Neuro#: A Distribution Tailored Model Counter

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

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

Abstract

We present `Neuro#`, an approach for learning branching heuristics to improve the performance of the exact model counter `SharpSAT` on instances from a given family of problems. 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 can be translated into model counting problems to be solved by #SAT solvers. Alas, exact #SAT solvers, such as `SharpSAT`, are often not scalable to industrial size instances. We experimentally show that `Neuro#` solves similarly distributed held-out instances in fewer steps and generalizes to much larger instances from the same problem family. In addition to step count improvements, `Neuro#` can achieve orders of magnitude wall-clock speedups over `SharpSAT` on larger instances in certain problem families, despite the runtime overhead of querying the learnt model.

1 Introduction

Propositional model counting is the problem of counting the number of satisfying solutions to a Boolean formula [12]. When the Boolean formula is expressed in conjunctive normal form (CNF), this problem is known as the #SAT problem. Modern exact #SAT solvers are based on the DPLL algorithm [7] and have been successfully applied to solve certain problems, e.g., inference in Bayes Nets [4, 20, 22, 9] and bounded-length probabilistic planning [8]; 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` [29] 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) [24] to operate on 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 [34, 26, 6, 10, 16, 13, 19], 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.

Related Work Recent work in applying machine learning to propositional satisfiability solvers has been directed along two paths: *heuristic improvement* [26, 18, 19, 34], and purely ML-based solvers [27, 2]. 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. 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 [34], who used Reinforcement Learning (RL) and GNNs to learn variable selection heuristics for the *local search-based* SAT solver `WalkSAT` [25]. Our method is also related to [19] and [10], where similar techniques were used in solving quantified Boolean formulas and mixed integer programs, respectively. The only related model counting work is Abboud et al. [1] where a GNN was trained 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 [15]. 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

The #SAT problem for a propositional Boolean formula ϕ in *Conjunctive Normal Form* (CNF) is to compute the number of *satisfying assignments*. A satisfying assignment for any formula ϕ is a mapping of its variables to $\{0, 1\}$ (**false/true**) such that all of its clauses are satisfied. If ℓ is a unit clause (a clause that contains a single literal) 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 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).

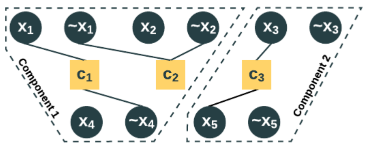


Figure 1: The Literal-Clause Incidence Graph of the formula: $(x_1 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_5)$.

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. Each component can be solved separately and their counts multiplied: COUNT$(\phi) = \prod_{i=1}^{k}$ 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 is not efficient for large n.

A formula ϕ or component C_i can be represented by a *literal-clause incidence graph* (LIG); see Figure 1. This graph contains a node for every clause and every literal of ϕ (i.e., v and $\neg v$ for every variable v of ϕ). An edge connects a clause node n_c and a literal node n_ℓ if and only if $\ell \in c$. Note that if C_i is a component of ϕ, then C_i's LIG will be a disconnected sub-graph of ϕ's LIG.

`SharpSAT`, like other modern exact #SAT solvers is based on DPLL [7] augmented with *clause learning* and *component caching* [3, 5]. This algorithm works on one component at a time. If that component's model count has already been cached, `SharpSAT` returns the cached value. Otherwise it selects a literal to branch on and computes the model count under each value of this literal. The sum of these two counts is the model count of the passed component ϕ, and so is stored in the cache. Through these variable assignments and subsequent unit propagations, a component is eventually divided into its sub-components which as indicated before, are independently solved and the product of their 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 [23] 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.

3 Method

We formalize the problem of learning the branching heuristic 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. 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. To that end, we give the agent a reward of 1 upon successfully solving the instance and for every step (i.e., branching decision) taken we penalize the agent by giving it the negative reward of $-r_{penalty}$. The episodes are aborted after a predefined maximum number of steps, in which case the agent does not receive the termination reward. For our experiments we set the $r_{penalty} = 10^{-4}$ and we cut off the episodes at 10^5 steps.

A major issue with the above formulation is that the size of the action space $|\mathcal{A}_t|$, as well as the horizon of the episodes can get quite large. As Vemula et al. [32] 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. A *parameter-space* exploration algorithm like ES however, is independent of these two factors. Therefore, we choose to use a version of ES proposed by Salimans et al. [21] for optimizing our agent.

