PLANNING UNDER UNCERTAINTY VIA STOCHASTIC SATISFIABILITY

by

Stephen Michael Majercik

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Michael L. Littman, Supervisor

_____
Donald W. Loveland

_____
Mark A. Peot

_____
Richard G. Palmer

_____
Henry A. Kautz

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

September 2000

ABSTRACT

(Computer Science)

PLANNING UNDER UNCERTAINTY VIA STOCHASTIC SATISFIABILITY

by

Stephen Michael Majercik

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Michael L. Littman, Supervisor

_____
Donald W. Loveland

_____
Mark A. Peot

_____
Richard G. Palmer

_____
Henry A. Kautz

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

September 2000

Abstract

I describe a new planning technique that efficiently solves probabilistic propositional planning problems by converting them into instances of stochastic satisfiability (SSAT) and solving these problems instead. This is the first work extending the planning-as-satisfiability paradigm to stochastic domains. I make fundamental contributions in two areas: the solution of SSAT problems and the solution of probabilistic planning problems.

I first describe three SSAT solvers: two exact solvers, based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, and one approximation technique. The first exact solver finds the optimal solution (a setting of variables in the SSAT problem that maximizes the probability of a satisfying assignment). The second applies bounding arguments to efficiently identify the first solution whose probability of satisfaction exceeds a prespecified threshold. I explore the application of several well-known DPLL heuristics to the solution of SSAT problems and empirically demonstrate that, while they improve performance significantly in randomly generated SSAT problems, in SSAT encodings of planning problems a simple heuristic based on temporal order dominates. The approximate solver uses a randomized approximation scheme (with or without hillclimbing) to rapidly identify a good solution to the SSAT problem. The epsilon-delta bound on the time required to evaluate SSAT solutions is polynomial in the size of the SSAT encoding *v.* exponential for the exact methods.

In the second half of the dissertation, I describe two probabilistic planners: MAXPLAN (for unobservable domains) and ZANDER (for partially observable domains). MAXPLAN converts a dynamic-belief-net representation of a noncontingent planning problem into an instance of E-MAJSAT, a restricted type of SSAT

problem in which *existentially quantified* variables precede *randomly quantified* variables. ZANDER solves general contingent planning problems by using a more general type of SSAT encoding in which existentially quantified and randomly quantified variables are interleaved. The conversion and solution algorithms exploit the structure of the domain, including factoring in the state representation and asymmetry in the distributions of propositions resulting from the application of actions.

ZANDER can solve arbitrary, goal-oriented, finite-horizon partially observable Markov decision processes (POMDPs). An empirical study comparing ZANDER's performance to that of three other leading probabilistic planners—a dynamic programming POMDP algorithm, MAHINUR, and SENSORY GRAPHPLAN (SGP)—shows that ZANDER's performance equals or betters that of these planners on a range of problems. ZANDER finds optimal contingent plans as much as two orders of magnitude faster, in spite of the fact that MAHINUR and SGP are specialized to more restricted classes of problems. In addition, experiments indicate that further significant efficiency gains will be possible, thus allowing ZANDER to scale up to larger problems.

Acknowledgements

This work was funded in part by NASA Ames Research Center through a Graduate Student Research Program Fellowship, and by the National Science Foundation through a Career Grant to Michael Littman (NSF grant IRI-9702576).

I would like to thank my advisor, Michael Littman, for many things. His arrival at Duke in the fall of 1996 made it possible for me to work in a fascinating area of Artificial Intelligence. Michael has been the kind of advisor they describe in articles on how to be a great advisor. He has been thoughtful, creative, energizing, encouraging, and supportive. His guidance has been invaluable, but he has always been willing to let me explore possibilities other than those he suggested. Michael has shown me, by his example, how to do good research and how to be a stimulating and caring teacher. He has always been concerned about what *I* needed, and has helped launch my career in computer science in many ways. Working with Michael has been a wonderful experience that I look forward to continuing.

I would also like to thank Tom Dean, for suggesting to Michael that GRAPHPLAN and SATPLAN were worth thinking about, in spite of the fact that they are deterministic planners. This dissertation is evidence that he was right.

Many thanks to Donald Loveland, Mark Peot, Henry Kautz, and Richard Palmer for serving on my committee. Their work has inspired me and their insights and comments have been invaluable.

I don't know where to begin to thank my wife, Faith Barnes. The list of ways in which she has made this dissertation possible would themselves require a chapter. Pulling up roots in Maine so that I could pursue a Ph.D. at Duke and enduring six years of Ph.D. widowhood are perhaps the most obvious things. But,

beyond that, I am grateful for her love and support—they have helped sustain me through times when the light at the end of the tunnel seemed no more reachable than a distant star.

My parents, Stephen and Eleanor, began contributing to my Ph.D. forty-five years ago and have never stopped. I am grateful for their love, their nurturing, and their encouragement. Their help "in residence" over the past six years and, especially, during the last few months, has been invaluable and helped keep Faith and me sane.

Special thanks to my father-in-law, Benjamin Barnes, for his unflagging interest in and support of what I was doing. I wish he could have been here to see the successful conclusion of my efforts.

Thanks to my son, Alexander, for providing a little extra motivation—I promised myself I would have my Ph.D. before he entered kindergarten. And, thanks to both Alexander and Charlotte, my daughter, for helping me keep my priorities straight, and for their screams of delight when I come home at night.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Planning—making a sequence of action choices to achieve a goal—has been an important area of artificial intelligence (AI) research since the field of AI began about fifty years ago. It promises to be an active area for at least that much longer. There are two main reasons for planning's prominence. First, the need to plan is pervasive; to a greater or lesser extent, *all* problems can be characterized as planning problems: how should one act (bring resources to bear) to change an existing state into a more desired state? The ability to act in a goal-directed fashion is critical to any notion of intelligent agency. Second, planning is an extremely hard problem. Deterministic STRIPS planning (arguably the "easiest" type of propositional planning that is still capable of expressing interesting problems) is PSPACE-complete [19]; probabilistic propositional planning in partially observable domains is undecidable [73].

Traditionally, the decision-making models that have been studied in AI planning admit no uncertainty: every aspect of the world that is relevant to the generation and execution of a plan is known to the agent in advance. This unrealistic assumption has been a major impediment to the practical use of AI planning techniques, and there has been a great deal of research in the past decade to create planning techniques that are capable of handling uncertainty in the environment (uncertain initial conditions, probabilistic effects of actions, and uncertain state estimation). One of the attractive features of AI planning is its ability, in some cases, to operate in large domains ($\sim 10^{20}$ states). One reason for this ability is that AI planning typically uses a problem representation that

allows significant features of the problem states and actions to be exploited by the solution method.

Planning is an important research area in other disciplines as well. Its centrality, in terms of human activities, and its difficulty have made planning a longstanding and important area of research in operations research (OR), psychology, and cognitive science. The planning work in OR is of particular interest. In contrast to traditional AI planning, OR planning addresses the need to deal with uncertainty (uncertain initial conditions, probabilistic effects of actions, uncertain state estimation). OR planning approaches, such as algorithms for Markov decision processes (MDPs) and partially observable Markov decision processes (POMDPs), are quite comfortable dealing with domains in which the agent is uncertain about the effects of its actions and even about what state it currently is in. Classical OR techniques, however, use an impoverished problem representation that does not capture relationships among states, and these techniques are capable of solving problems only in relatively small domains ($\sim 10^6$ states for exact MDP solution methods and many fewer for exact POMDP solution methods in typical domains).

This dissertation investigates the potential of merging the best characteristics of AI planning (large domains) and OR planning (stochastic domains) to produce a system that can reason efficiently about plans in complex, uncertain applications. The planners I have developed are rooted in the planning-as-satisfiability paradigm. In this paradigm, the planning problem is converted into a satisfiability problem and the efficient solution of the resulting satisfiability problem produces the required plan. This work is inspired in large measure by the success of SATPLAN, a similar planning technique for deterministic domains [59]. By encoding the planning problem as a Boolean satisfiability problem, and using stochastic

local search to solve the resulting satisfiability problem, SATPLAN can solve very hard deterministic planning problems as much as an order of magnitude faster than the next best planning system.

As I will discuss in more detail in subsequent chapters, there are significant problems in developing a probabilistic version of SATPLAN. Plans in a stochastic domain can be very complex. Unlike plans in a deterministic setting, optimal plans in a stochastic domain frequently require contingent branches that specify different actions depending on the stochastic outcome of the current action, or loops that repeat an action until a desired result is achieved. In addition, evaluating plans in stochastic domains is difficult. In the deterministic setting, plan evaluation can be accomplished by executing the plan and checking the single execution trace to see whether the final state is a goal state. In the stochastic setting, the uncertainty in the domain means that, in general, there will be multiple possible execution traces for a given plan, with some subset of these traces ending in a goal state. For this reason, plan evaluation requires the equivalent of checking each possible execution trace and summing the probability of each trace whose final state is a goal state.

The main contribution of my dissertation research is to show that the planning-as-satisfiability paradigm can be successfully extended to support contingent planning in partially observable stochastic domains. To my knowledge, the planners I have developed are the only existing planners that augment the planning-as-satisfiability paradigm to support stochastic domains. ZANDER (Section 7.3), the most advanced planner I developed, can solve arbitrary, goal-oriented, finite-horizon partially observable Markov decision processes (POMDPs). An empirical study comparing ZANDER's performance to that of three other leading probabilistic planners—a dynamic programming POMDP algorithm, MAHINUR, and

SENSORY GRAPHPLAN (SGP)—shows that ZANDER's performance equals or betters that of these three planners on a range of problems. ZANDER finds optimal contingent plans as much as two orders of magnitude faster, in spite of the fact that MAHINUR and SGP are specialized to more restricted classes of problems. I will describe specific contributions in the following guide to the remaining chapters of this dissertation.

Chapter 2:

This chapter reviews the planning research that is relevant to the work I have done. It traces the roots of my research back to the earliest days of AI planning and describes other planning research that has attacked problems similar to those my research has addressed.

Chapter 3:

My work is an extension of the planning-as-satisfiability paradigm, developed by Kautz and Selman [59]. I start this chapter by describing a representation for deterministic planning problems. I describe the satisfiability problem and show how deterministic planning problems can be encoded as satisfiability problems. Finally, I describe how SATPLAN [59] uses stochastic local search to solve the resulting satisfiability problem.

Chapter 4:

This chapter describes some planning complexity results that indicate what is needed in order to extend the planning-as-satisfiability paradigm to probabilistic planning.

Chapter 5:

In this chapter, I formally describe the stochastic satisfiability problem. I describe an algorithm for solving stochastic satisfiability problems and report results using this algorithm on randomly generated stochastic satisfiability problems. These tests compare the performance of six heuristics commonly used in satisfiability problems, and show empirically that four of them decrease the size of the search tree created during the solution of the problem given certain conditions.

I also describe an approximation algorithm I developed for stochastic satisfiability problems. The algorithm uses random sampling to bound the size of the plan search space, and stochastic local search to find the best plan in that reduced space. I report tests on random problems that indicate that this is a viable approximation approach.

Chapter 6:

In this chapter, I describe MAXPLAN, the first planner I developed. MAXPLAN converts a dynamic belief network representation of the planning problem to an E-MAJSAT problem, a type of stochastic satisfiability problem, and solves the E-MAJSAT problem instead. I report results comparing MAXPLAN's performance with other planning techniques. I also describe a caching technique I developed to overcome the prohibitive memory requirements of MAXPLAN.

Chapter 7:

MAXPLAN assumes complete unobservability and so can be used only when a good open-loop plan can be found. In this chapter, I describe ZANDER, a planner that extends the probabilistic-planning-as-stochastic-satisfiability paradigm to support contingent planning in stochastic domains with partial

observability. ZANDER encodes the contingent planning problem as a more general type of stochastic satisfiability problem. Rather than finding a single best assignment, as MAXPLAN does, ZANDER finds an assignment tree that specifies the optimal action choice for each possible sequence of observations. I report tests that show ZANDER to be competitive with other state-of-the-art planners on a range of problems drawn from the literature.

Chapter 8:

In this chapter, I discuss open problems and future work in five areas: improvements to ZANDER, extensions to ZANDER, approximation techniques, developing planners with less independence, and a unifying framework for planning and scheduling under uncertainty. I report some preliminary work in the areas of 1) developing alternate stochastic satisfiability encodings for probabilistic planning problems, and 2) applying the approximation algorithm described in Section 5.2.3 to planning problems.

Chapter 9:

I conclude with a brief summary of the major contributions of this dissertation.

Chapter 2

Approaches to Planning

The AI planning literature is vast with many threads: Planning as theorem proving, planning as state-space search, planning as plan-space search, planning as task-network search, and planning as constraint satisfaction are a few of the major threads. The OR planning literature is similarly broad.

I will provide a framework for my review of AI planning approaches by making two orthogonal distinctions: deterministic planning *v.* probabilistic planning, and classical planning *v.* constraint satisfaction planning. The most important threads in AI planning with regard to my work are those that deal with the deterministic and probabilistic variants of both planning as plan-space search and planning as constraint satisfaction (in particular, planning as satisfiability). The most relevant thread in OR planning is planning using the Markov decision process framework, so I will confine my discussion of OR planning approaches to planning in that framework.

2.1 Deterministic Planning

I will describe the deterministic planning problem formally in Section 3.1. Informally, a planning problem is characterized by;

- a finite set of states that the planning agent could find itself in,
- a finite set of operators, or actions, that transform states to states deterministically,

- a designated initial state, and
- a set of goal states.

A solution to the planning problem is a sequence of actions that transforms the initial state to one of the goal states.

2.1.1 Classical Planning

My review of the classical, deterministic planning literature begins with Green's theorem-proving problem solver [42] and the STRIPS planner [35]. Both of these are important in the chronology that leads to planning as satisfiability. In Green's problem solver—which could be used for planning, as demonstrated by its solution to the Monkeys and Bananas problem [42]—the planning problem is expressed in first-order logic. The solver then uses resolution theorem proving to answer the question of whether there is a state in which the goal—monkey has the bananas—is true. Extending Green's work proved difficult due to the inefficiency of first-order theorem provers at that time and the necessity of dealing with the *frame problem* in first-order logic. Briefly, the frame problem is the problem of needing to prove the persistence of conditions that are unchanged by a given action.

Both of these problems were solved (or sidestepped) by the introduction of the STRIPS planner [35], which did not use resolution theorem proving, and which dealt with the frame problem by specifying that all conditions not explicitly changed by an action remain unchanged. (But, as I will discuss in Section 2.1.2, the use of general reasoning systems for planning resurfaces quite successfully almost 25 years later in Kautz and Selman's SATPLAN planner.)

The STRIPS planner used a propositional state representation, in which the

characteristics of a state are described by a collection of Boolean variables, and an action representation that specifies, via sets of propositions, the necessary preconditions for an action to be applied and the effects of the action (which propositions become `True`—*add* effects—and which become `False`—*delete* effects—as a result of executing the action).

The STRIPS representation has been enormously influential, becoming more or less the standard planning problem representation, in spite of some serious limitations. As Allen [1] points out, the fact that the representation of the actions is separate from the representation of the environment means that:

- only one action can occur at any time,
- the environment changes only in response to planned actions, and
- actions are effectively instantaneous.

In addition, propositional planning has difficulty expressing maintenance goals (*e.g.* "maintain condition c from 4:00 pm to 5:30 pm") and dealing with partial goal achievement or tradeoffs among multiple goals.

Besides its influential problem representation, the STRIPS planner was one of the first planners to introduce the notion of planning as a search for a path through state space from the initial state to a goal state. This search for a path uses *means-ends analysis* either to build forward from the initial state (*progression* or *forward planning*) in search of a state that includes the goal conditions, or to build backward from the goal state (*regression* or *backward planning*) from the goal state in search of a state that includes the initial conditions.

This type of search, which locks in an action ordering as the path through the state space is constructed, can be too inflexible. Sometimes it is useful to be

able to specify that action A must be part of the plan, and perhaps even that it must precede action B, or follow action C, but that its exact position in the plan is unknown at the time it is added to the plan. The desire for this type of planning led to the notion of planning as search through the space of partial plans. Clearly, this is a more general notion of search, since state-space search can be seen as a type of restricted search through plan space (*i.e.* one in which the next plan in our search through plan space is the current plan with an action appended to the open end of the plan). Two examples of planners that search in plan space are SNLP [80] and UCPOP [93].

Searching in plan space is a powerful idea. As noted above, it is more flexible than searching through state space. But, in addition, by making the plan itself the object of our search, searching in plan space makes it possible to guide our search by directly operating on the objective—the plan. This may be more effective than trying to shape the plan indirectly by guiding the search through state space, and may make it easier to construct more complex plans.

This power brings with it two critical and difficult issues:

- How can plan space be searched efficiently?
- How can plans be evaluated efficiently?

The first issue—efficiency—needs to be dealt with no matter what kind of search one is engaged in. It is critical to avoid barren areas of the search space and focus the search as much as possible on areas containing, in this case, good plans, or partial plans that are relatively easy to extend to good plans.

The second issue is unique to searching in plan space. Although all search problems have an evaluation component (to decide how best to extend the search), the evaluation a planner must perform *after* a step is taken (to decide whether

a successful plan has been constructed) differs significantly in state-space search and plan-space search. After each step in a search through state space, it is only necessary to evaluate whether a goal state (forward planning) or the initial state (backward planning) has been reached, and this evaluation is merely a matter of comparing the status of the propositions in two states. After each step in a search through plan space, however, the current plan needs to be evaluated and, if it is not good enough, improved. The computational expense of this evaluation can vary drastically with the complexity of the plans being considered; it is critical that this process be as efficient as possible. I will return to these issues in Chapter 6.

2.1.2 Constraint Satisfaction Planning

In recent years, two planning methods based on constraint satisfaction—GRAPHPLAN and SATPLAN—have received a great deal of attention in the planning research community. In fact, in 1998 in a planning competition held at the Fourth International Conference on Artificial Intelligence Planning Systems, four out of the five planners competing were GRAPHPLAN- or SATPLAN-based planners. (Interestingly, in the next planning competition, held at the Fifth International Conference on Artificial Intelligence Planning and Scheduling in 2000, the dominant planners in a field of approximately fifteen entrants were temporal-logic-based planners.)

Both GRAPHPLAN and SATPLAN make use of the notion of search through plan space, albeit in a somewhat different fashion from the implicitly sequential notion of plan modification and evaluation described in the previous section. Both of these planners, in a sense, consider *all* plans up to a certain length *simultaneously* and attempt to extract a successful plan from this collection.

GRAPHPLAN [10] works by creating a *planning graph* that interleaves layers of nodes representing the status of propositions at a time step with layers of nodes representing possible actions at a time step. Edges in this directed, leveled graph connect actions to their preconditions and their add and delete effects, thus indicating all feasible actions at each time step and their impact on the domain propositions. GRAPHPLAN operates by constructing a planning graph forward from the initial conditions until a layer of propositions appears that contains all the goal propositions. The planner then searches for a plan using backward chaining; if none is found it extends the graph another time step and the search is repeated. The key element of GRAPHPLAN is a scheme for efficiently identifying and propagating pairwise inconsistencies (*e.g.* two actions that cannot be executed at the same time). GRAPHPLAN outperforms UCPOP on several natural and artificial planning problems [10]; it remains one of the best current planners and research on this paradigm is quite active (see Section 2.2.2).

SATPLAN [58, 59] works by first converting the planning problem to a propositional satisfiability problem and then using stochastic local search to solve the resulting satisfiability problem.[1] Kautz and Selman [59], in an early paper describing SATPLAN, argue that the planning community, in rejecting general reasoning systems in favor of specialized planning algorithms, learned the wrong lesson from the failure of Green's theorem-proving problem solver. They argue that the lesson to be learned was not that general reasoning systems are inappropriate for planning but that first-order deductive theorem-proving does not scale well. In contrast, propositional satisfiability testing has great potential as a tool for reasoning about plans.

[1]Deterministic STRIPS planning is PSPACE-complete [19]; SATPLAN converts this to an NP-complete problem by considering only polynomial-length plans.

Briefly, SATPLAN converts a deterministic planning problem to a Boolean satisfiability problem by constructing a CNF Boolean formula that has the property that any satisfying assignment to the variables in the formula—any *model*—corresponds to a plan that achieves the goal. The satisfiability of the resulting CNF formula is determined using WALKSAT, a generic satisfiability algorithm based on stochastic local search. It is worth noting here that although SATPLAN uses stochastic local search, other satisfiability testing algorithms exist. The original Davis-Putnam procedure for satisfiability testing [28] uses resolution as a key algorithmic component. Resolution was later replaced by *variable splitting* [27], and this latter procedure has completely overshadowed the earlier version (unjustifiably so, some have argued [30]). Although stochastic local search (used by SATPLAN) generally outperforms these systematic satisfiability testers by an order of magnitude or more on hard *random* satisfiability problems, there is some evidence that the systematic testers are competitive with stochastic local search on more structured, real-world planning problems [7]. I use a modified version of the Davis-Putnam-Logemann-Loveland satisfiability tester [27] in my planners (Section 6.4).

Planning as satisfiability has been an active area of research. Researchers have looked at the issues that arise in connection with efficient conversion of planning problems to satisfiability problems [56, 34], improving systematic satisfiability testers [7, 66], understanding and improving stochastic local search [100, 81, 57], accelerating the search for a plan by including domain-specific knowledge [61], and incorporating the various constraint satisfaction planning techniques in a single planning system [60].

2.2 Probabilistic Planning

Like a deterministic planning problem, a probabilistic planning problem is specified by a finite set of states, a finite set of actions, an initial state, and a set of goal states. In a probabilistic domain, however, actions transform states to states probabilistically; for a given state and action, there is a probability distribution over possible next states. The solution to a probabilistic planning problem is an action selection mechanism for the planning domain that reaches a goal state with sufficiently high probability. Note that probability of success is not the only objective that makes sense to consider. Examples of other possible objectives include:

- minimizing the length or size of the plan, or
- maximizing the expected utility achieved by the plan (if there is a *utility function* that assigns a numerical value to each component of the goal, thus providing a quantitative measure of the importance, or *utility*, of each goal component).

The defining characteristic of probabilistic planning is that the actions are probabilistic; the outcome of an action in a given state is a probability distribution over possible next states. There is another type of nondeterministic planning that is relevant in this review, however. It is possible to frame planning problems using *nonprobabilistic* actions.[2] A nonprobabilistic action can have multiple possible outcomes that depend only on the state in which the action is executed.

[2]Such actions have historically been called *conditional* actions [39, 40, 94]. In my taxonomy of planning under uncertainty, however, I wish to make a distinction between the type of planning and the type of actions used, so I will will use the term *nonprobabilistic action* to avoid confusion.

The effect of the action is deterministic given the state in which it is executed, but the agent may not know *a priori* the state in which it will be executing the action and, hence, its effect. Thus, the uncertainty is represented as a list of possible state/outcome pairs, rather than as a probability distribution over possible outcomes.

A simple example will clarify this distinction between probabilistic actions and nonprobabilistic actions. A probabilistic action move(a,b,c) in a blocks-world domain (*i.e.* move block a off of block b onto block c) might specify that the action is successful with probability 0.85, that block a ends up on the table with probability 0.10, and that nothing happens with probability 0.05. A nonprobabilistic version of the same action might specify that if the gripper is functioning and dry, the action will succeed, if the gripper is functioning, but wet, block a will end up on the table, and if the gripper is not functioning, nothing will happen.

The type of planning an agent engages in is, in this sense, a function of the agent's knowledge about the domain. A probability distribution over possible outcomes of an action may, in some cases, be a substitute for better domain knowledge. In the blocks world example, the agent may not know that the move action fails sometimes because the gripper is wet. But experience may allow the agent to estimate a probability distribution over outcomes of that action. Or it may be the case, to extend this example further, that the agent knows that when the gripper is wet, the action *usually* fails, but that with probability 0.05 it succeeds. If the agent does not know *why* the action sometimes succeeds, the agent may still be able to attach a probability distribution to the execution of the action, and plan using that probability distribution.

I will also make a distinction between *conditional planning* and *contingent planning*. In conditional planning, the effects, but not the execution, of actions

are contingent on the outcomes of previous actions. In contingent planning, both the effects and execution of actions are contingent on the outcomes of previous actions.[3] Thus, in contingent planning, the agent can make observations and construct a branching plan in which actions are made contingent on these observations. Without the ability to observe its environment and condition its actions accordingly, an agent can only execute a *straightline plan*, a simple noncontingent sequence of actions, and hope for the best.

These two distinctions (conditional planning *v.* contingent planning and nonprobabilistic actions *v.* probabilistic actions) produce the following taxonomy of planners:

1. Conditional planning with nonprobabilistic actions: These types of planners engage in *conformant* planning: producing a straightline plan that is guaranteed to succeed no matter what conditions are encountered. Example: CONFORMANT GRAPHPLAN [103].

2. Contingent planning with nonprobabilistic actions: Sensing allows this type of planner to produce a contingent plan, but the lack of probabilistic actions means that the planner must look for a plan that will succeed under all circumstances. Examples: CNLP [94], PLINTH [40], and SENSORY GRAPHPLAN [113].

3. Conditional planning with probabilistic actions: As in Case 1, these planners engage in conformant planning, but the probabilities attached to action outcomes allow the planner to specify the straightline plan that has the

[3]Note that the term *conditional* has been used in different ways in the literature. Plans in which the execution of actions depends on the outcomes of earlier actions were originally called conditional plans [111]. Some researchers [32] suggested calling such plans contingent plans, reserving the term conditional for plans in which only the effects of actions are contingent on the outcomes of earlier actions, and this terminology has been generally adopted.

highest probability of succeeding, even if that probability is less than 1.0. Example: BURIDAN [65] and UDTPOP [95]. The first planner I developed, MAXPLAN (Chapter 6), falls into this category.

4. Contingent planning with probabilistic actions: As in Case 2, sensing allows planners in this category to produce contingent plans. As in Case 3, probabilistic actions allow the planner to specify the plan that has the highest probability of succeeding. Examples: C-BURIDAN [32], DTPOP [95], MAHINUR [88, 89], and PGRAPHPLAN/TGRAPHPLAN [11]. ZANDER, the contingent planner I developed (Chapter 7), falls into this category.

I will discuss these planners more fully in the sections that follow.

2.2.1 Classical Planning

CNLP and PLINTH are contingent planners that plan with nonprobabilistic actions. CNLP [94], a complete, contingent nonlinear (partial-order) planner based on SNLP [80], differentiates among plan branches by establishing mutually exclusive *contexts* for actions. Briefly, whenever the planner encounters a possible context in which an action, chosen for its effects in a different context, cannot be executed, the planner creates a branch and attempts to construct a separate plan for that context. PLINTH [40] is a linear conditional planner that operates like a conventional linear planner, but with nonprobabilistic actions. It creates a contingent plan by repeatedly selecting an unrealized goal and nondeterministically choosing an action to achieve that goal while respecting existing constraints.

BURIDAN [65] is a conditional planner that uses probabilistic actions. BURIDAN uses a propositional state representation and tree-based, probabilistic STRIPS operators to extend partial-order planning to stochastic domains. Its represen-

tation can express arbitrary MDPs (see Section 2.2.3), sometimes logarithmically more compactly than traditional OR representations. BURIDAN searches for a plan whose probability of success meets or exceeds some prespecified threshold. The plans found by BURIDAN are partially ordered sequences of actions that, in execution, become simple sequences of actions (like the plans found by classical deterministic planners).

The basic BURIDAN algorithm alternates between plan assessment (is the plan good enough?) and plan refinement (what can be done to improve it?), and its performance emphasizes the importance of the two issues raised in Section 2.1.1: efficient plan-space searching and efficient plan evaluation. As I will show in Section 6.6, BURIDAN performs relatively poorly even on some very simple domains. With regard to plan evaluation, BURIDAN has four different plan-evaluation methods that can yield a wide range of performance results even on simple problems. For example, on the BOMB-IN-TOILET problem, BURIDAN's performance [65], varies from a low of 6.9 CPU seconds to a high of 6736.0 CPU seconds depending on which of four plan-evaluation methods it uses. Since there exist problems for which each method is best and since it is not possible, in general, to determine beforehand which method will provide the best performance, plan evaluation remains problematic for BURIDAN.

C-BURIDAN [32] augments BURIDAN's expressivity to encompass contingent plans (although it is still not powerful enough to express policies, or universal plans; see Section 2.2.3). As such, C-BURIDAN is trying to solve a somewhat different problem; its actions can provide information to the agent, but this information might be noisy or incomplete, as in a POMDP (see Section 2.2.3).

UDTPOP [95] is a sound and complete, decision-theoretic, partial-order, conditional planner that uses probabilistic actions. UDTPOP finds the plan that

maximizes the value of a multi-attribute value function, which is the sum of a reward value function and step cost functions. Thus, UDTPOP can reason about the relative utility of plan objectives. UDTPOP uses a single-support causal link mechanism in contrast to BURIDAN, which can instantiate multiple causal links to support a goal condition.

DTPOP [95] is an extension of UDTPOP that supports contingent planning with probabilistic actions and is designed to address three problems with C-BURIDAN described by Peot [95]: an inability to determine the relevance of observations, too-early commitment to an execution policy, and inefficient and unnecessary branch replication.

MAHINUR [88, 89], a contingent, partial-order planner that uses probabilistic actions, was designed primarily to address the following weakness in C-BURIDAN described by Onder and Pollack [88]: C-BURIDAN does not reason about whether the branches it constructs are actually worth constructing; it automatically creates branches to resolve otherwise unresolvable conflicts. MAHINUR combines BURIDAN's probabilistic action representation and system for managing these actions with a CNLP-style approach to handling contingencies. The novel feature of MAHINUR is that it identifies those contingencies whose failure would have the greatest negative impact on the plan's success and focuses its planning efforts on generating plan branches to deal with those contingencies. This selectivity in adding branches to the plan boosts MAHINUR's speed considerably relative to the partial-order planners described above, but Onder and Pollack [88] identify several domain assumptions (including a type of subgoal decomposability) that underlie the design of MAHINUR, and there are no guarantees on the correctness of MAHINUR for domains in which these assumptions are violated.

2.2.2 Constraint Satisfaction Planning

There has been significant recent work on augmenting GRAPHPLAN to handle stochastic domains. Some researchers have extended GRAPHPLAN to handle actions with conditional effects [3]. CONFORMANT GRAPHPLAN [103] deals with uncertainty in the initial conditions and in the outcome of actions by attempting to construct a noncontingent plan that will succeed in all cases. Since the resulting plan is expected to succeed under all possible circumstances, CONFORMANT GRAPHPLAN has no sensing actions. PGRAPHPLAN [11] employs forward search through the planning graph to find the contingent plan with the highest expected utility in an MDP-style environment (the state of the world is known, but actions are probabilistic; see Section 2.2.3 for MDP details). SENSORY GRAPHPLAN (SGP) [113], unlike CONFORMANT GRAPHPLAN, constructs plans with sensing actions that gather information to be used later in distinguishing between different plan branches. However, SGP has not been extended to handle probabilistic actions and imperfect observations, so it is only applicable to a subset of partially observable planning problems.

To my knowledge, there has been no work done outside of this dissertation to augment SATPLAN to handle stochastic domains.

2.2.3 Operations Research Planning

Much of the planning work in OR takes place in the framework of the Markov decision process (MDP) model, an intuitively appealing model capable of representing AI planning problems. In the MDP model, initially developed by Bellman [8] and Howard [51], an agent with a finite set of actions at its disposal is embedded in an environment capable of being in a finite number of states. Each action in each

possible state:

- yields some (possibly negative) reward, and
- probabilistically changes the state of the environment according to the MDP's state transition model.

The objective of the agent is to act so as to maximize its success as measured by the expected cumulative reward received.

Although the MDP model captures some essential components of the planning problem, it contains a number of simplifications (shared with STRIPS-style planning) that make it tractable: the action set and state space are finite, time is discrete, only one action is taken at a time, the effects of actions are instantaneous, the environment does not change except as a result of the agent's actions, rewards are not time dependent, and the environment is completely observable; the agent sees everything and can use this information in deciding what action to take next. In addition, MDPs make the Markov assumption that state transitions and rewards depend only on the current state and action, and not on time, or on past states or actions. Viewed from a slightly different perspective, the Markov assumption says that knowledge of the current state and action is sufficient; the agent can do *no better* even if it remembers previous states and actions. This is a powerful constraint that allows a solution to be constructed by calculating a *value function*, a mapping from states to values that measures how "good" it is for an agent to be in each possible state of the MDP *independently of how the agent arrived at that state.*

Value functions form the backbone of virtually all MDP algorithms. Their widespread use is largely the result of three factors:

- Value functions form the basis of a universal plan, or *policy*; since each state has a value, and the state transition table is known to the agent, there is an optimal choice no matter what state the agent is in. The agent selects the action that maximizes the probability-weighted average of the values of the possible resulting states.

- There is always an *optimal* value function the agent can use to make the best choice.

- An approximately correct value function at time t can be used to construct a more accurate value function at time $t + 1$.

There are several standard methods for solving MDPs exactly; dynamic programming techniques such as value iteration [8], policy iteration [51], and modified policy iteration [96], and linear programming [31]. These techniques, however, are computationally feasible only for relatively small MDPs ($\sim 10^5$ states). Additional techniques have been developed to try to overcome this limitation. *Envelope methods* [29, 45, 106] try to solve a smaller sub-MDP defined over those states that the agent is most likely to encounter. *Priority queue methods* [83] concentrate on iteratively improving the value function of states whose value, as assessed by the current value function, seems likely to change significantly. *Reinforcement learning approaches* [6, 55, 104, 105] stochastically generate trajectories through the state space and concentrate on producing an accurate value function for those states the agent is likely to visit.

None of these methods have strong guarantees regarding running time or approximation error, but they can solve somewhat larger MDPs ($\sim 10^6$ states). One technique that has been used to successfully solve realistically large MDPs is *value function approximation*. In this technique, the table of states and values typically

used to represent the value function is replaced with a parameterized function approximator. There are various flavors of this technique, largely depending on the type of function approximator used, but the main idea is that instead of maintaining an exponentially large table of state values, one uses something like a neural network to learn an approximate value function.

