MAXPLAN: A New Approach to Probabilistic Planning*

Stephen M. Majercik and **Michael L. Littman**

Department of Computer Science
Duke University
Durham, NC 27708-0129
{majercik,mlittman}@cs.duke.edu

Abstract

Classical artificial intelligence planning techniques can operate in large domains but traditionally assume a deterministic universe. Operations research planning techniques can operate in probabilistic domains but break when the domains approach realistic sizes. MAXPLAN is a new probabilistic planning technique that aims at combining the best of these two worlds. MAXPLAN converts a planning instance into an E-MAJSAT instance, and then draws on techniques from Boolean satisfiability and dynamic programming to solve the E-MAJSAT instance. E-MAJSAT is an NP^{PP}-complete problem that is essentially a probabilistic version of SAT. MAXPLAN performs as much as an order of magnitude better on some standard stochastic test problems than BURIDAN—a state-of-the-art probabilistic planner—and scales better on one test problem than two algorithms based on dynamic programming.

INTRODUCTION

Classical artificial intelligence planning techniques can operate in large domains but, traditionally, assume a deterministic universe. Operations research planning techniques can operate in probabilistic domains, but algorithms for solving Markov decision processes (MDPs) and partially observable MDPs are capable of solving problems only in relatively small domains. Research in probabilistic planning aims to explore a middle ground between these two well-studied extremes with the hope of developing systems that can reason efficiently about plans in complex, uncertain domains.

In this paper we introduce MAXPLAN, a new approach to probabilistic planning. MAXPLAN converts a planning problem into an E-MAJSAT problem, an NP^{PP}-complete problem that is essentially a probabilistic version of SAT. We draw on techniques from Boolean satisfiability and dynamic programming to solve the resulting E-MAJSAT problem. In the first section, we discuss complexity results that motivated our research strategy, and compare MAXPLAN to SATPLAN, a similar planning technique for deterministic domains. The next three sections contain the details of MAXPLAN's operation: the planning domain representation, the conversion of problems to E-MAJSAT form, the solution of E-MAJSAT problems and results. The next section compares MAXPLAN to three other planning techniques and the final sections discuss future work and conclusions.

COMPLEXITY RESULTS

A probabilistic planning domain is specified by a set of states, a set of actions, an initial state, and a set of goal states. The output of a planning algorithm is a controller for the planning domain whose objective is to reach a goal state with sufficiently high probability. In its most general form, a plan is a *program* that takes as input observable aspects of the environment and produces actions as output. We classify plans by their *size* (the number of internal states) and *horizon* (the number of actions produced *en route* to a goal state). In a *propositional* planning domain, states are specified as assignments to a set of propositional variables.

If we place reasonable bounds—polynomial in the size of the planning problem—on both plan size and plan horizon, the planning problem is NP^{PP}-complete (Goldsmith, Littman, & Mundhenk 1997) (perhaps easier than PSPACE-complete) and may be amenable to heuristics. For a survey of relevant results, see (Littman, Goldsmith, & Mundhenk 1998). Membership in this complexity class suggests a solution strategy analogous to that of SATPLAN (Kautz & Selman 1996), a successful deterministic planner that converts a planning problem into a satisfiability (SAT) problem and solves the SAT problem instead. In the same way that deterministic planning can be expressed as the NP-complete problem SAT, probabilistic planning can be expressed as the NP^{PP}-complete problem E-MAJSAT (Littman, Goldsmith, & Mundhenk 1998):

> Given a Boolean formula with *choice variables* (variables whose truth status can be arbitrarily set) and *chance variables* (variables whose truth status is determined by a set of independent probabilities), find the setting of the choice variables that maximizes the probability of a satisfying assignment with respect to the chance variables.

As we will discuss below, the choice variables can be made to correspond to a possible plan, while the chance variables can be made to correspond to the uncertainty in the planning domain. Thus, our research strategy is

*Appears in the *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 86–93, AAAI Press, 1998.

