A Generalization of SAT and \#SAT for Robust Policy Evaluation*

Erik Zawadzki  
Carnegie Mellon University  
Computer Science Dept.  
epz@cs.cmu.edu

André Platzer  
Carnegie Mellon University  
Computer Science Dept.  
aplatzer@cs.cmu.edu

Geoffrey J. Gordon  
Carnegie Mellon University  
Machine Learning Dept.  
ggordon@cs.cmu.edu

Abstract

Both SAT and \#SAT can represent difficult problems in seemingly dissimilar areas such as planning, verification, and probabilistic inference. Here, we examine an expressive new language, \#∃SAT, that generalizes both of these languages. \#∃SAT problems require counting the number of satisfiable formulas in a concisely-describable set of existentially-quantified, propositional formulas. We characterize the expressiveness and worst-case difficulty of \#∃SAT by proving it is complete for the complexity class \#P[1], and relating this class to more familiar complexity classes. We also experiment with three new general-purpose \#∃SAT solvers on a battery of problem distributions including a simple logistics domain. Our experiments show that, despite the formidable worst-case complexity of \#P[1], many of the instances can be solved efficiently by noticing and exploiting a particular type of frequent structure.

1 Introduction

\#∃SAT is similar to SAT and \#SAT—determining if a propositional boolean formula has a satisfying assignment, or counting such assignments. SAT may be written as $\exists \bar{x} \phi(\bar{x})$, and \#SAT may be written as $\Sigma \exists \bar{x} \phi(\bar{x})$, where $\bar{x}$ is a vector of finitely many boolean variables and $\phi(\bar{x})$ is a propositional formula. \#∃SAT allows a more general way of quantifying than SAT or \#SAT. Specifically, a \#∃SAT problem is $\Sigma \exists \bar{y} \phi(\bar{x}, \bar{y})$, which corresponds to counting the number of choices for $\bar{x}$ such that there is a $\bar{y}$ satisfying $\phi(\bar{x}, \bar{y})$.

The integer answer to a \#∃SAT instance has a natural interpretation: the number of formulas that are SAT from a concisely-described but exponentially large set of formulas. Each full assignment to the $\Sigma$-variables ‘selects’ a particular, entirely $\exists$-quantified, residual formula—i.e., $\exists \bar{y} \phi(\bar{x}, \bar{y})$ for some $\bar{x}$—from the set. If a concise quantifier-free representation of $\exists \bar{y} \phi(\bar{x}, \bar{y})$ could be found efficiently, \#∃SAT would reduce to \#SAT. In most instances, however, the existential quantification is required for concise representation.

\#∃SAT captures a simple type of probabilistic interaction useful for testing the robustness of a policy under uncertainty. As an example, imagine a delivery company pondering whether to purchase more vehicles to improve quality-of-service (QoS). They wonder if, under some world model, the probability of timely delivery could be significantly improved with more vehicles. We answer this question by counting\(^1\) how many random scenarios (e.g., truck breakdowns and road closures) permit delivery plans (sequences of vehicle movements, pickups, and dropoffs) that meet QoS constraints (every package is delivered to its destination by some predetermined time) for both the current fleet and the augmented one.

This logistics problem can be pseudo-formalized as $\Sigma \bar{b}, \bar{c}, \bar{r}, \bar{p} \#QoS(\bar{b}, \bar{c}, \bar{r}, \bar{p})$, where the vector $\bar{b}$ describes which vehicles break down, $\bar{c}$ lists road closures, $\bar{r}$ lists delivery requests, and $\bar{p}$ defines the plan of action. QoS is a formula that describes initial positions, goals, and action feasibility. After realizing all uncertainty, we are left with an instance of a famous $NP$-complete problem: finding $\bar{p}$ is bounded deterministic planning.

\#∃SAT is a subset of general planning under uncertainty that requires that all uncertainty is revealed initially. This excludes the succinct description of any problem that has a more complicated interlacing of action and observation. For example, the logistics problem does not describe the random breakdown of trucks after they leave the depot.

However, \#∃SAT is still very expressive—we characterize its complexity in §2. We provide three exact solvers for \#∃SAT in §3, before testing implementations of these approaches in §3.1 and §3.2.

