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, $\#\exists$SAT, that generalizes both of these languages. $\#\exists$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 $\#\exists$SAT by proving it is complete for the complexity class $\#P^{NP[1]}$, and relating this class to more familiar complexity classes. We also experiment with three new general-purpose $\#\exists$SAT solvers on a battery of problem distributions including a simple logistics domain. Our experiments show that, despite the formidable worst-case complexity of $\#P^{NP[1]}$, many of the instances can be solved efficiently by noticing and exploiting a particular type of frequent structure.

1 Introduction

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

The integer answer to a $\#\exists$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 Σ-variables 'selects' a particular, entirely $\exists$-quantified, residual formula—*i.e.*, $\exists\vec{\mathbf{y}}\ \phi(\vec{\mathbf{x}}, \vec{\mathbf{y}})$ for some $\vec{\mathbf{x}}$—from the set. If a concise quantifier-free representation of $\exists\vec{\mathbf{y}}\ \phi(\vec{\mathbf{x}}, \vec{\mathbf{y}})$ could be found efficiently, $\#\exists$SAT would reduce to $\#$SAT. In most instances, however, the existential quantification is required for concise representation.

$\#\exists$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

$$\Sigma\vec{\mathbf{b}}, \vec{\mathbf{c}}, \vec{\mathbf{r}}\ \exists\vec{\mathbf{p}}\ \mathsf{QoS}(\vec{\mathbf{b}}, \vec{\mathbf{c}}, \vec{\mathbf{r}}, \vec{\mathbf{p}}),$$

where the vector $\vec{\mathbf{b}}$ describes which vehicles break down, $\vec{\mathbf{c}}$ lists road closures, $\vec{\mathbf{r}}$ lists delivery requests, and $\vec{\mathbf{p}}$ 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 $\vec{\mathbf{p}}$ is bounded deterministic planning.

$\#\exists$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, $\#\exists$SAT is still very expressive—we characterize its complexity in §2. We provide three exact solvers for $\#\exists$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 $\#\exists$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.

*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.

[1] Throughout, for simplicity, we discuss *unweighted* $\#\exists$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.

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 preprocessing 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 Σ quantifiers yields another hierarchy of problems, and our $\#\exists$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 $\#\exists$SAT. It asks if a plan has over a 50% probability of success, and can be thought of as asking an $\exists\#$SAT thresholding question—the opposite alternation to our $\#\exists$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 $\#\exists$SAT agent observes first, then acts. MAXPLAN is NP^{PP}-complete (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 $\#\exists$SAT in §2.

While #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 $\#\exists$SAT and QBF to be similar.

$\#\exists$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 $\#\exists$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 $\#\exists$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 $\#\exists$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). *$\#\exists$SAT is complete for $\#P^{NP[1]}$.*

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 Σ-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 $\#\exists$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 $\#\exists$SAT formula ϕ. Here, we use Σ-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. $\square$

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

Thm. 2. *If $\#\exists$SAT reduces to $\#P$, then the polynomial hierarchy collapses to Σ_2^P.*

This proof is based on the fact that a 'uniquifying' Turing machine M_{UNQ}—a machine that can take a propositional boolean formula (p.b.f) ϕ and produce another p.b.f. ψ that has a unique solution iff ϕ has any (and none otherwise)—cannot run in deterministic polynomial time unless the polynomial hierarchy collapses to Σ_2^P (a corollary of [Dell *et al.*, 2012] and [Karp and Lipton, 1982]).

Proof. Suppose #∃SAT reduces to $\#P$. Then there is a polynomial time Turing machine M_{RED} that reduces any $\Sigma\exists$-quantified p.b.f. $\Phi = \Sigma\vec{\mathbf{x}}\exists\vec{\mathbf{y}}\ \phi(\vec{\mathbf{x}},\vec{\mathbf{y}})$ to a p.b.f. ψ such that counting solutions to ψ answers our #∃-counting question about Φ. Therefore, Φ must have the same number of $\Sigma\exists$-solutions as ψ has solutions: $\mathsf{Count}_{\#\exists}(\Phi) = \mathsf{Count}_{\#}(\psi)$.

