Stochastic Boolean Satisfiability

Michael L. Littman (mlittman@cs.duke.edu) *

Stephen M. Majercik (majercik@cs.duke.edu) †
Department of Computer Science
Duke University, Durham, NC 27708-0129

Toniann Pitassi (toni@cs.arizona.edu) ‡
Department of Computer Science
The University of Arizona, Tucson, AZ 85721-0077

January, 2000

Abstract. Satisfiability problems and probabilistic models are core topics of artificial intelligence and computer science. This paper looks at the rich intersection between these two areas, opening the door for the use of satisfiability approaches in probabilistic domains. The paper examines a generic stochastic satisfiability problem, SSAT, which can function for probabilistic domains as SAT does for deterministic domains. It shows the connection between SSAT and well studied problems in belief network inference and planning under uncertainty, and defines algorithms, both systematic and stochastic, for solving SSAT instances. These algorithms are validated on random SSAT formulae generated under the fixed-clause model. In spite of the large complexity gap between SSAT (PSPACE) and SAT (NP), the paper suggests that much of what we've learned about SAT transfers to the probabilistic domain.

1. Introduction

There has been a recent focus in artificial intelligence (AI) on solving problems exhibiting various forms of uncertainty. In parallel, there is a great deal of work in AI and computer science on solving deterministic problems using techniques for testing Boolean satisfiability. Some recent work has looked at combinations of these ideas, viewing planning under uncertainty as stochastic Boolean satisfiability (Majercik & Littman, 1998). This paper extends these ideas, and provides a general approach for combining reasoning about uncertainty and satisfiability.

The remainder of this section reviews deterministic satisfiability, introduces the stochastic satisfiability (SSAT) framework, describes the relationship between special cases of SSAT and planning and reasoning under uncertainty, and discusses the family of random SSAT instances.

* Research supported in part by NSF CAREER grant IRI-9702576.

† Research supported by a NASA Graduate Student Research Program Fellowship.

‡ Research supported by NSF grant CCR-9457782 and US-Israel BSF Grant 95-00238.

Section 2 describes three algorithms for SSAT, one based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for SAT, a variation based on a "universal" search algorithm for SAT, and one that combines stochastic sampling and local search. Section 4 presents empirical results and puts them into perspective.

1.1. Deterministic Satisfiability

In the deterministic satisfiability problem, or SAT, we are given a Boolean formula and wish to determine whether there is some assignment to the variables in the formula that results in the formula evaluating to `True`. This problem is connected to problems throughout computer science, from circuit design and complexity theory to AI. The last several years has seen tremendous progress in our ability to solve SAT problems. In part due to the success of solving satisfiability problems, researchers are finding efficient ways of modeling diverse problems within the satisfiability framework, such as planning (Kautz & Selman, 1996).

A formal definition of the SAT decision problem follows. Let $\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$ be a collection of n Boolean (0/1) variables, and $\phi(\mathbf{x})$ be a k-CNF Boolean formula on these variables with m clauses. For example, $(x_1 \vee \overline{x_2} \vee x_4)(x_2 \vee x_3 \vee x_4)(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$ is a formula with $k = 3$ literals per clause, $n = 4$ variables and $m = 3$ clauses. An assignment α to the Boolean variables $\mathbf{x}$ is *satisfying* if it makes the formula `True` ($\phi(\alpha) = 1$) and unsatisfying if it makes the formula `False` ($\phi(\alpha) = 0$). The decision problem SAT asks, given a Boolean formula $\phi(\mathbf{x})$ in k-CNF, does it have a satisfying assignment? Or, symbolically, we want to know whether $\exists x_1, \ldots, \exists x_n (\phi(\mathbf{x}) = 1)$. Table I summarizes notation.

1.2. Stochastic Satisfiability

Papadimitriou (1985) explored an extension of SAT in which a randomized quantifier is introduced. The stochastic SAT (SSAT) problem is to evaluate a Boolean formula containing both existential and randomized quantifiers. For example:

$$\exists x_1, \text{R} y_1, \ldots, \exists x_n, \text{R} y_n (E[\phi(\mathbf{x})] \geq \theta).$$

In words, this formula asks whether there is a value for x_1 such that *for random* values of y_1 (choose 0 or 1 with equal probability), there exists a value of x_3, such that... the *expected* value of the Boolean formula $\phi(\mathbf{x})$ meets or exceeds a threshold $0 \leq \theta \leq 1$. This particular example of stochastic satisfiability consists of alternating between making a

Table I. Summary of notation used for Boolean formulae. Some notation is introduced in later sections.

$0 < n$	number of Boolean variables
$0 < k$	number of literals per clause
$0 < m \leq \sum_{i=1}^{k} \binom{2n}{i}$	number of clauses
$b \in \{0, 1\}$	Boolean value
$\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$	Boolean variable, $x_i \in \{0, 1\}$
$\phi(\mathbf{x})$	Boolean formula in CNF
$\phi\lceil_{x_i=b}$	restriction (variable assignment) of ϕ (renumbers variables)
α	assignment to $\mathbf{x}$
$\exists$, $\forall$, ꓤ	existential, universal, and randomized quantifiers
$\mathcal{F}_m^{k,n}$	fixed-clause distribution
$3 \leq s \leq 2^n$	size of smallest DPLL tree
$0 \leq w$	size of an assignment sample
$0 \leq \theta \leq 1$	stochastic satisfaction threshold
$0 \leq t \leq n$	satisfaction threshold, where 2^t is the number of assignments needed for a positive instance
$0 \leq c \leq n$	number of choice variables in E-MAJSAT
$Q(x_i) \in \{\exists, \forall, \text{ꓤ}\}$	quantifier for variable x_i

choice of value for an x variable (perhaps based on values of lower-numbered variables) and a chance selection of a value for a y variable. For this reason, Papadimitriou referred to SSAT as a "game against nature." An instance of SSAT is *positive* if the threshold is met or exceeded. This paper also considers a generalization of Papadimitriou's class, Extended SSAT, that allows universal, existential and randomized quantifiers, and a more constrained version, Alternating SSAT, in which existential and randomized quantifiers strictly alternate.

1.2.1. *Semantics of Extended* SSAT *and Decision Trees*

An Extended SSAT formula is defined by a triple (ϕ, θ, Q) where ϕ is a CNF formula with underlying ordered variables $x_1, \ldots, x_n$, $0 \leq \theta \leq 1$ is a threshold, and Q is a mapping from variables to quantifiers (existential $\exists$, universal $\forall$, and randomized ꓤ). The value of ϕ (under quantifier order Q), $val(\phi, Q)$, is defined by induction on the number of quantifiers as follows. Let x_1 be the variable associated with the outermost quantifier: (1) if ϕ contains an empty clause, then $val(\phi, Q) = 0$; (2) if ϕ contains no clauses then $val(\phi, Q) = 1$; (3) if $Q(x_1) = \exists$, then $v(\phi) = \max(val(\phi\lceil_{x_1=0}, Q), val(\phi\lceil_{x_1=1}, Q))$; (4) if $Q(x_1) = \forall$, then $val(\phi, Q) = \min(val(\phi\lceil_{x_1=0}, Q), val(\phi\lceil_{x_1=1}, Q))$; (5) if $Q(x_1) = \text{ꓤ}$, then $val(\phi, Q) = (val(\phi\lceil_{x_1=0}, Q) + val(\phi\lceil_{x_1=1}, Q))/2$.

For example, the formula

$$\rotatebox[origin=c]{180}{R}x, \exists y, \rotatebox[origin=c]{180}{R}z(E[(x \vee y)(y \vee \overline{z})(\overline{x} \vee \overline{y})] \geq 1/2).$$

has value 3/4. Now, given ϕ, Q, and a threshold θ, (ϕ, θ, Q) is `True` if and only if $val(\phi, Q) \geq \theta$.

Note that this coincides with the usual definition of the quantified Boolean formula problem (QBF), where all of the quantifiers are $\forall$ or $\exists$. The usual definition of truth for ϕ on an instance of QBF is as follows. If ϕ contains an empty clause, then ϕ is false; if ϕ contains no clauses, then ϕ is `True`; otherwise, if $\phi = \exists x_1 \phi'$, then ϕ is `True` if and only if $\phi'\lceil_{x_1=0}$ is `True`, or $\phi'\lceil_{x_1=1}$ is `True`, and finally if $\phi = \forall x_1 \phi'$, then ϕ is `True` if and only if both $\phi'\lceil_{x_1=0}$ and $\phi'\lceil_{x_1=1}$ are `True`. Using the definition of value, ϕ is `True` if and only if $val(\phi) > 0$. The base case, when ϕ contains an empty clause or no clauses clearly coincide. If $\phi = \exists x_1 \phi'$, then we define it to be `True` if and only if $\max(val(\phi'\lceil_{x_1=0}, \psi\lceil_{x_1=1}) > 0$, which is `True` if and only if $\phi'\lceil_{x_1=0}$ or $\phi'\lceil_{x_1=0}$, as desired. The definitions similarly coincide when $\phi = \forall x_1 \phi'$.

It will be useful to visualize the above inductive procedure as a decision tree. A *decision tree* over variables $x_1, \ldots, x_n$ is a binary decision tree where each internal node in the tree is labeled with a variable x_i, and the two outgoing edges are labeled with the two different settings of x_i: $x_i = 0$ and $x_i = 1$. Each leaf of the tree is labeled with either 0 or 1. Note that any total truth assignment α to the underlying variables is consistent with exactly one path of T. A decision tree T over $x_1, \ldots, x_n$ represents a CNF formula $\phi(x_1, \ldots, x_n)$ if for every truth assignment α, the value of the leaf node on the path consistent with α is equal to $\phi(\alpha)$. A decision tree T for ϕ is *complete* if it is a full binary tree of height n; notice that a given ϕ will have many decision trees that can represent it; some may be very small while others (such as the complete tree) can have exponential size in n. Figure 1 gives an example of a decision tree for $(x \vee y)(y \vee \overline{z})(\overline{x} \vee \overline{y})$.

A *canonical* decision tree for an Extended SSAT instance (ϕ, θ, Q) is a decision tree T where the ordering of the variables in T is consistent with the quantifier order. Note that if many variables occur in a contiguous block, all with the same quantifier, then these variables can be permuted within the block, both in ϕ and in the tree T without altering the value.

Given a canonical tree T for an Extended SSAT formula ϕ, the value of ϕ can be computed from T by labeling the intermediate nodes of T in a bottom-up fashion: Inductively, if v is unlabeled, and has children v', v'' with labels $L(v')$, $L(v'')$ respectively, then: (i) if $Q(v) = \exists$, then $L(v) = \max(L(v'), L(v''))$; (ii) if $Q(v) = \forall$, then $L(v) = \min(L(v'), L(v''))$; and (iii) if $Q(v) = \rotatebox[origin=c]{180}{R}$, then $L(v) = (L(v') + L(v''))/2$.

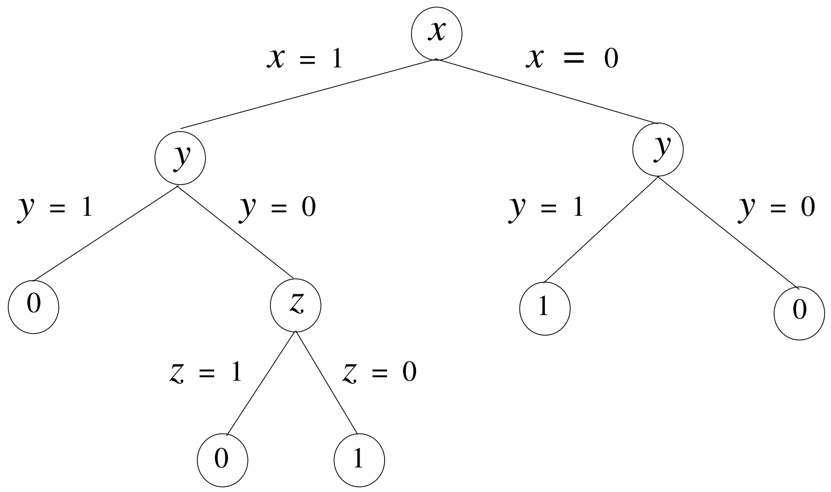


Figure 1. A decision tree for the formula $(x \vee y)(y \vee \overline{z})(\overline{x} \vee \overline{y})$ encodes which assignments are satisfying and which are not.

The value of ϕ is the value of the root, and ϕ is a positive instance if and only if $val(\phi, Q)$ is greater than or equal to θ.

Note that, while not explored in detail in this paper, the definition of the randomized quantifier Ⅎ can be extended to allow arbitrary rational probability distributions over $\{0, 1\}$ instead of the uniform distribution considered in this paper. The resulting generalization does not change any of the complexity results described in the next section, and is key to making a connection between SSat, planning under uncertainty, and belief network inference.

1.3. Satisfiability and Uncertainty

As AI techniques are used more and more to attack real-world problems, many researchers have embraced probability theory as a way of representing the pervasive uncertainty they find. Two specific examples of this trend are the increasing use of Markov decision process models in planning and learning, and belief networks in reasoning and knowledge representation; however, the influence of probabilistic models in AI is felt quite broadly.

Whereas many basic deterministic problems are complete for the well-known complexity class NP, and are therefore formally equivalent to Sat, many planning and reasoning problems in probabilistic models lie in other complexity classes, such as #P or PP (Roth, 1996; Littman, Goldsmith, & Mundhenk, 1998). This means that, with respect to the current state of complexity theory, these problems cannot be polynomially reduced to Sat. However, many standard uncertain reasoning problems can be reduced to special cases of SSAT, as described next. The probabilistic problems discussed are complete for NP, PP, $\mathrm{NP}^{\mathrm{PP}}$, and PSPACE.

The NP-complete problem SAT is the SSAT problem obtained by using only existential quantifiers and setting $\theta = 1$. The problem of finding the most probable explanation (MPE) in a belief network (Dechter, 1996) is (polynomially) equivalent to SAT. A related problem from planning under uncertainty is determining whether there is some choice of actions such that the most likely trajectory through state space to the goal exceeds a given probability threshold.

The class PP, or probabilistic polynomial time, can be defined in several ways. For the purposes of this paper, it is useful to define it by one of its complete problems, MAJSAT. In its standard formulation, MAJSAT asks, given a Boolean formula $\phi(\mathbf{x})$ in CNF, are at least half of its assignments satisfying? Thus, MAJSAT is concerned, not just with the existence of a satisfying assignment, but the *number* of satisfying assignments.

The MAJSAT problem connects satisfiability to probability in the sense that, if we imagine that all assignments are equally likely, MAJSAT asks whether the probability of a satisfying assignment is at least 1/2. Thus, MAJSAT can be expressed as an instance of SSAT:

$$\text{Я} x_1, \ldots, \text{Я} x_n (E[\phi(\mathbf{x})] \geq 1/2)$$

(or θ, more generally; Papadimitriou (1985) refers to this as "Threshold SAT", but this paper uses simply "MAJSAT" for this generalization). Thus, MAJSAT is obtained from SSAT by using only randomized quantifiers.

As with many decision problems, MAJSAT is polynomially equivalent to the problem of actually computing the probability of a satisfying assignment, since binary search on θ can be used to find the exact value of θ for which the probability is at least as big, but no bigger than, θ. Thus, the class PP can be viewed as the decision problem version of #P, which actually *counts* the number of satisfying assignments.

The problem of belief network inference—given a belief network, values for its evidence nodes, and the value for a query node, what is the probability that the query node takes on the given value given the evidence?—is #P-complete (Roth, 1996). In addition, any belief network (with rational conditional probability tables) can be represented as a Boolean formula. A PP-complete planning problem is plan evaluation in a probabilistic domain (Littman, Goldsmith, & Mundhenk, 1998).