In fact, the reinforcement learning approach along with value function approximation has led to an impressive number of successes: the world's best backgammon-playing program (and one of the world's best backgammon players, including human players) [107, 108], a controller for a bank of elevators [25], a system for making cellular-phone-channel assignments [101], and a job-shop scheduler for space-shuttle payload processing [118]. This approach does not always succeed, however, and although some work has been done to establish its theoretical underpinnings [9, 109, 110, 4, 41], its applicability is still not clearly understood.

Recent work on MDP algorithms has focussed on using insights from classical AI planning to help solve larger MDPs faster; in particular, exploiting the structure of compact, *factored* representations of the problem [15], and using approximation schemes [14]. In the former work, a technique called *structured policy iteration* (SPI) was developed that constructs optimal policies without needing to explicitly enumerate the state space. SPI avoids explicit enumeration by exploiting the structure (the regularities and independencies) in a factored representation of the planning domain that uses two-time-slice Bayes nets (described in Section 6.1).

Koller and Parr [63] have presented evidence that the value function of a factored MDP can often be well approximated using a factored value function; *i.e.* a linear combination of restricted basis functions, each of which refers to only a small subset of variables in the MDP. They show that this value function

approximation technique can be used as a subroutine in a policy iteration process to solve factored MDPs [64].

Partially observable Markov decision processes (POMDPs) generalize MDPs to the case in which there is uncertainty about the current state as well as about the effects of actions. The POMDP model is very general; it can account for tradeoffs among the following:

- taking uncertain actions with differing costs to advance toward one or more possibly competing goals, and
- taking actions to gain partial knowledge about one's surroundings.

A POMDP [54, 72, 82, 114] is an MDP augmented by a finite set of observations the agent can experience and an *observation function* that maps each action and resulting state to a probability distribution over the possible observations. Thus, the agent's knowledge of the actual state of its environment is the result of potentially unreliable observations. The resulting uncertainty regarding the actual current state is frequently handled by maintaining a *belief state*, which is a probability distribution over the underlying states of the process and summarizes all the information contained in past actions and observations and the current observation. In effect, one can think of the POMDP as an MDP over these belief states.

In the POMDP model, actions transform belief states to belief states using a belief-state transition function that is based on the state transition function of the underlying MDP and the observation probabilities of the POMDP. Similarly, rewards are based on the current belief state and action.

The value function is again critical to POMDP solution methods. In the case of POMDPs, the value function is usually represented as a piecewise-linear, convex

function, and the solution techniques generally use dynamic programming to iteratively transform an approximate value function into a more accurate value function.

Active areas of POMDP research include the use of heuristics [18, 44, 112], factored, propositional representations [16, 32], reachability analysis [12], and approximation [21, 49, 69, 67, 92, 115, 117].

2.3 Summary

During the past decade, AI researchers have recognized that in order for AI techniques to be used in the real world, they must be able, in many cases, to deal with the pervasive uncertainty found there. Thus, much recent AI research—in planning and other areas—has focused on dealing with uncertainty, and the demarcation between AI planning research and OR planning research has become less and less clear. Many AI researchers have adopted the Markov decision process formalism and, in fact, much of the current AI research on probabilistic planning is taking place in that framework.

My research has established an alternate framework for planning with probabilities based on stochastic satisfiability. In the next two chapters (Chapters 3 and 4), I will describe the planning-as-satisfiability paradigm and discuss complexity issues that suggest what is necessary to extend the paradigm to probabilistic planning. In the remaining chapters, I will describe the planners I have developed based on this extension, and report results indicating that this is a promising alternative approach to attacking problems expressed in the MDP and POMDP framework.

Chapter 3

Deterministic Planning as Satisfiability

I briefly described the operation of SATPLAN in Section 2.1.2. Since my work is an extension of this planning-as-satisfiability paradigm, I will describe SATPLAN in more detail in this chapter. SATPLAN operates by converting a deterministic planning problem to an instance of SAT and solving the SAT problem instead (Figure 3.1). In the following sections, I will describe a representation for deterministic planning problems, provide a formal definition for the satisfiability problem, show how deterministic planning problems can be encoded as SAT problems, and describe how SATPLAN solves the SAT encoding of a planning problem.

3.1 Representing Deterministic Planning Problems

A planning domain $M = \langle \mathcal{S}, s_0, \mathcal{A}, \mathcal{G} \rangle$ is characterized by a finite set of states $\mathcal{S}$, an initial state $s_0 \in \mathcal{S}$, a finite set of operators or actions $\mathcal{A}$, and a set of goal states $\mathcal{G} \subseteq \mathcal{S}$. The application of an action a in a state s results in a deterministic transition to a new state s'. The objective is to choose actions, one after another, to move from the initial state s_0 to one of the goal states.

The STRIPS representation [35] of M, which I will describe informally, uses a propositional state representation; a state is described by an assignment to a set of Boolean variables. Actions are specified by three sets of propositions:

1. The *preconditions* set specifies what propositions need to be `True` for the action to be executed.

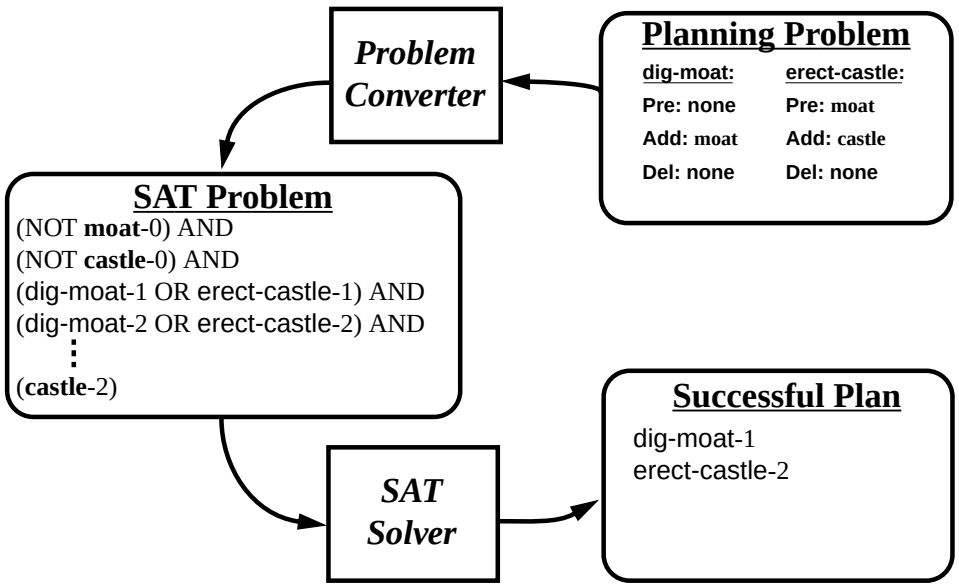


Figure 3.1: The SATPLAN approach converts a planning problem to a SAT instance and solves that problem instead.

2. The *add effects* set specifies those propositions that become `True` as a result of executing the action, and

3. The *delete effects* set specifies those propositions that become `False` as a result of executing the action.

An example will help flesh out this brief description. The STRIPS representation for a simple problem is shown in Figure 3.2. In this problem, the agent desires to build a sand castle in a certain location on the beach. Unfortunately, that location is subject to occasional waves that make it difficult to construct the castle. Fortunately, the agent has a shovel and so has the capability of digging a moat around the construction area to protect the castle. In this simple problem, there are two action: dig-moat and erect-castle. The dig-moat action has no preconditions; the action can always be executed. It has a single add effect—**moat** becomes `True`—and no delete effects. The erect-castle action has a precondition;

State Propositions: **moat**, **castle**

Actions: dig-moat, erect-castle

dig-moat	erect-castle
• Preconditions: none	• Preconditions: **moat**
• Add: **moat**	• Add: **castle**
• Delete: none	• Delete: none

Figure 3.2: The STRIPS representation for the deterministic SAND-CASTLE-67 problem.

there must be a **moat** in order to execute this action. Again, there is a single add effect—**castle** becomes `True`—and no delete effects.

3.2 Deterministic Satisfiability

Informally, a deterministic satisfiability (SAT) problem asks whether a given Boolean formula has a satisfying assignment; that is, is there an assignment of truth values to the variables used in the formula such that the formula evaluates to `True`. SAT is a fundamental problem in computer science. It was the first NP-complete problem and many important, practical problems in areas such as planning and scheduling, network design, and data storage and retrieval (to name just a few) can be expressed as SAT problems [38]. As such, SAT is a very well-studied problem, both from a theoretical point of view (*e.g.* how does the solution difficulty of random SAT problems vary as one varies the parameters of the problem?) as well as a practical point of view (*e.g.* how can one solve SAT problems efficiently?).

Formally, let $\mathbf{x} = \langle \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n} \rangle$ be a collection of n Boolean variables, and $\phi(\mathbf{x})$ be a Boolean formula on these variables in conjunctive normal form (CNF) with m clauses. Each clause is a disjunction of *literals*; a literal is a variable

or its negation. Thus, ϕ evaluates to `True` if and only if there is at least one literal with the value `True` in every clause. (Note: I will sometimes use 1/0 to denote `True`/`False`.) An *assignment* is a mapping from $\mathbf{x}$ to the set {`True`, `False`}. An assignment A is *satisfying*, and $\phi(\mathbf{x})$ is said to be *satisfied*, if $\phi(\mathbf{x})$ evaluates to `True` under the mapping A. This can be expressed using existential quantifiers and, anticipating the notation necessary for stochastic satisfiability, the expectation of formula satisfaction:

$$\exists x_1, \ldots, \exists x_n (E[\phi(\mathbf{x}) \leftrightarrow \texttt{True}] = \mathbf{1.0})$$

In words, this asks whether there exist values for all the variables such that the probability of the formula evaluating to `True` is certain. Note that I am using equivalence ($\leftrightarrow$ `True`) to denote the event of the formula evaluating to `True`.

3.3 Encoding Deterministic Planning Problems as SAT Problems

The generality of propositional satisfiability makes it possible to encode deterministic planning problems in a number of different ways; many different approaches to planning can be converted to propositional satisfiability. For example, both state-space planning and plan-space (causal) planning can be used as a basis for satisfiability encodings [56, 78]. The advantages of this generality are clear, but there is also a disadvantage. The multiplicity of possible SAT encodings for a particular problem and the absence of a principled way of selecting the best encoding make it difficult to develop a system that operates as efficiently as possible on a broad range of planning problems. In fact, one of the current challenges in the planning-as-satisfiability paradigm is to automate the process of producing

the most efficient SAT encoding of a planning problem [34].

To provide a sense of what a SAT encoding of a planning problem looks like, I will describe one possible SAT encodings—the linear encoding with classical frame axioms [56]—for the 2-step deterministic SAND-CASTLE-67 problem described in Section 3.1 above. In this type of SAT encoding, satisfiability is made equivalent to goal achievement by enforcing the following conditions:

- the initial conditions and goal conditions hold at the appropriate times (note that the initial state is completely specified whereas the goal state may be only partially specified),
- exactly one action is taken at each time step,
- if an action holds at time t, its preconditions hold at time $t-1$, its add effects hold at time t, and the negation of each of its delete effects holds at time t, and
- if an action does not affect a state variable, then that state variable remains unchanged when that action is executed (classical frame conditions).

There are five groups of clauses in this encoding (Figure 3.3). The first two groups of clauses enforce the initial conditions and goal conditions. The initial conditions are that there is no moat and no castle. To enforce the first condition, the negation of **moat**-0, the variable indicating the status of the moat at time step 0, appears by itself in a clause. Since all clauses must be satisfied, this ensures that this initial condition is honored. To enforce the second initial condition, the negation of **castle**-0 appears alone in a clause. The goal condition, having a castle at time step 2, is enforced by placing **castle**-2 in a clause by itself.

The third group of clauses enforces the restriction that exactly one action occurs at each time step. This is accomplished at each time step with a disjunction over all actions and binary disjunctions over all possible pairs of negated actions.

The fourth group of clauses enforces action preconditions and effects. A simple example will clarify the construction of these clauses. The dig-moat action at time step 1 implies its add effect **moat** at time step 1:

$$(\text{dig-moat-1} \rightarrow \textbf{moat}\text{-1})$$

Negating the antecedent and changing the implication to a disjunction yields clause 8:

$$(\overline{\text{dig-moat-1}} \vee \textbf{moat}\text{-1})$$

The fifth group of clauses enforces the frame conditions. For example, executing the dig-moat action at time step 1 has no impact on the status of the **castle** variable at time step 1. This generates two implications:

$$(\text{dig-moat-1} \wedge \overline{\textbf{castle}\text{-0}} \rightarrow \overline{\textbf{castle}\text{-1}})$$

$$(\text{dig-moat-1} \wedge \textbf{castle}\text{-0} \rightarrow \textbf{castle}\text{-1})$$

Again, negating the antecedent and replacing the implication with a disjunction produces clauses 14 and 15:

$$(\overline{\text{dig-moat-1}} \vee \textbf{castle}\text{-0} \vee \overline{\textbf{castle}\text{-1}})$$

$$(\overline{\text{dig-moat-1}} \vee \overline{\textbf{castle}\text{-0}} \vee \textbf{castle}\text{-1})$$

The encoding of this simple problem also provides a sense of how satisfying assignments are equivalent to successful plans. It is particularly easy to see how

Initial Conditions:

1. $(\overline{\textbf{moat-0}}) \wedge$
2. $(\overline{\textbf{castle-0}}) \wedge$

Goal Conditions:

3. $(\textbf{castle-2}) \wedge$

Exactly One Action Per Time Step:

4. $(\text{dig-moat-}\mathbf{1} \vee \text{erect-castle-}\mathbf{1}) \wedge$
5. $(\overline{\text{dig-moat-}\mathbf{1}} \vee \overline{\text{erect-castle-}\mathbf{1}}) \wedge$
6. $(\text{dig-moat-}\mathbf{2} \vee \text{erect-castle-}\mathbf{2}) \wedge$
7. $(\overline{\text{dig-moat-}\mathbf{2}} \vee \overline{\text{erect-castle-}\mathbf{2}}) \wedge$

Action Effects:

8. $(\overline{\text{dig-moat-}\mathbf{1}} \vee \textbf{moat-1}) \wedge$
9. $(\overline{\text{dig-moat-}\mathbf{2}} \vee \textbf{moat-2}) \wedge$
10. $(\overline{\text{erect-castle-}\mathbf{1}} \vee \textbf{moat-0}) \wedge$
11. $(\overline{\text{erect-castle-}\mathbf{2}} \vee \textbf{moat-1}) \wedge$
12. $(\overline{\text{erect-castle-}\mathbf{1}} \vee \textbf{castle-1}) \wedge$
13. $(\overline{\text{erect-castle-}\mathbf{2}} \vee \textbf{castle-2}) \wedge$

Frame Axioms:

14. $(\overline{\text{dig-moat-}\mathbf{1}} \vee \textbf{castle-0} \vee \overline{\textbf{castle-1}}) \wedge$
15. $(\overline{\text{dig-moat-}\mathbf{1}} \vee \overline{\textbf{castle-0}} \vee \textbf{castle-1}) \wedge$
16. $(\overline{\text{dig-moat-}\mathbf{2}} \vee \textbf{castle-1} \vee \overline{\textbf{castle-2}}) \wedge$
17. $(\overline{\text{dig-moat-}\mathbf{2}} \vee \overline{\textbf{castle-1}} \vee \textbf{castle-2}) \wedge$
18. $(\overline{\text{erect-castle-}\mathbf{1}} \vee \textbf{moat-0} \vee \overline{\textbf{moat-1}}) \wedge$
19. $(\overline{\text{erect-castle-}\mathbf{1}} \vee \overline{\textbf{moat-0}} \vee \textbf{moat-1}) \wedge$
20. $(\overline{\text{erect-castle-}\mathbf{2}} \vee \textbf{moat-1} \vee \overline{\textbf{moat-2}}) \wedge$
21. $(\overline{\text{erect-castle-}\mathbf{2}} \vee \overline{\textbf{moat-1}} \vee \textbf{moat-2})$

Figure 3.3: The CNF formula for a 2-step deterministic SAND-CASTLE-67 plan.

unsuccessful plans will lead to unsatisfying assignments. Clauses 4–7 ensure that, in any satisfying assignment, all the action variables will have an assigned value. Consider the two plans that begin with the action erect-castle. This means that erect-castle-1 will be `True`. This, along with clause 10, means that **moat**-0 must be `True`, which will make it impossible to satisfy the initial condition in clause 1. Of the two remaining plans, consider the plan dig-moat, dig-moat. This means that both dig-moat-1 and dig-moat-2 will be `True`. clause 3 forces **castle**-2 to be `True` and this, along with dig-moat-2 = `True` forces **castle**-1 to be `True` (clause 16). But, since dig-moat-1 is `True`, the fact that **castle**-1 must be `True` forces **castle**-0 to be `True` (clause 14). And, this makes it impossible to satisfy the initial condition in clause 2.

The plan dig-moat, erect-castle is the only possible successful plan and the only plan that allows a satisfying assignment: **moat**-1, **moat**-2, **castle**-2, dig-moat-1, and erect-castle-2 are all `True`, while **moat**-0, **castle**-0, **castle**-1, dig-moat-2, and erect-castle-1 are all `False`.

3.4 Solving Deterministic Satisfiability Problems

The most straightforward technique for solving the SAT encoding of the planning problem is *systematic search* for a satisfying assignment. This can perhaps best be visualized by thinking of it as a search on an *assignment tree*. First, impose an arbitrary ordering on the variables. An assignment tree is a binary tree in which each node represents a variable and a partial assignment. The root node at level 0 represents the first variable in the ordering and the empty partial assignment. For node q at level d representing the dth variable v in the variable ordering

and partial assignment A, the left child of node q, q_l, represents the variable following v in the variable ordering and the partial assignment A extended by setting v to `True`. The right child of node q, q_r, represents the variable following v in the variable ordering and the partial assignment A extended by setting v to `False`. The 2^n nodes at level n represent all possible complete assignments to the n variables. A traversal of this tree, evaluating the Boolean formula given the full assignment at each leaf, will consider all possible assignments and, hence, is guaranteed to find a satisfying assignment if one exists. The full assignment tree is, of course, exponential in the number of variables, and practical considerations demand that a systematic solver search as little of this tree as possible. I will describe heuristics for this purpose later in this section.

Even using heuristics, however, systematic search is impractical for very large problems. SAT encodings of even moderately-sized planning problems can be very large (> 5000 variables), and for problems of this size a more practical approach is to use *stochastic local search*. SATPLAN, in fact, uses WALKSAT [100], a generic satisfiability algorithm based on stochastic local search. WALKSAT initially makes a random assignment to the variables in the 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. Otherwise, it randomly flips that variable that would unsatisfy the fewest clauses. The randomness of this flip (the assignment may stay the same) is intended to prevent the solver from getting trapped in a local maximum or oscillating between two partial assignments. 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. In addition, it cannot report that a satisfying assignment does not exist (although recent work [99] provides probability bounds on the likelihood of missing a satisfying assignment if one exists). WALKSAT, however, can solve satisfiability problems that are orders of magnitude larger than those the best systematic solvers can handle [100].

3.5 Summary and Discussion

I have described how deterministic planning problems can be efficiently solved by encoding them as SAT problems and using stochastic local search to solve the SAT problem. There are a number of advantages to this approach. The expressivity of Boolean satisfiability allows us to construct a very general planning framework. It is relatively straightforward to express planning problems in the framework of propositional satisfiability. In addition, this framework makes it easy to add additional constraints to the planning problem (such as domain-specific knowledge [61]) to improve the efficiency of the planner. Another advantage echoes the intuition behind reduced instruction set computers; we wish to translate planning problems into satisfiability problems for which we can develop highly optimized solution techniques using a small number of extremely efficient operations. Supporting this goal is the fact that satisfiability is a fundamental problem in computer science and, as such, has been studied intensively. Numerous heuristics and solution techniques have been developed to solve satisfiability problems as efficiently as possible.

There are disadvantages to this approach. Problems that can be compactly expressed in representations used by other planning techniques often suffer a significant blowup in size when encoded as Boolean satisfiability problems, de-

grading the planner's performance. As I noted above in Section 3.3, automatically producing maximally efficient plan encodings is a difficult unsolved problem. In addition, translating the planning problem into a satisfiability problem may obscure the structure of the problem, making it difficult to use one's knowledge of and intuition about the planning process to develop search control heuristics or prune plans. This issue has also been addressed; Kautz and Selman [61], for example, report impressive performance gains resulting from the incorporation of domain-specific heuristic axioms in the SAT encodings of deterministic planning problems.

In spite of these disadvantages, however, SATPLAN is one of the most successful deterministic planners that has been developed. A natural question, which I will address in the next chapter, is whether this paradigm can be extended to probabilistic planning domains.

Chapter 4

Extending the Planning-as-Satisfiability Paradigm

In this chapter, I review some complexity results that suggest what would be necessary to extend the planning-as-satisfiability paradigm to stochastic domains.

In its most general form, a plan is a *program* that takes as input observable aspects of the environment and produces actions as output. I will classify plans by their *size* (the number of internal states) and *horizon* (the number of actions produced *en route* to a goal state). The computational complexity of propositional planning varies with bounds on the plan size and plan horizon. In the deterministic case, for example, unbounded STRIPS planning is PSPACE-complete [19]. If we put a polynomial bound on the plan horizon [59], however, STRIPS planning becomes an NP-complete problem.

The complexity of probabilistic propositional planning varies in a similar fashion. If the plan size is unbounded and the plan horizon is infinite, the problem is EXP-complete, if completely observable [70], or, in the more general case, undecidable [73]. If plan size *or* plan horizon alone is bounded by a polynomial in the size of the representation of the problem, the problem is PSPACE-complete [70]. Contingent planning with polynomial bounds on the plan horizon falls into this class. Evaluating a probabilistic plan—calculating the probability that the given plan reaches a goal state—is PP-complete [70]. Finally, if we place bounds—polynomial in the size of the planning problem—on both plan size and plan horizon, the planning problem is $\mathrm{NP}^{\mathrm{PP}}$-complete [70].

Whereas the class NP can be thought of as the set of problems solvable by guessing the answer and checking it in polynomial time, the class NP^{PP} can be thought of as the set of problems solvable by guessing the answer and checking it using a probabilistic polynomial-time (PP) computation.[1] The class PP can be informally characterized as the set of problems in which one needs to *count* the number of answers that satisfy some conditions. PSPACE is the class of problems solvable using polynomial space.

To the extent that we take the planning problem to be one of constructing a good controller and executing it to solve the problem, polynomial bounds on plan size and plan horizon are reasonable. In some cases, it may not help to know whether a plan exists if that plan is intractable to express, requiring, say, exponential space (and exponential time) to write down. The polynomial bound on plan horizon is perhaps less defensible but nonetheless seems like a reasonable restriction. When a contingent plan is required (see Chapter 7), the polynomial restriction on plan size may be too severe to allow a good plan (one with a sufficiently high probability of reaching a goal state) to be found, but the polynomial bound on plan horizon is still necessary to keep the problem in a "reasonable" complexity class (PSPACE).

The success of SATPLAN encourages us to try a similar approach for probabilistic planning problems, but these complexity results make it clear that we cannot encode probabilistic planning problems as SAT problems. The relationship

[1] The class NP^{PP} is a particularly interesting class. It is likely that NP^{PP} characterizes many problems of interest in the area of uncertainty in artificial intelligence; the work described in this dissertation and earlier work [84] give initial evidence of this. A survey of relevant results is available elsewhere [70].

Complexity Class	Complete Satisfiability Problem	Planning Problem
PSPACE		Probabilistic planning with: polynomially bounded plan size **or** polynomially bounded plan horizon
NP^{PP}		Probabilistic planning with: polynomially bounded plan size **and** polynomially bounded plan horizon
PP		Probabilistic plan evaluation
NP	SAT	Deterministic planning with: polynomially bounded plan size **which is equivalent to** polynomially bounded plan horizon

Table 4.1: Probabilistic planning is in a higher complexity class than deterministic planning; this chart will be completed in Chapter 5.

among these classes can be summarized as follows:

$$\mathrm{NP} \subseteq \mathrm{PP} \subseteq \mathrm{NP}^{\mathrm{PP}} \subseteq \mathrm{PSPACE}.$$

Although it is not known whether these are proper subsets, it seems likely that complete problems in PSPACE really are more difficult than complete problems in NP. In any case, we currently cannot express an NP^{PP}-complete or PSPACE-complete problem as a compact instance of SAT; if we want to extend the planning-as-satisfiability paradigm to probabilistic planning, we will need a different type of satisfiability problem. Table 4.1 summarizes the situation.

To extend the planning-as-satisfiability paradigm, we need to fill in the empty boxes in this chart. We need satisfiability problems that are complete for the

complexity classes that contain the planning problems of interest. Stochastic satisfiability, which I will describe in the next chapter, satisfies this requirement for all these planning problems.

Chapter 5

Stochastic Satisfiability

Portions of this chapter have appeared in an earlier paper:
"Stochastic Boolean satisfiability" [71] with Littman and Pitassi.

Stochastic satisfiability (SSAT) is at the core of the probabilistic planning techniques I have developed; both MAXPLAN and ZANDER operate by converting the planning problem to an instance of stochastic satisfiability and solving that problem instead. In this chapter, I will describe the stochastic satisfiability problem, including two special cases of SSAT—MAJSAT and E-MAJSAT—that are of particular interest from the probabilistic planning perspective.

I will then present two algorithms for solving SSAT problems—an exact algorithm and an approximation technique. Since the SSAT problem is PSPACE-complete, one might expect solution techniques for the NP-complete problem SAT to be of little or no use in solving SSAT problems. In fact, in the same way that SSAT is an extension of SAT, the exact algorithm I describe is an extension of a solution technique developed for SAT. I will also present empirical evidence that heuristics developed for SAT problems are applicable to SSAT problems as well. The approximation technique I will describe also uses a technique that has been used successfully on SAT problems—stochastic local search—but combines this algorithmic idea with random sampling in a novel way. I will present some empirical results using this approximation algorithm on randomly generated SSAT problems.

5.1 Stochastic Satisfiability

I will describe the general stochastic satisfiability problem first. Certain restrictions on this general problem lead to two special cases of stochastic satisfiability—MAJSAT and E-MAJSAT—that are of special interest with respect to probabilistic planning. These three problems—SSAT, MAJSAT, and E-MAJSAT—are complete for different complexity classes and turn out to be just what is needed to fill in the empty boxes in Table 4.1.

5.1.1 SSAT

Recall the definition of satisfiability from Section 3.2. Given Boolean variables $\mathbf{x} = \langle \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n} \rangle$ and a CNF formula $\phi(\mathbf{x})$ constructed from these variables, the satisfiability problem asks

$$\exists x_1, \ldots, \exists x_n (E[\phi(\mathbf{x}) \leftrightarrow \texttt{True}] = \mathbf{1.0}) :$$

Do there exist values for $x_1, x_2, \ldots, x_n$ such that the probability of $\phi(\mathbf{x})$ evaluating to `True` is certain?

The key idea underlying stochastic satisfiability (SSAT) is the introduction of a *randomized quantifier*: $\mathbb{R}$. Randomized quantifiers introduce uncertainty into the question of whether there is a satisfying assignment. I will formalize this notion later in this section but, for now, a simple example will illustrate the operation of this quantifier. Suppose I have the following formula:

$$\exists x_1, \mathbb{R} y_2 (E[(x_1 \vee \overline{y}_2) \wedge (\overline{x}_1 \vee y_2) \leftrightarrow \texttt{True}] \geq 0.75). \tag{5.1}$$

This instance of SSAT asks whether a value for x_1 can be chosen such that *for random* values of y_2 (choose `True` or `False` with equal probability) the *expected*

probability of satisfying the indicated Boolean formula is at least 0.75. This extension of SAT was first explored by Papadimitriou [90].

There are two important points to be made here. First, the presence of randomized quantifiers means that obtaining a satisfying assignment is no longer completely under the control of the solver. In the above example, after the solver has chosen a value for the existentially quantified variable x_1, the value of the randomly quantified variable y_2 will be chosen by flipping a fair coin. Thus, there is a certain *probability* that the choice of a value for x_1 will lead to a satisfied formula. If the solver sets x_1 to `True`, then there is a 0.5 probability that the formula will be satisfied (if the coin flip for y_2 comes up `True`) and a 0.5 probability that the formula will be unsatisfied (if the coin flip comes up `False`). The situation is similar if the solver sets x_1 to `False`. (Since the solver can choose values for the existentially quantified variables and the probability of satisfaction depends on the chance outcomes of the randomized variables, I will sometimes refer to existentially quantified variables as *choice variables* and randomly quantified variables as *chance variables*.)

Second, quantifier ordering is now critical. In the example, a value for x_1 must be chosen that yields a sufficiently high probability of satisfaction regardless of the randomly chosen value for y_2. This is impossible; either value of x_1 will result in an unsatisfied formula for one of y_2's values, so the maximum probability of satisfaction is 0.5. Suppose, however, the order of the quantifiers were reversed:

$$\text{R}y_1, \exists x_2(E[(x_2 \vee \overline{y}_1) \wedge (\overline{x}_2 \vee y_1) \leftrightarrow \texttt{True}] \geq 0.75). \tag{5.2}$$

Here, the choice of a value for x_2 can be made *contingent* on the random outcome of the coin flip establishing y_1's value. In this case, choosing x_2's value to be the same as y_1's value leads to a satisfied formula regardless of the coin flip. The

probability of satisfaction is now 1.0, exceeding the specified threshold.

Formally, an SSAT formula is defined by a triple (ϕ, Q, θ) where ϕ is a CNF formula with underlying ordered variables $x_1, \ldots, x_n$, Q is a mapping from variables to quantifiers (existential $\exists$ and randomized $\mathfrak{R}$), and $0 \leq \theta \leq 1$ is a satisfaction threshold. Define $\phi\lceil_{x_i=b}$ to be the $(n-1)$-variable CNF formula obtained by 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, or leftmost, quantifier and x_n to the innermost.)

The maximum probability of satisfaction, or value, of ϕ (under quantifier order Q), $val(\phi, Q)$, is defined by induction on the number of quantifiers. Let x_1 be the variable associated with the outermost quantifier. Then:

1. if ϕ contains an empty clause, then $val(\phi, Q) = 0.0$;
2. if ϕ contains no clauses then $val(\phi, Q) = 1.0$;
3. if $Q(x_1) = \exists$, then $val(\phi, Q) = \max(val(\phi\lceil_{x_1=0}, Q), val(\phi\lceil_{x_1=1}, Q))$;
4. if $Q(x_1) = \mathfrak{R}$, then $val(\phi, Q) = (val(\phi\lceil_{x_1=0}, Q) + val(\phi\lceil_{x_1=1}, Q))/2$.

Given ϕ, Q, and a threshold θ, (ϕ, Q, θ) is `True` if and only if $val(\phi, Q) \geq \theta$.

Let us examine the application of this definition to the original example (Equation 5.1). The outermost quantifier is existential, so Rule 3 dictates that the value of the formula is the maximum of the value of the subformula if x_1 is `True` and the value of the subformula if x_1 is `False`. If x_1 is `True`, the formula reduces to $\mathfrak{R}y_1(E[(y_1) \leftrightarrow \texttt{True}] \geq 0.75)$ (after variable renumbering). Since the outermost quantifier is now randomized, Rule 4 dictates that the value of this subformula is the average of the values if y_1 is `True` and if y_1 is `False`. If y_1 is `True`, the

new subformula contains no clauses and the value is 1.0 (Rule 2). If y_1 is `False`, the new subformula contains an empty clause and the value is 0.0 (Rule 1). The average of these, 0.5, is thus the value of the subformula when x_1 is `True`. If x_1 is `False`, a similar calculation establishes the value of the subformula to be 0.5. Taking the maximum, the value of the original formula is 0.5. Since the threshold θ is 0.75, the SSAT instance $(\phi = (x_1 \vee \overline{y}_2) \wedge (\overline{x}_1 \vee y_2),\ Q = \{(x_1, \exists), (y_2, \mathrm{R})\},\ \theta = 0.75)$ is `False`.

One further modification is necessary to encode planning problems as stochastic satisfiability problems. I will allow an arbitrary, rational probability to be attached to a randomly quantified variable. This probability will specify the likelihood with which that variable will have the value `True`. Thus, the value of a randomly quantified variable will be determined according to this probability, rather than choosing `True` or `False` with equal probability. This has an impact both on notation and on the inductive definition of value. Randomized quantifiers can now be superscripted with an associated probability other than 0.5. For example, $\mathrm{R}^{0.65} y_1$ indicates that the chance variable y_1 is `True` with probability 0.65. Rule 4 in the inductive definition of $val(\phi, Q)$ becomes:

4. if $Q(x_1) = \mathrm{R}^{\pi}$, then $val(\phi, Q) = (val(\phi\lceil_{x_1=0}, Q) \times (1.0 - \pi) + val(\phi\lceil_{x_1=1}, Q) \times \pi)$.

In other words, the value in this case is the probability weighted average of the values of the two possible subformulas.

For the sake of completeness, I note here that stochastic satisfiability can be extended to include universally quantified variables as well as existentially and randomly quantified variables. Although this version of stochastic satisfiability might be useful for encoding planning problems when there is an adversarial

situation, I do not use this version in any of my SSAT-based planners. Details regarding this *Extended* SSAT problem are available elsewhere [71].

5.1.2 MAJSAT

As I described in Section 3.2, SAT is the special case of stochastic satisfiability in which all the quantifiers are existential, making the ordering irrelevant, and the threshold is 1.0 (the assignment *must* satisfy the formula). MAJSAT is the special case of stochastic satisfiability in which all the quantifiers are randomized quantifiers. Again, the ordering is irrelevant, but now the threshold can be any probability $0 \leq \theta \leq 1$. This is somewhat different from the standard formulation of the MAJSAT problem: given a Boolean formula $\phi(\mathbf{x})$ in CNF, are at least half of its assignments satisfying? But, both formulations share the property of, in some sense, counting the number of satisfying assignments. To see this in the SSAT formulation of the problem, suppose that all assignments are equally likely (which is the case if all variables have an associated probability of 0.5). Then, asking whether at least half the assignments are satisfying is equivalent to asking whether the probability of satisfaction is at least 0.5:

$$\text{R}x_1, \ldots, \text{R}x_n(E[\phi(\mathbf{x}) \leftrightarrow \texttt{True}] \geq \mathbf{0.5}).$$

(Note that $\text{R} = \text{R}^{0.5}$.) The more general SSAT formulation of the problem substitutes a general threshold θ for the specific threshold of 0.5. (Papadimitriou [91] refers to this as "Threshold SAT".) Note that this formulation is polynomially equivalent to finding the exact probability of satisfaction, since one can use binary search on the threshold probability θ to determine this probability exactly.

