

UNSAT Solver Synthesis via Monte Carlo Forest Search

Chris Cameron¹, Jason Hartford², Taylor Lundy¹, Tuan Truong¹,
Alan Milligan¹, Rex Chen³, Kevin Leyton-Brown¹

¹ Department of Computer Science, University of British Columbia, Vancouver, BC
{cchris13, tlundy, kevinlb}@cs.ubc.ca, manhtuan15042000@gmail.com,
alanmil@student.ubc.ca

² Valence Labs, Montréal, QC jason@valencelabs.com

³ School of Computer Science, Carnegie Mellon University, Pittsburgh, PA rexcmu.edu

Abstract. We introduce Monte Carlo Forest Search (MCFS), a class of reinforcement learning (RL) algorithms for learning policies in tree MDPs, for which policy execution involves traversing an exponential-sized tree. Examples of such problems include proving unsatisfiability of a SAT formula; counting the number of solutions of a satisfiable SAT formula; and finding the optimal solution to a mixed-integer program. MCFS algorithms can be seen as extensions of Monte Carlo Tree Search (MCTS) to cases where, rather than finding a good path (solution) within a tree, the problem is to find a small tree within a forest of candidate trees. We instantiate and evaluate our ideas in an algorithm that we dub Knuth Synthesis, an MCFS algorithm that learns DPLL branching policies for solving the Boolean satisfiability (SAT) problem, with the objective of achieving good average-case performance on a given distribution of unsatisfiable problem instances. Knuth Synthesis is the first RL approach to avoid the prohibitive costs of policy evaluations in an exponentially-sized tree, leveraging two key ideas: first, we estimate tree size by randomly sampling paths and measuring their lengths, drawing on an unbiased approximation due to Knuth (1975); second, we query a strong solver at a user-defined depth rather than learning a policy across the whole tree, to focus our policy search on early decisions that offer the greatest potential for reducing tree size. We matched or exceeded the performance of a strong baseline on three well-known SAT distributions, facing problems that were two orders of magnitude more challenging than those addressed in previous RL studies.

1 Introduction

Silver et al. [66, 67] took the world by storm when their AlphaGo system beat world champion Lee Sedol 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. MCTS rollouts avoid the exponential cost of enumerating all subsequent sequences of actions [13, 33] and can be extremely efficient when leveraging (1) multi-arm bandit policies to trade

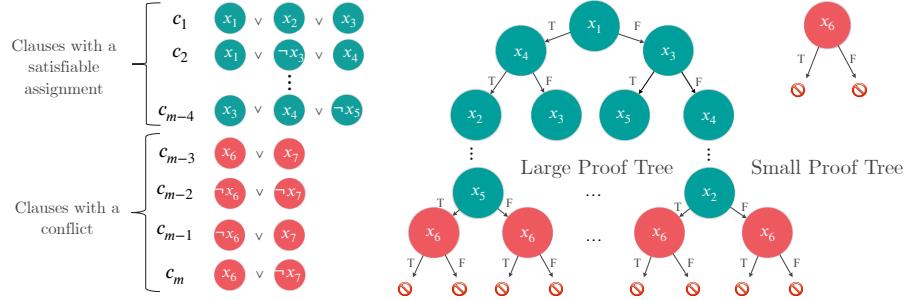


Fig. 1: A proof tree shows that any variable assignment to an unsatisfiable instance leads to a conflict. Its size substantially varies with the choice of branching policy.

off exploration and exploitation [40] and (2) function approximation of the policies and values from previous problems to provide priors that further focus rollouts on promising paths [14, 50, 71].

MCTS is most useful in combinatorial spaces that have a natural hierarchical decomposition into a search tree. As a result, most of the notable applications of MCTS were to search for good paths of actions through game trees [3, 5, 13, 63, 78]. MCTS is also useful for searching for solutions to NP-hard combinatorial problems where a solution is a path through a search tree [2, 8, 36, 56]. However, many other 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 build out a *proof tree* demonstrating that all possible variable assignments lead to a conflict. Rather than finding a high-reward path within a single tree, an algorithm designer’s goal is to find a small tree within the forest of possible trees (e.g., see Figure 1). This difference matters because of the high cost of evaluating each candidate policy: individual trees can be exponentially large.

This paper introduces Monte Carlo Forest Search (MCFS), a class of MCTS-inspired RL algorithms for such settings. We leverage Scavuzzo et al. [61]’s recent concept of *tree MDPs*. When an action is taken in a tree Markov Decision Process (MDP), the environment transitions to two or more new child states; each of these new child states satisfies the Markov property in that each child only depends on its state. Algorithms that recursively decompose problems into two or more simpler subproblems in a history-independent way can be represented as tree MDP policies. An MCFS rollout is a policy evaluation in the tree MDP and hence builds out a tree rather than a path, with the value of each node corresponding to the sum of rewards over its descendants in the rollout tree. Similar to MCTS, MCFS is a broad class of algorithms that can be adapted for different domains and problem sizes through choices in how policies are evaluated, how actions are selected and how rewards are aggregated. For example, an MCFS algorithm with complete policy evaluation (evaluating every child in a rollout) would be sufficient for tree MDPs giving rise to extremely small policy trees (e.g., log-linear trees in MergeSort). However, we are interested in tree MDP problems where policies produce exponential-

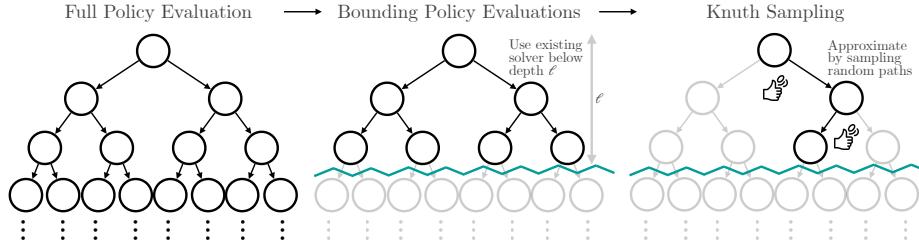


Fig. 2: Two keys ideas of Knuth Synthesis to avoid the prohibitive costs of exact tree-size policy evaluations: (1) bounded-depth search and (2) Knuth samples.

sized trees, making exact policy evaluation prohibitive. We are not aware of any method that addresses this issue; existing work using RL [21, 42, 45, 54, 61, 74] require exact policy evaluation and have only been used to train on very easy instances by industry standards (solvable on the order of 1000 decisions).

To solve this problem, we present Knuth Synthesis, an MCFS algorithm that allows for cheaper policy evaluation and is tailored for *pure-DPLL* [17] algorithms, a popular class of SAT algorithms that can straightforwardly be defined as tree MDPs.⁴ We use the idea of *Knuth sampling* [39] to obtain linear and unbiased Monte Carlo approximations of tree sizes, following Lobjois and Lemaître [48] who showed these can be effective for cheaply comparing algorithms despite their high variance. Knuth samples correspond to path-based rollouts, allowing us to integrate them with MCTS. Extensive follow-up work has developed alternatives for approximating tree size (e.g., [11, 12, 37, 57]), but these estimates are not decomposable into path-based root-to-leaf rollouts, and would require new ideas to be integrated into MCFS. Knuth Synthesis is designed to be used offline to synthesize new solvers under the data-driven algorithm design paradigm [4], where we optimize algorithm performance over a training set of instances. As is standard in the literature, we encode our policies using deep neural networks, which are far more expensive to evaluate than standard heuristics. Such policies are too expensive to evaluate at every node of the search tree. We mitigate this cost by limiting our learned policy to the search tree’s most important nodes, querying some existing subsolver below a certain depth in the proof tree; this also substantially reduces the policy space. To ensure that we find strong policies for online use, we enforce the same procedure offline for constraining on which nodes to call the policy. Figure 2 illustrates the two key ideas for making policy evaluation tractable.

Knuth Synthesis is the first MCTS-like method that has been used to learn a branching policy for combinatorial search. We deviated from standard MCTS ideas as appropriate, notably making a novel change to the bandit algorithm to account for our tree size cost function. Unlike AlphaZero, which uses MCTS both offline and online, we use Knuth Synthesis only offline, as we cannot afford the computational overhead online; we only query our policy online.

⁴ Other more practically useful algorithms such as CDCL and Branch-and-Bound cannot easily be represented as tree MDPs because they correspond to deeply history-dependent policies; we are not aware of any RL methods that are tractable for learning policies for such algorithms.

We evaluated our method on specific unsatisfiable problem distributions where \approx 100,000 decisions were required to solve problems during training, a considerable leap beyond other prominent published work. We note that focusing on either satisfiable or unsatisfiable instances is commonplace in SAT. There are many examples of specialized algorithms targeting one of satisfiable or unsatisfiable instances; indeed, there have been specialized tracks of the SAT competition for both cases. For example, local search is a very well-studied class of algorithms that cannot prove unsatisfiability. In practice, one can always use a portfolio strategy running specialized SAT and UNSAT algorithms in parallel and terminating when either returns an answer. Solving such problems efficiently is important in practice, e.g., for system debugging [70] and formal verification [9].

First, we evaluated our method on uniform random 3-SAT at the solubility phase transition, perhaps the best-studied SAT distribution (e.g., featured in the SAT Competition [1] from 2002 to 2018). We matched the performance of `kcnfs`, which was specifically designed to target this distribution. Second, we evaluated our method on the `sgen` [69] distribution, which is notoriously difficult for its problem size. We improved running time on `sgen` by 8% over `kcnfs`, which in turn is $3.2\times$ faster on this distribution than the `hKis` solver that solved the most unsatisfiable instances in the 2021 SAT Competition. Lastly, we evaluated our method on the `satfc` [22] distribution, consisting of radio-spectrum repacking feasibility problems from an FCC spectrum auction, improving running time by 28% over `kcnfs`. These results show the initial promise of MCFS; through further scaling, we believe these ideas will lead to stronger industrial solvers for specific applications.

2 Related Work

Branching Tree search is a fundamental algorithm for combinatorial optimization that iteratively partitions 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. The order in which variables are assigned—the branching policy—has a dramatic effect on the size of the tree and the corresponding time to solve the problem, so algorithm designers seek branching policies that lead to small trees. In domains such as Mixed Integer Programming (MIP) solving, a “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, a “simple and cheap” paradigm dominates: high-performance SAT solvers make very cheap branching decisions that depend only superficially on subproblem state. After 20 years, Variable State Independent Decaying Sum (VSIDS) [51] has remained among the state of the art for Quantified Boolean Formula (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 [31], but these generally could not compete with VSIDS in terms of running time. A few notable exceptions are: (1) look-ahead SAT solvers, which have a similar concept of a lookahead as MCFS and choose a branching variable after looking ahead at the resulting state for different possible branches [26, 28]; (2) the bsh heuristic used in the `kcnfs` solver [18] which branches based on a measure of how constraining the assignment

of one variable is on the assignment of another; (3) Bayesian moment matching to initialize VSIDS per-variable scores for literals based on how likely they are to be part of a satisfying assignment [19].