We use M_{RED} to uniquify any boolean formula ϕ as follows. First, form the #∃SAT formula $\Phi = \Sigma x\exists\vec{\mathbf{y}}\ [x \wedge \phi(\vec{\mathbf{y}})]$. By design, $\mathsf{Count}_{\#\exists}(\Phi) = 1$ iff $\mathsf{Count}_{\#}(\phi) \geq 1$.

Then, since we have assumed that #∃SAT reduces to #SAT, we can run Φ through M_{RED} to produce a p.b.f. ψ. Since $\mathsf{Count}_{\#\exists}(\Phi) = \mathsf{Count}_{\#}(\psi)$, ψ is the uniquified version of ϕ. 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 #∃SAT is (probably) more general than #SAT.

Finally, we combine some existing results to show that NP^{PP} contains $PP^{NP[1]}$, a decision class closely related to our counting class. Class $PP^{NP[1]}$ is 'close' in the sense that it Cook-reduces to our counting class $\#P^{NP[1]}$.

Cor. 1. $PP^{NP[1]} \subseteq NP^{PP}$

Proof. Follows from Toda's theorem [Toda, 1991] (middle inclusion): $PP^{NP[1]} \subseteq PP^{PH} \subseteq P^{PP} \subseteq NP^{PP}$. □

This establishes that a closely related decision problem to our $\#P^{NP[1]}$ is contained in NP^{PP}, the complexity class that MAXPLAN is complete for. The result suggests that thresholding questions for #∃SAT are possibly less expressive than MAXPLAN, but also easier in the worst case.

3 Algorithms

The previous section establishes #∃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 #∃SAT instances. #∃SAT generalizes both SAT and #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 #∃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 #∃SAT problems is that instances with a small number of Σ-variables might be solvable by running a SAT solver until it sweeps across every Σ-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 Σ-assignments (consistent total or partial assignments to the Σ-variables), pruning whenever a Σ-assignment can be shown to be SAT or UNSAT. Each Σ-assignment defines a subproblem $S = \langle \phi, A_\Sigma, U_\Sigma, U_\exists \rangle$, where ϕ is the original formula, $A_\Sigma \subset \mathcal{L}_\Sigma$ is the Σ-assignment, and $U_\Sigma \subseteq \mathcal{V}_\Sigma$ and $U_\exists \subseteq \mathcal{V}_\exists$ are the unassigned Σ and $\exists$ variables. $\mathcal{L}_\Sigma$, $\mathcal{L}_\exists$, $\mathcal{V}_\Sigma$, $\mathcal{V}_\exists$ are sets of the Σ 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 subproblem 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 ϕ is UNSAT assuming A_Σ ($(\bigwedge_{a \in A_\Sigma} a) \wedge \phi$ is UNSAT), and a SAT leaf if the solver produces a model where each clause in ϕ is satisfied by at least one literal not in U_Σ. If S is not a leaf then the subproblem is split into two subproblems S_x and $S_{\neg x}$ in the Branch function by branching on some Σ-variable in U_Σ.[2]

Algorithm 1

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

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. $\exists$-literal unit propagation is handled by the black box solver.

Thm. 3. *For any #∃SAT formula $\Sigma x \exists y\, \phi(x,y)$ with $\Sigma\exists$-count κ, `mDPLL` returns κ.*

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

`mDPLL/C`: a #SAT inspired solver. For problem instances with a large number of Σ-variables we might suspect that #SAT's techniques are more useful than SAT's. There are at least two families of exact #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

[2]We use an activity-based branching heuristic similar to VSIDS [Moskewicz *et al.*, 2001] in our implementation.

initial experiments with BDD solvers were unpromising.[3]