To represent the component that the agent is working on at any step we use LIG and utilize a GNN based on that LIG to compute a literal selection heuristic. In detail, given the LIG $G = (V, E)$ of a component ϕ, we denote the set of clause nodes as $C \subset V$, and the set of 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 of which are learnable model parameters. We run the following message passing steps iteratively:

$$\text{Literal to Clause (L2C):} \quad h_c^{(k+1)} = \mathcal{A}\Big(h_c^{(k)}, \sum_{l \in c} [h_l^{(k)}, h_{\bar{l}}^{(k)}]; W_C^{(k)}\Big), \quad \forall c \in C,$$

$$\text{Clause to Literal (C2L):} \quad h_l^{(k+1)} = \mathcal{A}\Big(h_l^{(k)}, \sum_{c, l \in c} h_c^{(k)}; W_L^{(k)}\Big), \quad \forall l \in L,$$

where $\mathcal{A}$ is a nonlinear aggregation function, parameterized by $W_C^{(k)}$ for clause aggregation and $W_L^{(k)}$ for literal aggregation at the k^{th} iteration. To ensure that the graph representation is invariant under literal negation, we concatenate the literal representations corresponding to the same variable $h_l^{(k)}, h_{\bar{l}}^{(k)}$ when running L2C 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 we choose the literal with the highest score. We choose the GNN architecture *Graph Isomorphism Network* (GIN) Xu et al. [33] for the parameterization of $\mathcal{A}$. Architectural details are provided in Appendix B.

4 Experiments

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, we need to test: 1. If a model trained on instances from a given distribution can generalize to unseen instances of the same distribution (*I.I.D. Generalization*); and 2. If a model trained on small instances can generalize directly on larger instances of the same problem family (*Upward Generalization*).

We searched SAT and planning benchmarks for problems whose generative processes were publicly available or feasible to implement, so that we could sample instances of varying sizes for training

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`. Episodes are capped at 100k steps.

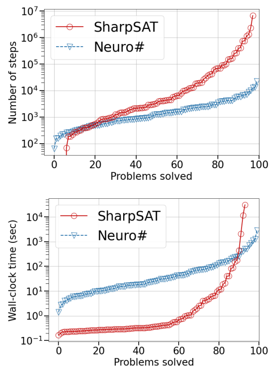
	i.i.d.					Upward Generalization			
	# vars	# clauses	`SharpSAT`	`Neuro#`		# vars	# clauses	`SharpSAT`	`Neuro#`
sudoku(9, 25)	182	3k	220	**195(1.1x)**	sudoku(16, 105)	1k	31k	2,373	**2,300 (1.03x)**
n-queens(10, 20)	100	1.5k	466	**261(1.7x)**	n-queens(12, 20)	144	2.6k	12,372	**6,272 (1.9x)**
sha-1(28)	3k	13.5k	52	**24(2.1x)**	sha-1(40)	5k	25k	387	**83 (4.6x)**
island(2, 5)	1k	34k	86	**30(1.8x)**	island(2, 8)	1.5k	73.5k	193	**46 (4.1x)**
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$\leq$	**2,234 (–)**
					grid_wrld(12, 14)	2k	8k	100k$\leq$	**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)**
it_expr(2, 2)	82	264	412	**266(1.5x)**	it_expr(2, 4)	162	510	7,894	**2,635 (3x)**

and testing. We chose our problems from a diverse set of domains: sudoku, blocked n-queens, cell *(combinatorial)*; sha-1 preimage attack *(cryptography)*; island, grid_wrld (*planning*), bv_expr, it_expr *(circuits)*. More details about the datasets and our experimental setup are in the Appendix.

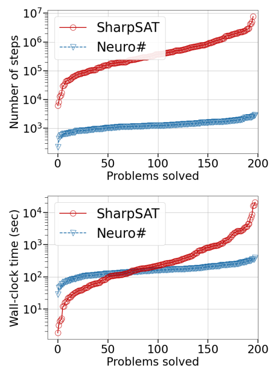
4.1 Results

Table 1 summarizes the results of the both the i.i.d. and upward generalization. We report the average number of branching steps on the test set. `Neuro#` outperformed the baseline across all datasets, and as we increase the instance size the gains become quite substantial compared to `SharpSAT`.

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 2). 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.



(a) cell(49, 256, 200)



(b) grid_wrld(10, 12)

Figure 2: Cactus plots comparing `Neuro#` to `SharpSAT` on cell and grid_wrld. 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 (top) and seconds (bottom).

5 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.

6 Broader Impact

This work impacts the performance of reasoning components that can be used in the construction of larger systems. As such its societal impacts will be dependent on the impact of the larger systems utilizing these components. Hopefully, the work will be used by others in the construction of larger systems that positively impact society.

Acknowledgement 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

[1] Ralph Abboud, İsmail İlkan Ceylan, and Thomas Lukasiewicz. Learning to reason: Leveraging neural networks for approximate DNF counting. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference*, pages 3097–3104. AAAI Press, 2020. URL `https://aaai.org/ojs/index.php/AAAI/article/view/5705`.

[2] 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`.

[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*. Morgan Kaufmann, 2003. `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=909&proceeding_id=19`.

[5] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Solving #SAT and Bayesian Inference with Backtracking Search. *J. Artif. Intell. Res.*, 34, 2009. doi: 10.1613/jair.2648. `https://doi.org/10.1613/jair.2648`.

[6] 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*. PMLR, 2018. `http://proceedings.mlr.press/v80/balcan18a.html`.

[7] 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`.

[8] 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`.

