Neuro#: A Distribution Tailored Model Counter

Pashootan Vaezipoor[†] Gil Lederman[‖] Yuhuai Wu[†‡] Chris J. Maddison[†‡] Roger Grosse[†‡] Sanjit A. Seshia[‖] Fahiem Bacchus[†]

[†]Dept of Computer Science, University of Toronto [‖]Department of Electrical Engineering and Computer Sciences, UC Berkeley [‡]Vector Institute

[†]{pashootan, ywu, cmaddis, rgrosse, fbacchus}@cs.toronto.edu, [‖]{gilled, sseshia}@eecs.berkeley.edu

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.

Background

The #SAT problem for a propositional Boolean formula ϕ in *Conjunctive Normal Form* (CNF) is to compute the number of *satisfying assignments*.

Components

Let $\mathcal{L}(\phi)$ (resp. $\mathcal{C}(\phi)$) be the set of literals (resp. clauses) of ϕ. 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 $C_1, \ldots, C_k$. Each component can be solved separately and their counts multiplied:

$$\text{COUNT}(\phi) = \prod_{i=1}^{k} \text{COUNT}(C_i)$$

SharpSAT

`SharpSAT` is a modern exact #SAT solvers that is based on DPLL algorithm that's augmented with *clause learning* and *component caching*.

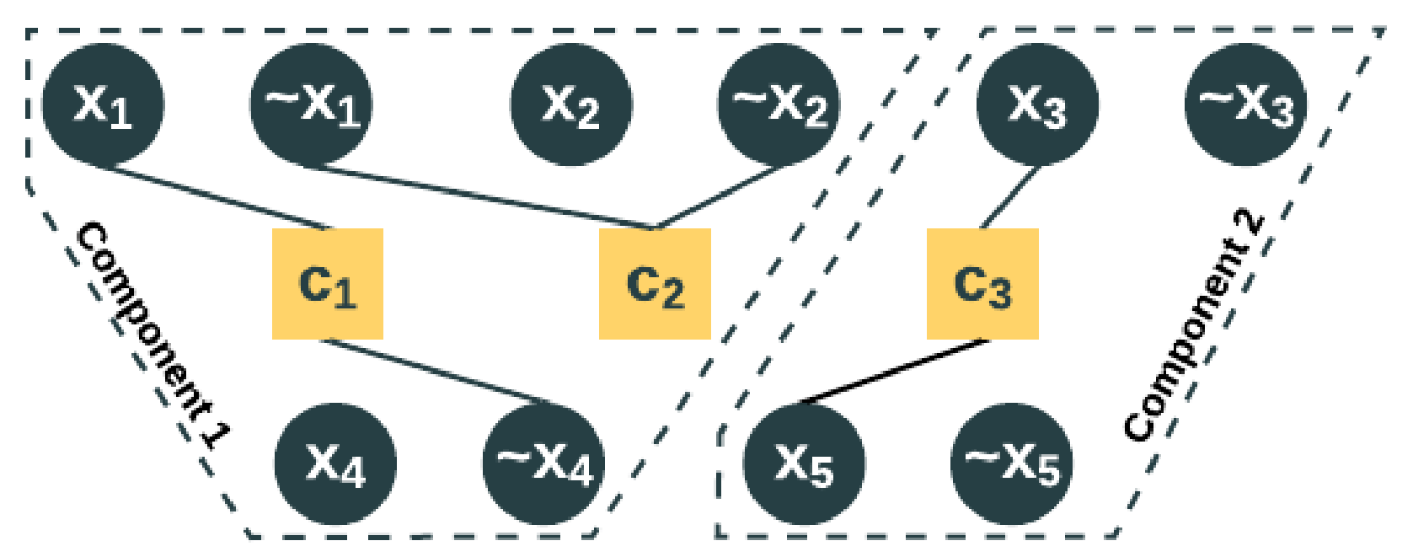
Algorithm 1 Component Caching DPLL

```
function #DPLLCache($\phi$)
    if InCache($\phi$) return CacheLookUp($\phi$)
    Pick a literal $\ell \in \mathcal{L}(\phi)$ ▷ We replace VSADS heuristic here
    $\#\ell$ = CountSide($\phi, \ell$)
    $\#\neg\ell$ = CountSide($\phi, \neg\ell$)
    AddToCache($\phi, \#\ell + \#\neg\ell$)
    return $\#\ell + \#\neg\ell$
end function
function CountSide($\phi, \ell$)
    $\phi_\ell$ = UnitPropagate($\phi, \ell$)
    if $\phi_\ell$ contains an empty clause return 0
    if $\phi_\ell$ contains no clauses then
        $k$ = # of unset variables
        return $2^k$
    end if
    $K$ = FindComponents($\phi_\ell$)
    return $\prod_{\kappa \in K}$ #DPLLCache($\kappa$)
end function
```

`SharpSAT`'s default heuristic for variable branching is VSADS. We try to learn a better heuristic.

