Learning Branching Heuristics for Propositional Model Counting

Pashootan Vaezipoor,1* Gil Lederman,2* Yuhuai Wu,1,3 Chris Maddison,1,3
Roger B. Grosse,1,3 Sanjit A. Seshia,2 Fahiem Bacchus1
1 University of Toronto
2 UC Berkeley
3 Vector Institute
{pashootan, ywu, cmaddis, rgrosse, fbacchus}@cs.toronto.edu
{gilled, sseshia}@eecs.berkeley.edu

Abstract

Propositional model counting, or #SAT, is the problem of computing the number of satisfying assignments of a Boolean formula. Many problems from different application areas, including many discrete probabilistic inference problems, can be translated into model counting problems to be solved by #SAT solvers. Exact #SAT solvers, however, are often not scalable to industrial size instances. In this paper, we present Neuro#, an approach for learning branching heuristics to improve the performance of exact #SAT solvers on instances from a given family of problems. We experimentally show that our method reduces the step count on similarly distributed held-out instances and generalizes to much larger instances from the same problem family. It is able to achieve these results on a number of different problem families having very different structures. In addition to step count improvements, Neuro# can also achieve orders of magnitude wall-clock speedups over the vanilla solver on larger instances in some problem families, despite the runtime overhead of querying the model.

1 Introduction

Propositional model counting is the problem of counting the number of satisfying solutions to a Boolean formula (Gomes, Sabharwal, and Selman 2009). When the Boolean formula is expressed in conjunctive normal form (CNF), this problem is known as the #SAT problem. #SAT is a #P-complete problem, and by Toda’s theorem (Toda 1991) 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 include many discrete probabilistic inference problems, e.g., inference in Bayes Nets (Bacchus, Dalmao, and Pitassi 2003b; Li, Poupart, and van Beek 2011; Sang, Beame, and Kautz 2005b; Domshlak and Hoffmann 2007) and bounded-length probabilistic planning (Domshlak and Hoffmann 2006); 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 exact #SAT solvers.

In particular, we learn problem family specific branching heuristics for the 2012 version of the DPLL-based #SAT solver SharpSAT (Thurley 2006) which uses a state-of-the-art search procedure. We cast the problem as a Markov Decision Process (MDP) in which the task is to select the best literal to branch on next. We use a Graph Neural Network (GNN) (Scarselli et al. 2009) to represent the particular component of the residual formula the solver is currently working on. The model is trained end-to-end via Evolution Strategies (ES), with the objective of minimizing the mean number of branching decisions required to solve instances from a given distribution of problems. In other words, given a training set of instances drawn from a problem distribution, the aim is to automatically tailor the solver’s branching decisions for better performance on unseen problems of that distribution.

We found that our technique, which we call Neuro#, can generalize not only to unseen problem instances of similar size but also to much larger instances than those seen at training time. Furthermore, despite Neuro#’s considerable runtime overhead from querying the learnt model, on some problem domains Neuro# can achieve orders-of-magnitude improvements in the solver’s wall-clock runtime. This is quite remarkable in the context of prior related work (Yolcu and Póczos 2019; Selsam and Björner 2019; Balcan et al. 2018; Gasse et al. 2019; Khalil et al. 2016; Hanksnecht, Joormann, and Stiller 2018; Lederman et al. 2020), where using ML to improve combinatorial solvers has at best yielded modest wall-clock time improvements (less than a factor of two), and positions this line of research as a viable path towards improving the practicality of exact model counters.1

1Our code and the extended version of the paper (with the appendix) are available at: github.com/NeuroSharp

*Equal Contribution

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
1.1 Related Work

The first successful application of machine learning to propositional satisfiability solvers was the portfolio-based SAT solver SATZilla (Xu et al. 2008). Equipped with a set of standard SAT solvers, a classifier was trained offline to map a given SAT instance to the solver best suited to solve it. Recent work has been directed along two paths: heuristic improvement (Selsam and Bjørner 2019; Kurin et al. 2019; Lederman et al. 2020; Yoku and Póczos 2019), and purely ML-based solvers (Selsam et al. 2019; Amizadeh, Matusevych, and Weimer 2019). In the former, a model is trained to replace a heuristic in a standard solver, thus the model is embedded as a module within the solver’s framework and guides the search process. In the latter, 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 where an exact answer is required, e.g., in formal verification. 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. More concretely, our work is similar to (Yoku and Póczos 2019), who used Reinforcement Learning (RL) and GNNs to learn variable selection heuristics for the local search-based SAT solver WalkSAT (Selman, Kautz, and Cohen 1993). Local search cannot be used to solve exact #SAT, and empirically (Yoku and Póczos 2019) obtained only modest improvements on much smaller problems. Our method is also related to (Lederman et al. 2020) and (Gasse et al. 2019), where similar techniques were used in solving quantified Boolean formulas and mixed integer programs, respectively.