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.[4] 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 Shatter step by partitioning $U_\Sigma \cup U_\exists$ into disjoint pairs $U_\Sigma^{(1)} \cup U_\exists^{(1)}, \ldots, U_\Sigma^{(m)} \cup U_\exists^{(m)}$ so that no clause in ϕ contains literals from different pairs. Each component $C^{(i)} = \langle\ \phi^{(i)}, A_\Sigma, U_\Sigma^{(i)}, U_\exists^{(i)}\ \rangle$ has a formula $\phi^{(i)}$ that is restricted to only involve literals from $U_\Sigma^{(i)} \cup U_\exists^{(i)}$—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 ϕ 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 Σ∃-count κ, mDPLL/C returns κ.*

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

Algorithm 2

```
function POPS(ϕ, U_Σ, U_∃)
    ⟨ϕ′, U′_Σ⟩ ← Rewrite(ϕ, U_Σ)
    return POPS_helper(⟨ϕ′, ∅, U′_Σ, U_∃⟩)
function POPS_helper(S = ⟨ϕ, A_Σ, U_Σ, U_∃⟩)
    if SatSolve(Pess(S)) then return 2^|U_Σ|
    if ¬SatSolve(Opt(S)) then return 0
    x ← Branch(S)
    S_x ← ⟨ϕ, A_Σ ∪ {p_x, ¬n_x}, U_Σ \ {p_x, n_x}, U_∃⟩
    S_¬x ← ⟨ϕ, A_Σ ∪ {¬p_x, n_x}, U_Σ \ {p_x, n_x}, U_∃⟩
    return POPS_helper(S_x) + POPS_helper(S_¬x)
```

POPS: pessimistic and optimistic pruning search. The final algorithm, POPS, is based on being agnostic about values of Σ-variables whenever possible. If, during a SAT solve, we notice a subproblem can be satisfied with just the ∃-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 Σ-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_Σ with Σ-variables is SAT. The second is the *optimistic problem*, which is UNSAT only if every way of extending A_Σ with Σ-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 Σ-variable to take one of four values—true (T), false (F), unknown but optimistic (O), or unknown but pessimistic (P). If a Σ-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 Σ-variable x with a fresh ∃-variable n_x and every positive literal with a ∃-variable p_x. A Σ-variable x may be set to any of four values by making different assertions about n_x and p_x:

x	O	T	F	P
p_x	T	T	F	F
n_x	T	F	T	F

This encoding yields a simple formulation of the optimistic and pessimistic problems: for some rewritten problem S the purely ∃-variable optimistic problem is $Opt(S) = \langle \phi, A_\Sigma \cup \{u \mid u \in U_\Sigma\}, \emptyset, U_\exists \rangle$ and the pessimistic problem is $Pess(S) = \langle \phi, A_\Sigma \cup \{\neg u \mid u \in U_\Sigma\}, \emptyset, U_\exists \rangle$. For example, $\Sigma x \exists y\ [x \vee y] \wedge [-x \vee y]$ is rewritten as $\exists y, n_x, n_p\ [p_x \vee y] \wedge [n_x \vee y]$. 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 (SatSolve($Pess(S)$); SatSolve is the black box solver) or if the relaxed optimistic problem is UNSAT ($\neg$SatSolve($Opt(S)$)). Otherwise POPS chooses to Branch on one of the Σ-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 Σ∃-count κ, POPS returns κ.*

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 Σ-variables.

Graph 3-coloring. Color an undirected graph where we have uncertainty about which edges are present: for every edge there is a Σ-variable to disable the edge iff true. Parameters are number of vertices V and proportion of edges P_E.

---

[3]We built a BDD with a special stratified variable ordering and eliminated any ∃-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.

[4]Fully counted components are cached in a hash table with LRU eviction. Components are represented as $U_\Sigma \cup U_\exists$ and the set of active clauses (not already SAT) that involve these variables.