There is now a long history of machine learning being used for automated algorithm design [6, 10, 29, 41]. One might expect that machine learning could learn more informative branching heuristics for tree search that are worth their cost, especially for shallow-depth branches where branching decisions are most consequential. Two approaches have shown promise for learning models to make branching decisions: imitation learning and reinforcement learning.

First, imitation learning works by learning a cheap approximation of (1) an expensive existing heuristic [23, 30, 35, 52] or (2) an expensive feature that is a good proxy for a good branching decision [64, 75]. Khalil et al. [35] learned a cheaper approximation of the strong branching heuristic in MIP using hand-crafted variable-level features to achieve state-of-the-art performance; Gasse et al. [23] and Nair et al. [52] learned an improved approximation of strong branching by learning a feature representation by representing MIPs as graphs and leveraging graph neural networks. Relevant to SAT, Wang et al. [75] predict backdoor variables and integrate these predictions into the VSIDS heuristic while Selsam and Bjørner [64] learned to approximate small unsatisfiable core computation and then branched on variables predicted to belong to a core. Both report practical performance improvements. Selsam and Bjørner’s work is particularly relevant to the UNSAT setting, which we focus on. They computed small unsatisfiable 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 unsatisfiable core, there exists a corresponding short proof, but not vice versa. In other related SAT work, Nejati et al. [53] trained a branching policy to predict the branching variable that leads to the smallest running time from a single branch starting at the root of the tree with a fixed downstream solver. Their motivating application is parallel SAT solving where each branch of the tree are solved in parallel.

Second, it is tempting to use reinforcement learning to directly synthesize heuristic policies that are optimized for problem distributions of interest. This idea dates back to Lagoudakis and Littman [43], who used TD-learning to train a policy to select between seven predefined branch heuristics based on simple hand-crafted features. With the advent of modern deep learning architectures, practitioners train models that take the raw representation of a problem instance as the model input. Yolcu and Póczos [77] learned a local search heuristic for SAT; Tönshoff et al. [73] built a generic graph neural network-based method for iteratively changing assignments in CSPs; Lederman et al. [45] used the REINFORCE algorithm to learn an alternative to the VSIDS heuristic for QBF that improved CPU time within a competitive solver; [68] used a Q-learning approach for to learn a branching policy for CSP; and Vaezipoor et al. [74] used a black-box evolutionary strategy to learn a state-of-the-art branching heuristic for model counting, demonstrating improvements in walltime performance over the competitive SharpSAT solver. There are also various applications of RL for learning to branch in MIP [21, 54, 61]. Finally, Kurin et al. [42]’s work, which used Q-learning for branching in the CDCL solver Minisat, is closest to our own. They trained on random satisfiable problems with 50 variables

and generalized online to 250 variables for both satisfiable and unsatisfiable instances and improved Minisat running time. Instances in their training set required on the order of 100 decisions. The authors found that increasing the difficulty of the training set generalized poorly and training was less stable; the authors cite higher variance returns from longer episodes, challenges for temporal credit assignment, and difficulties with exploration. They resolved the path/tree distinction by treating a traversal through a tree as a path (which could be exponential in length) and allowing backtracking state transitions. This, coupled with not incorporating the stack of backtracking points into their state violates the Markov assumption when backtracking, which is necessary for proving unsatisfiability and is important for satisfiability for all but the optimal policy.

We think that the problem of using RL to train a branching policy deserves study even if the field remains a few steps behind practical relevance. The primary challenge arising in this body of past work is scaling RL methods to more practical problems requiring more decisions. Each of these RL methods do exact policy evaluation and have only been trained with ≤ 1000 decisions and are therefore constrained to training on easy instances. From the perspective of a modern SAT solver, many of these results may appear trivial, but they nevertheless quantify the extent to which RL methods can learn policies that improve on hand-tuned heuristics.

RL with trees Scavuzzo et al. [61] were the first to cleanly pinpoint how the structure of the tree-search algorithm changes credit assignment in RL, introducing the concept of *tree MDPs* and formulating a policy gradient for tree policies. Much earlier, Lagoudakis and Littman [43] recognized that the one-to-two state transition violated the MDP definition and resolved this by cloning the MDP and creating one copy for each transition. A number of other recent papers have also realized the inefficiency of credit assignment when treating an episode as a path (rather than a tree) in a tree-search algorithm [21, 54, 68].

Approximating Tree Size We approximate the tree size of a DPLL policy via *Knuth samples* [39], which Lobjois and Lemaître [48] showed can be effective for cheaply comparing algorithms despite its high variance. A Knuth sample provides a path-based rollout, which allows us to easily leverage MCTS. There has been extensive follow-up work developing alternatives for approximating tree size (e.g., Chen [11], Cornuéjols et al. [12], Kilby et al. [37], Purdom [57]), but these estimates are not decomposed into path-based rollouts from root to leaf, and require new ideas to be integrated into MCFS.

MCTS for Combinatorial Problems Various researchers have used MCTS as an algorithm to directly search for a satisfying solution to a CSP that lie somewhere between local search and DPLL [34, 49, 56, 62]. In these cases, a rollout can be interpreted as a guess at a satisfying assignment. If an unseen node is reached or a conflict is reached along a rollout path, a reward is assigned based on some measure of how close the path is to being a satisfying assignment (e.g., number of satisfied constraints). We see two main differences between these approaches and ours: (1) they use MCTS online to solve CSPs rather than as an offline procedure for training model-based branching policies and

(2) they are not designed for the unsatisfiable case where policies produce trees rather than paths.

Previti et al. [56] developed UCTSAT, which assigns 0 reward to a conflict node and explores various alternatives for value estimation at non-terminal nodes such as (1) number of satisfied clauses at current state and (2) average number of satisfied clauses based on random paths from that state. A number of follow-up works have incorporated clause learning into UCTSAT. Schloeter [62] added new clauses to the problem for every conflict that is encountered. This addition does not affect the UCT tree since state is defined as the set of assigned variables and therefore independent of additional clauses. Keszocze et al. [34] made better use of the learned clauses, filtering out any expanded nodes that are ruled out by a learned clause. They modified Previti et al.’s reward function by weighting each satisfied clause by its activity (number of times occurring in resolution) and propose a number of other variations including penalizing by conflict depth; we penalized by 2^{depth} , which corresponds to minimizing the size of the proof tree. [49] developed a version of UCTSAT for Constraint Programming (CP) that was simplified to be state independent. Rather than each tree node representing a different multi-armed bandit (MAB), they have a single MAB that they update at every node of the tree. This change was made to incorporate restarts, where there would be too few samples for each node. They reward a path by the length until conflict. Wattez et al. [76] used a single MAB to choose which branching heuristic to use on a given instance. They rewarded a path to a conflict by the number of assignments ruled out by that path.

Outside of CSP, there are a number of other existing uses of MCTS for solving *path-based* NP-hard problems. Browne et al. [8] 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. [2] searched over assignment paths in graph problems such as choosing edges to cut in a graph; and Khalil et al. [36] searched over paths of variables to find MIP backdoors.

Relation to conventional ML for SAT There is a significant amount of work over the past few decades in ML for SAT using algorithm selection [41] and algorithm configuration [29], however we do not leverage those ideas in our work. We think that RL for learning a branching policy is conceptually distinct from selection and configuration. We see selection and configuration as hand-engineered ML; they rely on either experts exposing algorithm parameters or instance features. On a spectrum of “degree of automation”, we think “RL for branching” falls somewhere between hand-engineered ML and pure end-to-end approaches (e.g., NeuroSAT [64]) where the network directly outputs a SAT solution. We leverage the scaffolding of existing tree-search solvers but we are flexible to learn any arbitrary state-dependent policy within that space. We could in principle define our problem of learning a branching heuristic as an algorithm configuration problem (i.e., the neural net parameters are our algorithm parameters) but most algorithm configuration approaches use black-box optimization whereas we are opening up the black box by setting it up as an RL problem.

We also note that our method is compatible with algorithm selection or configuration at the level of the *subsolver*. Knuth Synthesis is agnostic to the choice of subsolver; we could in principle train an algorithm selector or configure a solver as the subsolver and

we think there could be some exciting synergies there. For example, we considered a novel algorithm selection setting where we would select amongst solvers for different subproblems within a single instance, rather than the typical setting where one algorithm is selected per instance. That would elicit a new design dimension; we would want to find branching policies that produce subproblems that can exploit the complementary of an algorithm portfolio.

3 Preliminaries

We now provide the required technical background for MCTS, the Boolean satisfiability problem (our application area), the DPLL algorithm (the framework that defines our policy space), and tree MDPs (the class of problems to which we apply MCFS).

3.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a general-purpose RL algorithm framework for MDPs. An MDP $M(\mathcal{S}, \mathcal{A}, p, r)$ is defined by a set of states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, transition distribution $p(s_{i+1}|s_i, a_i)$, and reward function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. For every state s_i that is visited, MCTS stores a vector of counts $c_i \in \mathbb{Z}^{|\mathcal{A}|}$ and value estimates $v_i \in \mathbb{R}^{|\mathcal{A}|}$. A *rollout* θ is a sequence of state, action pairs $((s_0, a_0), \dots, (s_n, a_n))$. MCTS makes a series of rollouts starting from the current state s_0 defined with the below four steps. Our definition is more general than presented in Sutton and Barto [72] to help us later define MCFS, notably the introduction of γ that separates the concepts of (1) choosing an action from (2) terminating a rollout.

1. **Selection.** A *tree policy* $\alpha : s \in \mathcal{S}, c \in \mathbb{R}^{|\mathcal{A}|}, v \in \mathbb{R}^{|\mathcal{A}|} \mapsto \mathcal{A}$ selects an action based on the action values v , counts c , and often a prior that depends on s . Then a *step policy* $\gamma : \theta \in (\mathcal{S} \times \mathcal{A})^n, a \in \mathcal{A}, s \in \mathcal{S} \mapsto \mathcal{A} \cup \emptyset$ takes in the state s_i , action a_i , and the current rollout history $((s_0, a_0), \dots, (s_i, a_i))$ and chooses to either play the action selected by α or to play no action (\emptyset), terminating the rollout at that state.
2. **Expansion.** Counts and value estimates are tracked for any state reached by our selection step and together form a tree rooted at the current state. For any previously unexplored state s_i along the path, we add a node with vectors c_i and v_i to our tree.
3. **Simulation.** *rollout policy* $\pi : \mathcal{S} \mapsto \mathbb{R}$ then estimates the sum of rewards before reaching terminal state of the MDP from the node where no action was played. π is commonly a value network estimating the value of a path originating from the node's state.
4. **Backup.** Pass back rewards through the path of the MCTS tree, so that v_i at s_i are updated with the sum of rewards from its descendants: $\pi(s_n) + \sum_{x=i+1}^{n-1} r(s_x)$.

After a fixed number of rollouts, MCTS commits to the action at the root according to accumulated statistics of the tree (e.g., for AlphaZero, the largest number of samples) and the resulting state becomes the new root node. This procedure repeats until MCTS finds a path to a leaf of the tree. It is then possible to train a policy network to approximate MCTS by using the counts c_i and values v_i of each node along this path as training examples. The policy network can then be used within α to further focus rollouts on promising paths as in AlphaZero.

