DC-SSAT: A Divide-and-Conquer Approach to Solving Stochastic Satisfiability Problems Efficiently

Stephen M. Majercik
Department of Computer Science
Bowdoin College
Brunswick, ME, USA 04011
smajerci@bowdoin.edu

Byron Boots
Center for Cognitive Neuroscience
Duke University
Durham, NC, USA 27708
bboots@duke.edu

Abstract

We present DC-SSAT, a sound and complete divide-and-conquer algorithm for solving stochastic satisfiability (SSAT) problems that outperforms the best existing algorithm for solving such problems (ZANDER) by several orders of magnitude with respect to both time and space. DC-SSAT achieves this performance by dividing the SSAT problem into subproblems based on the structure of the original instance, caching the viable partial assignments (VPAs) generated by solving these subproblems, and using these VPAs to construct the solution to the original problem. DC-SSAT does not save redundant VPAs and each VPA saved is necessary to construct the solution. Furthermore, DC-SSAT builds a solution that is already human-comprehensible, allowing it to avoid the costly solution rebuilding phase in ZANDER. As a result, DC-SSAT is able to solve problems using, typically, 1-2 orders of magnitude less space than ZANDER, allowing DC-SSAT to solve problems ZANDER cannot solve due to space constraints. And, in spite of its more parsimonious use of space, DC-SSAT is typically 1-2 orders of magnitude faster than ZANDER. We describe the DC-SSAT algorithm and present empirical results comparing its performance to that of ZANDER on a set of SSAT problems.

Introduction

Stochastic Boolean satisfiability (SSAT) (Papadimitriou 1985; Littman, Majercik, & Pitassi 2001) is a generalization of satisfiability (SAT) that is similar to quantified Boolean formulae (QBF). The ordered variables of the Boolean formula in an SSAT problem, instead of being existentially or universally quantified, are existentially or *randomly* quantified. Randomly quantified variables are `true` with a certain probability, and an SSAT instance is satisfiable with some probability that depends on the ordering of and interplay between the existential and randomized variables. The goal is to choose values for the existentially quantified variables that maximize the probability of satisfying the formula.

Like QBF, SSAT is PSPACE-complete, so it is theoretically possible to transform many probabilistic planning and reasoning problems of great practical interest into SSAT instances (Littman, Majercik, & Pitassi 2001). While such theoretically guaranteed translations are not always practical, previous work has shown that, in some cases, the transformation, and the solution of the resulting SSAT instance, can be done efficiently. For example, Majercik & Littman (2003) have developed ZANDER, a probabilistic planner that works by translating the planning problem into an SSAT instance and solving that problem to get the optimal plan. Freudenthal & Karamcheti (2003) have shown that SSAT can form the basis of a *trust management* (TM) system that addresses some of the limitations in state-of-the-art TM systems. And there is evidence that a belief net inferencing problem could be solved efficiently by translating it into a MAJSAT instance, a type of SSAT problem (Roth 1996; Bacchus, Dalmao, & Pitassi 2003). Thus, a more efficient SSAT solver could potentially give us better solution methods for a number of important practical problems. In addition, the development of such a solver would be valuable for the insights it might provide into solving probabilistic reasoning problems, in general, and other PSPACE-complete problems, such as QBF.

We describe DC-SSAT, a novel divide-and-conquer technique for solving SSAT problems that outperforms the best existing SSAT solver by several orders of magnitude with respect to both time and space. In the following sections, we describe the SSAT problem and DC-SSAT, present and discuss performance comparisons on a set of SSAT problems, and conclude with a discussion of related and further work.

Stochastic Satisfiability

An SSAT problem $\Phi = Q_1 v_1 \ldots Q_n v_n \phi$ is specified by a *prefix* $Q_1 v_1 \ldots Q_n v_n$ that orders a set of n Boolean variables $V = \{v_1, \ldots, v_n\}$ and a *matrix* ϕ that is a Boolean formula constructed from these variables. The prefix associates a quantifier Q_i, either existential ($\exists_i$) or randomized ($\mathfrak{R}_i^{\pi_i}$), with the variable v_i. The value of an existentially quantified (existential) variable can be set arbitrarily by a solver; the value of a randomly quantified (randomized) variable is determined stochastically by π_i, a rational number specifying the probability that v_i will be `true`. A *sub-block* of variables in the prefix is any sequence of adjacent, similarly quantified variables; a *block* is a maximal sub-block. An existential (randomized) block is a block of existential (randomized) variables. In this paper, we will use $x_1, x_2, \ldots$ for existential variables and $y_1, y_2, \ldots$ for randomized variables.

The matrix ϕ is assumed to be in conjunctive normal form, i.e. a set C of m conjuncted clauses, where each clause is a set of distinct disjuncted literals. A *literal* l is either a variable v (a *positive* literal) or its negation $\overline{v}$ (a *negative* literal). For a literal l, $|l|$ is the variable v underlying that literal and $\bar{l}$ is the "opposite" of l, i.e. if l is v, $\bar{l}$ is $\overline{v}$; if l is $\overline{v}$, $\bar{l}$ is v. A

literal l is `true` if it is positive and $|l|$ has the value `true`, or if it is negative and $|l|$ has the value `false`. A literal is *existential* (*randomized*) if $|l|$ is existential (randomized). The probability that a randomized variable v has the value `true` (`false`) is denoted $Pr[v]$ ($Pr[\overline{v}]$). The probability that a randomized literal l is `true` is denoted $Pr[l]$. A clause is satisfied if at least one literal is `true`, and unsatisfied, or *empty*, if all its literals are `false`. The formula is satisfied if all its clauses are satisfied.