5.1.3 E-MAJSAT

E-MAJSAT [70] is more general than MAJSAT in that an E-MAJSAT problem contains both existentially quantified variables (choice variables) and randomly quantified variables (chance variables). E-MAJSAT is still a special case of SSAT, however, because of the following restriction on the quantifier ordering: all choice variables must precede all chance variables in the ordering. In other words, in an E-MAJSAT problem, one must find settings for an initial block of choice variables that will yield a sufficiently high probability of satisfaction given the probabilities attached to the subsequent block of chance variables:

$$\exists x_1, \ldots, \exists x_c, \mathbb{R} y_{c+1}, \ldots, \mathbb{R} y_n (E[\phi(\mathbf{x}, \mathbf{y}) \leftrightarrow \texttt{True}] \geq \theta).$$

In this E-MAJSAT problem, the values of choice variables x_1 through x_c are under the control of the solver, while those of chance variables y_{c+1} through y_n are not. The difficulty lies in the necessity of choosing the values for the choice variables before the random settings of the choice variables are known. Notice that once a setting for the choice variables has been chosen, only chance variables remain and the problem reduces to a MAJSAT problem: does the probability of satisfaction, given those choice variable settings, exceed the specified threshold? In effect, to solve an E-MAJSAT problem, one needs to solve multiple MAJSAT problems (naively, 2^c MAJSAT problems). In fact, E-MAJSAT is short for "EXISTS-MAJSAT": does there *exist* a setting for the choice variables such that the probability of satisfaction meets or exceeds the specified threshold?

As was the case for MAJSAT, this threshold formulation of E-MAJSAT is polynomially equivalent, via binary search, to a formulation of E-MAJSAT in which one is required to find the settings of the choice variables that maximizes the

probability of satisfaction given the probability distribution over chance variable settings.

5.2 Solving Stochastic Satisfiability Problems

I will describe two algorithms for solving SSAT problems. The exact algorithm in Section 5.2.1 is both sound and complete: given an arbitrary SSAT instance (ϕ, θ, Q), this algorithm is 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 (briefly described in Section 3.4) have been shown to be successful for large problem instances. Similarly, MAJSAT instances are very naturally approximated using random sampling. Section 5.2.3 describes a new SSAT algorithm that combines local search with random sampling to compute approximate answers to SSAT problems.

5.2.1 A Davis-Putnam-Logemann-Loveland Algorithm

The evalssat algorithm described in this section can be viewed as an extension of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving SAT problems [27]. To my knowledge, DPLL and its variants are the best systematic satisfiability solvers known. As such (and also because of its simplicity), DPLL was the obvious choice as a basis for an SSAT solver. DPLL works by enumerating all possible assignments, simplifying the formula whenever possible. These simplifications, or pruning rules, make it possible to solve problems whose entire set of assignments could not be completely enumerated. Since DPLL is designed

to solve SAT problems, the pruning rules only need to deal with existential quantifiers. The evalssat algorithm extends the DPLL algorithm to SSAT by providing pruning rules for randomized quantifiers.

The evalssat algorithm (Figure 5.1) 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 SSAT formula is less than θ_l, a value greater than θ_h if and only if the value of the SSAT formula is greater than θ_h, and otherwise the exact value of the SSAT formula. (Note that π_b^v denotes the probability that randomized variable v has value b.) Thus, this algorithm can be used to solve the 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 SSAT formula from its definition (Section 5.1.1); this takes place in the first two lines of pseudocode and in the section of 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.

Unit Propagation

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 propagation*, 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 recurring. As in DPLL, this is valid because assigning $x_i = 1 - b$ is guaranteed to make ϕ `False`, and $x_i = b$ can be no worse. Similarly, if the quantifier associated with x_i is randomized, it is the case that one

```
evalssat(φ, Q, θ_l, θ_h) := {
  if φ is the empty set, return 1
  if φ contains an empty clause, return 0
  /* Unit Propagation */
  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 with sign b and Q(x_i) = ∀,
    return evalssat(φ⌈_{x_i=b}, Q, θ_l/π_b^{x_i}, θ_h/π_b^{x_i})π_b^{x_i}
  /* Pure Variable Elimination */
  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 − π_1^{x_1})/π_0^{x_1}, θ_h/π_0^{x_1})
    if v_0π_0^{x_1} + π_1^{x_1} < θ_l, return v_0π_0^{x_1}
    if v_0π_0^{x_1} ≥ θ_h, return v_0π_0^{x_1}
    v_1 = evalssat(φ⌈_{x_1=1}, Q, (θ_l − v_0π_0^{x_1})/π_1^{x_1}, (θ_h − v_0π_0^{x_1})/π_1^{x_1})
    return v_0π_0^{x_1} + v_1π_1^{x_1}
  }
}
```

Note: π_b^v denotes the probability that randomized variable v has value b.

Figure 5.1: The evalssat algorithm generalizes the DPLL algorithm for satisfiability to solve SSAT problems.

branch of the computation will return a zero, so x_i can be eliminated from the formula by assigning it value b and continuing recursively. The resulting value is multiplied by the probability associated with the forced value of the randomized quantifier ($\pi_b^{x_i}$), since it represents the value of only one branch.

Pure Variable Elimination

Pure variable elimination applies when there is a pure variable; *i.e.* 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 recurs. This is valid because there are no unsatisfied clauses that would be satisfied if $x_i = 1 - b$ but unsatisfied if $x_i = b$. Interestingly, pure variable elimination does not appear to be possible for randomized variables. Both assignments to a randomized variable give *some* contribution to the value of the SSAT formula, and must be considered independently.

Thresholding

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 propagation, 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. Note that thresholding is similar to MINIMAX tree **cutoffs* [5].

If $Q(x_1) = \exists$, after the first recursive call computing v_0 (the value of the current formula with x_1 set to `False`), 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 (the value of the current formula with x_1 set to `True`). In particular, it is possible that $v_1 > v_0$, but all that is significant is whether one 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.

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 v_0, the value obtained by assigning 0 to x_1 is so low that, even if v_1, the value obtained by assigning 1 to x_1, attains its maximum value of 1, the low threshold will not be met ($v_0\pi_0^{x_1} + \pi_1^{x_1} < \theta_l$), then the algorithm can return $v_0\pi_0^{x_1}$ without calculating v_1. Second, if v_0 is high enough to meet the high threshold even if $v_1 = 0$ ($v_0\pi_0^{x_1} \geq \theta_h$), the algorithm can, again, return $v_0\pi_0^{x_1}$ without computing v_1. If both tests fail, the algorithm needs to compute v_1, but can adjust the thresholds accordingly.

For a detailed explanation of thresholds, see the proof of correctness of evalssat immediately following.

Correctness of evalssat

The following lemma shows that the evalssat algorithm produces the correct value of the given SSAT instance $(\phi, Q, \theta_l, \theta_h)$. Recall that π_b^v denotes the probability that randomized variable v has value b.

Lemma 1. *For all n, for all formulae ϕ with at most n variables, for all thresholds θ_l, θ_h, such that $\theta_l \leq \theta_h$,* 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 $\phi\lceil_{x_i=b}$. If x_i is a unit variable and $Q(x_i) = \forall$, then $val(\phi, Q) = val(\phi\lceil_{x_i=b}, Q)\pi_b^{x_i} + val(\phi\lceil_{x_i=1-b}, Q)\pi_{1-b}^{x_i} = val(\phi\lceil_{x_i=b}, Q)\pi_b^{x_i}$. 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 $1/\pi_b^{x_i}$. That is, when $val(\phi, Q) = v$, $val(\phi\lceil_{x_i=b}, Q)$ is $v/\pi_b^{x_i}$; thus if $\theta_l \leq v \leq \theta_h$, then by induction evalssat$(\phi\lceil_{x_i=b}, Q, \theta_l/\pi_b^{x_i}, \theta_h/\pi_b^{x_i})$ will return the value $v/\pi_b^{x_i}$ since $\theta_l/\pi_b^{x_i} \leq v/\pi_b^{x_i} \leq \theta_h/\pi_b^{x_i}$. Similarly, the algorithm is correct for $v \leq \theta_l$ and $v \geq \theta_h$.

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

The third case is when x_1 is removed and $Q(x_1) = \exists$. Let $v_0 =$ 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 $v_1 =$ 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 and final case is when x_1 is removed and $Q(x_1) = \forall$. Let $v_0' = val(\theta\lceil_{x_1=0}, Q)$ and let $v_1' = val(\theta\lceil_{x_1=1}, Q)$. If $v_0' < (\theta_l - \pi_1^{x_1})/\pi_0^{x_1}$ then $v_0'\pi_0^{x_1} + v_1'\pi_1^{x_1} < [(\theta_l - \pi_1^{x_1})/\pi_0^{x_1}]\pi_0^{x_1} + \pi_1^{x_1} = \theta_l$. If $v_0' > \theta_h/\pi_0^{x_1}$, then $v_0'\pi_0^{x_1} + v_1'\pi_1^{x_1} > (\theta_h/\pi_0^{x_1})\pi_0^{x_1} = \theta_h$. So, no detailed answer is needed if $v_0' < (\theta_l - \pi_1^{x_1})/\pi_0^{x_1}$ or $v_0' > \theta_h/\pi_0^{x_1}$. Thus, it suffices to calculate $v_0 = \mathsf{evalssat}(\phi\lceil_{x_1=0}, Q, (\theta_l - \pi_1^{x_1})/\pi_0^{x_1}, \theta_h/\pi_0^{x_1})$. If $v_0\pi_0^{x_1} + \pi_1^{x_1} < \theta_l$, then $v_0'\pi_0^{x_1} + v_1'\pi_1^{x_1} < \theta_l$, so return $v_0\pi_0^{x_1}$ (anything less than θ_l). If $v_0\pi_0^{x_1} \geq \theta_h$, then $v_0'\pi_0^{x_1} + v_1'\pi_1^{x_1} \geq v_0\pi_0^{x_1} \geq \theta_h$, so return $v_0\pi_0^{x_1}$. Otherwise, calculate $v_1 = \mathsf{evalssat}(\phi\lceil_{x_1=1}, (\theta_l - v_0\pi_0^{x_1})/\pi_1^{x_1}, (\theta_h - v_0\pi_0^{x_1})/\pi_1^{x_1})$. Now, if $v_1 < (\theta_l - v_0\pi_0^{x_1})/\pi_1^{x_1}$, then $v_0'\pi_0^{x_1} + v_1'\pi_1^{x_1} < \theta_l$, and if $v_1 > (\theta_h - v_0\pi_0^{x_1})/\pi_1^{x_1}$, then $v_0'\pi_0^{x_1} + v_1'\pi_1^{x_1} > \theta_h$. □

5.2.2 An Evaluation of Heuristics for evalssat

In all but the most trivial SAT problems, the DPLL algorithm will exhaust opportunities to apply unit propagation and pure variable elimination 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`. For peak performance, 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_i is set to `True` the formula becomes unsatisfiable. If x_i 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_i to `True` makes the formula unsatisfiable. If, however, it selects x_i 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 propagation, as described in Section 5.2.1, 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 [7, 24, 36, 37, 48, 50, 53, 66]. 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 selection from among a specific class of variables: 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 5.1.1. (Recall that quantifier ordering is important; adjacent quantifiers commute if they are of the same type, but not if they are of different types. See the example in Section 5.1.1.)

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, I conducted tests 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 [50]) chooses a variable whose literals appear in a large number of short clauses,
- POS_2S_JW (positive, 2-sided Jeroslow-Wang [50]) 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
- MAX_UNIT [7], is similar to heuristics used by the successful POSIT [36] and TABLEAU [24] solvers, preferring variables that will lead to a maximal number of unit propagations.

With the exception of RAND, these heuristics try to select the variable that will most simplify the formula and establish its satisfiability (or lack thereof) most quickly. SATS_MOST attempts to do this by choosing the literal that will satisfy, and thus eliminate, the most clauses. The remaining heuristics try to choose a variable that maximizes the number of unit clauses likely to be produced. The baseline for comparison is the splitting rule in Figure 5.1, 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 formulas were generated with the following char-

acteristics:

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 the usual fixed-clause random k-CNF model [71], 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 without replacement and randomly negating it with probability 0.5. This distribution is denoted $\mathcal{F}_m^{k,n}$. In my experiments, I used a program called `makewff`, available from AT&T Labs – Research, to generate random formulas from this distribution. In order to ensure that the formulas generated contained exactly n variables, however, I modified `makewff` 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 143 MHz Sun Ultra-1 with 128 Mbytes of RAM, running SunOS-5.7. Performance was measured by counting the change in the number of recursive calls to the central function in the algorithm (*i.e.* the function that extends the current partial assignment and checks for satisfiability or unsatisfiability). 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. The DPLL tree is the partial tree of assignments generated during evalssat's enumeration of possible assignments. This tree has the property that for every i, j, $i < j$, every variable in block i is split upon before every variable in block j, with the possible exception of variables that become irrelevant, variables that appear in unit clauses, and pure variables.

Measuring the number of recursive calls to this central funciton is 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 CPU 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 (Figures 5.2 and 5.3) generally supported the hypothesis that efficiency improves with block size, 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, MAX_UNIT, 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 5.2). 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 than they do for a block size of 12 (see Figure 5.3). My conjecture is that this phenomenon is due to the fact that pure variable elimination cannot be used in these problems. Given a total of 24 variables and a block size of 24, the randomly quantified outer block of variables is the *only* block of variables—a MAJSAT problem—and pure variable elimination cannot be applied to randomly quantified variables (Section 5.2.1: Pure Variable Elimination).

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 that might have demonstrated improved performance, was similarly neutral. 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 else-

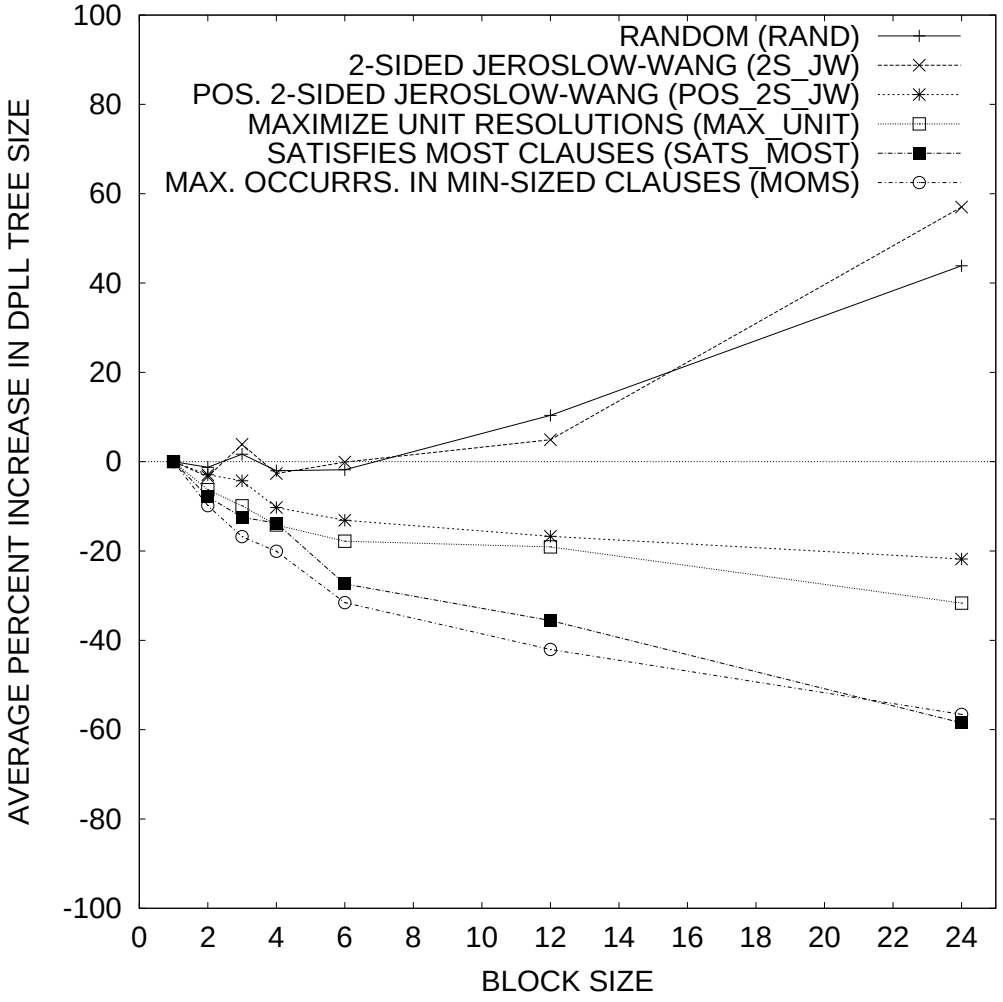


Figure 5.2: The heuristics POS_2S_JW, MAX_UNIT, SATS_MOST, and MOMS reduce the size of the DPLL tree generated relative to splitting in strict quantifier order when the formula block size is greater than 1 and variables in the first block are existentially quantified.

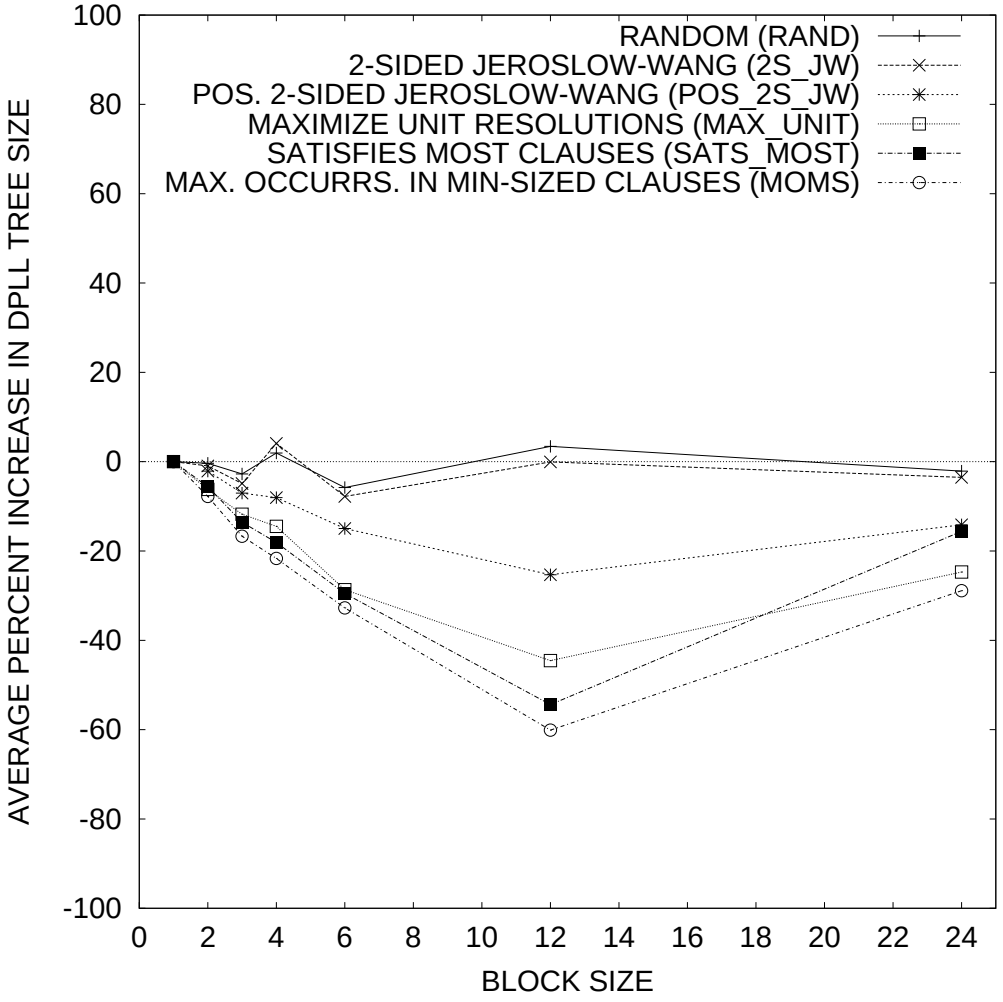


Figure 5.3: The heuristics POS_2S_JW, MAX_UNIT, SATS_MOST, and MOMS reduce the size of the DPLL tree generated relative to splitting in strict quantifier order when the formula block size is greater than 1 and variables in the first block are randomly quantified.

where [50]; on a set of hard, random SAT instances, Hooker and Vinay reported that the number of nodes visited (variable assignments made) in the search for a satisfying assignment 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. MAX_UNIT, 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.

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 formulas that encode planning problems, these results may be different. In addition, there may be other good splitting heuristics that take advantage of this structure. In Section 6.4, I will describe results that show 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 MAX_UNIT or MOMS, was somewhat surprising and warrants further study. 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 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.

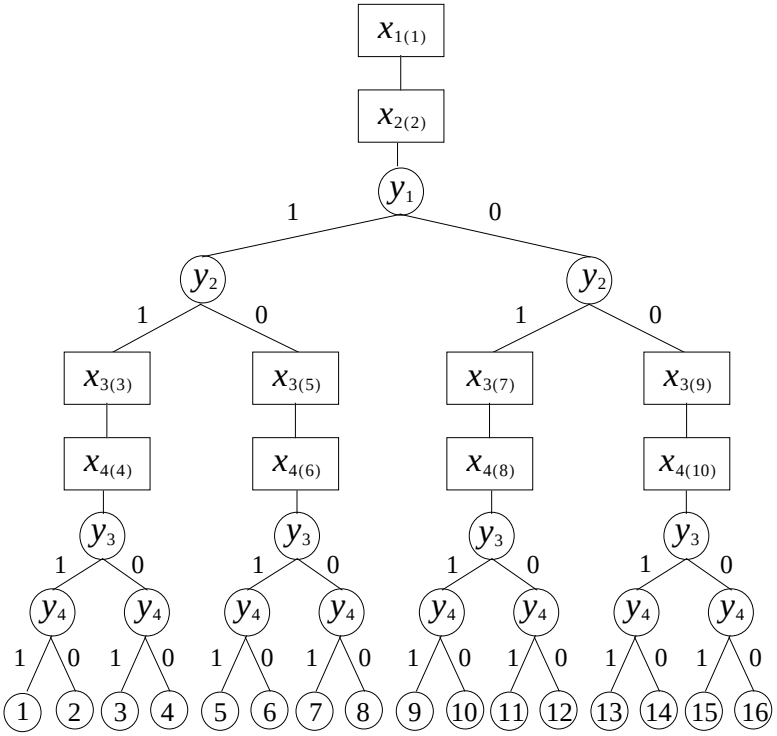
5.2.3 A Stochastic Sampling Algorithm

In general, DPLL appears to be a powerful and flexible approach to solving SSAT problems. A less systematic approach, however, may be necessary to attack very large SSAT problems. Randomized local search algorithms, like WALKSAT [100], have been extremely successful in solving difficult satisfiability problems (see Section 3.4). The presence of randomly quantified variables in SSAT problems, however, makes it impossible to apply local search techniques directly to SSAT problems. A possible approach is suggested by an approximation technique for MAJSAT problems: stochastic sampling. In this approach, some number of assignments to the random variables are generated according to their probabilities, and then the probability that the formula is satisfiable is estimated from that sample. This section develops an approximation algorithm for SSAT problems that combines stochastic sampling to reduce the size of the problem (Section 5.2.3: Stochastic Sampling) and randomized local search to solve the reduced problem efficiently (Section 5.2.3: Randomized Local Search).

Policy Trees

The starting point for the stochastic sampling algorithm for SSAT is the *policy-tree* representation. Figure 5.4 provides an example policy tree for the SSAT instance

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



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 5.4: A policy tree represents the set of contingent choices in an SSAT problem.

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, are 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 5.4, 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 formulas in Figure 5.4 are subscripted with the new numbers only.

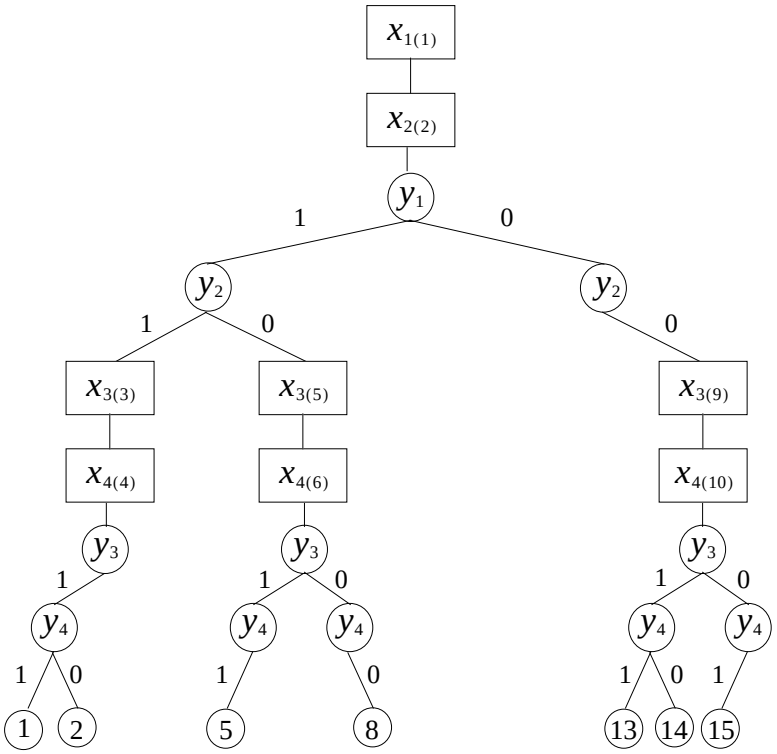
Each leaf of the policy tree represents a partial assignment consisting of an assignment to all randomized variables in the SSAT formula. The *probability of a leaf* is $1/2^R$ in a formula with R randomized variables. (Note that the stochastic sampling algorithm assumes that all randomized variables have an associated probability of 0.5. In the more general case, in which randomized variables can have arbitrary associated probabilities, the probability of a leaf is the product of the probabilities of the outcomes of the randomized variables 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 5.4). 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 5.4 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 5.4 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 solution tree generated by the DPLL algorithm, and the value of a policy tree is exactly that of a DPLL tree for the formula. It is a convenient representation for organizing the sampling algorithm described next.

Stochastic Sampling

The value of a fixed policy with respect to an SSAT instance can be computed in time linear 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 constructing and evaluating a *partial policy tree*. A partial policy tree is constructed by choosing a set W of w assignments to the randomized variables proportional to their probability and independently of the policy. 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 5.5 gives a partial policy tree derived from the sample in the bottom of the figure and the policy tree of Figure 5.4. Let s be the number of sampled leaves that are satisfied by the policy. Then, the value



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 5.5: 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.

of the policy is estimated as $\frac{s}{w}$, the proportion of the w sampled leaves that the policy satisfies. Let v be the true value of the policy and $\hat{v}$ be the estimate found via sampling. The following lemma allows us to bound the error in our estimate of v.

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

Proof. Given a fixed policy to evaluate, define $X_i = 1 - v$ to be the event of sampling a leaf that yields a satisfying assignment. The probability of this event is just v, the true value of the policy. Define $X_i = -v$ to be the event of sampling a leaf that yields an unsatisfying assignment; the probability of this event is $1 - v$. All the X_is are mutually independent, and

$$X = \sum_{i=1}^{w} X_i = (1 - v)s - v(w - s) = s - vw \tag{5.3}$$

where w is the number of samples and s is the number of sampled leaves that are satisfied by the policy. The Chernoff bound ([2], Corollary A.7, p. 236) states that, given the above assumptions, for $a > 0$,

$$\Pr[|X| > a] < 2\exp(-2a^2/w). \tag{5.4}$$

Substituting Equation 5.3 into Equation 5.4 and setting $a = \epsilon w$ yields:

$$\Pr[|s - vw| > \epsilon w] < 2\exp(-2\epsilon^2 w)$$

Since $\frac{s}{w} = \hat{v}$, the result follows. □

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

```
sampleevalssat(φ, Q, W) := {
  if φ is the empty set, return 1
  if φ contains an empty clause, return 0
  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) = ∀, {
    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 5.6: A stochastic sampling algorithm for SSAT problems chooses the best approximate value.

number does not depend on n, m, k, or how the SSAT formula or the policy was created.

Lemma 2 directly gives a polynomial time 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 5.6 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 approximation to the true value of the given SSAT formula with high probability.

Theorem 3. *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 2, 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 ensure an accurate estimate of the value of the formula with high probability. Note that for SAT, the number of policies is $P = 2^n$, MAJSAT, $P = 1$, and E-MAJSAT, $P = 2^c \leq 2^n$. Since the sample complexity for sampleevalssat is logarithmically dependent on P, these SSAT problems have polynomial sample complexity. However, for SSAT problems in which existential and randomized quantifiers strictly alternate, $P = \Theta(2^{2^{n/2}})$ (the $n/2$ randomized quantifiers induce a policy tree with $2^{n/2}$ decision nodes). In this case, Theorem 3 gives an exponential sample bound. In fact, there are SSAT formulas that require exponentially many samples to compute an accurate estimate using the sampleevalssat

algorithm [71].

Randomized Local Search

The basic approach just described 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. As described earlier in this section, stochastic sampling can be used to construct a partial policy tree and, given a policy, the partial policy tree can be evaluated (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 5.7 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 5.7). As illustrated in Figure 5.4, the effect of a policy on a leaf can be summarized by a Boolean formula, called a *path formula*, for that leaf. A path formula for path p is the original formula ϕ simplified by the randomized variable assignments described by path p. Each random sample can potentially produce a different path formula, and the union of all the path formulas produces the treeSAT formula. Thus, on one level the treeSAT formula is a collection of path formulas. while on another level it is merely a collection of clauses (all the clauses in all the path formulas).

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 formulas 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 formulas 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 [100]. 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 formulas satisfied. (In fact, in the tests conducted, counting satisfied clauses did not seem to be as important as counting satisfied path formulas; randevalssat performed as well with an objective function that counted only satisfied path formulas. 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.

Related Work

Many other researchers have explored approximations for probability-based problems. Dagum and Luby showed that approximating probabilistic inference in belief networks is NP-hard [26]. In particular, this means that the problem of finding an absolute approximation of the value of an SSAT formula (*i.e.* finding an approximate value $\hat{v}$ of an SSAT formula with true value v such that $\Pr[|v - \hat{v}| < 0.5] > 0.5$) is NP-hard if NP $\subseteq$ RP (where RP is the class of problems that can be solved in polynomial time with a Monte Carlo Turing machine [91]). The difficulty in this general case is that one needs to compute conditional probabilities and it may be exponentially difficult to generate the cases that are needed for the denominator (*i.e.* those cases that are valid for the particular conditional probability being calculated). Lemma 2, which shows that it is possible to find an absolute approximation of the value of a MAJSAT problem in polynomial time, is possible because in sampling to arrive at an estimate of the value of the MAJSAT problem, *every* sample is useful for the estimate.

Roth [98] studied the complexity of inference in belief networks. He showed that computing the probability of a query node (akin to evaluating a MAJSAT

```
randevalssat(φ, W) := {
  ρ := construct_treesat_formula(φ, W);
  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 5.7: A randomized algorithm for SSAT problems uses hillclimbing on a SAT formula derived from a partial policy tree created by sampling.

formula) is #P-complete. (#P is the "counting" version of PP; for example, #SAT asks *how many* satisfying assignments there are for a given Boolean formula.) 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 2 shows that the complexity of getting within an *additive* factor for MAJSAT is polynomial.

Kearns et al. [62] show how approximately optimal actions can be chosen with high probability in infinite-horizon discounted 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 described earlier in this section was directly inspired by their work.

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 [71]. 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 formulas.

5.2.4 Experiments with Stochastic Sampling Algorithms

The performance of randevalssat for computing the value of SSAT instances was tested on 27 sets of 100 random formulas. The characteristics of these 27 sets of formulas 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 5.2.2, these sets were generated according to $\mathcal{F}_m^{k,n}$ 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 that 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 approximately 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 143 MHz Sun Ultra-1 with 128 Mbytes of RAM, running SunOS-5.7. 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.

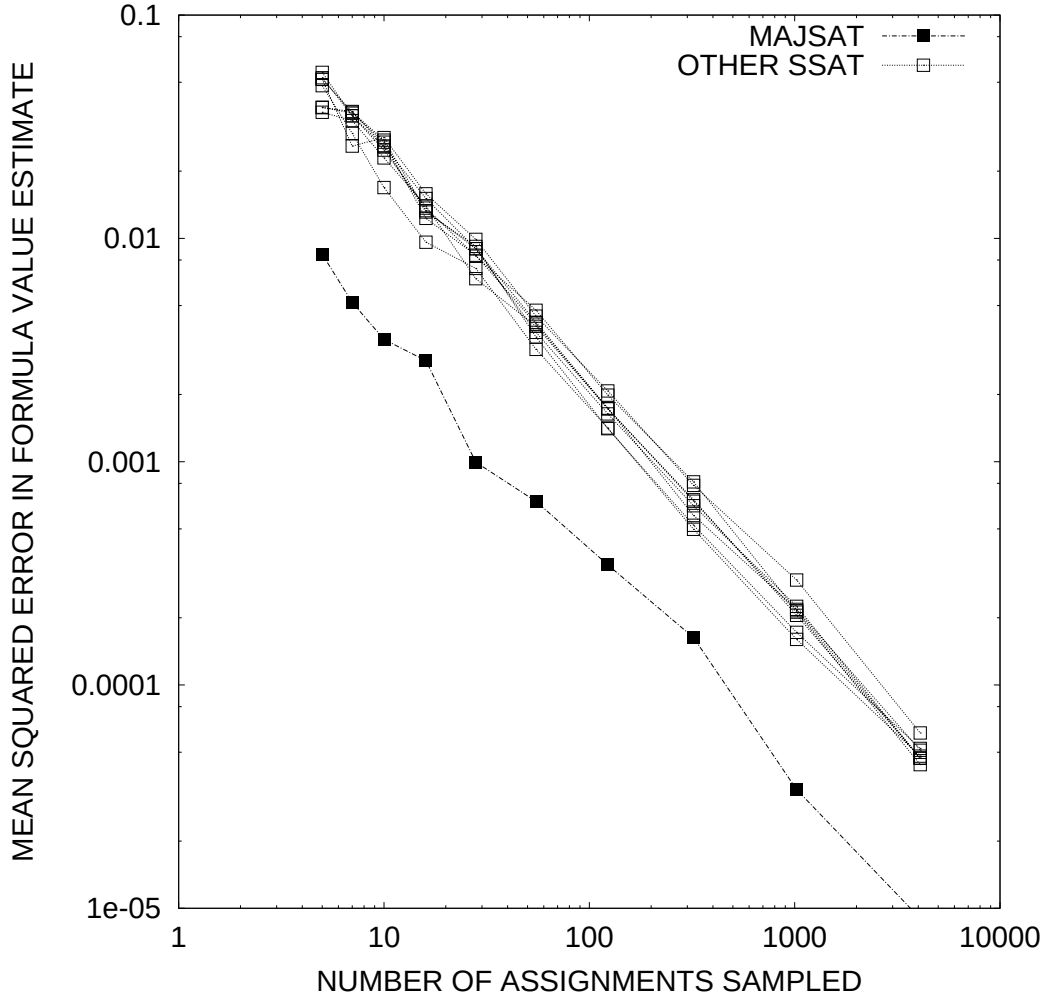


Figure 5.8: The partial policy tree produced by random sampling can be solved exactly using a DPLL-style approach. Approximation error decreases as sample size increases. These problems contained 12 variables, 24 clauses, and 3 literals per clause.