3.2 Boolean Satisfiability

A Boolean satisfiability (SAT) instance S is defined by a set of clauses $C = \{c_1, \dots, c_m\}$ over a set of variables $X = \{x_1, \dots, x_n\}$. 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 an assignment of variables to demonstrate that a problem is satisfiable, or to construct proofs showing that no setting of the variables can satisfy the problem. We consider only unsatisfiable problems.

3.3 DPLL and Variable Selection Policies

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 or any subproblem within a larger instance. A policy ϕ is a mapping $\phi : S \rightarrow (v)$ that determines which variable v to assign in DPLL.

Given a policy, the DPLL algorithm selects a variable to branch on and recursively checks both the *True* and *False* assignments in a tree-like fashion, performing *unit propagation* at each step. Unit propagation assigns variables forced by single-variable (*unit*) clauses; propagates them to other clauses; and repeats until no unit clauses remain. Each recursive DPLL call terminates when a *conflict* (variable assignments form a contradiction) is found, forming a *proof tree* (e.g., Figure 1). There can be massive gaps in performance between variable selection policies. For example, Figure 1 shows a formula that leads to a three-node proof tree if x_6 is selected first by the policy (assigning x_6 results in $x_7 \wedge \neg x_7$, implying a contradiction), but a tree that could have as many as $2^{|X|} - 1$ nodes if x_6 is selected last. This also illustrates how top-level decisions are much more powerful; the proof tree doubles in size for every level at which a good branching decision (branch on x_6) is not made. 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 require fewer decisions overall and therefore yield smaller search trees.

For an instance S 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 $\mathcal{L}(\phi; \mathbb{P}) = \mathbb{E}_{S' \sim \mathbb{P}} [T_\phi(S')]$. Finding policies is computationally challenging; for an n -variable problem, there are $O(n^{3^n})$ possible variable selection policies (3^n states representing each variable as *True*, *False*, or unassigned, and n choices per state); and exact evaluation of $T_\phi(S)$ takes $O(2^n)$ operations; Liberatore [46] showed that identifying the optimal DPLL branching variable at the root decision is both NP-hard and coNP-hard.⁵ If we assume

⁵ Liberatore [47] later showed the problem is $\Delta_2^p[\log(n)]$ -hard i.e., at least poly time with $\log(n)$ oracle queries to an NP problem.

that the optimal variable selection policy, $\phi_{\mathbb{P}}^* = \arg \min_{\phi'} \mathcal{L}(\phi'; \mathbb{P})$, is learnable by 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 to design a procedure that efficiently minimizes $\mathcal{L}(\phi; \mathbb{P})$ so that labeled training examples can be collected through rollouts of learned policies. This is a challenging reinforcement learning problem. For a particular variable selection policy, even evaluating the loss function on a single instance takes time exponential in n , and the number of policies is doubly exponential in n .

3.4 Tree MDPs

Tree MDPs [61] are a generalization of MDPs with 1-to-many transitions. Similar to the Markov property of MDPs, tree MDPs have a *tree Markov property*: each subtree depends only on its preceding state and action. We show how the DPLL algorithm can be represented with tree MDPs in Appendix A.2.

Definition 2 (Scavuzzo et al. [61]). *Tree MDPs are augmented MDPs $tM = (\mathcal{S}, \mathcal{A}, p_{init}, p_{ch}^L, p_{ch}^R, r, l)$, with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, initial state distribution $p_{init}(s_0)$, left and right child transition distributions $p_{ch}^L(s_{ch_i}^L | s_i, a_i)$ and $p_{ch}^R(s_{ch_i}^R | s_i, a_i)$, reward function $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ and leaf indicator $l : \mathcal{S} \rightarrow \{0, 1\}$. Each non-leaf state s_i (i.e., such that $l(s_i) = 0$), together with an action a_i , produces two new states $s_{ch_i}^L$ (left child) and $s_{ch_i}^R$ (right child). Leaf states (i.e., such that $l(s_i) = 1$) are the leaf nodes of the tree, below which no action can be taken.*

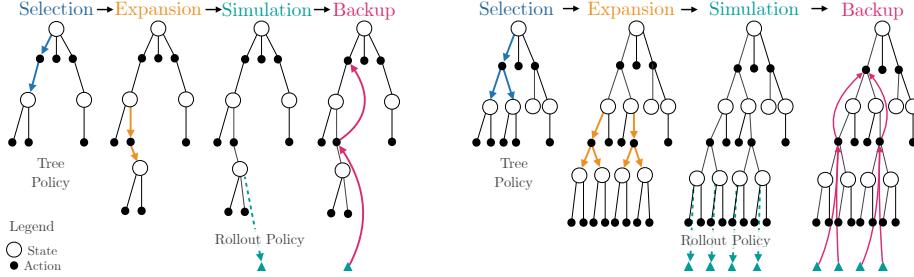
4 Monte Carlo Forest Search

We define Monte Carlo Forest Search (MCFS) as a class of RL algorithms for learning policies in *tree MDPs*. An MCFS rollout θ is a tree of (state, action) pairs structured according to the underlying tree MDP where a parent node s_i shares edges with its children $s_{ch_i}^L$ and $s_{ch_i}^R$. The aggregation of *rollout trees* forms a forest, hence the name. The Expansion step is defined the same as MCTS and so is the Simulation step with the exception of being applied to many states rather than just one. We define the Selection and Backup step for an MCFS rollout below. See Figure 3 for a side-by-side illustration and pseudocode of MCTS and MCFS in their most general forms.

Selection. When an action selected by α is played by γ at state s_i , the state transitions to new states $s_{ch_i}^L$ and $s_{ch_i}^R$ from sampling $p_{ch}^L(s_{ch_i}^L | s_i, a_i)$ and $p_{ch}^R(s_{ch_i}^R | s_i, a_i)$. These new states are added to a queue q . MCFS iterates over q , calling α and adding the corresponding new states back to q if γ chooses to step and terminates when q is empty.

Backup. The rewards are passed back through θ , so that the value estimate v_i at s_i is updated with the sum of rewards from its descendants in θ .

We can now see the importance of defining the step policy γ ; it allows us to evaluate any subset of the full proof tree in a *rollout tree* which is important when exact policy evaluation is intractable. MCFS commits to the action a at the root in the same way as MCTS does and the corresponding child nodes $s_{ch_0}^L, s_{ch_0}^R$ become the roots of new search trees and are solved sequentially. This procedure repeats until MCFS finds a tree where every leaf s_i of the tree is a tree MDP leaf ($l(s_i) = 1$).

Algorithm 1: MCTSRollout(α, γ, π)

```

Input:  $s$ 

 $\theta \leftarrow$  list containing  $S$ 
while  $\gamma(s)$  is not  $\emptyset$  do
     $s \leftarrow \alpha(S)$ 
    add  $s$  to  $\theta$ 
end while

expand( $\theta$ ) /*Expand nodes in  $\theta$  yet to exist*/
for node  $n$  in path (reverse order from leaf) do
    if  $n$  is leaf then
         $\text{value}(n) = \pi(n)$ 
    end if
    child value =  $\text{value}(\text{child}(n))$ 
     $\text{value}(n) = \text{reward}(n) + \text{child value}$ 
    Update( $n$ ,  $\text{value}(n)$ )
end for

```

Algorithm 2: MCFSRollout(α, γ, π)

```

Input:  $s$ 

 $\theta \leftarrow$  tree with  $s$  as root
 $q \leftarrow$  FIFO queue containing  $s$ 
while  $q$  is not empty do
     $s \leftarrow \text{pop}(q)$ 
     $S_{ch}^L, S_{ch}^R \leftarrow \alpha(s)$ 
    if  $\gamma(s)$  is not  $\emptyset$  then
        Add  $s$  to  $\theta$ 
        add  $S_{ch}^L, S_{ch}^R$  to  $q$ 
    end if
end while

expand( $\theta$ ) /*Expand nodes in  $\theta$  yet to exist*/
for node  $n$  in tree (bottom up) do
    if  $n$  is leaf then
         $\text{value}(n) = \pi(n)$ 
    end if
    children value =  $\sum_{i \in \text{Children}(n)} \text{value}(i)$ 
     $\text{value}(n) = \text{reward}(n) + \text{children value}$ 
    Update( $n$ ,  $\text{value}(n)$ )
end for

```

Fig. 3: Side-by-side comparison of MCTS and MCFS highlighted by the Selection, Expansion, Simulation and Backup steps (Adapted from Sutton and Barto [72]).

5 Knuth Synthesis

For tree MDP problems like DPLL for UNSAT or model counting, the tree depth can be linear in the number of actions; therefore the tree size (and thus the cost of evaluating a policy) can grow exponentially with the number of actions. Knuth Synthesis is an MCFS implementation based on two key ideas that avoid the prohibitive costs of exact policy evaluations (See Figure 2). First, in the step policy we nest a sampling procedure within a given Monte Carlo rollout, using a set of random paths through the rollout tree to approximate tree size. Second, we bound the depth of our forest search; below this point, we call out to a rollout policy that leverages existing SAT solvers. We describe these ideas in the following subsections and then provide our full implementation.

5.1 Nested Monte Carlo Sampling

To obtain a Monte Carlo sample through our tree MDP and determine the associated tree size $T_\alpha(S)$, we may need to visit $O(2^n)$ states. This is computationally expensive, both because of the absolute number of states and because at each node in the tree we need to query α to decide which variable to assign next (which is nontrivial when α is parameterized by a neural network). To address this issue, we nest a sampling procedure within a given Monte Carlo rollout sample, effectively Monte Carlo sampling a Monte Carlo rollout of our tree policy. More precisely, we stochastically approximate $T_\alpha(S)$ by using Knuth samples [39]. We first state the general version of Knuth’s theorem for estimating the total weight of a weighted k -ary tree with a random path through the tree.

Theorem 1 (Knuth [39]) *Let N be the set of nodes in a weighted k -ary tree T where each node $n \in N$ has weight $w(n)$. The total weight of this tree is $w_T = \sum_{n \in N} w(n)$. Let (n_0, \dots, n_ℓ) be a path of nodes through the tree from root to leaf that is chosen uniformly at random. Then, $w_T = \mathbb{E}_{(n_0, \dots, n_\ell)} \sum_{i=0}^\ell k^i w(n_i)$.*

In the case of binary trees where all nodes have weight 1, a simple corollary holds.

Corollary 1 *Let ℓ_P be the length of a path P (n_0, \dots, n_{ℓ_P}) where $\forall i, w(n_i) = 1$ and sampled uniformly at random from a binary tree T with size s_T . Then, $s_T = \mathbb{E}_P [2^{\ell_P+1} - 1]$.*

Proof. When $w(n_i) = 1$, $s_T = E_P[\frac{k^{\ell_P+1}-1}{k-1}]$. This is true because $\sum_0^{n-1} k^i = \frac{k^n-1}{k-1}$. Plugging in $k=2$ (binary tree in UNSAT), we get $s_T = E_P[2^{\ell_P} - 1]$.