A partial assignment α of the variables V is a sequence of $k \leq n$ literals $l_1; l_2; \ldots; l_k$ such that no two literals in α have the same underlying variable. Given l_i and l_j in an assignment α, $i < j$ implies that the assignment to $|l_i|$ was made before the assignment to $|l_j|$. A positive (negative) literal v ($\overline{v}$) in an assignment α indicates that the variable v has the value `true` (`false`). The notation $\Phi(\alpha)$ denotes the SSAT problem Φ' remaining when the partial assignment α has been applied to Φ (i.e. clauses with `true` literals have been removed from the matrix, `false` literals have been removed from the remaining clauses in the matrix, and all variables and associated quantifiers not in the remaining clauses have been removed from the prefix) and $\phi(\alpha)$ denotes ϕ', the matrix remaining when α has been applied. Similarly, given a set of literals L, such that no two literals in L have the same underlying variable, the notation $\Phi(L)$ denotes the SSAT problem Φ' remaining when the assignments indicated by the literals in L have been applied to Φ, and $\phi(L)$ denotes ϕ', the matrix remaining when the assignments indicated by the literals in L have been applied. A literal $l \notin \alpha$ is *active* if some clause in $\phi(\alpha)$ contains l; otherwise it is *inactive*.

Given an SSAT problem Φ, the maximum probability of satisfaction of ϕ, denoted $Pr[\Phi]$, is defined according to the following recursive rules:

1. If ϕ contains an empty clause, $Pr[\Phi] = 0.0$.
2. If ϕ is the empty set of clauses, $Pr[\Phi] = 1.0$.
3. If the leftmost quantifier in the prefix of Φ is existential and the variable thus quantified is v, then $Pr[\Phi] = \max(Pr[\Phi(v)], Pr[\Phi(\overline{v})])$.
4. If the leftmost quantifier in ϕ is randomized and the variable thus quantified is v, then $Pr[\Phi] = (Pr[\Phi(v)] \times Pr[v]) + (Pr[\Phi(\overline{v})] \times Pr[\overline{v}])$.

The solution of an SSAT instance is an assignment of truth values to the existential variables that yields the maximum probability of satisfaction, denoted $Pr[\Phi]$. Since the values of existential variables can be made contingent on the values of randomized variables that appear earlier in the prefix, the solution is, in general, an *assignment tree* that specifies the optimal assignment to each existential variable x_i for each possible instantiation of the randomized variables that precede x_i in the prefix.

DC-SSAT

A direct implementation of the rules defining $Pr[\Phi]$ would yield a functional, though impractical, SSAT solver. ZANDER, the most highly optimized SSAT solver developed by Majercik & Littman (2003), improves on this by adapting the Davis-Putnam-Logemann-Loveland (DPLL) SAT algorithm (Davis, Logemann, & Loveland 1962). It traverses the tree of possible assignments, following the variable ordering in the prefix of Φ, and applies the rules described above to calculate $Pr[\Phi]$. The solver prunes subtrees, when possible, by using a *unit propagation* rule adapted from DPLL and *thresholding*. Details are available in Majercik & Littman (2003). Even with unit propagation and thresholding, however, this DPLL-based approach is often infeasible due to the size of the assignment tree that must still be explored. Furthermore, ZANDER, like DC-SSAT, produces the optimal solution tree (the size of which can be exponential in the number of randomized variables). Since different partial assignments often lead to the same subproblem, the efficiency of a solver can be improved by caching and reusing the results of solved subproblems. ZANDER relies heavily on this technique (called *memoizing* in Majercik & Littman (2003)) to solve larger problems. ZANDER, however, saves *every* solved subproblem, which can quickly exhaust the available memory, and often only a small percentage of cached subproblems are actually reused.

Since ZANDER is tailored to solve SSAT encodings of probabilistic planning problems (Majercik & Littman 2003), DC-SSAT was developed to solve the same kind of problems. In this paper, we focus on completely observable probabilistic planning (COPP) problems and so use a version of ZANDER that is optimized for this type of problem. DC-SSAT however, can be generalized to solve SSAT encodings of partially observable planning problems and general SSAT problems; we will describe this in a future paper.

The key idea underlying DC-SSAT is to enlarge the size and scope of the units being manipulated by the solver. In order to solve an SSAT problem, any solver needs to consider, directly or indirectly, *every* possible satisfying assignment. Rather than covering this territory at the fine-grained resolution of considering possible values for individual variables, DC-SSAT covers this territory more efficiently by constructing subproblems that include multiple variables, and caching information from the solution of these subproblems. Furthermore, these subproblems are based on the structure of the particular SSAT instance being solved; the variable ordering in the prefix and the relationships among the variables induced by the clauses dictate the composition of these building blocks for the final solution. Each of these building blocks is necessary to compute the final solution and, together, they almost always require much less space than ZANDER's cached subproblems, resulting in a more efficient calculation of the solution.

