small cluster of approximately 20 nodes that has 20 8 GB Tesla M60 GPUs. For MCFS runs, we also used a large shared cluster of CPU nodes. Each MCFS run was allocated 16 GB of memory and a maximum of 48 hours. Each solver benchmarking run was allocated 8 GB of memory.

6 RESULTS

We evaluated (1) search tree size and (2) running time for our approach against the two baselines on each of our eight test instance sets. We present the full results in Table 1. We measured running time as the cumulative CPU and GPU time with `runsolver` (Roussel, 2011).

[3]The parameters and architecture described in this paper are only for the last iteration of MCFS. We made several minor improvements across iterations, and we expect that the performance would have been at least as good had we used an identical setup for all iterations.

Distribution	#Vars	Tree size (1000s) [Reduction]			CPU+GPU time (s) [Reduction]		
		uniform+kcnfs	kcnfs07	NN (Ours)	uniform+kcnfs	kcnfs07	NN (Ours)
sgen	65	158.2	162.3	**132.2 [1.23x]**	7.2	**4.0**	27.4
	75	1,799.3	1,792.3	**1,594.0 [1.12x]**	46.3	**44.1**	63.6
	85	8,932.9	8,874.8	**8,156.2 [1.09x]**	**213.0**	219.9	217.3
	95	98,534.0	97,979.7	**92,407.9 [1.06x]**	2364.8	2454.7	**2243.7 [1.09x]**
random	300	44.3	8.9	**8.6 [1.02x]**	11.5	**1.9**	13.9
	350	215.7	43.3	**42.7 [1.01x]**	56.6	**10.5**	23.2
	400	1,103.5	226.9	**223.5 [1.02x]**	290.8	**53.2**	66.1
	450	5,207.9	**989.7**	994.9	1526.6	**291.8**	309.6

Table 1: Mean decisions (1000s) and running time (CPU+GPU s) over 100 instances of our approach, kcnfs07, and uniform+kcnfs. The reductions in decisions and running time are reported with respect to kcnfs.

On R3SAT, kcnfs07 is an extremely strong solver and there is a question whether there is anything better. It far surpassed random branching on the top-level decisions (4–5x reduction in tree size and walltime). Despite the strength of kcnfs, we were able to squeeze out 1-2% performance improvements in term of average tree size over kcnfs07 on up to 400 variables. With the overhead from our neural network calls, these reductions in tree size were not sufficient for improving running time.

On sgen, there is no known specialized solver and therefore much greater scope for improvement. kcnfs07 is not specialized for sgen; we observed that it typically found only marginally smaller trees than the uniform+kcnfs baseline, but it was the best existing solver that we found. The best UNSAT solver from the most recent SAT competition (hKis) was 3.2× slower than kcnfs07 (See Appendix A.3). We reduced average tree size over kcnfs07 on our training distribution (65 variables) by a factor of 1.23. Our model generalized well to larger problem sizes, reducing tree size even on 95-variable problems, which took approximately 40 minutes to solve and are 700x more difficult than our training distribution. Our solver incurred roughly constant-time overhead that prevented us from improving running time on the 65-variable and 75-variable datasets. The community cares much more about asymptotic behaviour and we were able to improve the running time over kcnfs by 1% on 85-variable problems and 9% (≈ 3 minutes faster) on 95-variable problems.

Our GPU overhead was especially large because of our outdated GPU hardware. We were able to access one high-performance GPU (Nvidia TITAN Xp) for an experiment on the 65 and 75 variable sgen datasets and the 300 and 350 variable R3SAT datasets. This hardware change closed the running-time gap between our method and kcnfs by on average 16.5 seconds on sgen and 6.1 seconds on R3SAT. We report those results in Appendix A.4.

7 Conclusions and Future Work

In this paper, we presented Monte Carlo Forest Search (MCFS), an algorithm for finding small trees while retaining the benefits of Monte Carlo Tree Search. Our method works when the tree size can be approximated with Knuth samples, a cheap and unbiased Monte Carlo approximation from a single path. We evaluated our approach on the well-known DPLL tree search algorithm for Boolean satisfiability, where the tree size of a policy can be approximated via Knuth samples. We matched or improved performance over a strong baseline on two well-known distributions.