For the tree T produced by α , we can use Theorem 1 to get an unbiased estimate of $T_\alpha(S)$. In the context of a DPLL solver, a Knuth sample amounts to replacing a complete traversal of the binary tree of all *True / False* assignments with a path through the tree where assignments are chosen uniformly at random. We can take the length of the resulting path, ℓ , and update the tree size estimate of each node at depth d with $2^{\ell-d} - 1$ for the corresponding decision by α . The average of these estimates across a set of paths yields an unbiased approximation of the tree size.

5.2 Bounding Policy Evaluations

Knuth Synthesis is an offline procedure for training a variable selection policy ϕ . At test time, we want proofs of unsatisfiability rather than estimates of tree size, and so cannot use Knuth samples. We represent ϕ using a deep neural network, which is far more expensive to evaluate than conventional tree-search heuristics. This raises the risk that the computational cost of evaluating ϕ will exceed its benefits via proof tree size reductions, and indeed makes this risk a certainty for small enough subproblems, such as those for which existing solvers are faster than single network calls. Any online policy must therefore have a procedure for constraining the nodes at which ϕ will be queried. As we saw earlier, decisions become more important the closer they are to the root of an UNSAT search tree. We thus apply ϕ at states having depth $\leq \ell$, after which we either (1) at both test and training time, call a “subsolver” (a pre-existing variable selection

policy); or (2) at training time, sometimes instead call a value network that more cheaply approximates the tree size of this fixed policy (see “Rollout Policy” for details).

Although we could potentially afford to explore larger ℓ values at training time, we do not do so; this ensures that the training phase identifies a policy that will leverage the subsolver appropriately at test time. To determine the appropriate tree size estimation, we add a weight of $T_{\phi_{sub}}(S^*)$ to any node representing state S^* at which we call the subsolver policy ϕ_{sub} . This represents the additional tree size incurred from the subsolver call at this node. For a given Knuth sample of length ℓ that terminates at S^* , we can use the weighted version of Knuth’s theorem to update the value of a node at depth d along the path with

$$2^{\ell-d}T_{\phi_{sub}}(S^*) + 2^{\ell-d} - 1. \quad (1)$$

5.3 Implementation

Beyond changes to avoid exponential evaluation costs, Knuth Synthesis has a few other noteworthy components. We alter the Action Selection step to account for the tree size cost function that scales with node depth; we adapt the Simulation step to account for an unreliable value network and a fixed policy; we use a model architecture to respect the structural invariances of SAT; we use a graph-based state-transition data structure to improve sample efficiency; and we share policy priors with child nodes to speed up lookaheads. Find pseudocode in Appendix A.

5.3.1 Tree Policy α

At every state s , α returns action

$$a = \arg \min_{a'} (Q(s, a') - Q_d U(s, a')). \quad (2)$$

$Q(s, a)$ is the cost estimate of action a at state s , Q_d is the running average tree size of nodes at depth d , which is used to calibrate confidence intervals across different depths, and

$$U(s, a) = c_{PUCT} P(s, a) \frac{\sqrt{\sum_{a'} N(s, a')}}{1 + N(s, a)} \quad (3)$$

is the corresponding confidence interval [58, 65]. The confidence interval is parameterized by $N(s, a)$, the number of lookaheads that branch on action a at state s , c_{PUCT} , a constant that controls exploration, and $P(s, a)$, a prior distribution over the actions for a given state, predicted by the policy network. $N(s, a)$ is initialized to 1 and $Q(s, a)$ is initialized with the tree size of the first lookahead at that state. This provides an unbiased measure of the performance of our incumbent policy (i.e., the first sample is exactly the neural network policy), which we seek to improve upon. This is similar to AlphaZero’s choice of initializing Q to 0, which is the baseline they aim to improve upon. Because $Q(s, v)$ scales exponentially with the depth of s , we introduce Q_d to calibrate confidence intervals. Since $U(s, v)$ is independent of the scale of costs (it is only a function of counts), we calibrate each depth with Q_d to keep $U(s, v)$ at the same scale as $Q(s, v)$. Otherwise, the choice of the c_{PUCT} would trade off tight confidence intervals at shallow nodes against loose confidence intervals at deeper nodes. We compute Q_d online as the running average of tree sizes encountered at depth d .

Equivariant Architecture SAT problems vary in size and are invariant to permutations of their clauses and variables within clauses. We enforce these invariances using a permutation equivariant architecture in which permutations of the input guarantee a corresponding permutation in the output [25]. A beneficial side effect of this architecture is that it can handle input matrices of any size.

Sharing Prior with Child Nodes Calls to the policy network tend to dominate running time, so we save time by only computing $P(s, a)$ once at the root of the MCFS forest and passing down the prediction to its child nodes. A child state has a subset of the action space of its parent state. For child state s' , we set $P(s', a) \leftarrow P(s, a)$, and renormalize for the subset of actions remaining in s' .

Directed Acyclic Graph as Forest Search Data Structure We improve sample efficiency by changing our forest search data structure from a tree to a graph, which leverages the fact that states are reached independently of the order of previous decisions [16]. This change ensures information sharing across different paths that lead to the same state.

5.3.2 Step Policy γ An MCFS Step policy γ controls how much of the proof tree to evaluate in a rollout. In order for γ to implement Knuth sampling, it should play actions along a single path through the rollout tree, choosing actions uniformly at random. We must therefore ensure that for every pair of newly visited states (s^L, s^R) , γ only plays an action at one of them. If γ is called at a state s and its sibling s^{sib} has yet to be processed, then γ chooses between $\alpha(s,)$ and \emptyset with uniform probability. If s^{sib} has already been processed, then γ returns the opposite of the decision made at s^{sib} . γ will also return \emptyset if it encounters a state s at depth ℓ or it encounters a leaf node (s s.t. $l(s) = 0$) i.e., a conflict in the underlying SAT problem.

5.3.3 Rollout Policy π At each state s , π can be described via two cases: (1) if s 's sibling s^{sib} played an action (i.e. $\gamma(s^{sib}) \neq \emptyset$) then $\pi(s)$ returns the cumulative reward at s^{sib} , ensuring that the reward estimates at any given node reflect the correct Knuth estimate; (2) no action was played at s or its sibling because s is at depth ℓ . In this case our rollout policy π is to call the subsolver at a state S^* at depth ℓ of our Knuth sample. Rather than explicitly solving the subproblem at depth ℓ , we use a value network to access a much cheaper reward signal. This network is trained from an initial batch of subsolver calls and then retrained as we collect new batches of data, so its accuracy varies across time steps of a training run. When the output of the network is too unreliable, its signal about the true number of decisions may be insufficient for policy learning. We address the issue of an unreliable network by randomly deciding between calling the subsolver or the value network, with the probability depending on an online estimate of the value network's accuracy. Specifically, we track the mean multiplicative error ϵ of our value network over time by querying the value network with every subsolver call. For 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 that the probability of calling the subsolver halves as the error halves.

6 Experimental Setup

6.1 Benchmarks

We targeted instance distributions that are well known and difficult for modern SAT solvers. We controlled instance size so that (1) the size of the action space was similar to Go and (2) solving required $\approx 100,000$ decisions. We did not consider industrial SAT Competition instances, as they often contain millions of variables with state spaces significantly larger than any deployed MCTS application of which we are aware. We evaluated our approach on three distinct distributions: (1) a canonical random distribution: uniform random 3-SAT at the solubility phase transition (`R3SAT`), (2) a notoriously difficult crafted distribution (`sgen` [69]), and one real-world application: station repacking problems from the 2016 FCC incentive auction (`satfc` [22]). For `R3SAT`, we trained on 300-variable instances, for which calling a subsolver was quick (≈ 1 second solving time), and filtered out satisfiable instances for training and testing. To evaluate upward size generalization, we set aside 100 test instances at both our training size of 300 variables as well as at 350, 400, and 450 variables. For `sgen`, we trained on 65 variables; to evaluate upward size generalization we set aside 100 test instances at both our training size of 65 variables as well as at 75, 85, and 95 variables. For `satfc`, we trained on small (<2000-variable) instances from localized regions of the U.S. interference graph, and filtered out satisfiable instances. We set aside 100 test instances of our training distributions. For full details, see Appendix C.

6.2 Baselines

Since Knuth Synthesis learns DPLL policies, we sought to evaluate against other purely DPLL solvers rather than to make apples-to-oranges comparisons to clause learning (CDCL) solvers. We ultimately aim to integrate our MCFS ideas into the much richer design space of CDCL solvers, but anticipate that this will require going beyond the tree MDP formalism. To make our comparisons as fair as possible, we set our baseline and subsolver to be the same: `kcnfs` [18], which is a pure-DPLL solver and is specifically designed for and is among the strongest solvers for `R3SAT`. We used its most recent competition submission at the 2007 SAT Competition [7], where it won the silver medal in the Random UNSAT track and had previously won the gold medal in the 2005 Random UNSAT track. The lookahead `march` solver [27] is $\approx 10\%$ faster but we chose `kcnfs` because it is a pure-DPLL solver. `kcnfs` is a much stronger baseline than the `minisat` baseline which Kurin et al. [42] compare to (two orders of magnitude faster on their benchmarks). We also evaluated a `uni+kcnfs` baseline on each dataset, where we replaced Knuth Synthesis neural network calls with uniform-at-random decisions and called the `kcnfs` solver for subproblems at the same user-defined depth. We tried a purely random policy without calling `kcnfs` as a subsolver on 65-variable `sgen`, which led to poor performance $40\times$ slower than `kcnfs`.

6.3 Knuth Synthesis

We integrated our Knuth Synthesis algorithm into the CDCL solver `Maple_LCM_Dist_ChronoBT` [60], which won the 2018 SAT Competition and is based on the MiniSAT

framework [20]. We removed all clause-learning components so that the solver ran pure DPLL search. We trained our neural networks with PyTorch [55] in Python and ported them to our C++ solver using tracing. Find our code here: <https://github.com/ChrisCameron1/MCFS>.

Two important hyperparameters were (1) the constant for the level of exploration c_{PUCT} and (2) the number of lookaheads k . Using a coarse grid search, we selected $c_{PUCT} = 0.5$ and $k = 100,000$, which found the best policies within a 48-hour window. We chose $\ell = 5, 6$, and 8 for the `satfc`, `R3SAT` and `sgen` distributions respectively, based on the average tree size of Knuth Synthesis after 48 hours for values between 2 and 10 over a few instances. This ℓ parameter trades off between (1) flexibility of the policy (neural net makes more decisions) and (2) size of the search space. There is a sweet spot where the search space is large enough to find an improving policy but small enough that we can learn that policy in a reasonable amount of time. Fewer variables in a problem means we can search deeper for an equivalently-sized search space so that likely explains why we observed better results for larger ℓ as the number of variables increased: 5 for `satfc` (≈ 1000 vars), 6 for `R3SAT` (300 vars), 8 for `sgen` (65 vars). We search deepest in `sgen` where there are fewest variables. The best ℓ will also depend on the amount of computation we allocate for Knuth Synthesis; with more compute and/or a more efficient MCFS algorithm, we could search considerably deeper. We made decisions with Knuth Synthesis until depth ℓ ; each MCFS decision yielded a training point consisting of a (state, policy vector, Q -value) triple, where the policy vector represents normalized action counts from MCFS.