Recently, Abdou, Ceylan, and Lukasiewicz (2020) trained GNNs as a stand-alone approximate solver for Weighted DNF Model Counting (#DNF). However, approximating #DNF is a much easier problem: it has a fully polynomial randomized approximation scheme (Karp, Luby, and Madras 1989). So the generalization to larger problem instances demonstrated in that paper is not comparable to the scaling on exact #SAT our approach achieves.

2 Background

#SAT A propositional Boolean formula consists of a set of propositional (true/false) variables composed by applying the standard operators “and” (∧), “or” (∨) and “not” (¬). A literal is any variable v or its negation ¬v. A clause is a disjunction of literals ∨i=1ni li. A clause is 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 L(φ) and C(φ), respectively. We 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 π(v) = 1 or when ℓ = ¬v and π(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 ℓ′ = ¬ℓ ∨ ℓ′′ is in φ, then every satisfying assignment must also make ℓ′ true since ¬ℓ ∨ ℓ and ℓ′′ 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 variable and branches on it by setting it to true or false. When a literal ℓ is set to true the formula φ can be reduced by finding all forced literals using UP (this includes ℓ), 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 ⊆ C(φ) is a subset of φ’s clauses that is disjoint from its complement C(φ) − C. A formula φ can be efficiently broken up into a maximal number of disjoint components C1, . . . , Ck. 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: COUNT(φ) = ∏k i=1 COUNT(Ci). In contrast, solving the formula as a monolith takes 2Θ(n) where n is the number of variables in the input formula, and so is not efficient for large n.

A formula φ or component Ci can be represented by a literal-clause incidence graph (LIG). This graph contains a node for every clause and every literal of φ (i.e., v and ¬v for every variable v of φ). An edge connects a clause node nc and a literal node nl if and only if ℓ ∈ c. Note that if Ci is a component of φ, then Ci’s LIG will be a disconnected sub-graph of φ’s LIG (Figure 1).

Both exact (Thurley 2006; Sang et al. 2004; Oztok and Darwiche 2015), approximate (Chakraborty et al. 2014; Meel and Akshay 2020), and probabilistically correct (Sharma et al. 2019) model counters have been developed. In this paper, we focus on exact model counting using the SharpSAT solver (Thurley 2006). SharpSAT and other modern exact #SAT solvers are based on DPLL (Davis, Logemann, and Loveland 1962) augmented with clause learning and component caching ( Bacchus, Dalmao, and Pitassi 2003a, 2009). A simplified version of the algorithm with the clause learning parts omitted is given in Algorithm 1.

The #DPLLCAKE 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 φℓ is
which is a linear combination of a heuristic aimed at making
the graph to a vector space. Let
\( h^{(k)}_v \) from iteration
\( k \) be an undirected
Markov Decision Process
(GNNs) are a class of zeroth order black-box optimization algorithms (Beyer and Schwefel 2002; Wierstra et al. 2014). 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 (Salimans et al. 2017): Let \( f : \Theta \rightarrow \mathbb{R} \) denote the fitness function for a parameter space \( \Theta \), e.g., in an RL environment, \( f \) computes the stochastic episodic reward of a policy \( \pi_\theta \). To produce the new generation of parameters of size \( n \), (Salimans et al. 2017) uses an additive Gaussian noise with standard deviation \( \sigma \) to perturb the current generation: \( \theta_{t+1} = \theta_t + \sigma \epsilon^{(i)} \), 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,
\[
\theta_{t+1} = \theta_t + \eta \nabla_{\theta} \mathbb{E}_{\theta \sim \mathcal{N}(\theta_t, \sigma^2 I)}[f(\theta)] \\
\approx \theta_t + \eta \frac{1}{n\sigma} \sum_{i=1}^{n} f(\theta^{(i)}_{t+1}) \epsilon^{(i)},
\]
where \( \eta \) 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 (Salimans et al. 2017) to scale the reward function and reduce the variance of the gradient, respectively.