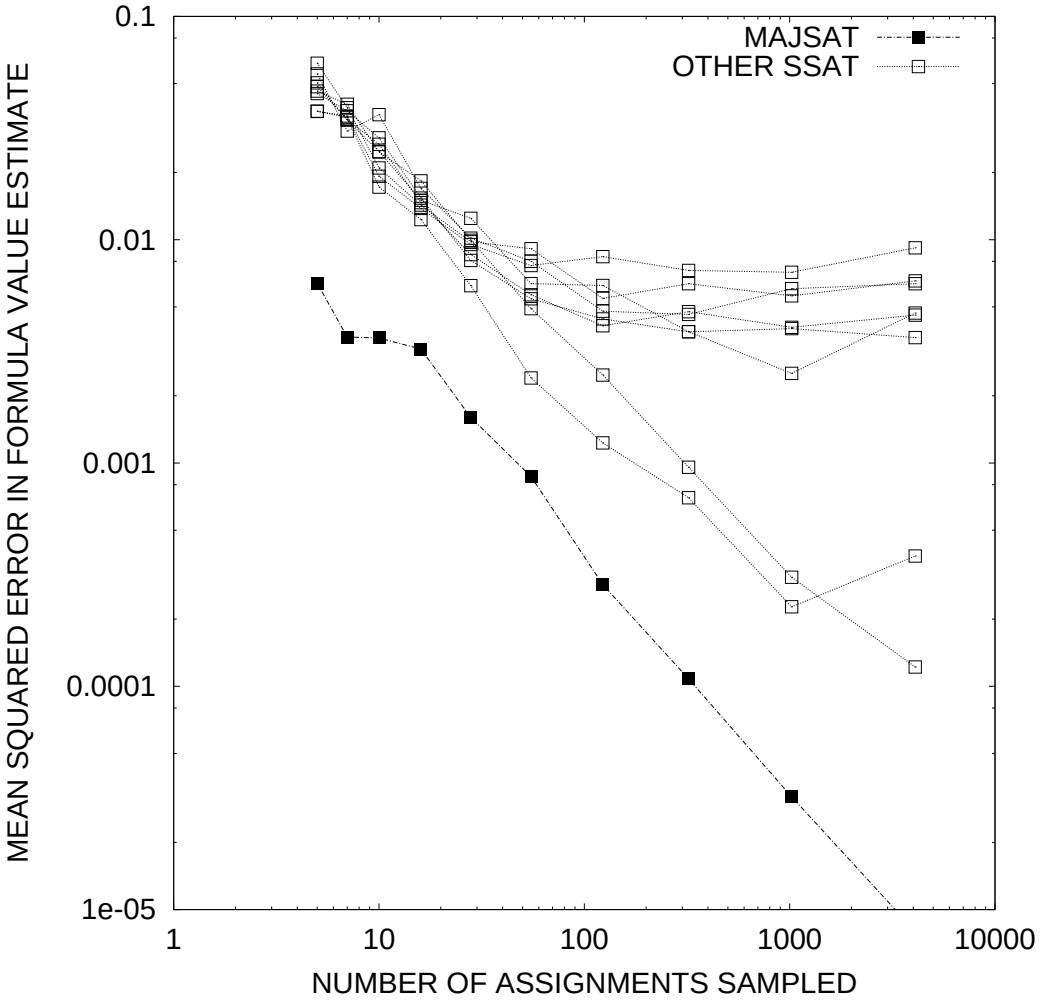


Figure 5.9: As the number of sampled assignments increases, the accuracy of the randomized local search algorithm increases. Because of local optima in the search space, increasing the number of sampled assignments does not drive the error to zero. These problems contained 12 variables, 24 clauses, and 3 literals per clause.

Figure 5.8 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 except the SAT problems. (The "tree" for a SAT problem is just a single node containing all the existential variables, and sampleevalssat performs a systematic search for satisfying assignments, so the value of the SAT problems is estimated without error in all cases.) 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 5.9 shows the same graphs for the results of the randevalssat algorithm. Note that the results for all 27 sets of problems were similar; we show the results only for the set of problems with 12 variables, 24 clauses, and 3 literal per clause. (Again, results for the SAT problems are not reported. Stochastic local search was able to find satisfying assignments, when they existed, in all cases except for the very lowest level of sampling (5 samples), which limited the number of iterations too severely to allow stochastic local search to work.) 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 local optima. Initially, generating a policy tree with more 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; because more policy-tree nodes means more treeSAT variables, the algorithm is increasingly likely to become stuck in local maxima and, therefore, is 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 in Figure 5.7 as a

subroutine in an algorithm that periodically restarts with a new set of samples. The approach could also be refined using simulated annealing.

5.3 Summary

In Sections 3.2 and 5.1.1, I described four types of satisfiability problems: the deterministic satisfiability problem (SAT) and three types of stochastic satisfiability problems (MAJSAT, E-MAJSAT, and SSAT). Each of these satisfiability problems is complete for a particular complexity class containing an important probabilistic planning problem. The three stochastic satisfiability problems completely fill in the gaps in the chart of complexity classes, complete satisfiability problems, and planning problems in Table 4.1, and they provide the basis for extending the planning-as-satisfiability paradigm to probabilistic planning. Table 5.1 shows the completed chart.

In Section 5.2, I described two algorithms for solving SSAT problems—an exact algorithm and an approximation algorithm. I showed that solution techniques developed for SAT problems could be extended to SSAT problems, and described some empirical results for both algorithms on large sets of randomly generated SSAT problems.

Stochastic satisfiability unites the fundamental computer science areas of satisfiability and probabilistic models. As such, I expect that it will function as the essential, defining problem for probabilistic domains in the same way that SAT does for deterministic domains. In Chapters 6 and 7, I will show how stochastic satisfiability can be used to encode and solve probabilistic planning problems.

Complexity Class	**Complete Satisfiability Problem**	**Planning Problem**
PSPACE	SSAT	Probabilistic planning with: polynomially bounded plan size **or** polynomially bounded plan horizon
NP^{PP}	E-MAJSAT	Probabilistic planning with: polynomially bounded plan size **and** polynomially bounded plan horizon
PP	MAJSAT	Probabilistic plan evaluation
NP	SAT	Deterministic planning with: polynomially bounded plan size **which is equivalent to** polynomially bounded plan horizon

Table 5.1: SSAT, E-MAJSAT, and MAJSAT provide the complete satisfiability problems that correspond to various Probabilistic planning problems.

Chapter 6

Planning Without Observability

Portions of this chapter have appeared in an earlier papers: "MAXPLAN: A new approach to probabilistic planning" [75] with Littman and "Using Caching to Solve Larger Probabilistic Planning Problems" [76] with Littman.

As described in Chapter 5, both E-MAJSAT, a type of stochastic satisfiability problem, and probabilistic propositional planning with polynomial bounds on plan size and plan horizon, are NP^{PP}-complete problems. This suggests a solution strategy for this type of planning problem analogous to that of SATPLAN [59]. In the same way that deterministic planning can be expressed as the NP-complete problem SAT, is it possible to express probabilistic planning as the NP^{PP}-complete problem E-MAJSAT?

MAXPLAN provides an affirmative answer to this question. MAXPLAN is a probabilistic planning technique that operates by converting a planning problem into an instance of E-MAJSAT and solving that problem instead (Figure 6.1). Briefly, this is accomplished by using the choice variables to encode candidate plans and the chance variables to encode the uncertainty in the planning domain. Given the E-MAJSAT encoding, MAXPLAN draws on techniques from Boolean satisfiability, dynamic programming, and caching to solve this E-MAJSAT instance, and the solution of the E-MAJSAT problem is directly interpretable as a solution of the planning problem.

MAXPLAN assumes complete unobservability and, therefore, produces plans that are a simple sequence of actions. I will refer to such plans—a finite se-

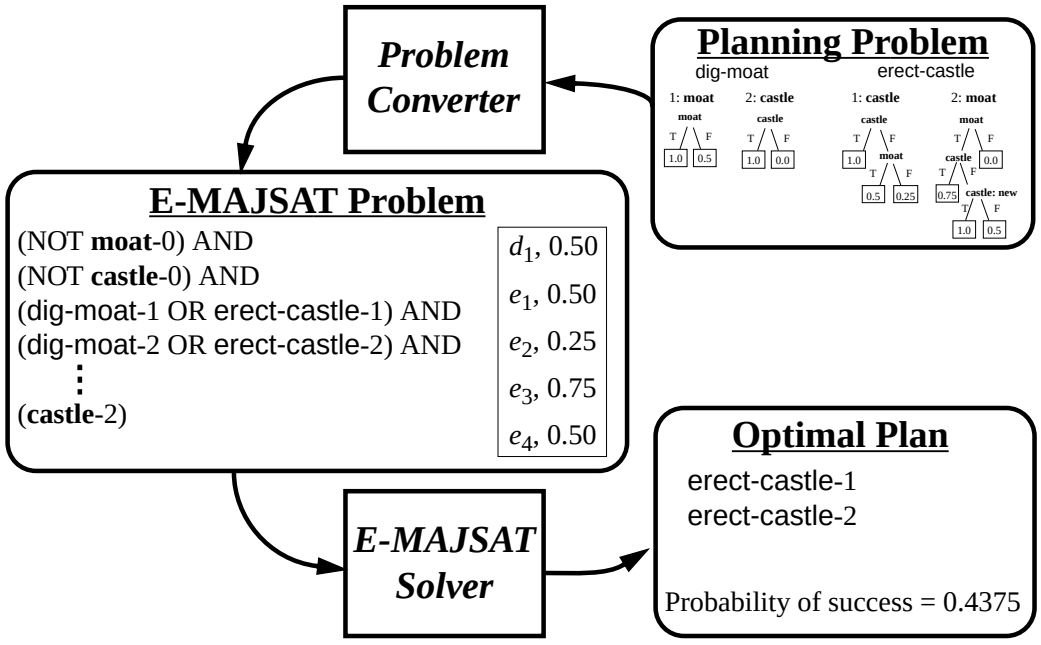
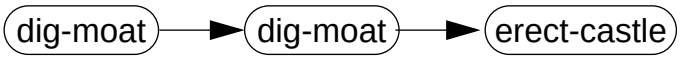


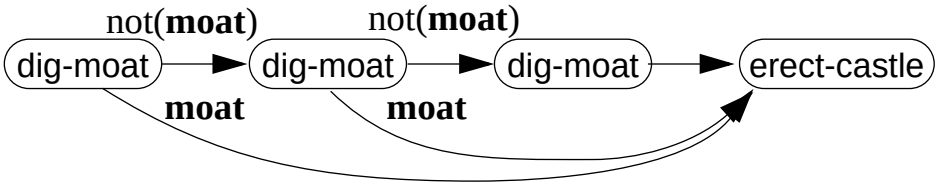
Figure 6.1: MAXPLAN converts a probabilistic planning problem to an instance of E-MAJSAT and solves that problem instead.

quence of actions that are executed in order without regard to the state of the environment—as *totally ordered plans* (Figure 6.2(a)). These are the simplest of three types of plans I will be referring to in this and subsequent chapters. The second type, *acyclic plans* are a generalization of totally ordered plans that include contingent actions (Figure 6.2(b)). Finally, *looping plans* are a generalization of acyclic plans in which actions can be repeated (Figure 6.2(c)).

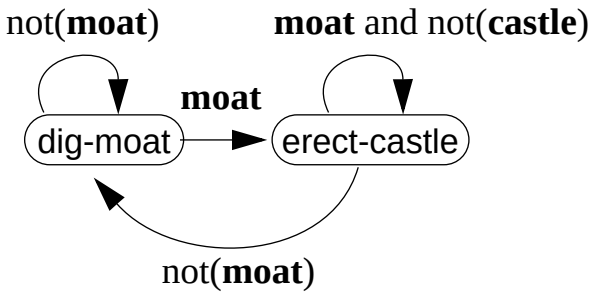
In broad outline, the operation of MAXPLAN parallels that of SATPLAN. There are, however, some significant differences due to the fact that MAXPLAN must deal with a stochastic domain. As I described in Chapter 3, SATPLAN converts a planning problem to an instance of SAT. Since there is no uncertainty in the planning environment, SATPLAN can use choice variables to encode both the plan and its outcome. In contrast, MAXPLAN makes a distinction between choice variables, which encode the possible plans, and chance variables, which encode the



(a) A totally ordered plan is a simple sequence of actions.



(b) An acyclic plan can express contingent actions.



(c) In a looping plan, actions can be repeated.

Figure 6.2: Totally ordered plans, acyclic plans, and looping plans are three types of probabilistic plans [70].

uncertainty in the probabilistic planning problem. Together, the chance variables determine the probability that a given truth assignment for the choice variables (a given plan) will actually lead to a satisfying assignment (reach the goal).

Given a CNF formula with chance variables, MAXPLAN must find, not merely a single satisfying assignment (as in SATPLAN), but rather the assignment of choice variables that has the highest probability of producing a complete satisfying assignment. This means that MAXPLAN must, in effect:

- determine all possible satisfying assignments,
- calculate the probability of each satisfying assignment (based on the chance variable probabilities)
- sum the satisfying assignment probabilities for each possible choice-variable assignment, and
- return the choice variable assignment (plan) with the highest probability of producing a satisfying assignment (successful plan).

Given these requirements, it is very difficult to avoid doing a systematic search over the set of all assignments for those that are satisfying. In fact, as I will describe in more detail in Section 6.4, the MAXPLAN algorithm for solving E-MAJSAT problems is based on evalssat, the extension of the systematic DPLL algorithm I described in Section 5.2.1.

Thus, even if only totally ordered plans are considered, the problem of plan evaluation is significantly more difficult in a stochastic domain. Intuitively, in the deterministic setting, evaluating a plan means executing the plan and checking the single execution trace to see whether the final state is a goal state; given a polynomial-length plan, a restriction I assume for this research, this can be

done in polynomial time. In the stochastic setting, however, one needs to do the equivalent of checking each possible execution trace and summing the probability of each trace whose final state is a goal state; this is a PP-complete problem [70]. To place this in perspective, recall that NP $\subseteq$ PP $\subseteq$ NP$^{\text{PP}}$ $\subseteq$ PSPACE.

Furthermore, plans in a stochastic domain frequently need to be more complex than plans in a deterministic domain; a noncontingent plan that may suffice in a deterministic setting frequently fails when uncertainty is introduced. Optimal plans in a stochastic domain frequently require conditional branches that specify different actions depending on the stochastic outcome of the current action, or loops that repeat an action until a desired result is achieved. Complexity results indicate that as plan complexity increases, both searching through plan space and evaluating potential plans becomes more difficult [70].

6.1 Representing Probabilistic Planning Problems

Recall from Section 3.1 that a deterministic planning domain $M = \langle \mathcal{S}, s_0, \mathcal{A}, \mathcal{G} \rangle$ is characterized by a finite set of states $\mathcal{S}$, an initial state $s_0 \in \mathcal{S}$, a finite set of operators or actions $\mathcal{A}$, and a set of goal states $\mathcal{G} \subseteq \mathcal{S}$. I will use the same tuple to characterize probabilistic planning problems, except that now the initial state is characterized by a probability distribution over states (*i.e.* the initial state is uncertain) and the application of an action a in a state s results in a *probabilistic* transition to a new state s'. The objective is to choose actions, one after another, to move from the initial probability distribution to a probability distribution in which the probability of being in a goal state is greater than or equal to some

threshold θ.[1]

For the work described in this chapter, I have assumed a completely unobservable domain—the effects of previous actions cannot be used in selecting the current action. This is a special case of the POMDP formulation of the problem (which can be viewed as the control form of a hidden Markov model). Because no information is gained during plan execution, optimal plans are noncontingent sequences of actions. I relax this assumption in Chapter 7, where I discuss contingent planning in the planning-as-satisfiability framework (C-MAXPLAN and ZANDER).

All of my planners use a propositional representation called the sequential-effects-tree representation (ST) [68], which is a syntactic variant of two-time-slice Bayes nets (2TBNs) with conditional probability tables represented as trees [13, 15]. (This representation is also equivalent to probabilistic state-space operators (PSOs) [43, 68].) A 2TBN is a probabilistic action representation in which the effect of a probabilistic action is represented by a two-level Bayes net. Nodes containing all the propositions that condition the action, or that the action has an impact on, are contained in the first level, which represents the status of these propositions at time t. These nodes are replicated in the second level, representing the status of the propositions at time $t+1$. Each proposition on the second level has arcs pointing to it from all propositions (on either level) that condition the effect of the action on that proposition. Each node on the second level has a conditional probability table that describes the status of that node's proposition after the action has been executed.

[1]It is also possible to formulate the objective as one of maximizing expected discounted reward [15], but the two formulations are essentially polynomially equivalent [22]. The only difficulty is that compactly represented domains may require discount factors exponentially close to one for this equivalence to hold.

Recall that using such a factored state representation was one of the AI planning insights I mentioned earlier. Using a factored representation allows us to reason about states more efficiently by reasoning about state attributes rather than states as a whole. As we shall see in Sections 6.3 and 7.3.1, this compactness and efficiency transfers to the SSAT formulas when planning problems are converted to SSAT problems.

In the ST representation of a planning domain there is a finite set P of n distinct propositions, any of which may be `True` or `False` at any (discrete) time t. A state is an assignment of truth values to P. A probabilistic initial state is specified by a set of decision trees, one for each proposition. A proposition p whose initial assignment is independent of all other propositions has a tree consisting of a single node labeled by the probability with which p will be `True` at time 0. A proposition q whose initial assignment is not independent has a decision tree whose nodes are labeled by the propositions that q depends on and whose leaves specify the probability with which q will be `True` at time 0. As in most propositional representations, the states in the set of goal states $\mathcal{G}$ are not explicitly enumerated in ST. Instead, $\mathcal{G}$ is defined by a partial assignment G to the set of propositions; any state that extends G is considered to be a goal state.

The set of actions $\mathcal{A}$ is explicitly enumerated in ST. Each of the finite set $\mathcal{A}$ of actions probabilistically transforms a state at time t into a state at time $t+1$ and so induces a probability distribution over the set of all states. In this work, the effect of each action on each proposition is represented as a separate decision tree [16]. For a given action a, each of the decision trees for the different propositions are ordered, so the decision tree for one proposition can refer to both the new and old values of previous propositions. This allows the effects of an action to be correlated (*e.g.* when action a is executed under a certain set

of circumstances, proposition p is `True` with probability 0.35 and proposition q takes on the same value as p). The leaves of a decision tree describe how the associated proposition changes as a function of the state and action, perhaps probabilistically. Note that using decision trees captures *variable independence* (independence among variables regardless of their values) as well as *propositional independence* (independence of specific variable assignments) [13].

The planning task is to find a plan that selects an action for each step t as a function of the value of observable propositions for steps before t. We want to find a plan that maximizes (or exceeds a user-specified threshold for) the probability of reaching a goal state.

The ST representation of a planning domain $M = \langle \mathcal{S}, s_0, \mathcal{A}, t, \mathcal{G} \rangle$ can be defined more formally as $\mathbb{M} = \langle \mathbb{P}, \mathbb{I}, \mathcal{A}, \mathbb{T}, \mathbb{G}_{\mathbf{T}}, \mathbb{G}_{\mathbf{F}} \rangle$. Here, $\mathbb{P}$ is a finite set of n distinct propositions. The set of states $\mathcal{S}$ is the power set of $\mathbb{P}$; the propositions in $s \in \mathcal{S}$ are said to be "true" in s.

The transition function t is represented by a function $\mathbb{T}$, which maps each action in $\mathcal{A}$ to an ordered sequence of binary decision trees. Each of these decision trees has a label proposition, decision propositions at the nodes (optionally labeled with the suffix "**:new**"), and probabilities at the leaves. The ith decision tree $\mathbb{T}(a)_i$ for action a defines the transition probabilities $t(s, a, s')$ as follows. For each decision tree i, let $\mathbf{p}_i$ be its label proposition. Define ρ_i to be the value of the leaf node found by traversing decision tree $\mathbb{T}(a)_i$, taking the left branch if the decision proposition is in s (or s' if the decision proposition has the "**:new**" suffix) and the right branch otherwise. Finally, we let

$$t(s, a, s') = \prod_i \begin{cases} \rho_i, & \text{if } \mathbf{p}_i \in s', \\ 1 - \rho_i, & \text{otherwise.} \end{cases}$$

This definition of t constitutes a well-defined probability distribution over s' for each a and s.

To ensure the validity of the representation, we only allow "**p:new**" to appear as a decision proposition in $\mathbb{T}(a)_i$ if $\mathbf{p}$ is a label proposition for some decision tree $\mathbb{T}(a)_j$ for $j < i$. For this reason, the order of the decision trees in $\mathbb{T}(a)$ is significant.

The initial state $\mathbb{I}$ can be thought of as a special transition from a state s_{init} in which all propositions are `False` (the actual truth values are immaterial) via a mandatory "set-up" action a_{set-up} that establishes the actual initial state for a particular instance of the planning problem. Note that the decision trees for a_{set-up} are either single nodes specifying the probability that $\mathbf{p}_i$ is `True` in the initial state (*i.e.* the probability does not depend on any other proposition in the initial state), or all the decision propositions in the tree have the suffix **:new** (*i.e.* the probability of $\mathbf{p}_i$ being `True` in the initial state depends on the truth value of one or more propositions whose decision trees precede that of $\mathbf{p}_i$ in the ordering of decision trees for a_{set-up}).

The sets $\mathbb{G}_\mathtt{T}$ and $\mathbb{G}_\mathtt{F}$ are the sets of propositions that are, respectively, `True` and `False` in a goal state, so the set of goal states $\mathcal{G}$ is the set of states s such that $\mathbb{G}_\mathtt{T} \subseteq s$ and $\mathbb{G}_\mathtt{F} \subseteq \mathbb{P}-s$.

6.2 Example Domain

To help make these ideas more concrete, I present the following simple probabilistic planning domain based on the problem of building a sand castle at the beach (SAND-CASTLE-67) [68]. There are a total of four states in the domain, described by combinations of two Boolean propositions, **moat** and **castle** (propositions are

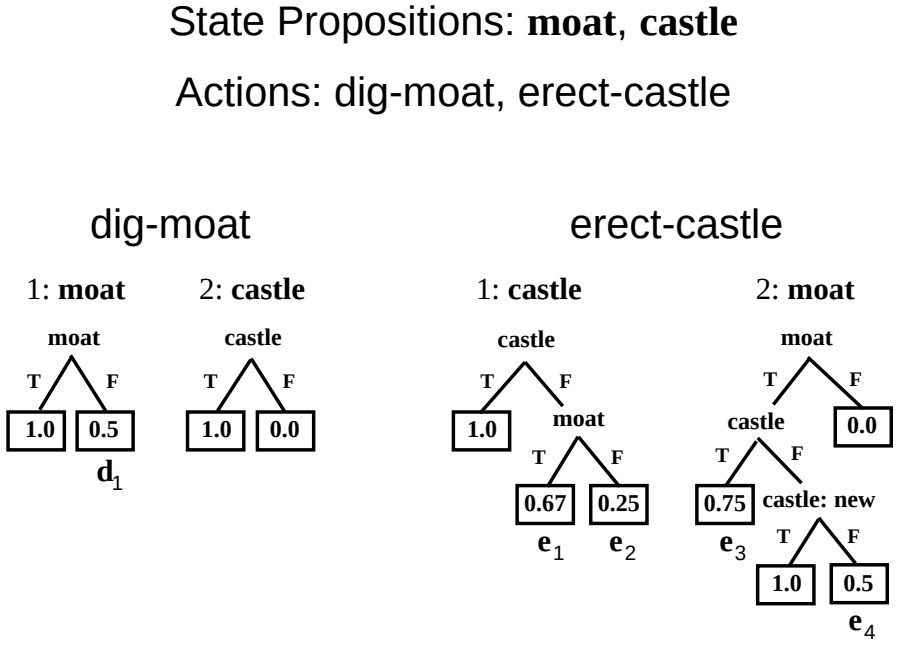


Figure 6.3: The sequential-effects-tree representation of SAND-CASTLE-67 consists of a set of decision trees. Decision tree leaves containing probabilities strictly between 0.0 and 1.0 give rise to chance variables ($\mathbf{d}_1$, $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, and $\mathbf{e}_4$).

written in boldface). The proposition **moat** signifies that a moat has been dug in the sand, and the proposition **castle** signifies that the castle has been built. In the initial state, both **moat** and **castle** are false, and the goal set is {**castle**}.

There are two actions: dig-moat and erect-castle (actions are given in sans serif). Figure 6.3 illustrates these actions in the ST representation. Executing dig-moat when **moat** is false causes **moat** to become true with probability 0.5; if **moat** is already true, dig-moat leaves it unchanged. The dig-moat action has no impact on **castle**. The dig-moat action is depicted in the left half of Figure 6.3.

The second action is erect-castle, which appears in the right half of Figure 6.3. The decision trees are numbered to allow sequential dependencies between their effects to be expressed. The first decision tree is for **castle**, which does not change value if it is already true when erect-castle is executed. Otherwise, the probability that it becomes true is dependent on whether **moat** is true; the castle is built

with probability 0.67 if **moat** is true and only probability 0.25 if it is not. The idea here is that first building a moat protects the castle from being destroyed prematurely by the ocean waves.

The second decision tree is for the proposition **moat**. Because erect-castle cannot make **moat** become true, there is no effect when **moat** is false. On the other hand, if the moat exists, it may collapse as a result of trying to erect the castle. The label **castle:new** in the diagram refers to the value of the **castle** proposition *after* the first decision tree is evaluated. If the castle was already built when erect-castle was selected, the moat remains built with probability 0.75. If the castle had not been built, but erect-castle successfully builds it, **moat** remains true. Finally, if erect-castle fails to make **castle** true, **moat** becomes false with probability 0.5 and everything is destroyed.

Note the differences between this formulation of the problem and the deterministic version of the problem described in Section 3.1. Unlike the deterministic dig-moat action, the probabilistic dig-moat action does not always work. In the probabilistic version of the problem, the erect-castle action sometimes works even if there is no **moat** (unlike the deterministic version, in which **moat** is a precondition for the erect-castle action), but sometimes the erect-castle action fails even if there *is* a **moat**. Even worse, if the erect-castle action fails to produce a **castle**, an existing **moat** may be destroyed.

6.3 Converting Planning Problems to E-MAJSAT Problems

The problem conversion unit of MAXPLAN is a LISP program that takes as input an ST representation of a planning problem and converts it into an E-MAJSAT

$$\overbrace{\exists x_1, \ldots, \exists x_{c_1}}^{\text{the plan}} \overbrace{\text{ꓤ}^{\rho_1} z_1, \ldots, \text{ꓤ}^{\rho_{c_2}} z_{c_2}}^{\substack{\text{domain} \\ \text{uncertainty}}} \overbrace{\text{ꓤ} y_1, \ldots, \text{ꓤ} y_{c_3}}^{\substack{\text{states} \\ \text{encountered}}}$$

$$\begin{array}{l}(E[\text{(initial/goal conditions } (y,z)\text{-clauses)} \\ \quad \text{(action exclusion } (x)\text{-clauses)} \\ \quad \text{(action outcome } (x,y,z)\text{-clauses)}] \geq \theta).\end{array}$$

c_1 = number of choice variables needed to specify the plan,
c_2 = number of chance variables (one for each possible stochastic outcome at each time step), and
c_3 = number of state variables (one for each proposition at each time step).

Figure 6.4: A generic E-MAJSAT encoding of a noncontingent planning problem.

formula with the property that, given an assignment to the choice variables (the plan), the probability of a satisfying assignment with respect to the chance variables is the probability of success for the plan specified by the choice variables. A generic E-MAJSAT encoding of a planning problem is given in Figure 6.4.

The converter produces the E-MAJSAT encoding of the planning problem by selecting a plan horizon T, time-indexing each proposition and action so the planner can reason about what happens when, and then making satisfiability equivalent to the enforcement of the following conditions:

- the initial conditions hold at time 0 and the goal conditions at time T,
- actions at time t are mutually exclusive ($1 \leq t \leq T$),
- proposition p is `True` at time t if it was `True` at time $t-1$ and the action taken at t does not make it `False`, *or* the action at t makes p `True` ($1 \leq t \leq T$).

The first two conditions are not probabilistic and can be encoded in a straight-

forward manner (Section 3.3), but the third condition is complicated by the fact that chance variables sometimes intervene between actions and their effects on propositions. Essentially, the encoding must be constructed to include:

- clauses that constrain the ways in which the initial conditions can be transformed into the goal conditions, and
- clauses that attach probabilities to these transformations.

I will illustrate the conversion process by describing the construction of the CNF formula corresponding to a one-step plan for SAND-CASTLE-67.

6.3.1 Variables

The converter first creates a set of propositions that capture the uncertainty in the domain. For each decision-tree leaf l labeled with a probability π_l that is strictly between 0.0 and 1.0, the converter creates a *random proposition* $\mathbf{r}_l$ that is true with probability π_l. For example, in the first decision tree of the dig-moat action (Figure 6.3), $\mathbf{d}_1$ is a random proposition that is `True` with probability 0.5. The leaf l is then replaced with a node labeled $\mathbf{r}_l$ having a left leaf of 1.0 and a right leaf of 0.0. This has the effect of slightly increasing the size of decision trees and the number of propositions, but also of simplifying the decision trees so that all leaves are labeled with either 0.0 or 1.0 probabilities.

Variables are created to record the status of actions and propositions in an T-step plan by taking three separate cross products: actions and time steps 1 through T, propositions and time steps 0 through T, and random propositions and time steps 1 through T. The total number of variables in the CNF formula

is

$$V = (A + P + R)T + P,$$

where A, P, and R are the number of actions, propositions, and random propositions, respectively.

The variables generated by the actions are the choice variables. In our example, these are the variables dig-moat-1 and erect-castle-1. The variables generated by the random propositions are the chance variables. In our example, we have five random propositions ($\mathbf{d}_1$, $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, and $\mathbf{e}_4$) and the variables generated are $\mathbf{d}_1$-1, $\mathbf{e}_1$-1, $\mathbf{e}_2$-1, $\mathbf{e}_3$-1, and $\mathbf{e}_4$-1. (I will describe the generation and use of these chance variables in more detail later in this section.)

The variables generated by the propositions for time steps 1 through T must also be chance variables. The E-MAJSAT solver returns 1) the setting of choice variables that yields the highest probability of a satisfying assignment given the chance variables, and 2) that probability. Given a particular setting of the action variables, there may be more than one setting of the proposition variables (we can think of this as the proposition-status history as the plan executes) that leads to a satisfying assignment, and we wish to sum the probabilities of all such proposition-status histories for a particular plan. If propositions also generated choice variables, the solver would return the best plan *and* proposition-status history and its associated probability rather than the best plan and its probability.

I will refer to these proposition variables as *pseudo-chance variables* to distinguish them from the chance variables generated by decision-tree leaf node probabilities. In the SAND-CASTLE-67 example, the pseudo-choice variables are **moat**-0, **castle**-0, **moat**-1, and **castle**-1. The probability associated with all pseudo-chance variables is 0.5, so that all proposition-status histories have the

same probability mass with respect to the pseudo-chance variables. Note that this lowers the probability of any choice variable setting by a factor of 0.5^{PT} and, thus, a suitable adjustment to the probabilities calculated by the solver must be made. This is not the only way to handle these proposition variables; they could be encoded as choice variables (where the choice is forced, given a choice of values for the action variables and an instantiation of values for the chance variables encoding the domain uncertainty). This is the approach taken by ZANDER (Section 7.3.1). This produces an SSAT encoding (PSPACE-complete) rather than an E-MAJSAT encoding (NP^{PP}-complete), and this increase in complexity is not necessary. The values of the proposition variables will be forced, given a choice of values for the action variables and an instantiation of values for the chance variables encoding the domain uncertainty, whether the proposition variables are choice variables or chance variables, so their values will be set efficiently by unit propagation in either case.

Each variable indicates the status of an action, proposition, or decision-tree leaf node at a particular time step. So, for example, the variable dig-moat-1, if `True`, indicates that the dig-moat action was taken at time step 1, and the variable e_1-1, if `True`, indicates that the decision-tree leaf node associated with e_1 is `True` at time step 1.

6.3.2 Clauses

Each initial condition and goal condition in the problem generates a unit clause in the CNF formula. The initial conditions in our example generate the clauses ($\overline{\textbf{moat}\text{-}0}$) and ($\overline{\textbf{castle}\text{-}0}$) and the goal condition generates the clause (**castle**-1). The number of clauses thus generated is bounded by $2P$.