Instead of uniformly random sampling true/false variable assignments during Knuth samples, we tried to reduce variance by importance sampling based on value network estimates. We did not observe any variance reduction likely because our value network was not sufficiently strong.

Before a good policy network is learned, Knuth Synthesis tends to be less efficient. We pretrained our policy and value networks by running Knuth Synthesis with 10,000 lookaheads on 1,000 instances for `R3SAT` and `sgen`. We ran one iteration of Knuth Synthesis with 10,000 lookaheads on 1,000 instances to further improve the policy and value network, and then a final iteration with 100,000 lookaheads on 2,000 instances to train our final model. Pretraining runs of Knuth Synthesis took approximately 24 hours per instance and were respectively run on 300-variable and 55-variable problems from `R3SAT` and `sgen`. The subsequent iterations took approximately 48 hours per instance. On `R3SAT` and `sgen`, they were respectively run on 300-variable and 65-variable problems. For `satfc`, we pretrained with 10,000 lookaheads on 441 instances and ran a final iteration using the prior with 100,000 lookaheads. The parameters and architecture described in this paper are only for the last iteration of Knuth Synthesis. We made several minor improvements across iterations.

6.4 Model Training

We used the exchangeable architecture of Hartford et al. [25]. 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 128-dimensional trainable embeddings representing the true and false literal. Following Hamilton et al. [24], 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 did not observe any improvements.

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 and the Q -value head were trained to predict the normalized counts and Q -values of Knuth Synthesis, respectively. We optimized cross-entropy loss for both the policy and Q -value head. The purpose of the Q -value head was as an auxiliary task to help learn 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 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; loss tended to double when using a network trained with only a value head. 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. We used the Adam optimizer [38], with a learning rate of 0.0001 and a batch size of 1. We evaluated mean tree size on a held-out validation to select a model. Online, we branched on the argmax variable from our neural network prediction.

6.5 Computing Resources

We ran our model training and solver benchmarking experiments on a shared cluster with A100 GPUs, equipped with 32 2.10GHz Intel Xeon E5-2683 v4 CPUs with 40960 KB cache and 96 GB RAM each, running openSUSE Leap 42.1 (x86_64). For Knuth Synthesis runs, we used a large shared cluster of CPU nodes with 1109 nodes, equipped with 64 2.40 GHz 2 x AMD Rome 7532 with 256 MB of L3 cache. Each Knuth Synthesis run was allocated 16 GB of memory and a maximum of 48 hours for `R3SAT` and `sge` and 72 hours for `satfc`. Each benchmarking run was allocated 8 GB of memory.

7 Results

We evaluated (1) search tree size and (2) running time for Knuth Synthesis against the two baselines on each of our instance sets; the full results are presented in Table 1. We measured running time as the cumulative CPU and GPU time with `runsolver` [59]. Overall, we reduced tree size on each of the three training distributions as well as 5/6 upward-size generalization distributions. This led to running time improvements on 3/9 datasets where the neural network overhead was manageable. `R3SAT` was the most challenging benchmark as `kcnfs` is an extremely strong solver specialized to this

	Size	In/Out	Tree size (1000s)			CPU+GPU time (s)		
			uni+kcnfs	kcnfs	Ours	uni+kcnfs	kcnfs	Ours
R3SAT	300	In	44.3	8.9	8.6 (2%)	15.7	1.1	10.8
	350	Out	215.7	43.3	42.7 (1%)	59.5	6.1	15.8
	400	Out	1,103.5	226.9	223.5 (2%)	289.1	30.8	36.5
	450	Out	5,207.9	989.7	994.9	1591.1	168.9	173.4
sgen	65	In	158.2	162.3	132.2 (23%)	8.5	2.1	8.0
	75	Out	1,799.3	1,792.3	1,594.0 (12%)	26.6	23.1	26.1
	85	Out	8,932.9	8,874.8	8,156.2 (9%)	115.0	114.4	105.6(8%)
	95	Out	98,534.0	97,979.7	92,407.9 (6%)	1214.7	1272.2	1,178.7 (8%)
satfc		In	631.3	250.1	67.1 (372%)	24.8	8.3	6.5 (128%)

Table 1: Mean tree size (1000s of nodes) and running time (CPU+GPU seconds) comparing Knuth Synthesis (Ours), kcnfs, and uni+kcnfs over 100 test instances from each distribution. Reductions in decisions and running time are relative to kcnfs. In/Out denotes whether the benchmark was in or out of the training distribution.

distribution; it far surpassed random branching on top-level decisions ($4\text{--}5\times$ reduction in tree size and walltime) and it was unclear whether it could be improved upon. Despite the strength of kcnfs, we squeezed out performance improvements of 1-2% in average tree size over kcnfs on up to 400 variables. Given the overhead of our neural network calls, these reductions in tree size did not lead to improvements in running time. For sgen, kcnfs was also very strong (see Appendix B) however there is greater scope for improvement since it was not optimized for sgen. We reduced average tree size over kcnfs on our training distribution (65 variables) by $1.23\times$. Our model trained on the 65-variable distribution generalized well to larger problem sizes; it reduced tree size even on 95 variables, which took ≈ 20 minutes to solve ($700\times$ more difficult). Our solver incurred a roughly constant overhead that prevented us from improving running time on 65 variables and 75 variables. We were able to improve the running time over kcnfs by 8% on 85 variables and 8% (≈ 1.5 minutes faster) on 95 variables. Even without using a GPU for model queries, we improved running time by 8% at 95 variables (constant policy-query overhead is swamped by tree-search time at higher running times). For satfc, evaluating against kcnfs allowed us to evaluate the (more realistic) scenario where there was no strong hand-tailored baseline. Such settings offer the likelihood of large scope for branching policy improvements; we saw the question of whether Knuth Synthesis discovered such policies as an important test. kcnfs ran $\approx 3\times$ faster than uni+kcnfs. We reduced tree size by $3.72\times$ and reduced running time by $1.28\times$ over kcnfs. Find expanded results in Appendix B including an experiment that shows the infeasibility of an existing RL approach on our datasets.

7.1 Knuth Speed Up

We evaluated how much quicker we could find good policies using Knuth samples. Each d -bounded Knuth sample is in expectation 2^d cheaper than evaluating the full proof

tree but the variance of the estimates can be high for unbalanced proof trees. We hoped variance would be sufficiently small that we would need much fewer than 2^d samples to distinguish between good and bad policies, allowing us to more quickly move to better parts of the policy space. Figure 4 shows average tree size over time comparing Knuth Synthesis to full-tree evaluation for our three training benchmarks. We normalized tree size across instances by the per-instance minimum and maximum tree size ever observed. In all cases, there was a clear efficiency improvement using Knuth samples. On R3SAT and sgen, Knuth sampling reached a given tree size approximately an order of magnitude more quickly. The improvement on satfc was substantial but not as pronounced; we hypothesized this was because the satfc proof trees tended to be less balanced, which resulted in higher-variance Knuth estimates.

8 Conclusions and Future Work

We presented MCFS, a class of RL algorithms for learning policies in tree MDPs. We introduced an MCFS implementation called Knuth Synthesis that approximates tree size with Knuth samples to avoid the prohibitive costs of exact policy evaluation that existing RL approaches suffer from. We matched or improved performance over a strong baseline in a diverse trio of distributions, tackling problems within these distributions that were two orders of magnitude more challenging than those in previous studies. In future work, we first 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. This information sharing means there is no straightforward way to encode the problem as a tree MDP. Approximating the size of a CDCL search tree with Knuth samples is non-trivial: unlike DPLL, single paths from the root to leaves cannot be evaluated independently to approximate tree size. The only obvious way to apply MCTS for CDCL policies is for a path to represent a full exploration through the proof tree, which would make the search space exponentially larger. Next, we would like to scale MCFS to industrial-size problems with millions of variables. There are two main bottlenecks to achieving this: (1) GPU memory constraints and (2) efficiently searching a much larger action space.

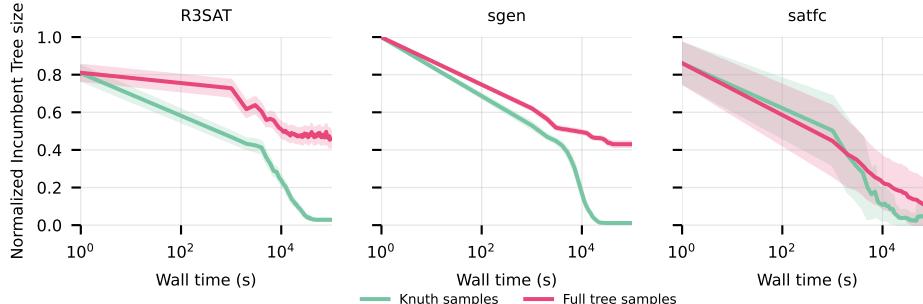


Fig. 4: Convergence rate improvements with Knuth samples. We normalize tree size across instances by the per-instance minimum and maximum tree size ever observed. Shaded regions represent 95% confidence intervals.

Memory requirements scale linearly with problem size and our resources restrict us from training beyond 10,000 variables; we would like to explore model parallelism, memory-efficient gradient approximations, and smaller state representations. We also would like to reduce the action space through learning an action representation and collapsing similar actions together. Finally, we would like to explore MCFS for satisfiable instances, which breaks our tree MDP property. Finding a small proof tree may be complementary to finding a SAT solution; if we can quickly prove that most of the search space does not have a solution, we can search for a solution in a more restricted space. We are interested in exploring a hybrid idea that simultaneously looks for SAT solutions at the same time being rewarded for shrinking the search space.

Acknowledgments

We thank the reviewers for all the constructive feedback. This work was funded by an NSERC Discovery Grant, a DND/NSERC Discovery Grant Supplement, a CIFAR Canada AI Research Chair (Alberta Machine Intelligence Institute), a Compute Canada RAC Allocation, awards from Facebook Research and Amazon Research, and DARPA award FA8750-19-2-0222, CFDA #12.910 (Air Force Research Laboratory). We thank Greg d’Eon for many useful conversations, especially one infamous AlphaGo tutorial.