An SSAT encoding of a COPP problem (a COPP-SSAT problem) has some structural characteristics that are exploited by both DC-SSAT and ZANDER. For DC-SSAT, the important characteristics are that 1) a COPP-SSAT problem always starts with an existential block, and 2) variables in an existential block appear in clauses only with variables from the same existential block and variables from adjacent randomized blocks. Given SSAT problem $\Phi = Q_1 v_1 \ldots Q_n v_n \phi$, let $X \subseteq V$ be the set of all existential variables in Φ and let $Y \subseteq V$ be the set of all randomized variables in Φ. We define the following relation R on the set of

variables X. Variable x_1 is related to variable x_m if there is a sequence of clauses $c_1, \ldots c_m$, $m \geq 1$, such that x_1 is in clause c_1, x_m is in clause c_m, and, if $m > 1$, the clauses in each pair c_i and c_{i+1}, $1 \leq i < m$, both contain a common variable $x_i \in X$. Clearly, R defines an equivalence relation on X. Note that because of the special structure of a COPP-SSAT problem, the variables in each existential block will form an equivalence class. For each distinct equivalence class $X_i = [x]_R \subseteq X$ induced by R:

1. Let $C_i = \{c_i \mid l \in c_i \wedge |l| \in X_i \}$, the set of clauses containing at least one literal whose underlying variable is in X_i.
2. Let $W_i = \bigcup_{c_i \in C_i} \{|l| \mid l \in c_i\}$, the set of variables underlying the literals in the clauses in C_i. Clearly, $X_i \subseteq W_i$.
3. Put a clause c_π containing all random literals in the clause set C_i such that the number of literals in c_π whose underlying variables are in W_i is maximized, breaking ties in favor of the clause set containing the left-most existential variable in the prefix of Φ. Let Y_i be the set of all randomized variables in such clauses added to C_i.

Let $V_i = W_i \cup Y_i$, and let $Q_1 v_1 \ldots Q_{m_i} v_{m_i}, (m_i = |V_i|)$ be the prefix obtained by sequencing the variables in V_i along with their quantifiers such that this sequence respects the ordering in the prefix of Φ. Let $\phi_i = C_i$. Then subproblem $\Phi_i = Q_1 v_1 \ldots Q_{m_i} v_{m_i} \phi_i$. Finally, we define an ordering on the subproblems. Define *rank*(v), the *rank* of a variable $v \in V$, to be the number of its position in the prefix $Q_1 v_1 \ldots Q_n v_n$, i.e. v_1 has rank 1, v_2 has rank 2, etc. Then subproblem Φ_i precedes subproblem Φ_j if $\min_{v \in V_i}(\textit{rank}(v)) < \min_{v \in V_j}(\textit{rank}(v))$, i.e. subproblems are ordered according to the rank of the variable in the subproblem that occurs earliest in prefix of Φ. In a COPP-SSAT problem, this means that subproblems will be in the same order as the existential blocks in the prefix of Φ. Creating these subproblems can be done in time $\mathcal{O}(n+m)$ in a sweep through the variables, assuming that each variable points to a list of clauses containing that variable.

DC-SSAT solves each subproblem $\Phi_1, \Phi_2, \ldots, \Phi_s$ using a DPLL-based algorithm to generate a set of *viable partial assignments* (VPAs) that satisfy the clauses in that subproblem, and then uses these VPAs to construct the solution to Φ. A good analogy is that in the same way a DPLL-based SSAT solver does a depth-first search on variable assignments, DC-SSAT does a depth-first search on VPAs. More formally, a VPA α in subproblem Φ_i is a partial assignment to the variables V_i such that the sequence of literals in α respects the ordering of the prefix of Φ_i and all the clauses in C_i are satisfied. The probability of α is defined as $\Pi_{l \in \alpha \wedge |l| \in Y} Pr[l]$, where we are using set membership notation to indicate that l is in the sequence of literals constituting the assignment α. Two VPAs, α_i and α_j, are *contradictory* if, for any $l \in \alpha_i$, we have $\bar{l} \in \alpha_j$. The *union* of two contradictory VPAs, $\alpha_i \cup \alpha_j$, is the empty assignment $\alpha_\emptyset$ and the probability of $\alpha_\emptyset$ is defined to be 0.0. Two VPAs, α_i and α_j, *agree* ($\alpha_i \sim \alpha_j$) or are *compatible* iff they are not contradictory. Let $\alpha_i \sim \alpha_j$ for VPAs α_i and α_j. Then the set of literals in α_k, the union of α_i and α_j, is the union of the literals in α_i and α_j, sequenced to respect the variable ordering in the prefix of Φ. The probability of α_k is defined as before. This definition can be extended to the union of a finite number of VPAs in a natural way.

Let $\text{VPA}_1, \text{VPA}_2, \ldots, \text{VPA}_s$ be the sets of VPAs generated by the solution of subproblems $\Phi_1, \Phi_2, \ldots, \Phi_s$. Let VPA_Φ be the set of all non-empty VPAs obtained by taking the union of VPAs in each s-tuple of $\text{VPA}_1 \times \text{VPA}_2 \times \ldots \times \text{VPA}_s$. It is easy to see that each VPA in VPA_Φ satisfies all the clauses in Φ and so could be a path in the optimal assignment tree for Φ. In fact, we have the following lemmas:

Lemma 1: *Each path in the optimal assignment tree for Φ describes a VPA that is contained in VPA$_\Phi$.*

Lemma 2: *All VPAs in VPA$_\Phi$ must be considered in order to construct the optimal assignment tree for Φ.*