#	Solvers	Dist.	Parameters	Insts per param
1	`cachet`, `mDPLL/C`	Pure # Jobs	$J \in \{1, \ldots, 12\}, M \in \{2, 3\}, T \in \{3, 4, 5\}, P = 2$	1
2	`mDPLL`, `mDPLL/C`, `POPS`	Jobs	$J \in \{2, \ldots, 16\}, M \in \{2, 3, 4\}, T \in \{6, 8, 10\}, P \in \{2, 3\}$	1
3	`mDPLL`, `mDPLL/C`, `POPS`	Color	$V \in \{3, \ldots, 24\}, P_E \in \{0.7, 0.8, 0.9\}$	10
4	`mDPLL`, `mDPLL/C`, `POPS`	Logistics	$C \in \{3, \ldots, 10\}, R \in \{1.0, 1.1\}, V \in \{2, 3, 4\}, B \in \{2, 3, 4\}, T \in \{6, 8\}$	5
5	`mDPLL`, `mDPLL/C`, `POPS`	Random	$V \in \{10, 15, \ldots, 150\}, P_P \in \{0.1, 0.2, 0.3\}, R_C \in \{2.5, 3, 3.5, 4\}$	10

Table 1: Parameter settings for the five experiments.

Logistics. Similar to the problem in §1, except the delivery requests are deterministic. Parameters are the number of cities C, the ratio of roads to cities R, the number of vehicles V, the number of delivery requests B, and the time bound T. The undirected road network is generated by uniformly scattering cities about a unit square and selecting the $\lfloor C \cdot R \rfloor$ shortest edges. The roads are disabled iff a particular Σ-variable is true. Initial positions for the trucks and boxes, and goal positions for the boxes, are selected uniformly. Trucks break down independently at random.

Random 3#∃SAT. Out of V variables, $\lfloor V \cdot P_P \rfloor$ are declared to be Σ-variables. Then we build $\lfloor R_C \cdot V \rfloor$ clauses, each with three non-conflicting literals chosen uniformly at random without replacement.

3.2 Experiments

Our experiments ran on a 32-core AMD Opteron 6135 machine with 32×4GiB of RAM, on Ubuntu 12.04. Each run was capped at 4GiB of RAM and cut off after two hours. The experiments ran for roughly 160 CPU days.[5] Table 1 shows the parameter settings. Each instance and solver pair was run only once because the solvers are deterministic.[6]

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 Σ-variables and full assignment to the $\exists$-variables—the remaining Σ-variables are unconstrained and can take on any value. We call this Σ-*independence*, and expect it to occur more frequently in lightly constrained formulas, and in formulas close to being either VALID or UNSAT.[7] `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 2.1×10^7 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:

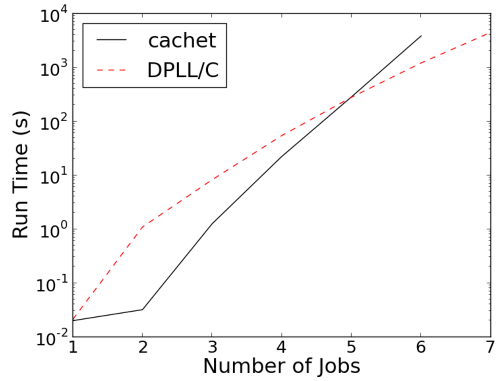

Figure 1: #SAT job shop scheduling problems with 2 machines, 2 bits of uncertainty and 4 times steps with varying numbers of jobs.

`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 Σ-assignments to Σ-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.

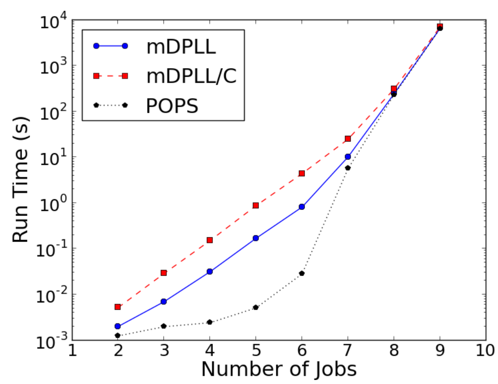

Figure 2: Log runtimes for #∃SAT job shop scheduling instances with 2 machines, 2 bits of uncertainty and 8 times steps.

