UC Berkeley
UC Berkeley Previously Published Works

Title
Learning Branching Heuristics for Propositional Model Counting

Permalink
https://escholarship.org/uc/item/991301nd

Authors
Vaezipoor, Pashootan
Lederman, Gil
Wu, Yuhuai
et al.

Publication Date
2020-07-06

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

arXiv:2007.03204v1 [cs.LG] 7 Jul 2020

Learning Branching Heuristics for Propositional Model Counting

Pashootan Vaezipoor[1,*], Gil Lederman[2,*], Yuhuai Wu[1,3], Chris J. Maddison[1,3], Roger Grosse[1,3], Edward Lee[2], Sanjit A. Seshia[2], Fahiem Bacchus[1]

[1]University of Toronto, [2]UC Berkeley, [3]Vector Institute

Abstract

Propositional model counting or #SAT is the problem of computing the number of satisfying assignments of a Boolean formula and many discrete probabilistic inference problems can be translated into a model counting problem to be solved by #SAT solvers. Generic "exact" #SAT solvers, however, are often not scalable to industrial-level instances. In this paper, we present `Neuro#`, an approach for learning branching heuristics for exact #SAT solvers via evolution strategies (ES) to reduce the number of branching steps the solver takes to solve an instance. We experimentally show that our approach not only reduces the step count on similarly distributed held-out instances but it also generalizes to much larger instances from the same problem family. The gap between the learned and the vanilla solver on larger instances is sometimes so wide that the learned solver can even overcome the run time overhead of querying the model and beat the vanilla in wall-clock time by orders of magnitude.

1 Introduction

Propositional model counting is the problem of counting the number of satisfying solutions to a Boolean formula. When the Boolean formula is expressed in conjunctive normal form (CNF), this problem is known as the #SAT problem [15]. #SAT is a #P-complete problem, and by Toda's theorem [38] any problem in the polynomial-time hierarchy (PH) can be solved by a polynomial number of calls to a #SAT oracle. This means that effective #SAT solvers, if they could be developed, have the potential to help solve problems whose complexity lies beyond NP, from a range of applications. The tremendous practical successes achieved by encoding problems to SAT and using modern SAT solvers [24] demonstrates the benefits of such an approach.

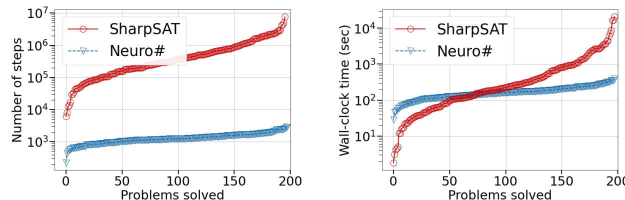


Figure 1: Cactus plots comparing `Neuro#` to `SharpSAT` on the grid_wrld$(10, 12)$ benchmark. For any point t on the y axis, the plot shows the number of benchmark problems that are individually solvable by the solver, within t steps (left) and seconds (right).

[*] Equal contribution (correspondence to <pashootan@cs.toronto.edu>).

Preprint. Under review.

Modern exact #SAT solvers are based on the DPLL [10, 11] algorithm and have been successfully applied to solve certain problems, e.g., inference in Bayes Nets [4, 23, 31, 13] and bounded-length probabilistic planning [12]; however, many applications remain out of reach of current solvers. For example, in problems such as inference in Markov Chains, which have a temporal structure, exact model counters are still generally inferior to earlier methods such as Binary Decision Diagrams (BDDs). In this paper we show that machine learning methods can be used to greatly enhance the performance of #SAT solvers, potentially making a wider range of applications feasible.

In particular, we learn the branching heuristic of the state of the art DPLL-based #SAT solver: `SharpSAT`. We cast the problem as a Markov Decision Process (MDP) in which the agent has to select the best literal for `SharpSAT` to branch on next. We use a Graph Neural Network (GNN) [33] to represent the part of the input formula the solver is currently working on. The model is trained end-to-end using an Evolution Strategies algorithm, with the objective of minimizing the mean number of branching decisions required to solve instances from a given distribution of problems. We call this augmented solver `Neuro#`.

We found that `Neuro#` can generalize to unseen problem instances from the same distribution as well as to instances that were much larger than those trained on. Furthermore, despite the run time overhead of querying the model, that `Neuro#` has to overcome, on some problem domains our approach achieved *orders of magnitude improvements* in the solver's *wall-clock* run time (Figure 1). This is quite remarkable in the context of prior related work [45, 35, 5, 14, 19, 17, 22], where using ML to improve combinatorial solvers had at best yielded modest wall-clock time improvements (less than than a factor of two) and positions this line of research as a viable path to improve the run time performance of exact model counters.

The rest of the paper is organized as follows: In Section 2 we provide some needed background and fix the terminology. We describe the learning approach in Section 3, and compare it to related work in Section 4. Section 5 details our dataset generation process that we later use in our experiments in Section 6. We conclude with a short discussion in Section 7.

2 Background

2.1 #SAT

A propositional Boolean formula consists of a set of propositional (true/false) variables composed by applying the standard operators "and" ($\wedge$), "or" ($\vee$) and "not" ($\neg$). A *literal* is any variable v or its negation $\neg v$. A *clause* is a disjunction of literals $\bigvee_{i=1}^{n} l_i$. A clause is said to be a *unit clause* if it contains only one literal. Finally, a Boolean formula is in *Conjunctive Normal Form* (CNF) if it is a conjunction of clauses. We denote the set of literals and clauses of a CNF formula ϕ by $\mathcal{L}(\phi)$ and $\mathcal{C}(\phi)$, respectively. We will assume that all formulas are in CNF.

A *truth assignment* for any formula ϕ is a mapping of its variables to $\{0, 1\}$ (**false/true**). Thus there are 2^n different truth assignments when ϕ has n variables. A truth assignment π satisfies a literal ℓ when ℓ is the variable v and $\pi(v) = 1$ or when $\ell = \neg v$ and $\pi(v) = 0$. It satisfies a clause when at least one of its literals is satisfied. A CNF formula ϕ is satisfied when all of its clauses are satisfied under π in which case we call π a *satisfying assignment* for ϕ.

The #SAT problem for ϕ is to compute the number of satisfying assignments. If ℓ is a unit clause of ϕ then all of ϕ's satisfying assignments must make ℓ true. If another clause $c' = \neg\ell \vee \ell'$ is in ϕ, then every satisfying assignment must also make ℓ' true since $\neg\ell \in c'$ must be false. This process of finding all literals whose truth value is forced by unit clauses is called *Unit Propagation* (UP) and is used in all SAT and #SAT solvers. Such solvers traverse the search tree by employing a branching heuristic. This heuristic selects an unforced literal and branches on it by setting it to, in turn, to **true** and **false**. When a literal ℓ is set to **true** the formula ϕ can be reduced by finding all forced literals using UP (this will include ℓ and its negation), removing all clauses containing a true literal, and finally removing all false literals from all clauses. The resulting formula is denoted by UP(ϕ, ℓ).

Two sets of clauses are called *disjoint* if they share no variables. A component $C \subset \mathcal{C}(\phi)$ is a subset of ϕ's clauses that is disjoint from its complement $\mathcal{C}(\phi) - C$. A formula ϕ can be efficiently broken up into a maximal number of disjoint components $C_1, \ldots, C_k$. Although most formulas initially consist of only one component, as variables are set by branching decisions and clauses are removed, the reduced formulas will often break up into multiple components. Components are important for

improving the efficiency of #SAT solving as each component can be solved separately and their counts multiplied: $\text{COUNT}(\phi) = \prod_{i=1}^{k} \text{COUNT}(C_i)$. In contrast, solving the formula as a monolith takes $2^{\Theta(n)}$ where n is the number of variables in the input formula, and so not efficient for large n.

A formula ϕ can be represented by a *literal-clause incidence graph* (LIG). This graph contains a node for every clause and a node for every literal of ϕ (i.e., v and $\neg v$ for every variable v). An edge connects a clause node n_c and a literal node n_ℓ if and only if $\ell \in c$. Figure 2 shows an example. Note that every component of ϕ forms a disconnected sub-graph of the LIG.

Both exact [37, 30, 26] and approximate [9, 25] model counters have been developed. In this paper, we focus on the former, using the state of the art exact model counter `SharpSAT` [37]. `SharpSAT` and other modern exact #SAT solvers are based on backtracking search DPLL [10, 11] augmented with *clause learning* and *component caching* [3, 2]. A simplified version of the algorithm with the clause learning parts omitted is given in Algorithm 1. A more detailed version along with more elaborate analysis is provided in Appendix A.

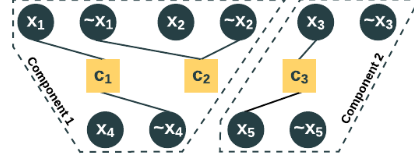


Figure 2: The Literal-Clause Incidence Graph of the formula: $(x_1 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_5)$.