Two VPAs, α_i and α_j, *existential-agree* or are *existential-compatible* ($\alpha_i \sim_\exists \alpha_j$) iff for each existential literal l_i in α_i and existential literal l_j in α_j, if $|l_i| = |l_j|$ and $l_i \neq l_j$, there is a random literal l_k in that appears before l_i in α_i such that $\overline{l_k}$ appears before l_j in α_j. In other words, α_i and α_j cannot prescribe a different value for an existential variable unless some randomized variable that appears earlier in both VPAs has a different value. Thus, two different, but existential-compatible, VPAs α_i and α_j can be combined to produce an assignment tree with a single path that splits at some point. The initial common path segment is the union of the compatible initial subsequences of α_i and α_j, and the path splits at the randomized variable whose value they disagree on. The probability of this tree is defined in a manner similar to the rules defined for determining $Pr[\Phi]$: the probability of an existential variable node is the maximum of the probabilities of its children; the probability of a randomized variable node is the probability-weighted average of the probabilities of its children. In both cases, the probability of a missing child is 0.0. This combination operation can be extended to a finite number of VPAs in a natural way; we define a *viable solution* (VS) as the combination of a finite set of existential-compatible VPAs from VPA_Φ. Then we have:

Theorem 1: *The optimal assignment tree for Φ is a viable solution with maximal probability.*

One can think of the solution tree for Φ as the "best" combination of existential-compatible VPAs in VPA_Φ. Thus, a straightforward way of calculating the solution to Φ would be to generate all viable solutions and calculate their probabilities, selecting one with maximal probability. But, generating VPAs in each subproblem independently can produce many VPAs that could never be part of a solution because they contradict every VPA that could be part of a solution in earlier subproblems and so could not be part of a path in any possible solution tree. We can avoid this, and speed up the solution process considerably, by using the VPAs generated by earlier subproblems to constrain the VPAs generated by the remaining subproblems. Since the variables in an existential block in a COPP-SSAT problem appear in clauses only with variables from the same block and/or variables from adjacent randomized blocks, the variables shared between subproblems Φ_i and Φ_{i+1} will be some number of variables that appear at the end of the prefix of Φ_i and the beginning of

the prefix of Φ_{i+1}. While DC-SSAT is solving Φ_i, it can generate all the instantiations of the variables it shares with Φ_{i+1} that might be part of a solution. An assignment β to these shared variables is called a *branch assignment* (BA), since these variables are all randomized variables and constitute the possible branches in the solution tree that Φ_i could generate. We will denote the set of BAs between Φ_i and Φ_{i+1} by B_i^{i+1} and each $\beta_j \in B_i^{i+1}$ will point to a structured VPA list (see next paragraph) of those VPAs in Φ_i that agree with β_i. Furthermore, the solution of Φ_{i-1} has already generated B_{i-1}^i, which is the set of possible assignments to the variables shared between Φ_{i-1} and Φ_i, i.e. all the possible assignments to the initial variables in the prefix of Φ. So DC-SSAT can use each $\beta_j \in B_{i-1}^i$ as a starting point from which to solve Φ_i, thus guaranteeing that all VPAs generated will agree with some VPA in Φ_{i-1}; any VPA α thus generated is added to the VPA list of $\beta_j \in B_{i-1}^i$ and set to point to the BA $\beta_k \in B_i^{i+1}$ that it is compatible with, creating that BA if it does not already exist.

The structured list of VPAs that $\beta_j \in B_{i-1}^i$ points to is a list of *existential assignments* (EAs), i.e. assignments to the existential variables that subproblem Φ_i was based on that are compatible with some VPA in B_{i-1}^i. $E_{\beta_j}(\beta_j \in B_{i-1}^i)$ denotes the list of EAs that agree with the BA β_j, and $\varepsilon_k \in E_{\beta_j}$ points to the list of VPAs in Φ_i that agree with both the BA β_j and the EA ε_k. Finally, each VPA α in this list points to the BA $\beta_p \in B_i^{i+1}$ that α agrees with, i.e. the instantiation of the shared variables that this VPA would give rise to and that the following subproblem would have to "respond to." We will denote the BA $\beta_p \in B_i^{i+1}$ that α points to as $\alpha_{\rightarrow\beta}$. For each VPA α in Φ_s, $\alpha_{\rightarrow\beta}$ is defined to be NULL. This process of generating and installing VPAs can be done in time linear in the number of VPAs generated. Pseudocode for this process follows:

PROCESS-VPAS(Φ)
  partition Φ into subproblems $\Phi_1, \Phi_2, \ldots, \Phi_s$
  for each $\Phi_i, i = 1$ to s:
    for each $\beta_j \in B_{i-1}^i$:
      solve $\Phi_i(\beta_j)$ using a DPLL-based algorithm.
      for each VPA α generated:
        if $(\neg\exists\ \varepsilon_k \in E_{\beta_j}$, such that $\alpha \sim_\exists \varepsilon_k)$
          create ε_k such that $\alpha \sim_\exists \varepsilon_k$
        add α to the list of VPAs in ε_k
        if $(i < s \wedge \neg\exists\ \beta_p \in B_i^{i+1},$ such that $\alpha \sim \beta_p)$
          create $\beta_p \in B_i^{i+1}$ such that $\alpha \sim \beta_p$
        if $(i < s)$ set $\alpha_{\rightarrow\beta}$ to β_p; else set $\alpha_{\rightarrow\beta}$ to NULL