In future work, we would like to generalize MCFS to CDCL solvers. Most high-performance industrial solvers use the CDCL algorithm, which adds a clause-learning component to DPLL to allow information sharing across the search tree. We saw no clear way of approximating the size of a CDCL search tree with Knuth samples as unlike DPLL, policy decisions in CDCL are made with knowledge of the history of past decisions. We would also like to focus on richer experimental results that go beyond this methodologically-focused paper. We would like to explore more distributions (especially those with weaker solvers), experiment with other subsolvers beyond kcnfs, and add more computation time for additional lookaheads per instance and deeper searches.

Reproducibility Statement In Section 5, we describe our datasets, baselines, experimental pipeline, and compute resources. We present the pseudocode of our algorithm in Appendix A.2 and the model training details in Appendix B. The supplementary material contains our source code and datasets.

REFERENCES

Kenshin Abe, Zijian Xu, Issei Sato, and Masashi Sugiyama. Solving NP-hard problems on graphs by reinforcement learning without domain knowledge. *arXiv preprint*, arXiv:1905.11623:1–24, 2019.

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

Petr Baudiš and Jean loup Gailly. PACHI: State of the art open source Go program. In *Proceedings of the 13th International Conference on Advances in Computer Games*, ACG '11, pp. 24–38, 2012.

Daniel Le Berre, Olivier Roussel, and Laurent Simon. SAT 2007 competition. The International SAT Competition Web Page, http://www.satcompetition.org/, 2007.

Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. An abstraction-based decision procedure for bit-vector arithmetic. *International Journal on Software Tools for Technology Transfer*, 11(2):95–104, 2009.

Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, CG '06, pp. 72–83, 2006.

Rémi Coulom. Computing "Elo ratings" of move patterns in the game of Go. *ICGA Journal*, 30(4): 198–208, 2007.

James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3SAT. *Artificial Intelligence*, 81:31–57, 1996.

Johannes Czech, Patrick Korus, and Kristian Kersting. Monte-Carlo graph search for AlphaZero. *arXiv preprint*, arXiv:2012.11045:1–11, 2020.

Gilles Dequen and Olivier Dubois. *kcnfs*: An efficient solver for random k-SAT formulae. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, SAT '06, pp. 486–501, 2003.

Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, SAT '03, pp. 502–518, 2003.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodia. Exact combinatorial optimization with graph convolutional neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, NeurIPS '19, pp. 15580–15592, 2019.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NeurIPS '17, pp. 1024–1034, 2017.

Jason S. Hartford, Devon R. Graham, Kevin Leyton-Brown, and Siamak Ravanbakhsh. Deep models of interactions across sets. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *ICML '18*, pp. 1914–1923, 2018.

Jinbo Huang and Adnan Darwiche. A structure-based variable ordering heuristic for SAT. In *Proceedings of the 18th International Joint Conference on Artificial intelligence*, IJCAI '03, pp. 1167–1172, 2003.

Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2):193–208, 2002.

Elias B. Khalil, Pashootan Vaezipoor, and Bistra Dilkina. Finding backdoors to integer programs: A Monte Carlo tree search framework. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence*, AAAI '22, pp. 1–10, 2022.

Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *Proceedings of the 21st National Conference of Artificial Intelligence*, AAAI '06, pp. 1014–1019, 2006.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, ICLR '14, pp. 1–15, 2014.

Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29 (129):122–136, 1975.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML '06, pp. 282–293, 2006.

Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan 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*, NeurIPS '20, pp. 9608–9621, 2019.

Michail G. Lagoudakis and Michael L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001.

Gil Lederman, Markus Rabe, Edward A. Lee, and Sanjit A. Seshia. Learning heuristics for quantified boolean formulas through reinforcement learning. In *Proceedings of the 8th International Conference on Learning Representations*, ICLR '19, pp. 1–18, 2019.

Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in Go using deep convolutional neural networks. *arXiv preprint*, arXiv:1412.6564:1–8, 2014.

Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pp. 530–535, 2001.

Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. Solving mixed integer programs using neural networks. *arXiv preprint*, arXiv:2012.13349:1–57, 2020.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, NeurIPS '19, pp. 8024–8035, 2019.