The `#DPLLCache` algorithm works on one component at a time. If that component's model count has already been cached it returns the cached value. Otherwise it selects a literal to branch on (line 4) and computes the model count under each value of this literal by calling `CountSide()`. The sum of these two counts is the model count of the passed component ϕ, and so is stored in the cache (line 7). The `CountSide` function first unit propagates the input literal. If an empty clause is found, then the current formula ϕ_ℓ is unsatisfiable and has zero models. Otherwise, ϕ_ℓ is divided into its components which are independently solved. The product of sub-component model counts is returned. Critical to the performance of the algorithm is the choice of which literal from the current formula ϕ to branch on. This choice affects the efficiency of clause learning and the effectiveness of component generation and caching lookup success. `SharpSAT` uses the VSADS heuristic [32] which is a linear combination of a heuristic aimed at making clause learning effective (VSIDS) and a count of the number of times a variable appears in the current formula.

Algorithm 1 Component Caching DPLL

```
1:  function #DPLLCache(ϕ)
2:      if inCache(ϕ) then
3:          return cacheLookUp(ϕ)
4:      Pick a literal ℓ ∈ L(ϕ)
5:      #ℓ = CountSide(ϕ, ℓ)
6:      #¬ℓ = CountSide(ϕ, ¬ℓ)
7:      addToCache(ϕ, #ℓ + #¬ℓ)
8:      return #ℓ + #¬ℓ

9:  function CountSide(ϕ, ℓ)
10:     ϕ_ℓ = UP(ϕ, ℓ)
11:     if ϕ_ℓ contains an empty clause then
12:         return 0
13:     if ϕ_ℓ contains no clauses then
14:         k = # of unset variables
15:         return 2^k
16:     K = findComponents(ϕ_ℓ)
17:     return ∏_{κ∈K} #DPLLCache(κ)
```

2.2 Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks used for representation learning over graphs [16, 33]. Utilizing a neighbourhood aggregation (or message passing) scheme, GNNs map the nodes of the input graph to a vector space. Let $G = (V, E)$ be an undirected graph with node feature vectors $h_v^{(0)}$ for each node $v \in V$. GNNs use the graph structure and the node features to learn an embedding vector h_v for every node. This is done through iterative applications of a neighbourhood aggregation function. In each iteration k, the embedding of a node $h_v^{(k)}$ is updated by aggregating the embeddings of its neighbours from iteration $k-1$ and passing the result through a nonlinear aggregation function A parameterized by $W^{(k)}$:

$$h_v^{(k)} = \text{A}\Big(h_v^{(k-1)}, \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}; W^{(k)}\Big), \tag{1}$$

where $\mathcal{N}(v) = \{u | u \in V \wedge (v, u) \in E\}$. After K iterations, $h_v^{(K)}$ is extracted as the final node embedding h_v for node v. Through this scheme, v's node embedding at step k incorporates the structural information of all its k-hop neighbours.

2.3 Evolution Strategies

Evolution Strategies (ES) are a class of zeroth order black-box optimization algorithms [7, 42]. Inspired by natural evolution, a population of parameter vectors (genomes) is perturbed (mutated) at every iteration, giving birth to a new generation. The resulting offspring are then evaluated by a predefined fitness function. Those offspring with higher fitness score will be selected for producing the next generation.

We adopt a version of ES that has shown to achieve great success in the standard RL benchmarks [29]: Let $f : \Theta \rightarrow \mathbb{R}$ denote the fitness function for a parameter space Θ, e.g., in an RL environment, f computes the stochastic episodic reward of a policy π_θ. To produce the new generation of parameters of size n, [29] uses an additive Gaussian noise with standard deviation σ to perturb the current generation: $\theta^{(i)}_{t+1} = \theta_t + \sigma\epsilon^{(i)}$, where $\epsilon^{(i)} \sim \mathcal{N}(0, I)$. We then evaluate every new generation with fitness function $f(\theta^{(i)}_{t+1})$ for all $i \in [1, \ldots, n]$. The update rule of the parameter is as follows,

$$\begin{aligned}\theta_{t+1} &= \theta_t + \eta \nabla_\theta \mathbb{E}_{\theta\sim\mathcal{N}(\theta_t,\sigma^2 I)}[f(\theta)] \\ &\approx \theta_t + \eta \frac{1}{n\sigma} \sum_i^n f(\theta^{(i)}_{t+1})\epsilon^{(i)},\end{aligned}$$

where η is the learning rate. The update rule is intuitive: each perturbation $\epsilon^{(i)}$ is weighted by the fitness of the corresponding offspring $\theta^{(i)}_{t+1}$. We follow the rank-normalization and mirror sampling techniques of [29] to scale the reward function and reduce the variance of the gradient, respectively.

3 Method

3.1 Learning Branching Heuristic as a Markov Decision Process (MDP)

We formalize the problem of learning the branching heuristic for `#DPLLCache` as an MDP. In our setting, the environment is `SharpSAT`, which is deterministic except for the initial state, where an instance (CNF formula) is chosen randomly from a given distribution. A time step t is equivalent to an invocation of the branching heuristic by the solver (Algorithm 1: line 4). At time step t the agent observes state s_t, consisting of the component ϕ_t that the solver is operating on, and performs an action from the action space $\mathcal{A}_t = \{l | l \in \mathcal{L}(\phi_t)\}$. The objective function is to reduce the number of decisions the solver makes, while solving the counting problem. In detail, the reward function is defined by,

$$R(s) = \begin{cases} 1 & \text{if } s \text{ is a terminal state with ``instance solved'' status,} \\ r_{penalty} & \text{otherwise.} \end{cases}$$

If not finished, episodes are aborted after a predefined maximum number of steps, without receiving the termination reward.

Training with Evolution Strategies. With the objective being defined, we observe that for our task, the potential action space as well as the horizon of the episode can be quite large (up to 20,000 and 1,000, respectively). As [41] shows, the exploration complexity of an action space-exploration RL algorithm (e.g, Q-Learning, Policy Gradient) increases with the size of the action space and the problem horizon. On the other hand, a parameter space-exploration algorithm like ES is independent of these two factors. Therefore, we choose to use a version of ES proposed by [29] for optimizing our agent.

3.2 SharpSAT Components as GNN

As the task for the neural network agent is to pick a literal l from the component ϕ, we opt for a literal-clause incidence graph representation of the CNF formula (see Section 2 for details). We use GNNs to compute a literal selection heuristic based on the LIG graph. The LIG representation is similar to the one used by [36, 14, 22], in contrast to the *variable-clause incidence graph* of [45]. In detail, given the literal-clause incidence graph $G = (V, E)$ of a component ϕ, we denote the set of all clause nodes as $C \subset V$, and the set of all literal nodes as $L \subset V$, $V = C \cup L$. The initial vector

representation is denoted by $h_c^{(0)}$ for each clause $c \in C$ and $h_l^{(0)}$ for each literal $l \in L$. Both are learn-able model parameters. We run the following message passing steps iteratively:

$$\text{Literal to Clause:} \quad h_c^{(k+1)} = \mathrm{A}\Big(h_c^{(k)}, \sum_{l \in c} [h_l^{(k)}, h_{\bar{l}}^{(k)}]; W_C^{(k)}\Big), \quad \forall c \in C,$$

$$\text{Clause to Literal:} \quad h_l^{(k+1)} = \mathrm{A}\Big(h_l^{(k)}, \sum_{c, l \in c} h_c^{(k)}; W_L^{(k)}\Big), \quad \forall l \in L,$$

where A is a nonlinear aggregation function, parameterized by $W_C^{(k)}$ for clause aggregation and $W_L^{(k)}$ for literal aggregation at the kth iteration. Following [36, 22], to ensure the graph representation is invariant under negating every literal (negation invariance), we also concatenate the literal representations corresponding to the same variable $h_l^{(k)}, h_{\bar{l}}^{(k)}$ when running literal-to-clause message passing. After K iterations, we obtain a d-dimensional vector representation for every literal in the graph. We pass each literal representation through a policy network, a Multi-Layer Perceptron (MLP), to obtain a score, and choose the literal with the highest score. Recently, Xu et al. [43] developed a simple GNN architecture named *Graph Isomorphism Network* (GIN), and proved that it achieves maximum expressiveness among the class of GNNs. We hence choose GIN for the parameterization of the aggregation function A. Specifically, $\mathrm{A}(x, y; W) = \mathrm{MLP}((1+\epsilon)x + y; W)$, where ϵ is a hyperparameter. Architectural details are included in Appendix C.

3.3 Semantic Features

In practice, CNF formulas are encoded from a higher level problem in some other domain, with its own semantics. These features of the original problem domain, which we call *semantic features*, are all but lost during the encoding process. Classical constraint solvers only process CNF formulas, and so their heuristics by definition are entirely independent of any specific problem domain, and only consider internal solver properties, such as variable activities. These internal solver properties are a function of the CNF representation and internal solver dynamics, and quite detached from the original problem domain. Thus, it is not unreasonable that semantic features of the original problem domain could contain additional useful structure that can be exploited by the low-level solver heuristic.