The experiments are encouraging, and show a type of structure that can be noticed and exploited by solvers. Our experiments and algorithms may be useful not just for \#∃SAT problems, but also for problems with more complicated uncertainty. We are hopeful that similar structure can be discovered and exploited in these settings, and that our solvers can be used as components or heuristics for more general solvers.

---

\(^1\)Throughout, for simplicity, we discuss unweighted \#∃SAT, where each scenario is equally likely. Our algorithms also work for the weighted problem; furthermore, some weighted problems reduce to unweighted ones by proper encoding.

---

*This material is based upon work supported by the Army Research Office under Award No. W911NF-09-1-0273. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Army Research Office.
Related work. SAT is the canonical $NP$-complete problem. Many important problems like bounded planning (e.g., [Kautz and Selman, 1999]) and bounded model checking (e.g., [Biere et al., 2003]) can be solved by encoding problem instances in a normal form—like conjunctive normal form (CNF) or DNNF ([Darwiche, 2001])—and using an off-the-shelf SAT solver such as GRASP [Marques-Silva and Sakallah, 1999], Chaff [Moskewicz et al., 2001], zChaff [Fu et al., 2004], or MiniSat [Eén and Sörensson, 2006]. Current work in satisfaction modulo theory (SMT; e.g., [Nieuwenhuis et al., 2006]) is a continuation of this successful program.

This method of solving $NP$-complete problems (convert to normal form and solve with a SAT solver) succeeds because SAT solvers can automatically notice and exploit some kinds of structure that occur frequently in practice. Techniques include the venerable unit-propagation rule [Davis et al., 1962], various preprocessor methods (e.g., [Eén and Biere, 2005]), clause learning [Marques-Silva and Sakallah, 1999], restarting [Gomes et al., 1998], and many others. As a result, modern SAT solvers can tackle huge industrial problem instances. Scientists and engineers can largely treat them as black boxes, not delving too deeply into their code; and, improvements to SAT solvers have immediate and far-reaching impact.

SAT is not fully general and there are many reasons to examine more general settings. Many of these settings amount to allowing a richer mixture of quantifiers: there is a SAT-like problem at each level of the polynomial hierarchy, formed by bounded alternations of $\forall$ and $\exists$. QBF is even more general, allowing an unbounded number of $\exists$ and $\forall$ alternations; QBF is $PSPACE$-complete [Samulowitz and Bacchus, 2006].

Bounded alternation of $\exists$ and $\Sigma$ quantifiers yields another hierarchy of problems, and our $\#\Sigma SAT$ problem is one of the two problems at its second level. Other members of this hierarchy include the pure counting problem, $\#SAT$ (the canonical $P$-complete problem) and Bayesian inference (also $P$-complete [Roth, 1996]), as well as the two-alternation decision problem MAXPLAN [Majercik and Littman, 1998; 2003] and the unbounded-alternation $PSPACE$-complete decision problem stochastic SAT (SSAT; [Littman et al., 2001]).

Our counting problem is related to a restriction of SSAT. MAXPLAN bears a number of similarities to $\#\Sigma SAT$. It asks if a plan has over a $50\%$ probability of success, and can be thought of as asking an $\exists\forall\Sigma SAT$ thresholding question—the opposite alternation to our $\#\Sigma SAT$. MAXPLAN has a different order of observation that, in the planning analogy, means the MAXPLAN agent commits to a plan first, then observes the outcome of this commitment. The $\#\Sigma SAT$ agent observes first, then acts. MAXPLAN is $NP^{PP}$-complete (for the class of problems that are solvable by an $NP$ machine with access to a $PP$ oracle), and we compare its expressiveness to $\#\Sigma SAT$ in §2.

While $\#\Sigma SAT$ is in $PSPACE$ and, could in theory, be solved by a QBF solver we are not aware of any empirically useful reductions of $\#SAT$ to QBF. Indeed, we are not aware of a reduction that does not involve simulating a $\#SAT$ solver with a counting circuit—these are thought to be a difficult case for QBF solvers (e.g., [Janota et al., 2012]). We expect the relation between $\#\Sigma SAT$ and QBF to be similar.

$\#\Sigma SAT$ is also a special case of another general problem—

it is a probabilistic constraint satisfaction problem [Fargier et al., 1995] with complete knowledge and binary variables. The restriction to $\#\Sigma SAT$ not only allows us to develop both novel algorithms but also stronger theoretical results.