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

[5] We attempted to compare our solvers to DC-SSAT [Majercik and Boots, 2005], a SSAT-based planner. We determined—after personal communication with the authors—that we are unable to faithfully represent a number of our problem instances in their slightly restricted *COPP-SSAT* language. The restrictions are reasonable for planning, but make representation of some #∃SAT formulas impossible—*e.g.*, no purely #SAT problem can be directly encoded. Consequently, performing a valid comparison with DC-SSAT is still interesting, but unfortunately out-of-scope for this paper.

[6] Randomizing might be beneficial, *e.g.*, in branching heuristics.

[7] There are exceptions. Parity formulas like $\Sigma\vec{\mathbf{x}}\exists y\ [\bigoplus \vec{\mathbf{x}}] \leftrightarrow y$ are difficult because while they are VALID, proving this requires reasoning about cases that are difficult to summarize.

dramatically faster than the other two solvers until 6 jobs.

	Jobs			3-color		
	mDPLL	mDPLL/C	POPS	mDPLL	mDPLL/C	POPS
mDPLL	-	135	2	-	190	27
mDPLL/C	0	-	0	0	-	2
POPS	136	138	-	173	198	-

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.

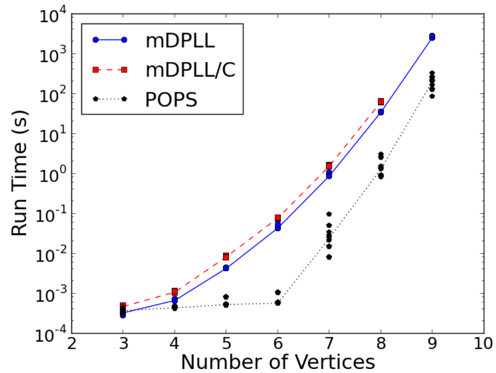

Figure 3: Log runtime for 3-Coloring instances on graphs with 70% of the possible edges. Medians are plotted as a trend line, and individual instances are plotted as points.

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. mDPLL, however, is faster than POPS for a relatively large number of the instances—especially compared to previous experiments. Instances where mDPLL 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.

	Logistics			Random		
	mDPLL	mDPLL/C	POPS	mDPLL	mDPLL/C	POPS
mDPLL	-	813	222	-	3038	1359
mDPLL/C	124	-	176	77	-	447
POPS	737	783	-	1849	2761	-

Table 3: Number of instances where the row solver beats the column solver. **Left:** based on 960 logistics instances. **Right:** based on 3360 random 3#∃SAT instances.

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

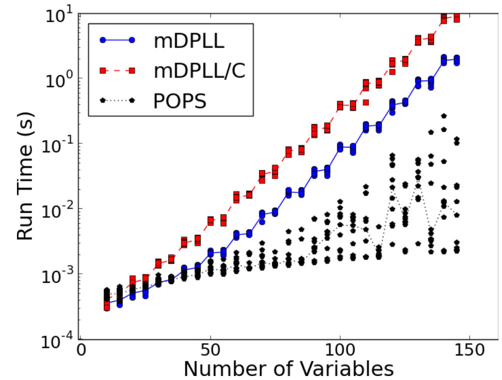

Figure 4: Log runtimes for random 3#∃SAT instances on graphs with a clause ratio of 2.5 and 10% Σ-variables. Medians are plotted as a trend line, and individual instances are plotted as points.

in 28% of such instances, and is only beaten by mDPLL 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 mDPLL and mDPLL/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, mDPLL emerges as the superior algorithm, beating POPS in 29% of such instances while POPS beats mDPLL 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 mDPLL 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

[Biere *et al.*, 2003] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[Bryant, 1992] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *CSUR*, 24(3):293–318, 1992.