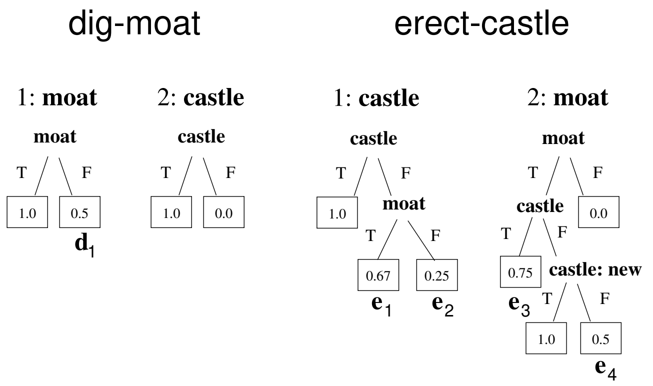

Figure 1: The sequential-effects-tree representation of SAND-CASTLE-67 consists of a set of decision trees.

to show that we can efficiently turn planning problems into E-MAJSAT problems, and then focus our efforts on finding algorithms to solve E-MAJSAT.

PROBLEM REPRESENTATION

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 probabilistic 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 with probability above some threshold θ.

In this work, we assume a completely unobservable domain—the effects of previous actions cannot be used in selecting the current action. This is a special case of the partially observable MDP 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 sequences of actions. A future direction is to extend our approach to completely or partially observable domains, in which more complex plan structures will be needed.

MAXPLAN represents planning domains in the sequential-effects-tree (ST) representation (Littman 1997), which is a syntactic variant of two-time-slice Bayes nets (2TBNs) with conditional probability tables represented as trees (Boutilier, Dearden, & Goldszmidt 1995). This representation is expressively equivalent to 2TBNs and probabilistic state-space operators, so it is likely that a similar planner could be built based on any of these representations.

In ST, the effect of each action on each proposition is represented as a separate decision tree. For a given action a, each of the decision trees for the different propositions is ordered, so the decision tree for one proposition can refer to both the new and old values of previous propositions. The leaves of a decision tree describe how the associated proposition changes as a function of the state and action, perhaps probabilistically. A formal description of ST is available elsewhere (Littman 1997); for brevity, we present the representation in the form of an example (see Figure 1).

SAND-CASTLE-67, a simple probabilistic planning domain, is concerned with building a sand castle. There are four states, described by combinations of two Boolean propositions, **moat** and **castle** (propositions appear in boldface). The proposition **moat** signifies that a moat has been dug, 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**} (a goal state is any state in which all the propositions in the goal set are `True`).

There are two actions: dig-moat and erect-castle (actions appear in sans serif). The first decision tree in the dig-moat action describes the effect of this action on the proposition **moat**. If **moat** is already `True` (left branch), dig-moat leaves it unchanged; if **moat** is `False` (right branch), dig-moat causes **moat** to become `True` with probability 0.5. The second decision tree in the dig-moat action indicates that this action has no impact on the proposition **castle**.

The second action is erect-castle. Note that here the first decision tree describes the impact of this action on the proposition **castle** rather than the proposition **moat**; this is necessary because the new value of **castle** affects the new value of **moat** (as explained below). The proposition **castle** 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 building a moat protects the castle from being destroyed prematurely by waves.

The second decision tree describes the effect of the erect-castle action on the proposition **moat**. The idea here is that the process of erecting the castle, whether successful or not, may destroy an existing moat. Thus, the **moat** decision tree indicates that if the castle was already built when erect-castle was selected, the moat remains intact with probability 0.75. If the castle had not been built, but erect-castle successfully builds it (the label **castle:new** in the diagram refers to the value of the **castle** proposition *after* the first decision tree is evaluated) **moat** remains `True`. If erect-castle fails to build a castle, **moat** remains `True` with probability 0.5. Finally, because erect-castle cannot cause a moat to be dug, there is no effect when **moat** is `False`.