We note that this paper concerns exact solvers rather than approximate solvers (e.g., [Wei and Selman, 2005] or [Gomes et al., 2007]). This is for several reasons. First, we are interested in solvers that provide non-trivial anytime bounds on the probability range—so we can terminate if our bounds become sufficiently tight or are sufficient to answering a thresholding question. Secondly, we believe that exact solvers will generalize better to first-order settings such as [Sanner and Kersting, 2010] or [Zawadzki et al., 2011].

2 Complexity

The previous section mentions a number of other problems that generalize SAT. In this section we clarify how expressive $\#\Sigma SAT$ is compared to them with three theoretical statements. For each, we provide a proof sketch and some intuition about how to interpret the result.

Our first result is that $\#\Sigma SAT$ is complete for $P^{NP[1]}$. $P$, by itself, is the class of counting problems that can be answered by a polynomially-bounded counting Turing machine. A counting Turing machine is a nondeterministic machine that counts paths rather than testing if there is a path. The machine’s polynomial bound applies to the length of its nondeterministic execution paths.

The superscripted oracle notation used in $P^{NP[1]}$ refers to a generalization of the $P$ counting machine that allows the machine to make a single query to an $NP$-complete oracle per path. This oracle seems weak at first glance—there is a simple reduction from $NP$ to $P$, so why would a single call to this oracle help? A later result shows, however, that this oracle call does change the complexity class unless the polynomial hierarchy collapses.

**Thm. 1 (Complete).** $\#\Sigma SAT$ is complete for $P^{NP[1]}$.

 deciding Turing machine can solve this problem by nondeterministically choosing the $\Sigma$-variables, and then asking the oracle whether the entirely $\exists$-quantified residual formula is SAT or not.

Second, we show that an arbitrary problem $A \in P^{NP[1]}$ can be converted to an instance of $\#\Sigma SAT$ in polynomial time. This is done through a Cook-Levin-like argument: since there must be some oracle-enhanced, polynomially-bounded counting Turing machine $M$ that counts $A$, we will just simulate it in a $\#\Sigma SAT$ formula $\phi$. Here, we use $\Sigma$-variables to describe the time-bounded operation of the underlying counting Turing machine, and $\exists$-variables to describe the time-bounded operation of the $NP$ oracle. We omit the technical detail of this description in the interests of brevity, but the time required to construct this simulation is bounded by a polynomial in the size of the original input.

We now turn to whether the oracle call actually adds something: $P^{NP[1]}$ is not merely $P$ in disguise.

**Thm. 2.** If $\#\Sigma SAT$ reduces to $P$, then the polynomial hierarchy collapses to $\Sigma_2^P$.

Proof. First we show that our problem is in $P^{NP[1]}$. Our oracle-enhanced, polynomially-bounded counting Turing machine can solve this problem by nondeterministically choosing the $\Sigma$-variables, and then asking the oracle whether the entirely $\exists$-quantified residual formula is SAT or not.
This proof is based on the fact that a ‘uniquifying’ Turing machine $M_{DPLL}$—a machine that can take a propositional boolean formula (p.b.f) $\phi$ and produce another p.b.f. $\psi$ that has a unique solution iff $\phi$ has any (and none otherwise)—cannot run in deterministic polynomial time unless the polynomial hierarchy collapses to $\Sigma_2^P$ (a corollary of [Dell et al., 2012] and [Karp and Lipton, 1982]).

**Proof.** Suppose $\#\exists\text{SAT}$ reduces to $\#\text{P}$. Then there is a polynomial time Turing machine $M_{RED}$ that reduces any $\Sigma_3$-quantified p.b.f. $\Phi = \exists x \exists y \phi(x, y)$ to a p.b.f. $\psi$ such that counting solutions to $\psi$ answers our $\#\exists$-counting question about $\Phi$. Therefore, $\Phi$ must have the same number of $\exists$-solutions as $\psi$ has solutions: $\text{Count}_{\#\exists}(\Phi) = \text{Count}_{\#}(\psi)$. We use $M_{RED}$ to uniquify any boolean formula $\phi$ as follows. First, form the $\#\exists\text{SAT}$ formula $\Phi = \exists x \exists y [x \land \phi(y)]$. By design, $\text{Count}_{\#\exists}(\Phi) = 1$ iff $\text{Count}_{\#}(\phi) \geq 1$.