[Darwiche, 2001] A. Darwiche. Decomposable negation normal form. *JACM*, 48(4):608–647, 2001.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Dell *et al.*, 2012] H. Dell, V. Kabanets, D. van Melkebeek, and O. Watanabe. Is the Valiant-Vazirani isolation lemma improvable? In *CCC*, volume 27, 2012.

[Eén and Biere, 2005] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*. Springer, 2005.

[Eén and Sörensson, 2006] N. Eén and N. Sörensson. Minisat v2.0. *Solver description, SAT race*, 2006, 2006.

[Fargier *et al.*, 1995] Hélene Fargier, Jérôme Lang, Roger Martin-Clouaire, and Thomas Schiex. A constraint satisfaction framework for decision under uncertainty. In *UAI*, pages 167–174, 1995.

[Fu *et al.*, 2004] Z. Fu, Y. Mahajan, and S. Malik. New features of the SAT04 versions of zChaff. *SAT Competition*, 2004.

[Gomes *et al.*, 1998] C.P. Gomes, B. Selman, H. Kautz, et al. Boosting combinatorial search through randomization. In *AAAI*, pages 431–437, 1998.

[Gomes *et al.*, 2007] C.P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. From sampling to model counting. In *IJCAI*, pages 2293–2299, 2007.

[Janota *et al.*, 2012] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. Solving qbf with counterexample guided refinement. In *SAT*, pages 114–128. Springer, 2012.

[Karp and Lipton, 1982] R.M. Karp and R. Lipton. Turing machines that take advice. *Enseign. Math*, 28(2):191–209, 1982.

[Kautz and Selman, 1999] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *IJCAI*, volume 16, pages 318–325, 1999.

[Kilby *et al.*, 2005] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *AAAI*, volume 20, page 1368, 2005.

[Littman *et al.*, 2001] M.L. Littman, S.M. Majercik, and T. Pitassi. Stochastic boolean satisfiability. *JAR*, 27(3):251–296, 2001.

[Majercik and Boots, 2005] S.M. Majercik and B. Boots. DC-SSAT: a divide-and-conquer approach to solving stochastic satisfiability problems efficiently. In *AAAI*, volume 20, page 416, 2005.

[Majercik and Littman, 1998] S.M. Majercik and M.L. Littman. MAXPLAN: A new approach to probabilistic planning. In *ICAPS*, volume 86, page 93, 1998.

[Majercik and Littman, 2003] S.M. Majercik and M.L. Littman. Contingent planning under uncertainty via stochastic satisfiability. *Artificial Intelligence*, 147(1):119–162, 2003.

[Marques-Silva and Sakallah, 1999] J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[Moskewicz *et al.*, 2001] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

[Nieuwenhuis *et al.*, 2006] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[Roth, 1996] D. Roth. On the hardness of approximate reasoning. *AI*, 82(1):273–302, 1996.

[Samulowitz and Bacchus, 2006] H. Samulowitz and F. Bacchus. Binary clause reasoning in QBF. *SAT*, pages 353–367, 2006.

[Sang *et al.*, 2004] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. *SAT*, 4:7th, 2004.

[Sang *et al.*, 2005] T. Sang, P. Beame, and H. Kautz. Heuristics for fast exact model counting. In *SAT*, pages 226–240. Springer, 2005.

[Sanner and Kersting, 2010] S. Sanner and K. Kersting. Symbolic dynamic programming for first-order POMDPs. In *AAAI*, 2010.

[Toda, 1991] S. Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20:865, 1991.

[Wei and Selman, 2005] W. Wei and B. Selman. A new approach to model counting. In *SAT*, pages 96–97. Springer, 2005.

[Zawadzki *et al.*, 2011] E. Zawadzki, G.J. Gordon, and A. Platzer. An instantiation-based theorem prover for first-order programming. *AISTATS*, 15:855–863, 2011.

[Zawadzki *et al.*, 2013] E. Zawadzki, G.J. Gordon, and A. Platzer. A Generalization of SAT and #SAT for Robust Policy Evaluation. Technical report, CMU, CSD, 04 2013.