PROBLEM CONVERSION

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 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. The converter does this by selecting a plan horizon N, time-indexing each proposition and action so the planner can reason about what happens

(not(**moat**-0)) ∧ (not(**castle**-0)) ∧ (**castle**-1) ∧
(dig-moat-1 ∨ erect-castle-1) ∧ (not(dig-moat-1) ∨ not(erect-castle-1)) ∧
(not(dig-moat-1) ∨ not(**moat**-0) ∨ **moat**-1) ∧ (not(dig-moat-1) ∨ **moat**-0 ∨ not($\mathbf{d}_1$-1) ∨ **moat**-1) ∧
(not(dig-moat-1) ∨ **moat**-0 ∨ $\mathbf{d}_1$-1 ∨ not(**moat**-1)) ∧
(not(erect-castle-1) ∨ not(**moat**-0) ∨ not(**castle**-0) ∨ not($\mathbf{e}_3$-1) ∨ **moat**-1) ∧
(not(erect-castle-1) ∨ not(**moat**-0) ∨ not(**castle**-0) ∨ $\mathbf{e}_3$-1 ∨ not(**moat**-1)) ∧
(not(erect-castle-1) ∨ not(**moat**-0) ∨ **castle**-0 ∨ not(**castle**-1) ∨ **moat**-1) ∧
(not(erect-castle-1) ∨ not(**moat**-0) ∨ **castle**-0 ∨ **castle**-1 ∨ not($\mathbf{e}_4$-1) ∨ **moat**-1) ∧
(not(erect-castle-1) ∨ not(**moat**-0) ∨ **castle**-0 ∨ **castle**-1 ∨ $\mathbf{e}_4$-1 ∨ not(**moat**-1)) ∧
(not(erect-castle-1) ∨ **moat**-0 ∨ not(**moat**-1)) ∧ (not(dig-moat-1) ∨ not(**castle**-0) ∨ **castle**-1) ∧
(not(dig-moat-1) ∨ **castle**-0 ∨ not(**castle**-1)) ∧ (not(erect-castle-1) ∨ not(**castle**-0) **castle**-1) ∧
(not(erect-castle-1) ∨ **castle**-0 ∨ not(**moat**-0) ∨ not($\mathbf{e}_1$-1) ∨ **castle**-1) ∧
(not(erect-castle-1) ∨ **castle**-0 ∨ not(**moat**-0) ∨ $\mathbf{e}_1$-1 ∨ not(**castle**-1)) ∧
(not(erect-castle-1) ∨ **castle**-0 ∨ **moat**-0 ∨ not($\mathbf{e}_2$-1) ∨ **castle**-1) ∧
(not(erect-castle-1) ∨ **castle**-0 ∨ **moat**-0 ∨ $\mathbf{e}_2$-1 ∨ not(**castle**-1)) ∧

Figure 2: The CNF formula for a 1-step SAND-CASTLE-67 plan constrains the variable assignments.

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 N,
- actions at time t are mutually exclusive ($1 \leq t \leq N$),
- 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` ($1 \leq t \leq N$).

The first two conditions are not probabilistic and can be encoded in a straightforward manner (Blum & Furst 1997), but the third condition is complicated by the fact that chance variables sometimes intervene between actions and their effects on propositions. We will illustrate the conversion process by describing the construction of the CNF formula corresponding to a one-step plan for SAND-CASTLE-67.

Variables

We first introduce a set of propositions that capture the randomness is the domain. For each decision-tree leaf l labeled with a probability π_l that is strictly between 0.0 and 1.0, we create 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 1), $\mathbf{d}_1$ is a random proposition that is `True` with probability 0.5. We then replace the leaf l with a node labeled $\mathbf{r}_l$ with 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.

We create variables to record the status of actions and propositions in an N-step plan by taking three cross products: actions and time steps 1 through N, propositions and time steps 0 through N, and random propositions and time steps 1 through N. The total number of variables in the CNF formula is

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

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.

The variables generated by the propositions for time steps 1 through N must also be chance variables, since their values are not completely determined by the plan. The probability associated with each of these variables is 0.5 so that all possible assignments to these variables have the same probability mass. Note that this lowers the probability of any choice variable setting by a factor of 0.5^{PN} and, thus, a suitable adjustment to the probabilities calculated by the solver must be made.

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 (not(**moat**-0)) and (not(**castle**-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 containing all actions (one action must be taken) and $\binom{A}{2}$ clauses that enforce the requirement that two actions not be taken simultaneously. In our example, the clauses generated are (dig-moat-1 ∨ erect-castle-1) and (not(dig-moat-1) ∨ not(erect-castle-1)).