Then, since we have assumed that $\#\exists\text{SAT}$ reduces to $\#\text{SAT}$, we can run $\Phi$ through $M_{RED}$ to produce a p.b.f. $\psi$. Since $\text{Count}_{\#\exists}(\Phi) = \text{Count}_{\#}(\psi)$, $\psi$ is the uniquified version of $\phi$. This whole process runs in polynomial time if $M_{RED}$ is, so $M_{RED}$ cannot exist unless $PH$ collapses. □

Thus, the oracle call (probably) adds expressiveness and our problem $\#\exists\text{SAT}$ is (probably) more general than $\#\text{SAT}$.

Finally, we combine some existing results to show that $NP^{PP}$ contains $PP^{NP}$, a decision class closely related to our counting class. Class $PP^{NP}$ is ‘close’ in the sense that it ‘cooks’-reduces to our counting class $#P^{NP}$.

**Cor. 1.** $PP^{NP} \subseteq NP^{PP}$

**Proof.** Follows from Toda’s theorem [Toda, 1991] (middle inclusion): $PP^{NP} \subseteq PP^{PH} \subseteq PP^{PP} \subseteq NP^{PP}$.

This establishes that a closely related decision problem to our $#P^{NP}$ is contained in $NP^{PP}$, the complexity class that MAXPLAN is complete for. The result suggests that thresholding questions for $\#\exists\text{SAT}$ are possibly less expressive than MAXPLAN, but also easier in the worst case.

### 3 Algorithms

The previous section establishes $\#\exists\text{SAT}$’s worst-case difficulty, but we know from many other problems (e.g., SAT) that the empirical behavior of solvers in practice can be radically different than the worst-case complexity.

In the next two sections we explore the empirical behavior of three different solvers on several distributions of $\#\exists\text{SAT}$ instances. $\#\exists\text{SAT}$ generalizes both SAT and $\#\text{SAT}$, so the first two solvers are adaptations of algorithms for those settings. The final solver is a novel DPLL-like procedure, and capitalizes on an observation specific to the $\#\exists\text{SAT}$ setting.

Our design principle for these solvers is to use a black box DPLL solver as an inner loop. First, our solvers automatically get faster whenever there is a better DPLL solver. Second, the inner loop of the black-box solver is already highly optimized, so we can avoid zealously optimizing much of our solver and focus on higher-level design questions.

**mDPLL: A SAT inspired solver.** One intuition for $\#\exists\text{SAT}$ problems is that instances with a small number of $\Sigma$-variables might be solvable by running a SAT solver until it sweeps across every $\Sigma$-assignment (rather than returning after finding the first satisfying assignment, like we would in SAT). We test this intuition by generalizing DPLL.

Our first algorithm, mDPLL, searches over $\Sigma$-assignments (consistent total or partial assignments to the $\Sigma$-variables), pruning whenever a $\Sigma$-assignment can be shown to be SAT or UNSAT. Each $\Sigma$-assignment defines a subproblem $S = (\phi, A_\Sigma, U_\Sigma, U_\Sigma)$, where $\phi$ is the original formula, $A_\Sigma \subseteq L_\Sigma$ is the $\Sigma$-assignment, and $U_\Sigma \subseteq V_\Sigma$ and $U_\Sigma \subseteq V_\Sigma$ are the unassigned $\Sigma$ and $\exists$ variables. $L_\Sigma, L_\exists, V_\Sigma, V_\exists$ are sets of the $\Sigma$ and $\exists$ variables and literals.

Our implementation is iterative (we maintain an explicit stack), but for clear exposition we present mDPLL as a recursive procedure. mDPLL is a special case of mDPLL/C (Alg 1) that skips lines 4-8. These two cases are explained later in the description for mDPLL/C. mDPLL first checks if a subprob-