Christopher D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

Olivier Roussel. Controlling a solver execution with the *runsolver* tool. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, 2011.

Vadim Ryvchin and Alexander Nadel. Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking. In *Proceedings of SAT Competition 2018 — Solver and Benchmark Descriptions*, volume B-2018-1 of *Department of Computer Science Series of Publications B*, pp. 29, 2018.

Lara Scavuzzo, Feng Yang Chen, Didier Chételat, Maxime Gasse, Andrea Lodi, Neil Yorke-Smith, and Karen Aardal. Learning to branch with tree MDPs. *arXiv preprint*, arXiv:2205.11107:1–18, 2022.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588 (7839):604–609, 2020.

Daniel Selsam and Nikolaj Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing*, SAT '19, pp. 336–353, 2019.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, 2017.

Ivor Spence. sgen1: A generator of small but difficult satisfiability benchmarks. *ACM Journal of Experimental Algorithmics*, 15:1.1–1.15, 2010.

Andre Suelflow, Goerschwin Fey, Roderick Bloem, and Rolf Drechsler. Using unsatisfiable cores to debug multiple design errors. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, pp. 77–82, 2008.

Ilya Sutskever and Vinod Nair. Mimicking Go experts with convolutional neural networks. In *Proceedings of the 18th International Conference on Artificial Neural Networks*, ICANN '08, pp. 101–110, 2008.

Jan Tönshoff, Berke Kisin, Jakob Lindner, and Martin Grohe. One model, any CSP: Graph neural networks as fast global search heuristics for constraint satisfaction. *arXiv preprint*, arXiv:2208.10227: 1–23, 2022.

Pashootan Vaezipoor, Gil Lederman, Yuhuai Wu, Chris Maddison, Roger B. Grosse, Sanjit A. Seshia, and Fahiem Bacchus. Learning branching heuristics for propositional model counting. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence*, AAAI '21, pp. 12427–12435, 2021.

Emre Yolcu and Barnabás Póczos. Learning local search heuristics for boolean satisfiability. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, NeurIPS '19, pp. 7992–8003, 2019.

Alexander Zook, Brent Harrison, and Mark O. Riedl. Monte-Carlo tree search for simulation-based strategy analysis. *arXiv preprint*, arXiv:1908.01423:1–9, 2019.

A APPENDIX

A.1 DPLL

The DPLL algorithm makes calls to two subroutines.

Unit Propagation Unit propagation finds all unit clauses, or clauses which contain only a single unassigned literal. It then removes all other clauses which contain that literal; from all clauses that contain the literal's complement, it removes the literal's complement.

Pure Literal Assignment Pure literal assignment finds all literals such that their complements are not present in the SAT instance, which are known as pure literals. It then removes every clause that contains a pure literal.

Algorithm 1 DPLL

Input: SAT instance S, policy ϕ
$S \leftarrow \text{UnitPropagation}(S)$
$S \leftarrow \text{PureLiteralAssign}(S)$
if S is empty **then**
  return *True*
end if
if S contains an empty clause **then**
  return *False*
end if
$v \leftarrow \phi(S)$
return $\text{DPLL}(S_\wedge v = 0)$ OR $\text{DPLL}(S_\wedge v = 1)$

A.2 MONTE CARLO FOREST SEARCH PSEUDO-CODE

The red text represents where we diverge from MCTS.

Algorithm 2 MCFS: Inner Loop

Input: SAT instance S, subsolver policy ϕ_{sub}, policy depth ℓ
$S_{curr} \leftarrow S$, $D \leftarrow \{\}$
while $i < \ell$ **do**
  Choose variable $x_i \in S_{curr}$ with lowest LCB: $Q(S, v) - Q_d U(S, v)$
  Append (x_i, S_{curr}) to D
  Draw z from Bernoulli with $p = 0.5$
  $S_{curr} \leftarrow S_{curr} \wedge (x_i = z)$
  $S_{curr} \leftarrow \text{DPLL}_{\text{step}}(S_{curr})$
  $i \leftarrow i + 1$
end while
$\text{Cost} \leftarrow T_{\phi_{sub}}(S_{curr})$
$j = 0$
for (x, S) in D **do**
  Update LCB with cost $2^{\ell-j}\text{Cost} + 2^{\ell-j} - 1$
  $j \leftarrow j + 1$
end for