The third condition—effects of actions on propositions—generates a clause for each path through each decision tree in each action. For example, the right-most path in the **moat** decision tree of the action dig-moat (Figure 1) specifies that if we take the dig-moat action and we do not already have a **moat**, then we will have a moat at the next time step with probability 0.5. This generates the implication:

$$\textsf{dig-moat-1} \wedge \text{not}(\mathbf{moat}\text{-}0) \wedge \mathbf{d}_1\text{-}1 \rightarrow \mathbf{moat}\text{-}1,$$

Note that a chance variable has the same time index as the action it modifies. This yields the disjunction:

$$\text{not}(\textsf{dig-moat-1}) \vee \mathbf{moat}\text{-}0 \vee \text{not}(\mathbf{d}_1\text{-}1) \vee \mathbf{moat}\text{-}1.$$

Figure 2 shows the complete formula for a 1-step plan.

The total number of clauses generated by the third condition is bounded by $2N\sum_{i=1}^{A} L_i$ 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

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

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.

SOLUTION AND RESULTS

We next describe an algorithm for solving the E-MAJSAT problem produced by the conversion described above. We need to find the assignment to the choice variables that maximizes the probability of a satisfying assignment; for plan-generated formulas, such an assignment is directly interpretable as an optimal straight-line plan. Our algorithm is based on an extension of the Davis-Putnam-Logemann-Loveland (DPLL) procedure for determining satisfiability (Davis, Logemann, & Loveland 1962). To our knowledge, DPLL (and its variants) is the best systematic satisfiability solver known. As such, DPLL was the obvious choice as a basis for the E-MAJSAT solver, which needs to determine all possible satisfying assignments, sum the probabilities of the satisfying assignments for each possible choice-variable assignment, and then return the choice-variable assignment (plan) with the highest probability of producing a satisfying assignment (goal satisfaction). Although we eventually modified the DPLL algorithm significantly, these modifications provides insight into the operation of the E-MAJSAT solver.

Determining all the satisfying assignments can be envisioned as constructing a binary tree in which each node represents a choice (chance) variable and the subtrees represent the subproblems given the two possible assignments (outcomes) to the parent choice (chance) variable. It is critical to construct an efficient tree to avoid evaluating an exponential number of assignments. 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,
- the *current CNF subformula* is uniquely specified by the current set of active clauses and the set of variables that appear in those clauses, and
- the *value* of a CNF subformula is

$$\max_{\mathbf{D}} \sum_{\mathbf{S}} \left(\prod_{v_i \in \mathbf{CH}} \pi_i^{v_i} (1 - \pi_i)^{1 - v_i} \right),$$

where $\mathbf{D}$ is the set of all possible assignments to the active choice variables, $\mathbf{S}$ is the set of all satisfying assignments to the active chance variables, $\mathbf{CH}$ is the set of all chance variables, and π_i is the probability that chance variable i is `True`.

To prune the number of assignments that must be considered, we do the following, whenever possible:

- select an active variable that no longer appears in any active clause 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**),[1] 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 (purification, **PURE**).

Each of these actions can be shown to leave the value of the current CNF formula unchanged.

When there are no more irrelevant variables, unit clauses, or pure variables, we must select and split on an active variable (always preferring choice variables). If we split on a choice (chance) variable, we return the maximum (probability weighted average) of assigning `True` to the variable and recursing or assigning `False` and recursing. The splitting heuristic used is critical to the efficiency of the algorithm (Hooker & Vinay 1994). The possibilities we tested were:

- Choose the next active variable from an initially random ordering of the variables,
- Choose the active variable that would satisfy the most active clauses,
- Choose the active variable that would appear earliest in the plan (time-ordered),

The graph in Figure 3 compares the performance of these heuristics on the following SAND-CASTLE-67 problem: 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 plan horizon, the time-ordered splitting heuristic, given a fixed amount of time, is able to evaluate longer plans than the other two heuristics.