One such semantic feature that often naturally arises in real-world problems is *time*. Many problems are iterative in nature, with a distinct temporal dimension to them, e.g., dynamical systems, bounded model checking. At the original problem domain, there is often a state that is evolved through time via repeated applications of a state transition function. A structured CNF encoding of such problems usually maps every state s_t to a set of variables, and adds sets of clauses to represent the dynamical constraints between every transition (s_t, s_{t+1}). As explained, this process removes all temporal information. In contrast, with a learning-based approach, the time step feature from the original problem can be readily incorporated as additional input to the network, effectively annotating each variable with its time-step. In our experiments, we represented time by appending to each literal embedding a scalar value (representing the normalized time-step t) before passing it through the output MLP. We perform an ablation study to investigate the impact of this additional feature in Section 6.

4 Related Work

The first successful application of machine learning to propositional satisfiability solvers was the *portfolio-based* SAT solver `SATZilla` [44]. Equipped with a set of standard SAT solvers, a classifier was trained offline that could map a given SAT instance to the solver from the set that was best suited to solve that instance. Considering that each solver from the set can be regarded as a configuration of a set of heuristics, this method was effectively performing a *heuristic selection* task.

Recent work has been directed along two paths: *heuristic improvement* [35, 21, 22, 45], and purely ML-based solvers [36, 1]. In the former, a model is trained to replace a particular solver heuristic in a standard solver, thus it is embedded as a module within the solver's framework and guides the search process. In the latter approach, the aim is to train a model that acts as a stand-alone "neural" solver. These neural solvers are inherently stochastic and often *incomplete*, meaning that they can only provide an estimate of the satisfiability of a given instance. This is often undesirable in applications of SAT solvers (e.g., formal verification) where an exact answer is required. In terms of functionality,

our work is analogous to the first group, in that we aim at improving the branching heuristics of a standard solver. To our knowledge, no prior work has applied ML to improve exact model counters. More concretely, our work is similar to [45], which used *Reinforcement Learning* (RL) and graph neural networks to learn branching heuristics of a *local search-based* SAT solver `WalkSAT` [34]. Since the scope of local-search solvers is limited to small problems, their method does not scale to industrial-size instances. Our method is also related to [22] and [14], where similar techniques were used in solving quantified Boolean formulas and mixed integer programs, respectively. In contrast to [22], which incorporates a large set of hand-crafted, solver-specific features, our approach requires no prior knowledge about the dynamics of the solver.

5 Data Generation

To evaluate the versatility of our method, we generated a diverse set of problems from various domains to test on. Unlike other works in this area which often experiment on small random instances (e.g., random graphs [45, 36, 21]), we chose our problems from either known SAT benchmarks or real-world applications:

$\mathrm{sudoku}(n, k)$: Randomly generated partially filled $n \times n$ Sudoku problems ($n \in \{9, 16\}$) with k squares revealed (lower is harder). We allow our Sudoku problems to have more than one solution. The #SAT problem then is to count the number of solutions.

$\mathrm{cell}(R, n, r)$[1]: Elementary (i.e., one-dimensional, binary) Cellular Automata are simple systems of computation where the cells of an n-bit binary state vector are progressed through time by repeated applications of a rule R (seen as a function on the state space). Figure 4a shows the evolution grid of rules 9, 35 and 49 for 20 iterations. Reversing Elementary Cellular Automata was proposed as a benchmark problem in SAT Competition 2018 [18]. To generate an instance, we randomly sample a state T. The problem is then to compute the number of initial states I that would lead to terminal state T in r applications of R, i.e., $\left|\{I : R^r(I) = T\}\right|$. The proposed CNF encoding in [18] encodes the entire r-step evolution grid by mapping each cell to a single Boolean variable, $n \times r$ in total. The clauses impose the constraints between cells of consecutive rows as given by the rule R. The variables corresponding to T (last row of the evolution grid) are assigned as unit clauses.

$\mathrm{grid_wrld}(s, t)$: This dataset is based on encoding a grid world with different types of squares (e.g., lava, water, recharge), and a formal specification such as *"Do not recharge while wet"*, or *"avoid lava"* [39, 40]. We randomly sample a grid world of size s and a random starting position I for an agent. At each step, the agent chooses to move uniformly at random between the 4 available directions. We encode the following problem to CNF: *"Count the number of trajectories of length t beginning from I that always avoid lava"*. This number can be used to compute the probability of satisfaction of the agent policy, which can be used for example to infer specifications from demonstrations (see [39, 40] for details).

$\mathrm{bv_expr}(n, d, w)$: For this dataset we randomly generate arithmetic sentences of the form $e_1 \prec e_2$, where $\prec \in \{\leq, \geq, <, >, =, \neq\}$ and e_1, e_2 are expressions of maximum depth d over n binary vector variables of size w, random constants and operators $(+, -, \wedge, \vee, \neg, \mathrm{XOR}, |\cdot|)$. The problem is to count the number of integer solutions to the resulting relation in $([0, 2^w] \cap \mathbb{Z})^n$.

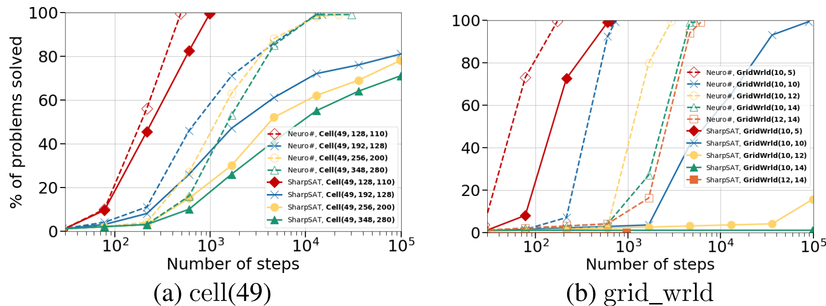
6 Experiments

To evaluate our method, we designed experiments to answer the following questions: **1) I.I.D. Generalization:** Can a model trained on instances from a given distribution generalize to unseen instances of the same distribution? **2) Upward Generalization:** Can a model trained on small instances generalize to larger ones? **3) Wall-Clock Improvement:** Can the model improve the run time substantially? **4) Interpretation:** Does the sequence of actions taken by the model exhibit any discernible pattern at the problem level? Additionally, we studied the impact of the trained model on a variety of solver-specific quality metrics (e.g., cache-hit rate, ...), the results of which are in Appendix D. Our baseline is `SharpSAT`'s heuristic.

[1]The parameters of the cellular automata dataset in a previous version of this paper were slightly different, causing small discrepancies in the results values while not affecting the overall conclusion.

Table 1: `Neuro#` generalizes to both i.i.d. test problems as well as larger, non-i.i.d. ones, sometimes achieving orders of magnitude improvements over `SharpSAT`'s heuristics. All episodes are capped at 100k steps.

	i.i.d.					Upward Generalization			
	# vars	# clauses	SharpSAT	Neuro#		# vars	# clauses	SharpSAT	Neuro#
sudoku(9, 25)	182	3k	220	**195(1.1x)**	sudoku(16, 105)	1k	31k	2,373	**2,300 (1.03x)**
cell(9, 20, 20)	210	1k	370	**184(2.0x)**	cell(9, 40, 40)	820	4k	53,349	**42,325(1.2x)**
cell(35, 128, 110)	6k	25k	353	**198(1.8x)**	cell(35, 192, 128)	12k	49k	21,166	**1,668 (12.5x)**
					cell(35, 256, 200)	25k	102k	26,460	**2,625 (10x)**
					cell(35, 348, 280)	48k	195k	33,820	**2,938 (11.5x)**
cell(49, 128, 110)	6k	25k	338	**206(1.6x)**	cell(49, 192, 128)	12k	49k	24,992	**1,829 (13.6x)**
					cell(49, 256, 200)	25k	102k	30,817	**2,276 (13.5x)**
					cell(49, 348, 280)	48k	195k	37,345	**2,671 (13.9x)**
grid_wrld(10, 5)	329	967	195	**66(3.0x)**	grid_wrld(10, 10)	740	2k	13,661	**367 (37x)**
					grid_wrld(10, 12)	2k	6k	93,093	**1,320 (71x)**
					grid_wrld(10, 14)	2k	7k	100k≤	**2,234 (–)**
					grid_wrld(12, 14)	2k	8k	100k≤	**2,782 (–)**
bv_expr(5, 4, 8)	90	220	328	**205(1.6x)**	bv_expr(7, 4, 12)	187	474	5,865	**2,139 (2.7x)**



(a) cell(49) (b) grid_wrld

Figure 3: `Neuro#` generalizes well to larger problems. Compare the robustness of `Neuro#` vs. `SharpSAT` as the problem sizes increase. Solid and dashed lines correspond to `SharpSAT` and `Neuro#`, respectively. All episodes are capped at 100k steps.