Algorithm 3 MCFS: Outer Loop

Input: SAT instance S, subsolver policy ϕ_{sub},
number of iterations k, policy depth ℓ

$S_{root} \leftarrow S$
Initialize Q values for every state to the value of single lookahead
while $i < \ell$ **do**
  Set $P(S', v) \leftarrow P(S_{root}, v)$ for all S' and v
  $j \leftarrow 0$
  while $j < k$ **do**
    MCFSInnerLoop$(S_{root}, \phi_{sub}, \ell)$
    $j \leftarrow j + 1$
  end while
  if Random draw with p=0.5 is True **then**
    Set v_{best} to the variable with the highest count after inner loop
  else
    $v_{best} \leftarrow argmax P(S_{root}, v)$
  end if
  Draw z from Bernouli with $p = 0.5$
  $S_{root} \leftarrow S_{root} \wedge v_{best} = z$
  $i \leftarrow i + 1$
end while

A.3 HKIS V.S. KCNFS07 ON SGEN

	Tree size (1000s)		CPU+GPU time (s)	
# Vars	hkis	kcnfs07	hkis	kcnfs07
65	422.1	162.3	12.2	4.0
75	2,996.7	1,792.3	127.7	44.1
85	111,435.8	97,979.7	699.1	219.9

We leave out 95 variables as hKis was not able to solve any problem within 6 hours.

A.4 FASTER GPU BENCHMARKING

		CPU+GPU time (s)		
Distribution	# Vars	uniform+kcnfs	kcnfs07	NN (Ours)
sgen	65	4.2	2.3	8.1
sgen	75	24.7	23.1	27.1
R3SAT	300	6.3	1.3	7.2
R3SAT	350	29.7	5.7	12.3

B MODEL TRAINING

We used the exchangeable architecture of Hartford et al. (2018). We represented a CNF SAT instance with n clauses and m variables as an $n \times m \times 128$ clause-variable permutation-equivariant tensor, where entry (i, j) is t_v if the true literal for variable i appears in clause j, f_v if the false literal for variable i appears in clause j, and 0 otherwise. t_v and f_v are random 128-dimensional vectors representing the true and false embeddings for a given literal. Following Hamilton et al. (2017), we also added a node degree feature to every literal embedding. We instantiated the permutation-equivariant portion of the exchangeable architecture as four exchangeable matrix layers with 512 output channels, with leaky RELU as the activation function. We mean-pooled the output to a vector, with each index representing a different variable. We experimented with attention pooling in the last exchangeable layer but didn't observe any improvements.

Given these shared exchangeable layers, we added three feed-forward heads: a policy head, a Q-value head, and a value head. The policy and Q-value heads both had two feed-forward layers with 512

channels and a final layer that mapped to the single output channel. The value head was the same, except there was a final mean pool that output a single scalar. The policy head was trained to predict the normalized counts of MCFS and the Q-value head was trained to predict the Q-values from MCFS. We used the cross-entropy loss for both. The purpose of the Q-value head was as an auxiliary task to help train a better shared representation for the policy head; we observed a 5% reduction in tree size after adding the Q-value head. We trained the value network with mean-squared error (MSE) against the $\log_2$ tree size of subsolver calls at leaf nodes. The value head was not backpropagated through the exchangeable layers. Using a shared representation was important for training the value head; MSE tended to be a factor of 2 worse when using a network trained with only a value head.

For training the exchangeable architecture, we used the Adam optimizer (Kingma & Ba, 2014), with a learning rate of 0.0001 and a batch size of 1 (the largest batch size that fit on our 8 GB GPUs). Online, we branched on the argmax variable from our neural network prediction. We used a held-out validation set to select a model that produced the minimum mean tree size when used within a DPLL solver. For a given depth d, MCFS makes 2^d decisions; therefore, we received exponentially more data points at deeper depths where decisions are less important. We experimented with exponentially upsampling nodes inversely proportional to their depth but observed no performance gains. For every leaf node, we recorded the true tree size to train our value network.