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

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

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

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

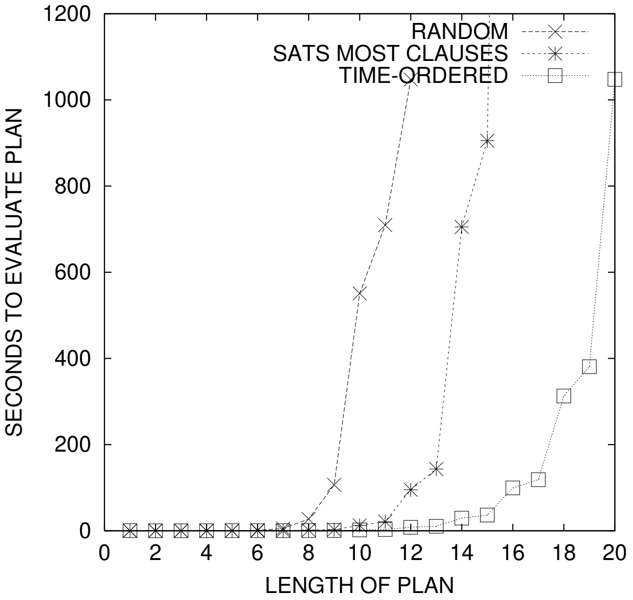

Figure 3: For DPLL on plan evaluation, the time-ordered splitting heuristic works best.

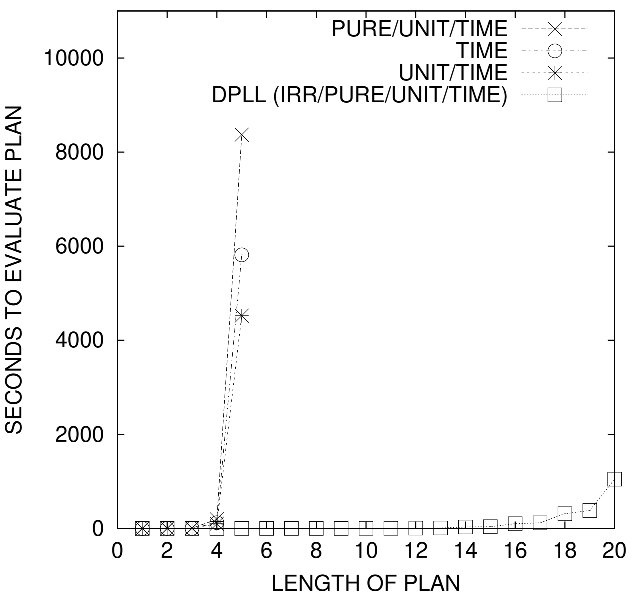

Figure 4: Full DPLL on plan evaluation without memoization performs better than DPLL with various elements removed.

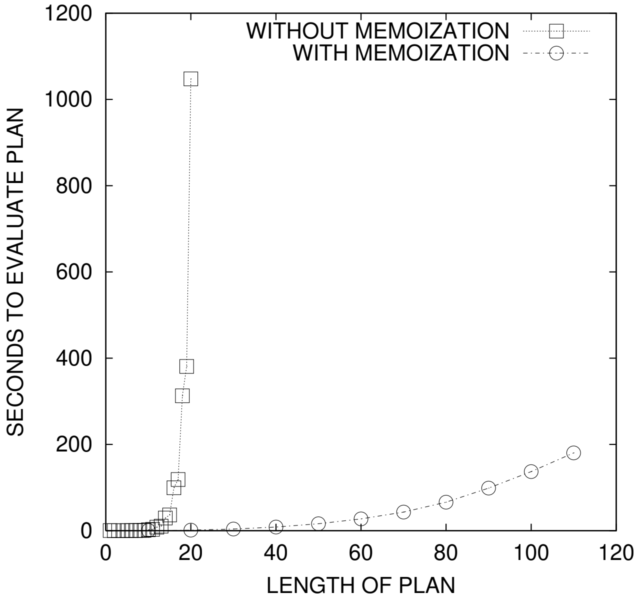

Figure 5: Full DPLL on plan evaluation runs faster with memoization.

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 us 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 we can feasibly evaluate (Figure 5). With memoization, we can evaluate a 110-step plan in less time than it took to evaluate an 18-step plan previously.

The behavior of the algorithm suggested, however, that DPLL was hindering efficient subproblem reuse, so we gradually removed the DPLL elements as we had done before. This time we found that removing DPLL elements greatly *improved* performance. As the graph in Figure 6 indicates, best performance is achieved with unit propagation and time-ordered splitting, or time-ordered splitting alone. Tests on other problems indicate that, in general, unit propagation and time-ordered splitting provide the best performance.