We define $Pr[\beta_j]$, the probability of a BA $\beta_j \in B_{i-1}^i$, as $\max_{\varepsilon_k \in E_{\beta_j}} Pr[\varepsilon_k]$ where $Pr[\varepsilon_k]$ is defined recursively:

$$Pr[\varepsilon_k] = \begin{cases} \sum_{\alpha \in \varepsilon_k} Pr[\alpha_{\rightarrow\beta}] & \text{if } \alpha_{\rightarrow\beta} \neq \text{ NULL} \\ \sum_{\alpha \in \varepsilon_k} Pr[\alpha] & \text{otherwise} \end{cases}$$

where $\alpha_{\rightarrow\beta} \in B_i^{i+1}$. Since a COPP-SSAT problem always starts with an existential block, the only branch instantiation in Φ_1 is $\beta_\emptyset$, the empty BA, so $B_0^1 = \{\beta_\emptyset\}$. All VPAs in Φ_1 are in the structured list pointed to by this BA since all VPAs are compatible with $\beta_\emptyset$, and it can be shown that $Pr[\Phi] = Pr[\beta_\emptyset]$. Thus, DC-SSAT finds the solution to Φ by computing $Pr[\beta_\emptyset]$ and, in the process makes each VPA α in each EA point to the optimal EA in the next subproblem, i.e. the optimal setting of the existential variables in the next subproblem given the current partial assignment. Thus, DC-SSAT avoids building a solution tree explicitly and prints the solution tree by following these pointers from the optimal EA in the first subproblem. As a final optimization, DC-SSAT caches the probability of each EA as it is calculated, so as DC-SSAT traverses the tree of VPAs it can avoid recalculating the probability of an EA. It can be shown that (like ZANDER):

Theorem 2: DC-SSAT is a *sound* and *complete* algorithm for solving COPP-SSAT problems.

An example will help clarify the algorithm. Suppose we have the following SSAT problem:

$$\exists x_1\ \text{ᗯ}^{0.2} y_1\ \exists x_2\ \text{ᗯ}^{0.6} y_2\ \{\{\overline{x_1}, y_1\}, \{x_1, \overline{y_1}\}, \{y_1, x_2, y_2\}, \{y_1, x_2, \overline{y_2}\}, \{\overline{y_1}, \overline{x_2}, y_2\}, \{\overline{x_2}, y_2\}, \{x_2, \overline{y_2}\}\}\ .$$

Since the existential variables x_1 and x_2 are not related under R, each one forms its own equivalence class. The variables x_1 and y_1 and the first two clauses comprise Φ_1, and the variables y_1, x_2, and y_2 and the remaining clauses comprise Φ_2.

DC-SSAT solves $\Phi_1(\beta_\emptyset)$, generating two VPAs: $\alpha_1 = x_1; y_1$ and $\alpha_2 = \overline{x_1}; \overline{y_1}$. If α_1 is generated first, DC-SSAT creates $(\varepsilon_1 = x_1) \sim_\exists \alpha_1$, puts ε_1 in $E_{\beta_\emptyset}$ and puts α_1 in the list of VPAs in ε_1. Also, since $B_1^2 = \emptyset$, DC-SSAT creates $\beta_1 = y_1$, puts β_1 in B_1^2, and makes α_1 point to β_1. Since $\alpha_2 \not\sim_\exists \varepsilon_1 \in E_{\beta_\emptyset}$, DC-SSAT creates $(\varepsilon_2 = \overline{x_1}) \sim_\exists \alpha_2$, adds ε_2 to $E_{\beta_\emptyset}$ and puts α_2 in the list of VPAs in ε_2. And, since $\alpha_2 \not\sim \beta_1 \in B_1^2$, DC-SSAT creates $\beta_2 = \overline{y_1}$, adds β_2 to B_1^2, and makes α_2 point to β_2. Since there are no more BAs in B_0^1, DC-SSAT has finished processing Φ_1. Note that B_1^2 now contains a list of viable assignments to the variables that are shared by Φ_1 and Φ_2.

DC-SSAT next solves Φ_2 by looking for assignments to the variables in Φ_2 that extend the assignments in B_1^2 and satisfy the clauses in Φ_2. It starts with $\beta_1 = y_1$. This generates two VPAs: $\alpha_3 = y_1; x_2; y_2$ and $\alpha_4 = y_1; \overline{x_2}; \overline{y_2}$. If α_3 is generated first, DC-SSAT creates $(\varepsilon_3 = x_2) \sim_\exists \alpha_3$, puts ε_3 in E_{β_1} and puts α_3 in the list of VPAs in ε_3. Since $\alpha_4 \not\sim_\exists \varepsilon_3 \in E_{\beta_1}$, DC-SSAT creates $(\varepsilon_4 = \overline{x_2}) \sim_\exists \alpha_4$, adds ε_4 to E_{β_1}, and puts α_4 in the list of VPAs in ε_4. Similarly, DC-SSAT finds that $\alpha_5 = \overline{y_1}; x_2; y_2$ extends $\beta_2 = \overline{y_1}$, so $(\varepsilon_5 = x_2) \sim_\exists \alpha_5$ is placed in E_{β_2}, and α_5 is placed in the list of VPAs in ε_5. Since there are no more subproblems, DC-SSAT makes α_3, α_4, and α_5 all point to NULL.

DC-SSAT then does a depth-first search of the VPAs, guided by the structure of the VPA lists, computing $Pr[\Phi]$ recursively. Briefly, α_1, the VPA associated with $\varepsilon_1 = x_1$ agrees with α_3, the VPA associated with $\varepsilon_3 = x_2$ and this solution ($x_1 = \texttt{true}, x_2 = \texttt{true}$) has a probability of satisfaction of 0.12 (both y_1 and y_2 must be `true`). The VPA α_1 also agrees with α_4, the VPA associated with $\varepsilon_4 = \overline{x_2}$, but this solution ($x_1 = \texttt{true}, x_2 = \texttt{false}$) has a probability of satisfaction of only 0.08 (y_1 must be `true` and y_2 must be `false`) and is rejected. The VPA α_2, associated with $\varepsilon_2 = \overline{x_1}$ agrees with α_5, the VPA associated with $\varepsilon_5 = x_2$ and this solution ($x_1 = \texttt{false}, x_2 = \texttt{true}$) has a probabil-