The complexity class $\mathrm{NP}^{\mathrm{PP}}$ is formed by combining NP and PP. It is the class of problems that can be solved by guessing a solution

(NP) and then performing a PP calculation for verification. A satisfiability problem that is complete for this class is E-MAJSAT ("exists" MAJSAT) (Littman, Goldsmith, & Mundhenk, 1998), which combines elements of SAT and MAJSAT. An E-MAJSAT instance is defined by a CNF Boolean formula $\phi(\mathbf{x})$ on n Boolean variables, a threshold value θ, and a number $0 \leq c \leq n$. The variables $x_{c+1}, \ldots, x_n$ are the "chance" variables, and the decision problem is to report whether there is an assignment to the "choice" variables $x_1, \ldots, x_c$ so that the probability that they combine with the chance variables to constitute a satisfying assignment of $\phi(\mathbf{x})$ is at least θ. Thus, E-MAJSAT is also an SSAT problem:

$$\exists x_1, \ldots, \exists x_c, \text{Ⅎ} x_{c+1}, \ldots, \text{Ⅎ} x_n (E[\phi(\mathbf{x})] \geq \theta).$$

Note that, if $c = n$, E-MAJSAT is simply SAT (for $\theta = 1$). If $c = 0$, E-MAJSAT is exactly MAJSAT. In terms of complexity classes, NP $\subseteq$ PP $\subseteq$ NP^{PP}; E-MAJSAT is at least as hard as MAJSAT, which is at least as hard as SAT. In some sense, then, the peak complexity for E-MAJSAT is achieved with intermediate values of c.

Other problems have been shown to be NP^{PP}-complete, such as finding optimal size-bounded plans in uncertain domains (Littman, Goldsmith, & Mundhenk, 1998) and generating "explanations" in belief networks. In addition, the belief network problems of calculating a maximum a posteriori (MAP) hypothesis or a maximum expected utility (MEU) solution (Dechter, 1996) are also complete for NP^{PP}. In these problems, the choice variables correspond to the plan or explanation and the chance variables to the uncertainty.

The class PSPACE consists of the problems solvable using a polynomial amount of space. All the previously mentioned classes (NP, PP, NP^{PP}) can be solved in polynomial space by enumerating all assignments and combining the results in the appropriate way. The stochastic SAT problem SSAT and the deterministic QBF are satisfiability problems that are PSPACE-complete. As mentioned earlier, each existential quantifier in an SSAT problem can be viewed as a simple type of maximization operator; the problem then becomes one of maximizing the probability that $\phi(\mathbf{x})$ is satisfied, given that some of the variables are under the control of "nature." This is closely related to solving a finite-horizon partially observable Markov decision process (POMDP), which is also PSPACE-complete (Papadimitriou & Tsitsiklis, 1987). The problem remains PSPACE-complete when the domain is specified compactly via probabilistic STRIPS operators or an equivalent representation (Mundhenk *et al.*, 1997), even if the domain is "fully observable" (Littman, 1997). The connection between SSAT and these planning problems has been exploited to create a competitive plan-

Table II. Different arrangements of quantifiers result in SSAT problems complete for different complexity classes and correspond to basic problems in uncertain reasoning and planning.

class	satisfiability problem Boolean formula	belief network problem planning problem
NP	SAT $\exists x_1, \ldots, \exists x_n (E[\phi(\mathbf{x})] \geq \theta)$	most probable explanation best trajectory
PP	MAJSAT $\rotatebox[origin=c]{180}{R} x_1, \ldots, \rotatebox[origin=c]{180}{R} x_n (E[\phi(\mathbf{x})] \geq \theta)$	belief updating (inference) plan evaluation
$\mathrm{NP}^{\mathrm{PP}}$	E-MAJSAT $\exists x_1, \ldots, \exists x_c, \rotatebox[origin=c]{180}{R} x_{c+1}, \ldots, \rotatebox[origin=c]{180}{R} x_n (E[\phi(\mathbf{x})] \geq \theta)$	maximum a posteriori hypothesis best polynomial-size plan
PSPACE	Alternating SSAT $\exists x_1, \rotatebox[origin=c]{180}{R} x_2, \ldots, \exists x_{n-1}, \rotatebox[origin=c]{180}{R} x_n (E[\phi(\mathbf{x})] \geq \theta)$	influence diagrams best polynomial-horizon plan

ning algorithm (Majercik & Littman, 1999). Influence diagrams are a belief-network-like representation for the same type of problem.

Table II summarizes the relations between the complexity classes, stochastic satisfiability problems, belief network problems, and planning problems discussed.

1.3.1. *Approximability of* SSAT

For applications in planning and decision making, the underlying problem is often the related optimization problem associated with SSAT: Given an instance (ϕ, θ, Q) output $val(\phi)$. Moreover, an efficient algorithm that will return a close approximation to the actual value is nearly as good as an exact solution. Below is a brief review of what is known about the approximability of SSAT.

Let *Opt* be an optimization problem; that is, *Opt* is a real-valued function $f(x)$ with domain $D \subset \{0,1\}^*$. Let A be an algorithm that, on input $x \in \{0,1\}^*$, produces an output $A(x)$. Algorithm A *approximates* *Opt* to within ratio $0 < \epsilon(n) < 1$ if for all $x \in D$, $\epsilon(|x|) \leq A(x)/f(x) \leq 1/(\epsilon(|x|)$. If such an A exists for *Opt*, then *Opt* can be *approximated* to within ratio $\epsilon(|x|)$. Condon *et al.* (1997) observed that there exists a constant $c > 0$ such that approximating SSAT to within ratio 2^{-n^c} is PSPACE-hard. Thus, the general problem, with unbounded alternations of quantifiers, is very hard to approximate. However, what about special cases of SSAT, such as SSAT with i alternations of quantifiers where i is a small constant, independent of n? The above argument breaks down, and it is open whether or not the problem remains as hard to approximate as the general version.

Similarly, Lusena, Goldsmith, & Mundhenk (1998) found that approximating solutions to Markov decision processes to within ratio δ is also intractable.

1.4. Random Formulae

To empirically evaluate algorithms for satisfiability and stochastic satisfiability, it is useful to have a good testbed of hard formulae. This is a nontrivial problem: specific hand-coded instances of Sat are known to be satisfiable or not, and usually for a clear (easily computable) reason. What is sought is a natural distribution over Sat instances that is hard on average. This has led many researchers to study randomly generated formulae (although this distribution is not known to be average-case hard). The usual fixed-clause random k-CNF model is used in this paper, which is defined in terms of three integer parameters n, k and m; a formula is generated by selecting m clauses of size k independently. A clause is generated by randomly selecting one of n variables k times, rejecting duplicate selections, and randomly negating it with probability 1/2. This distribution is denoted $\mathcal{F}_m^{k,n}$ and the ratio m/n is referred to as the *clause density.*

The random k-CNF model has been widely studied for several good reasons. First, it is an intrinsically natural model, analogous to the random graph model, that sheds light on fundamental structural properties of the satisfiability problem. As with Sat, it is possible to gain insight into the mathematical structure of SSat by looking at the behavior of randomly generated formulae. Second, using an appropriate choice of parameters, randomly chosen formulae are empirically difficult for satisfiability algorithms, and are a commonly used benchmark (Gu *et al.*, 1997). The next section briefly reviews some analytical results concerning random Sat and Majsat instances.

1.4.1. *Random* Sat

An interesting property of random k-CNF formulae is that when m is small relative to n, almost all formulae are satisfiable, and most algorithms find it easy to show this satisfiability. When m is large relative to n, almost all formulae are unsatisfiable and unsatisfiability is relatively easy to show. For intermediate values of m (around $4.2n$ for $k = 3$), approximately half of the formulae are satisfiable and it is very difficult to show satisfiability. Thus, with respect to m, random Sat instances exhibit an “easy-hardest-hard” pattern. This pattern is consistent for various values of n but becomes more pronounced with larger n. In addition, the hard region is very large for large n.

A key conjecture about the random k-CNF SAT formula model is that there exists a constant α_k such that a random k-CNF formula from $\mathcal{F}_m^{k,n}$ is almost certainly satisfiable for $m/n < \alpha_k$ (as n gets large), and almost certainly unsatisfiable if $m/n > \alpha_k$. As mentioned above, empirical evidence (Selman, Mitchell, & Levesque, 1996) suggests that this 0–1 threshold value for $k = 3$ occurs at around 4.2, and a recent theoretical result by Friedgut (1997) shows that for each n, there is a sharp 0–1 threshold $\alpha_k(n)$ that possibly varies with n. It is known that $\alpha_2 = 1$ (Chvátal & Reed, 1992; Goerdt, 1996), and that for $k = 3$, $3.003 \leq \alpha_k(n) \leq 4.601$ (Frieze & Suen, 1996; Kirousis *et al.*, 1998).

An interesting feature of the threshold is that the dual problems of bounding the threshold value from above and below appear to be asymmetric. At slightly below the 0–1 threshold, there are simple linear-time algorithms for finding a satisfying assignment, but at slightly above the 0–1 threshold, there are no known algorithms that quickly produce proofs of unsatisfiability. Moreover, it has been proven that no Resolution-based method (including the DPLL procedure that forms the basis of the algorithms in this paper) will succeed in sub-exponential time on a broad range of randomly generated formulae (Chvátal & Reed, 1992; Beame & Pitassi, 1996; Beame *et al.*, 1998). In particular, Beame *et al.* (1998) show that for $k = 3$ and $m \geq n^2/\log n$, almost certainly a random formula has a polynomial-size decision tree, but for $m < n^{6/5}$, almost certainly a random formula requires a superpolynomial-size decision tree. This suggests that DPLL will perform very badly (asymptotically in n) even for ranges of m way above the 0–1 threshold (up to $n^{6/5}$), and that the easy-hardest-hard pattern observed empirically may actually be misleading—for large enough n, the expected pattern is more like easy-hardest-hardest-hardest-easy. This points out that when analyzing algorithms for random instances of SAT it is important to know whether the algorithm is applied above, below, or at the threshold.

1.4.2. *Random* MAJSAT

A random MAJSAT instance is defined in terms of four integer parameters, k, n, m, and t. A random MAJSAT formula with these parameters is obtained by selecting a k-CNF formula uniformly and at random from the distribution $\mathcal{F}_m^{k,n}$. The instance is positive if and only if the probability of a satisfying assignment is at least $\theta = 1/2^{n-t}$, that is, there are at least 2^t satisfying assignments.

Let $f(k, n, m, t)$ be the probability that a random k-CNF formula on n variables with m clauses has at least 2^t satisfying truth assignments.

Given m and a function $t(n)$ of n, what can be said about the limit of f as n gets large? Surprisingly, in spite of a number of articles concerning the counting of satisfying assignments, there are very few analytical results known about this function f, and its 0–1 threshold behavior is far from determined. Some simple results are summarized next.

The expected number of satisfying assignments for a random 3-CNF formula from $\mathcal{F}_m^{3,n}$, is $2^n(7/8)^m = (2(7/8)^{m/n})^n$. For $m \geq (\log_{8/7} 2)n$ (about $5.19n$), $2(7/8)^{m/n} < 1$, and, thus, by Markov's inequality, with probability $1 - o(1)$, a random formula with parameters $(3, n, m, t = 0)$ is unsatisfiable for m in this range. What happens to this calculation for larger values of t? We want to calculate for what values of m does $(2(7/8)^{m/n})^n 2^{-t}$ go to zero as n goes to infinity. This will happen whenever $2(7/8)^{m/n} < 2^{t/n}$. Solving for m, whenever

$$m \geq (\log_{8/7} 2)(n - t) \approx 5.19(n - t), \tag{1}$$

a random 3-CNF formula with m clauses and n variables almost certainly has fewer than 2^t satisfying assignments. As expected, as t increases, fewer random clauses are needed to generate an instance that is not positive. Empirical estimates of the 0–1 threshold in Section 3.1 confirm the form of Equation 1.

While the above calculation gives an upper bound on m as a function of t and n, this upper bound is likely larger than the actual number. In several papers (Kamath *et al.*, 1995; Kirousis *et al.*, 1998), it has been demonstrated that the expected number of satisfying assignments is larger than the number of satisfying assignments almost certainly, because for those formulae that are satisfiable, they actually have many satisfying assignments (thus skewing the average). In particular, Kamath *et al.* (1995) show that if a random k-CNF formula is satisfiable, then with exponentially high probability (probability at least $1 - 2^{-\epsilon n}$) it has an exponential number of satisfying truth assignments. Let Z be the number of variables that are not mentioned at all when m clauses are chosen randomly. The expectation of Z, $E[Z]$, is $n(1-1/n)^{3m}$, which is approximately $ne^{-3m/n}$. Kamath *et al.* (1995) show amazingly tight tail bounds on this quantity. In fact, the weakest of their results shows that the probability of being more than a constant factor away from the expectation is exponentially small in n. Thus, if ϕ is a random formula where m is below the 0–1 threshold, then almost certainly ϕ will be satisfiable. Thus, by the above result, almost certainly ϕ will have at least $2^{ne^{-3m/n}}$ satisfying assignments, since each of the variables that are not mentioned at all can be set in any possible way. Thus, almost certainly, whenever m is less than the 0–1 threshold, a random MAJSAT instance with parameters $(3, m, n, t)$ is positive whenever $t \leq ne^{-3m/n}$.

The above argument states that while the expected number of satisfying assignments is $(2(7/8)^{m/n})^n$, almost certainly the number of satisfying assignments will be substantially below the expected value. The beautifully simple argument of Kirousis *et al.* (1998) demonstrates that this is not the only source of wastefulness in these calculations. (See the thesis of Achlioptas (1999) for more details.)

What can be said of a lower bound on m as a function of t? Here again, very little seems to be known. Assume that almost certainly, there are l disjoint clauses, $C_1, C_2, \ldots, C_l$ in a random MAJSAT formula with parameters k, m, n, t.

Let $k = 3$. There are 2^{3l} total assignments for the l disjoint clauses. There are 7 ways to make each clause `True` for 3-CNF formulae. Since the clauses are disjoint, there are 7^l ways to make all the clauses true, so the exact number of assignments that make one or more clauses `False` is $2^{3l} - 7^l$, which is between $2^{3(l-1)}$ and 2^{3l}. This gives a lower bound on m as a function of t, since l can be chosen as a function of m and n such that almost certainly a 3-CNF random formula with m clauses over n variables contains at least l variable-disjoint clauses.

The above gives very coarse upper and lower bounds on the value of m for which a random MAJSAT formula is `False`/`True` almost certainly as a function of t. Do we know that $f(k, m, n, t)$ actually exhibits a sharp 0–1 threshold behavior? As mentioned earlier, Friedgut (1997) has proven that random SAT has a sharp 0–1 threshold, although the 0–1 threshold value α_k may be a function of n. The ideas in his proof are general enough that they extend to MAJSAT. In other words, from Friedgut (1997), it follows that there exists a function $\alpha_k(n)$ such that for any $\epsilon > 0$, there exists $t > 1$ such that a random MAJSAT instance with parameters (k, m, n, t), where $m = (\alpha_k(n) - \epsilon)n$, is positive with high probability, while a random formula with parameters (k, m, n, t), where $m = (\alpha_k(n) + \epsilon)n$, is not a positive instance with high probability, and moreover has zero satisfying truth assignments with high probability. Therefore, although this line of argument doesn't lead to a prediction of where the 0–1 threshold is, or even whether it is a linear function of n, it does imply that it exists.

For random instances of more general SSAT problems, there are more parameters, since the validity of the formula is highly dependent on the exact sequence of quantifiers. To date, there are no theoretical results concerning the threshold behavior of random instances of general SSAT.

2. Algorithms for Stochastic Satisfiability

The SSAT problem is PSPACE complete, and therefore would seem to require a different set of algorithmic approaches from those developed for the NP-complete SAT. In fact, this section shows that the fundamental algorithmic ideas developed in the SAT community provide an excellent starting point for SSAT algorithms—the gap between NP and PSPACE may not be as large as it seems.

This section describes three algorithms for SSAT. Section 2.1 develops a DPLL-based algorithm. It shows that the basic structure of the SAT DPLL algorithm is applicable to Extended SSAT.

Section 2.2 describes another algorithm that is similar to the DPLL-based algorithm, but that has a guarantee of being efficient with respect to other decision-tree-based algorithms. This type of algorithm can be viewed as a "universal" search algorithm for DPLL-based procedures. The algorithm is guaranteed to solve the Extended SSAT instance in time that is efficient in the size of the smallest decision tree. As discussed in Section 4, although the theoretical performance of the universal algorithm is superior, it does not perform as well on random formulae as the DPLL algorithm.