3 Method

We formalize the problem of learning the branching heuristic for #DPLLCache as a Markov Decision Process (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 \( \phi_t \) that the solver is operating on, and performs an action from the action space \( A_t = \{l \in \mathcal{L}(\phi_t)\} \). 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:
\[
R(s) = \begin{cases} 
1 & \text{if } s \text{ is a terminal state with “instance solved” status} \\
-r_{\text{penalty}} & \text{otherwise}
\end{cases}
\]
If not finished, episodes are aborted after a predefined max number of steps, without receiving the termination reward.

**Training with Evolution Strategies** With the objective 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 Vemula, Sun, and Bagnell (2019) show, 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 Salimans et al. (2017) for optimizing our agent.

**SharpSAT Components as GNNs** As the task for the neural network agent is to pick a literal \( l \) from the component \( \phi \), we opt for a LIG representation of the component (see Section 2) and we use GNNs to compute a literal selection heuristic based on that representation. In detail, given the LIG \( G = (V, E) \) of a component \( \phi \), we denote the set of clause nodes as \( C \subseteq V \), and the set of literal nodes as \( L \subseteq V \). \( V = C \cup L \). The initial vector representation is denoted by \( h^{(0)} \) for each clause \( c \in C \) and \( h^{(0)} \) for each literal \( l \in L \), both of which are learnable model parameters. We run the following message passing steps iteratively:

- **Literal to Clause (L2C):**
  \[
  h^{(k+1)}_c = \mathcal{A}\left(h^{(k)}_c, \sum_{l \in L} [h^{(k)}_l; W^{(k)}_C] \right), \quad \forall c \in C,
  \]

- **Clause to Literal (C2L):**
  \[
  h^{(k+1)}_l = \mathcal{A}\left(h^{(k)}_l, \sum_{c \in C} [h^{(k)}_c; W^{(k)}_L] \right), \quad \forall l \in L,
  \]

where \( \mathcal{A} \) is a nonlinear aggregation function, parameterized by \( W^{(k)}_C \) for clause aggregation and \( W^{(k)}_L \) for literal aggregation at the \( k^{\text{th}} \) iteration. Following Selsam et al. (2019); Lederman et al. (2020), to ensure negation invariance (i.e., that the graph representation is invariant under literal negation), we concatenate the literal representations corresponding to the same variable \( h^{(k)}_l \), \( h^{(k)}_l \) when running L2C message passing. After \( K \) iterations, we obtain a \( d \)-dimensional vector representation for every literal in the graph. We pass each representation through a policy network, a Multi-Layer Perceptron (MLP), to obtain a score, and we choose the literal with the highest score. Recently, Xu et al. (2019) developed a 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 \( \mathcal{A} \). Specifically, \( \mathcal{A}(x, y; W) = \text{MLP}((1+\epsilon)x + y; W) \), where \( \epsilon \) is a hyperparameter.

**Sequential Semantics** Many problems, such as dynamical systems and bounded model checking, are iterative in nature, with a distinct temporal dimension to them. In 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 \( (s_t, s_{t+1}) \). Normally, all temporal information is lost in reduction to CNF. However, with a learning-based approach, the time-step feature from the original problem can be readily incorporated as an 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 5.

**Engineering Trade-offs and Constraints** Directly training on challenging #SAT instances of enormous size is computationally infeasible. We tackle this issue by training Neuro# on small instances of a problem (fast rollouts) and relying on generalization to solve the more challenging instances from the same problem domain. Thus the main requirement is the availability of the generative process that lets us sample problems with desired level of difficulty. Access to such generative process is not an unreasonable assumption in industry and research.

Although reducing the number of branching steps is itself interesting, to beat SharpSAT in wall-clock time, Neuro#’s lead needs to be wide enough to justify the imposed overhead of querying the GNN. Since the time-per-step ratio is relatively constant, for the method to be effective it is desirable that the step count reduction be superlinear, meaning it becomes more effective compared to the vanilla heuristic the larger the problem becomes.