Bibliography

- [1] SAT competition. The International SAT Competition Web Page, <http://www.satcompetition.org/> (2002–2024)
- [2] Abe, K., Xu, Z., Sato, I., Sugiyama, M.: Solving NP-hard problems on graphs by reinforcement learning without domain knowledge. arXiv preprint **arXiv:1905.11623**, 1–24 (2019)
- [3] Agostinelli, F., McAleer, S., Shmakov, A., Baldi, P.: Solving the Rubik’s cube with deep reinforcement learning and search. *Nature Machine Intelligence* **1**(8), 356–363 (2019)
- [4] Balcan, M.: Data-driven algorithm design. CoRR **abs/2011.07177** (2020), URL <https://arxiv.org/abs/2011.07177>
- [5] Baudiš, P., loup Gailly, J.: PACHI: State of the art open source Go program. In: Proceedings of the 13th International Conference on Advances in Computer Games, pp. 24–38, ACG ’11 (2012)
- [6] Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: a methodological tour d’horizon. *European Journal of Operational Research* **290**(2), 405–421 (2021)
- [7] Berre, D.L., Roussel, O., Simon, L.: SAT 2007 competition. The International SAT Competition Web Page, <http://www.satcompetition.org/> (2007)
- [8] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfschagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* **4**(1), 1–43 (2012)
- [9] Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: An abstraction-based decision procedure for bit-vector arithmetic. *International Journal on Software Tools for Technology Transfer* **11**(2), 95–104 (2009)
- [10] Cappart, Q., Chételat, D., Khalil, E.B., Lodi, A., Morris, C., Veličković, P.: Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research* **24**(130), 1–61 (2023)
- [11] Chen, P.C.: Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM Journal on Computing* **21**(2), 295–315 (1992)
- [12] Cornuéjols, G., Karamanov, M., Li, Y.: Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing* **18**(1), 86–96 (2006)
- [13] Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: Proceedings of the 5th International Conference on Computers and Games, pp. 72–83, CG ’06 (2006)
- [14] Coulom, R.: Computing “Elo ratings” of move patterns in the game of Go. *ICGA Journal* **30**(4), 198–208 (2007)
- [15] Crawford, J.M., Auton, L.D.: Experimental results on the crossover point in random 3SAT. *Artificial Intelligence* **81**, 31–57 (1996)
- [16] Czech, J., Korus, P., Kersting, K.: Monte-Carlo graph search for AlphaZero. arXiv preprint **arXiv:2012.11045**, 1–11 (2020)

- [17] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. *Communications of the ACM* **5**(7), 394–397 (1962)
- [18] Dequen, G., Dubois, O.: *kcnfs*: An efficient solver for random k -SAT formulae. In: *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, pp. 486–501, SAT ’06 (2003)
- [19] Duan, H., Nejati, S., Trimponias, G., Poupart, P., Ganesh, V.: Online Bayesian moment matching based SAT solver heuristics. In: *Proceedings of the 37th International Conference on Machine Learning*, pp. 2710–2719, ICML ’20 (2020)
- [20] Eén, N., Sörensson, N.: An extensible SAT-solver. In: *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, pp. 502–518, SAT ’03 (2003)
- [21] Etheve, M., Alès, Z., Bissuel, C., Juan, O., Kedad-Sidhoum, S.: Reinforcement learning for variable selection in a branch and bound algorithm. In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings*, pp. 176–185, Springer (2020)
- [22] Fréchette, A., Newman, N., Leyton-Brown, K.: Solving the station repacking problem. In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pp. 702–709, AAAI ’16 (2016)
- [23] Gasse, M., Chételat, D., Ferroni, N., Charlin, L., Lodi, A.: Exact combinatorial optimization with graph convolutional neural networks. In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pp. 15580–15592, NeurIPS ’19 (2019)
- [24] Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 1024–1034, NeurIPS ’17 (2017)
- [25] Hartford, J.S., Graham, D.R., Leyton-Brown, K., Ravanbakhsh, S.: Deep models of interactions across sets. In: *Proceedings of the 35th International Conference on Machine Learning, ICML ’18*, vol. 80, pp. 1914–1923 (2018)
- [26] Heule, M., van Maaren, H.: Look-ahead based sat solvers. *Handbook of satisfiability* **185**, 155–184 (2009)
- [27] Heule, M., van Maaren, H.: march_hi. In: *SAT Competition 2009: Solver and Benchmark Descriptions*, pp. 27–28, SAT ’09 (2009)
- [28] Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding cdcl sat solvers by lookaheads. In: *Haifa Verification Conference*, pp. 50–65, Springer (2011)
- [29] Hoos, H.H.: Automated algorithm configuration and parameter tuning. In: *Autonomous search*, pp. 37–71, Springer (2012)
- [30] Hottung, A., Tanaka, S., Tierney, K.: Deep learning assisted heuristic tree search for the container pre-marshalling problem. *Computers & Operations Research* **113**, 104781 (2020)
- [31] Huang, J., Darwiche, A.: A structure-based variable ordering heuristic for SAT. In: *Proceedings of the 18th International Joint Conference on Artificial intelligence*, pp. 1167–1172, IJCAI ’03 (2003)
- [32] Jeroslow, R.G., Wang, J.: Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence* **1**(1-4), 167–187 (1990)

- [33] Kearns, M., Mansour, Y., Ng, A.Y.: A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning* **49**(2), 193–208 (2002)
- [34] Keszocze, O., Schmitz, K., Schloeter, J., Drechsler, R.: Improving SAT solving using Monte Carlo tree search-based clause learning. In: *Advanced Boolean Techniques*, pp. 107–133, Springer (2020)
- [35] Khalil, E., Le Bodic, P., Song, L., Nemhauser, G., Dilkina, B.: Learning to branch in mixed integer programming. In: *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30 (2016)
- [36] Khalil, E.B., Vaezipoor, P., Dilkina, B.: Finding backdoors to integer programs: A Monte Carlo tree search framework. In: *Proceedings of the 36th AAAI Conference on Artificial Intelligence*, pp. 1–10, AAAI ’22 (2022)
- [37] Kilby, P., Slaney, J., Thiébaux, S., Walsh, T.: Estimating search tree size. In: *Proceedings of the 21st National Conference of Artificial Intelligence*, pp. 1014–1019, AAAI ’06 (2006)
- [38] Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: *Proceedings of the 3rd International Conference on Learning Representations*, pp. 1–15, ICLR ’14 (2014)
- [39] Knuth, D.E.: Estimating the efficiency of backtrack programs. *Mathematics of Computation* **29**(129), 122–136 (1975)
- [40] Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: *Proceedings of the 17th European Conference on Machine Learning*, pp. 282–293, ECML ’06 (2006)
- [41] Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. *Data mining and constraint programming: Foundations of a cross-disciplinary approach* pp. 149–190 (2016)
- [42] Kurin, V., Godil, S., Whiteson, S., Catanzaro, B.: 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*, pp. 9608–9621, NeurIPS ’20 (2019)
- [43] Lagoudakis, M.G., Littman, M.L.: Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics* **9**, 344–359 (2001)
- [44] Lauria, M., Elffers, J., Nordström, J., Vinyals, M.: Cnfgen: A generator of crafted benchmarks. In: Gaspers, S., Walsh, T. (eds.) *SAT*, Lecture Notes in Computer Science, vol. 10491, pp. 464–473, Springer (2017), ISBN 978-3-319-66263-3, URL <http://dblp.uni-trier.de/db/conf/sat/sat2017.html#LauriaENV17>
- [45] Lederman, G., Rabe, M., Lee, E.A., Seshia, S.A.: Learning heuristics for quantified boolean formulas through reinforcement learning. In: *Proceedings of the 8th International Conference on Learning Representations*, pp. 1–18, ICLR ’19 (2019)
- [46] Liberatore, P.: On the complexity of choosing the branching literal in dpll. *Artificial intelligence* **116**(1-2), 315–326 (2000)
- [47] Liberatore, P.: Complexity results on dpll and resolution. *ACM Transactions on Computational Logic (TOCL)* **7**(1), 84–107 (2006)
- [48] Lobjois, L., Lemaître, M.: Branch and bound algorithm selection by performance prediction. In: *Proceedings of the 15th National/10th Conference on Artificial Intelligence*, pp. 353–358, AAAI ’98 (1998)

- [49] Loth, M., Sebag, M., Hamadi, Y., Schoenauer, M.: Bandit-based search for constraint programming. In: Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, pp. 464–480, CP ’13 (2013)
- [50] Maddison, C.J., Huang, A., Sutskever, I., Silver, D.: Move evaluation in Go using deep convolutional neural networks. arXiv preprint [arXiv:1412.6564](#), 1–8 (2014)
- [51] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Annual Design Automation Conference, pp. 530–535, DAC ’01 (2001)
- [52] Nair, V., Bartunov, S., Gimeno, F., von Glehn, I., Lichocki, P., Lobov, I., O’Donoghue, B., Sonnerat, N., Tjandraatmadja, C., Wang, P., Addanki, R., Hapuarachchi, T., Keck, T., Keeling, J., Kohli, P., Ktena, I., Li, Y., Vinyals, O., Zwols, Y.: Solving mixed integer programs using neural networks. arXiv preprint [arXiv:2012.13349](#), 1–57 (2020)
- [53] Nejati, S., Frioux, L.L., Ganesh, V.: A machine learning based splitting heuristic for divide-and-conquer solvers. In: Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming, pp. 899–916, CP ’20 (2020)
- [54] Parsonson, C.W., Laterre, A., Barrett, T.D.: Reinforcement learning for branch-and-bound optimisation using retrospective trajectories. arXiv preprint [arXiv:2205.14345](#) (2022)
- [55] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems, pp. 8024–8035, NeurIPS ’19 (2019)
- [56] Previti, A., Ramanujan, R., Schaefer, M., Selman, B.: Monte-Carlo style UCT search for boolean satisfiability. In: Proceedings of the 12th Congress of the Italian Association for Artificial Intelligence, pp. 177–188, AI*IA ’11 (2011)
- [57] Purdom, P.W.: Tree size by partial backtracking. SIAM Journal on Computing **7**(4), 481–491 (1978)
- [58] Rosin, C.D.: Multi-armed bandits with episode context. Annals of Mathematics and Artificial Intelligence **61**(3), 203–230 (2011)
- [59] Roussel, O.: Controlling a solver execution with the *runsolver* tool. Journal on Satisfiability, Boolean Modeling and Computation **7**(4), 139–144 (2011)
- [60] Ryvchin, V., Nadel, A.: Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking. In: Proceedings of SAT Competition 2018 — Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, vol. B-2018-1, p. 29 (2018)
- [61] Scavuzzo, L., Chen, F., Chételat, D., Gasse, M., Lodi, A., Yorke-Smith, N., Aardal, K.: Learning to branch with tree mdps. Advances in Neural Information Processing Systems **35**, 18514–18526 (2022)
- [62] Schloeter, J.: A Monte Carlo tree search based conflict-driven clause learning SAT solver. In: Proceedings of the 2017 INFORMATIK Conference, pp. 2549–2560, INFORMATIK ’17 (2017)
- [63] Schrittwieser, J., Antonoglou, I., Hubert, T., et al.: Mastering Atari, Go, chess and shogi by planning with a learned model. Nature **588**(7839), 604–609 (2020)