The algorithms in Section 2.1 and Section 2.2 are both sound and complete algorithms: given an arbitrary Extended SSAT instance (ϕ, θ, Q), both of these algorithms are guaranteed to return the correct answer, although the running time can be exponential. As with SAT, incomplete or approximate algorithms can often solve large problems more quickly than complete algorithms like DPLL. For SAT, randomized local search procedures like WALKSAT have been shown to be successful for large problem instances. Similarly, MAJSAT instances are very naturally approximated using random sampling. Section 2.3 describes a new SSAT algorithm that combines local search with random sampling to compute approximate answers to SSAT problems (existential and randomized quantifiers only).

2.1. A Davis-Putnam-Logemann-Loveland Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm for Boolean satisfiability (Davis, Logemann, & Loveland, 1962) works by enumerating partial assignments and monitoring for opportunities to simplify the formula. These simplifications, or pruning rules, make it possible to solve problems whose entire set of assignments could not be fully enumerated.

DPLL is designed to solve SAT problems, and, thus, only needs to deal with existential quantifiers. The algorithm described in this section can be viewed as an extension of the DPLL algorithm to Extended SSAT by providing pruning rules for universal and randomized quantifiers. Cadoli, Giovanardi, & Schaerf (1998) provide a set of pruning rules for universal quantifiers (for solving QBF problems); the purification and unit propagation pruning rules described here are introduced and justified there. Birnbaum & Lozinskii (1999) describe a similiar adaptation to DPLL for counting satisfying assignments.

Define $\phi' = \phi\lceil_{x_i=b}$, where ϕ' is the $(n-1)$-variable CNF formula obtained from assigning the single variable x_i the Boolean value b in the n-variable CNF formula ϕ and simplifying the result (including any necessary variable renumbering). Variables are numbered so that x_1 corresponds to the outermost (leftmost) quantifier and x_n to the innermost. Recall that $Q(x_i)$ is the quantifier associated with variable x_i.

The evalssat algorithm in Figure 2 takes formula ϕ and low and high thresholds θ_l and θ_h. It returns a value less than θ_l if and only if the value of the Extended SSAT formula is less than θ_l, a value greater than θ_h if and only if the value of the Extended SSAT formula is greater than θ_h, and otherwise the exact value of the Extended SSAT formula. Thus, it can be used to solve the Extended SSAT decision problem by setting $\theta_l = \theta_h = \theta$. It can also be used to compute the exact value of the formula by setting $\theta_l = 0$ and $\theta_h = 1$. The algorithm's basic structure is to compute the value of the Extended SSAT formula from its definition (Section 1.2.1); this takes place in the section of the pseudocode labeled "*Splitting*", which enumerates all assignments, applying operators recursively from left to right. However, it is made more complex (and efficient) by a set of pruning rules, described next.

2.1.1. *Unit Resolution*

When a Boolean formula ϕ is evaluated that contains a variable x_i that appears alone in a clause in ϕ with *sign* b (0 if $\overline{x_i}$ is in the clause, 1 if x_i is in the clause), the normal left-to-right evaluation of quantifiers can be interrupted to deal with this variable. This is called *unit resolution*, by analogy with DPLL.

If the quantifier associated with x_i is existential, x_i can be eliminated from the formula by assigning it value b and recursing. As in DPLL, this is because assigning $x_i = 1 - b$ is guaranteed to make ϕ `False`, so $x_i = b$ can be no worse. Similarly, if the quantifier associated with x_i is randomized, it is the case that one branch of the computation will return a zero, so x_i can be eliminated from the formula by assigning it value b and recursing. The resulting value is divided by two (or multiplied by the probability associated with the randomized quantifier, more generally), since it represents the value of only one branch.

```
evalssat(φ, Q, θ_l, θ_h) := {
  if φ is the empty set, return 1
  if φ contains an empty clause, return 0
  /* Unit Resolution */
  if x_i is a unit variable with sign b and Q(x_i) = ∃,
    return evalssat(φ⌈_{x_i=b}, Q, θ_l, θ_h)
  if x_i is a unit variable and Q(x_i) = ∀,
    return 0
  if x_i is a unit variable with sign b and Q(x_i) = Ʀ,
    return evalssat(φ⌈_{x_i=b}, Q, 2θ_l, 2θ_h)/2
  /* Purification */
  if x_i is a pure variable with sign b and Q(x_i) = ∀,
    return evalssat(φ⌈_{x_i=1-b}, Q, θ_l, θ_h)
  if x_i is a pure variable with sign b and Q(x_i) = ∃,
    return evalssat(φ⌈_{x_i=b}, Q, θ_l, θ_h)
  /* Splitting */
  if Q(x_1) = ∃, {
    v_0 = evalssat(φ⌈_{x_1=0}, Q, θ_l, θ_h)
    if v_0 ≥ θ_h, return v_0
    v_1 = evalssat(φ⌈_{x_1=1}, Q, max(θ_l, v_0), θ_h)
    return max(v_0, v_1)
  }
  if Q(x_1) = ∀, {
    v_0 = evalssat(φ⌈_{x_1=0}, Q, θ_l, θ_h)
    if v_0 < θ_l, return v_0
    v_1 = evalssat(φ⌈_{x_1=1}, Q, θ_l, min(v_0, θ_h))
    return min(v_0, v_1)
  }
  if Q(x_1) = Ʀ, {
    v_0 = evalssat(φ⌈_{x_1=0}, Q, 2θ_l - 1, 2θ_h)
    if (v_0 + 1)/2 < θ_l, return v_0/2
    if v_0/2 ≥ θ_h, return v_0/2
    v_1 = evalssat(φ⌈_{x_1=1}, Q, 2θ_l - v_0, 2θ_h - v_0)
    return (v_0 + v_1)/2
  }
}
```

Figure 2. The evalssat algorithm generalizes the DPLL algorithm for satisfiability to solve Extended SSAT problems.

On the other hand, if the quantifier associated with x_i is universal, it is immediate that the entire formula is `False` since it won't be the case that all values of x_i make the formula `True`. The algorithm can return a zero in this case. This is stronger than the pruning possible for either existential or randomized variables.

2.1.2. *Purification and Thresholding*

The *purification* pruning rules apply when there is a variable x_i that appears only with one sign b in ϕ. If $Q(x_i) = \exists$, the algorithm assigns $x_i = b$ and recurses. This is valid because any clause satisfied by an assignment with $x_i = 1-b$ will also be satisfied by assigning $x_i = b$. By the same reasoning, if $Q(x_i) = \forall$, the algorithm recurses after assigning $x_i = 1 - b$. Interestingly, purification pruning does not appear possible for randomized variables. Both assignments to a randomized variable give *some* contribution to the value of the Extended SSAT formula, and must be considered independently.

Another useful class of pruning rules concerns the *threshold* parameters θ_l and θ_h. While some care must be taken to pass meaningful thresholds when applying unit resolution, threshold pruning mainly comes into play when variables are split to try to prevent recursively computing both assignments to x_1, the outermost quantified variable.

If $Q(x_1) = \exists$, after the first recursive call computing v_0, it is possible that θ_h has already been exceeded. In this case, the algorithm can simply return v_0, without ever computing v_1. In particular, it is possible that $v_1 > v_0$, but all that is significant is whether the larger of the two exceeds θ_h. If v_0 exceeds θ_l but falls short of θ_h, this can be used to increase the lower threshold for the recursive computation of v_1; since the algorithm must take the larger of v_0 and v_1, the precise value of v_1 is not needed if it less than v_0.

The complement of this argument provides a threshold pruning rule for universal quantifiers.

Threshold pruning is not as strong for randomized variables, although it can be done. There are two types of threshold pruning that apply. First, if assigning 0 to x_1 is sufficient to meet the threshold θ_h, then the algorithm need not recurse on assigning the variable to 1: if $v_0/2 \geq \theta_h$, return $v_0/2$.

If the first value v_0 is so low that, even if $v_1 = 1$, $(v_0 + v_1)/2 < \theta_l$, then again the algorithm need not compute v_1. If both tests fail, the algorithm needs to compute v_1, but can adjust the thresholds accordingly.

All of these pruning rules are incorporated into the algorithm given in Figure 2. Appendix A gives a correctness proof for the algorithm.

2.1.3. *Complexity of* evalssat

Not surprisingly, the worst-case running time of evalssat is exponential in the size of the formula. The reason is that on even simple instances, often the underlying decision tree is always large, and in addition, pruning does not help. A particular example of this is applying evalssat to unsatisfiable SAT formulae that are known to require exponential size Resolution proofs. Since the algorithm (in this special case) produces a Resolution refutation, the Resolution lower bounds for specific formulae (such as the propositional pigeonhole principle) imply exponential time lower bounds on the running time of evalssat on these examples.

It is also interesting to understand the average-case running time of evalssat and its performance on more general formulae, such as instances of MAJSAT or Alternating SSAT. This question can be partially answered in the context of random formulae, yielding results on the complexity of the DPLL algorithm assuming no pruning rules, that is, assuming that the algorithm needs to generate the entire decision tree before terminating. This is interesting, since for many hard formulae the entire decision tree must be generated in order to determine truth or falsity of the formula, and, thus, lower bounds on the decision tree size yield lower bounds on the running time of the algorithm. The following theorem follows from the results of Beame *et al.* (1998).

THEOREM 1. *Let ϕ be a 3-CNF formula chosen uniformly at random from $\mathcal{F}_m^{k,n}$. For all n sufficiently large, for all m, $n \leq m \leq n^{5/4}$, with high probability any decision tree for ϕ requires exponential size in n.*

This theorem shows that for random formulae in a large range, the decision tree-complexity of the underlying 3-CNF formula will be huge, and therefore the running time of any decision-tree based algorithm without pruning will be exponential. To make matters worse, the DPLL algorithm above searches for a decision tree in a very specific way; that is, it is not guaranteed to even search for the smallest tree. This leads to the following interesting question: Given a CNF formula ϕ, is it possible to efficiently produce a decision tree for ϕ such that the tree produced is close to an optimal one? An affirmative answer to this question suggests a more efficient algorithm for SAT, and moreover suggests that if one is using a decision-tree based method, that an "optimal" algorithm within this class exists. The next section describes such an algorithm: it is guaranteed to find a close-to-optimal size tree efficiently.

2.2. A Universal Search Algorithm for Extended SSat

As indicated earlier, it may be possible to determine whether a MAJSAT instance (ϕ, θ, Q) is positive without actually constructing the entire decision tree, via pruning. On the other hand, to determine the exact value of ϕ, the entire decision tree must be examined, and therefore a lower bound on the best possible running time of the algorithm described here will be a linear function of the size of the smallest decision tree for ϕ. Thus, it is useful to understand how hard it is to search for the smallest such decision tree.

If ϕ is an Extended SSAT formula, then ϕ can be determined to be `True` or `False` by examining a canonical decision tree for ϕ as described above. Given an Extended SSAT instance ϕ, the evalssat algorithm described in Section 2.1 actually searches through a decision tree that is more general than a canonical decision tree, but has the following properties: for every i, j $i < j$, every variable in the block X^i is split upon in the decision tree before every variable in the block X^j, with the possible exception of variables that appear in unit clauses or pure variables. A decision tree for ϕ with these properties will be called a *DPLL tree* for ϕ. The evaluation of a DPLL tree for ϕ is again obtained by evaluating each vertex, starting at the leaves, in the manner described in Section 1.2.1.

It was shown implicitly by Clegg, Edmonds, & Impagliazzo (1996) and more explicitly by Beame & Pitassi (1996) and Ben-Sasson & Wigderson (1999) that there is a relatively efficient deterministic procedure for finding a small tree-like Resolution proof. Following the proof of their theorem, it can be shown that there is an efficient deterministic procedure for finding a small DPLL tree.

THEOREM 2. *There is a deterministic algorithm that takes as input an Extended* SSAT *formula ϕ with m clauses and n underlying variables and outputs a DPLL decision tree, T for ϕ. Moreover, the running time of the algorithm is at most $n^{O(\log s)}O(m)$, where s is the size of the smallest DPLL decision tree for ϕ.*

Proof: The proof follows Beame *et al.* (1998), with small modifications. Let (ϕ, θ, Q) be an instance of Extended SSAT with κ blocks of variables $X^1, \ldots, X^\kappa$, where each variable within a block is quantified the same way. Assume that ϕ has a size s DPLL decision tree, and assume without loss of generality that the root node is labeled by the variable x. It follows that one of the two subtrees has size at most $s/2$. Therefore, the algorithm can search through all literals l occurring in the first block until it finds one, l^*, such that $\phi\lceil_{l^*=1}$ has a decision tree of size at most $s/2$. The tree T constructed for ϕ will, thus, have root labeled with l^* (not necessarily x, but not much worse, as indicated by the overall running time).

The main algorithm, **DPLLsearch** takes as input a Boolean formula ϕ. If there is a literal l occurring in a unit clause, then the output of **DPLLsearch**(ϕ) consists of a DPLL tree for $\phi\lceil_{l=1}$ followed by a DPLL tree for $\phi\lceil_{l=0}$, and then these two trees are combined by adding a top node and splitting on l. Otherwise (if there is no unit clause), then for each of the literals l occurring in the current block (the leftmost block of variables that have not already been set), apply **DPLLsearch** to the formula $\phi\lceil_{l=1}$ to identify the literal l for which **DPLLsearch**($\phi\lceil_{l=1}$) terminates fastest.

These calls to **DPLLsearch** are executed in a sequence of parallel rounds so that in round i the ith step of each of the calls is performed. In other words, if $l_1, \ldots, l_k$ are the variables in the current block, then the algorithm will execute the k recursive calls, **DPLLsearch**($\phi\lceil_{l_i=1}$), $i = 1, \ldots, k$, in a dovetail fashion, executing each for one time step, and then each for two time steps, etc. The first time that one of the recursive calls terminates, say for literal l^* **DPLLsearch**($\phi\lceil_{l^*=1}$) terminates, then we abort all other calls, except for the call corresponding to $\overline{l^*}$, which is run to completion.

The output of **DPLLsearch**(ϕ) consists of a DPLL tree for $\phi\lceil_{l^*=1}$ followed by a DPLL tree for $\phi\lceil_{l^*=0}$, and then these two trees are combined by adding a top node and splitting on l^*.

The analysis of the algorithm follows that of Beame *et al.* (1998). Let $T_1(n, s; m)$ denote the maximum running time of **Treesearch**(ϕ) over all formulae ϕ with at most n variables, m clauses and such that there exists a decision tree for ϕ of size at most s. Consider a decision tree of size at most s and let x be the splitting variable at the root. The left and right branches of the tree give decision trees for $\phi\lceil_{x=0}$ and $\phi\lceil_{x=1}$, and the smaller of these is of size at most $s/2$. Hence, at least one of the recursive calls terminates after at most $T_1(n-1, s/2; m)$ steps, and so the literal l^* is found after at most that number of rounds. The time for each round is at most Cn for some constant C. Once l^* is found, it takes at most $T_1(n-1, s; m)$ steps to complete the call to **Treesearch**($\phi\lceil_{l^*=0}$). Thus, for fixed m, the following recurrence relation is obtained: $T_1(n, s) \leq CnT_1(n-1, s/2) + T_1(n-1, s)$ whenever $n \geq 1$ and $s > 1$; $T_1(0, s) = 1$ and $T_1(n, 1) = \text{time}(O(m))$, which is the time to compute the value of a CNF formula with m clauses under a given truth assignment. It can be shown by induction that $T_1(n, s) \leq O(mn^{O(\log s)})$. ■

For SAT and MAJSAT formulae, any decision tree is a DPLL tree, but for more complicated Extended SSAT instances, the DPLL tree is more restricted since it must query variables in their order of quantification. In particular, it might be that for a given formula ϕ, there is a small decision tree for ϕ but not a small DPLL tree for ϕ, because of the quantifier ordering. Unfortunately, the following theorem due to Umans (1999) shows that an arbitrary decision tree cannot be used to evaluate an Extended SSAT formula efficiently.