**Algorithm 1**

1: function mDPLL/C($S = (\phi, A_\Sigma, U_\Sigma, U_\Sigma)$)  
2: if UnSatLeaf($S$) then return 0  
3: if SatLeaf($S$) then return $2^{|U_\Sigma|}$  
4: if InCache($S$) then return CachedValue($S$)  
5: if Shatterable($S$) then  
6: $S' = \text{Shatter}(S)$  
7: $\langle C^{(1)}, \ldots, C^{(m)} \rangle \leftarrow \text{Shatter}(S)$  
8: return $\prod_{i=1}^{m} \text{mDPLL/C}(C^{(i)})$  
9: $(S_x, S_{x\neg}) \leftarrow \text{Branch}(S)$  
10: return mDPLL/C($S_x$) + mDPLL/C($S_{x\neg}$)

lem $S$ is either an SAT or UNSAT leaf in the UnSatLeaf and SatLeaf functions. Both of these checks are done with the same black box SAT solver call. $S$ is an UNSAT leaf if $\phi$ is UNSAT assuming $A_\Sigma (\bigwedge_{u \in A_3} \neg \phi) \wedge \phi$ is UNSAT), and a SAT leaf if the solver produces a model where each clause in $\phi$ is satisfied by at least one literal not in $U_\Sigma$. If $S$ is not a leaf then the subproblem is split into two subproblems $S_x$ and $S_{x\neg}$ in the Branch function by branching on some $\Sigma$-variable in $U_\Sigma$.

**Σ-literal unit propagation is a special case of branching where the implementation has fast machinery to determine if one of the children is an UNSAT leaf. Σ-literal unit propagation is handled by the black box solver.**

**Thm. 3.** For any $\#\exists\text{SAT}$ formula $\Sigma x \exists y \phi(x, y)$ with $\Sigma$-count $\kappa$, mDPLL returns $\kappa$.

Proof by induction on structure omitted for brevity. See [Zawadski et al., 2013] for details.

**mDPLL/C: a #SAT inspired solver.** For problem instances with a large number of $\Sigma$-variables we might suspect that $\#\text{SAT}$’s techniques are more useful than SAT’s. There are at least two families of exact $\#\text{SAT}$ solvers: based on either binary decision diagrams (BDDs [Bryant, 1992]) or DPLL with component caching like cachet [Sang et al., 2005]. In this paper we focus on component caching. Modern caching solvers tend to outperform BDD solvers and our

2We use an activity-based branching heuristic similar to VSIDS [Moskewicz et al., 2001] in our implementation.
initial experiments with BDD solvers were unpromising.\footnote{We built a BDD with a special stratified variable ordering and eliminated any $\exists$-variable from the diagram. This solver was dramatically slower than any of our other algorithms on every problem instance—the first step of constructing the BDD with a restricted variable order was exceptionally time consuming. This approach, however, may still be useful if one has a particularly quick method of constructing BDDs for a particular application.}
mDPLL/C (Alg 1) adds two cases (lines 4–8) to mDPLL. If $S$ is not a leaf, then InCache checks a bounded-sized cache of previously counted components for a match.\footnote{Fully counted components are cached in a hash table with LRU eviction. Components are represented as $U_{\exists} \cup U_{\exists}$ and the set of active clauses (not already SAT) that involve these variables.} If there is a match the CachedValue is returned.

If $S$ is neither cached nor a leaf, then Shatterable checks $S$ for components using depth first search. Components are subproblems formed in the Splitter step by partitioning $U_{\exists} \cup U_{\exists}$ into disjoint pairs $U_{\exists}(1) \cup U_{\exists}(1), \ldots, U_{\exists}(m) \cup U_{\exists}(m)$ so that no clause in $\phi$ contains literals from different pairs. Each component $C(i) = \langle \phi(i), A_{\exists}, U_{\exists}(1), U_{\exists}(1) \rangle$ has a formula $\phi(i)$ that is restricted to only involve literals from $U_{\exists}(i) \cup U_{\exists}(i)$—the satisfiability of a component is relative to this restricted formula. Detection and shattering are expensive—profiling component caching algorithms reveals that solvers spend a large proportion of their time doing this work [Sang et al., 2004]—but can dramatically simplify counting in \#SAT.

In both mDPLL and mDPLL/C our implementations augment $\phi$ with learned clauses found by the black box solver. Since we explicitly check $S$ for feasibility in the UnSatLeaf check this is a safe operation [Sang et al., 2004].

**Thm. 4.** For any \#SAT formula $\Sigma x \exists y \phi(x, y)$ with \#E-count $\kappa$, mDPLL/C returns $\kappa$.

Proof omitted for brevity. See [Zawadzki et al., 2013].