The grid_wrld problem was a natural candidate for testing the effect of adding the time feature (Section 3.3), so we report the results for that problem with time feature included and later in this section we perform an ablation study on that feature.

Experimental Protocol. For each dataset, we sampled 1,800 instances for training and 200 for testing. We trained for 1000 ES iterations. At each iteration, we sampled 8 formulas from the training set and 48 perturbations with $\sigma = 0.02$. With mirror sampling, we obtained in total $96 = 48 \cdot 2$ perturbations. For each perturbation, we ran the agent on the 8 formulas (in parallel), to a total of $768 = 96 \cdot 8$ episodes per parameter update. All episodes, unless otherwise mentioned, were capped at 1000 steps during training and 100,000 during testing. The agent received a negative reward of $r_{penalty} = -10^{-4}$ at each step. We used the Adam optimizer [20] with default hyperparameters and a learning rate of 0.01. We used a weight decay of 0.005 and used the same architectural hyperparameters for our model for all datasets (details in Appendix C).

6.1 Results

I.I.D. Generalization. Table 1 summarizes the results of the i.i.d. generalization over the four problem domains of Section 5. We report the average number of branching steps on the test set. `Neuro#` outperformed the baseline across all datasets. Most notably, on grid_wrld, it reduced the number of branching steps by a factor of 3.0, from 195 down to 66. On cell, it reduced it by an average factor of 1.8 over the three different cellular rules. Similar improvement held for bv_expr. We observed less improvements on sudoku; we conjecture this is due to the dense structure of the problem. The sudoku encoding is *global*, in that every square is 1 hop away on the LIG from all other relevant squares, and there is no *local* problem structure to exploit. Appendix B.1 includes cactus plots comparing the performance of `SharpSAT` to `Neuro#` across all datasets.

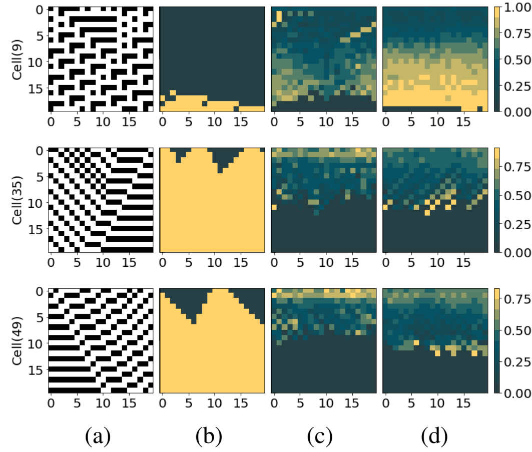


Figure 4: Contrary to `SharpSAT`, `Neuro#` branches earlier on variables of the bottom rows. (a) Evolution of a bit-vector through repeated applications of Cellular Automata rules. The result of applying the rule at each iteration is placed under the previous bit-vector, creating a two-dimensional, top-down representation of the system's evolution; (b) The initial formula simplification on a *single* formula. Yellow indicates the regions of the formula that this process prunes; (c) & (d) Variable selection ordering by `SharpSAT` and `Neuro#` averaged over the entire dataset. Lighter colours show that the corresponding variable is selected earlier on average.

Upward Generalization. Directly training on challenging #SAT problems of enormous size is computationally infeasible, as the agent's exploration during training can take forever. We tackled this issue by training `Neuro#` on small problem instances and relying on generalization to solve the more challenging instances from the same problem domain. We created instances of larger sizes (up to an order of magnitude more clauses and variable) for each of the datasets in Section 5. We took the models trained from the previous i.i.d. setting and directly evaluated on these larger instances without further training.

The evaluation results are shown in the right half of Table 1. We see that `Neuro#` generalized to the larger instances across all datasets and in almost all of them achieved substantial gains compared to the baseline as we increased the instance sizes. Figure 3 shows this effect for multiple sizes of $\mathrm{cell}(49)$ and $\mathrm{grid_wrld}$ by plotting the percentage of the problems solved within a number of steps (plots for other problems are included in Appendix B.2). The gaps get more pronounced once we remove the cap of 10^5 steps, i.e., let the episodes run to completion. In that case, on $\mathrm{grid_wrld}(10, 12)$, `Neuro#` took an average of 1,320 branching decisions, whereas `SharpSAT` took 809,408 (613x improvement).

Wall-Clock Improvement. Improvements of this scale in step count on large instances are significant enough for `Neuro#` to beat `SharpSAT` in wall-clock time, as evident in Figure 1 for $\mathrm{grid_wrld}$ and in Figure 5 for $\mathrm{cell}(49)$. Note that this is in spite of the imposed overhead of querying the model that limits the number of steps `Neuro#` can take per second compared to `SharpSAT`. For example, while solving $\mathrm{cell}(49, 256, 200)$, `SharpSAT` took 331 steps/sec on average whereas `Neuro#` was only able to take 17. We expect that this overhead could be greatly reduced, as our implementation is far from optimized: it calls an out of process Python code from within the solver's main loop (in C++), does not utilize a GPU nor does it perform any optimizations on the neural network's inference.

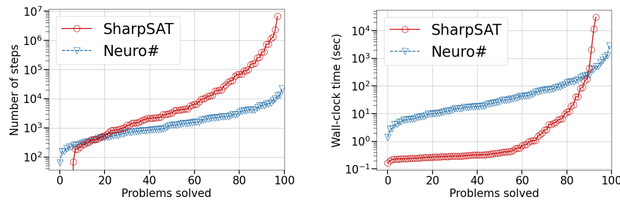


Figure 5: Cactus plots comparing `Neuro#` to `SharpSAT` on the $\mathrm{cell}(49, 256, 200)$ benchmark (lower and to the right is better). For any point t on the y axis, the plot shows the number of benchmark problems that are individually solvable by the solver, within t steps (left) and seconds (right).

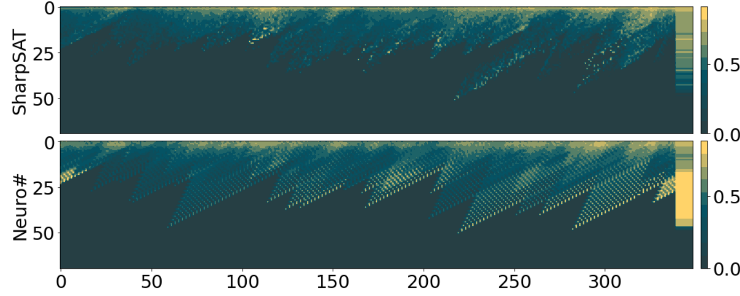


Figure 6: Full-sized variable selection heatmap on dataset $\mathrm{cell}(35, 348, 280)$. Lighter colours show that the corresponding variable is selected earlier on average across the dataset. We show the 99th percentile for each row of the heatmap in the last column. Notice `Neuro#`'s tendency towards selecting variables of the bottom rows earlier.

Problem-Level Interpretation. Encodings to CNF can be quite removed from the original problem domain. Consider $\mathrm{grid_wrld}$: the problems are encoded to a state machine, then to a circuit, and finally to CNF, and many new variables are created along this process. In contrast, cell has a straightforward encoding that directly relates the CNF representation to an easy-to-visualize evolution grid which coincides with the standard representation of Elementary Cellular Automata. This allows for interpretation of `Neuro#`'s policy in the original problem domain.

Our conjecture was that the model will learn to solve the problem from the bottom up. On the evolution grid, the known terminal state T is the bottom row, and the task is to count the number of distinct top rows I compatible with T. The natural way to decompose this problem is to start from the known state T and continue assigning variables to "guess" the preimage, row by row from bottom up. Different preimages can be computed *independently* upwards, and indeed, this is how a human would approach the problem.

Heat maps in Figure 4 (c) and (d) depict the behaviour under `SharpSAT` and `Neuro#` respectively. The heat map aligns with the evolution grid, with the terminal state T at the bottom. For each dataset, the hotter coloured cells indicate that, on average, the corresponding variable tends to be branched on earlier by the policy. The cooler colours show that the variable is often selected later or not at all, meaning that its value is often inferred through UP either initially or after some variable assignments. That is why the bottom row T and adjacent rows are completely dark, because they are simplified by the solver before any branching happens. We show the effect of this early simplification on a single formula per dataset in Figure 4 (b). Notice that in $\mathrm{cell}(35)$ and $\mathrm{cell}(49)$ the simplification shatters the problem space into few small components (dark triangles), while in $\mathrm{cell}(9)$ which is a more challenging problem, it only chips away a small region of the problem space, leaving it as a single component. Regardless of this, as conjectured, we can see a clear trend with `Neuro#` focusing more on branching early on variables of the bottom rows in $\mathrm{cell}(9)$ and in a less pronounced way in $\mathrm{cell}(35\&49)$. Moreover, as more clearly seen in the heatmap for the larger problem in Fig 6, the learned heuristics actually branches early according to the pattern of the rule.