Literal-Clause Incident Graph (LIG)

A formula ϕ or component C_i can be represented by *literal-clause incidence graph* (LIG). The LIG of formula $(x_1 \lor \neg x_4) \land (\neg x_1 \lor \neg x_2) \land (x_3 \lor x_5)$ and its components can be shown as:



Neuro#

#SAT as an MDP

We formalize the heuristic search for `SharpSAT` as a *Markov Decision Process* (MDP):

- $\mathcal{S}$: At each branching steps t the agent observes state s_t, consisting of the LIG of the component ϕ_t.
- $\mathcal{A}$: The agent performs an action from $\mathcal{A}_t = \{l \mid l \in \mathcal{L}(\phi_t)\}$, where $\mathcal{L}(\phi)$ is the set of literals of ϕ.
- $\mathcal{R}$: The objective is to ***reduce the number of branching decisions***, so the agent receives the reward of:

$$R(s) = \begin{cases} 1 & s \textit{ is a terminal state with "instance solved"}, \\ -10^{-4} & \textit{otherwise} \end{cases}$$

LIG as a Graph Neural Network (GNN)

At each step t we utilize a GNN to vectorize the LIG $G = (V, E)$ of a component ϕ_t. The initial vector representation is denoted by $h_c^{(0)}$ for each clause vertex c and $h_l^{(0)}$ for each literal vertex l in G. We run the following message passing steps iteratively:

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

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

where $\mathcal{A}$ is a nonlinear aggregation function, parameterized by $W_C^{(k)}$ and $W_L^{(k)}$ for clause and literal aggregation. After K iterations we pass each literal representation through a policy network (MLP), to obtain a score, and we choose the literal with the highest score to branch on.

Training

Problem 1: The size of action space $|\mathcal{A}_t|$ and the episode horizon gets quite large.
Solution: Replace an *action-space* exploration algorithm RL like Policy Gradient with a *parameter-space* exploration technique like *Evolution Strategies*.
Problem 2: Directly training the model on challenging problems is computationally infeasible.
Solution: Train on small instances of a problem (fast rollouts) and rely on generalization to solve the more challenging instances from the same problem domain.

Experiments

1. **I.I.D Generalization:** A model trained on instances from a given distribution can generalize to unseen instances of the same distribution.
2. **Upward Generalization:** A model trained on small instances can generalize directly on larger instances of the same problem family.