**4 Data Generation**

Our goal was to evaluate our method on more structured and much larger instances than the small random instances typically used in other related works (Yolcu and Póczos 2019; Selsam et al. 2019; Kurin et al. 2019). To that end, we searched SAT and planning benchmarks for problems whose generative processes were publicly available or feasible to implement. To test the versatility of our method, we made sure that these problems cover a diverse set of domains: sudoku, blocked n-queens, cell (combinatorial); sha-1 preimage attack (cryptography); island, grid_world (planning), bv_expr, it_expr (circuits). For brevity, we explain two of the problems that we will use in later sections to discuss the behaviour of our trained model and provide a more detailed description of the other datasets and their generation process in Appendix B:

- **cell(R, n, r):** 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 2a shows the evolution grid of rules 9, 35 and 49 for 20 iterations.
- **Reversing Elementary Cellular Automata:** Given a randomly sampled state \( T \), compute the number of initial
states $I$ that would lead to that terminal state $T$ in $r$ applications of $R$, i.e., $|\{I : R^r(I) = T\}|$. The entire $r$-step evolution grid is encoded by mapping each cell to a Boolean variable $(n \times r$ in total). The clauses impose the constraints between cells of consecutive rows as given by $R$. The variables corresponding to $T$ (last row of the evolution grid) are assigned as unit clauses. This problem was taken from SATCOMP 2018 (Heule, Järvisalo, and Suda 2018).

- grid wrld($s, t$): This bounded horizon planning problem from (Vazquez-Chanlatte et al. 2018; Vazquez-Chanlatte, Rabe, and Seshia 2019) 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”. We randomly sample a grid world of size $s$ and a starting position $I$ for an agent. We encode to CNF the problem of counting the number of trajectories of length $t$ beginning from $I$ that always avoid lava.

5 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 runtime substantially? 4) Interpretation: Does the sequence of actions taken by the model exhibit any discernible pattern at the problem level? Our baseline in all comparisons is SharpSAT’s heuristic. Also, to make sure that our model’s improvements are not trivially attainable without training we tested a Random policy that simply chooses a literal uniformly at random. We also 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.

The grid wrld, being a problem of an iterative nature (i.e., steps in the planning problem), was a natural candidate for testing our hypothesis regarding the effect of adding the “time” feature of Section 3, so we report the results for grid wrld with that feature included and later in this section we perform an ablation study on that feature.

Experimental Setup 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 and 48 perturbations ($\sigma = 0.02$). With mirror sampling, we obtained in total $96 = 48 \times 2$ perturbations. For each

<table>
<thead>
<tr>
<th>Dataset</th>
<th># vars</th>
<th># clauses</th>
<th>Random</th>
<th>SharpSAT</th>
<th>Neuro#</th>
</tr>
</thead>
<tbody>
<tr>
<td>sudoku(9, 25)</td>
<td>182</td>
<td>3k</td>
<td>338</td>
<td>220</td>
<td>195(1.1x)</td>
</tr>
<tr>
<td>n-queens(10, 20)</td>
<td>100</td>
<td>1.5k</td>
<td>981</td>
<td>466</td>
<td>261(1.7x)</td>
</tr>
<tr>
<td>sha(1,28)</td>
<td>3k</td>
<td>13.5k</td>
<td>2,911</td>
<td>52</td>
<td>242(2.1x)</td>
</tr>
<tr>
<td>island(2, 5)</td>
<td>1k</td>
<td>34k</td>
<td>155</td>
<td>86</td>
<td>301(1.8x)</td>
</tr>
<tr>
<td>cell(9, 20, 20)</td>
<td>210</td>
<td>1k</td>
<td>957</td>
<td>370</td>
<td>184(2.0x)</td>
</tr>
<tr>
<td>cell(35, 128, 110)</td>
<td>6k</td>
<td>25k</td>
<td>867</td>
<td>335</td>
<td>198(1.8x)</td>
</tr>
<tr>
<td>cell(49, 128, 110)</td>
<td>6k</td>
<td>25k</td>
<td>843</td>
<td>338</td>
<td>206(1.6x)</td>
</tr>
<tr>
<td>grid wrld(10, 5)</td>
<td>329</td>
<td>967</td>
<td>220</td>
<td>195</td>
<td>666(3.0x)</td>
</tr>
<tr>
<td>bv_expr(5, 4, 8)</td>
<td>90</td>
<td>220</td>
<td>1,316</td>
<td>328</td>
<td>205(1.6x)</td>
</tr>
<tr>
<td>it_expr(2, 2)</td>
<td>82</td>
<td>264</td>
<td>772</td>
<td>412</td>
<td>266(1.5x)</td>
</tr>
</tbody>
</table>