- [64] Selsam, D., Bjørner, N.: Guiding high-performance SAT solvers with unsat-core predictions. In: Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing, pp. 336–353, SAT ’19 (2019)
- [65] Silver, D., Huang, A., Maddison, C.J., et al.: Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489 (2016)
- [66] Silver, D., Hubert, T., Schrittwieser, J., et al.: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* **362**(6419), 1140–1144 (2018)
- [67] Silver, D., Schrittwieser, J., Simonyan, K., et al.: Mastering the game of Go without human knowledge. *Nature* **550**, 354–359 (2017)
- [68] Song, W., Cao, Z., Zhang, J., Xu, C., Lim, A.: Learning variable ordering heuristics for solving constraint satisfaction problems. *Engineering Applications of Artificial Intelligence* **109**, 104603 (2022)
- [69] Spence, I.: sgen1: A generator of small but difficult satisfiability benchmarks. *ACM Journal of Experimental Algorithms* **15**, 1.1–1.15 (2010)
- [70] Suelflow, A., Fey, G., Bloem, R., Drechsler, R.: Using unsatisfiable cores to debug multiple design errors. In: Proceedings of the 18th ACM Great Lakes Symposium on VLSI, pp. 77–82, GLSVLSI ’08 (2008)
- [71] Sutskever, I., Nair, V.: Mimicking Go experts with convolutional neural networks. In: Proceedings of the 18th International Conference on Artificial Neural Networks, pp. 101–110, ICANN ’08 (2008)
- [72] Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
- [73] Tönshoff, J., Kisim, B., Lindner, J., Grohe, M.: One model, any CSP: Graph neural networks as fast global search heuristics for constraint satisfaction. arXiv preprint [arXiv:2208.10227](https://arxiv.org/abs/2208.10227), 1–23 (2022)
- [74] Vaezipoor, P., Lederman, G., Wu, Y., Maddison, C., Grosse, R.B., Seshia, S.A., Bacchus, F.: Learning branching heuristics for propositional model counting. In: Proceedings of the 35th AAAI Conference on Artificial Intelligence, pp. 12427–12435, AAAI ’21 (2021)
- [75] Wang, W., Hu, Y., Tiwari, M., Khurshid, S., McMillan, K., Miikkulainen, R.: Neurocomb: Improving sat solving with graph neural networks (2022)
- [76] Wattez, H., Koriche, F., Lecoutre, C., Paparrizou, A., Tabary, S.: Learning variable ordering heuristics with multi-armed bandits and restarts. In: Proceedings of the 24th European Conference on Artificial Intelligence, pp. 1–8, ECAI ’20 (2020)
- [77] Yolcu, E., Póczos, B.: Learning local search heuristics for boolean satisfiability. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems, pp. 7992–8003, NeurIPS ’19 (2019)
- [78] Zook, A., Harrison, B., Riedl, M.O.: Monte-Carlo tree search for simulation-based strategy analysis. arXiv preprint [arXiv:1908.01423](https://arxiv.org/abs/1908.01423), 1–9 (2019)

The appendix is divided into three sections: Pseudocode, Expanded Results, and Benchmark Details.

A Pseudocode

A.1 Knuth Synthesis

We now provide pseudocode for Knuth Synthesis' implementation of the three pieces of MCFS: tree policy α (Algorithm 3), step policy γ (Algorithm 4), and rollout policy π (Algorithm 5). Parameters/inputs specific to the Knuth Synthesis implementation are declared in the first line of each algorithm.

Algorithm 3: $\alpha(s, c, v)$

Input: c_{PUCT} , depth calibration Q_d , prior p
 $\forall a, u_a = c_{PUCT} p_a \frac{\sqrt{\sum_{a'} c_{a'}}}{1+c_a}$
return $\arg \min_{a'} (v_{a'} - Q_d u_{a'})$

Algorithm 4: $\gamma(\theta, a, s)$

Input: rollout depth ℓ
if Depth(s, θ) $> \ell$ or $l(s) = 0$ **then**
 return \emptyset
end if
 $s^{sub} \leftarrow sibling(s)$
/ Case 1: If sibling processed, take opposite */*
if processed(θ, s^{sub}) **then**
 $a' \leftarrow \gamma_{cache}(s^{sub})$
 if $a' = \emptyset$ **then**
 step $\leftarrow a$
 else
 step $\leftarrow \emptyset$
 end if
else
/ Case 2: Flip a coin by Knuth */*
 if Random draw with $p=0.5$ is True **then**
 step $\leftarrow a$
 else
 step $\leftarrow \emptyset$
 end if
end if
return step

Algorithm 5: $\pi(s)$

Input: running multiplicative error ϵ , accuracy threshold t , sub-solver calls n , value net V , subsolver policy ϕ_{sub}

$$p \leftarrow 1 - (\epsilon/t)$$

if Random draw with prob p **then**

- return $V(s)$

else

- $m \leftarrow |\log(T_{\phi_{sub}}(s)) - \log(V(s))|$ /* multiplicative error */
- $\epsilon \leftarrow \epsilon + \frac{(m-\epsilon)}{n+1}$ /* $O(1)$ update */
- $n \leftarrow n + 1$
- return $T_{\phi_{sub}}(s)$

end if

A.2 DPLL

The DPLL algorithm (Algorithm 6) takes in some branching policy ϕ , which outputs a variable given the SAT instance. DPLL then recursively calls itself with assigning that variable to true and false. Given a variable assignment s , we call the transition function τ_{DPLL} to find the new simplified S . τ_{DPLL} makes calls to two subroutines.

Unit Propagation Unit propagation finds all unit clauses i.e., 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 because those can be trivially satisfied.

A.2.1 Tree MDP definition for DPLL UNSAT See Definition 2 for the definition of a tree MDP. An action a is a choice of variable to branch on. The reward function is $r(s_i) = -1 \forall i$, as we are trying to minimize the number of nodes. A leaf node i.e., $l(s_i) = 1$ is any state s_i where DPLL returns *False* or *True*. $s_{ch_i}^L$ will be $s \wedge (a_i = 0)$ and $s_{ch_i}^R$ will be $s \wedge (a_i = 1)$. Note that for satisfiable instances, this last property does not necessarily hold. We may only need to transition to $s_{ch_i}^L$ or $s_{ch_i}^R$ to solve the problem, if either of the subproblems is SAT. It is not correct to model SAT (as opposed to UNSAT) with tree MDPs.

B Expanded Results

B.1 Comparisons to past work

Past work has never scaled beyond 1000 decisions during training episodes and have no mechanism for approximating policy evaluations, therefore we hypothesized that these

Algorithm 6: DPLL

```

Input: SAT instance  $S$ , policy  $\phi$ , assignment  $x_i$ 
 $\tau_{DPLL}(S, x_i)$ 
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  $DPLL(S, \phi, v = 0)$  OR  $DPLL(S, \phi, v = 1)$ 

```

Algorithm 7: τ_{DPLL}

```

Input: SAT instance  $S$ ,
        assignment  $x_i$ 
 $S \leftarrow S \wedge x_i$ 
 $S \leftarrow \text{UnitPropagation}(S)$ 
 $S \leftarrow \text{PureLiteralAssign}(S)$ 
return  $S$ 

```

methods would be infeasible at scale. To test this, we evaluated our approach against Graph-Q-SAT [42], the RL-for-SAT approach that is closest to our own. We could not find code to compare to Lederman et al. [45] and we could not get Vaezipoor et al. [74]’s code working, however we think Graph-Q-SAT performance is a good indicator of the success of other existing approaches. Vaezipoor et al. [74]’s approach is a black-box rather than RL method but also relies on exact policy evaluation. Kurin et al. [42] trained Graph-Q-SAT on 50-variable R3SAT problems. We evaluated whether we could train Graph-Q-SAT on larger-sized problems from the same distribution up to 250 variables, restricting to unsatisfiable instances. For each size at increments of 50 variables, we trained a Graph-Q-SAT model for five days with a decisions cap of 100,000 over a training set of 1000 instances. We evaluated validation performance after every 1000 training episodes and evaluated the model with best validation performance. Graph-Q-SAT is integrated with `minisat` [20]; for a fair comparison, we trained our method using the same `minisat` as our rollout solver and all other parameters left unchanged. We could not easily benchmark Graph-Q-SAT with `kcnfs` (which is orders of magnitude faster), since they are tightly integrated with `minisat` with a customized SWIG interface bridging C++ and python. We reported number of decisions (no runtimes) since Graph-Q-SAT calls out to `minisat` in a very inefficient way that makes their method appear artificially slower.

Figure 5 shows performance on a held-out test set for each problem size. Performance is measured as the multiplicative reduction factor in tree size over the `minisat` baseline. Knuth Synthesis is better at every problem size including making over $2\times$ fewer decisions at 100 variables. Graph-Q-SAT still performs quite well. It improved over `minisat` by $2.5\times$ at 50 variables and by $1.5\times$ on 100-250 variables. We were confused how Graph-Q-SAT could make learning progress when these numbers of decisions were required

(100,000 at 250 variables). We then evaluated a *random* baseline that uniformly-random samples as many parameter settings as there are Graph-Q-SAT validation evaluations and chooses the model with best validation performance. We find that this baseline had virtually identical performance to Graph-Q-SAT between 150 and 250 variables, making it a possibility that the Graph-Q-SAT improvements at these sizes are occurring for similar reasons. We think the success of the random baseline is also independently interesting. It suggests that at least for this dataset, there are large areas in parameter space which contain good branching policies. This would be very surprising in vision for example. We would expect it would take an enormous number of random samples to, for example, build a better-than-random image classifier. We hypothesize that GNN has a helpful inductive bias. Regardless of the particular parameter setting, similar variables will be mapped to similar values in a GNN. That means that similar variables will tend to be branched in succession since the highest scoring variables are likely to be similar to one another. This may be a good inductive bias as similar variables may be more likely to induce early conflicts if branched on in short succession.

We note that the multiplicative improvement of Knuth Synthesis over `minisat` decreases with the number of variables. We suspect this is because the number of policy decisions is being held constant while the number of decisions needed to solve the problem is increasing exponentially.

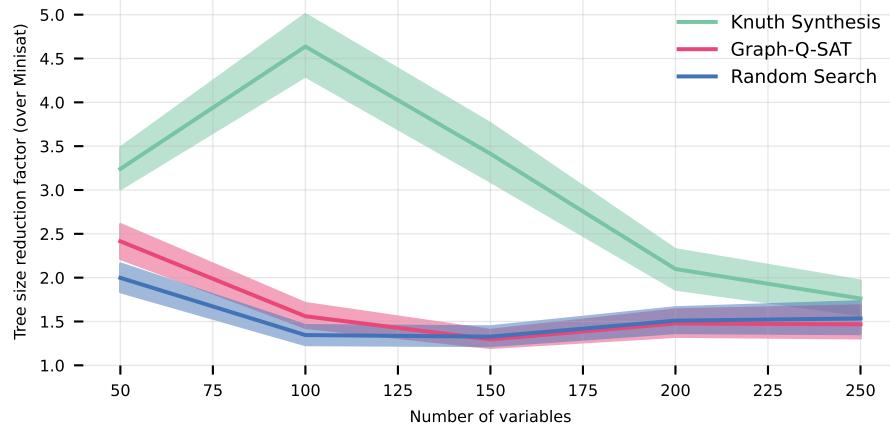


Fig. 5: Graph-Q-SAT vs. Knuth synthesis on unsatisfiable R3SAT problems with increasing size from 50 to 250 variables at 50 variable increments. Performance is measured as the multiplicative factor of improvement over `minisat`.