Time Feature. We tested the degree to which the "time" feature contributed to the upward generalization performance of $\mathrm{grid_wrld}$. We compared three architectures with `SharpSAT` as the baseline: **1.** *GNN*: The standard architecture proposed in Section 3.2, **2.** *GNN+Time*: Same as *GNN* but with the variable embeddings augmented with the "time" semantic feature (Section 3.3) and **3.** *Time*: Where no variable embedding is computed and only the "time" feature is fed to the policy network.

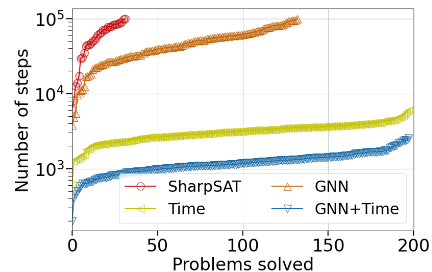


Figure 7: Ablation study on the impact of the "time" feature on upward generalization on $\mathrm{grid_wrld}(10, 12)$.

As can be seen in Figure 7, we discovered that the "time" feature is responsible for most of the improvement over `SharpSAT`. This fact is encouraging, because it demonstrates the potential gains that could be achieved by simply utilizing problem-level data, such as "time", that otherwise

would have been lost during the CNF encoding. More elaborate ablation studies can be found in Appendix E.

7 Conclusion

We studied the feasibility of enhancing the variable branching heuristic in propositional model counting via learning. We used the branching steps that the solver makes as a measure of its performance and trained our model to minimize that measure. We demonstrated experimentally that the resulting model not only is capable of generalizing to the unseen instances from the same problem distribution, but also maintains its lead relative to `SharpSAT` on larger problems. For certain problems, this lead widens to a degree that the trained model achieves wall-clock time improvement over the standard heuristic, in spite of the imposed run time overhead of querying the model. This is exciting as it positions this line of research as a potential path towards building better model counters and hence broadening their application horizon.

References

[1] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. Learning To Solve Circuit-SAT: An Unsupervised Differentiable Approach. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. `https://openreview.net/forum?id=BJxgz2R9t7`.

[2] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. solving #SAT and Bayesian inference with backtracking search.

[3] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and Complexity Results for #SAT and Bayesian Inference. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 340–351. IEEE Computer Society, 2003. doi: 10.1109/SFCS.2003.1238208. `https://doi.org/10.1109/SFCS.2003.1238208`.

[4] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Value Elimination: Bayesian Interence via Backtracking Search. In Christopher Meek and Uffe Kjærulff, editors, *UAI '03, Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence, Acapulco, Mexico, August 7-10 2003*, pages 20–28. Morgan Kaufmann, 2003. `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=909&proceeding_id=19`.

[5] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. Learning to Branch. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 353–362. PMLR, 2018. `http://proceedings.mlr.press/v80/balcan18a.html`.

[6] Roberto J. Bayardo, Jr. and Joseph Daniel Pehoushek. Counting Models Using Connected Components. In Henry A. Kautz and Bruce W. Porter, editors, *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pages 157–162. AAAI Press / The MIT Press, 2000. `http://www.aaai.org/Library/AAAI/2000/aaai00-024.php`.

[7] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies - A Comprehensive Introduction. *Nat. Comput.*, 1(1):3–52, 2002. doi: 10.1023/A:1015059928466. `https://doi.org/10.1023/A:1015059928466`.

[8] Elazar Birnbaum and Eliezer L. Lozinskii. The Good Old Davis-Putnam Procedure Helps Counting Models. *J. Artif. Intell. Res.*, 10:457–477, 1999. doi: 10.1613/jair.601. `https://doi.org/10.1613/jair.601`.

[9] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-Aware Sampling and Weighted Model Counting for SAT. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial*

Intelligence, July 27 -31, 2014, Québec City, Québec, Canada, pages 1722–1730. AAAI Press, 2014. http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8364.

[10] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. doi: 10.1145/321033.321034. URL http://doi.acm.org/10.1145/321033.321034.

[11] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. doi: 10.1145/368273.368557. URL https://doi.org/10.1145/368273.368557.

[12] Carmel Domshlak and Jörg Hoffmann. Fast probabilistic planning through weighted model counting. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 243–252. AAAI, 2006. URL http://www.aaai.org/Library/ICAPS/2006/icaps06-025.php.

[13] Carmel Domshlak and Jörg Hoffmann. Probabilistic Planning via Heuristic Forward Search and Weighted Model Counting. *J. Artif. Intell. Res.*, 30:565–620, 2007. doi: 10.1613/jair.2289. https://doi.org/10.1613/jair.2289.

[14] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 15554–15566, 2019. http://papers.nips.cc/paper/9690-exact-combinatorial-optimization-with-graph-convolutional-neural-networks.

[15] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model Counting. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009. https://doi.org/10.3233/978-1-58603-929-5-633.

[16] M. Gori, G. Monfardini, and F. Scarselli. A New Model for Learning in Graph Domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734 vol. 2, 2005.

[17] C. Hansknecht, I. Joormann, and S. Stiller. Cuts, Primal Heuristics, and Learning to Branch for the Time-Dependent Traveling Salesman Problem. Technical report, arXiv, May 2018. https://arxiv.org/abs/1805.01415.

[18] Marijn J. H. Heule, Matti Juhani Järvisalo, and Martin Suda, editors. *Proc. of SAT Competition 2018: Solver and Benchmark Descriptions*, 2018. University of Helsinki. http://hdl.handle.net/10138/237063.

[19] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra Dilkina. Learning to Branch in Mixed Integer Programming. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 724–731. AAAI Press, 2016. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12514.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. http://arxiv.org/abs/1412.6980.

[21] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning. *CoRR*, abs/1909.11830, 2019. http://arxiv.org/abs/1909.11830.

[22] Gil Lederman, Markus N. Rabe, Sanjit Seshia, and Edward A. Lee. Learning Heuristics for Quantified Boolean Formulas through Reinforcement Learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. https://openreview.net/forum?id=BJluxREKDB.

[23] Wei Li, Pascal Poupart, and Peter van Beek. Exploiting Structure in Weighted Model Counting Approaches to Probabilistic Inference. *J. Artif. Intell. Res.*, 40:729–765, 2011. http://jair.org/papers/paper3232.html.

[24] João Marques-Silva. Computing with SAT Oracles: Past, Present and Future. In Florin Manea, Russell G. Miller, and Dirk Nowotka, editors, *Sailing Routes in the World of Computation - 14th Conference on Computability in Europe, CiE 2018, Kiel, Germany, July 30 - August 3, 2018, Proceedings*, volume 10936 of *Lecture Notes in Computer Science*, pages 264–276. Springer, 2018. https://doi.org/10.1007/978-3-319-94418-0_27.

[25] Kuldeep S. Meel and S. Akshay. Sparse Hashing for Scalable Approximate Model Counting: Theory and Practice. *CoRR*, abs/2004.14692, 2020. https://arxiv.org/abs/2004.14692.

[26] Umut Oztok and Adnan Darwiche. A Top-Down Compiler for Sentential Decision Diagrams. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3141–3148. AAAI Press, 2015. http://ijcai.org/Abstract/15/443.

[27] Neil Robertson and Paul D. Seymour. Graph Minors. X. Obstructions to Tree-Decomposition. *J. Comb. Theory, Ser. B*, 52(2):153–190, 1991. doi: 10.1016/0095-8956(91)90061-N. https://doi.org/10.1016/0095-8956(91)90061-N.

[28] Neil Robertson and Paul D. Seymour. Graph Minors XXIII. Nash-Williams' Immersion Conjecture. *J. Comb. Theory, Ser. B*, 100(2):181–205, 2010. doi: 10.1016/j.jctb.2009.07.003. https://doi.org/10.1016/j.jctb.2009.07.003.

[29] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *CoRR*, abs/1703.03864, 2017. URL http://arxiv.org/abs/1703.03864.

[30] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004. http://www.satisfiability.org/SAT04/programme/21.pdf.

[31] Tian Sang, Paul Beame, and Henry A. Kautz. Performing Bayesian Inference by Weighted Model Counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 475–482. AAAI Press / The MIT Press, 2005. http://www.aaai.org/Library/AAAI/2005/aaai05-075.php.

[32] Tian Sang, Paul Beame, and Henry A. Kautz. Heuristics for Fast Exact Model Counting. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2005. https://doi.org/10.1007/11499107_17.

[33] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Trans. Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605. https://doi.org/10.1109/TNN.2008.2005605.

[34] Bart Selman, Henry A. Kautz, and Bram Cohen. Local Search Strategies for Satisfiability Testing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–531. DIMACS/AMS, 1993. doi: 10.1090/dimacs/026/25. https://doi.org/10.1090/dimacs/026/25.

[35] Daniel Selsam and Nikolaj Bjørner. NeuroCore: Guiding High-Performance SAT Solvers with Unsat-Core Predictions. *CoRR*, abs/1903.04671, 2019. http://arxiv.org/abs/1903.04671.