Table 1: Neuro# generalizes to unseen i.i.d. test problems often with a large margin compared to SharpSAT.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># vars</th>
<th># clauses</th>
<th>Random</th>
<th>SharpSAT</th>
<th>Neuro#</th>
</tr>
</thead>
<tbody>
<tr>
<td>sudoku(16, 105)</td>
<td>1k</td>
<td>31k</td>
<td>7,654</td>
<td>2,373</td>
<td>2,300(1.03x)</td>
</tr>
<tr>
<td>n-queens(12, 20)</td>
<td>144</td>
<td>2.6k</td>
<td>31,728</td>
<td>12,372</td>
<td>6,272(1.9x)</td>
</tr>
<tr>
<td>sha(1,40)</td>
<td>5k</td>
<td>25k</td>
<td>15k</td>
<td>387</td>
<td>83(4.6x)</td>
</tr>
<tr>
<td>island(2, 8)</td>
<td>1.5k</td>
<td>73.5k</td>
<td>1,335</td>
<td>193</td>
<td>46(4.1x)</td>
</tr>
<tr>
<td>cell(9, 40, 40)</td>
<td>820</td>
<td>4k</td>
<td>39,000</td>
<td>53,349</td>
<td>42,325(1.2x)</td>
</tr>
<tr>
<td>cell(35, 192, 128)</td>
<td>12k</td>
<td>49k</td>
<td>36,186</td>
<td>21,166</td>
<td>1,668(12.5x)</td>
</tr>
<tr>
<td>cell(35, 256, 200)</td>
<td>25k</td>
<td>102k</td>
<td>41,589</td>
<td>26,460</td>
<td>2,625(10x)</td>
</tr>
<tr>
<td>cell(35, 348, 280)</td>
<td>48k</td>
<td>195k</td>
<td>54,113</td>
<td>33,820</td>
<td>2,938(11.5x)</td>
</tr>
<tr>
<td>cell(49, 192, 128)</td>
<td>12k</td>
<td>49k</td>
<td>35,957</td>
<td>24,992</td>
<td>1,829(13.6x)</td>
</tr>
<tr>
<td>cell(49, 256, 200)</td>
<td>25k</td>
<td>102k</td>
<td>47,341</td>
<td>30,817</td>
<td>2,276(13.5x)</td>
</tr>
<tr>
<td>cell(49, 348, 280)</td>
<td>48k</td>
<td>195k</td>
<td>53,779</td>
<td>37,345</td>
<td>2,671(13.9x)</td>
</tr>
<tr>
<td>grid wrld(10, 10)</td>
<td>740</td>
<td>2k</td>
<td>22,054</td>
<td>13,661</td>
<td>367(37x)</td>
</tr>
<tr>
<td>grid wrld(10, 12)</td>
<td>2k</td>
<td>6k</td>
<td>100k≤</td>
<td>93,093</td>
<td>1,320(71x)</td>
</tr>
<tr>
<td>grid wrld(10, 14)</td>
<td>2k</td>
<td>7k</td>
<td>100k≤</td>
<td>100k≤</td>
<td>2,234(–)</td>
</tr>
<tr>
<td>grid wrld(12, 14)</td>
<td>2k</td>
<td>8k</td>
<td>100k≤</td>
<td>100k≤</td>
<td>2,782(–)</td>
</tr>
<tr>
<td>bv_expr(7, 4, 12)</td>
<td>187</td>
<td>474</td>
<td>35,229</td>
<td>5,865</td>
<td>2,139(27x)</td>
</tr>
<tr>
<td>it_expr(2, 4)</td>
<td>162</td>
<td>510</td>
<td>51,375</td>
<td>7,894</td>
<td>2,635(3x)</td>
</tr>
</tbody>
</table>

Table 2: Neuro# generalizes to much larger problems than what it was trained on, sometimes achieving orders of magnitude improvements over SharpSAT.
We used the Adam optimizer (Kingma and Ba 2015) with weight decay of 0.005. All episodes, unless otherwise mentioned, were capped at 32 steps during training and 100k during testing. The agent received a negative reward of \( r_{\text{penalty}} = 10^{-4} \) at each step. We used the default hyperparameters, a learning rate of \( \eta = 0.01 \) and a weight decay of 0.005.

GNN messages were implemented by an MLP with ReLU non-linearity. The size of literal and clause embeddings were 32 and the dimensionality of C2L (resp. L2C) messages was \( 32 \times 32 \times 32 \) (resp. \( 64 \times 32 \times 32 \)). We used \( T = 2 \) message passing iterations and final literal embeddings were 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 was 33 instead of 32. The initial (\( T = 0 \)) embeddings of both literals and clauses were trainable model parameters.

5.1 Results

I.I.D. Generalization Table 1 summarizes the results of the i.i.d. generalization over the problem domains of Section 4. 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 and on cell, by an average factor of 1.8 over the three cellular rules.