We also ran a similar experiment evaluating on our `sgen-55` and `satfc-easy+satfc-hard` datasets. For `satfc` there were a few especially difficult instances which caused extremely long training episodes in Graph-Q-SAT, which dominated the runtime. We removed those instances in our evaluation and reran our training runs. See Table 2 for the results. In this case, Graph-Q-SAT showed moderate improvements while

Dataset	Size	minisat	GraphQSAT (1000s)	Ours (1000s)
R3SAT	50	67	28	21
	100	619	397	134
	150	4144	3198	1214
	200	26853	18178	12794
	250	167993	114456	95318
s _{gen}	55	127374	127943	182,241
s _{atfc-easy+s_{atfc-hard}}		34410	32626	41569

Table 2: Improvement in tree size over `minisat` comparing Graph-Q-SAT to KnuthSynthesis.

Knuth Synthesis struggled. We think this is very likely because the subsolver in this case (`minisat`) is a CDCL solver which breaks our tree MDP assumption (i.e., subproblems are independent). Knuth Synthesis prevents `minisat` from sharing clauses across subproblems and forces the VSIDS heuristic to cold start for every subproblem. In the case of `sgen`, we could not even improve upon `minisat` on a given instance during offline training. This phenomenon did not happen in the R3SAT dataset likely because CDCL performs very poorly in that setting and there is little value for sharing clauses across subproblems. We also note that the success of cube-and-conquer algorithms shows that is not always detrimental to solve subproblems independently with CDCL [28].

B.2 CDFs of complete results for tree size

See Figure 6.

B.3 Demonstrating `kcnfs07` is a strong baseline

`kcnfs07` is a strong baseline for R3SAT and `sgen`, however CDCL solvers are much faster on `satfc`. It is well known that state-of-the-art industrial solvers (CDCL solvers with VSIDS-style branching heuristic) perform very poorly on R3SAT. `hkis` was the fastest solver on unsatisfiable instances at the 2021 SAT competition [7] and `kcnfs07` was $>100\times$ faster on R3SAT. Such strong industrial solvers also performed poorly on `sgen` (`kcnfs07` $>3\times$ faster). See Table 3 for complete results.

To better contextualize our results, we also evaluated the strong general-purpose DPLL heuristic Jeroslav-Wang (JW) [32] for top-level decisions in the same way we benchmarked our method (i.e., JW+`kcnfs`). `kcnfs` was substantially faster in most cases; see Table 4 for complete results.

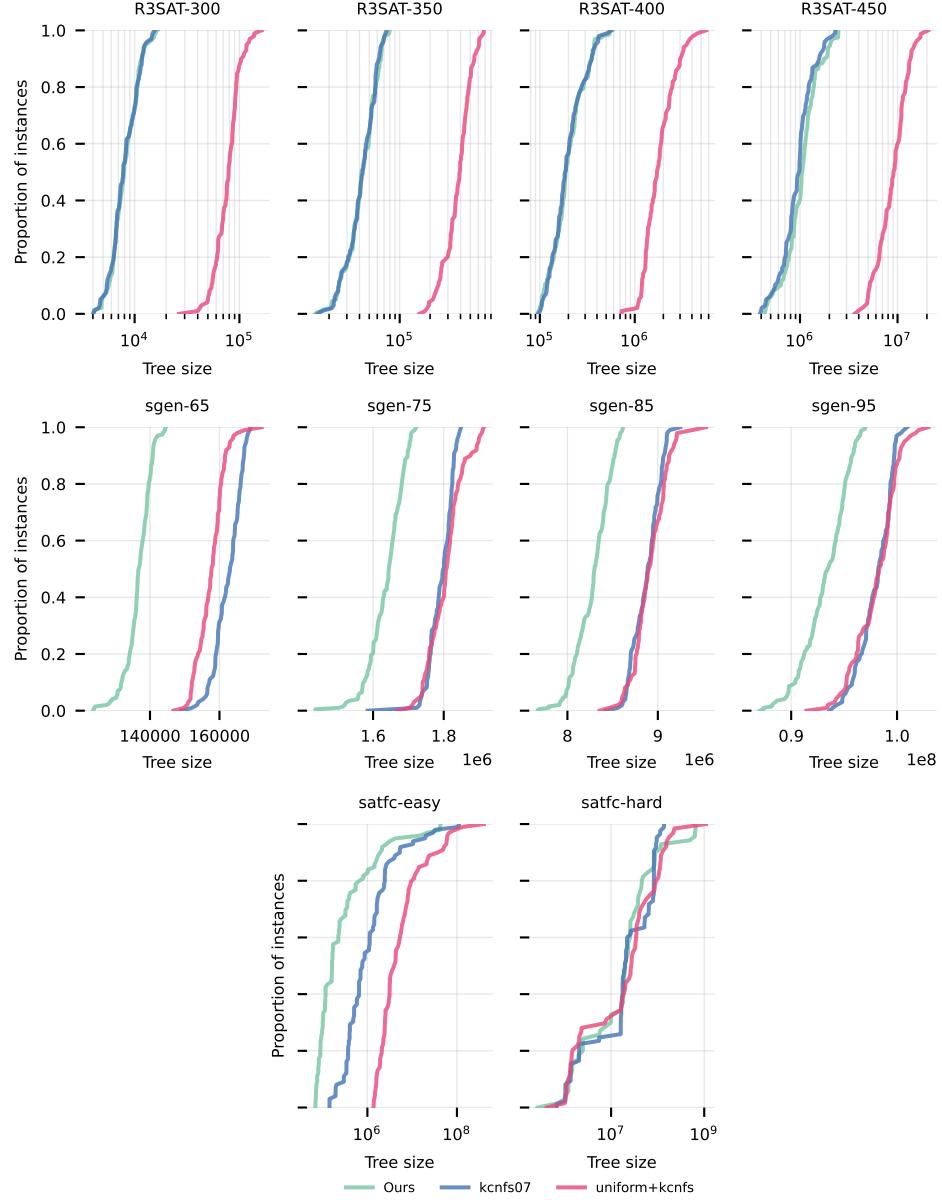


Fig. 6: Tree size CDFs for the results in Table 1. Comparison of Knuth Synthesis (Ours), kcnfs07, and uni+kcnfs on our 10 benchmarks.

Distribution	Tree size (1000s)		CPU time (s)	
	hkis	kcnfs07	hkis	kcnfs07
R3SAT-300	1,717.0	44.3	47.4	1.1
R3SAT-350	15,120.8	215.7	836.4	6.1
R3SAT-400	-	1,103.5	> 6 hours	30.8
R3SAT-450	-	5,207.9	> 6 hours	168.9
sgen-65	422.1	162.3	12.2	4.0
sgen-75	2,996.7	1,792.3	127.7	44.1
sgen-85	111,435.8	97,979.7	699.1	219.9
sgen-95	-	97,979.7	> 6 hours	1,178.6

Table 3: Mean tree size (1000s of nodes) and running time (CPU+GPU seconds) comparing `kcnfs` and `hKis`.

Distribution	JW+kcnfs	Tree size (1000s)	JW+kcnfs	CPU time (s)	Slower than kcnfs
R3SAT-300		16.9		4.5	3.7×
R3SAT-350		87.6		14.5	2.4×
R3SAT-400		409.8		62.9	2.0×
R3SAT-450		1,977.6		336.0	2.0×
sgen-65		199.7		5.5	2.6×
sgen-75		2,223.7		29.90	1.3×
sgen-85		10,915.0		134.7	1.2×
sgen-95		119,200.8		1433.3	1.1×

Table 4: Mean tree size (1000s of nodes) and running time (CPU+GPU seconds) for JW+kcnfs.

B.4 Challenging out-of-distribution generalization on `satfc`

We evaluated generalization of our `satfc` model on more challenging `satfc` instances (`satfc-hard`). These were instances where `kcnfs` took more than 100,000 decisions (as opposed to our training set where instances were solvable in <100,000 decisions). Bottom of Figure 6 illustrates how different the runtime distributions are between the easy and hard instances. Our model outperforms the `uni+kcnfs` baseline but could not match the performance of `kcnfs`. These results are almost entirely driven by the performance on a few especially hard instances as the CDFs show in Figure 6. See Table 5.

C Benchmark Details

For R3SAT, we followed Crawford and Auton [15] to estimate the location of the phase transition with a number of clauses-to-number of variables $n = 4.258 \cdot m + 58.26 \cdot m^{-2/3}$. We used the `CNFgen` [44] python package to generate.

Distribution	In/Out	Tree size (1000s) [Reduction]		CPU+GPU time (s) [Reduction]			
		uni+kcnfs	kcnfs07	Ours	uni+kcnfs	kcnfs07	Ours
satfc-hard	Out	5,914.3	3,201.8	3,538.7	259.2	147.7	166.3

Table 5: Mean tree size (1000s of nodes) and running time (CPU+GPU seconds) for hard `satfc` instances.

`sogen` [69] is a hand-crafted generator that is notoriously difficult for its size. An `sogen` 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 that generated instances that contain contradictions.

For `satfc` [22], we used this publicly available simulator of the 2016 FCC spectrum auction: <https://github.com/newmanne/SATFC/tree/development/simulator>. We ran 160 different simulator configurations and collected all of the station repacking instances. We solved each instance and filtered satisfiable instances. The FCC spectrum auction represented the airspace for the entirety of the USA. To create smaller problems that our architecture could handle, we built subsets of the USA station interference graph. For each subset graph, we started with all the stations from one of the following prominent american cities: NEW YORK, LOS ANGELES, CHICAGO, HOUSTON, PHILADELPHIA, PHOENIX, SAN ANTONIO, SAN DIEGO, DALLAS, SAN JOSE. We then added stations that were `Num links` hops away from our initial set of stations in the full interference graph. For each starting city, we created interference graphs for each setting of `Num links` from 1 to 4, leaving us with 40 interference graphs in total. For each graph, we ran four separate experiments with standard deviation of noise of the station valuation model to {0.01, 0.05, 0.1, 0.5}. Each auction simulation took at most one hour to complete.

For a given `City`, `Num links`, and `Noise`, we ran the following call string:

```
build/install/FCCSimulator/bin/FCCSimulator -CONFIG-FILE
SATFC/satfc/src/dist/bundles/ satfc_8.yaml -MAX-CHANNEL 29
-MAX-CHANNEL-FINAL 36
-VOLUMES-FILE src/dist/simulator_data/volumes.csv -POP-VALUES true
-START-CITY [City] -CITY-LINKS [Num links] -CNF-DIR ./cnfs
-UHF-ONLY true -INCLUDE-VHF false -NOISE-STD [Noise]
```

We solved each unsatisfiable instance with `minisat` [20] and recorded the number of decisions needed to solve each instance. `satfc` was each instance that took less than 100,000 decisions. There were some instances that `kcnfs07` immediately crashed on for unknown reasons for which we removed.