Mutual exclusivity of actions for each time step generates one clause con-

Initial Conditions:

1. $(\overline{\textbf{moat-0}}) \wedge$
2. $(\overline{\textbf{castle-0}}) \wedge$

Goal Conditions:

3. $(\textbf{castle-1}) \wedge$

Exactly One Action Per Time Step:

4. $(\text{dig-moat-1} \vee \text{erect-castle-1}) \wedge$
5. $(\overline{\text{dig-moat-1}} \vee \overline{\text{erect-castle-1}}) \wedge$

Action Effects:

6. $(\overline{\text{dig-moat-1}} \vee \overline{\textbf{moat-0}} \vee \textbf{moat-1}) \wedge$
7. $(\overline{\text{dig-moat-1}} \vee \textbf{moat-0} \vee \overline{\textbf{d}_1\textbf{-1}} \vee \textbf{moat-1}) \wedge$
8. $(\overline{\text{dig-moat-1}} \vee \textbf{moat-0} \vee \textbf{d}_1\textbf{-1} \vee \overline{\textbf{moat-1}}) \wedge$
9. $(\overline{\text{erect-castle-1}} \vee \overline{\textbf{moat-0}} \vee \overline{\textbf{castle-0}} \vee \overline{\textbf{e}_3\textbf{-1}} \vee \textbf{moat-1}) \wedge$
10. $(\overline{\text{erect-castle-1}} \vee \overline{\textbf{moat-0}} \vee \overline{\textbf{castle-0}} \vee \textbf{e}_3\textbf{-1} \vee \overline{\textbf{moat-1}}) \wedge$
11. $(\overline{\text{erect-castle-1}} \vee \overline{\textbf{moat-0}} \vee \textbf{castle-0} \vee \overline{\textbf{castle-1}} \vee \textbf{moat-1}) \wedge$
12. $(\overline{\text{erect-castle-1}} \vee \overline{\textbf{moat-0}} \vee \textbf{castle-0} \vee \textbf{castle-1} \vee \overline{\textbf{e}_4\textbf{-1}} \vee \textbf{moat-1}) \wedge$
13. $(\overline{\text{erect-castle-1}} \vee \overline{\textbf{moat-0}} \vee \textbf{castle-0} \vee \textbf{castle-1} \vee \textbf{e}_4\textbf{-1} \vee \overline{\textbf{moat-1}}) \wedge$
14. $(\overline{\text{erect-castle-1}} \vee \textbf{moat-0} \vee \overline{\textbf{moat-1}}) \wedge$
15. $(\overline{\text{dig-moat-1}} \vee \overline{\textbf{castle-0}} \vee \textbf{castle-1}) \wedge$
16. $(\overline{\text{dig-moat-1}} \vee \textbf{castle-0} \vee \overline{\textbf{castle-1}}) \wedge$
17. $(\overline{\text{erect-castle-1}} \vee \overline{\textbf{castle-0}} \vee \textbf{castle-1}) \wedge$
18. $(\overline{\text{erect-castle-1}} \vee \textbf{castle-0} \vee \overline{\textbf{moat-0}} \vee \overline{\textbf{e}_1\textbf{-1}} \vee \textbf{castle-1}) \wedge$
19. $(\overline{\text{erect-castle-1}} \vee \textbf{castle-0} \vee \overline{\textbf{moat-0}} \vee \textbf{e}_1\textbf{-1} \vee \overline{\textbf{castle-1}}) \wedge$
20. $(\overline{\text{erect-castle-1}} \vee \textbf{castle-0} \vee \textbf{moat-0} \vee \overline{\textbf{e}_2\textbf{-1}} \vee \textbf{castle-1}) \wedge$
21. $(\overline{\text{erect-castle-1}} \vee \textbf{castle-0} \vee \textbf{moat-0} \vee \textbf{e}_2\textbf{-1} \vee \overline{\textbf{castle-1}})$

Figure 6.5: The CNF formula for a 1-step SAND-CASTLE-67 plan constrains the variable assignments.

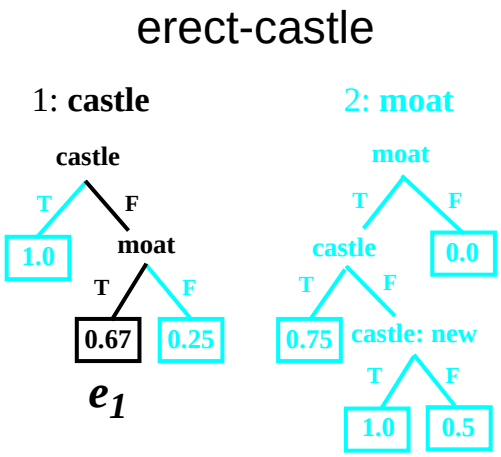


Figure 6.6: Each path through each decision tree generates clauses in the encoding.

taining all actions (one action must be taken) and $\binom{A}{2}$ clauses that enforce the requirement that two actions not be taken simultaneously.[2] In our example, the clauses generated are (dig-moat-1$\vee$erect-castle-1) and ($\overline{\text{dig-moat-1}}\vee\overline{\text{erect-castle-1}}$).

The third condition—effects of actions on propositions—generates one or two clauses for each path through each decision tree in each action. If the path is a *deterministic path* (the probability in the leaf is either 0.0 or 1.0), the path generates a single clause modeling the action's deterministic impact on the proposition given the circumstances described by that path. If the path is a *probabilistic path* (the probability in the leaf is strictly between 0.0 and 1.0), the path generates two clauses modeling the action's probabilistic impact on the proposition given the circumstances described by that path. An example will clarify this process.

Figure 6.6 shows the erect-castle decision tree. The highlighted path describes the impact of the erect-castle action on the **castle** proposition when there is no castle but there is a moat. Note that the probability in the leaf is strictly between

[2] "Exactly-one-of" clauses (not implemented yet), which specify that exactly one of the literals in the clause be `True`, would be a more efficient way of encoding mutual exclusivity of actions, generating only a single clause for each time step. See Chapter 8 for further discussion.

0.0 and 1.0; this path is a probabilistic path that will generate a chance variable associated with this probability ($\mathbf{e}_1$) and two clauses, one describing the impact of the action if the chance variable is `True` and one describing its impact if the chance variable is `False`. For the 1-step plan, this path generates the following two implications:

$$\text{erect-castle-1} \wedge \overline{\textbf{castle-0}} \wedge \textbf{moat-0} \wedge \mathbf{e}_1\text{-1} \rightarrow \textbf{castle-1}$$

$$\text{erect-castle-1} \wedge \overline{\textbf{castle-0}} \wedge \textbf{moat-0} \wedge \overline{\mathbf{e}_1\text{-1}} \rightarrow \overline{\textbf{castle-1}}$$

Note that a chance variable has the same time index as the action it modifies. Negating the antecedent and replacing the implication with a disjunction produces Clauses 18 and 19 (Figure 6.5):

$$\overline{\text{erect-castle-1}} \vee \textbf{castle-0} \vee \overline{\textbf{moat-0}} \vee \overline{\mathbf{e}_1\text{-1}} \vee \textbf{castle-1}$$

$$\overline{\text{erect-castle-1}} \vee \textbf{castle-0} \vee \overline{\textbf{moat-0}} \vee \mathbf{e}_1\text{-1} \vee \overline{\textbf{castle-1}}$$

Figure 6.5 shows the complete formula for a 1-step plan.

The total number of action-effect clauses is bounded by $2T\sum_{i=1}^{A} L_i$ where L_i is the number of leaves in the decision trees of action i, so the total number of clauses C is bounded by

$$2P + \left(\binom{A}{2} + 1\right) T + 2T \sum_{i=1}^{A} L_i,$$

which is a low-order polynomial in the size of the problem. The average clause size is dominated by the average path length of all the decision trees.

Note that by using a compact representation of a factored state space, such as the ST representation, and translating that representation directly into E-MAJSAT

form, we preserve the compactness of such a representation in our E-MAJSAT formula. The alternative—using a flat state space in which states are simply enumerated without regard to their characteristics, encoding states as propositions, and encoding in our clauses the impact of each action on each possible state—would be prohibitively expensive.

Also note that fixing a plan horizon does not prevent MAXPLAN from solving planning problems where the horizon is unknown. By using *iterative lengthening*, a process in which successive instances of the planning problem with increasing horizons are solved, the optimal plan horizon can be discovered dynamically. This, in fact, was the process used in the MAXPLAN/BURIDAN comparison described later. I have not yet determined the feasibility of *incremental iterative lengthening*, a more sophisticated approach, in which the current instance of the planning problem with horizon T is incrementally extended to the instance with horizon $T+1$ and earlier results are reused to help solve the extended problem.

6.4 Solving E-MAJSAT Problems

In this section I will describe the algorithm that solves the E-MAJSAT problem produced by the conversion described above in Section 6.3. The evalssat algorithm described in Section 5.2.1 was an obvious choice for an E-MAJSAT solver, but I decided to start with the most basic DPLL-style algorithm possible. The resulting algorithm, dpllssat, is essentially the evalssat algorithm without thresholding. Although I eventually modified dpllssat significantly, and eventually incorporated thresholding (in the contingent planners described in Chapter 7), describing the history of these modifications provides insight into the operation of the E-MAJSAT solver.

The DPLL algorithm systematically searches for a satisfying assignment. (Note that, although this is an exponential algorithm, recent work [52] indicates that DPLL can be modified to give a subexponential worst-case bound: c^n, where $c < 2.0$ and n is the number of variables). We can produce an algorithm to find all satisfying assignments by forcing the DPLL algorithm to continue until all assignments have been examined. This process can be envisioned as constructing and traversing an assignment tree. Recall from Section 3.4 that an assignment tree is a binary tree in which each node represents a choice (chance) variable and a partial assignment. A node can also be thought of as representing the subproblem remaining given the partial assignment represented by that node. Thus, the subtrees of a node can be thought of as representing the subproblems given the two possible assignments (outcomes) to the parent choice (chance) variable. To avoid evaluating an exponential number of assignments, it is critical to construct only as much of the tree as is necessary.

In the process of constructing this tree:

- an *active variable* is one that has not yet been assigned a truth value,
- an *active clause* is one that has not yet been satisfied by assigned variables, and
- the *current CNF subformula* is uniquely specified by the current set of active clauses and the set of variables that appear in those clauses.

The value of a subformula is defined in Section 5.1.1.

Like the evalssat algorithm, dpllssat takes advantage of the fact that partial assignments sometimes suffice to establish that the formula is already satisfied or unsatisfied. In addition, in order to prune further the number of assignments that must be calculated, dpllssat does the following, whenever possible:

- select an active variable that no longer appears in any active clause (*i.e.* it has become *irrelevant*) and assign `True` to that variable (irrelevant variable elimination, **IRR**),

- select a variable that appears alone in an active clause and assign the appropriate value (unit propagation, **UNIT**),[3] or

- select an active *choice* variable that appears in only one sense—always negated or always not negated—in all active clauses and assign the appropriate value (pure variable elimination, **PURE**).

Irrelevant variable elimination is implicit in the evalssat algorithm (*i.e.* there is no point in splitting on a variable that no longer appears in an active clause in the formula), but dpllssat makes this "heuristic" explicit. This action is justified since setting to `True` a variable v that no longer appears in an active clause will not affect the satisfying assignments of the current subformula and will result in the same value, or probability of satisfaction, as if v had been set to `False`. I have already discussed and justified unit propagation and pure variable elimination as elements of the algorithm evalssat (Section 5.2.1).

When there are no more irrelevant variables, unit clauses, or pure variables, the algorithm dpllssat must select and split on an active variable. Note that, in solving an E-MAJSAT problem, the algorithm must always give precedence to choice variables, since splitting on a chance variable when any choice variables remain unassigned would allow the solver to select different values for those choice variables contingent on the random value assigned to that chance variable. Since the chance variables encode the uncertainty in the environment as the plan is

[3]Note that for chance variable i with probability π_i, this decreases the success probability by a factor of π_i or $1 - \pi_i$.

being executed, this is equivalent to getting a glimpse of the future while constructing the plan. If the algorithm splits on a choice variable, it returns the maximum of the two satisfaction probabilities obtained by assigning `True` to the variable and recurring or assigning `False` to the variable and recurring. If it splits on a chance variable, it returns the average of these two probabilities weighted by the probabilities that that variable is `True` or `False`.

The splitting heuristic used is as critical to the efficiency of the E-MAJSAT solver as it is to the efficiency of SAT solvers [50]. In Section 5.2.2, I reported some empirical results regarding the efficiency of various splitting heuristics in the evalssat algorithm. These tests indicated that the impact of the splitting heuristic used increased with the size of blocks of similarly quantified variables in the problem. This is not surprising since contiguous, similarly quantified variables commute—they can be dealt with in any order—and large blocks of such variables provide more opportunities for a splitting heuristic to reorder variables advantageously. The results reported in Section 5.2.2 indicated that two heuristics were particularly beneficial in solving randomly generated SSAT problems with a block size of at least six:

- SATS_MOST (find the literal that satisfies the most clauses in the current formula and choose that variable), and
- MOMS (choose the variable that has "Maximum Occurrences in clauses of Minimum Size").

Although MOMS seemed to be a slightly more efficient heuristic in terms of the size of the assignment tree generated, SATS_MOST is computationally much simpler. For this reason, I decided to use the SATS_MOST heuristic in my tests of heuristics for the E-MAJSAT solver.

In addition to the SATS_MOST heuristic, I tested a random heuristic (RAND) and a new heuristic based on the time indexing of the variables (TIME_ORDERED). Although the random heuristic performed poorly in the tests reported in Section 5.2.2, it was possible that it would perform better on problems with structure—the plan-generated E-MAJSAT problems—than on randomly generated problems. The three heuristics I tested on E-MAJSAT plan encodings were:

- RAND (choose the next active variable from an initially random ordering of the variables, giving precedence to choice variables),
- SATS_MOST (find the literal that satisfies the most clauses in the current formula and choose that variable), and
- TIME_ORDERED (choose the active variable that would appear earliest in the plan, exhausting all choice variables first).

The graph in Figure 6.7 compares the performance of these three heuristics on a plan evaluation problem in the SAND-CASTLE-67 domain: given a plan that alternates dig-moat and erect-castle (with erect-castle as the last action), evaluate its probability of success. Although they all scale exponentially with the length of the plan, the TIME_ORDERED splitting heuristic, given a fixed amount of time, is able to evaluate longer plans than the SATS_MOST heuristic, and the SATS_MOST heuristic can evaluate longer plans than the RAND heuristic. Empirically, this appears to be because TIME_ORDERED splitting increases the opportunities for the application of heuristics, particularly unit propagation.

To verify the importance of the individual elements of DPLL, I compared performance on the same plan-evaluation problem of the following:

- full DPLL with time-ordered splitting,

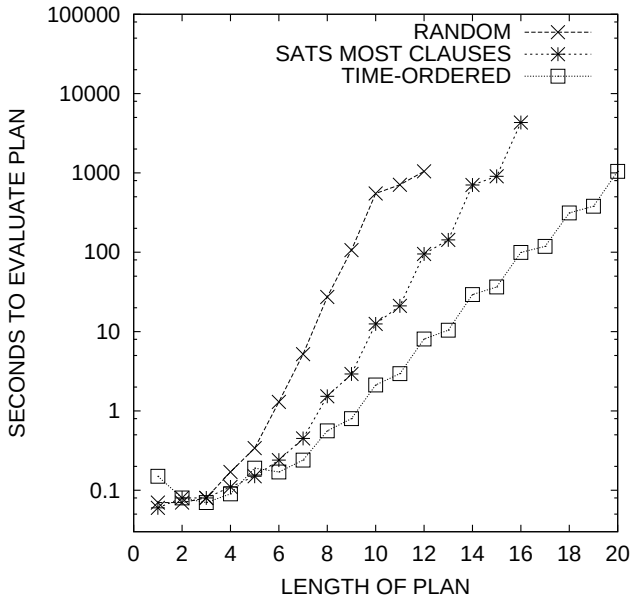


Figure 6.7: For DPLL on plan evaluation, the time-ordered splitting heuristic appears to work best.

- **PURE** and **UNIT** with time-ordered splitting,
- **UNIT** with time-ordered splitting, and
- time-ordered splitting alone.

As the graph in Figure 6.8 shows, removal of these DPLL elements severely degrades performance, preventing feasible evaluation of plans longer than about 4 steps.

This exponential behavior was surprising given the fact that a simple dynamic-programming algorithm (called "ENUM" in the next section) scales linearly with the plan horizon for plan evaluation. This observation led me to incorporate dynamic programming into the solver in the form of memoization: the algorithm caches the values of solved subformulas for possible use later. Memoization greatly extends the size of the plan the algorithm can feasibly evaluate (Figure 6.9). With memoization, the algorithm can evaluate a 110-step plan in

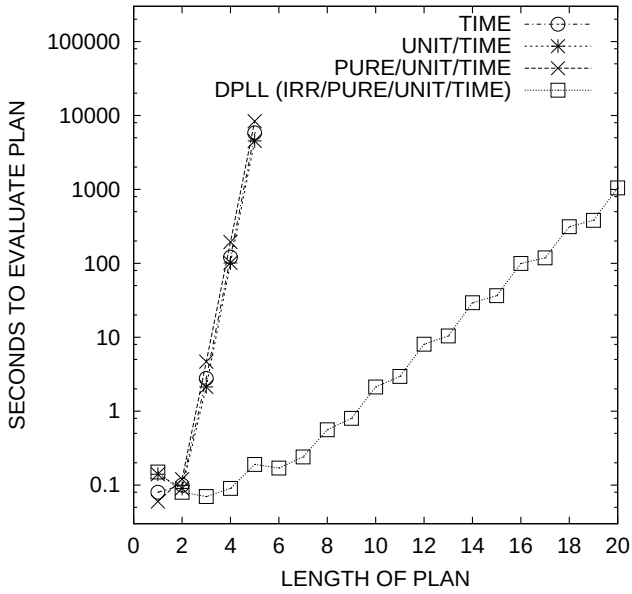


Figure 6.8: Full DPLL on plan evaluation performs better than DPLL with various elements removed.

less time than it took to evaluate an 18-step plan previously. These results are not surprising since the combination of time-ordered splitting and memoization essentially reproduces the dynamic programming solution of the planning problem. Robson [97] has used memoization to speed up the calculation of maximum independent sets, and has provided some theoretical results with respect to this approach.

The behavior of the algorithm suggested, however, that DPLL was hindering efficient subproblem reuse, so I gradually removed the DPLL elements as we had done before. This time, I found that removing DPLL elements greatly *improved* performance. As the graph in Figure 6.10 indicates, best performance is achieved with unit propagation and time-ordered splitting, or time-ordered splitting alone. Tests on other problems suggest that unit propagation and time-ordered splitting provide the best performance.

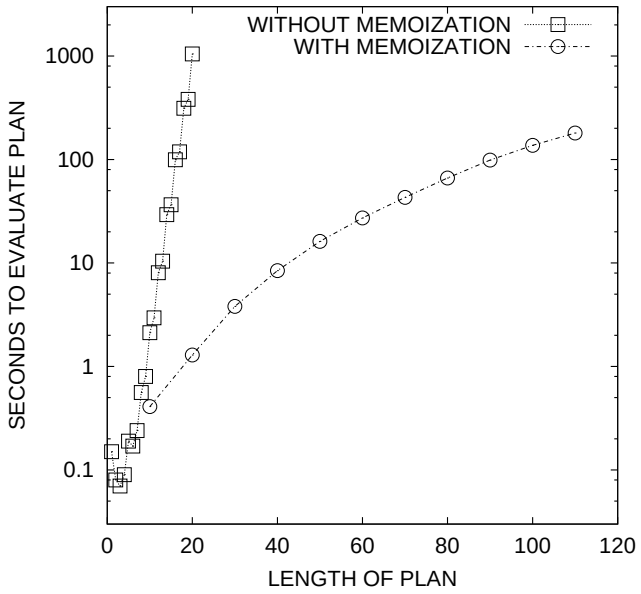


Figure 6.9: Full DPLL on plan evaluation runs faster with memoization.

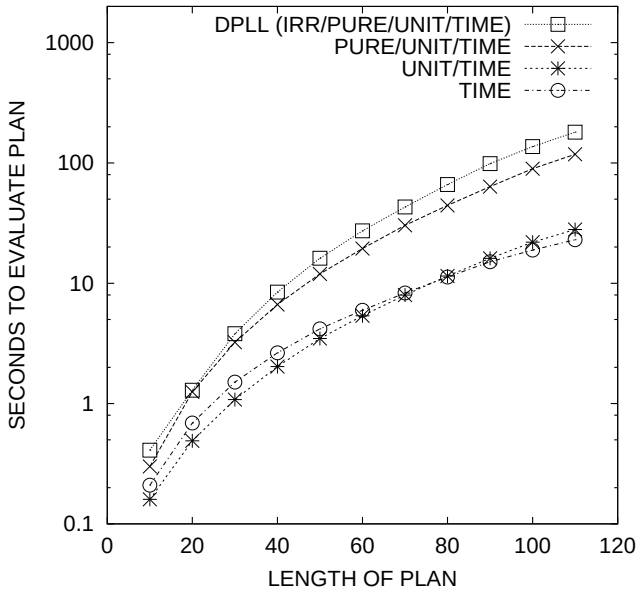


Figure 6.10: Full DPLL on plan evaluation with memoization performs worse than DPLL with various elements removed.

Plan Length	Plan (D = Dig-Moat, E = Erect-Castle)	Probability of Plan Success	Solution Time (CPU seconds, average of 5 runs)
1	E	0.250000	0.01
2	D-E	0.460000	0.01
3	D-E-E	0.629650	0.02
4	D-E-E-E	0.727955	0.03
5	D-E-D-E-E	0.815863	0.05
6	D-E-E-D-E-E	0.865457	0.10
7	D-E-D-E-D-E-E	0.908290	0.21
8	D-E-D-E-E-D-E-E	0.933433	0.49
9	D-E-D-E-D-E-D-E-E	0.954304	1.17
10	D-E-D-E-E-D-E-D-E-E	0.966887	2.68

Table 6.1: Optimal plans found by MAXPLAN in the SAND-CASTLE-67 domain exhibit a rich structure.

6.5 Results

Having identified a set of promising algorithmic components, I tested the resulting system on the full plan-generation problem in SAND-CASTLE-67 for plan horizons ranging from 1 to 10. The optimal one-step plan (the solution to the example problem in the previous section) is erect-castle. Larger optimal plans found by MAXPLAN exhibit a rich structure: beyond length 3, the optimal plan of length i is not a subplan of the optimal plan of length $i+1$ (Table 6.1). The optimal 10-step plan is D-E-D-E-E-D-E-D-E-E, where D indicates dig-moat and E indicates erect-castle. This plan succeeds with probability 0.966887 and MAXPLAN finds this plan in 2.7 seconds on a 400 MHz Dell Dimension XPS R400 with 128 Mbytes of RAM, running SunOS-5.7.

I tested MAXPLAN on four other domains. In the SLIPPERY-GRIPPER domain [65], an agent with a possibly wet (probability 0.30) gripper must paint a

block and then pick it up, ending up with a clean gripper. The pick-up action succeeds with probability 0.95 if the gripper is dry, but only with probability 0.50 if it is wet. There is a drying action that succeeds with probability 0.80. Painting always succeeds, but dirties the gripper with a probability that depends on whether the gripper is holding the block (1.0, if holding the block, 0.10 if not). A dirty gripper can be cleaned, and this action succeeds with probability 0.85.

The MEDICAL-SEQUENCE domain is my extension of the "drink-or-die" problem described in the original SENSORY GRAPHPLAN (SGP) paper [113]. In the original problem, a patient may be infected. If so, she must be medicated in order to eradicate the infection. If she takes the medication without being hydrated, however, she will die. Since SGP does not handle probabilistic initial conditions or probabilistic actions, the uncertain initial conditions are expressed as *possible worlds*, and actions are either deterministic (the drink action always hydrates the patient) or conditional (the effects are deterministic once the conditions under which the action is executed are known). For example, if the patient is infected and hydrated, the medicate action results in the patient becoming not infected, whereas if the patient is infected and not hydrated, the action results in the patient becoming not infected, but dead. In the MEDICAL-SEQUENCE extension of this problem, the patient must also eat after taking the medication in order to avoid death. In addition, the actions are probabilistic. The drink, eat, and medicate actions succeed with probability 0.90, 0.70, and 0.60, respectively.

The COFFEE-ROBOT-BLIND domain is an unobservable version of a problem from the POMDP literature [16]. In this problem, a robot must go to the cafeteria to fetch coffee for its user. It may be raining (probability 0.50), so the robot must take its umbrella. In addition, all the robot's actions may fail. Going to the cafeteria (or returning to the office) succeeds with probability 0.90, buying

coffee succeeds with probability 0.70, delivering coffee to the user succeeds with probability 0.80, and getting the umbrella succeeds with probability 0.75.

The DISARMING-MULTIPLE-BOMBS domain is an extension of the BOMB-IN-TOILET problems [65], in which varying number of packages possibly containing bombs must be disarmed by dunking them in toilets that may clog but can be unclogged by flushing. In the DISARMING-MULTIPLE-BOMBS domain, there is some number of packages, any or all of which contain bombs. The disarm action allows the agent to disarm all the bombs at once, but it requires that the agent know where the bombs are. The scan action gives the agent this knowledge.

Results for the SAND-CASTLE-67, SLIPPERY-GRIPPER, MEDICAL-SEQUENCE, and COFFEE-ROBOT-BLIND domains are reported in Figure 6.11. These tests were done on a 400 MHz Dell Dimension XPS R400 with 128 Mbytes of RAM, running SunOS-5.7. (Results for the DISARMING-MULTIPLE-BOMBS domain are reported in Section 6.6.) The graph in Figure 6.11 plots log solution time against increasing plan horizon and indicates growth in solution time exponential in the plan horizon. SAND-CASTLE-67 is the smallest domain (2 fluents and 2 actions) and shows slower growth. The SLIPPERY-GRIPPER, MEDICAL-SEQUENCE, and COFFEE-ROBOT-BLIND domains are larger than the SAND-CASTLE-67 domain: SLIPPERY-GRIPPER has 4 fluents and 4 actions, and MEDICAL-SEQUENCE and COFFEE-ROBOT-BLIND each have 6 fluents and 4 actions. The larger domain size is reflected not only in longer run times, but also in the fact that memoization in SLIPPERY-GRIPPER, MEDICAL-SEQUENCE, and COFFEE-ROBOT-BLIND exhausts the available memory before a plan length of 10 is reached. In the SAND-CASTLE-67 domain, however, MAXPLAN can find a 15-step plan before exhausting memory (see Appendix A). I developed a caching technique to mitigate this memory problem; for a full discussion, see Appendix A.

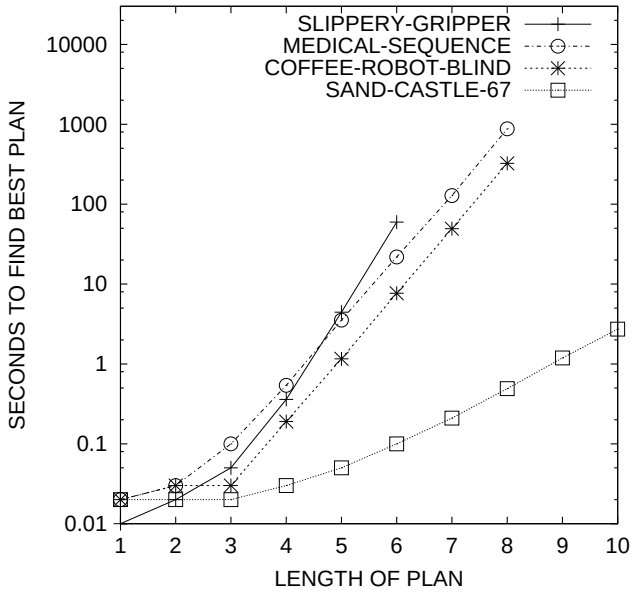


Figure 6.11: MAXPLAN performs similarly on three domains larger than SAND-CASTLE-67.

6.6 Comparison to Other Planning Techniques

I compared MAXPLAN to three other planning techniques:

- BURIDAN [65], described in Section 2.2.1,
- Plan enumeration with dynamic programming for plan evaluation (ENUM), and
- Dynamic programming ("Lark" pruning) to solve the corresponding finite-horizon POMDP (POMDP-DP) [20, 115].

These comparisons were motivated by a desire to compare MAXPLAN to other algorithms that can determine the best plan in a probabilistic domain, including domains in which no plan is certain of succeeding (thus ruling out lower complexity minimax planners).

A comparison of MAXPLAN and BURIDAN on SAND-CASTLE-67 is complicated by the fact that BURIDAN's performance varies depending on which of several available plan-evaluation methods it uses. As noted in Section 2.2.1, there exist problems for which each method is best and it is not possible, in general, to determine beforehand which method will provide the best performance. For this comparison, I will report the best results obtained.

BURIDAN operates by looking for a plan whose probability of success exceeds some user-specified threshold. For this comparison, I provided BURIDAN with probability thresholds that forced it to find increasingly longer plans. For example, a threshold of 0.20 is satisfied by the 1-step plan erect-castle, whereas a threshold of 0.70 requires the 4-step plan dig-moat, erect-castle, erect-castle, erect-castle. The results are reported in Table 6.2. BURIDAN is able to find the 1-step plan in only 0.05 CPU seconds, but the time and space needed to find longer plans grows exponentially and BURIDAN runs out of memory after about 875 CPU seconds searching for the 4-step plan that would exceed a probability of 0.70. In contrast, MAXPLAN is able to find this 4-step plan in 0.28 CPU seconds. Note that this is a composite time arrived at by using MAXPLAN to find optimal plans of increasing length until a plan is found that exceeds the threshold, and then adding together the run times for all the plans found. (These tests, and all tests in this section, were run on a 143 MHz Sun UltraSparc with 128 Mbytes of RAM, running SunOS-5.7.)

I next compared the scaling behavior of MAXPLAN—rather than its performance on a single problem—to that of ENUM and POMDP-DP. Since ENUM and POMDP-DP require that all 2^P problem states be explicitly enumerated, these methods necessarily scale exponentially in the number of propositions needed to describe a state even in the *best* case (in the worst case, all methods are expected

PROBABILITY THRESHOLD	PLAN	CPU SECS	
		MAXPLAN	BURIDAN
0.20	ERECT	0.07	0.05
0.45	DIG-ERECT	0.13	0.53
0.60	DIG-ERECT-ERECT	0.20	52.59
0.70	DIG-ERECT-ERECT-ERECT	0.28	>876.43

Table 6.2: MAXPLAN outperforms BURIDAN on SAND-CASTLE-67.

to be exponential). ENUM necessarily scales exponentially in the plan horizon as well, since all A^T plans are evaluated.

MAXPLAN tries to explore only as much of the problem space as is necessary to determine the best plan, and does not necessarily scale exponentially in either the size of the state space or the plan horizon. This effort is not always successful; Figure 6.12 shows performance results for MAXPLAN, ENUM, and POMDP-DP as the horizon increases in SAND-CASTLE-67. ENUM, as expected, scales exponentially, but so does MAXPLAN. We might expect POMDP-DP, which can solve arbitrary POMDPs, to perform poorly here. POMDP-DP, remarkably, scales linearly as the horizon increases. This linear scaling is due to the fact that the class of possibly useful policies (specifying what action to take for each belief state of the agent) is particularly simple for the SAND-CASTLE-67 domain, and POMDP-DP operates by iterating over sets of policies. In the SAND-CASTLE-67 domain, POMDP-DP reduces the set of active (possible useful) policies to fewer than four different policies after only a few iterations. This is true for all subsequent iterations, so the set of policies that are active at iteration i can be used to create the set of policies active at iteration $i+1$ in constant time.

We see much different behavior in the DISARMING-MULTIPLE-BOMBS problem, a problem that allows us to enlarge the state space without increasing the

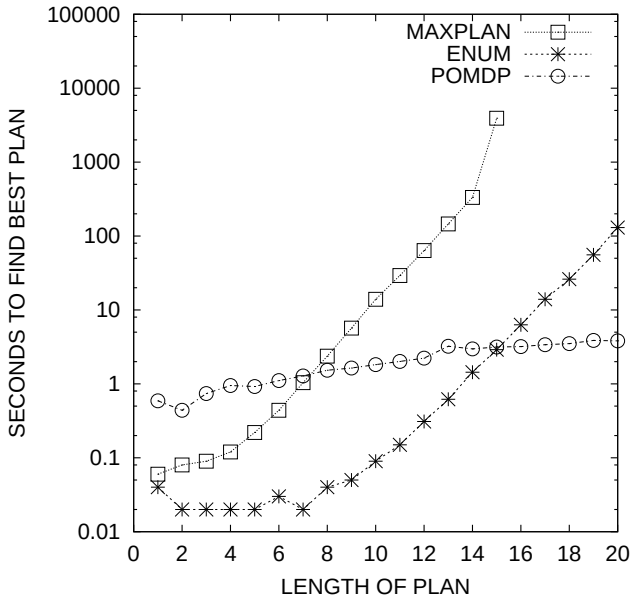


Figure 6.12: MAXPLAN solves SAND-CASTLE-67 more slowly than two dynamic programming-based approaches as plan length is increased.

length of the optimal plan. Recall that in the DISARMING-MULTIPLE-BOMBS domain the agent is presented with a number of packages, each of which may contain a bomb. The agent does not have time to scan each package separately, and so must scan them *en masse* to determine which ones contain bombs. With this knowledge, the agent can disarm them simultaneously by dunking all the packages that contain bombs in a large pool. (The agent must avoid soaking packages that do not contain bombs.) Thus, even as the state space scales exponentially with the number of packages, there is always a successful 2-step plan. Figure 6.13 shows performance results for MAXPLAN, ENUM, and POMDP-DP as the number of packages is increased. Both ENUM and POMDP-DP scale exponentially, while MAXPLAN remains constant over the same range, solving each problem in less than 0.1 seconds. Although DISARMING-MULTIPLE-BOMBS is a silly problem, it makes a valid point regarding the abilities of MAXPLAN, ENUM, and POMDP-DP.