Upward Generalization We created instances of larger sizes (up to an order of magnitude more clauses and variables) for each of the datasets in Section 4. 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 in Table 2 show 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 cell(49) and grid\_wrld by plotting the percentage of the problems solved within a number of steps. The superlinear gaps get more pronounced once we remove the cap of 10k steps, i.e., let the episodes run to completion. In that case, on grid\_wrld(10, 12), Neuro\# took an average of 1,320 branching decisions, whereas SharpSAT took 809,408 (613x improvement).

Wall-Clock Improvement Given the scale of improvements on the upward generalization benchmark, in particular cell(49) and grid\_wrld, we measured the runtime of Neuro\# vs. SharpSAT on those datasets (Figure 4). On both problems we observe that as Neuro\# widens the gap in the number of steps, it manages to beat SharpSAT in wall-clock. Note that the query overhead could still be greatly reduced in our implementation through GPU utilization, loading the model in the solver’s code in C++ instead of making out-of-process calls to Python, etc.

5.2 Model Interpretation

Formal analysis of the learned policy and its performance improvements is difficult, however we can form some high-level conjectures regarding the behaviour of Neuro\# by how well it decomposes the problem. Here we look at cell. The reason is that this problem has a straightforward encoding that directly relates the CNF representation to an easy-to-visualize evolution grid that coincides with the standard representation of Elementary Cellular Automata. The natural way to decompose this problem is to start from the known state \( T \) (bottom row) and to “guess” the preimage, row by row from the bottom up through variable assignment. Different preimages can
be computed independently upwards, and indeed, this is how a human would approach the problem. Heat maps in Figure 2 (c & 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. The hotter coloured cells indicate that, on average, the corresponding variable is 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 2 (b). Notice that in cell(35&49) the simplification shatters the problem space into few small components (dark triangles), while in 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 cell(9) and in a less pronounced way in cell(35&49). Moreover, as more clearly seen in the heatmap for the larger problem in Figure 5, Neuro# actually branches early according to the pattern of the rule.

5.3 Ablation Study

We tested the degree to which the “time” feature contributed to the upward generalization performance of grid_wrld. We compared three architectures with SharpSAT as the baseline: 1. GNN: The standard architecture proposed in Section 3, 2. GNN+Time: Same as GNN but with the variable embeddings augmented with the “time” feature and 3. Time: Where no variable embedding is computed and only the “time” feature is fed into the policy network. As depicted in Figure 6, we discovered that “time” 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.

5.4 Discussions

We observed varying degrees of success on different problem families. This raises the question of what traits make a problem more amenable for improvement via Neuro#. 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 into smaller components either by the initial simplification (e.g., UP) or after a few variable assignments, there is a high chance that the model fits to such distribution of components. If larger problems of the same domain also break down into similar component structures, then Neuro# can generalize well on them. This explains why sequential problems like grid_wrld benefit from our method, as they are naturally decomposable into similar iterations and addition of the “time” feature demarcates the boundaries between iterations even more clearer.

Recently (Bliem and Järvisalo 2019) showed branching according to the centrality scores of variable nodes of the CNF graph leads to significant performance improvements. We obtained their centrality enhanced version of SharpSAT and compared it with Neuro# trained on specific problem family. We found that, although centrality enhanced SharpSAT, Neuro# retained its orders of magnitude superiority over it. This indicates that whatever structure Neuro# is exploiting from the graph, it is not exclusively centrality. Also we compared the performance of Neuro# against the state-of-the-art probabilistic model counter GANAK. Note that this solver performs the easier task of providing a model count that is only probably correct within a given error tolerance. Thus to make the comparison more fair we set the error tolerance of GANAK to a small value of $10^{-3}$ and observed that its performance was again inferior to Neuro#. An interesting future direction would be to investigate if our method could also be used to customize GANAK’s heuristics. Appendix C includes cactus plots for every problem in our dataset along with comparison to both centrality and GANAK.

6 Conclusion

We studied the feasibility of enhancing the branching heuristic in propositional model counting via learning. We used solver’s branching steps 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 runtime overhead of querying the model. This is an exciting first step and it positions this line of research as a potential path towards building better model counters and thus broadening their application horizon.
Acknowledgements

Dr. Seshia’s contribution to this work was funded by DARPA contract FA8750-20-C-0156 (LOGiCS), NSF grant CNS-1545126 (VeHiCaL), Microsoft and Intel. Gil Lederman’s work was partially funded by NSF award CNS-1836601.

References