THEOREM 3. (Umans (1999)). *Let* QBF(i) *denote the complexity class of deciding a Extended* SSAT *formula with only* $\exists$ *and* $\forall$ *quantifiers, and with* i *quantifier alternations. Let* QBF*-tree*(i) *be the following problem. The input is a pair* (ϕ, T)*, where* ϕ *is a* QBF(i) *formula and where* T *is a decision tree that captures the CNF part of the formula* ϕ *(ignoring quantifiers). Then,* QBF*-tree*(i) *is in* P *if and only if* QBF$(i-1) = $ P*.*

In other words, the above theorem tells us that it is impossible to determine in polynomial time the value of an Extended SSAT formula given an *arbitrary* polynomial-size decision tree for the CNF part of the formula, unless a surprising collapse of complexity classes occurs. Nonetheless, there are somewhat less restricted decision trees than DPLL trees that could be used to evaluate an Extended SSAT formula. For instance, variables can be skipped altogether and still the values of the internal nodes can be calculated, as long as the variables that are mentioned are consistent with the quantification ordering (again, with the exception of unit resolution). However, at this time, there is no method known to efficiently search for these more general types of trees.

2.3. A Stochastic Sampling Algorithm

Randomized approximation algorithms have been an extremely successful approach to solving difficult satisfiability problems. For SAT, randomized local search algorithms like WALKSAT (Kautz & Selman, 1996) have been able to solve some very large problems. WALKSAT initially makes a random assignment to the variables in the SAT formula. If this is not a satisfying assignment, it randomly selects an unsatisfied clause. If it can satisfy this clause by flipping a variable without unsatisfying any other clauses, it does so. If not, it randomly flips that variable that would unsatisfy the fewest clauses. This process continues until a satisfying assignment is found or until a specified maximum number of variables have been flipped. This entire process can also be repeated a specified number of times starting with a new random assignment. Clearly, WALKSAT is not complete; it may not find a satisfying assignment when one exists. Further, it cannot determine when a satisfying assignment does not exist. But, compared to systematic solvers, randomized local search algorithms can solve random satisfiability problems that are orders of magnitude larger (Selman, Kautz, & Cohen, 1996).

For MAJSAT problems, random sampling is an obvious approach. Some number of assignments to the random variables are generated according to their probabilities; the probability that the formula is satisfiable can be extrapolated from the sample.

This section develops an approximation algorithm for SSAT problems by combining stochastic sampling (Section 2.3.2) with randomized local search (Section 2.3.3).

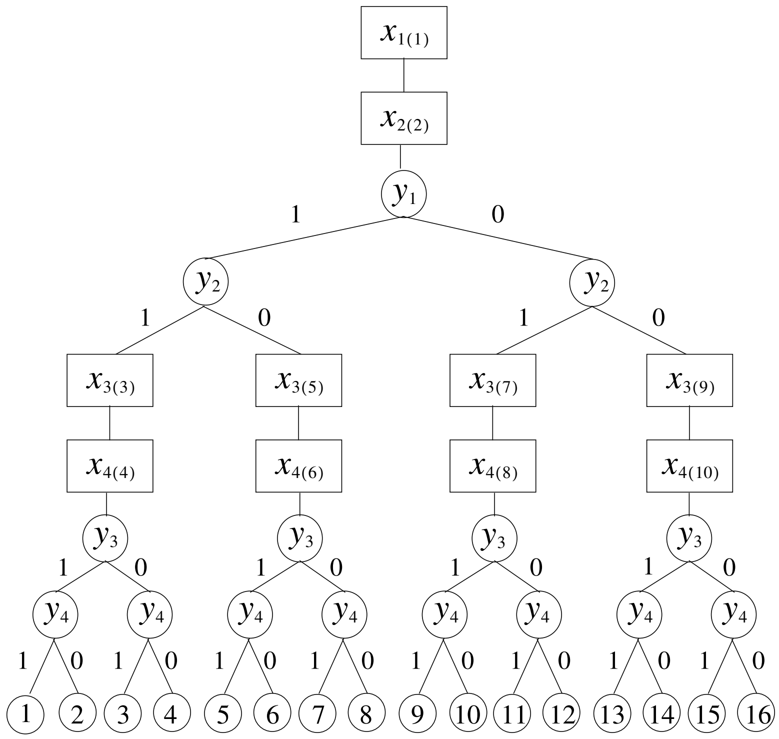
2.3.1. *Policy Trees*

The starting point for the stochastic sampling algorithm for SSAT is the policy tree representation. Figure 3 provides an example policy tree for the SSAT instance

$$\exists x_1, \exists x_2, \rotatebox[origin=c]{180}{R} y_1, \rotatebox[origin=c]{180}{R} y_2, \exists x_3, \exists x_4, \rotatebox[origin=c]{180}{R} y_3, \rotatebox[origin=c]{180}{R} y_4, E[(\overline{x_1} \vee x_3 \vee \overline{y_3})(\overline{x_2} \vee \overline{x_4} \vee y_2)(x_3 \vee y_1 \vee \overline{y_4})] \geq \theta.$$

Each existential variable can take on a different Boolean value as a function of the assignment to the randomized variables that appear to its left in the quantifier ordering. The *policy tree* representation makes this evident by including a copy of the variable for each of these assignments. Existential variables, and their copies, can be called *decision variables* and are represented as rectangular *decision nodes* in the policy tree. If x_i is an existential variable such that r randomized variables are to its left, then there will be 2^r copies of x_i in the policy tree representation. In the policy tree in Figure 3, for example, two randomized variables (y_1 and y_2) appear to the left of the existential variables x_3 and x_4 in the quantifier ordering, so there are four copies of x_3 and x_4. To emphasize that the value of each instance of a copied variable can be set independently, these variables are renumbered in the policy tree such that, given two decision nodes at the same level in the tree (and, therefore, containing the same variables), the variables in one decision node will be distinct from the variables in the other decision node. The renumbering of a variable appears in a parenthesized subscript next to the original number. Thus, $x_{3(7)}$ is the relevant copy of variable x_3 when y_1 is `False` and y_2 is `True`. Note that the existential variables in the simplified formulae in Figure 3 are subscripted only with the new numbers.

Each leaf of the policy tree represents a partial assignment consisting of an assignment to all randomized variables in the SSAT formula. The *probability of a leaf* is $1/2^R$ in a formula with R randomized variables (more generally, it is the product of the probabilities of the outcomes



Original Formula: $(\bar{x}_1 \vee x_3 \vee \bar{y}_3)(\bar{x}_2 \vee \bar{x}_4 \vee y_2)(x_3 \vee y_1 \vee \bar{y}_4)$

Simplified Formulae at Leaves Given Random Variable Assignments

Leaf 1:	$(\bar{x}_1 \vee x_3)$	Leaf 9:	$(\bar{x}_1 \vee x_7)(x_7)$
Leaf 2:	$(\bar{x}_1 \vee x_3)$	Leaf 10:	$(\bar{x}_1 \vee x_7)$
Leaf 3:	Satisfied	Leaf 11:	(x_7)
Leaf 4:	Satisfied	Leaf 12:	Satisfied
Leaf 5:	$(\bar{x}_1 \vee x_5)(\bar{x}_2 \vee \bar{x}_6)$	Leaf 13:	$(\bar{x}_1 \vee x_9)(\bar{x}_2 \vee \bar{x}_{10})(x_9)$
Leaf 6:	$(\bar{x}_1 \vee x_5)(\bar{x}_2 \vee \bar{x}_6)$	Leaf 14:	$(\bar{x}_1 \vee x_9)(\bar{x}_2 \vee \bar{x}_{10})$
Leaf 7:	$(\bar{x}_2 \vee \bar{x}_6)$	Leaf 15:	$(\bar{x}_2 \vee \bar{x}_{10})(x_9)$
Leaf 8:	$(\bar{x}_2 \vee \bar{x}_6)$	Leaf 16:	$(\bar{x}_2 \vee \bar{x}_{10})$

Figure 3. A policy tree represents the set of contingent choices in an SSAT problem.

along the path from the root to the leaf). A *policy* is an assignment of Boolean values to the existential nodes in the policy tree (boxes in Figure 3). Given a policy, all the root-to-leaf paths in the policy tree represent complete assignments to the variables in the formula, each of which is either satisfying (value 1) or unsatisfying (value 0). The *value* *of a policy* is the weighted sum of the probabilities of the satisfied leaves. For example, the policy that assigns "0" to all existential nodes in the policy tree in Figure 3 has value 3/4 (leaves 9, 11, 13, and 15 are unsatisfied).

The *value of a policy tree* is the maximum over all policies of the policy values. The value of the policy tree in Figure 3 is 1 because the policy $x_{1(1)} = 0$, $x_{2(2)} = 0$, $x_{3(7)} = 1$, $x_{3(9)} = 1$ satisfies all leaves (unmentioned variables can take on either value). Note that a policy tree is simply an alternative representation of the DPLL tree for a formula (Section 2.2), and the value of a policy tree is exactly that of a DPLL tree for the formula. However, it is a convenient representation for organizing the sampling algorithm described next.

2.3.2. *Stochastic Sampling*

Given a fixed policy, its value can be computed in linear time in the size of the policy tree. However, in a formula with R randomized variables, the size of the policy tree is roughly 2^R. An accurate estimate of the value of the policy can be obtained by randomly sampling the leaves. Choose a set W of w assignments to the randomized variables proportional to their probability and independently of the policy. Estimate the value of the policy as the proportion of the w sampled leaves that are satisfied by the policy. Let v be the true value of the policy and $\hat{v}$ be the estimate found via sampling.

LEMMA 4. *Let $\epsilon > 0$ be a target approximation error. The probability that $|v - \hat{v}| > \epsilon$ is less than $2e^{-2\epsilon^2 w}$.*

This is a direct application of a Chernoff bound.

Thus, to be sure with probability $1 - \delta$ of having an estimate no further than ϵ away from the true value, $w = O(\frac{1}{\epsilon^2}\log(\frac{1}{\delta}))$ samples are sufficient. Note that this number does not depend on n, m, k, or how the SSAT formula or the policy was created!

Lemma 4 directly gives an approximation algorithm for MAJSAT and can be extended to solve SSAT problems via systematic search, described next.

Given a sample W of size w of assignments to the randomized variables in an SSAT formula, it is straightforward to recursively compute the value of the policy with the largest approximate value. Figure 4 sketches this procedure; as before x_1 refers to the outermost quantified variable.

The running time of sampleevalssat is $w \cdot 2^E$, where E is the number of existential variables in the formula. The next theorem shows that, if the random sample W is large enough, sampleevalssat gives a good

```
sampleevalssat(φ, Q, W) := {
  if Q(x_1) = ∃, return
      max(sampleevalssat(φ⌈_{x_1=0}, Q, W), sampleevalssat(φ⌈_{x_1=1}, Q, W))
  else if Q(x_1) = Я, {
    Let W_0 be the samples in W that assign x_1 = 0
    Let W_1 be the samples in W that assign x_1 = 1
    return |W_0|/|W|sampleevalssat(φ⌈_{x_1=0}, Q, W_0) + |W_1|/|W|sampleevalssat(φ⌈_{x_1=1}, Q, W_1)
  }
}
```

Figure 4. A stochastic sampling algorithm for SSAT problems chooses the best approximate value.

approximation to the true value of the given SSAT formula with high probability.

THEOREM 5. *Let (ϕ, Q, θ) be an instance of* SSAT *and W be a size w random sample of the assignments to the randomized variables in ϕ. Let P be the number of possible policies for the* SSAT *instance. Let $v = val(\phi, Q)$ be the true value of the* SSAT *formula and $\hat{v}$ be the estimate found via* sampleevalssat. *Let $\epsilon > 0$ be a target approximation error. The probability that $|v - \hat{v}| > \epsilon$ is less than $P2e^{-2\epsilon^2 w}$. Thus,*

$$w = O\left(\frac{1}{\epsilon^2}\log\left(\frac{1}{\delta}\right) + \frac{1}{\epsilon^2}\log(P)\right),$$

samples are sufficient to be sure with probability $1 - \delta$ of having an estimate no further than ϵ away from the true value.

Proof: Imagine that we use the sample W to approximately evaluate all P policies. By Lemma 4, the probability that any given estimate is off by more than ϵ is $2e^{-2\epsilon^2 w}$. Therefore, the probability that at least one estimate out of P is off by more than ϵ is less than $P2e^{-2\epsilon^2 w}$. If all P policies have approximate values within ϵ of their true values, then the maximum of the approximate values, that is, $\hat{v}$, cannot differ by more than ϵ from the maximum of the true values, that is, v. ■

The *sample complexity* of the algorithm is the number of samples needed to insure that an accurate estimate of the value of the formula with high probability. Note that for SAT, $P = 2^n$, MAJSAT, $P = 1$, and E-MAJSAT, $P = 2^c \leq 2^n$. Therefore, the sample complexity for sampleevalssat for these SSAT problems is polynomial. However, for Alternating SSAT, $P = \Theta(2^{2^{n/2}})$, so Theorem 5 gives an exponential bound. Appendix B shows that this is a necessary feature of the algorithm by providing a family of SSAT instances that require an exponential sized sample for sampleevalssat to give accurate results with high probability.

2.3.3. *Randomized Local Search*

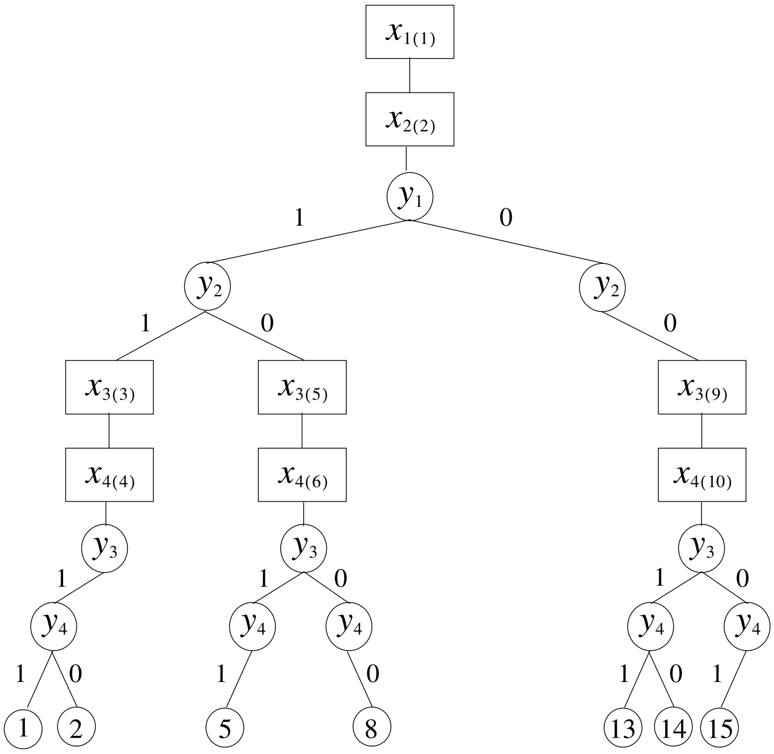
The basic approach described in the previous section is to randomly sample a set of assignments to be used to evaluate policies, then systematically search the space of policies to find one with the best approximate value. Although, for many problems, the number of samples required is not too large, the search for the best policy will be exponential in the number of existential variables in the formula.

Randomized local search can be used to substantially reduce the search time for a good policy, at the cost of finding one that is only locally optimal. An SSAT instance, combined with a sample of assignments to its randomized variables, can be used to construct a *partial policy tree*. The sampled assignments select out a set of leaves from the full policy tree, with the probability of an assignment given by its frequency of selection in the random sample. The top of Figure 5 gives a partial policy tree derived from the sample in the bottom of the figure and the policy tree of Figure 3.

Given a policy, a partial tree can be evaluated as described above (its value is the proportion of the sample corresponding to satisfied leaves). The goal is to search the space of policies to find one with high value.

The randevalssat algorithm in Figure 6 takes SSAT formula ϕ and returns an approximation of its value, along with the policy that approximately produces this value. The algorithm's local search component operates by first converting the partial policy tree to a two-level Boolean formula, called a *treeSAT* formula (function construct_treesat_formula in Figure 6). As illustrated in Figure 3, the effect of a policy on a leaf can be summarized by a Boolean formula, called a *path formula*, for that leaf. Each random sample can potentially produce a different path formula, and the union of all the path formulae produces the treeSAT formula. Thus, on one level the treeSAT formula is a collection of clauses, while on another level it is a collection of path formulae.