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

[10] 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*, 2019. `http://papers.nips.cc/paper/9690-exact-combinatorial-optimization-with-graph-convolutional-neural-networks`.

[11] Tomas Geffner and Hector Geffner. Compact policies for fully observable non-deterministic planning as SAT. In Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan, editors, *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, pages 88–96. AAAI Press, 2018.

[12] 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`.

[13] 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`.

[14] 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`.

[15] Richard M. Karp, Michael Luby, and Neal Madras. Monte-carlo approximation algorithms for enumeration problems. *J. Algorithms*, 10(3):429–448, 1989. doi: 10.1016/0196-6774(89)90038-2. URL `https://doi.org/10.1016/0196-6774(89)90038-2`.

[16] 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`.

[17] 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`.

[18] 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`.

[19] 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`.

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

[21] 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`.

[22] 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. AAAI Press / The MIT Press, 2005. http://www.aaai.org/Library/AAAI/2005/aaai05-075.php.

[23] 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.

[24] 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. https://doi.org/10.1109/TNN.2008.2005605.

[25] 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. https://doi.org/10.1090/dimacs/026/25.

[26] 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.

[27] 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.

[28] Volodymyr Skladanivskyy. Cgen. https://github.com/vsklad/cgen, 2018.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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 More on Datasets

Here we provide more detailed information about the problems used in the main document:

- $\text{sudoku}(n, k)$: Randomly generated partially filled $n \times n$ Sudoku problems ($n \in \{9, 16\}$) with multiple solutions, where k is the number of revealed squares.

  Sudoku problems are typically designed to have only one solution but as our goal is to improve a model counter, we relaxed this requirement to count the number of solutions instead.
- $\text{n-queens}(n, k)$: Blocked N-Queens problem of size n with k randomly blocked squares on the chess board. This is a standard SAT problem and the task is to count the number of ways to place the n queens on the remaining $n^2 - k$ squares such that no two queens attack each other.
- $\text{sha-1}(n, k)$: SHA-1 preimage attack of randomly generated messages of size n with k randomly chosen bits fixed. This problem was taken from SATRACE 2019 and we used the CGEN tool [28] to generate our instances.
- $\text{island}(n, m)$: This dataset was introduced by [11] as a *Fully-Observable Non-Deterministic* (FOND) planning problem. There are two grid-like islands of size $n \times n$, each connected by a bridge. The agent is placed at a random location in island 1 and the goal is for it to go to another randomly selected location in island 2. The short (non-deterministic) way is to swim from island 1 to 2, where the agent may drown, and the long way is to go to the bridge and cross it. Crossing the bridge is only possible if no animals are blocking it, otherwise the agent has to move the animals away from the bridge before it can cross it. The m animals are again randomly positioned on the two grids. We used the generative process of [11] to encode compact policies for this task in SAT.
- $\text{cell}(R, n, r)$: 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). 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., $\big|\{I : R^r(I) = T\}\big|$. 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 [14].
- $\text{grid\_wrld}(s, t)$: This bounded horizon planning problem from [30, 31] 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.
- $\text{bv\_expr}(n, d, w)$: Randomly generated 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 $(+, -, \wedge, \vee, \neg, \text{XOR}, |\cdot|)$. The problem is to count the number of integer solutions to the resulting relation in $([0, 2^w] \cap \mathbb{Z})^n$.
- $\text{it\_expr}(s, i)$: Randomly generated arithmetic expression circuits of size s (word size fixed at 8). Effectively implementing a function with input and output of a single word. This function is composed i times. We choose a random word c, and count the number of inputs such that the output is less than c. Formally, if the random circuit is denoted by f, we compute $|\{x | f^i(x) < c\}|$.

B Architecture Details

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 perturbation, we ran the agent on the 8 formulas (in parallel), to a total of $768 = 96 \times 8$ episodes per parameter update. All episodes, unless otherwise mentioned, were capped at 1k steps during training and 100k during testing. The agent received a negative reward of $r_{penalty} = 10^{-4}$ at each step. We used the Adam optimizer [17] with 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. The initial ($T = 0$) embeddings of both literals and clauses were trainable model parameters.

Hardware Infrastructure We used a small cluster of 3 nodes each with an AMD Ryzen Threadripper 2990WX processor with 32 cores (64 threads) and 128GB of memory. We trained for an average of 10 hours on a dataset of 1000 instances for each problem. For testing the wall-clock time we ran the problems sequentially, to avoid any random interference due to parallelism.

Range of Hyperparameters The model is relatively "easy" to train. The criteria for choosing the hyperparameters was the performance of generalization on i.i.d test set, i.e, lowest possible average number of branching decisions. Once calibrated on the first dataset ($\mathrm{cell}(35)$), we were able to train all models on all datasets without further hyperparameters tuning. The minimal number of episodes per optimization step that worked was 12. We tested a few different GNN architectures, and none was clearly superior over the others. We also varied the number of GNN message-passing iterations but going beyond 2 iterations had a negative effect (on i.i.d generalization) so we settled on 2.