Upwards Generalization

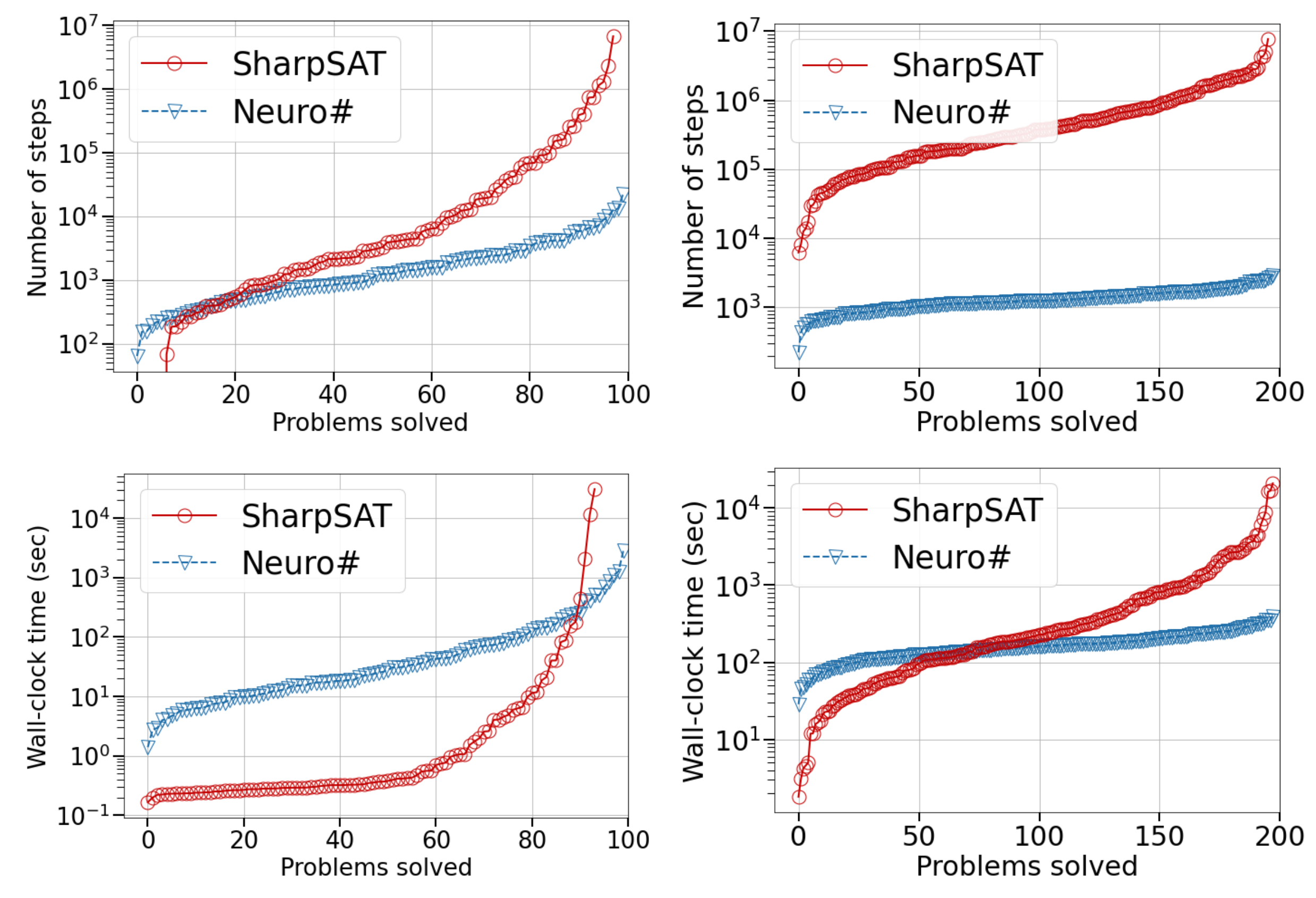
Dataset	# vars	# clauses	`Random`	`SharpSAT`	`Neuro#`
sudoku(16, 105)	1k	31k	7,654	2,373	**2,300 (1.03x)**
n-queens(12, 20)	144	2.6k	31,728	12,372	**6,272 (1.9x)**
sha-1(40)	5k	25k	15k	387	**83 (4.6x)**
island(2, 8)	1.5k	73.5k	1,335	193	**46 (4.1x)**
cell(9, 40, 40)	820	4k	39,000	53,349	**42,325 (1.2x)**
cell(35, 192, 128)	12k	49k	36,186	21,166	**1,668 (12.5x)**
cell(35, 256, 200)	25k	102k	41,589	26,460	**2,625 (10x)**
cell(35, 348, 280)	48k	195k	54,113	33,820	**2,938 (11.5x)**
cell(49, 192, 128)	12k	49k	35,957	24,992	**1,829 (13.6x)**
cell(49, 256, 200)	25k	102k	47,341	30,817	**2,276 (13.5x)**
cell(49, 348, 280)	48k	195k	53,779	37,345	**2,671 (13.9x)**
grid_wrld(10, 10)	740	2k	22,054	13,661	**367 (37x)**
grid_wrld(10, 12)	2k	6k	100k≤	93,093	**1,320 (71x)**
grid_wrld(10, 14)	2k	7k	100k≤	100k≤	**2,234 (–)**
grid_wrld(12, 14)	2k	8k	100k≤	100k≤	**2,782 (–)**
bv_expr(7, 4, 12)	187	474	35,229	5,865	**2,139 (2.7x)**
it_expr(2, 4)	162	510	51,375	7,894	**2,635 (3x)**

I.I.D Generalization

Dataset	# vars	# clauses	`Random`	`SharpSAT`	`Neuro#`
sudoku(9, 25)	182	3k	338	220	**195(1.1x)**
n-queens(10, 20)	100	1.5k	981	466	**261(1.7x)**
sha-1(28)	3k	13.5k	2,911	52	**24(2.1x)**
island(2, 5)	1k	34k	155	86	**30(1.8x)**
cell(9, 20, 20)	210	1k	957	370	**184(2.0x)**
cell(35, 128, 110)	6k	25k	867	353	**198(1.8x)**
cell(49, 128, 110)	6k	25k	843	338	**206(1.6x)**
grid_wrld(10, 5)	329	967	220	195	**66(3.0x)**
bv_expr(5, 4, 8)	90	220	1,316	328	**205(1.6x)**
it_expr(2, 2)	82	264	772	412	**266(1.5x)**

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:



(a) cell(49, 256, 200) (b) grid_wrld(10, 12)

Fig. 1: 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).