A satisfying assignment to the treeSAT formula corresponds to a setting of the decision variables that satisfies the original SSAT formula along all paths in the partial policy tree. In general, it will not be possible to find such a setting, so the challenge here is to find an assignment to the treeSAT formula that maximizes the number of path formulae that are satisfied. This can be accomplished by randomized local search. In the experiments in this paper, this is done by hillclimbing on an objective function that counts both clauses satisfied and path formulae



Original Formula: $(\bar{x}_1 \vee x_3 \vee \bar{y}_3)(\bar{x}_2 \vee \bar{x}_4 \vee y_2)(x_3 \vee y_1 \vee \bar{y}_4)$

Assignments Responsible for Leaves in Partial Decision Tree

Leaf 1:	$y_1 = 1$	$y_2 = 1$	$y_3 = 1$	$y_4 = 1$
Leaf 2:	$y_1 = 1$	$y_2 = 1$	$y_3 = 1$	$y_4 = 0$
Leaf 5:	$y_1 = 1$	$y_2 = 0$	$y_3 = 1$	$y_4 = 1$
Leaf 8:	$y_1 = 1$	$y_2 = 0$	$y_3 = 0$	$y_4 = 0$
Leaf 13:	$y_1 = 0$	$y_2 = 0$	$y_3 = 1$	$y_4 = 1$
Leaf 14:	$y_1 = 0$	$y_2 = 0$	$y_3 = 1$	$y_4 = 0$
Leaf 15:	$y_1 = 0$	$y_2 = 0$	$y_3 = 0$	$y_4 = 1$

Figure 5. Randomized local search can be applied to a partial policy tree, obtained by sampling, to provide an approximate answer for an SSAT instance. Boxes in the figure represent decision nodes and circles random nodes.

satisfied. A satisfied path formula is weighted as heavily as the number of clauses in the original SSAT formula, so that satisfying all the clauses in a path formula contributes more to the treeSAT formula's value than satisfying the same number of clauses scattered over more than one path formula. The function eval_treesat_formula calculates the value of the treeSAT formula using this objective function.

The operation of randevalssat is similar to that of WALKSAT (Kautz & Selman, 1996). The main difference is in the objective function. WALKSAT searches for an assignment that maximizes the number of satisfied clauses. As described above, the two-level structure of the treeSAT formula requires randevalssat to use an objective function that counts both clauses satisfied and path formulae satisfied. (In fact, in the tests conducted, counting satisfied clauses did not seem to be as important as counting satisfied path formulae; randevalssat performed as well with an objective function that counted only satisfied path formulae. Larger problems might exhibit different behavior.)

Essentially, randevalssat randomly sets the values of the treeSAT variables, then hillclimbs on the objective function described above. On each iteration (up to a maximum of IterationLimit iterations), the algorithm randomly chooses an unsatisfied clause. If it is possible to improve the value of the objective function by flipping the truth value of a variable in that clause, it does so. Otherwise, it calculates which variable flip in that clause will lead to the least decrease in the value of the objective function, and accepts that flip with a probability of 0.5. The algorithm returns the treeSAT assignment that produced the highest value of the objective function over all iterations. If the algorithm fails to make progress on a prespecified number of consecutive iterations (FailureLimit) it restarts with a new random setting of the treeSAT variables. A prespecified number of restarts (RestartLimit) is allowed.

Empirical results with randevalssat are discussed in Section 3.4.

2.3.4. *Related Work*

Many other researchers have explored approximations for probability-based problems. Roth (1996) studied the complexity of inference in belief networks. He showed that computing the probability of a query node (akin to evaluating a MAJSAT formula) is #P-complete. In addition, approximating the probability of a query node to within a multiplicative factor of its true value is just as hard as computing the exact value. In contrast, Lemma 4 shows that the complexity of getting within an *additive* factor for MAJSAT is polynomial.

Kearns, Mansour, & Ng (1999) show how approximately optimal actions can be chosen with high probability in infinite-horizon discounted

```
randevalssat(φ) := {
  ρ := construct_treesat_formula(φ);
  currentpolicy := generate a random truth assignment for ρ;
  P := eval_treesat_formula(ρ); bestP := P;
  bestpolicy := currentpolicy;
  for (r = 1 to RestartLimit) {
    numfailures := 0;
    for (i = 1 to IterationLimit) {
      if (all clauses in ρ are satisfied)
        return(P, currentpolicy);
      randomly choose an unsatisfied clause c in ρ;
      bestflipvar := 0; bestnewP := 0;
      for (each literal l in c) {
        flip the truth assignment of the variable v corresponding to l;
        newP = eval_treesat_formula(ρ);
        if (newP > P) {
          numfailures = 0;
          P = newP;
          if (newP > bestP) {
            bestP = newP; bestpolicy = currentpolicy;
          }
          break out of literal loop and start next iteration;
        }
        else if (newP > bestnewP) {
          bestflipvar = v; bestnewP = newP;
        }
        restore the truth assignment of the variable v corresponding to l;
      } /* literal loop */
      with probability 0.5 {
        flip the truth assignment of variable bestflipvar;
        P = bestnewP;
      }
      numfailures := numfailures + 1;
      if (numfailures > FailureLimit){
        currentpolicy := generate a random truth assignment for ρ;
        P := eval_treesat_formula(ρ);
        break out of iteration loop and start next restart;
    } /* iteration loop */
  } /* restart loop */
  return(bestP, bestpolicy);
}
```

Figure 6. A randomized algorithm for SSAT problems uses hillclimbing on a SAT formula derived from a partial policy tree created by sampling.

Markov decision processes. Their strategy is to show that it is sufficient to consider the first H actions in the sequence, for an appropriate choice of H. The space of all H-step action sequences is then evaluated using random sampling. The sampleevalssat algorithm in Section 2.3.2 was directly inspired by this work.

In summary, the stochastic sampling algorithm randevalssat appears to be a promising approximation method for SSAT problems. Its primary strength is the use of sampling to convert the problem to a lower complexity problem and its use of randomized local search to solve that problem efficiently. A feature of this process is that it does not necessarily completely discard the probabilities of the original problem (as would, say, a conversion that merely rounded off the probabilities of the random variables to 0 and 1 and set their truth values accordingly). It would, for example, be possible to modify the algorithm such that the probabilities of the random variables are used to direct the construction of the partial policy tree. A more substantial modification would be to iteratively build the partial policy tree by using the solution of the partial policy tree in a given iteration to construct a better partial policy tree (and solution) in the next iteration.

This algorithm has some weaknesses. First, unlike WALKSAT, randevalssat does not return an answer whose correctness is "easy" to certify; this is to be expected, given the complexity of SSAT problems. Second, there are SSAT problems for which the algorithm needs provably large samples (see Section B for an example of such a problem). Third, randevalssat is subject to the same pitfalls as other randomized local search algorithms; in particular, it may become stuck in local optima. Allowing the algorithm to restart after a period of no progress helps minimize, but does not erase, this problem. Finally, the memory requirements of the algorithm can be prohibitively large. This weakness can be partially overcome by more memory-efficient implementations, but problems that require a large number of samples to produce an accurate answer will inherently generate large treeSAT formulae.

3. Results

This section describes some empirical explorations of solving SSAT problems of various types using the algorithms presented in Section 2.

3.1. Varying Problem Type

Throughout the experiments in this section, the DPLL-based SSAT algorithm from Section 2.1 (variables split in strict quantifier order) is

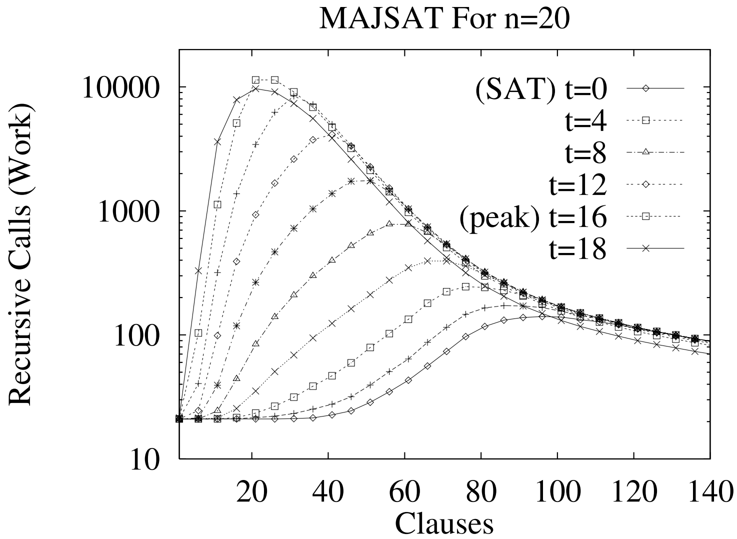


Figure 7. For random $n = 20$ variable formulae ($k = 3$ literals per clause), the difficulty of solving random MAJSAT instances via DPLL varies with both the number of clauses to variables and the satisfaction threshold t. The plot includes curves for thresholds $t = 0$ through $t = 18$ by 2s (not all curves are labeled). Peak difficulty occurs at $t = 16$ at roughly 20 clauses.

used. A set of 1,000 formulae with $n = 20$ variables, $k = 3$ literals per clause, and 141 clauses were randomly generated using `makewff`. All thresholds were expressed as $\theta = 1/2^{n-t}$ for integer t in the range 0 to n; this defines t so that 2^t is the number of satisfying assignments required. Formulae with m clauses were created by using the first m clauses from each of the 1,000 formulae.

3.1.1. MAJSAT *vs.* SAT

The first experiment compares and contrasts MAJSAT with SAT. Figure 7 illustrates the average work required to solve random MAJSAT instances, varying the number of clauses and values of the threshold parameter t.

The line on the plot labeled $t = 0$ is the curve for SAT; this problem is asking whether the probability of satisfaction is at least $1/2^n$, which is reached as long as there is even a single satisfying assignment. The classic easy-hardest-hard pattern is visible. In fact, nearly all the threshold values produce the same basic shape.

Note that the peak difficulty over all thresholds occurs at $t = 16$; it is much higher than the peak difficulty for SAT and occurs at a much lower setting of the number of clauses m. High values of the threshold are difficult because the threshold pruning rules for randomized quantifiers

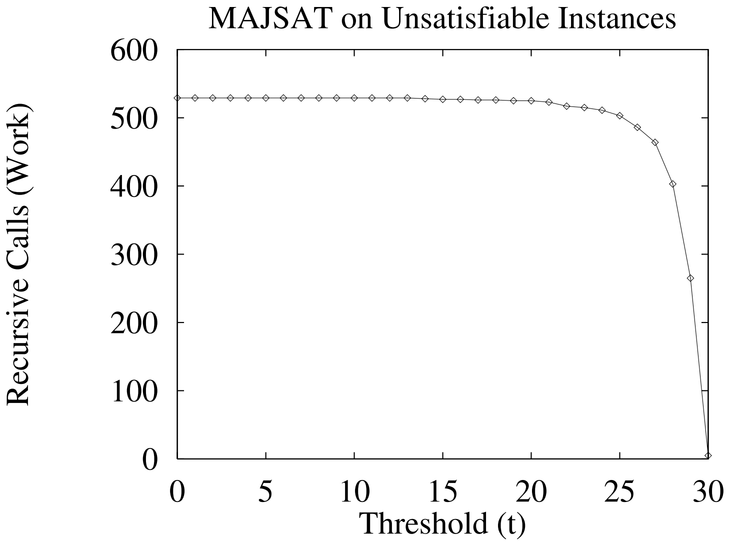


Figure 8. For random unsatisfiable $n = 30$ variable formulae ($m = 141$ clauses, $k = 3$ literals per clause), the difficulty of solving random MAJSAT instances via DPLL varies with the satisfaction threshold t. When the threshold is high, pruning becomes more effective, allowing instances to be solved more quickly.

rarely apply—it is almost always necessary to check both branches to determine whether the probability threshold can be met.

Note that when no pruning rules are used, MAJSAT and SAT are equally difficult for the DPLL algorithm; it must create the entire DPLL tree for both, and has the freedom to split in any order. This implies that, for unsatisfiable formulae, MAJSAT is actually *easier* to decide than SAT, since threshold pruning is only possible in the case of MAJSAT. Figure 8 shows that the average difficulty for solving MAJSAT problems based on the 689 random $n = 30$ formulae that are unsatisfiable at $m = 141$ clauses does decrease with increasing satisfiability threshold.

3.1.2. E-MAJSAT

On average, however, MAJSAT instances were substantially more difficult to solve than SAT instances. That MAJSAT is more difficult to solve on average than SAT is not surprising, since it belongs to a higher complexity class. A more interesting pattern occurs in E-MAJSAT formulae. As mentioned in Section 1.3, the E-MAJSAT parameter c interpolates between MAJSAT ($c = 0$) and SAT ($c = n$). In terms of complexity theory, however, intermediate values of c place E-MAJSAT in a more difficult complexity class than either endpoint. This suggests analyzing the peak difficulty of E-MAJSAT as a function of c.

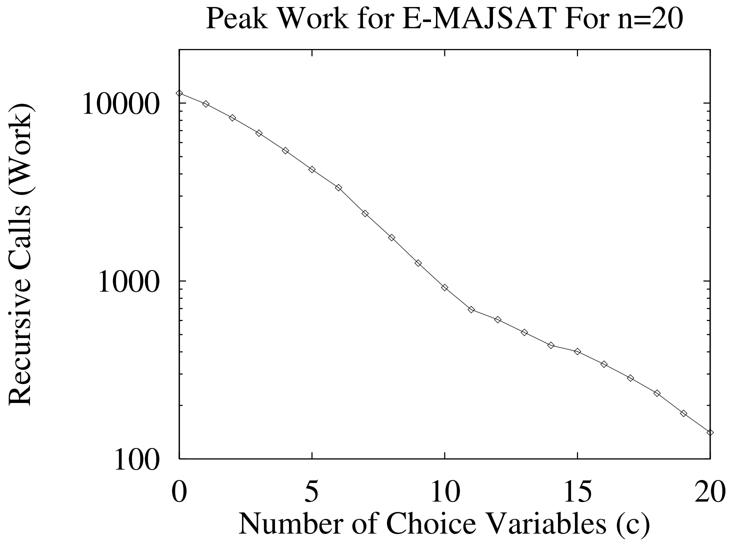


Figure 9. The difficulty of solving random E-MAJSAT instances via DPLL decreases with increasing number of choice variables c. The plot is for random formulae with $k = 3$ literals per clause.

The following experiment was carried out. For each value of c from 0 to $n = 20$, the formulae were solved for each clause size m from 1 to 141 by 5s and threshold parameter t from 0 to 19. Work was averaged separately for each combination of settings, and the value for t and m that resulted in the maximum average work was selected for each value of c. Figure 9 summarizes the results.

The results of these experiments are somewhat counterintuitive. Instead of the peak difficulty being obtained for $c = n/2$ as might be assumed from complexity theory, the difficulty is *logarithmically* related to the number of choice variables. It appears that the main effect is that each randomized quantifier that is changed to an existential quantifier results in a constant fraction of savings of work. This result calls for more careful study—perhaps by varying the instance size.

3.1.3. SSAT *Subtypes*

Figure 10 plots the work vs. number of clauses for each of four types of SSAT problems described earlier. For each problem class, the most difficult (on average) setting of the threshold parameter observed was used. There are several things to note here. One is that all these randomized satisfiability problems exhibit the same easy-hardest-hard pattern described for SAT. However, the number of clauses corresponding to the peak work differs for each problem. Also, the higher the peak, the smaller the number of clauses at the peak.