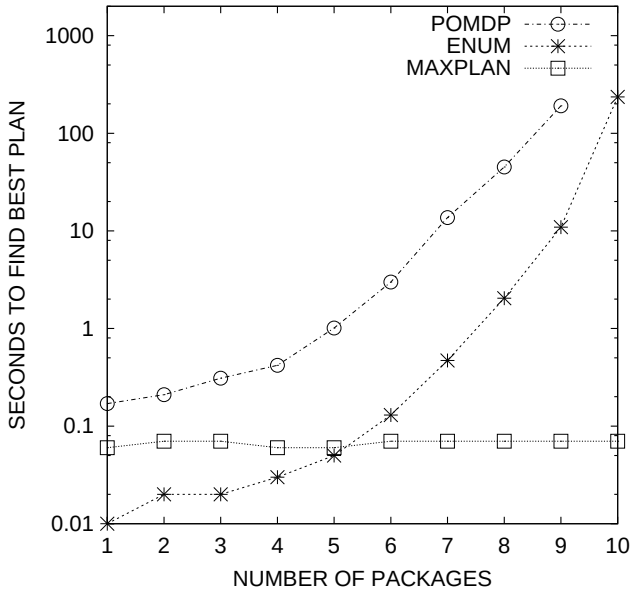


Figure 6.13: MAXPLAN solves DISARMING-MULTIPLE-BOMBS faster than two dynamic programming-based approaches as the number of packages is increased.

6.7 Summary

In this chapter, I described how a probabilistic planning problem can be solved by encoding it as an E-MAJSAT problem and solving that problem instead. I showed that, in spite of the potentially large complexity gap between SAT (NP-complete) and E-MAJSAT (NP^{PP}-complete), insights gained in developing solution techniques for SAT can be successfully exploited in the more general, probabilistic setting of E-MAJSAT.

MAXPLAN has a severe limitation. Because it produces noncontingent plans, it is limited to problems for which the optimal plan is a noncontingent sequence of actions. This is much too restrictive to allow MAXPLAN to be applied to real-world planning problems. In this next chapter, I remove this restriction by showing how the probabilistic-planning-as-stochastic-satisfiability paradigm can be extended to support contingent planning in partially observable stochastic

domains.

Chapter 7

Contingent Planning

Portions of this chapter have appeared in an earlier paper: "Contingent planning under uncertainty via stochastic satisfiability" [77] with Littman.

When planning under uncertainty, any information about the state of the world is precious. A *contingent plan* is one that can make action choices contingent on such information. In this chapter, I present an implemented framework for contingent planning under uncertainty using stochastic satisfiability.

In both of the contingent planners I developed—C-MAXPLAN and ZANDER—a subset of the state variables is declared *observable*, meaning that any action can be made contingent on any of these variables. This scheme is sufficiently expressive to allow a domain designer to make a domain fully observable, unobservable, or to have observations depend on actions and states in probabilistic ways.

Recall that MAXPLAN uses a two-phase approach: a problem conversion unit converts the planning problem to an E-MAJSAT problem, and an E-MAJSAT solver solves the E-MAJSAT encoding of the planning problem. This framework provides two ways to extend the paradigm to support contingent planning. The E-MAJSAT encoding can be changed so that it can express contingent planning problems, or the solution method can be changed so as to be able to extract contingent plans from the encoding (see Figure 7.1).

C-MAXPLAN takes the first approach. C-MAXPLAN encodes the contingent planning problem as an E-MAJSAT instance. In order to encode a contingent planning problem, new types of variables and clauses are introduced and, al-

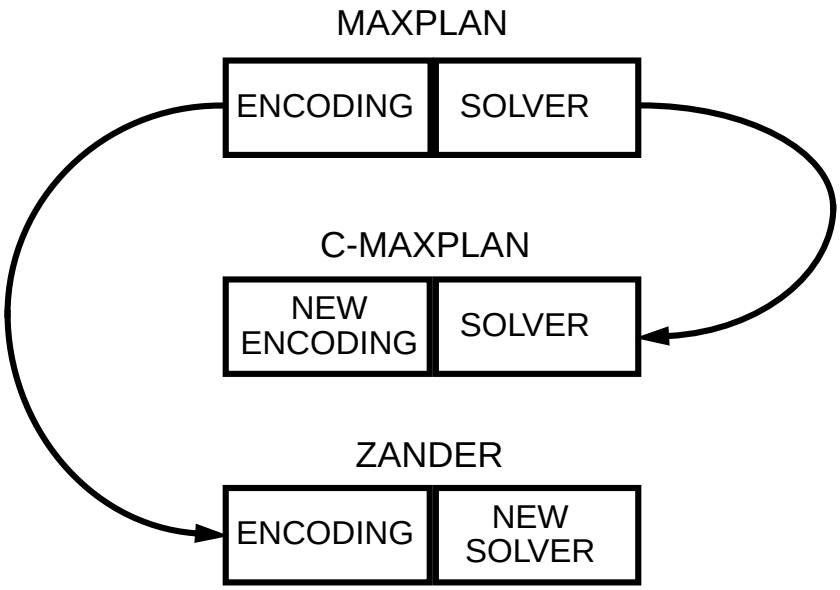


Figure 7.1: Two approaches to extending MAXPLAN to support contingent planning.

though the encoding is still an E-MAJSAT instance, it is substantially different from a MAXPLAN encoding. Because the C-MAXPLAN encoding is an E-MAJSAT instance, however, C-MAXPLAN can use the same algorithm as MAXPLAN to solve the E-MAJSAT-encoded problem.

ZANDER takes the second approach, although the problem encoding is somewhat changed. The variables and clauses are substantially the same as a MAXPLAN encoding. The main difference is the presence of clauses that describe how observations are produced. Note that this allows for observations to be noisy, allowing any degree of observability. The other difference—which has an impact on the solution process rather than the encoding—is the quantifier ordering. A ZANDER encoding models observations as chance variables and interleaves choice and chance variables so that values for choice variables encoding actions can be chosen contingent on the values of earlier chance variables encoding observations. This interleaving means that ZANDER produces an SSAT instance rather than an

E-MAJSAT instance, and requires a modified solution process. ZANDER encodings are substantially more compact than C-MAXPLAN encodings, and this appears to more than offset the fact that ZANDER's SSAT encodings have a higher complexity (SSAT is PSPACE-complete) than C-MAXPLAN's E-MAJSAT encodings (E-MAJSAT is NP^{PP}-complete). In fact, the size of the C-MAXPLAN encodings cripples C-MAXPLAN's performance severely, relative to that of ZANDER, and I will not describe C-MAXPLAN in detail in this dissertation.

Both C-MAXPLAN and ZANDER produce more complex plans than MAXPLAN. While MAXPLAN was capable of producing only simple sequences of actions (totally ordered plans (Figure 6.2(a)), both C-MAXPLAN and ZANDER are capable of producing plans that branch (acyclic, contingent plans Figure 6.2(b)). Actions in such plans can be made contingent on observations of the environment, and represent a significant step toward a planner that can produce any type of plan that is required, including plans in which actions can be repeated an unspecified number of times (looping plans Figure 6.2(c)).

7.1 Representing Probabilistic Contingent Planning Problems

The contingent planners I developed work on partially observable probabilistic propositional planning domains. The representation described in Section 6.1 is augmented by declaring a subset of the set of propositions to be *observable propositions*. Each observable proposition has, as its basis, a proposition that represents the actual status of the thing being observed. (Note that although values are assigned to observable propositions in the initial state, no action at time 1 makes use of these propositions in its decision trees, since there are no

valid observations at time 0.)

The planning task is to find a plan that selects an action for each step t as a function of the value of observable propositions for steps before t. We want to find a plan that maximizes (or exceeds a user-specified threshold for) the probability of reaching a goal state.

7.2 Example Domain

Consider a simple domain based on the TIGER problem [54]. The domain consists of four propositions: **tiger-behind-left-door**, **dead**, **rewarded** and **hear-tiger-behind-left-door**, the last of which is observable. In the initial state, **tiger-behind-left-door** is `True` with probability 0.5, **dead** is `False`, **rewarded** is `False`, and **hear-tiger-behind-left-door** is `False` (although irrelevant). The goal states are specified by the partial assignment (**rewarded**, (not **dead**)). The three actions are listen-for-tiger, open-left-door, and open-right-door (Figure 7.2). Actions open-left-door and open-right-door make **reward** `True`, as long as the tiger is not behind that door (we assume the tiger is behind the right door if **tiger-behind-left-door** is `False`). Since **tiger-behind-left-door** is not observable, the listen action becomes important; it causes the observable **hear-tiger-behind-left-door** proposition to become equal to **tiger-behind-left-door** with probability 0.85 (and its negation otherwise). By listening multiple times, it becomes possible to determine the likely location of the tiger.

7.3 ZANDER

ZANDER encodes a planning problem as an SSAT instance. Choice variables and chance variables are interleaved in a way that allows action choices, encoded

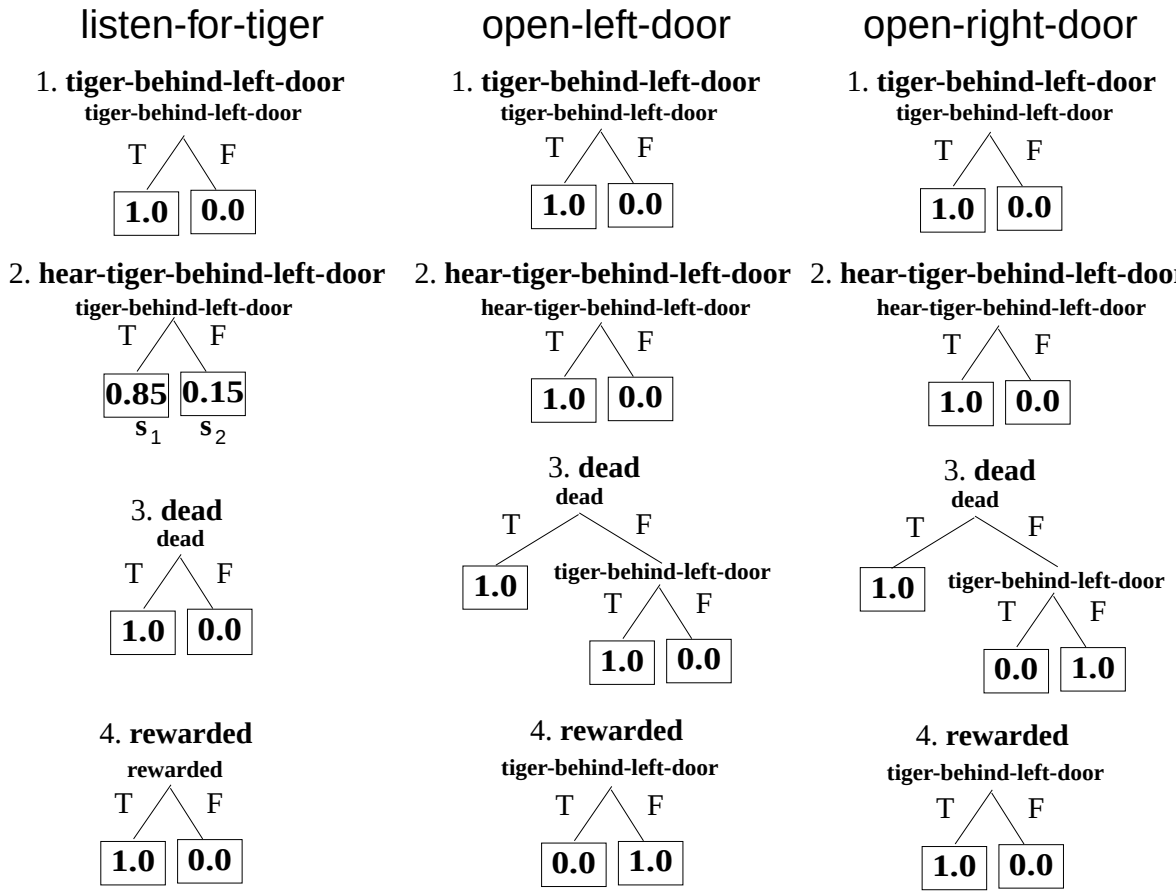


Figure 7.2: The effects of the actions in the TIGER problem are represented by a set of decision trees.

by choice variables, to be made contingent on observations (encoded by chance variables). The solution of the SSAT problem is a tree, each branch of which describes a plan execution history in the optimal contingent plan.

7.3.1 Encoding Contingent Planning Problems as SSAT Problems

In an SSAT formula, the value of an existential variable x can be selected on the basis of the values of all the variables to x's left in the quantifier sequence. Thus, viewing an existential variable as an action choice, the value of all "earlier" variables in the quantifier sequence are observable at the time x's value is selected. So, the choice represented by x is contingent on the earlier variables. This allows one to map contingent planning problems to stochastic satisfiability by encoding the contingent plan in the decision tree induced by the quantifier ordering associated with the SSAT formula. By alternating blocks of existential variables that encode actions and blocks of randomized variables that encode observations, one can condition the value chosen for any action variable on the possible values for all the observation variables that appear earlier in the ordering. This is in contrast to MAXPLAN and C-MAXPLAN encodings, in which all the choice variables occur before all the chance variables. A generic SSAT encoding for contingent plans appears in Figure 7.3. Note that this approach is agnostic as to the structure of the plan; the type of plan returned is algorithm dependent. ZANDER constructs tree-structured proofs; these correspond to tree-structured plans that contain a branch for each observable variable. Other SSAT solvers could produce DAG-structured, subroutine-structured, or value-function-based plans.

The quantifiers naturally fall into three segments: a plan-execution history, the domain uncertainty, and the result of the plan-execution history given the do-

$$\overbrace{\exists x_{1,1}, \ldots, \exists x_{1,c_1}}^{\text{first action}} \overbrace{\text{ꓤ} w_{1,1}, \ldots, \text{ꓤ} w_{1,c_2}}^{\text{first observation}} \ldots \overbrace{\text{ꓤ} w_{n-1,1}, \ldots, \text{ꓤ} w_{n-1,c_2}}^{\text{last observation}} \overbrace{\exists x_{n,1}, \ldots, \exists x_{n,c_1}}^{\text{last action}}$$

$$\overbrace{\text{ꓤ}^{\rho_1} z_1, \ldots, \text{ꓤ}^{\rho_{c_4}} z_{c_4}}^{\substack{\text{domain} \\ \text{uncertainty}}} \overbrace{\exists y_1, \ldots, \exists y_{c_3}}^{\substack{\text{states} \\ \text{encountered}}}$$

$$\begin{aligned}(E[&\text{(initial/goal conditions } (y,z)\text{-clauses)} \\ &\text{(action exclusion } (x)\text{-clauses)} \\ &\text{(action outcome } (w,x,y,z)\text{-clauses)}] \geq \theta).\end{aligned}$$

c_1 = number of variables it takes to specify a single action (the number of actions),
c_2 = number of variables it takes to specify a single observation,
c_3 = number of state variables (one for each proposition at each time step), and
c_4 = number of chance variables (one for each possible stochastic outcome at each time step).

Figure 7.3: A generic SSAT encoding of a contingent planning problem.

main uncertainty. The plan-execution-history segment is an alternating sequence of choice-variable blocks (one for each action choice) and chance-variable blocks (one for each set of possible observations at a time step). This segment begins with the action block for the first (noncontingent) action choice and ends with the action block for the last action choice. (Although there may be observation variables that are set by the last action, they are irrelevant to the success of the plan.) The action choice encoded in each action block can, thus, be conditioned on the values of all the preceding observation variables in all the observation-variable blocks to the left of that action-variable block. In the TIGER problem, each action variable block would be composed of the three possible actions—listen-for-tiger, open-left-door, and open-right-door—and each observation variable block would be composed of the single variable **hear-tiger-behind-left-door**.

This means that the values of the variables in the second action-variable block (*i.e.* the action chosen) can be conditioned on the value of **hear-tiger-behind-left-door** in the observation-variable block immediately preceding them; *i.e.* the planner can specify one action if the tiger is heard behind the left door, and a different action otherwise.

A naive approach might try to accomplish this action conditioning by putting the variables being observed in the observation-variable blocks (*e.g.* tiger-behind-left-door instead of hear-tiger-behind-left-door). This would be both incorrect and limiting. It is incorrect because an observation action cannot be allowed to set the status of a chance variable that encodes uncertainty in the environment. This would be equivalent to an observation forcing a certain reality. And, even if conditioning actions on the observed variables could somehow be made to work, it would be severely limiting, since conditioning on an observed variable—the underlying reality—would make it impossible to model noisy, imperfect observations.

The domain uncertainty segment is a single block containing all the chance variables that modulate the impact of the actions on the observation and state variables. These variables are associated with randomized quantifiers; when the algorithm considers a variable that represents uncertainty in the environment, it needs to take the probability weighted average of the success probabilities associated with the two possible settings of the variable. In the TIGER problem, there would be a chance variable (probability = 0.85) associated with the outcome of each **listen-for-tiger** action.

The result segment is a single block containing all the non-observation state variables. These variables are associated with existential quantifiers, indicating that the algorithm can choose the best truth setting for each variable. In reality,

all such "choices" are forced by the settings of the action variables in the first segment and the chance variables in the second segment. If these forced choices are compatible, then the preceding plan-execution history is possible and has a non-zero probability of achieving the goals. Otherwise, either the plan-execution history is impossible, given the effects of the actions, or it has a zero probability of achieving the goals.

The probability of satisfaction, or value, of ϕ (under quantifier order Q), $val(\phi, Q)$, where ϕ and Q are an SSAT encoding of a contingent planning problem is defined by induction on the number of quantifiers, and is similar to the value of an SSAT formula defined in Section 5.1.1. Let x_1 be the variable associated with the outermost quantifier. Then:

1. if ϕ contains an empty clause, then $val(\phi, Q) = 0.0$;
2. if ϕ contains no clauses then $val(\phi, Q) = 1.0$;
3. if $Q(x_1) = \exists$, then $val(\phi, Q) = \max(val(\phi\lceil_{x_1=0}, Q), val(\phi\lceil_{x_1=1}, Q))$;
4. if $Q(x_1) = \text{R}^{\pi}$ and x_1 is not an observation variable, then $val(\phi, Q) = (val(\phi\lceil_{x_1=0}, Q) \times (1.0 - \pi) + val(\phi\lceil_{x_1=1}, Q) \times \pi)$;
5. if $Q(x_1) = \text{R}^{0.5}$ and x_1 is an observation variable, then $val(\phi, Q) = val(\phi\lceil_{x_1=0}, Q) + val(\phi\lceil_{x_1=1}, Q)$.

The only difference between these rules and those stated in Section 5.1.1 for a general SSAT formula, is the addition of Rule 5 to handle chance (randomized) variables encoding observations. This rule states that the value of a formula whose outermost quantifier is a chance variable encoding an observation is the *sum* of the value of the formula if that variable is assigned the value `True` and the

value of the formula if that variable is assigned the value `False`, rather than the probability weighted average of these two values (as in Rule 4, for the value of a formula whose outermost quantifier is a chance variable that does *not* encode an observation). This special treatment of some chance variables requires some explanation.

The chance variables representing observations in the plan-execution history are used only to mark possible branch points in the plan, and not to encode the probability of actually making that observation. (The actual probability of the observation being `True` is encoded by a chance variable that appears in the domain uncertainty segment.) For example, in the 2-step TIGER problem, there is a choice-variable block representing a choice between actions **listen-for-tiger**, **open-left-door**, and **open-right-door** at time step 1, followed by a single observation chance variable hear-tiger-behind-left-door, followed by another choice-variable block, representing a choice between actions **listen-for-tiger**, **open-left-door**, and **open-right-door** at time step 2. The function of chance variable hear-tiger-behind-left-door is to allow the solver to choose one action at time step 2 if hear-tiger-behind-left-door is `True` and a different action if hear-tiger-behind-left-door is `False`.

In order to calculate the correct probability of success of such a branching plan, the algorithm needs to sum the success probabilities over all branches. Making hear-tiger-behind-left-door a chance variable (instead of a choice variable) allows one to combine the success probabilities of the two branches, but, as defined for a standard SSAT problem (Rule 4 above), chance variables must combine the success probabilities associated with their two values (`True`/`False`) by taking the probability weighted average of these success probabilities, instead of the sum. To get around this, I associate a probability of 0.5 with the chance variable hear-

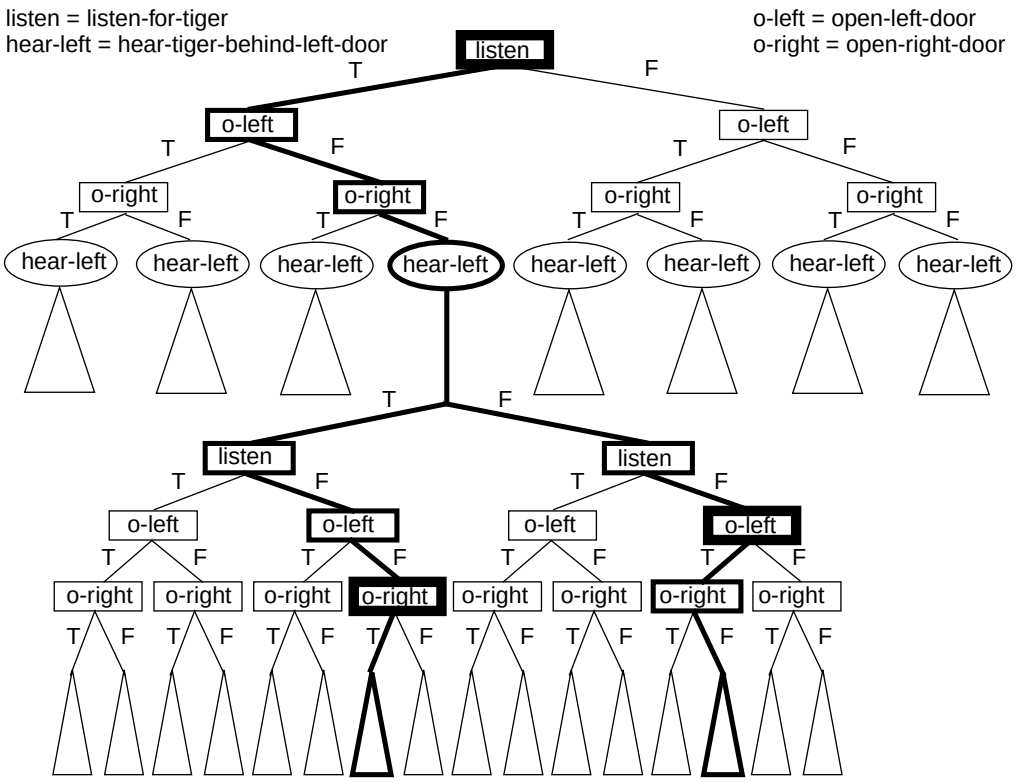


Figure 7.4: ZANDER selects an optimal subtree.

tiger-behind-left-door and adjust the calculated probability of success upward by a factor of 2. This is equivalent to summing the success probabilities of the two branches.

7.3.2 Algorithm Description

In contrast to MAXPLAN and C-MAXPLAN, which find a single optimal choice-variable assignment, ZANDER finds an assignment *tree* that specifies the optimal choice-variable assignment given all possible settings of the observation variables. Note that I am no longer limiting the size of the plan to be polynomial in the size of the problem; the assignment tree can be exponential in the size of the problem. The most basic variant of the solver follows the variable ordering exactly, constructing a binary tree of all possible assignments. Figure 7.4 depicts

such a tree; each node contains a variable under consideration, and each path through the tree describes a plan-execution history, an instantiation of the domain uncertainty, and a possible setting of the state variables. The tree shows the first seven variables in the ordering for the 2-step TIGER problem: the three choice variables encoding the action at time step 1—**listen-for-tiger**-1, **open-left-door**-1, **open-right-door**-1, the single observation chance variable hear-tiger-behind-left-door-1, and the three choice variables encoding the action at time step 2—**listen-for-tiger**-2, **open-left-door**-2, **open-right-door**-2. The root node of the tree contains the variable **listen-for-tiger**-1, the two nodes on the next level of the tree contain the variable **open-left-door**-1, and so forth (triangles indicate subtrees for which details are not shown). The observation variable hear-tiger-behind-left-door-1 is a branch point; the optimal assignment to the remaining choice variables (**listen-for-tiger**-2, **open-left-door**-2, **open-right-door**-2) will be different for different values of this variable.

This representation of the planning problem is similar to *AND/OR trees* and *MINIMAX trees* [85]. Choice variable nodes are analogous to OR, or MAX, nodes, and chance variable nodes are analogous to AND, or MIN, nodes. However, the probabilities associated with chance variables (our opponent is nature) make the analogy somewhat inexact. Our trees are more similar to *MINIMAX trees with chance nodes* [5] but without the MIN nodes—instead of a sequence of alternating moves by opposing players mediated by random events, our trees represent a sequence of moves by a single player mediated by the randomness in the planning domain. Note that the SSAT framework can accommodate a scenario in which an agent faces an opposing player; the opposing player's actions are modeled by universally quantified variables [71].

The solver essentially implements the DPLL-based algorithm described in

Section 5.2.1. It does a depth-first search of the tree, constructing a solution subtree by calculating, for each node, the probability of a satisfying assignment given the partial assignment so far. For a choice variable, this is a maximum probability and produces no branch in the solution subtree; the solver notes which value of the variable yields this maximum. For a chance variable, the probability will be the probability weighted average of the success probabilities for that node's subtrees and will produce a branch point in the solution subtree. The solver finds the optimal plan by determining the subtree with the highest probability of success. In Figure 7.4, the plan portion of this subtree appears in bold, with action choices (action variables set to `True`) in extra bold. The optimal plan is: **listen-for-tiger**; if hear-tiger-behind-left-door is `True`, **open-right-door**; if `False`, **open-left-door**.

Like the solutions found by ZANDER, the solution of an AND/OR tree is a subtree satisfying certain conditions. Algorithms for solving these trees, such as AO* [85, 79, 74] and LAO* [46, 47] try to combine the advantages of dynamic programming (reuse of subproblems) with advantages of forward search (use of heuristic estimates to avoid evaluating the entire state space and thereby speed up the search process). These algorithms operate by repeating a two-phase operation: use heuristic estimates to identify the next node to expand, then use dynamic programming to re-evaluate all nodes in the current subgraph. LAO* is of particular interest in that it extends this approach to find solutions with loops (something ZANDER is currently unable to do) and, as a result, can solve infinite-horizon MDPs (and, thus, problems in which the optimal plan requires that an action be repeated an indefinite number of times).

In contrast to heuristic search approaches, which must follow a prescribed variable ordering, ZANDER can consider variables out of the quantifier ordering

specified in the SSAT problem when this allows it to prune subtrees (as in irrelevant variable elimination, unit propagation, and pure variable elimination). The main novelty of my approach, in fact, lies in my use of the stochastic satisfiability formulation of the problem, which allows ZANDER to use satisfiability heuristics, such as unit propagation and pure variable elimination, to prune subtrees. It is possible that the algorithm could use heuristic search to solve the trees generated by our planning problems. A worthwhile area of research would be to compare the performance of these two approaches and attempt to develop techniques that combine the advantages of both.

7.3.3 Results

I tested three variants of ZANDER on problems drawn from the planning literature (see Figure 7.1). All tests were done on a 300 MHz UltraSparc II with 384 Mbytes of RAM, running SunOS-5.7. ZANDER:SPLIT, the basic solver, uses variable splitting with no heuristics. It checks partial assignments for satisfiability or unsatisfiability, but, otherwise, checks every possible assignment. ZANDER:HEUR is the basic solver augmented with irrelevant variable elimination, unit propagation, and pure variable elimination (see Section 6.4). ZANDER:THRESH is the basic solver augmented with these heuristics and thresholding (see Section 5.2.1). Note that the threshold probability provided to ZANDER:THRESH was the probability of success of the optimal plan, thus ensuring that the problems were not made artificially easy by requiring plans with an unreasonably low probability of success.

The problems selected cover a range of different conditions. The TIGER problems [54] (with horizon increasing from one to four) contain uncertain initial conditions and a noisy observation action. In this domain, the agent is faced

with two doors, one concealing a hungry tiger, the other concealing a treasure. Before opening one of the doors, the agent can listen for the tiger, but this observation is accurate only 85% of the time. A unique feature of this problem is that, in some cases (*e.g.* the 4-step TIGER problem), the agent needs to condition its actions on the entire observation history in order to act correctly.

The SHIP-REJECT problem [32] has the same characteristics as the TIGER problem (uncertain initial conditions and a noisy observation action), along with a causal action (**paint**) that succeeds only part of the time. In this domain, a part is initially flawed (not visible) and blemished (visible) with probability 0.30. The objective is to paint and process the part, where processing consists of deciding whether to ship the part (if it is not flawed) or reject the part (if it is flawed). While painting the part erases the blemish, it does not correct the internal flaw, so the agent must observe whether the part is blemished, paint the part, and then condition the ship/reject decision on it earlier observation.

In the MEDICAL-4ILL problem [113], there are uncertain initial conditions, multiple perfect observations, and causal actions with no uncertainty. In this domain, a patient is either healthy or has one of four illnesses (with equal probability). Fortunately, there is a medication for each illness that will cure the patient with certainty. The patient, however, will die if she receives any medication for which she does not have the corresponding illness. Thus, it is critical to disambiguate the initial conditions. There is a stain test that allows the agent to determine which of the following three categories the patient's illness falls into: 1) no illness, 2) illness 1 or 2, or 3) illness 3 or 4. There is a white cell count test that allows the agent to distinguish between illnesses 1 and 2 and illnesses 3 and 4. Together, these tests allow the agent to determine the patient's illness with certainty and administer the correct medication.

The EXTENDED-PAINT problems [86] have no uncertainty in the initial conditions, but require that probabilistic actions be interleaved with perfect observations. A part must be painted, cleaned, and polished. These actions succeed only half the time, so it is likely that they will need to be repeated. It is, however, an error to paint, clean, or polish a part that is already painted cleaned, or polished, respectively, so, beyond a certain point, the agent must condition its actions on observations of the part.

Finally, the COFFEE-ROBOT problem, similar to a problem from the POMDP literature [16], is a larger domain (7 actions, 2 observation variables, and 8 state propositions in each of 6 time steps) with uncertain initial conditions, but perfect causal actions and observations. In this domain, there is a robot that is capable of going back and forth between a user's office and the cafeteria. It can purchase coffee at the cafeteria and deliver it to the user in her office. If the user does not want coffee, the robot does not need to do anything. If the user does want coffee, the robot must get and deliver the coffee. It may be raining and, since the robot should not get wet, it must take an umbrella if it is raining. Initially, it is raining with probability 0.50 and the user wants coffee with probability 0.50. The robot can ask the user if she wants coffee and can look out the window to see if it is raining.

As expected, the performance of ZANDER:SPLIT is poor except on the simplest problems (Table 7.1). But, the results for ZANDER:HEUR and ZANDER:THRESH are very encouraging; the techniques used in these variants are able to reduce solution times by as much as five orders of magnitude or more (the reduction from ZANDER:SPLIT to ZANDER:HEUR in the 7-step EXTENDED-PAINT problem is unclear since ZANDER:SPLIT did not run to completion). I compared the performance of these three variants of ZANDER to that of:

Problem	Threshold Probability of Success	Solution Time (CPU seconds, average of 5 runs)					
		ZANDER: SPLIT	ZANDER: HEUR	ZANDER: THRESH	POMDP: LARK	MAHINUR	SGP
TIGER-1	0.5	0.02	0.01	0.02	0.03	0.02	X
TIGER-2	0.85	0.13	0.03	0.02	0.04	0.10	X
TIGER-3	0.85	2.87	0.04	0.02	0.07	X	X
TIGER-4	0.93925	67.38	0.17	0.07	0.20	X	X
SHIP-REJECT	0.9215	23.91	0.06	0.05	0.18	0.16	X
MEDICAL-4ILL	1.0	157.33	1.42	0.20	3.04	X	44.30
EXTPAINT-4	0.3125	12,652.75	0.43	0.12	0.84	0.60	X
EXTPAINT-7	0.773437	24+ hrs	143.27	31.47	34.64	23.11	X
COFFEE-ROBOT	1.0	35,354.72	1,424.51	594.72	33.80	T	M

X = Planner not applicable to problems of this type

M = Insufficient memory

T = Not available due to technical difficulties

Table 7.1: ZANDER with heuristics (ZANDER:HEUR) and ZANDER with heuristics and thresholding (ZANDER:THRESH) outperform POMDP:LARK, MAHINUR, and SGP on many problems.

- the "Lark" pruning POMDP algorithm [20, 115] on the corresponding finite-horizon POMDP formulations of these problems,
- MAHINUR [88], and
- SGP (SENSORY GRAPHPLAN) [113].

The results of these comparisons are reported in Table 7.1.

The performance of ZANDER:HEUR equals or betters that of POMDP:LARK on every problem except the 7-step EXTENDED-PAINT problem and the COFFEE-ROBOT problem. Subsequent experiments indicate that augmenting the SSAT encoding of the COFFEE-ROBOT problem in two simple ways can improve ZANDER's performance tremendously:

1. Enforce the implicit preconditions of the buy-coffee and deliver-coffee actions. For the former, the robot must be at the cafeteria; for the latter, the robot must have the coffee and must be at the user's office.
2. Forbid any actions in the final time step that do not directly support a goal condition (change-location, buy-coffee, get-umbrella, look-out-window, and ask-user-if-wants-coffee).

These two changes reduce the solution time of ZANDER:HEUR from 1425 CPU seconds to 98 CPU seconds (compared to 34 CPU seconds for POMDP:LARK). The performance of ZANDER:THRESH equals or betters that of POMDP:LARK on every problem except the COFFEE-ROBOT problem. And, the augmented encoding described above reduces the solution time of ZANDER:THRESH from 595 CPU seconds to 23 CPU seconds (compared to 34 CPU seconds for POMDP:LARK).

MAHINUR [88] makes some limiting assumptions (see Section 2.2.1), but these assumptions are not violated in any of my test problems, and, in principle,

MAHINUR is applicable to all the problems in this test suite. In its current form, however, MAHINUR is not capable of handling certain situations, and so is not applicable to as broad a range of problems as POMDP:LARK. Although MAHINUR provides a framework to reason about the relationship between observation actions (either the same observation action repeated or a sequence of different observation actions), this capability has not been implemented yet [87]. This means that MAHINUR cannot currently solve some planning problems in the TIGER domain (repeated observations), or any problems in the MEDICAL-4ILL domain (a sequence of different observations).