Problem (States in Plan Prob)	Size Statistic	Number of Steps = Number of ECs/SPs		
		5	t	50
COF (256)	NV	103	$19t + 8$	958
	NC	290	$56t + 10$	2810
	AVSP	27.0	27.0	27.0
	ACSP	58.0	~ 57	56.2
SPF (256)	NV	103	$19t + 8$	958
	NC	362	$70t + 12$	3512
	AVSP	27.0	27.0	27.0
	ACSP	72.4	~ 71	70.2
FAC (65536)	NV	276	$52t + 16$	2616
	NC	2007	$397t + 22$	19872
	AVSP	68.0	68.0	68.0
	ACSP	401.4	~ 399	397.4
LI10 (1024)	NV	145	$27t + 10$	1360
	NC	660	$128t + 20$	6420
	AVSP	37.0	37.0	37.0
	ACSP	132.0	~ 130	128.4
EX4 (16)	NV	59	$11t + 4$	554
	NC	178	$34t + 8$	1708
	AVSP	15.0	15.0	15.0
	ACSP	35.6	~ 35	34.2

NV=Num Vars, NC=Num Clauses, EC=Equiv Class, SP=Subproblem, AVSP=Avg NV/SP, ACSP=Avg NC/SP

Table 1: Problem and decomposition characteristics.

Problem (States in Plan Prob)	Size Statistic	Number of Steps = Number of ECs		
		5	t	50
GO6 (128)	NV	107	$20t + 7$	1007
	NC	374	$72t + 14$	3614
	AVSP	27.0	27.0	27.0
	ACSP	74.8	~ 73	72.3
GO7 (256)	NV	123	$23t + 8$	1158
	NC	451	$87t + 16$	4366
	AVSP	31.0	31.0	31.0
	ACSP	90.2	~ 88	87.3
GO8 (512)	NV	139	$26t + 9$	1309
	NC	533	$103t + 18$	5168
	AVSP	35.0	35.0	35.0
	ACSP	106.6	~ 104	103.4
GO9 (1024)	NV	155	$29t + 10$	1460
	NC	620	$120t + 20$	6020
	AVSP	39.0	39.0	39.0
	ACSP	124.0	~ 121	120.4

NV=Num Vars, NC=Num Clauses, EC=Equiv Class, SP=Subproblem, AVSP=Avg NV/SP, ACSP=Avg NC/SP

Table 2: Problem and decomposition characteristics.

ity of satisfaction of 0.48 (y_1 must be `false` and y_2 must be `true`), so this is the optimal assignment.

Experimental Results

We tested DC-SSAT and ZANDER on a set of COPP-SSAT problems adapted from Majercik & Littman (2003) (COF = coffeebot, GOx = general-operations-x) and from (Hoey *et al.* 1999) (FAC = factory, EX4 = exponential04, LI10 = linear10), and on one additional problem (SPF = spearfishing). We omit problem descriptions, but provide information regarding problem size and the DC-SSAT decomposition of each problem in Tables 1 and 2. Tables 3 and 4 describe resource usage on problem instances of increasing size; the better time/space usage in each case is indicated in bold face. Reported results are representative of all results obtained. All tests were run on a Power Mac G5 with dual 2.5 GHz CPUs, 2 GB RAM, 512KB L2 cache/CPU, running Mac OS X 10.3.8. We omit probabilities of satisfaction due to lack of space.

In the COF problem, DC-SSAT is 1-2 orders of magnitude faster and uses 1-2 orders of magnitude less space (Table 3). In the largest COF problem that ZANDER was able to solve before exhausting memory (the 110-step problem, not shown, with 2098 variables and 6170 clauses), ZANDER required 43.61 CPU seconds and used 1411.32 MB of memory, while DC-SSAT was able to solve the problem in 0.35 CPU second using 25.12 MB of memory, 2 orders of magnitude faster and using 2 orders of magnitude less space.

Table 3 shows that, in most cases, ZANDER exhausted memory on the SPF problem and was timed out after 20 minutes on the FAC problem (which provided the largest SSAT instances in our test set; see Table 1). In the 10-step SPF problem, the one problem that both solvers completed that had a non-zero probability of satisfaction, the difference is dramatic: DC-SSAT was 3 orders of magnitude faster and used 2 orders of magnitude less space. Although ZANDER frequently uses much of the time and space it consumes to rebuild its solution tree, this was not the case in SPF and FAC, where the resources were consumed mostly in the solution calculation phase. We argue below that this is due to the larger number of actions (which increases the size of the existential blocks) in SPF (6 actions) and FAC (14 actions) compared to COF (4 actions).