[36] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT Solver from Single-Bit Supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. https://openreview.net/forum?id=HJMC_iA5tm.

[37] Marc Thurley. SharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006. https://doi.org/10.1007/11814948_38.

[38] Seinosuke Toda. PP is as Hard as the Polynomial-Time Hierarchy. *SIAM J. Comput.*, 20(5): 865–877, 1991. https://doi.org/10.1137/0220053.

[39] Marcell Vazquez-Chanlatte, Susmit Jha, Ashish Tiwari, Mark K. Ho, and Sanjit A. Seshia. Learning Task Specifications from Demonstrations. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, pages 5372–5382, 2018. http://papers.nips.cc/paper/7782-learning-task-specifications-from-demonstrations.

[40] Marcell Vazquez-Chanlatte, Markus N. Rabe, and Sanjit A. Seshia. A Model Counter's Guide to Probabilistic Systems. *CoRR*, abs/1903.09354, 2019. http://arxiv.org/abs/1903.09354.

[41] Anirudh Vemula, Wen Sun, and J. Andrew Bagnell. Contrasting Exploration in Parameter and Action Space: A Zeroth-Order Optimization Perspective. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*, volume 89 of *Proceedings of Machine Learning Research*, pages 2926–2935. PMLR, 2019. http://proceedings.mlr.press/v89/vemula19a.html.

[42] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural Evolution Strategies. *J. Mach. Learn. Res.*, 15(1):949–980, 2014. http://dl.acm.org/citation.cfm?id=2638566.

[43] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. https://openreview.net/forum?id=ryGs6iA5Km.

[44] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *J. Artif. Intell. Res.*, 32:565–606, 2008. doi: 10.1613/jair.2490. https://doi.org/10.1613/jair.2490.

[45] Emre Yolcu and Barnabás Póczos. Learning Local Search Heuristics for Boolean Satisfiability. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 7990–8001, 2019. http://papers.nips.cc/paper/9012-learning-local-search-heuristics-for-boolean-satisfiability.

A #SAT Algorithms

In this section we provide some more details about exact algorithms for solving #SAT, see [2] for the full formal details including all proofs.

The simplest algorithm for #SAT is to extend the backtracking search DPLL algorithm to make it explore the full set of truth assignments. This is the basis of the CDP solver presented in [8], shown in Algorithm 2. In particular, when the current formula contains an empty clause it has zero models, and when it contains no clauses each of the remaining k unset variables can be assigned **true** or **false** so there are 2^k models (line 6).

Algorithm 2 DPLL extended to count all solutions (CDP)

```
1: function CDP(φ)
2:     if φ contains an empty clause then
3:         return 0
4:     if φ contains no clauses then
5:         k = # of unset variables
6:         return 2^k
7:     Pick a literal l ∈ φ
8:     return CDP(UP(φ, l)) + CDP(UP(φ, ¬l))
```

Algorithm 3 Using Components

```
1: function Relsat(φ)
2:     Pick a literal l ∈ φ
3:     #l = CountSide(φ, l)
4:     #¬l = CountSide(φ, ¬l)
5:     return #l + #¬l

6: function CountSide(φ, l)
7:     φ_l = UP(φ, l)
8:     if φ_l contains an empty clause then
9:         return 0
10:    if φ_l contains no clauses then
11:        k = # of unset variables
12:        return 2^k
13:    K = findComponents(φ_l)
14:    return ∏_{κ∈K} Relsat(κ)
```

This algorithm is not very efficient, running in time $2^{\Theta(n)}$ where n is the number of variables in the input formula. Note that the algorithm is actually a class of algorithms each determined by the procedure used to select the next literal to branch on. The complexity bound is strong in the sense that no matter how the branching decisions are made, we can find a sequence of input formulas on which the algorithm will take time exponential in n as the formulas get larger.

Breaking the formula into components and solving each component separately is an approach suggested by Bayardo and Pehoushek [6] and used in the `Relsat` solver. This approach is shown in Algorithm 3. This algorithm works on one component at a time and is identical to `#DPLLCache` (Algorithm 1) except that caching is not used.

Breaking the formula into components can yield considerable speedups depending on n_0, the number of variables needed to be set before the formula is broken into components. If we consider a hypergraph in which every variable is a node and every clause is a hyperedge over the variables mentioned in the clause, then the branch-width [27] of this hypergraph provides an upper bound on n_0. As a result we can obtain a better upper bound on the run time of `Relsat` of $n^{O(w)}$ where w is the branch-width of the input's hypergraph. However, this run time will only be achieved if the branching decisions are made in an order that respects the branch decomposition with width w. In particular, there exists a sequence of branching decisions achieving a run time of $n^{O(w)}$. Computing that sequence would require time $n^{O(1)}2^{O(w)}$ [28], hence a run time of $n^{O(w)}$ can be achieved.

Finally, if component caching is used we obtain Algorithm 1 which has a better upper bound of $2^{O(w)}$. Again this run time can be achieved with a $n^{O(1)}2^{O(w)}$ computation of an appropriate sequence of branching decisions.

In practice, the branch-width of most instances is very large, making a run time of $2^{O(w)}$ infeasible. Computing a branching sequence to achieve that run time is also infeasible. Fortunately, in practical instances unit propagation is also very powerful. This means that making only a few decisions ($< w$) often allows unit propagation to set w or more variables thus breaking the formula apart into separate components. Furthermore, most instances are falsified by a large proportion of their truth assignments. This makes clause learning an effective addition to #SAT solvers, as with it the solver can more effectively traverse the non-solution space.

In sum, for #SAT solvers the branching decisions try to achieve complex and sometimes contradictory objectives. Making decisions that split the formula into larger components near the top of the search tree (i.e., after only a few decisions are made) allows greater speedups, while generating many small components near the bottom of the search trees (i.e., after many decision are made) does not help the solver. Making decisions that generate the same components under different branches allows more

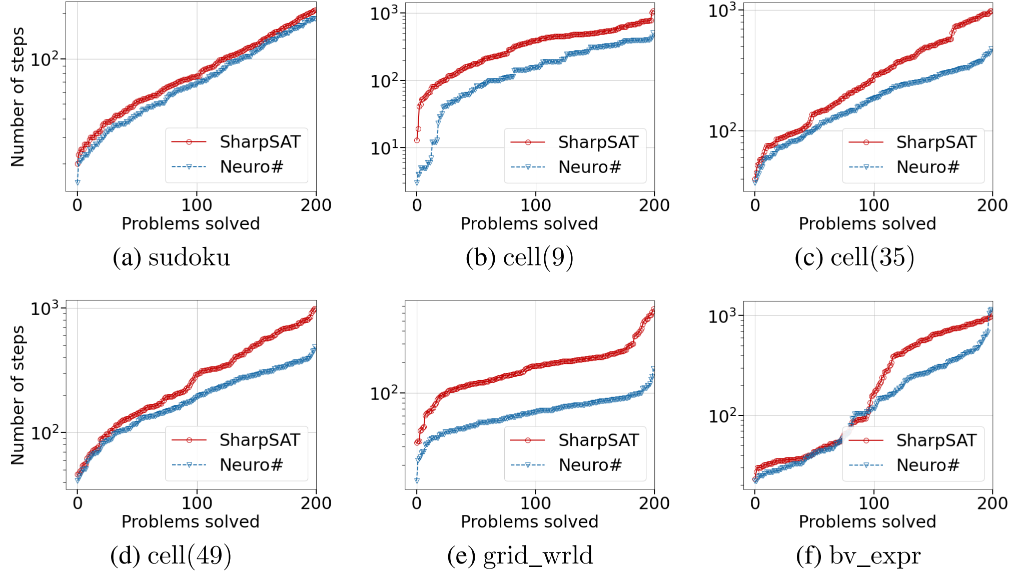


Figure 8: Cactus Plot – Neuro# outperforms SharpSAT on all i.i.d benchmarks (lower and to the right is better). A cut-off of 100k steps was imposed though both solvers managed to solve the datasets in less than that many steps.

effective use of the cache. And making decisions that allow the solver to learn more effective clauses allows the solver to more efficiently traverse the often large space of non-solutions.

B More on the Results

In this section we present a more elaborate discussion of our results. Aggregated measures of performance, such as average number of decisions (Table 1) only give us an overall indication of Neuro#'s lead compared to SharpSAT and as such, they are incapable of showing whether it is performing better on easier or harder instances in the dataset. Cactus plots are the standard way of comparing solver performances in the SAT community. Although typically used to compare the wall-clock time (Figure 1b), here we use them to compare the number of steps (i.e., branching decisions).

B.1 I.I.D. Generalization

Figure 8 shows cactus plots for all of the i.i.d. benchmark problems. Unsurprisingly, the improvements on sudoku are relatively modest, albeit consistent across the dataset. On all cell datasets, and grid_wrld, an exponential growth is observed with Neuro#'s lead over SharpSAT as the problems get more difficult (moving right along the x axis). Lastly, on bv_expr, Neuro# does better almost universally, except near the 100 problems mark and at the very end (3 most difficult problems).