Comparisons between ZANDER and MAHINUR are further complicated by the fact that the two planners take different approaches to determining when to stop in the search for a plan. ZANDER either searches until it finds the optimal plan (ZANDER:SPLIT and ZANDER:HEUR) or until it finds a plan whose probability of success meets or exceeds a specified threshold (ZANDER:THRESH). MAHINUR, as it is currently implemented, constructs a skeletal plan and then attempts to improve the plan a specified number of times by extending it to cover the most critical contingency unaddressed by the current plan. I compared the two planners for a given problem instance by increasing MAHINUR's improvement iteration limit until the probability of success of the plan produced equaled or exceeded that of the plan produced by ZANDER on the same instance.

The performance of ZANDER:HEUR and ZANDER:THRESH equals or betters that of MAHINUR on every problem that MAHINUR solved successfully except the 7-step EXTENDED-PAINT problem. And, in this problem, by enforcing implicit action preconditions, the solution times of ZANDER:HEUR and ZANDER:THRESH can be reduced to 67 CPU seconds and 15 CPU seconds respectively. Thus, with this improvement in the encoding of the problem, ZANDER:THRESH is able to find

the optimal plan faster than MAHINUR (23 CPU seconds).

MAHINUR's performance in the TIGER and COFFEE-ROBOT domains requires some explanation. In the TIGER domain, because MAHINUR cannot reason about repeated observations, it can only find plans with a horizon of one or two (no observations or one observation, respectively). On these problems, ZANDER equals or exceeds MAHINUR's performance; in the 2-step case, the solution times of the ZANDER:HEUR and ZANDER:THRESH are an order of magnitude faster than that of MAHINUR. For a plan horizon T greater than two, the optimal plan is to listen for $T-1$ steps, then open-left or open-right as dictated by a majority vote of the observations (with ties broken randomly). As the horizon increases, more listen actions are possible, and the probability of success increases. ZANDER is able to find such plans, while MAHINUR cannot.

In principle, MAHINUR should be able to handle the COFFEE-ROBOT problem. Indeed, MAHINUR was able to find the optimal plan in a simplified version of this problem, in which all uncertainty is removed (it is known initially that the user wants coffee and that it is not raining). But, a run in which the initial conditions were changed to specify that it is raining (with certainty) generated an error condition. Because I have been unable to determine the source of this error, a comparison on this problem is unavailable at this time.

Since the current version of SGP [113] is unable to handle probabilistic actions or noisy observations, comparisons were restricted to two problems: MEDICAL-4ILL and COFFEE-ROBOT. The solution times of ZANDER:HEUR and ZANDER:THRESH on the MEDICAL-4ILL problem are one and two orders of magnitude faster, respectively, than that of SGP. A comparison on the COFFEE-ROBOT problem was impossible because SGP ran out of memory while attempting to solve the problem.

7.4 Summary and Discussion

In this chapter, I described ZANDER, a planner that extends the probabilistic-planning-as-stochastic-satisfiability paradigm to support contingent planning in partially observable stochastic domains. ZANDER encodes a contingent planning problem as a more general SSAT problem. This allows ZANDER to interleave choice variables encoding actions with chance variables encoding observations. Since, in an SSAT problem, the value of a choice variable can be made contingent on the value of the chance variables that precede it in the quantifier ordering, this allows ZANDER to produce contingent plans. Interestingly, although plan encodings in ZANDER allow a more complex quantifier ordering than those in MAXPLAN, the actual variables and clauses in a ZANDER plan encoding are remarkably similar to those in a MAXPLAN encoding.

Because ZANDER can encode any degree of observability (both in terms of which state propositions can be observed, and how accurately they can be observed) and because ZANDER does not limit the size (only the horizon) of the resulting plan, ZANDER can solve arbitrary finite-horizon factored POMDPs. This is in sharp contrast to both MAHINUR and SGP, both of which are limited in the types of problems they can handle. As I pointed out in Chapter 2, MAHINUR assumes a type of subgoal decomposability that limits its applicability. In addition, MAHINUR cannot currently handle multiple observations. SGP cannot currently handle probabilistic actions or noisy observations. Thus, both MAHINUR and SGP are applicable only to a subset of partially observable planning problems. ZANDER represents significant progress toward the goal of exploiting the state information available in a factored POMDP in order to efficiently solve planning problems in stochastic, partially observable domains.

Chapter 8

Open Problems and Future Work

The results I have obtained with ZANDER are especially encouraging, given that;

- ZANDER can solve a more general class of problems than SGP and MAHINUR, and
- there are a number of improvements to ZANDER that have shown promise for scaling up to larger problems.

I think it is unlikely, however, that these improvements will be sufficient to allow ZANDER to handle some of the larger, real-world problems I would like to be able to attack. In order to scale up to these problems, it will probably be necessary to develop approximate SSAT solution techniques. In this chapter, I will discuss possible improvements to and extensions of ZANDER and approximation techniques for SSAT problems. I will discuss the idea of constructing planners that operate less independently (leading to the notion of human-computer collaborative planning). Finally, I will discuss the potential of this paradigm for providing a unifying framework for planning and scheduling under uncertainty.

8.1 Improvements to ZANDER

There are a number of improvements to ZANDER that have strong potential to improve performance significantly. Given ZANDER's two-phase approach, these improvement naturally fall into two categories:

- improvements in the SSAT encoding of planning problems, and

- improvements in the algorithm for solving the SSAT encodings.

In the following sections, I will describe each of these improvements and discuss initial efforts to implement them in ZANDER.

8.1.1 Improved SSAT Encodings

Compiling Away State Variables

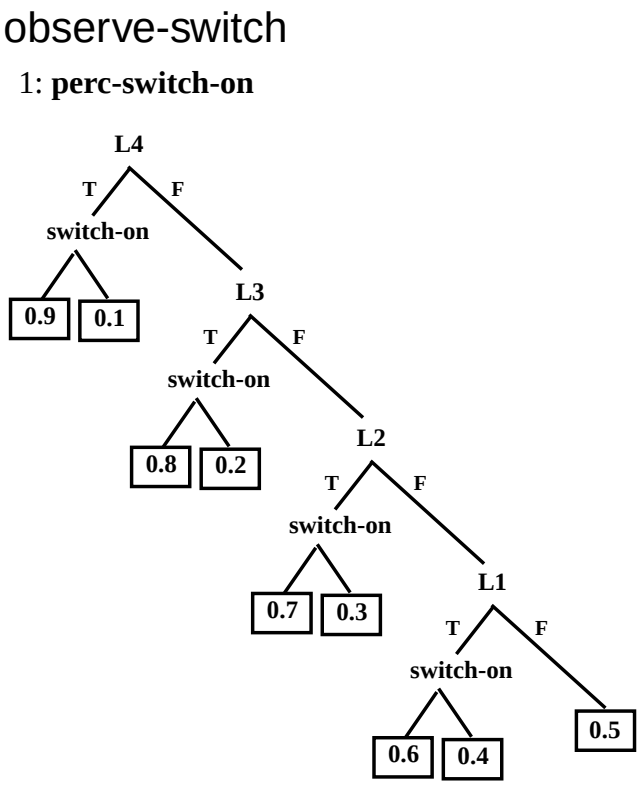
MAXPLAN's efficiency could possibly be improved by using a more efficient CNF encoding of the planning problem. One can use resolution to eliminate any subset of variables [56], but this usually leads to an exponential blowup in the number of clauses in the encoding. For GRAPHPLAN-based encodings, however, eliminating the propositional variables that describe the state of the environment leads to an increase that is polynomial in the number of these propositions [56]. Although I have not conducted extensive tests, my SSAT solver seems to be more sensitive to the number of variables than to the number of clauses. It is possible that the efficiency of the solver could be improved as the result of identifying a group of variables whose elimination would entail only a polynomial increase in the number of clauses.

More Efficient Versions of Current Type of Encoding

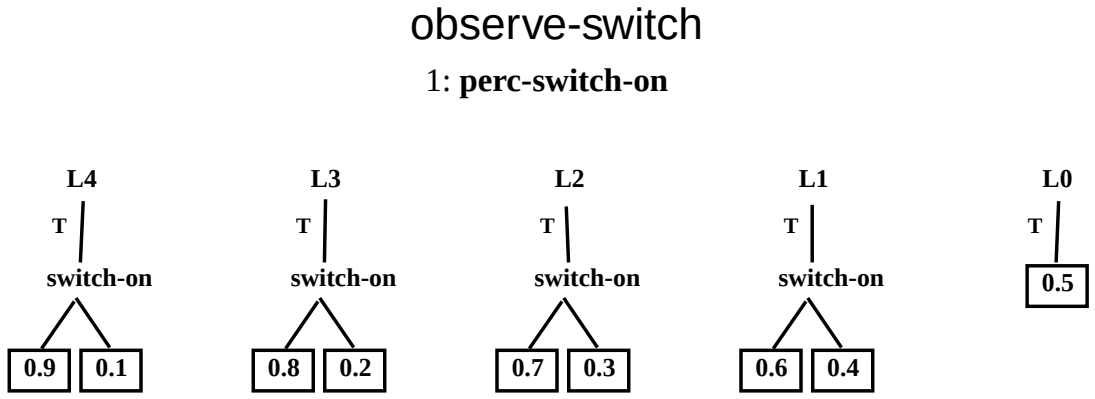
There are some encoding efficiencies to be realized even within the current type of SSAT encoding used by ZANDER. For example, the current encoding enforces mutual exclusivity of actions with $\binom{A}{2}$ binary clauses, where A is the number of actions. These clauses, quadratic in the number of actions, merely specify that, for each possible pair of actions, the negation of one of them has to be `True`. By extending the SSAT formalism to include a type of clause that specifies that

exactly one of the literals in that clause must be `True`, I could produce more compact encodings *and* improve the performance of the solver. The increased compactness of the encoding, of course, comes from replacing a quadratic number of clauses with a single clause. The increased efficiency in the solver is due to the fact that, currently, choosing an action a at a particular time step results in $A-1$ unit clauses; the mutual exclusivity clauses specify that since a is `True`, all other actions at that time step must be `False`. The algorithm then goes through a series of $A-1$ recursive calls to process these $A-1$ unit clauses. A properly implemented "exactly-one-of" clause would cause the solver to make the $A-1$ forced assignments as soon as one of the literals in the clause was assigned the value `True`. The use of such a clause is not limited to mutual exclusivity of actions. Plan encodings frequently contain groups of variables that represent mutually exclusive states of some entity in the domain (*e.g.* discrete settings on some control or gauge). Exactly-one-of clauses could be used to encode the mutual exclusivity constraints on these variables as well.

Another possible efficiency improvement stems from the existence of action decision trees that are essentially encoding a cascade of conditions. For example, suppose there is a switch in the domain that can be ON or OFF, and that there is an observe-switch action. Suppose further that there are five discrete levels of lighting—L0, L1, L2, L3, and L4—and that the probability of correctly observing the dial increases with the lighting level. At level L0 there is no light and the agent is guessing, so the probability of **perc-switch-on** being set correctly when the switch is on is only 0.5. For each increased level of lighting, the probability increases by 0.1 so that at level L4, the probability is 0.9. One possible decision tree for proposition **perc-switch-on** is shown in Figure 8.1(a). This decision tree would generate 18 clauses with an average of 6.6 literals per clause.



(a) Decision trees are sometimes an inefficient cascade of conditions.



(b) A more efficient encoding can be thought of as being produced by shorter decision trees and an exactly-one-of clause.

Figure 8.1: Cascaded decision trees can ne replaced by more efficient encodings.

Since L0, L1, L2, L3, and L4 are mutually exclusive, the same information can be encoded by the "decision trees" shown in Figure 8.1(b) and one exactly-one-of clause for L0, L1, L2, L3, and L4. Note that these are not real decision trees; they are provided only for illustrative purposes. What they indicate is that the clauses could separately model what happens if each of L0, L1, L2, L3, and L4 is `True`. This produces 19 clauses, but now each clause has an average of 4.6 literals (compared to 6.6 literals per clause for the previous encoding). This difference of 2.0 literals per clause may not seem significant, but 1) as the number of lighting levels increases, so does this difference, and 2) since many splitting heuristics operate by trying to produce shorter clauses (unit clauses) it seems likely that a reduction in the clause size of the original encoding (with virtually no increase in the number of clauses) would have a beneficial impact on performance.

Encoding Domain-Specific Knowledge

Domain-specific knowledge could be exploited in either the construction of the SSAT formula or its subsequent solution. The first approach has been explored by Kautz and Selman in the context of SATPLAN [61]. In their work, four types of clauses that can be added to a SAT encoding of a planning problem were described:

- *Conflict* clauses and *derived effect* clauses implied by the domain's action descriptions.
- *State invariant* clauses implied by the domain's actions and initial conditions.
- *Optimality condition* clauses implied by the actions, initial conditions, and plan length.

- *Simplifying assumption* clauses.

The first three types of clauses make knowledge that was previously implicit in the problem domain explicit and are analogous to providing lemmas to a theorem prover. The fourth type of clause is not implicit in the domain and, in fact, may prevent some solutions from being found [61]. Adding such clauses to the SAT encoding can accelerate the solution process enormously, particularly for systematic satisfiability testers, reducing the solution time on some problems from in excess of 48 hours to a few seconds [61].

Another way of incorporating domain-specific knowledge is to use such knowledge to guide the SSAT solution process. For example, we might be able to use optimality criteria or means-ends analysis to efficiently identify high probability plans or prune low probability plans.

Alternate SSAT Encodings

Alternate SSAT encodings are of particular interest. Our current encodings are analogous to the state-based encodings with classical frame axioms described by Kautz, McAllester, and Selman [56]. Two other possibilities are state-based encodings with explanatory frame axioms, and GRAPHPLAN-style encodings [56]. A third alternative encodes a causal link representation of the planning problem [78].

I have begun preliminary explorations into the use of other types of encodings. For example, in my current encodings, if all but one of A actions leaves a fluent unchanged, each of the $A-1$ actions that leave the fluent unchanged generates two clauses at each time step modeling that fact. This is essentially using classical frame axioms and is redundant, given that all that really needs to be encoded is

the fact that the fluent will remain unchanged *unless the one action that changes it is executed.* By using explanatory frame axioms that encode the latter notion, the encodings could be made more compact.

GRAPHPLAN-style encodings are particularly attractive in that they allow for the possibility of executing actions in parallel. Parallel action execution can potentially reduce the length of the plan that needs to found and, thus, the solution time. ZANDER should be able to accommodate parallel actions with minimal difficulty.

A further interesting possibility is to construct hybrid encodings. Although it is not always true that more constraints (clauses) is better, the more nonredundant constraints encoded, the more efficiently the solver will be guided to a solution. For example, in a more difficult version of the coffee-robot problem, ZANDER needed about 9300 cpu seconds to solve the state-based (with classical frame axioms) encoding of the problem. Using a state-based encoding with explanatory frame axioms *augmented* with GRAPHPLAN-style axioms (actions imply their preconditions, and facts at time t imply the disjunction of all actions at time $t-1$ that could produce that fact), reduced the solution time from approximately 9300 CPU seconds to less than a second.

There are two other possibilities for alternate SSAT encodings that are more speculative. In Section 2.2.3, I discussed the power of using a value function—a mapping from states to values that measures how "good" it is for an agent to be in each possible state—to solve MDPs and POMDPs. POMDP:LARK, the most successful of the three probabilistic planning techniques I compared ZANDER to (Section 7.3.3), derives its excellent performance in part from this value-based approach. Perhaps it would be possible to develop a value-based encoding for ZANDER that would benefit from the power of this approach. If such an encoding

could be used to perform value approximation, it would be particularly useful in the effort to scale up to much larger domains.

The second possibility borrows a concept from belief networks to address the difficulty faced by an agent who must decide which of a battery of possible observations is actually relevant to the current situation. *D-separation* [23] is a graph-theoretic criterion for reading independence statements from a belief net. Perhaps there is some way to encode the notion of d-separation in an SSAT plan encoding in order to allow the planner to determine which observations are relevant under what circumstances.

8.1.2 Improved SSAT Solution Techniques

More Efficient Data Structures

More sophisticated data structures in which to store the CNF encoding would almost certainly improve the efficiency of the solver. For example, the *trie* data structure has been used to represent SAT problems [116]. Briefly, variables are given unique integer indices (the index of a negated variable is the negative index of the positive variable), and each clause is arranged so that the indices of its literals are in ascending order by absolute value. A clause is represented as a path through a trinary tree (a trie) in the following manner. Each node v implicitly represents a clause prefix C_P (described by the path from the root to that node), contains a positive variable index i that extends that prefix, and has three children:

1. a *positive child* containing the set of all clauses having the prefix $C_P \vee i$,
2. a *negative child* containing the set of all clauses having the prefix $C_P \vee -i$, and

3. a *remaining child* containing the set of all remaining clauses having the prefix C_P.

For example, the set of clauses $\{(1 \vee 2), (-1 \vee -3), (-1 \vee 4), (-2 \vee 3)\}$, where the integers are variable indices, would be represented by the trie:

$$\langle 1, \langle 2, \Box, \emptyset, \emptyset \rangle, \langle 3, \emptyset, \Box, \langle 4, \Box, \emptyset, \emptyset \rangle \rangle, \langle 2, \emptyset, \langle 3, \Box, \emptyset, \emptyset \rangle, \emptyset \rangle \rangle$$

where $\langle i, P, N, R \rangle$ denotes the node containing variable index i, with positive child P, negative child N, and remaining child R, and $\Box$ is the end-of-clause symbol. In this manner, clauses are stored such that two clauses sharing a prefix of n literals will share a path of length n in the trie. Several advantages have been claimed for this use of tries [116]:

- duplicate clauses are automatically eliminated when the trie is constructed,
- memory requirements are reduced due to shared clause prefixes,
- unit clauses can be found quickly, and
- the unit propagation operation can be computed efficiently; multiple unit propagation operations can be done in a single trie traversal.

It seems likely that better data structures, such as tries, would improve the efficiency of the SSAT solver by reducing the overhead associated with the operation of the solver.

More Sophisticated Splitting Heuristics

The current splitting heuristic orders groups of candidate variables according to the order of their appearance in the quantifier ordering. In the plan-execution

history segment (variables encoding actions and observations), this coincides with the ordering that would be imposed by time-ordered splitting. The chance variables in the domain uncertainty segment and the choice variables in the segment that encodes the result of the plan-execution history given the domain uncertainty, are time-ordered.

The current heuristic, however, does not specify an ordering for variables within the blocks of similarly quantified variables that have the same time index. This may be insignificant in small problems, but in real-world problems with a large number of variables at each time step, a splitting heuristic that addresses this subordering issue could provide a significant performance gain. The experiments reported in Section 5.2.2 provide evidence that performance could be improved by using more sophisticated variable ordering heuristics within such groups.

It might also be possible to develop a dynamic measure of the "criticality" of variables to make better choices when selecting the next variable to split on. This notion is an extension of unit propagation, which gives priority to a variable whose value is forced. The idea here would be to develop a measure of how "forced" the value of each non-unit-clause variable is, and to give splitting priority to those variables with a high degree of "forcedness." If this could be used to locate the high-probability assignments quickly, it would be of tremendous use in approximation algorithms. Initially, this index could be based on structural considerations such as how many clauses a variable appears in, how small these clauses are, and the ratio of positive to negative instances of the variable. Learning could possibly be used to refine this index (see the following section on Using Learning to Improve Performance).

Memoization of Intermediate Results

With regard to memoization, ZANDER separately explores and saves two plan execution histories that diverge and remerge, constructing a plan tree when a directed acyclic graph would be more efficient. ZANDER should be able to memoize subplan results so that when it encounters previously solved subproblems, it can merge the current plan execution history with the old history. Memoization boosted MAXPLAN's performance tremendously and it is likely that it would have a similar beneficial effect on ZANDER's performance.

Improved Caching Techniques

My initial experiments with caching yielded promising results, but there are at least three areas where further work needs to be done. First, although smart LRU caching yielded significant performance improvements (up to a 37% decrease in CPU seconds compared to straight LRU caching), it is not practical unless I can determine the optimal difficulty range for subformulas to be cached without extensive testing. Second, the algorithm can, in principle, do better than smart-caching. In one test, performance deteriorated significantly when the cache size was set below 5000, yet only 1786 distinct subformulas were reused in the solution process, implying that a more sophisticated cache replacement policy could yield additional performance gains. Such a policy could be based on many attributes of the cached subformulas, including difficulty, last use, and frequency of use.

Using CSP Techniques to Accelerate Performance

ZANDER could probably be improved by adapting other techniques that have been developed for constraint satisfaction problems (CSPs). In CSP terms, ZANDER

uses backtrack search with forward checking and a variable ordering heuristic that gives priority to unit-domained and pure variables. I would like to explore the possibility of incorporating CSP look-back techniques, such as backjumping and learning (deriving no-goods) [7].

In backjumping, when the solver reaches a deadend, rather than backtracking to the assignment immediately previous to the deadend (regardless of whether that assignment is the one that caused the problem), the solver back*jumps* to the most recent assignment that directly contributes to the current deadend. This helps prevents the solver from engaging in needless backtracking. For example, suppose I am going to the store to buy some groceries. Upon leaving the house, I notice that my children have left some of their toys on the driveway, so I clear all the toys from the driveway. I open the garage door, and then discover I have forgotten the car keys. Simple backtracking would force me to close the garage door, and put the toys back on the driveway before I returned to the house to get the keys. Backjumping would allow me to jump over these most recent activities without undoing them and return to the house for the keys. Although backjumping is a very attractive idea, it is not immediately clear how this could be applied to solving SSAT problems, where the solver needs to find *all* the satisfying assignments.

Deriving no-goods is a process of dynamically augmenting the encoding to record the reasons for deadends encountered in the solution process. In other words, when the solver reaches a deadend, it analyzes the reason for that deadend (a "reason" being a partial assignment) and adds a clause encoding this "no-good" to the formula to prevent that situation from arising again. Since unbounded learning (saving all no-goods throughout the solution process) is, in general, not feasible, a solver that uses this approach must deal with the same

issue that arises when using memoization: If I can't save everything, what *should* I save? Bayardo and Schrag [7] suggest two solutions: *size-bounded* learning, which retains indefinitely only those no-goods whose size is less than a specified maximum, and *relevance-bounded* learning, which retains any no-good that contains no more than a specified maximum number of variables whose assignments have changed since the no-good was derived.

Using Learning to Improve Performance

In the previous section, I discussed the possibility of adapting a CSP learning technique (learning no-goods [7]) for use in ZANDER. Learning could also potentially improve the planner in two other areas: splitting heuristics and cache replacement policies. Learning might be able to uncover an optimal subordering of variables within the groups of variables specified by the quantifier ordering, or even an entirely different heuristic that improves performance. Learning might also be a suitable technique for finding a better cache replacement policy. There are some obvious attributes of the cached subformulas on which to base a replacement policy, but it is not clear how these attributes should be weighted for optimal performance. Learning provides a technique for optimizing these weights automatically.

Learning could also be used to learn useful relationships among the variables in the SSAT formula. In binary clauses, for example, the relationship is clear. Assigning a value to one variable that is contrary to the sign of that variable as it appears in the binary clause, forces the other variable's assignment. During the course of searching for satisfying assignments, the algorithm may be able to learn other, less direct, relationships between variables, or among a group of variables. Perhaps a group of variables has a single collective assignment that

is overwhelmingly likely, so the algorithm does not need to consider any other assignments for this group of variables in its search for high probability satisfying assignments.

Another recent and highly successful use of learning that is closely related to my work is STAGE, a learning approach that automatically improves search performance on optimization problems [17]. This approach, which learns good starting points for a specified local search technique, has been used to solve some extremely difficult satisfiability problems, and could possibly be adapted for use in an SSAT solver.

One important issue that would need to be addressed in order to use learning successfully is the availability of a sufficiently long training period to ensure adequate learning. With the exception of very large planning problems, such a training period will not, in general, be attainable in the course of solving a single problem. Thus, the problem becomes one of finding a set of "typical" problems to train the system on, the results of which will transfer to the solution of problems the system has not seen yet. Boyan and Moore report positive results with respect to the transfer of learning between problem instances [17].

8.2 Extensions of ZANDER

The improvements discussed in the sections above focus on accelerating ZANDER's performance. This section discusses two extensions to ZANDER that would significantly broaden its scope, both in terms of the type of planning problems it is able to handle and in terms of its plan-evaluation criteria.

8.2.1 Ability to Produce More Complex Plans

MAXPLAN produces totally ordered plans (Figure 6.2(a)); ZANDER produces acyclic, contingent plans (Figure 6.2(b)). This is a significant improvement in the potential usefulness of this paradigm for real-world planning, but it is not hard to think of planning domains in which the only realistic plan is a looping plan (Figure 6.2(c)), in which an action—or sequence of actions—is repeated an indefinite number of times until some effect is achieved. I would like to extend ZANDER to be able to produce looping plans. The problem of finding such plans is still in PSPACE [70], so it is possible that ZANDER could be extended to find such plans.

One possibility is suggested by an approach taken by C-MAXPLAN, a contingent planning extension of MAXPLAN briefly described at the beginning of Chapter 7. In one version of C-MAXPLAN, instead of searching for the optimal contingent plan of a given length, the algorithm searches for an optimal small *policy* to be applied for a given number of steps. Perhaps the SSAT encodings of ZANDER could be modified to generate policy-like solutions as well. Such solutions would allow ZANDER to specify plans in which an action is to be repeated as many times as is necessary, up to the step limit specified. If no successful policy could be found for a given step limit, because a particular action could not be repeated often enough, iteratively increasing the step limit would eventually lead to a successful combination of policy and step limit.

8.2.2 Incorporating Decision-Theoretic Criteria

Many AI planners evaluate plans according to a single, rather simplistic, criterion: does the plan reach a goal state? Planning in the real world often requires more

subtle evaluations. Partial goal fulfillment is sometimes better than complete failure, but sometimes not. The existence of multiple independent goals brings up issues of tradeoffs among these goals: if I can't accomplish everything, what it the best subset of goals to accomplish. Decision-theoretic planning, which incorporates utility-based preferences, allows one to answer these difficult questions in a principled manner. Currently, ZANDER incorporates utility-based preferences in a very limited way, preferring plans with a higher probability of success. Many types of utility can be encoded in success probabilities, but the resulting encodings are prohibitively large. I would like to extend ZANDER so that it can evaluate plans using a broader conception of *utility* than probability of success alone. For example, ZANDER sometimes returns an unnecessarily large plan; a modest initial goal would be to give ZANDER the ability to discriminate between plans with equal probability of success using length as a criterion.

8.3 Approximation Techniques for Solving SSAT Problems

Although improvements to the current planner may allow ZANDER to scale up to problems of moderate complexity, they are unlikely to be sufficient to achieve my ultimate goal of planning efficiently in large, real-world domains. I think it is likely that I will need to develop an approximation technique for solving SSAT problems to scale up to problems of this size. Optimality is sacrificed for "anytime" planning and performance bounds, and although this may not improve worst-case complexity, it is likely to help for typical problems.

8.3.1 Removing Some of the Uncertainty

Another possibility is to convert the probabilistic planning problem into a deterministic planning problem by rounding each decision-tree leaf probability to 0.0 or 1.0, solving the resulting deterministic planning problem relatively efficiently and then gradually reintroducing uncertainty to improve the quality of the solution. It is not clear, however, how to reintroduce the uncertainty without sacrificing the efficiency gained by removing it.

8.3.2 Using Stochastic Local Search

ZANDER systematically searches for satisfying assignments by setting the truth value of each variable in turn and considering the remaining subformula. This is significantly different from the WALKSAT approach in SATPLAN, which begins with a complete truth assignment and adjusts it through stochastic local search to achieve a satisfying assignment. In the same way that stochastic local search can solve much larger SAT problems than systematic search (in general), it is possible that adapting stochastic local search to the solution of SSAT problems would provide significant performance gains. The fact that an SSAT solver needs to systematically evaluate all possible assignments to solve the SSAT problem exactly, argues for a systematic approach. There are, however, a number of ways that stochastic local search could be incorporated into an SSAT solver.

One obvious possibility for adapting stochastic local search for use in an SSAT solver is to use stochastic local search repeatedly to find many satisfying assignments. Each time a satisfying assignment is found, the assignment would be added to a list of "forbidden" assignments (or possibly encoded as a "no-good") to prevent the local search process from rediscovering satisfying assignments al-

ready found, and a new local search would be started from a random assignment (discarding those that duplicate forbidden assignments). The probability of each satisfying assignment could easily be calculated, and the probabilities of the satisfying assignments found could be used to estimate the optimal plan and its probability of success (in the same way that the true optimal plan and its probability of success would be calculated given *all* possible satisfying assignments).

Combining Systematic Search and Stochastic Local Search

The approach just described suggests a further possibility: combine systematic search and stochastic local search by starting multiple local searches from points in the space of assignments systematically chosen to cover the entire space. This idea is similar to an underwater search in which multiple divers are sent down at (roughly) evenly spaced points, and each diver randomly searches an area around his entry point. A possible refinement of this idea is to use the results of the stochastic local searches conducted so far to help choose the next point from which to begin a new local search for additional satisfying assignments. This is similar to the STAGE approach [17] mentioned in Section 8.1.2: Using Learning to Improve Performance.

Using Stochastic Local Search in Plan Space

One possible approximation technique would be to use stochastic local search in plan space in conjunction with simulation as an approximate plan-evaluation technique. In this approach, I would start with a random setting of all the variables (which implies a particular plan) and then, in the remaining available planning time, adjust this plan through stochastic local search to improve its probability of success. The probability of a given plan's success would be esti-

mated using random sampling (see Section 5.2.3).

The most promising approach, however, is to use stochastic local search in a reduced plan space, *i.e.* adapt the approximation technique described in Section 5.2.3 to planning problems. Recall that this approach uses random sampling to select a subset of possible randomized variable instantiations (thus limiting the size of the contingent plans considered) and stochastic local search to find the best size-bounded plan. This approach has the potential to quickly generate a suboptimal plan and then, in the remaining available planning time, adjust this plan to improve its probability of success. This approach appears to be the most promising possibility, and I will describe it further.

Once a probabilistic planning problem has been translated into an SSAT instance, it would seem to be straightforward to apply the stochastic sampling algorithm to the SSAT problem to find an approximation of the optimal contingent plan and an approximation of its probability of success. The situation is complicated, however, by the fact that randomized variables are used to describe observations. This means that a random sample of the randomized variables describes an observation sequence as well as an instantiation of the uncertainty in the domain, and the observation sequence thus produced may not be *observationally consistent*. Informally, an observation sequence is observationally consistent if there exists a sequence of actions and an instantiation of the environment that could possibly produce that observation sequence. For example, assuming perfect sensors, it would be observationally inconsistent to observe at time step t that some device is permanently failed and at time step $t+1$ that it is operational.

Applying the stochastic sampling algorithm directly to planning problems can result in the generation of observationally inconsistent paths in the partial policy tree. And these paths, which are unsatisfied paths regardless of the setting of

the action variables chosen, need to be treated differently from observationally consistent paths, which have the potential to become satisfied paths, depending on the values chosen for the decision variables. If the algorithm includes observationally inconsistent paths in its estimate of the probability of success of a given policy, it will tend to underestimate this probability. The algorithm needs to either avoid generating observationally inconsistent paths in the first place, or ignore them in its calculations.

Considerations of efficiency suggest that the first strategy is to be preferred whenever possible, and this suggests two alterations to the stochastic sampling algorithm in order to make it applicable to SSAT encodings of planning problems:

- Sample tree paths only from P, the set of paths that are observationally consistent for some action sequence and instantiation of the environment.

- Adjust the evaluation of policy trees as follows: Instantiating the decision variables in the policy tree selects $P' \subseteq P$, the set of paths in P that are observationally consistent for that setting of decision variables and some instantiation of the environment. Let $S \subseteq P'$ be those paths in P' that are satisfied paths. Then, the probability of success of the contingent plan represented by that policy tree is $\frac{|S|}{|P'|}$.

There are various algorithmic issues that need to be addressed. First, I need to find an efficient way of constructing P and P'. One possibility is to use user-supplied information to prune at least some of the observationally inconsistent paths when constructing P. A second possibility is to construct encodings such that unit propagation can be used to efficiently detect observational inconsistencies.

Second, the algorithm returns a partial assignment strategy, or policy, that specifies how each decision variable should be set given the settings of the decision and observation variables that precede it in the quantifier ordering, but this is only specified for those situations represented by paths in the random sample used to construct the partial tree. 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. One possible strategy for addressing this issue would be an iterative approach that alternates planning and evaluation. In this approach, each iteration would perform an evaluation of the current policy (perhaps by simulating executions of the policy) and use that evaluation to guide the construction of a partial policy tree that would lead to a better policy. This is similar to approaches that identify the most severe problems in an imperfect plan and then attempt to correct them [33, 88].

One possible use for this (or any) approximation technique is to use the approximation technique in a framework that interleaves planning and execution, in order to scale up to even larger domains than approximation alone could attack. The idea here would be to use the approximation technique to calculate a "pretty good" first action (or action sequence), execute that action or action sequence, and then continue this planning/execution cycle from the new initial state. This approach could improve efficiency greatly (at the expense of optimality) by focusing the planner's efforts only on those contingencies that actually materialize.

8.4 Developing Planners With Less Independence

One of the hallmarks of AI planning research has been an insistence on forcing the planning algorithm to operate completely automatically and independently of the particular domain. There has been less insistence on this in OR approaches to planning, where planners are often engineered to take advantage of the peculiar characteristics of the problem being addressed. While it is certainly worthwhile to to push the planner to operate as automatically and independently as possible, the OR approach allows techniques to be applied to larger domains. Combining the best results of both approaches could be particularly productive.

Once one starts considering engineering planners to incorporate human knowledge about the problem being addressed, it is natural to consider the possibility of extended human-computer interaction: human-computer collaborative planning. My interest in this was sparked by a series of experiments I conducted in which minimal human input, in the form of additional constraints on the desired plan, had a significant positive impact on the efficiency of one of my planners. This suggests that a powerful planner could be built by formalizing a human-computer collaboration that combines human strengths (*e.g.* quickly recognizing possible constraints that productively narrow the space of plans to consider) and computer strengths (*e.g.* evaluating such suggestions rapidly and providing reasons for their failure if they do not work). Such a planning system would be particularly useful in situations in which humans need or desire to be part of the planning process.