Neither the SSAT problem size nor the size of the subproblems in the DC-SSAT decomposition seems to be a reliable indicator of DC-SSAT's performance; e.g. the 50-step FAC problem is approximately 2-3 times larger than the 50-step LI10 problem on all size statistics (Table 1), but DC-SSAT's solution time for that FAC problem was approximately $\frac{1}{4}$ its solution time for the LI10 problem. LI10 (like EX4) is an artificial problem; together, they shed some light on this issue and on the generally extreme differences between the performances of DC-SSAT and ZANDER. The LI10 problem requires a number of time steps (existential blocks) that is *linear* in the number of variables in each of these blocks in order to obtain a nonzero probability of satisfaction, although the actions must be executed in just the right order. Aided by unit propagation, ZANDER quickly establishes the correct sequence of actions, starting with the final action and working its way back to the initial action (although ZANDER exhausts memory as the problem size increases; see Table 3). DC-SSAT is fooled by the fact that each of the 10 actions

Problem	Solver	Resource	Resource Usage by Number of Steps in Plan									
			5	10	15	20	25	30	35	40	45	50
COF	DC	CS	**0.02**	**0.02**	**0.03**	**0.04**	**0.07**	**0.07**	**0.08**	**0.12**	**0.12**	**0.14**
		MB	0.04	0.27	**0.63**	**1.08**	**1.63**	**2.27**	**3.01**	**3.84**	**4.64**	**5.65**
	ZA	CS	0.06	0.03	0.07	0.25	0.84	2.19	3.85	5.62	7.53	9.48
		MB	**0.00**	**0.14**	1.55	9.46	38.78	111.59	192.60	273.65	354.72	435.83
SPF	DC	CS	**0.01**	**0.03**	**0.06**	**0.08**	**0.11**	**0.13**	**0.17**	**0.20**	**0.22**	**0.25**
		MB	0.07	**0.48**	**1.07**	**1.80**	**2.70**	**3.74**	**4.94**	**6.30**	**7.59**	**9.23**
	ZA	CS	0.02	29.87	–	–	–	–	–	–	–	–
		MB	**0.01**	28.69	M	M	M	M	M	M	M	M
FAC	DC	CS	**0.17**	**2.65**	**7.81**	**14.26**	**19.95**	**26.01**	**30.51**	**37.20**	**43.36**	**50.80**
		MB	**0.70**	**16.44**	**50.47**	**100.27**	**162.08**	**237.36**	**326.11**	**428.33**	**544.02**	**673.18**
	ZA	CS	3.47	1200+	1200+	1200+	1200+	1200+	1200+	1200+	1200+	1200+
		MB	0.99	–	–	–	–	–	–	–	–	–
LI10	DC	CS	0.11	8.71	25.12	44.55	66.94	92.01	**119.16**	**146.81**	**178.58**	**205.37**
		MB	1.20	24.53	66.75	126.72	**196.84**	**280.08**	**376.46**	**497.12**	**621.39**	**758.78**
	ZA	CS	**0.02**	**0.02**	**0.47**	**2.60**	**7.60**	**24.83**	–	–	–	–
		MB	**0.00**	**0.00**	**3.83**	**37.48**	206.83	1002.53	M	M	M	M
EX4	DC	CS	0.00	**0.01**	**0.01**	**0.02**	**0.02**	**0.03**	**0.04**	**0.04**	**0.06**	**0.06**
		MB	0.01	0.07	0.19	0.37	0.56	**0.81**	**1.09**	**1.35**	**1.70**	**2.08**
	ZA	CS	0.00	0.03	39.89	201.93	291.22	355.95	1200+	1200+	1200+	1200+
		MB	**0.00**	**0.00**	**0.00**	**0.04**	**0.27**	0.99	–	–	–	–

DC=DC-SSAT, ZA=ZANDER, CS=CPU secs, MB=MB mem, 1200+=Timed out, M=Mem exhausted, 0.00=Rounding

Table 3: DC-SSAT consistently outperforms ZANDER on the test problems.

can lead to a different state, creating 10 possible BAs in B_1^2 and forcing DC-SSAT to search through a large number of VPA combinations; its solution times on smaller problems in which ZANDER does not exhaust memory are 1-2 orders of magnitude larger than ZANDER's.

By contrast, the EX4 problem requires a number of time steps (existential blocks) that is *exponential* in the number of variables in each of these blocks in order to obtain a nonzero probability of satisfaction, and there are many relatively long paths in the tree of assignments that lead to unsatisfiability. ZANDER builds a huge tree exploring these paths, and caches an enormous number of subproblems, *none of which it is ever able to use*. DC-SSAT however, by focusing on "reachable" VPAs, is able to establish quickly (and using little space) which actions are potentially useful. For example, in subproblem Φ_1, there are only two possible BAs in B_1^2 (no matter how many actions there are), which greatly limits the VPAs generated in subproblem Φ_2. (ZANDER would explore a subtree for each possible action.) DC-SSAT is consistently 3-4 orders of magnitude faster on these problems, and, although ZANDER uses modestly less space on smaller instances of this problem, DC-SSAT uses less space by the time the number of plan steps reaches 30 (Table 3).

ZANDER's performance seems to be most strongly influenced by the number of actions in the problem (the size of the existential blocks). Each additional existential variable creates another subtree for ZANDER to build and explore, and this occurs at multiple levels of the tree, producing an exponential impact. The effect of this seems clear in SPF and FAC, and in the series of GOx problems (Table 4), where ZANDER's performance deteriorates as x, the number of actions, increases. DC-SSAT's performance seems more strongly influenced by the number of branch assignments produced in each subproblem (the size of the randomized blocks), since this dictates the number of VPA combinations it must explore; thus, DC-SSAT is more likely to have difficulty as both the number of randomized variables and the percentage of their instantiations that are viable increases.

Related and Further Work

Our main contribution is to show that a useful class of SSAT problems, COPP-SSAT problems, can be solved by a divide-and-conquer technique that is efficient in terms of both time and space. Our approach is similar to SAT cut-set approaches that attempt to divide the SAT problem into subproblems by instantiating a subset of variables. In a COPP-SSAT problem, the variables in the branching assignments (BAs) can be viewed as a cut-set that decomposes the problem. Instead of instantiating variables in the cut-set and propagating this information, however, DC-SSAT uses the BAs as a conduit for information between adjacent subproblems. This interplay between subproblems, mediated by a subset of common variables, is similar to the *delayed cut variable binding* technique of Park & Van Gelder (1996).