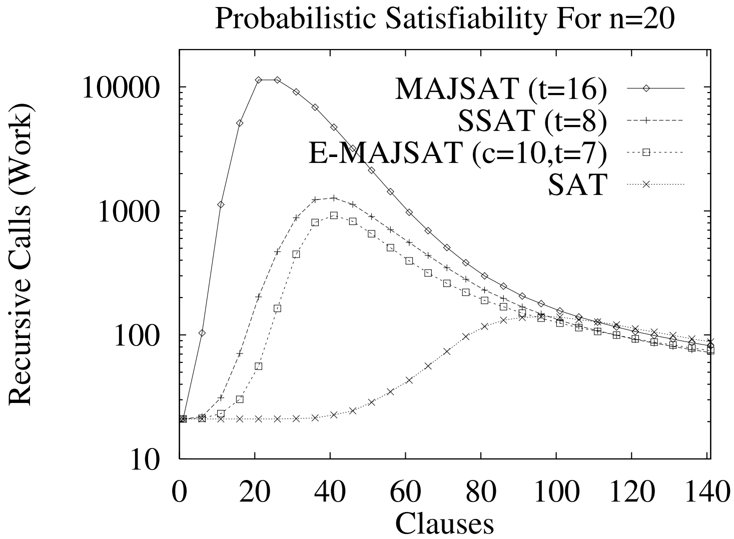


Figure 10. The peak difficulty of solving different SSAT instances via DPLL varies with the structure of the operators and the number of clauses. The plot is for random formulae with $k = 3$ literals per clause. MAJSAT problems appear to be the hardest, on average.

Another observation is that the relative peak difficulty of different problems does not match what might be predicted by complexity theory. In particular, as $\mathrm{NP} \subseteq \mathrm{PP} \subseteq \mathrm{NP}^{\mathrm{PP}} \subseteq \mathrm{PSPACE}$, one might expect SAT < MAJSAT < E-MAJSAT < Alternating SSAT in terms of peak work (here, "<" is being used as shorthand for "requiring less work than"). In fact, the experiments come out with SAT < E-MAJSAT < Alternating SSAT < MAJSAT. That is, MAJSAT comes out as the hardest instead of the second easiest. This pattern can be observed with a wide range of values of n, with the MAJSAT instances getting harder much faster than the other problems with increasing n.

The facts that E-MAJSAT is easier than MAJSAT and that SAT is easier than Alternating SSAT probably stem from the fact that randomized quantifiers are harder to prune than are existential quantifiers. Recall from the DPLL algorithm of Section 2.1 that existential quantifiers can be simplified by purification whereas randomized quantifiers cannot. In addition, the threshold pruning rules for randomized quantifiers are weaker and will apply less often. Thus, the presence of randomized quantifiers can make an SSAT problem more challenging. Supporting this observation is the fact that Alternating SSAT and E-MAJSAT ($c = n/2$) both consist of half existential and half randomized quantifiers, and have very similar curves in Figure 10.

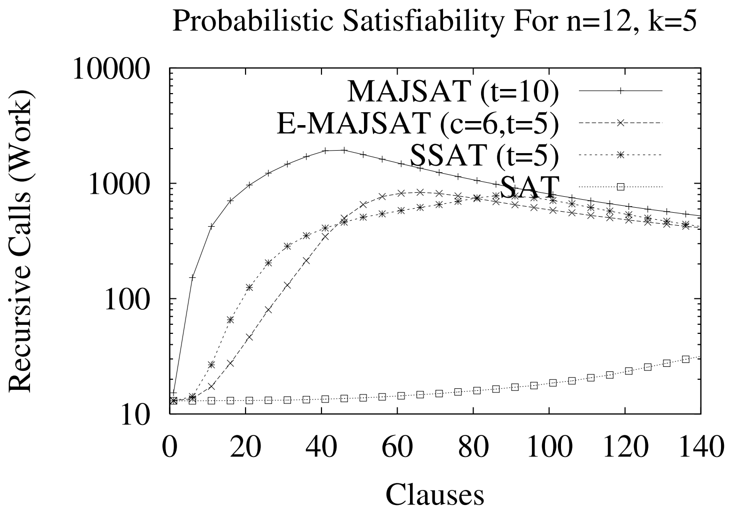


Figure 11. The peak difficulty of solving different SSAT instances via DPLL varies with the structure of the operators and the number of clauses. With more than $k = 3$ literals per clause, the difference in difficulty of the problem types lessens.

A similar observation concerning problem difficulty and quantifier type was made by Gent & Walsh (1998); they argue that random QBF formulae are trivially easy because of the fact that a single clause consisting of all (or nearly all) universal variables can be solved with no backtracking at all. Thus, in spite of the fact that QBF is PSPACE-complete, QBF $<$ SAT for random CNF formulae. This effect was also observed by Cadoli, Giovanardi, & Schaerf (1997), who found that increasing the number of universal quantifiers at the expense of existential quantifiers makes problems easier to solve.

Figure 11 shows a similar pattern when the number of literals per clause $k = 5$, although the curves for MAJSAT and Alternating SSAT have grown closer together.

3.1.4. *Threshold*

With SAT, the peak hardness of determining whether an instance is positive appears to correspond to the point at which about half of all random instances are positive; informal tests support the same conclusion for MAJSAT. Figure 12 examines the threshold, showing results for MAJSAT for $k = 3$, $n = 30$, varying the number of clauses m and threshold t. As the satisfaction threshold increases, few clauses are needed to prevent a random instance from being positive. There is a strong linear relationship between m and t visible in the graph that begs explanation; the observed relationship between n, m, and t at the

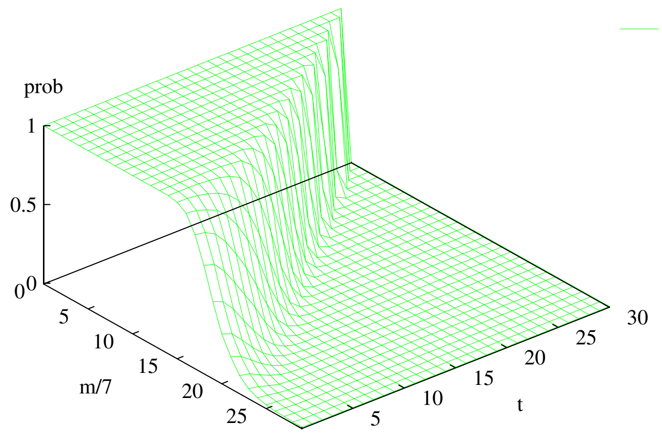


Figure 12. The probability that a random instance of MAJSAT ($n = 30$, $k = 3$) is positive varies with both the satisfaction threshold t and the number of clauses m. Peak difficulty for a fixed t occurs around the number of clauses m for which half of the instances are positive.

threshold is $m \approx 4.2(n-t)$. This relation makes sense for $t = 0$, since it says that the 0–1 threshold for SAT occurs at $m \approx 4.2n$. It also makes sense for $t = n$, since only a single clause is needed to prevent there from being $2^t = 2^n$ satisfying assignments. Equation 1 gives a lower bound on this threshold with this form, but no upper bound of this type has been proven.

3.2. Varying Problem Size

To see how peak problem difficulty scales with n, the following experiment was carried out. For each of $n = 5, 10, 15, 20, 25, 30$, a thousand $k = 3$-CNF MAJSAT formulae were generated. For each n, the DPLL solver solved all 1,000 formulae for all values of t from 0 to n and all values of m from 1 to about $7n$ (at intervals of approximately $n/4$ clauses). This was used to identify the hardest settings of t and m. At this point, all values of m and t in the vicinity of the maximum were run to get a more accurate answer. Finally, a peak combination of values of m and t was identified for each n.

The results were quite clean (the lines are sufficiently straight that they are not shown). For all values of n, the peak work was achieved with $t \approx 0.80n$ and $m \approx 1.14n$. The peak work for these settings follows a clear exponential growth curve of 3.96×1.49^n. This is interesting, because, even with a difficult problem like MAJSAT, this simple version of DPLL achieves a running time that is asymptotically faster than that of full enumeration ($O(2^n)$).

3.3. An Evaluation of Heuristics for evalssat

In all but the most trivial SAT problems, the DPLL algorithm will exhaust opportunities to apply unit resolution and purification before a satisfying assignment has been discovered. When this occurs, the algorithm must select an unassigned variable x from the current simplified formula to split on, *i.e.* check for satisfying assignments both when x is assigned `True` and when x is assigned `False`. The order in which variables are selected is critical. For example, suppose a SAT problem is presented to DPLL with n variables and, for the sake of this example, that no pruning is done. Suppose further that if the variable x_a is set to `True` the formula becomes unsatisfiable. If x_a is the last variable the algorithm chooses to split on, it will potentially generate 2^{n-1} assignments, discovering as many times that setting x_a to `True` makes the formula unsatisfiable. If, however, it selects x_a as the first variable to split on, it will discover that setting that variable to `True` leads to unsatisfiability *before* generating and checking those 2^{n-1} assignments. Unit resolution, as described in Section 2.1.1, handles this extreme case; other ideas are needed to address this more generally.

Considerations such as this have prompted a great deal of research into efficient splitting heuristics for SAT (Hooker & Vinay, 1994; Bayardo & Schrag, 1997; Gallo & Urbani, 1989; Harche, Hooker, & Thompson, 1994; Jeroslow & Wang, 1990; Li & Anbulagan, 1997; Crawford & Auton, 1996; Freeman, 1995). An appropriate splitting heuristic can reduce the running time of the DPLL algorithm by several orders of magnitude. But, do these splitting heuristics improve efficiency in SSAT problems?

Relative to its counterpart in SAT problems, splitting in SSAT is restricted to select among a specific class of variables. A *block* of variables is a maximal set of variables that appear in the formula with the same quantifier and are adjacent in the quantifier ordering. To ensure that the value of the formula is computed correctly, splitting heuristics for SSAT must choose a variable from the first (leftmost or outermost) block, as described in Section 1.2.1.

The reasoning behind this restriction is because adjacent quantifiers commute if they are of the same type, but not if they are of different types. Returning to the example from Section 1.2, consider the formula

$$\text{ꓤ}x, \text{ꓤ}z, \exists y, E[(x \vee y)(y \vee \overline{z})(\overline{x} \vee \overline{y})] \geq 3/5.$$

The value of the formula is 3/4, and this can be determined by splitting either on x or z first (both are in the first block). On the other hand, if y is used as the variable in the first split, the value is calculated incorrectly as 1/2.

A hypothesis is that splitting heuristics would have the greatest impact when there are large blocks of interchangeable variables (the extremes being SAT and MAJSAT, both of which consist of a single block of size n), and would not be as useful when there was a great deal of quantifier alternation (since, barring unit clauses and pure variables, DPLL must split on all variables in the first block before splitting on a variable from another block). In the case where blocks are single variables, the splitting heuristic can be expected to have no impact on performance.

To test this hypothesis, tests were conducted using six splitting heuristics (where, in all cases, it is understood that the choice is made from the outermost block of variables):

- RAND chooses a variable randomly,
- SATS_MOST finds the literal that satisfies the most clauses in the current formula and chooses that variable,
- 2S_JW (2-sided Jeroslow-Wang, described by Hooker & Vinay (1994)) chooses a variable whose literals appear in a large number of short clauses,
- POS_2S_JW (positive, 2-sided Jeroslow-Wang, described by Hooker & Vinay (1994)) is 2S_JW applied to the subset of variables that appear in at least one clause containing all positive literals,
- MOMS chooses the variable that has Maximum Occurrences in clauses of Minimum Size, and
- RELSAT, a splitting heuristic used by Bayardo & Schrag (1997) in some of their work on satisfiability algorithms, is similar to heuristics used by the successful POSIT (Freeman, 1995) and TABLEAU (Crawford & Auton, 1996) solvers, preferring variables that will lead to a maximal number of unit resolutions.

With the exception of RAND, these heuristics fall into two broad categories. SATS_MOST, 2S_JW, and POS_2S_JW all try to select a variable that is likely to lead to a satisfying assignment. MOMS and RELSAT, however, try to select a variable that is likely to produce unit clauses, thereby simplifying the formula and discovering unsatisfying assignments more quickly. The baseline for comparison is the splitting rule in Figure 2, which splits on the next variable in the initially specified quantifier ordering.

To test the performance of these heuristics on problems with varying block sizes, eight sets of 100 random formulae were generated with the following characteristics:

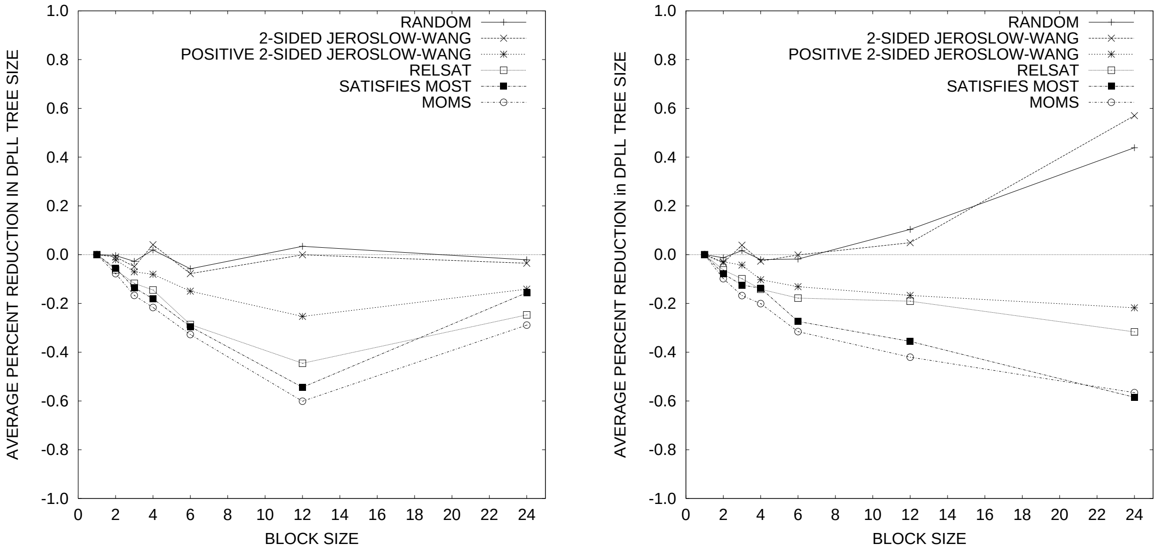
24 variables, 24 clauses, 3 literals per clause
24 variables, 48 clauses, 3 literals per clause
24 variables, 72 clauses, 3 literals per clause
24 variables, 96 clauses, 3 literals per clause
32 variables, 32 clauses, 3 literals per clause
32 variables, 64 clauses, 3 literals per clause
32 variables, 96 clauses, 3 literals per clause
32 variables, 128 clauses, 3 literals per clause.

These sets were generated according to $\mathcal{F}_m^{k,n}$ (Section 1.4) using `makewff`. In order to ensure that the formulae generated contained exactly n variables, however, `makewff` was modified to repeatedly generate a formula with the desired characteristics until one was generated that contained all n variables. For each block size in a specified range of block sizes (1, 2, 3, 4, 6, 12, and 24 for problems with 24 variables and 1, 2, 4, 8, 16, and 32 for problems with 32 variables) two problem types were generated: a type E problem, which has an existentially quantified outermost block, and an R problem, which has a randomly quantified outermost block. The block sizes chosen were all those that generate an equal number of existential and randomized quantifiers, plus SAT and E-MAJSAT (block size 24 or 32). In this manner, each set of 100 24-variable problems generated 1400 instances, and each set of 100 32-variable problems generated 1200 instances.

The `evalssat` algorithm was implemented in C and each splitting heuristic was tested on all 10,400 problem instances generated as described above. These tests were conducted on a Sun Ultra-1 running SunOS-5.7 with 128 MB of RAM. Performance was measured by counting the change in the number of recursive calls to the central function in the algorithm. This function first tests for satisfaction of the formula; if the current partial assignment is insufficient to establish satisfaction or unsatisfaction, the function extends the partial assignment. In the most general case, the function selects an unassigned variable, first setting the variable to `True` and calling itself recursively, and then setting the variable to `False` and calling itself recursively. Whenever possible, as in the case of a variable in a unit clause, one of these recursive calls is eliminated. Measuring the number of calls to this function is equivalent to measuring the size of the DPLL tree generated by the algorithm and is thus a more direct measure of the efficiency of the heuristic than measuring, for example, CPU time. For perspective, however, the running time, measured in CPU seconds, for a non-MAJSAT 24-variable, 48-clause problem was typically less than a second. Running times for MAJSAT problems varied from 1 to 5 seconds. A connection between these two ways of measuring performance is provided by the fact that for the results reported below (problems with 24 variables, 48 clauses, 3 literals per clause) the algorithm performs approximately 20,000 recursive calls per cpu second.