8.5 A Unifying Framework for Planning and Scheduling Under Uncertainty

I believe this paradigm also has strong potential to provide a unifying framework for planning and scheduling under uncertainty. The line between these two areas is not clearly defined, but the solution of a scheduling problem tends to be less concerned with *which* action choices to make and more concerned with placing a required set of actions on a timeline such that resource constraints are respected and some metric is optimized. Many planning problems have a strong scheduling component (*e.g.* probabilistic logistics, emergency evacuation scenarios, and oil-spill management), and scheduling problems frequently require some planning (*e.g.* production line scheduling with exogenous events—such as machine breakdowns—that require a planned response).

Some very difficult, practical problems have characteristics of both planning and scheduling. Existing techniques, which tend to focus on planning issues *or* scheduling issues, find it difficult, if not impossible, to deal with these problems [102]. Although scheduling is a challenging area for the conversion-to-stochastic-satisfiability paradigm and raises some difficult new issues (*e.g.* expressing and reasoning about exogenous events, continuous time, resources, and metric quantities), successfully addressing these issues will considerably augment the expressiveness and power of the paradigm, and will produce techniques that can handle problems in which planning *and* scheduling are important components.

Probabilistic logistics and space-related applications are two important application areas for such techniques. New techniques to solve logistics problems under conditions of uncertainty are likely to be a significant factor in the success

of e-commerce. Billions of dollars are being spent on "last-mile delivery systems" to narrow the gap between the time of a customer's online order and its delivery, and probabilistic reasoning will be necessary in order to optimize the scheduling decisions made in these systems. Many problems in space-related applications (*e.g.* autonomous spacecraft, planetary rovers, and space-based observatories) would also benefit from a unified approach [102].

8.6 Summary and Discussion

ZANDER is a promising approach to contingent planning in partially observable stochastic domains. Even relatively basic implementations of this planning technique are competitive with other planners, and, as I have outlined in this chapter, there are a number of improvements and optimizations that show great potential for scaling up to larger problems: *e.g.* better data structures to optimize the application of heuristics, more compact and efficient SSAT encodings, encoding domain-specific knowledge, memoization for contingent planning, using learning to accelerate the solution process, and more sophisticated splitting heuristics. These improvements and extensions have already, in some cases, shown promise for improving ZANDER's performance significantly. I also outlined some possible approaches to approximation techniques and described initial efforts to develop one particularly promising approximation technique based on the randevalssat algorithm described in Section 5.2.3.

But, although my research provides evidence that the conversion-to-stochastic-satisfiability paradigm is a strong candidate in the search for an efficient, scalable probabilistic planning technique, it has also raised some general issues that argue for a broader perspective on planning under uncertainty in the real world.

In spite of the increased interest among researchers in developing planning techniques that explicitly reason about probabilities in the domain, it is an open question whether such a framework is best for real-world planning under uncertainty. First, this framework assumes that the probabilities describing the uncertainty in the domain are always known; this may not always be the case, particularly in domains where experience is limited or where certain events are rare and even lengthy experience will fail to provide an accurate estimate of their probability.

Second, probabilistic reasoning is very difficult and tends not to scale well. Is exact probabilistic planning affordable in the real world? Currently, the answer is "not in most cases." As I noted in Chapter 2, much research using MDPs and POMDPs as a framework has focused on approximation as a way of scaling up to larger problems. I will almost certainly need to resort to approximation techniques to scale up to very large problems. But, although approximation may allow us to scale up, it raises its own set of questions. When is approximation a good idea? Is it better to solve an approximation of the problem exactly, or approximately solve the exact problem?

Finally, there are situations in which, due to expense or threat to life, a plan that will succeed with "only" virtual certainty is unacceptable. In such situations, regardless of the probability of success, one must have a fast replanning capability to recover if the plan fails. The argument here is that calculating the probability of success is both costly and not worth much, since the agent must be prepared to replan anyway. In fact, some current approaches to planning under uncertainty construct a plan as if the environment were deterministic and then replan if the original plan fails. This type of approach works best in tightly-constrained domains in which efforts have been made to engineer out uncertainty and there

is usually time to replan (although, I would argue that, even in such domains, a planner that calculated and planned for the most likely contingencies would be valuable).

I am not suggesting that probabilistic planning is the wrong approach. I have spent the last three years developing new probabilistic planning techniques and none of my research suggests to me that this is the wrong road. But, although I think that probabilistic planning will be an important element in a successful approach to real-world planning under uncertainty, I think there are a number of important issues that need to be addressed. What these issues highlight is the need for a principled approach to planning under uncertainty in the real world, and we are still many dissertations away from satisfying that need.

Chapter 9

Conclusion

Planning is especially difficult under conditions of uncertainty. Many real-world problems, however, demand techniques that can be applied to large problems in situations where knowledge of the environment is imperfect and actions do not always have their intended effects. Research in developing techniques to cope with uncertainty in large domains is, therefore, critical to the acceptance and use of AI planning techniques in the real world. As I described in Chapter 2, researchers have developed a number of techniques for planning under uncertainty. All of these planners, however, suffer from various limitations: they can handle only certain types of uncertainty, they make assumptions that limit their applicability, or they cannot scale up to larger problems.

In this dissertation, I have described a new probabilistic planning technique that solves a very general class of planning problems at state-of-the-art speeds and has strong potential for scaling up to large problems. I showed that a probabilistic propositional planning problem can be solved by converting it into a stochastic satisfiability (SSAT) problem and solving that problem instead. I described efficient algorithms for both the conversion process and for solving the resulting SSAT problems. These algorithms exploit the structure of both a compact, factored state representation and a decision-tree action representation. ZANDER, the most advanced planner, can solve arbitrary, finite-horizon partially observable Markov decision processes, and operates at state-of-the-art speeds on contingent planning problems drawn from the literature. I also described an SSAT approximation algorithm that will form the basis of an SSAT-based approximate

planning technique.

It has often been said that as soon as a longstanding AI problem has been solved, people are quick to say "Oh, that's not really artificial intelligence." Chess is usually cited as the most recent example of this phenomenon. Planning under uncertainty remains a very active area of research and is in no immediate danger of being removed from the pantheon of "true" AI problems. But, although many difficult problems remain to be solved, significant progress has been made during the last five years. This dissertation represents a facet of that progress and has, I hope, moved the field a step closer to the day when planning under uncertainty is no longer regarded as "real" AI.

Appendix A

Caching in MAXPLAN

As the tests in the Section 6.5 indicated, MAXPLAN's use of memoization leads to a significant problem. Because it stores the value of all subformulas encountered in the solution process, MAXPLAN is very memory intensive and, in fact, is unable to find the best plan with horizon greater than 6 in the SLIPPERY-GRIPPER domain due to insufficient memory. The maximum plan horizon before exhausting memory for the MEDICAL-SEQUENCE and COFFEE-ROBOT-BLIND domains is 8, and the maximum horizon for the SAND-CASTLE-67 domain is 15. The situation for the SAND-CASTLE-67 domain is illustrated in Figure A.1, which compares the performance of full DPLL without memoization to that of modified DPLL (**UNIT/TIME**) with memoization. Note that this is a log plot and that results are shown starting with a plan horizon of 4 since the asymptotic behavior of the algorithm does not become clear until this point. (These tests, and all tests in this section, were run on a 143 MHz Sun UltraSparc with 128 Mbytes of RAM, running SunOS-5.7.)

The top plot shows the performance of full DPLL without memoization. We can extrapolate to estimate performance on larger problems (dotted line), but solution times become prohibitively long. The lower plot ending with an "X" shows the performance of modified DPLL with memoization. The much lower slope of this line (2.24 compared to 3.82 for full DPLL without memoization) indicates the superior performance of this algorithm, but the "X" indicates that no extrapolation is possible beyond horizon 15 due to insufficient memory. In fact, performance data for the horizon 15 plan already indicate memory problems.

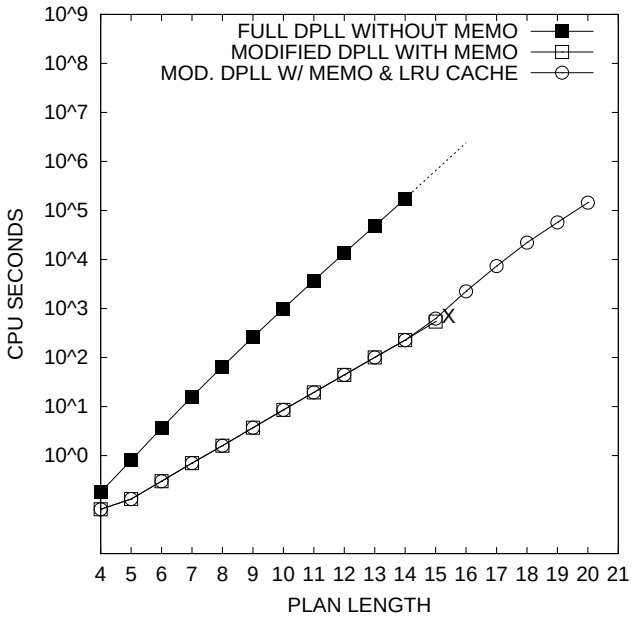


Figure A.1: Full DPLL without memoization on SAND-CASTLE-67 runs into a time bound. Modified DPLL with memoization runs into space limitations. Modified DPLL with memoization and LRU caching overcomes these time and space bounds.

The wall-clock time is more than three times the CPU time, indicating that the computation is I/O-bound. Memoization allows the algorithm to run orders of magnitude faster but ultimately limits the size of problems that can be solved.

A.0.1 LRU Caching

My solution was to treat the fixed amount of memory available as a *cache* for subformulas and their values. Given a cache size appropriate for the amount of memory on the machine running the algorithm, the problem becomes one of finding the best replacement policy for this cache. I compared two well-known cache replacement policies: first-in-first-out (FIFO) and least-recently-used (LRU). Both were implemented through a linked list of subformulas maintained in the order they were saved. When the cache is full, the algorithm merely removes the subformula at the head of the list. Under an LRU policy, however, whenever the algorithm

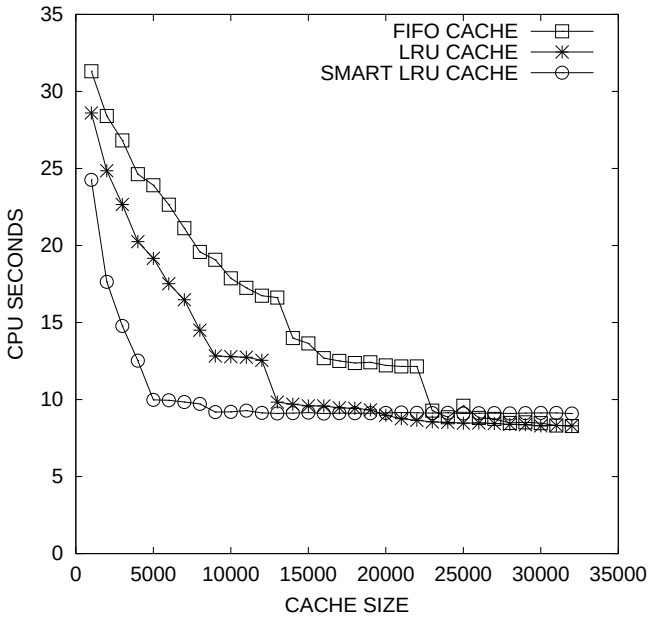


Figure A.2: Smart LRU caching for the 10-step SAND-CASTLE-67 problem allows the solver to use much less memory but still maintain performance.

uses a subformula it moves it to the end of the linked list. As shown in Figure A.2, the LRU cache outperforms the FIFO cache by an average of approximately 18% across the entire range of cache sizes for the 10-step SAND-CASTLE-67 plan.

I tested the LRU caching technique for generating SAND-CASTLE-67 plans with horizons ranging from 1 to 20. For plan horizons from 1 to 15, it was possible to make the cache large enough to save *all* subproblems without producing significant I/O problems. For larger plan horizons—with larger subformulas to be saved—I calculated cache size (expressed in maximum number of subproblems that could be saved) so as to keep total cache bytes approximately constant. Specifically, given a total cache size of C_T bytes for the largest plan horizon in which all subformulas could be saved, and a subproblem size of C_P bytes in a problem with a larger horizon, the cache size for the larger-horizon problem was calculated to be $\frac{C_T}{C_P}$ problems. (Note that, since I used a bit representation for

subproblems, all subproblems for a given planning problem require the same size storage space in the cache.)

These results are shown in Figure A.1; the performance plot for LRU caching is essentially coincident with the modified DPLL plot up to a plan horizon of 15. For longer plan horizons, LRU caching allows us to break through the memory insufficiency that blocked the algorithm before (the "X"), yet retain a significant degree of the improved performance that memoization provided. Solution times for full DPLL without memoization grow as $O(3.82^N)$, where T is the plan horizon. In contrast, modified DPLL with memoization scales as only $O(2.24^N)$, but can only solve problems of a bounded size. Modified DPLL with memoization and caching behaves like modified DPLL with memoization for small T, and then appears to grow as $O(2.96^N)$ once T is large enough for the cache replacement policy to kick in. Thus, the rate of increase for the variant with caching exhibits a growth rate comparable to that of modified DPLL with memoization while eliminating that algorithm's limitation on problem size—it trades away some performance for the ability to solve larger problems. As reported above, ENUM scales as $O(2.05^N)$ without any memory problems.

A.0.2 Smart LRU Caching

Sometimes performance can be improved by being selective about the subformulas cached. The hierarchical relationship among subformulas makes it redundant to save every subformula, but which ones should the algorithm save? Large subformulas are unlikely to be reused, and small subformulas, whose value the algorithm could quickly recompute, yield little time advantage. This suggests a strategy of saving mid-size subformulas, but experiments indicate that this approach fails for the SAND-CASTLE-67 problem. Another approach is to save

subformulas based on difficulty, defined as the number of recursive function calls required to solve that subformula. I experimentally determined that the optimal difficulty range for the SAND-CASTLE-67 problem was 5 to 14. Saving only those subformulas with a difficulty in this range allowed the algorithm to find the best 10-step plan for the SAND-CASTLE-67 problem with only an approximate 10% increase in CPU time, and this performance was nearly constant over a broad range of cache sizes (10,000 up to the maximum usable cache of 32,000); see Figure A.2.

Applying this "smart" LRU caching technique to plan generation in the SAND-CASTLE-67 domain, I obtained as much as a 37% decrease in CPU seconds compared to straight LRU caching (Figure A.3). Unfortunately, the advantages of smart caching seem to disappear as plan horizon increases. Also, it is not obvious how this technique could be exploited in a practical algorithm since I performed extensive tests to determine the optimal difficulty range, and this range is problem dependent.

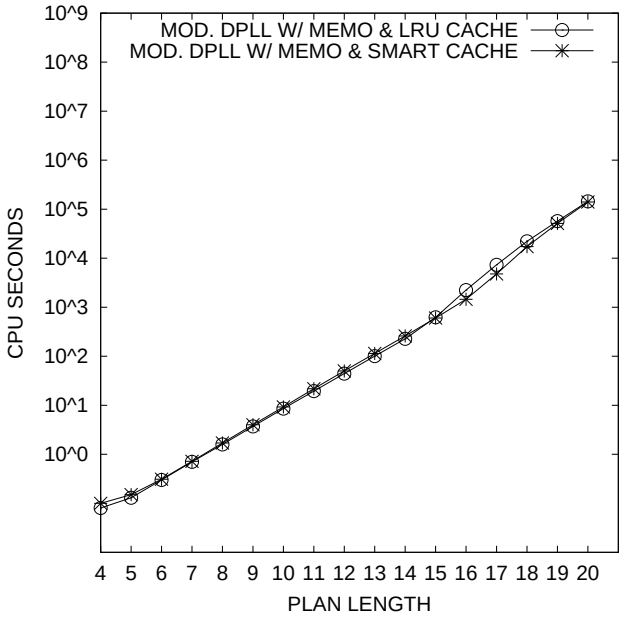


Figure A.3: Modified DPLL with memoization and LRU caching v. modified DPLL with memoization and smart LRU caching for SAND-CASTLE-67

Bibliography

[1] James Allen. Formal models of planning. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 50–54. Morgan Kaufmann, San Mateo, CA, 1990.

[2] Noga Alon, Joel H. Spencer, and Paul Erdos. *The Probabilistic Method.* John Wiley & Sons, New York, NY, 1992.

[3] Corin R. Anderson, David E. Smith, and Daniel S. Weld. Conditional effects in Graphplan. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning*, pages 44–53. AAAI Press, 1998.

[4] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37, San Francisco, CA, 1995. Morgan Kaufmann.

[5] Bruce W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[6] Andrew G. Barto, S. J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

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

[8] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[9] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA, 1996.

[10] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):279–298, 1997.

[11] Avrim L. Blum and John C. Langford. Probabilistic planning in the Graphplan framework. In *Proceedings of the Fifth European Conference on Plan-*

ning, pages 320–332, 1999.

[12] C. Boutilier, R. I. Brafman, and C. Geib. Structured reachability analysis for Markov decision processes. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–98)*, 1998.

[13] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[14] Craig Boutilier and Richard Dearden. Approximating value trees in structured dynamic programming. In Lorenza Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 54–62, 1996.

[15] Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1104–1113, 1995.

[16] Craig Boutilier and David Poole. Computing optimal policies for partially observable decision processes using compact representations. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1168–1175. AAAI Press/The MIT Press, 1996.

[17] Justin A. Boyan and Andrew W. Moore. Learning evaluation functions for global optimization and Boolean satisfiability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 3–10. The AAAI Press/The MIT Press, 1998.

[18] R. I. Brafman. A heuristic variable grid solution method for POMDPs. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 727–733, 1997.

[19] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:161–204, 1994.

[20] Anthony Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental Pruning: A simple, fast, exact method for partially observable Markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, pages 54–61, San Francisco, CA, 1997. Morgan Kaufmann Publishers.

[21] Anthony Rocco Cassandra. *Exact and Approximate Algorithms for Partially Observable Markov Decision Problems.* PhD thesis, Department of Computer Science, Brown University, May 1998.

[22] Anne Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, February 1992.

[23] Robert Cowell. Introduction to inference for Bayesian networks. In Michael I. Jordan, editor, *Learning in Graphical Models*, pages 9–26. The MIT Press, 1999.

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

[25] Robert H. Crites and Andrew G. Barto. Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1017–1023, Cambridge, MA, 1996. The MIT Press.

[26] P. Dagum and M. Luby. Approximating probabilistic inference in bayesian belief networks is NP-hard. *Artificial Intelligence*, 60(1):141–153, 1993.

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

[28] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[29] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1-2):35–74, 1995.

[30] Rina Dechter and Irina Rish. Directional resolution: The Davis-Putnam procedure, revisited. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pages 134–145, 1994.

[31] Eric V. Denardo. *Dynamic Programming: Models and Applications.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[32] Denise Draper, Steve Hanks, and Daniel Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of*

the AAAI Spring Symposium on Decision Theoretic Planning, pages 76–82, 1994.

[33] Mark Drummond and John Bresina. Anytime synthetic projection: Maximizing the probability of goal satisfaction. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 138–144. Morgan Kaufmann, 1990.

[34] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, Nagoya, Aichi, Japan, 1997.

[35] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971. Reprinted in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate, eds., Morgan Kaufmann, 1990.

[36] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

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

[38] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, CA, 1979.

[39] Robert P. Goldman and Mark S. Boddy. Conditional linear planning. In Kristian Hammond, editor, *The Second International Conference on Artificial Intelligence Planning Systems*, pages 80–85. The AAAI Press / The MIT Press, 1994.

[40] Robert P. Goldman and Mark S. Boddy. Representing uncertainty in simple planners. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pages 238–245, 1994.

[41] Geoffrey J. Gordon. Stable function approximation in dynamic programming. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, San Francisco, CA, 1995. Morgan Kaufmann.

[42] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence*, pages 219–239, Washington, D.C., 1969.

[43] Steve Hanks and Drew McDermott. Modeling a dynamic and uncertain world I: Symbolic and probabilistic reasoning about change. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1993.

[44] Eric A. Hansen. Cost-effective sensing during plan execution. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1029–1035. AAAI Press/The MIT Press, 1994.

[45] Eric A. Hansen. *Finite-Memory Control of Partially Observable Systems.* PhD thesis, University of Massachusetts, 1998.

[46] Eric A. Hansen and Shlomo Zilberstein. Heuristic search in cyclic AND/OR graphs. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 412–418. The AAAI Press/The MIT Press, 1998.

[47] Eric A. Hansen and Shlomo Zilberstein. Solving Markov decision problems using heuristic search. In *Proceedings of the AAAI Spring Symposium on Search Techniques for Problem Solving Under Uncertainty and Incomplete Information*, pages 42–47, Stanford, CA, 1999.

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

[49] Milos Hauskrecht. Incremental methods for computing bounds in partially observable Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 734–739, 1997.

[50] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[51] Ronald A. Howard. *Dynamic Programming and Markov Processes.* The MIT Press, Cambridge, Massachusetts, 1960.

[52] Russell Impagliazzo, Michael L. Littman, and Toniann Pitassi. On the complexity of counting satisfying assignments. Submitted, 2000.

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

[54] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.

[55] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[56] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, pages 374–385, 1996.

[57] Henry Kautz, David McAllester, and Bart Selman. Exploiting variable dependency in local search. In *Abstracts of the Poster Sessions of IJCAI-97, Nagoya, Japan*, 1997.

[58] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 359–363, 1992.

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

[60] Henry Kautz and Bart Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working Notes of the Workshop on Planning as Combinatorial Search*, pages 58–60, 1998. Held in conjunction with the Fourth International Conference on Artificial Intelligence Planning.

[61] Henry Kautz and Bart Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning*, pages 181–189. AAAI Press, 1998.

[62] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In

Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, pages 1324–1331. The AAAI Press/The MIT Press, 1999.

[63] Daphne Koller and Ronald Parr. Computing factored value functions for policies in structured MDPs. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1332–1339. The AAAI Press/The MIT Press, 1999.

[64] Daphne Koller and Ronald Parr. Policy iteration for factored MDPs. In *Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI 2000)*, 2000.

[65] Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, September 1995.

[66] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 366–371, Nagoya, Aichi, Japan, 1997.

[67] Michael L. Littman. Solving partially observable Markov decision processes via VFA. In Justin A. Boyan, Andrew W. Moore, and Richard S. Sutton, editors, *Proceedings of the Workshop on Value Function Approximation, Machine Learning Conference 1995, Technical report CMU-CS-95-206*, 1995.

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

[69] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In Armand Prieditis and Stuart Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370. Morgan Kaufmann, San Francisco, CA, 1995. Reprinted in *Readings in Agents*, M. H. Huhns and M. P. Singh, eds., Morgan Kaufmann, 1998.

[70] Michael L. Littman, Judy Goldsmith, and Martin Mundhenk. The computational complexity of probabilistic plan existence and evaluation. *Journal of Artificial Intelligence Research*, 9:1–36, 1998.

[71] Michael L. Littman, Stephen M. Majercik, and Toniann Pitassi. Stochastic Boolean satisfiability. *Journal of Automated Reasoning*, 2000. In press.

[72] William S. Lovejoy. A survey of algorithmic methods for partially observable Markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[73] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 541–548. The AAAI Press/The MIT Press, 1999.

[74] A. Mahanti and A. Bagchi. AND/OR graph heuristic search methods. *Journal of the ACM*, 32(1):28–51, 1985.

[75] Stephen M. Majercik and Michael L. Littman. MAXPLAN: A new approach to probabilistic planning. In Reid Simmons, Manuela Veloso, and Stephen Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 86–93. AAAI Press, 1998.

[76] Stephen M. Majercik and Michael L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 954–959. The AAAI Press/The MIT Press, 1998.

[77] Stephen M. Majercik and Michael L. Littman. Contingent planning under uncertainty via stochastic satisfiability. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 549–556. The AAAI Press/The MIT Press, 1999.

[78] Amol D. Mali and Subbarao Kambhampati. On the utility of plan-space (causal) encodings. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 557–563. The AAAI Press/The MIT Press, 1999.

[79] Alberto Martelli and Ugo Montanari. Optimizing decision trees through heuristically guided search. *Communications of the ACM*, 21(12):1025–1039, 1978.

[80] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *Proceedings of the 9th National Conference on Artificial Intelligence*, pages

634–639, 1991.

[81] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 321–326. AAAI Press/The MIT Press, 1997.

[82] George E. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, January 1982.

[83] Andrew W. Moore and Christopher G. Atkeson. Memory-based reinforcement learning: Efficient computation with prioritized sweeping. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 263–270, San Mateo, CA, 1993. Morgan Kaufmann.

[84] Martin Mundhenk, Judy Goldsmith, and Eric Allender. The complexity of policy-evaluation for finite-horizon partially-observable Markov decision processes. In *Proceedings of the 25th Symposium on Mathematical Foundations of Computer Science (published in Lecture Notes in Computer Science #1295)*, pages 129–138. Springer-Verlag, 1997.

[85] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, CA, 1980.

[86] Nilufer Onder, 1998. Personal communication.

[87] Nilufer Onder, 2000. Personal communication.

[88] Nilufer Onder and Martha E. Pollack. Contingency selection in plan generation. In *Proceedings of the Fourth European Conference on Planning: Recent Advances in AI Planning*, pages 364–376, 1997.

[89] Nilufer Onder and Martha E. Pollack. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 577–584. The AAAI Press/The MIT Press, 1999.

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

[91] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, MA, 1994.

[92] Ronald Parr and Stuart Russell. Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1088–1095, 1995.

[93] J. S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, pages 103–114, 1992.

[94] Mark A. Peot and David E. Smith. Conditional nonlinear planning. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197, 1992.

[95] Mark Alan Peot. *Decision-Theoretic Planning.* PhD thesis, Department of Engineering-Economic Systems and Operations Research, Stanford University, May 1998.

[96] Martin L. Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted Markov decision processes. *Management Science*, 24:1127–1137, 1978.

[97] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986.

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

[99] Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the Fortieth Annual IEEE Symposium on Foundations of Computer Science*, pages 410–414, 1999.

[100] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability*, pages 521–531. American Mathematical Society, 1996. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26.

[101] Satinder P. Singh and Dimitri Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. In Michael C. Mozer, Michael I. Jordan, and Thomas Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 974–980. The MIT Press, 1997.

[102] David E. Smith, Jeremy Frank, and Ari K. Jonsson. Bridging the gap between planning and scheduling. *To appear in Knowledge Engineering Review*, 15(1), 2000.

[103] David E. Smith and Daniel S. Weld. Conformant Graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 889–896. AAAI Press/The MIT Press, 1998.

[104] Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[105] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[106] Jonathan Tash and Stuart Russell. Control strategies for a stochastic planner. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 1079–1085, 1994.

[107] Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

[108] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, pages 58–67, March 1995.

[109] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In M. C. Mozer, P. Smolensky, D. S. Touretzky, J. L. Elman, and A. S. Weigend, editors, *Proceedings of the Fourth Connectionist Models Summer School*, Hillsdale, NJ, 1993. Lawrence Erlbaum.

[110] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

[111] D. Warren. Generating conditional plans and programs. In *Proceedings of the Summer Conference on AI and Simulation of Behavior*, pages 344–354. University of Edinburgh, 1976.

[112] R. Washington. Incremental Markov-model planning. In *Proceedings of TAI-96, Eighth IEEE International Conference on Tools With Artificial Intelligence*, pages 41–47, 1996.

[113] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 897–904. AAAI Press/The MIT Press, 1998.

[114] Chelsea C. White, III. Partially observed Markov decision processes: A survey. *Annals of Operations Research*, 32, 1991.

[115] Chelsea C. White, III and William T. Scherer. Solution procedures for partially observed Markov decision processes. *Operations Research*, 37(5):791–797, September-October 1989.

[116] Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1–3):277–296, 2000.

[117] Nevin L. Zhang and Wenju Liu. A model approximation scheme for planning in stochastic domains. *Journal of Artificial Intelligence Research*, 7:199–230, 1997.

[118] Wei Zhang and Thomas G. Dietterich. A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intellience*, pages 1114–1120, 1995.

Biography

Stephen Michael Majercik was born 23 July 1955 in Chicago, Illinois, USA. His first brush with computers occurred in high school, when he wrote a program (on punch cards!) to multiply two numbers. Although he thought it was one of the neatest things he had ever done, and although he came *this close* to concentrating in computer science at Harvard, his interest in computers lay dormant for about twenty years. He returned to computer science in 1990, when he began taking courses at the University of Southern Maine to obtain his master's degree. By the time he received his master's degree, he realized that his love of teaching made a Ph.D. necessary, and he began work on this last degree at Duke University in 1994. Stephen will be joining the faculty of the Computer Science Department at Bowdoin College in the fall of 2000.

Research Interests

Efficient algorithms for planning, reasoning, and learning in uncertain environments (uncertain initial conditions, probabilistic effects of actions, uncertain state estimation), scheduling under uncertainty, human-computer collaborative planning, model-theoretic planning, decision-theoretic planning, satisfiability and stochastic satisfiability, constraint satisfaction programming, Markov decision processes, partially observable Markov decision processes, reinforcement learning, and belief networks.

Employment

- Assistant Professor, Computer Science Department, Bowdoin College, Brunswick, Maine, Fall, 2000.
- Research Assistant, Duke University, Durham, North Carolina, 1997–2000.
- Graduate Research Assistant, Glaxo Wellcome, Inc., Research Triangle Park, North Carolina, 1996–1997.
- Instructor, University of Southern Maine, Portland, Maine, 1993–1994.
- Business Manager, Portland Stage Company, Portland, Maine, 1986–1987.
- Financial Analyst, Tri-Star Pictures, Inc., New York, New York, 1983–1986.
- Associate Senior Business Analyst, Westinghouse Broadcasting and Cable, Inc., New York, New York, 1981–1983.

Education

- Ph.D. in Computer Science, Duke University, 2000.
  Thesis Title: Planning Under Uncertainty via Stochastic Satisfiability
  Advisor: Michael L. Littman
- M.S. in Computer Science, University of Southern Maine, 1994.
  Thesis Title: Structurally Dynamic Cellular Automata
  Advisor: Stephen A. Fenner
- M.B.A. in Finance, Yale School of Management, 1981.
- M.F.A. in Theatre Administration, Yale School of Drama, 1981.
- A.B. *cum laude* in Government, Harvard University, 1977.

Refereed Publications

- Stephen M. Majercik and Michael L. Littman. Approximate Planning in the Probabilistic-Planning-as-Stochastic-Satisfiability Paradigm. In *Second International NASA Workshop on Planning and Scheduling for Space*, 2000.
- Michael L. Littman, Stephen M. Majercik, and Toniann Pitassi. Stochastic Boolean satisfiability. To appear in the *Journal of Automated Reasoning.*
- Stephen M. Majercik and Michael L. Littman. Contingent planning under uncertainty via stochastic satisfiability. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 549–556, 1999.
- Stephen M. Majercik. Planning under uncertainty via stochastic satisfiability. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, page 950, 1999. Presented at the SIGART/AAAI–99 Doctoral Consortium.
- Stephen M. Majercik and Michael L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 954–959, 1998.
- Stephen M. Majercik and Michael L. Littman. MAXPLAN: A new approach to probabilistic planning. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 86–93, 1998.
- Michael L. Littman and Stephen M. Majercik. Large-Scale Planning Under Uncertainty: A Survey. In *International Workshop on Planning and Scheduling for Space Exploration and Science*, pages 27:1–8, 1997.

Unrefereed Publications

- Stephen M. Majercik and Michael L. Littman. ZANDER: A model-theoretic approach to planning in partially observable stochastic domains. In *Working Notes of the Workshop on Model-Theoretic Approaches to Planning*, pages 48–54, 2000. Held in conjunction with AIPS–00.
- Stephen M. Majercik. C-MAXPLAN: Contingent planning in the MAXPLAN framework. In *AAAI Spring Symposium on Search Techniques for Problem Solving Under Uncertainty and Incomplete Information*, pages 83–88, 1999.
- Stephen M. Majercik and Michael L. Littman. MAXPLAN: A new approach to probabilistic planning. In *AAAI Fall Symposium on Planning with Partially Observable Markov Decision Processes*, pages 121–128, 1998.
- Stephen M. Majercik and Michael L. Littman. Probabilistic planning with MAXPLAN. In *Working Notes of the Workshop on Planning as Combinatorial Search*, pages 85–88, 1998. Held in conjunction with AIPS–98.
- Stephen M. Majercik and Michael L. Littman. Reinforcement learning for selfish load balancing in a distributed memory environment. In *Proceedings of the International Conference on Information Sciences*, volume 2, Paul Wang editor, pages 262–265, 1997.
- Stephen M. Majercik. Structurally dynamic cellular automata. Master's Thesis, Department of Computer Science, University of Southern Maine, 1994.

Invited Talks

- *Planning Under Uncertainty via Stochastic Satisfiability*, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1999.
- *Planning Under Uncertainty via Stochastic Satisfiability*, Honeywell Technology Center, Minneapolis, Minnesota, 1999.
- *Probabilistic Planning as Probabilistic Satisfiability*, NASA Ames Research Center, Moffett Field, California, 1999.

Teaching Experience

- Teaching Assistant and Guest Lecturer, Duke University, 1995–1998.
- Instructor, University of Southern Maine, 1993–1994.

- Teaching Assistant, University of Southern Maine, 1990–1993.

Fellowships and Honors

- NASA Graduate Student Research Program Fellowship, NASA Ames Research Center, 1998–2000.
- Honorable Mention, Department of Defense Graduate Fellowship Competition, 1995.
- Computer Science Department Fellowship, Duke University, 1994–95.
- Honorable Mention, National Science Foundation Graduate Research Fellowship Competition, 1994.
- Phi Kappa Phi Honor Society, University of Southern Maine, 1993.
- Senior Thesis awarded honors *magna cum laude*, Harvard University, 1977.

Professional Activities

- Triangle Area Neural Network Society, Membership Chair, 1998–2000.
- Journal of Computational Intelligence, Referee, 1999.
- International Joint Conference on Artificial Intelligence, Referee, 1999.
- Joint Conference on the Science and Technology of Intelligent Systems (ISIC/CIRA/ISAS), Referee, 1998.

University Service

- Mentor, Computer Science Department, Duke University, 1998.
- Graduate Student Voting Member, Graduate Admissions Committee, Computer Science Department, Duke University, 1998. (Note: A single student was nominated by the graduate students and approved by the faculty.)
- Member, Student Faculty Search Committee, Computer Science Department, Duke University, 1996.
- Department Photographer, Computer Science Department, Duke University, 1995–1997.