Although we view DC-SSAT primarily as a contribution to solving SSAT problems efficiently, our tests suggest that it is also a viable technique for solving probabilistic planning problems. (For a discussion of the probabilistic planning literature see Majercik & Littman (2003).) In fact, since the structure of a COPP-SSAT problem ensures that all the subproblems, except for the first and last ones, are isomorphic, it

Problem	Solver	Resource	Resource Usage by Number of Steps in Plan									
			5	10	15	20	25	30	35	40	45	50
GO6	DC	CS	**0.03**	**0.11**	**0.22**	**0.34**	**0.43**	**0.53**	**0.68**	**0.76**	**0.84**	**0.96**
		MB	0.31	2.44	5.43	**9.22**	**13.78**	**19.13**	**26.07**	**33.12**	**40.95**	**49.56**
	ZA	CS	0.07	0.65	0.73	1.17	2.95	9.08	26.28	–	–	–
		MB	**0.04**	**0.77**	**3.25**	20.75	104.82	402.50	1254.17	M	M	M
GO7	DC	CS	**0.05**	**0.29**	**0.57**	**0.87**	**1.21**	**1.57**	**1.99**	**2.46**	**2.95**	**3.46**
		MB	0.49	**5.38**	**13.05**	**22.22**	**34.37**	**47.42**	**64.06**	**83.09**	**102.12**	**125.64**
	ZA	CS	0.14	12.81	13.42	15.30	21.03	45.04	–	–	–	–
		MB	**0.06**	9.34	13.41	51.00	293.12	1379.37	M	M	M	M
GO8	DC	CS	**0.06**	**1.02**	**2.54**	**4.59**	**7.11**	**10.24**	**13.70**	**17.87**	**20.71**	**23.60**
		MB	0.81	**11.54**	**29.20**	**52.23**	**80.65**	**114.44**	**153.61**	**198.16**	**248.09**	**303.40**
	ZA	CS	0.18	368.04	393.09	427.61	451.85	1200+	1200+	1200+	1200+	1200+
		MB	**0.09**	228.36	234.54	301.16	878.43	–	–	–	–	–
GO9	DC	CS	**0.10**	**6.73**	**20.85**	**35.97**	**55.11**	**76.22**	**100.63**	**122.56**	**141.97**	**162.15**
		MB	1.18	**24.07**	**64.16**	**120.40**	**185.86**	**270.46**	**370.00**	**474.25**	**602.14**	**731.77**
	ZA	CS	0.26	1200+	1200+	1200+	1200+	1200+	1200+	1200+	1200+	1200+
		MB	**0.00**	–	–	–	–	–	–	–	–	–

DC=DC-SSAT, ZA=ZANDER, CS=CPU secs, MB=MB mem, 1200+=Timed out, M=Mem exhausted, 0.00=Rounding

Table 4: ZANDER's performance deteriorates significantly as the number of actions in the GOx problems increases.

might be possible to use the solution from the first of these to extend the time horizon of the overall solution/plan an arbitrary number of steps at very little incremental cost, producing a planner that is able to solve very large problems efficiently. We plan to implement this idea and test DC-SSAT against other probabilistic planning techniques. Also, in a future paper, we will show how DC-SSAT can be extended to partially observable probabilistic planning problems and to general SSAT problems. One difficulty here is that a VPA in subproblem Φ_i may contain variables that are also in subproblem $\Phi_j, j > i+1$; hence, checking VPA compatibility only between adjacent subproblems is no longer sufficient. One consequence is that the depth-first search that calculates $Pr[\Phi]$ must maintain a list of VPAs that contain variables in later subproblems so that compatibility between non-adjacent subproblems can be checked as needed.

There are other interesting directions for further work. In the case of general SSAT problems, can we characterize when the DC-SSAT approach will be beneficial? It will fail if all the existential variables are related and form a single equivalence class, but what is the impact of the connection topology in general? Variables that are far apart in the prefix, but share a clause, may induce the solver to explore a long, unproductive path. Assignment compatibility issues that were local in a COPP-SSAT problem become global in the general case. Walsh (2001) investigated the impact of the connection topologies of graphs associated with real-world search problems and related this to the notion of a *backbone* (variables that have the same value in all solutions). What role do these notions play in SSAT problems?

Finally, although QBF problems do not require a solver to consider all possible satisfying assignments, SSAT and QBF are related; any QBF instance can be solved by transforming it into an SSAT instance—replace the universal quantifiers with randomized quantifiers and check whether $Pr[\Phi] = 1.0$. We are exploring whether the DC-SSAT approach could form the basis for an efficient QBF solver that also produces the solution tree in the case of an affirmative answer.

References

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. Value elimination: Bayesian inference via backtracking search. In *UAI-2003*, 20–28.

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

Freudenthal, E., and Karamcheti, V. 2003. QTM: Trust management with quantified stochastic attributes. Technical Report TR2003-848, Courant Institute, NYU.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *UAI-99*, 279–288.

Littman, M. L.; Majercik, S. M.; and Pitassi, T. 2001. Stochastic Boolean satisfiability. *Journal of Automated Reasoning* 27(3):251–296.

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

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

Park, T. J., and Van Gelder, A. 1996. Partitioning methods for satisfiability testing on large formulas. In *CADE*, 748–762.

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

Walsh, T. 2001. Search on high degree graphs. In *IJCAI-01*, 266–274.