Although the results of the tests (Figure 13) generally supported the hypothesis stated above, there are some peculiarities that warrant further study. (Note that the results for all eight sets of problems were similar; we show the results only for 24-variable, 48-clause problems.) On both type E and type R problems, four of the heuristics tested (POS_2S_JW, RELSAT, SATS_MOST, and MOMS) provided some improvement in performance (reduction in the size of the DPLL tree generated). For type E problems, this improvement was positively correlated with block size (see Figure 13(b)). For type R problems the improvement was positively correlated with block size up to a block size of 12; all four of these heuristics show less improvement for a block size of 24 (MAJSAT problems, in which the formula cannot be simplified by purification) than they do for a block size of 12 (see Figure 13(a)). SATS_MOST and MOMS were particularly effective, reducing the number of recursive calls in DPLL by approximately half (over both problem types) for a block size of 12. RAND performed as expected on type R problems, neither improving nor degrading performance. 2S_JW, a heuristic which might have demonstrated improved performance, was similarly neutral. And neither of these two heuristics performed as expected on type E problems; both heuristics degraded performance, and this degradation was positively correlated with block size. The fact that 2S_JW performed no better than RAND agrees with results reported by Hooker & Vinay (1994); on a set of hard, random SAT instances, they reported that the DPLL trees constructed using the 2S_JW heuristic were significantly larger than those constructed using a random heuristic.

For the heuristics that improved performance, the block size did not have to be very large to produce significant reductions in the size of the DPLL tree generated. For the four successful heuristics, tree size was reduced as soon as blocks were larger than a single variable and, in some cases, the reduction became significant even for relatively small block sizes. RELSAT, SATS_MOST, and MOMS, for example, provided average reductions in tree size of approximately 23%, 28%, and 32%, respectively, for a block size of six.



(a) first block R (b) first block E

Figure 13. The heuristics POS_2S_JW, RELSAT, SATS_MOST, and MOMS reduce the size of the DPLL tree generated relative to splitting in strict quantifier order when the formula block size is greater than 1. For the most part, larger block sizes yield more improvement.

The results suggest that a successful SAT heuristic is likely to be a successful heuristic in MAJSAT and SSAT problems, but block size is a limiting factor. In the case where blocks are single variables, the splitting heuristic has no impact on performance. It is important to note, however, that the tests were conducted on small, random problems. For problems with more structure, such as SSAT formulae that encode planning problems, these results may not hold. In addition, there may be other good splitting heuristics that take advantage of this structure. For example, Majercik & Littman (1998) report improved performance on such SSAT problems using a *time-ordered* splitting heuristic in which variables with a lower time index are chosen first. The success of SATS_MOST, which is computationally much simpler than RELSAT or MOMS, was somewhat surprising in light of some evidence that satisfying heuristics, like SATS_MOST, tend not to perform as well as simplifying heuristics, like RELSAT and MOMS (Hooker & Vinay, 1994). It is possible that performance disparities might appear in these three splitting heuristics as the problem size increases beyond that of the problems in these tests. Finally, it is possible that splitting heuristics unique to Extended SSAT problems could be developed. Such heuristics might use the probabilities associated with the random variables to select variables in a way that provides potentially greater simplification of the problem.

3.4. Experiments with the Stochastic Sampling Algorithms

The performance of randevalssat for computing the value of SSat instances was tested on 27 sets of 100 random formulae. The characteristics of these 27 sets were generated by taking the cross product of the following sets: $\{n = 8, 12, 16\}$, $\{m = n, 2n, 3n\}$, and $\{k = 3, 4, 5\}$. As was the case for the evaluation of heuristics in Section 3.3, these sets were generated according to $\mathcal{F}_m^{k,n}$ (Section 1.4) using a modified version of `makewff` that guarantees a formula with exactly n variables. For each problem a Sat instance (all existential variables), a Majsat instance (all randomized variables), and all possible instances which contain an equal number of existential and randomized variables in alternating blocks of the same size were constructed. This generated 8 instances for each 8-variable problem (block sizes 1, 2, 4, and 8, each with an instance in which the initial block is existentially quantified and an instance in which the initial block is randomly quantified), 10 instances for each 12-variable problem (block sizes 1, 2, 3, 6, and 12 with instances differing by initial quantifier), and 10 instances for 16-variable problems (block sizes 1, 2, 4, 8, and 16 with instances differing by initial quantifier). Thus, each set of 100 8-variable problems generated 800 distinct problems, and all other sets generated 1000 distinct problems, for a total of 25,200 problems. For each of these problems, 10 different partial policy trees were created by sampling random variable assignments and constructing the tree paths specified by the samples. The number of assignments sampled w ranged from 5 to 4086 on an approximate logarithmic scale; the larger the number of samples, the greater the similarity of the partial tree to the full tree.

Both sampleevalssat and randevalssat were implemented in C. Each of the partial policy trees generated for the 25,200 problem instances described above was solved using randevalssat. For comparison, those problem instances with 12 variables, 24 clauses, and 3 literals per clause were also solved using sampleevalssat. For randevalssat, the number of restarts and the number of iterations were both specified as a multiple of the number of variables in the CNF formula constructed from the tree, so increasing the number of samples tended to increase the number of restarts and iterations. These tests were conducted on a Sun Ultra-1 running SunOS-5.7 with 128 MB of RAM. Performance was measured by comparing the estimate of (or, in the case of sampleevalssat, the exact calculation of) the value of the partial policy tree to the exact value of the full policy tree (and, hence, the formula). This difference is referred to below as the error in the value estimate. Note that since the running time is proportional to the number of restarts and iterations allowed, and since these limits are increased for larger problems, running time is not a very informative measure of performance. For perspective, however, running times, measured in CPU seconds, for 12-variable, 24-clause problems, with 4086 assignment samples, varied from 1 to 5 seconds.

Figure 14 shows results for the sampleevalssat algorithm on problems with 12 variables, 24 clauses, and 3 literals per clause; the graph shows mean squared error in the value estimate as a function of the number of sampled assignments for all problem types. Since the algorithm is performing perfect optimization on the partial tree, any error is due entirely to the incompleteness of the partial tree due to sampling. As expected, the mean and variance of the squared error approach zero as the number of assignment samples increases, and the partial tree constructed approaches the full policy tree.

Figure 15 shows the same graphs for the results of the randevalssat algorithm. Here, the mean and variance of the squared error decrease as the number of assignment samples increases. But, even in the limiting case, where the number of assignments sampled is sufficient to construct the full policy tree with high probability, randevalssat is not always capable of finding the optimal setting of the decision variables. (Note that in some cases, however, randevalssat can generate almost as accurate an answer as sampleevalssat in far less time. For one problem with 36 variables, 72 clauses, and 3 literals per clause, randevalssat was able to return an answer with squared error of approximately 10^{-4} in 442 cpu seconds, while sampleevalssat required over 4 times as long to compute an answer with squared error of approximately 10^{-5}.)

It should also be noted that the randevalssat algorithm can exhibit anomalous behavior as the number of assignment samples increases. A set of 100 random problems with 12 variables, 24 clauses, and 3 literals per clause was generated, but with a block size of 4 and an initial block of existential variables. This set was not included in the results reported above, since this quantifier ordering results in problem instances with an unequal number of existential and random variables. The mean squared error for this set of problems decreases to approximately 0.008 as the number of sampled assignments increases to 55. As the number of sampled assignments increases further, however, the mean squared error also increases, appearing to asymptote at approximately 0.016. Since this behavior does not appear when the same problem instances are solved using sampleevalssat, the anomaly appears to be due to the imperfect optimization technique employed by randevalssat becoming trapped in a local optimum. Initially, generating a policy tree with more

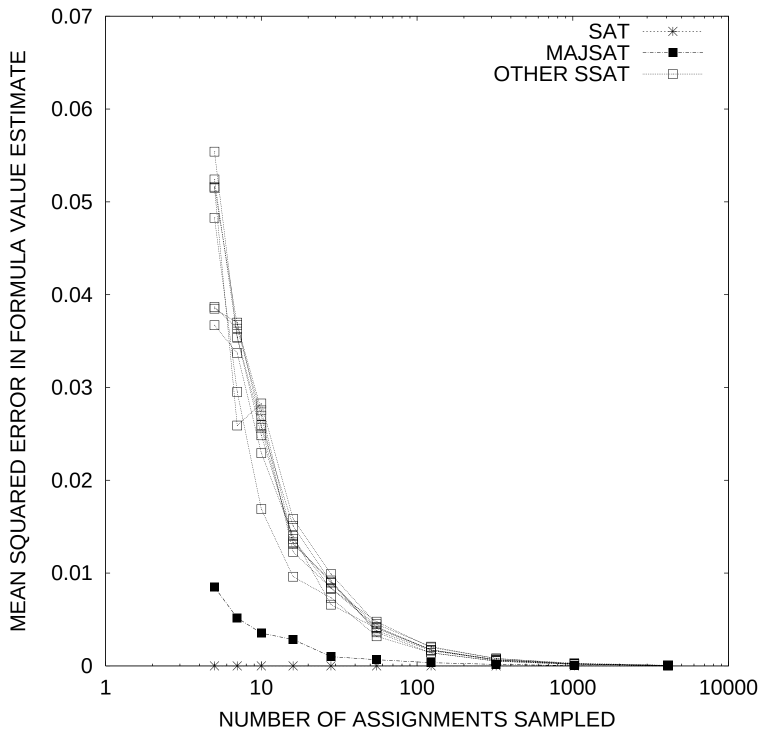


Figure 14. The partial policy tree produced by random sampling can be solved exactly using a DPLL-style approach. Approximation error decreases as sample size increases.

branches allows the algorithm to return a more accurate estimate of the value of the formula. Beyond a certain point, however, this advantage seems to be outweighed by the increasing difficulty of the optimization problem; the algorithm is increasingly likely to become stuck in local maxima and so less likely to find the global maximum within the permitted number of restarts. Future work will examine striking the proper balance between sampling completeness and optimization difficulty.

Future work will also consider more sophisticated randomized local search procedures. For instance, the algorithm could be modified to select the decision node or the variable to flip in that decision node according to some set of criteria. Another variation would use the entire local search algorithm above as a subroutine in an algorithm that periodically restarts with a new set of samples. The approach could also be refined using simulated annealing.

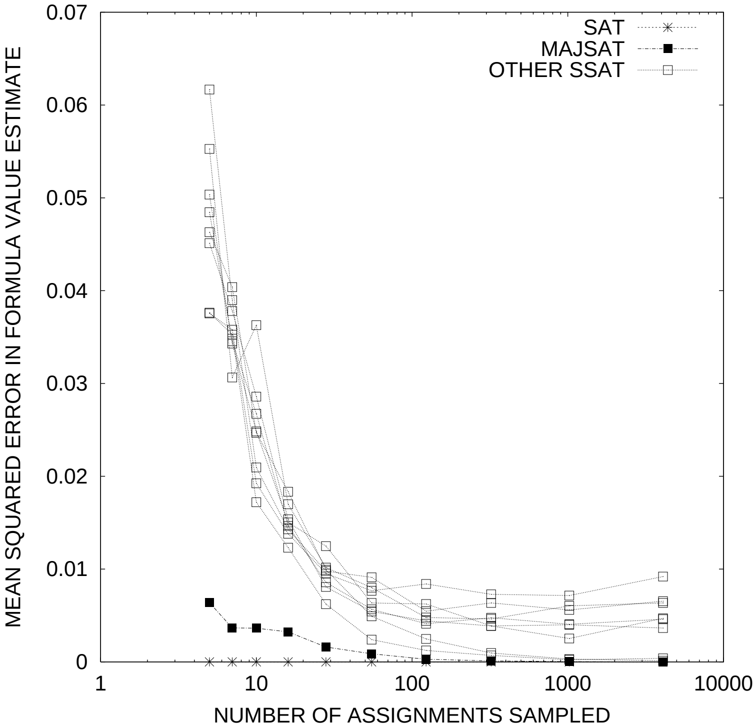


Figure 15. As the number of sampled assignments increases, the accuracy of the randomized local search algorithm increases. Because of local optima in the search space, increasing the number of sampled assignments does not drive the error to zero.

4. Conclusion and Future Work

This paper described a stochastic satisfiability framework, relating it to existing problems in reasoning and planning under uncertainty. It described several algorithms, inspired by work in deterministic satisfiability: a DPLL-based algorithm, a "universal" search algorithm, and a stochastic sampling algorithm. The paper also explored the behavior of randomly generated satisfiability problems.

This paper takes a first stab at developing practical algorithms for SSAT, with particular emphasis on understanding which algorithms, intuitions and ideas developed for the satisfiability problem extend to more general satisfiability-type problems. In spite of the gap in computational complexity between the general form of stochastic satisfiability (SSAT, PSPACE-complete) and deterministic satisfiability (SAT, NP-complete), this work shows that many insights obtained from studying SAT carry over to the more general probabilistic setting. Many more questions remain unanswered; some directions for future work are sketched next.

A primary motivation for studying Extended SSAT problems is to gain a better understanding of algorithms for decision making under uncertainty. These problems are modeled by Extended SSAT in that existential quantifiers act like "maximizing" decisions, universal quantifiers act like a "minimizing" decisions, and randomized quantifiers act like an oblivious random influence. Condon (1992) studied a simple stochastic game model with precisely these three types of nodes. There are two fundamental differences between simple stochastic games and stochastic satisfiability. Stochastic games have no predefined "horizon"; games can continue indefinitely. Extended SSAT problems, in contrast, model domains with a fixed number of decisions or events (one for each quantifier), at which point the "game" ends. On the other hand, Extended SSAT problems model very complex decisions, in that a value for an existential quantifier can be conditioned on all the preceding randomized quantifiers—possibly an exponential-sized set of conditions. This allows Extended SSAT problems to model compactly represented AI planning domains, an insight that has led to an implemented state-of-the-art planning algorithm (Majercik & Littman, 1999). Representing a planning problem via a simple stochastic game leads to a state-space explosion, rendering it an inefficient formalism for complex domains. An important direction for future work involves combining the core propositional representation of Extended SSAT with the unbounded horizon of simple stochastic games.

The study of randomized satisfiability algorithms has helped spur the development of algorithms and applications of deterministic satisfiability. While this paper has proposed an initial stochastic sampling algorithm for SSAT, more experiments and more algorithmic insights are needed. One area in need of more research is how the approximation algorithm can be applied to solve planning problems. In particular, the algorithm returns a partial assignment strategy, or policy, that specifies how each existential variable should be set given the settings of the variables that precede it in the quantifier ordering, but this only happens for those situations represented by paths in the random sample. The algorithm implicitly assumes that performance on missing branches is the same as the average performance over all paths in the partial tree, but doesn't actually find a plan that achieves this performance. A topic for future work is extending the stochastic sampling algorithm to return successful plans, perhaps in an iterative approach that integrates planning and execution.

In addition, it is not clear how to apply the proposed algorithm to formulae with intermixed existential, randomized, *and* universal quantifiers. Such problems arise, for example, in applications in which two competitive agents interact with a stochastic environment, and a randomized algorithm for this broader class of SSAT problems would be useful.

The proposed universal search algorithm is a promising approach to solving Extended SSAT problems exactly and efficiently. Future work will be to implement and test the algorithm, comparing it to the splitting heuristics examined in Section 3.3. Although they are not well understood theoretically, the threshold pruning rules described in Section 2.1.2 could be helpful in a practical implementation of the universal search algorithm.

There has been a recent flurry of activity in analyzing common Davis-Putnam algorithms, and proving that these algorithms are guaranteed to solve any k-SAT instance in subexponential-time (Monien & Speckenmeyer, 1985; Paturi, Pudlak, & Zane, 1997; Paturi *et al.*, 1998). (For example, a very simple randomized DPLL-type algorithm solves 3-SAT in worst-case time $2^{(2/3)n}$.) It would be interesting to show that these analyses carry over to the SSAT setting.