B.2 Upwards Generalization

On some datasets, namely cell(49) and grid_wrld, the Neuro#'s lead over SharpSAT becomes more pronounced as we test the upwards generalization (using the model trained on smaller instances and testing on larger ones). Cactus plots of Figure 9&10 show this effect clearly for these datasets. In each figure, the i.i.d. plot is included as a reference on the left and on the right the plots for test sets with progressively larger instances are depicted.

The upward generalization lead is less striking, although still significant, on bv_expr (2.7x up from 1.6x). On sudoku and cell(9) Neuro#'s lead is still maintained but it becomes less prominent on more difficult datasets. Figure 11 summarizes these points by comparing the percentage of the problems solvable by SharpSAT vs. Neuro# under a given number of steps. Notice the robustness of the learned model in cell(35&49) and grid_wrld. As these datasets get more difficult, SharpSAT either

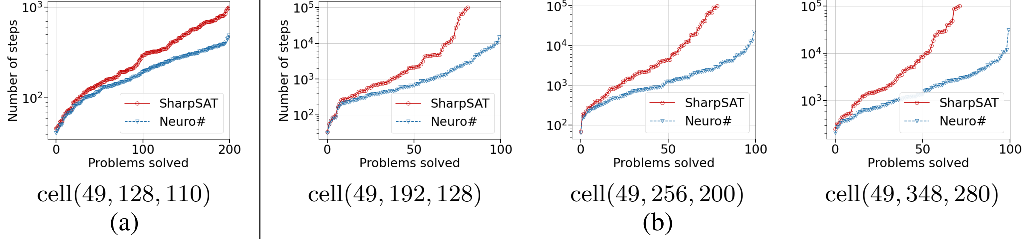


Figure 9: Cactus Plot – cell(49): `Neuro#` maintains its lead over `SharpSAT` on larger datasets (lower and to the right is better). A cut-off of 100k steps was imposed. **(a)** i.i.d. generalization on $\text{cell}(49, 128, 110)$; **(b)** Upward generalization of the model trained on $\text{cell}(49, 128, 110)$ over larger datasets.

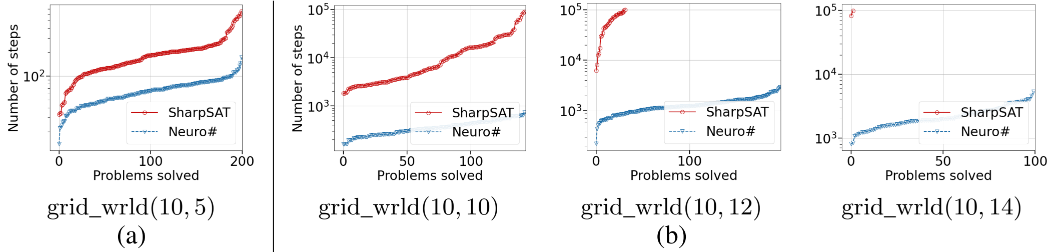


Figure 10: Cactus Plot – grid_wrld: `Neuro#` maintains its lead over `SharpSAT` on larger datasets (lower and to the right is better). A cut-off of 100k steps was imposed. **(a)** i.i.d. generalization on $\text{grid_wrld}(10, 5)$; **(b)** Upward generalization of the model trained on $\text{grid_wrld}(10, 5)$ over larger datasets.

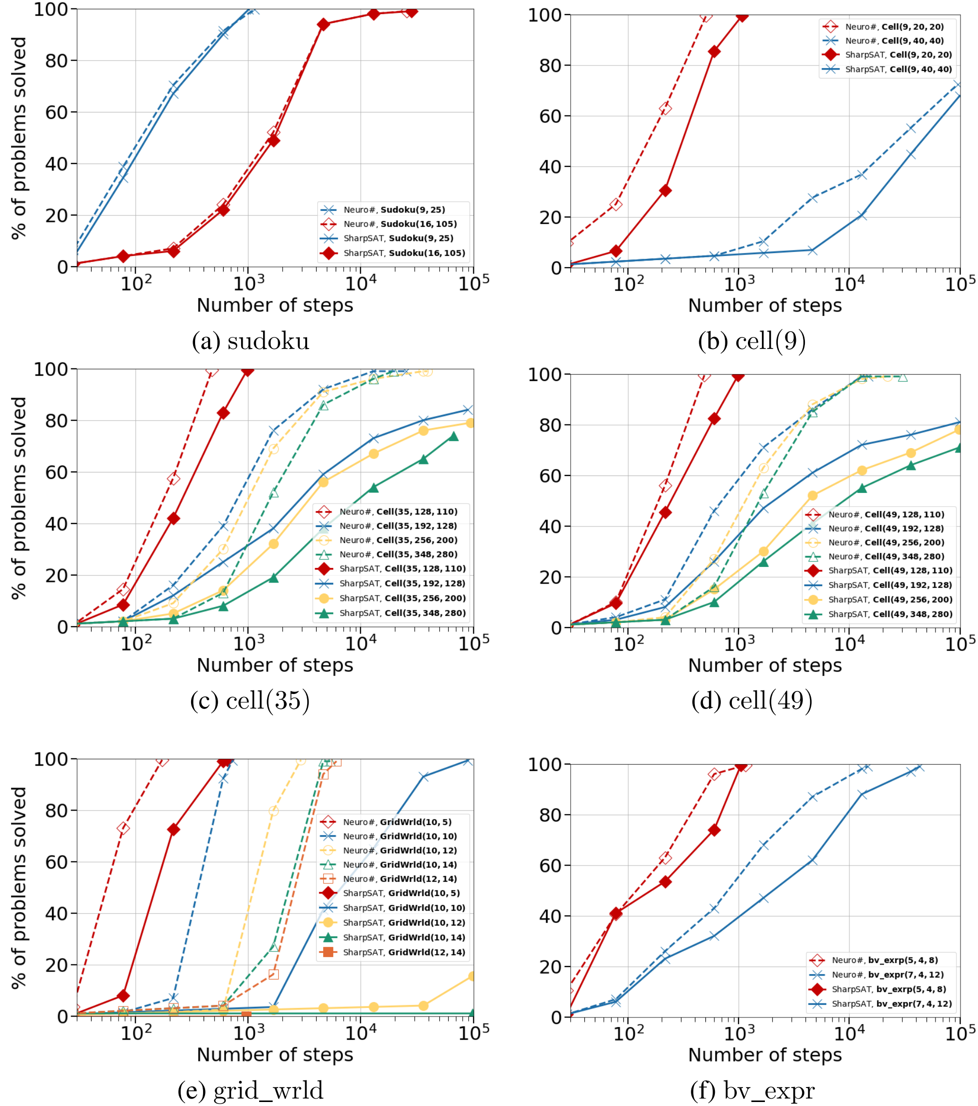
takes more steps or completely fails to solve the problems altogether, whereas `Neuro#` relatively sustains its performance.

B.3 Discussion

Many dataset attributes may lead to the upward generalization success of the aforementioned datasets, but one of the main contributing factors is the model's ability to observe similar components many times during training. In other words, if a problem gets shattered by the initial simplification (unit propagation) into smaller components, there is a high chance that the model's behaviour learns to solve such components. If larger problems of the same domain also break down into similar components, then `Neuro#` can generalize well on them. In Section 6.1, we discussed this phenomena for cell via heat maps. In Figure 12 we provide full heat maps for larger datasets of both $\text{cell}(35)$ and $\text{cell}(49)$. Not only the "shattering" effect is evident from these plots, we can also observe that in both datasets `Neuro#` branches on variables from the bottom going up. This matches with our conjecture presented in Section 6.1.

C Architecture Details

Both our literal and clause embeddings are of size 32. GNN messages are implemented by an MLP with $ReLU$ non-linearity. Clause-to-literal messages are of dimensions $32 \times 32 \times 32$, and literal-to-clause messages are of dimension $64 \times 32 \times 32$ (as described in Section 3 we "tie" the literals to achieve negation-invariance, hence the doubled first dimension). We use 2 iterations in the GNN, and final literal embeddings are passed through the MLP policy network of dimensions $32 \times 256 \times 64 \times 1$ to get the final score. When using the extra time feature, the first dimension of the decision layer is 33. The initial (iteration 0) embeddings of both literals and clauses are trainable model parameters. In Appendix E, where we augment the literal features with "*variable scores*" we start with a feature vector of size 2 for each literal, and pass it through an MLP of dimensions $2 \times 32 \times 32$ to get the initial literal embedding.