**Algorithm 2**

1: function POPS($\phi, U_{\exists}, U_{\exists}$)  
2: \langle $\phi', U_{\exists}'$ \rangle \leftarrow Rewrite($\phi, U_{\exists}$)  
3: return POPS_helper($\langle \phi', U_{\exists}' \rangle, U_{\exists}$)  
4: function POPS_helper($S = \langle \phi, A_{\exists}, U_{\exists}, U_{\exists} \rangle$)  
5: if SatSolve($\text{Pess}(S)$) then return 2[$U_{\exists}$]  
6: if \neg SatSolve($\text{Opt}(S)$) then return 0  
7: $x \leftarrow \text{Branch}(S)$  
8: $S_x \leftarrow \langle \phi, A_{\exists} \cup \{p_x, n_x\}, U_{\exists} \setminus \{p_x, n_x\}, U_{\exists} \rangle$  
9: $S_{\neg x} \leftarrow \langle \phi, A_{\exists} \cup \{\neg p_x, n_x\}, U_{\exists} \setminus \{p_x, n_x\}, U_{\exists} \rangle$  
10: return POPS_helper($S_x$) + POPS_helper($S_{\neg x}$)

POPS: pessimistic and optimistic pruning search. The final algorithm, POPS, is based on being agnostic about values of $\Sigma$-variables whenever possible. If, during a SAT solve, we notice a subproblem can be satisfied with just the $\exists$-variables then we can declare the problem to be a SAT leaf. On the other hand, if we notice that a subproblem cannot be satisfied regardless of how the $\Sigma$-variables are assigned we can declare it to be a UNSAT leaf.

This pruning is done by SAT-solving two modified formula per subproblem (mDPLL and mDPLL/C solved one formula per subproblem). The first is the pessimistic problem, which is SAT only if every way of extending $A_{\exists}$ with $\Sigma$-variables is SAT. The second is the optimistic problem, which is UNSAT only if every way of extending $A_{\exists}$ with $\Sigma$-variables is SAT. We prune if the pessimist is SAT, or the optimist is UNSAT, and branch otherwise.

Both problems use the same black box solver instance by rewriting the original CNF formula. This allows activity information and learned clauses to be shared, and saves memory allocations. We rewrite the formula to essentially allow any $\Sigma$-variable to take only one of four values—true ($T$), false ($F$), unknown but optimistic ($O$), or unknown but pessimistic ($P$). If a $\Sigma$-variable is $O$ a clause can be satisfied by either the positive or the negative literal of that variable; if it is $P$, a clause cannot be satisfied by either literal. $T$ and $F$ behave as usual—only the appropriate literal satisfies clauses.

This four-valued logic is encoded through the literal splitting rule. It replaces every negative literal of a $\Sigma$-variable $x$ with a fresh $\exists$-variable $n_{x}$ and every positive literal with a $\Sigma$-variable $p_{x}$. A $\Sigma$-variable $x$ may be set to any of four values by making different assertions about $n_{x}$ and $p_{x}$:

$$
\begin{align*}
\end{align*}
$$

This encoding yields a simple formulation of the optimistic and pessimistic problems: for some rewritten problem $S$ the purely $\exists$-variable optimistic problem is $\text{Opt}(S) = \langle \phi, A_{\exists} \cup \{u \mid u \in U_{\exists}\}, \emptyset, U_{\exists}\rangle$ and the pessimistic problem is $\text{Pess}(S) = \langle \phi, A_{\exists} \cup \{\neg u \mid u \in U_{\exists}\}, \emptyset, U_{\exists}\rangle$. For example, $\Sigma x \exists y [x \lor y] \land [\neg x \land y]$ is rewritten as $\exists y, n_{x}, p_{y}, [p_{x} \lor y] \land [n_{x} \lor y]$. The pessimistic problem (i.e., $[p_{x} = F, n_{x} = F]$) is SAT so we return 2 at the root without any branching.

POPS initially Rewrites the problem by literal splitting. A subproblem is pruned if the overly constrained pessimistic problem is SAT ($\text{SatSolve}(\text{Pess}(S))$; SatSolve is the black box solver) or if the relaxed optimistic problem is UNSAT ($\neg \text{SatSolve}(\text{Opt}(S))$). Otherwise POPS chooses to Branch on one of the $\Sigma$-variables $x$ and solves the child subproblems $S_{x}$ and $S_{\neg x}$ (see Alg 2).