In addition to Resolution-based methods, there are several alternative methods for solving satisfiability instances. In particular, algebraic approaches based on discrete versions of the Grobner basis algorithm are looking promising for solving satisfiability problems (Clegg, Edmonds, & Impagliazzo, 1996). Future work will examine to what extent these methods carry over to the SSAT domain.

A number of recent studies have shown that the satisfiability framework is useful for examining a variety of search problems. An exciting element of the present work is that it illustrates how the satisfiability approach applies outside of SAT's complexity class, greatly expanding the realm of problems that can be attacked.

Acknowledgments

Thanks to Pankaj Agarwal, Julien Basch, Ian Gent, Judy Goldsmith, Moises Goldszmidt, Henry Kautz, Don Loveland, Matt Peters, John Reif, Bart Selman, and Toby Walsh for feedback and suggestions. Avrim Blum suggested the use of sampling and Chernoff bounds, and Noam Shazeer provided the insight behind the counter example in Appendix B. We would also like to thank Steve Cook, Mike Molloy, and especially Dimitris Achlioptas for illuminating conversations on the 0–1 threshold properties of SAT and MAJSAT formulae.

Appendix

A. Correctness of evalssat

This section proves that the SSAT algorithm described in Section 2.1 produces the correct value of the given SSAT instance (ϕ, Q, θ).

LEMMA 6. *For all n, for all formulae ϕ with at most n variables, for all thresholds θ_l, θ_h, such that $\theta_l \leq \theta_h$, $\mathsf{evalssat}(\phi, Q, \theta_l, \theta_h)$ returns a value v such that: (i) if $\theta_l \leq val(\phi, Q) \leq \theta_h$, then $val(\phi, Q) = v$; (ii) if $val(\phi, Q) \leq \theta_l$, then $v \leq \theta_l$; and (iii) if $val(\phi, Q) \geq \theta_h$, then $v \geq \theta_h$.*

Proof: The correctness of the algorithm will be proved by induction on the number of variables of the formula. If there are no variables, then ϕ either consists of no clauses (*i.e.*, it contains the empty set), or it contains an empty clause. If ϕ consists of no clauses, then ϕ evaluates to 1 by definition. Likewise, if ϕ contains an empty clause, this implies that $val(\phi, Q) = 0$, and thus the lemma holds in this case.

Now, assume that ϕ has $n > 0$ variables, and as usual, the variables are named $x_1, \ldots, x_n$ where x_1 is the outermost quantified variable. As above, the lemma holds if ϕ consists of no clauses or contains an empty clause. In all other cases, a single variable is removed. The first case is where a variable x_i is removed because it occurs in a singleton clause in ϕ with sign b, and $Q(x_i) = \exists$. In this case, $val(\phi, Q) = val(\phi\lceil_{x_i=b}, Q)$ because when x_i is set to $1-b$, ϕ evaluates to 0. Then, by the inductive hypothesis, the algorithm outputs a correct value on $\phi\lceil_{x_i=b}$. Similarly, if x_i is a unit variable in ϕ and $Q(x_i) = \forall$, then $val(\phi, Q) = 0$ and, thus, the algorithm outputs a correct value. Finally, if x_i is a unit variable and $Q(x_i) = \rotatebox[origin=c]{180}{R}$, then $val(\phi, Q) = (val(\phi\lceil_{x_i=b}, Q) + val(\phi\lceil_{x_i=1-b}, Q))/2 = val(\phi\lceil_{x_i=b}, Q)/2$. In this case, when calling evalssat recursively on $\phi\lceil_{x_i=b}$, the low and high bounds θ_l and θ_h must be shifted by a factor of two. That is, when $val(\phi, Q) = x$, $val(\phi\lceil_{x_i=b}, Q)$ is $2x$; thus if $\theta_l \leq x \leq \theta_h$, then by induction $\mathsf{evalssat}(\phi\lceil_{x_i=b}, Q, 2\theta_l, 2\theta_h)$ will return the value $2x$ since $2\theta_l \leq 2x \leq 2\theta_h$. Similarly, the algorithm is correct for $x \leq \theta_l$ and $x \geq \theta_h$.

The second case is where x_i is a pure variable with sign b and $Q(x_i) = \forall$. In this case, $val(\phi, Q) = \min(val(\phi\lceil_{x_i=b}, Q), val(\phi\lceil_{x_i=1-b}, Q)) = val(\phi\lceil_{x_i=1-b}, Q)$, so by induction the algorithm is correct. Similarly, when x_i is a pure variable and $Q(x_i) = \exists$, then

$$val(\phi, Q) = \max(val(\phi\lceil_{x_i=b}, Q), val(\phi\lceil_{x_i=1-b}, Q)) = val(\phi\lceil_{x_i=b}, Q).$$

The third case is when x_1 is removed and $Q(x_1) = \exists$. Let $v_0 = \mathsf{evalssat}(\phi\lceil_{x_1=0}, Q, \theta_l, \theta_h)$. If $val(\phi\lceil_{x_1=0}, Q) \geq \theta_h$, then $v_0 \geq \theta_h$ by the induction hypothesis, so v_0 is a correct value. Otherwise, $v_0 < \theta_h$. Let $v_1 = \mathsf{evalssat}(\phi\lceil_{x_1=1}), Q, \max(\theta_l, v_0), \theta_h)$. There are several simple cases to check: If $val(\phi\lceil_{x_1=0}, Q) \geq \theta_l$, then v_0 will equal $val(\phi\lceil_{x_1=0}, Q)$ by the inductive hypothesis, so $\max(\theta_l, v_0) = v_0$, and thus it is easy to check that $\max(v_0, v_1)$ is a correct answer. Similarly, it can be checked that the answer returned is correct if $val(\phi\lceil_{x_1=0}, Q) < \theta_l$.

The fourth case, when x_1 is removed and $Q(x_1) = \forall$, is very similar to the above case.

The fifth and final case is when x_1 is removed and $Q(x_1) = \text{Я}$. Let $v'_0 = val(\theta\lceil_{x_1=0}, Q)$ and let $v'_1 = val(\theta\lceil_{x_1=1}, Q)$. If $v'_0 < 2\theta_l - 1$ then $(v'_0 + v'_1)/2 < (2\theta_l - 1 + 1)/2 = \theta_l$. If $v'_0 > 2\theta_h$, then $(v'_0 + v'_1)/4 > (2\theta_h)/2 = \theta_h$. So, a detailed answer only matters if $v'_0 < 2\theta_l - 1$ or $v'_0 > 2\theta_h$. Thus, it suffices to calculate $v_0 = \mathsf{evalssat}(\theta\lceil_{x_1=0}, Q, 2\theta_l - 1, 2\theta_h)$. If $(v_0 + 1)/2 < \theta_l$, then $(v'_0 + v'_1)/2 < \theta_l$, so return $v_0/2$ (anything less than θ_l). If $v_0/2 \geq \theta_h$, then $(v'_0 + v'_1)/2 \geq v_0/2 \geq \theta_h$, so return $v_0/2$. Otherwise, calculate $v_1 = evalpset(\theta\lceil_{x_1=1}, 2\theta_l - v_0, 2\theta_h - v_0)$. Now, if $v_1 < 2\theta_l - v_0$, then $(v'_0 + v'_1)/2 < \theta_l$, and if $v_1 > 2\theta_h - v_0$, then $(v'_0 + v'_1)/2 > \theta_h$. ■

B. A Difficult Instance for sampleevalssat

Consider the n-variable SSAT formula

$$\text{Я}y_1, \text{Я}y_2, \ldots, \text{Я}y_{n-2}, \exists x_{n-1}, \text{Я}y_n((x_{n-1} \vee \overline{y_n})(\overline{x_{n-1}} \vee y_n)).$$

The value of this formula is $1/2$, since the formula is `True` if and only if $x_{n-1} = y_n$, which happens half of the time regardless of the value of x_{n-1}.

The number of policies for this SSAT formula is $P = 2^{2^{n-2}}$ since the setting for x_{n-1} can be conditioned on the preceding $n-2$ randomized variables.

Let W be a size w sample of the assignments to the randomized variables in the formula. The estimated value of the formula is

$$\sum_{\alpha \in W} |\{\beta | \beta = \alpha, \beta \in W\}| / |\{\beta | \beta_{n-2} = \alpha_{n-2}, \beta \in W\}|,$$

where α_{n-2} represents the part of the sample that assigns values to variables $y_1, \ldots, y_{n-2}$. The justification for this expression is that the existential variable immediately above any given sampled leaf will be set to maximize the number of sampled leaves below it that are made `True`. That is, it will be set to match the majority of the assignments to y_n immediately below it. For example, if a decision node has only one leaf beneath, the value of that leaf will be 1.

For any sample, α, with probability at least $1 - w/2^{n-1}$,

$$\{\beta|\beta = \alpha, \beta \in W\}| = |\{\beta|\beta_{n-2} = \alpha_{n-2}, \beta \in W\}|$$

because the second set is only larger than the first set if there is a sample that matches α in its assignment to $y_1, \ldots, y_{n-2}$ but differs in its assignment to y_n. Thus, the expected value of the sampleevalssat estimate $E[\hat{v}] \geq 1 - w/2^{n-1}$.

Let ϵ be a target approximation error and $1 - \delta$ be the desired probability of the estimate being within ϵ of the true value. This restricts $E[\hat{v}] \leq (1 - \delta)(1/2 + \epsilon) + (\delta)(1)$. Thus, $1 - w/2^{n-1} \leq E[\hat{v}] \leq (1 - \delta)(1/2 + \epsilon) + \delta$, or

$$w \geq 2^{n-2}(1 - \delta(1 - 2\epsilon) - 2\epsilon)).$$

This means that the number of samples needed to insure a good estimate from sampleevalssat is exponential in n.

References

Achlioptas, D. 1999. *Threshold phenomena in random graph colouring and satisfiability.* Ph.D. Dissertation, University of Toronto.

Bayardo, Jr., R. J., and Schrag, R. C. 1997. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 203–208. AAAI Press/The MIT Press.

Beame, P., and Pitassi, T. 1996. Simplified and improved resolution lower bounds. In *37th Annual Symposium on Foundations of Computer Science*, 274–282. IEEE.

Beame, P.; Karp, R.; Pitassi, T.; and Saks, M. 1998. On the complexity of unsatisfiability of random k-CNF formulas. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, 561–571.

Ben-Sasson, E., and Wigderson, A. 1999. Short proofs are narrow: Resolution made simple. To appear in STOC 1999.

Birnbaum, E., and Lozinskii, E. L. 1999. The good old Davis-Putnam procedure helps counting models. 10:457–477.

Cadoli, M.; Giovanardi, A.; and Schaerf, M. 1997. Experimental analysis of the computational cost of evaluating quantified Boolean formulae. In *Fifth Conference of the Italian Association for Artificial Intelligence (AI*IA'97)*, number 1321, 207–218. Springer-Verlag.

Cadoli, M.; Giovanardi, A.; and Schaerf, M. 1998. An algorithm to evaluate quantified Boolean formulae. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 262–267. The AAAI Press/The MIT Press.

Chvátal, V., and Reed, B. 1992. Mick gets some (the odds are on his side). In *33nd Annual Symposium on Foundations of Computer Science*, 620–627.

Clegg, M.; Edmonds, J.; and Impagliazzo, R. 1996. Using the Gröbner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, 174–183.

Condon, A.; Feigenbaum, J.; Lund, C.; and Shor, P. 1997. Random debators and the hardness of approximating stochastic functions. *SIAM Journal on Computing* 26(2):369–400.

Condon, A. 1992. The complexity of stochastic games. *Information and Computation* 96(2):203–224.

Crawford, J. M., and Auton, L. D. 1996. Experimental results in the crossover point in random 3SAT. *Artificial Intelligence* 81(1-2):31—57.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem proving. *Communications of the ACM* 5:394–397.

Dechter, R. 1996. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI-96)*, 211–219. Morgan Kaufmann Publishers.

Freeman, J. W. 1995. *Improvements to propositional satisfiability search algorithms*. Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania.

Friedgut, E. 1997. Necessary and sufficient conditions for sharp thresholds of graph properties, and the k-SAT problem. Preprint.

Frieze, A., and Suen, S. 1996. Analysis of two simple heuristics on a random instance of k-SAT. *Journal of Algorithms* 20(2):312–355.

Gallo, G., and Urbani, G. 1989. Algorithms for testing the satisfiability of propositional formulae. *Journal of Logic Programming* 7:45–61.

Gent, I., and Walsh, T. 1998. Beyond NP: the QSAT phase transition. Technical Report APES-05-1998.

Goerdt, A. 1996. A threshold for unsatisfiability. *Journal of Computer and System Sciences* 53:469–486.

Gu, J.; Purdom, P. W.; Franco, J.; and Wah, B. J. 1997. Algorithms for the satisfiability (SAT) problem: A survey. In *Satisfiability (SAT) Problem*, 19–151. DIMACS, American Mathematical Society.

Harche, F.; Hooker, J. N.; and Thompson, G. 1994. A computational study of satisfiability algorithms for propositional logic. *ORSA Journal on Computing* 6:423—435.

Hooker, J. N., and Vinay, V. 1994. Branching rules for satisfiability. Technical Report GSIA Working Paper 1994-09, Carnegie Mellon University. Revised January 1995.

Jeroslow, R., and Wang, J. 1990. Solving propositional satisfiability problems. *Annals of Mathematics and AI* 1:167—187.

Kamath, A.; Motwani, R.; Palem, K.; and Spirakis, P. 1995. Tail bounds for occupancy and the satisfiability threshold conjecture. *Random Structures and Algorithms* 7(1):59–80.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1194–1201. AAAI Press/The MIT Press.

Kearns, M.; Mansour, Y.; and Ng, A. Y. 1999. A sparse sampling algorithm for near-optimal planning large markov decision processes. Submitted.

Kirousis, L. M.; Kranakis, E.; Krizanc, D.; and Stamatiou, Y. C. 1998. Approximating the unsatisfiability threshold of random formulas. *Random Structures and Algorithms* 12(3):253–269.

Li, C. M., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 366—371.

Littman, M. L.; Goldsmith, J.; and Mundhenk, M. 1998. The computational complexity of probabilistic plan existence and evaluation. *Journal of Artificial Intelligence Research* 9:1–36.

Littman, M. L. 1997. Probabilistic propositional planning: Representations and complexity. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 748–754. AAAI Press/The MIT Press.

Lusena, C.; Goldsmith, J.; and Mundhenk, M. 1998. Nonapproximability results for Markov decision processes. Technical Report UK CS Dept TR 275-98, University of Kentucky.

Majercik, S. M., and Littman, M. L. 1998. MAXPLAN: A new approach to probabilistic planning. In Simmons, R.; Veloso, M.; and Smith, S., eds., *Proceedings of the Fourth International Conference on Artificial Intelligence Planning*, 86–93. AAAI Press.

Majercik, S. M., and Littman, M. L. 1999. Contingent planning under uncertainty via probabilistic satisfiability. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*. To appear.

Monien, B., and Speckenmeyer, E. 1985. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics* 10:287–295.

Mundhenk, M.; Goldsmith, J.; Lusena, C.; and Allender, E. 1997. Encyclopaedia of complexity results for finite-horizon Markov decision process problems. Technical Report UK CS Dept TR 273-97, University of Kentucky.

Papadimitriou, C. H., and Tsitsiklis, J. N. 1987. The complexity of Markov decision processes. *Mathematics of Operations Research* 12(3):441–450.

Papadimitriou, C. H. 1985. Games against nature. *Journal of Computer Systems Science* 31:288–301.

Paturi, R.; Pudlak, P.; Saks, M.; and Zane, F. 1998. An improved exponential time algorithm for k-SAT. Preprint.

Paturi, R.; Pudlak, P.; and Zane, F. 1997. Satisfiability coding lemma. In *FOCS*, 566–574.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence* 82(1–2):273–302.

Selman, B.; Kautz, H.; and Cohen, B. 1996. Local search strategies for satisfiability testing. In Johnson, D. S., and Trick, M. A., eds., *Cliques, Coloring, and Satisfiability*. American Mathematical Society. 521–531. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26.

Selman, B.; Mitchell, D.; and Levesque, H. 1996. Generating hard satisfiability problems. *Artificial Intelligence* 81:17–29.

Umans, C. 1999. Personal communication.