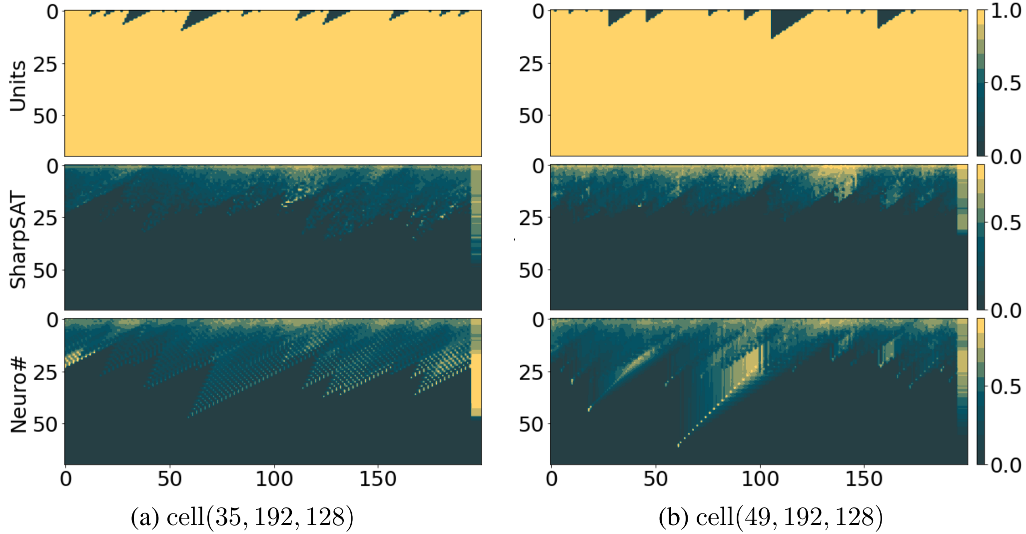
(a) sudoku (b) cell(9) (c) cell(35) (d) cell(49) (e) grid_wrld (f) bv_expr

Figure 11: `Neuro#` generalizes well to larger problems on almost all datasets (higher and to the left is better). Compare the robustness of `Neuro#` vs. `SharpSAT` as the problem sizes increase. Solid and dashed lines correspond to `SharpSAT` and `Neuro#`, respectively. All episodes are capped at 100k steps.

D Trained Policy's Impact on Solver Performance Measures

In this section we analyze the impact of `Neuro#` on solver's performance through the lens of a set of solver-specific performance measures. These measures include: **1.** Number of conflict clauses that the solver encounters while solving a problem (`num conflicts`), **2.** Total (hit+miss) number of cache lookups (`num cache lookups`), **3.** Average size of components stored on the cache (`avg(comp size stored)`), **4.** Cache hit-rate (`cache hit-rate`) and **5.** Average size of the components that are successfully found on the cache (`avg(comp size hit)`).

A conflict clause is generated whenever the solver encounters an empty clause, indicating that the current sub-formula has zero models. Thus the number of conflict clauses generated is a measure of the amount of work the solver spent traversing the *non-solution space* of the formula. Cache hits and the size of the cached components, on the other hand, give an indication of how effectively the solver is able to traverse the formula's *solution space*. In particular, when a component with k variables is found in the cache (a cache hit) the solver does not need to do any further work to count the number of solutions over those k variables. This could potentially save the solver $2^{O(k)}$ computations. This

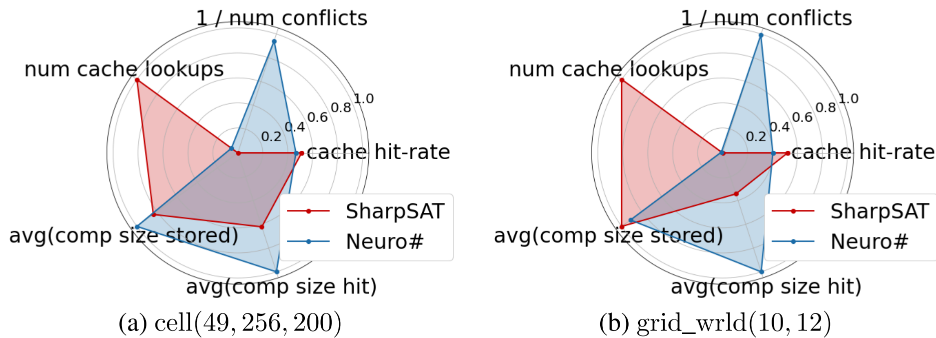


(a) $\mathrm{cell}(35, 192, 128)$ (b) $\mathrm{cell}(49, 192, 128)$

Figure 12: Clear depiction of Neuro#'s pattern of variable branching. The "Units" plots show the initial formula simplification the solvers. Yellow indicates the regions of the formula that this process prunes. Heatmaps show the variable selection ordering by SharpSAT and Neuro#. Lighter colours show that the corresponding variable is selected earlier on average across the dataset.

$2^{O(k)}$ worst case time is rarely occurs in practice; nevertheless, the number of cache hits, and the average size of the components in those cache hits give an indication of how effective the solver is in traversing the formula's solution space. Additional indicators of solver's performance in traversing the solution space are the number of components generated and their average size. Every time the solver is able to break its current sub-formula into components it is able to reduce the worst case complexity of solving that sub-formula. For example, when a sub-formula of m variables is broken up into two components of k_1 and k_2 variables, the worst case complexity drops from $2^{O(m)}$ to $2^{O(k_1)} + 2^{O(k_2)}$. Again the worst case rarely occurs (as indicated by the fact that #SAT solvers do not display worst case performance on most inputs), so the number of components generated and their average size provide only an indication of the solver's effectiveness in traversing the formula's solution space.

In Figure 13 we plot these measures for $\mathrm{cell}(49, 256, 200)$ and $\mathrm{grid_wrld}(10, 12)$. Looking at the individual performance measures, we see that the Neuro# encounters fewer conflicts (larger `1/num conflicts`), meaning that it is traversing the non-solution space more effectively in both datasets. The cache measures, indicate that the standard heuristic is able to traverse the solution space a bit more effectively, finding more components (`num cached lookups`) of similar or larger average size. However, Neuro# is able to utilize the cache as efficiently (with comparable cache hit rate) while finding components in the cache that are considerably larger than those found by the



(a) $\mathrm{cell}(49, 256, 200)$ (b) $\mathrm{grid_wrld}(10, 12)$

Figure 13: Radar charts showing the impact of each policy across different solver-specific performance measures.

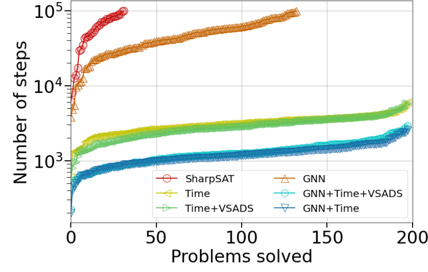


Figure 14: Cactus Plot – Ablation study on the impact of the "time" and VSADS features over upward generalization on $\text{grid_wrld}(10, 12)$ (lower and to the right is better). A termination cap of 100k steps was imposed on the solver.

standard heuristic. In sum, the learnt heuristic finds an effective trade-off of learning more powerful clauses, with which the solver can more efficiently traverse the non-solution space, at the cost of a slight degradation in its efficiency traversing the solution space. The net result in an improvement in the solver's run time.

E Ablation Study

Variable Score We mentioned in Section 2 that `SharpSAT`'s default way of selecting variables is based on the VSADS heuristic which incorporates the number of times a variable v appears in the current sub-formula, and (a function of the) number of conflicts it took part in. At every branching juncture, the solver picks a variable among the ones in the current component with maximum score and branches on one of its literals (see Algorithm 1). As part of our efforts to improve the performance of our model, we performed an additional ablation study over that of Section 6.1. Concretely, we measured the effect of including the variable scores in our model (as detailed in Appendix C) and tested on the $\text{grid_wrld}(10, 12)$ and $\text{cell}(49, 256, 200)$ datasets (Figures 14 & 15). For both datasets, the inclusion of the variable scores produced results inferior to the ones achieved without them! This is surprising, though consistent with what was observed in [22].

Random Policies As an essential sanity check, we tested how a "random policy" performs compared to the trained model, in order to assure that our model's performance improvements are not trivially attainable without training. To that end, we tested on $\text{cell}(35, 128, 110)$ dataset two such random policies: **1) Random Literal:** which chooses a literal uniformly at random; **2) Random Network**: where we randomly set our model's weights instead of training. Both of these policies were inferior to the `SharpSAT`'s results of 353 steps (Table 1), achieving an average of 867 and 740, respectively.

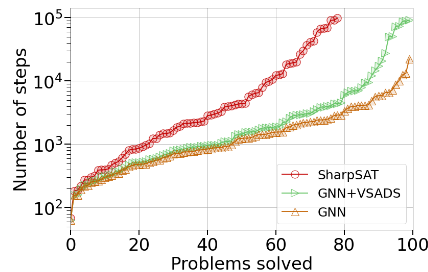


Figure 15: Cactus Plot – Inclusion of VSADS score as a feature hurts the upward generalization on $\text{cell}(49, 256, 200)$ (lower and to the right is better). A termination cap of 100k steps was imposed on the solver.