**Thm. 5.** For any \#SAT formula $\Sigma x \exists y \phi(x, y)$ with \#E-count $\kappa$, POPS returns $\kappa$.

Proof omitted for brevity. See [Zawadzki et al., 2013].

### 3.1 Problem distributions

We explore the empirical characteristics of the these algorithms by running them on a number of instances drawn from four problem distributions—job shop scheduling, graph 3-coloring, a logistics problem, and random 3\#SAT. The distributions touch a number of properties: job shop scheduling is a packing problem that uses binary-encoded uncertainty, the 3-coloring problems are posed on dense graphs, the logistics problem is a bounded-length deterministic planning problem, and random 3\#SAT is unstructured.

**Job shop scheduling.** Schedule $J$ jobs of varying length on $M$ machines with time bound $T$. Job lengths are described by $P$ bits of uncertainty per job, encoded by $\Sigma$-variables.

**Graph 3-coloring.** Color an undirected graph where we have uncertainty about which edges are present: for every edge there is a $\Sigma$-variable to disable the edge iff true. Parameters are number of vertices $V$ and proportion of edges $P_{E}$. 
Table 1: Parameter settings for the five experiments.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Solvers</th>
<th>Dist.</th>
<th>Parameters</th>
<th>Jobs</th>
<th>Insts per param</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cachet, mDPLL/C</td>
<td>Pure</td>
<td>${C}$, $R$</td>
<td>$M \in {2, 3}$, $T \in {3, 4, 5}$</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>mDPLL, mDPLL/C, POPs</td>
<td>Random</td>
<td>$R \in {0.7, 0.8, 0.9}$</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>mDPLL, mDPLL/C, POPs</td>
<td>Jobs</td>
<td>$M \in {2, 3, 4}$, $T \in {6, 8, 10}$</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>mDPLL, mDPLL/C, POPs</td>
<td>Logistics</td>
<td>$C \in {1, 0.1}$, $B \in {2, 3, 4}$, $T \in {6, 8}$</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>mDPLL, mDPLL/C, POPs</td>
<td>Random</td>
<td>$V \in {10, 15, \ldots, 150}$, $P \in {0, 1, 0.2, 0.3}$, $R \in {2.5, 3, 4, 5}$</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

3.2 Experiments

Our experiments ran on a 32-core AMD Opteron 6135 machine with $32 \times 4$ GiB of RAM, on Ubuntu 12.04. Each run was capped at 4 GiB of RAM and cut off after two hours. The experiments ran for roughly 160 CPU days. Table 1 shows the parameter settings. Each instance and solver pair was run only once because the solvers are deterministic.

We hypothesize that POPs exploits a type of structure reminiscent of conditional independence in probability theory or backdoors in SAT (e.g., [Kilby et al., 2005]). By solving the pessimistic problem POPs can demonstrate that—given some small partial assignment to the $\Sigma$-variables and full assignment to the $\exists$-variables—the remaining $\Sigma$-variables are unconstrained and can take on any value. We call this $\Sigma$-independence, and expect it to occur more frequently in lightly constrained formulas, and in formulas close to being either VALID or UNSAT. mDPLL and mDPLL/C are generally unable to exploit this type of structure.

Experiment 1, checking mDPLL/C implementation.

In this experiment we demonstrate that we have a reasonable implementation of component caching by comparing mDPLL/C and cachet to each other on a 72 instances of purely $\#SAT$ job shop scheduling (see Table 1 for details). We capped both programs at $2.1 \times 10^9$ cache entries.

A clear trend emerged. For each machine ($M \in \{2, 3\}$) and time-step ($T \in \{3, 4, 5\}$) the graph is similar to Fig 1:

mDPLL/C is an order of magnitude slower than cachet on small problems, but eventually becomes somewhat faster. We suspect that this scaling behavior has to do with our different way of handling UNSAT components. Problems from this distribution have an increasingly small ratio of SAT $\Sigma$-assignments to UNSAT $\Sigma$-assignments as jobs are added, so the effect of this difference becomes more pronounced. However, since many of our problem distributions have this ‘larger problems have a smaller ratio’ property, we believe that Fig 1 argues strongly that our solver specializes to be a reasonable $\#SAT$ solver for the instances that we examine.

Figure 1: $\#SAT$ job shop scheduling problems with 2 machines, 2 bits of uncertainty and 4 times steps with varying numbers of jobs.

Experiment 2, job scheduling scaling.

The job scheduling instances exhibited a pattern that repeats in most of our experiments: the POPs solver tended to outperform the other two, especially when instances were close to being either VALID or UNSAT. Additionally, augmenting the mDPLL solver with component caching did not help—mDPLL/C was the slowest solver on every job scheduling instance. These results are summarized in Table 2 (left). Fig 2 is typical of the scaling curves on this distribution. We see that POPs is
dramatically faster than the other two solvers until 6 jobs.

<table>
<thead>
<tr>
<th>Jobs</th>
<th>mdPPL</th>
<th>mdPPL/C</th>
<th>POPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdPPL</td>
<td>135</td>
<td>2</td>
<td>190</td>
</tr>
<tr>
<td>mdPPL/C</td>
<td>136</td>
<td>138</td>
<td>173</td>
</tr>
<tr>
<td>POPS</td>
<td>135</td>
<td>2</td>
<td>190</td>
</tr>
</tbody>
</table>

Table 2: Number of instances where the row solver beats the column solver. **Left:** based on 270 job scheduling instances. **Right:** based on 880 3-coloring instances.

**Experiment 3, 3-color scaling.** The trends in 3-coloring are similar to those found in the job shop experiments—POPS is the fastest solver on almost every instance (see Table 2 right). Unlike in the jobs setting, the performance gap between POPS and the other solvers does not close. Fig 3 illustrates this phenomenon for graphs with 70% edge density, but denser graphs are similar. These trends may indicate that only a small number of the edges are important to reason about.

**Experiment 4, logistics scaling.** The logistics experiments are more difficult to summarize than previous experiments, but the left of Table 3 shows that POPS is again the fastest solver for most instances. mdPPL, however, is faster than POPS for a relatively large number of the instances—especially compared to previous experiments. Instances where mdPPL is superior might have common properties—they might lack Σ-independence, or perhaps independence is present but POPS fails to exploit it with our current heuristics.

**Experiment 5, random 3#SAT scaling.** The right of Table 3 paints a different picture than the previous experiments: here, neither POPS nor mdPPL seem to be the true victor. Both beat the other on a number of different instances—although, again, mdPPL/C seems to be the slowest solver.

Taking a look at the different clause ratios is informative, and the different parameterizations have very dissimilar scaling trends. The instances where the clause ratio is 2.5 paints a rosy picture for POPS (e.g., Fig 4—it is the fastest algorithm in 28% of such instances, and is only beaten by mdPPL in 2% of these instances). We note that the variance for POPS grows quickly with the number of variables, indicating more sensitivity to problem structure than mdPPL and mdPPL/C. However, if we restrict our attention to more constrained instances with a clause ratio of 4.0, then we get a much different picture. Here, mdPPL emerges as the superior algorithm, beating POPS in 29% of such instances while POPS beats mdPPL only 3% of the time—a reversal of the previous trend.

**4 Conclusions**

In this paper we introduced #∃SAT, a problem with a number of interesting properties. #∃SAT can, for example, represent questions about the robustness of a policy space for a simple type of planning under uncertainty. Not only did we provide theoretical statements about the expressiveness and worst-case difficulty of #∃SAT, but we also built the first three dedicated #∃SAT solvers.

We ran these solvers through their paces on four different distributions and many different instances. These experiments led us to three conclusions. First, our algorithm POPS shows promise on many of these instances, sometimes running many orders of magnitude faster than the next fastest algorithm, due to its ability to exploit Σ-independence. Second, the instances on which POPS solver was slower than mdPPL should serve as focal instances for understanding the exploitable structure that occurs in #∃SAT. Finally, they suggest that #SAT-style component caching is detrimental to solving #∃SAT problems. This does not rule out lighter-weight component detection tailored to #∃SAT’s unique trade-offs.

There are a number of research directions: our theory about the importance of Σ-independence should be tested on more problem distributions. Further profiling should guide the design of better heuristics; POPS, in particular, will benefit from a branching heuristic tuned to its style of reasoning. Profiling data may inspire additional methods for exploiting independence structures and symmetry in #∃SAT problems. A final direction is to build approximate solvers that maintain bounds on their approximation. These may be necessary for tackling larger real-world applications.
References