Having identified a set of promising algorithmic components, we tested the resulting system on the full plan-generation problem in SAND-CASTLE-67 for plan horizons ranging from 1 to 10. Optimal plans found by MAXPLAN exhibit a rich structure: beyond length 3, the plan of length i is not a subplan of the plan of length $i+1$. 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 near certainty (probability 0.9669) and MAXPLAN finds this plan in approximately 12 seconds on a Sun Ultra-1 Model 140 with 128 Mbytes of memory.

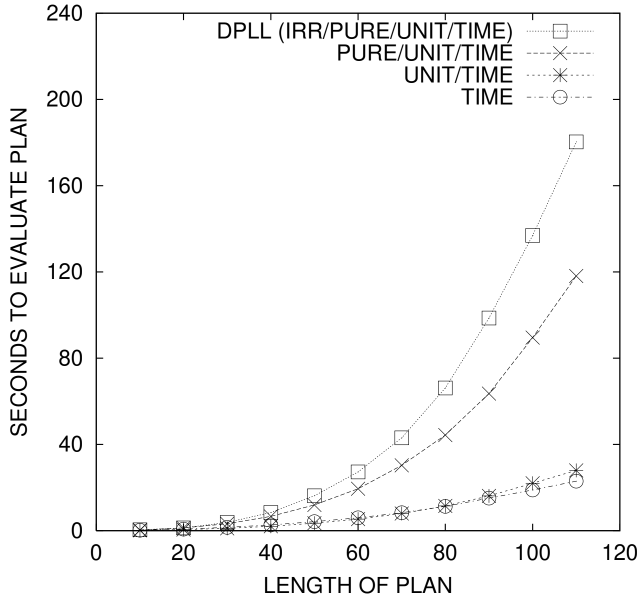

Figure 6: Full DPLL on plan evaluation with memoization performs worse than DPLL with various elements removed.

COMPARISON TO OTHER PLANNING TECHNIQUES

We compared MAXPLAN to three other planning techniques:

- BURIDAN (Kushmerick, Hanks, & Weld 1995), a classical AI planning technique that extends partial-order planning to probabilistic domains,
- Plan enumeration with dynamic programming for plan evaluation (ENUM), and
- Dynamic programming (incremental pruning) to solve the corresponding finite-horizon partially observable MDP (POMDP) (Cassandra, Littman, & Zhang 1997).

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 BOMB-IN-TOILET, a problem described by Kushmerick, Hanks, & Weld (1995) is complicated by two facts. First, BURIDAN's performance, as reported by Kushmerick, Hanks, & Weld (1995) 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 (see Figure 7). 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, none of these figures seems appropriate for a comparison (although they do establish a range). Second, the BURIDAN results are likely to be somewhat slower due to machine differences.

MAXPLAN		BURIDAN	
Solution Method	CPU secs	Evaluation Method	CPU secs
UNIT/TIME	0.3	FORWARD	6.9
FULL DPLL	0.3	QUERY	64.9
PURE/UNIT/TIME	0.3	NETWORK	152.0
TIME-ORDERED	0.7	REVERSE	6736.0
AVERAGE	0.4	AVERAGE	1740.0

Figure 7: MAXPLAN outperforms BURIDAN on BOMB-IN-TOILET.

The performance of MAXPLAN (Figure 7) depends on which elements of DPLL we include, ranging from a low of 0.3 CPU seconds (for all methods except time-ordered splitting alone) to a high of 0.7 CPU seconds (time-ordered splitting alone). All times are 5-run averages. Unlike BURIDAN, however, we found that one variation (unit propagation with time-ordered splitting) consistently outperforms the others in all our trials. Comparing MAXPLAN's best time to BURIDAN's (problematic) best time, we see that MAXPLAN has an order of magnitude advantage.

We next compared the scaling behavior of MAXPLAN—rather than its performance on a single problem—to that of ENUM and POMDP. Since ENUM and POMDP require that all 2^P problem states be explicitly enumerated, these methods necessarily scale exponentially in the size of the state space even in the *best* case (in the worst case, all methods are expected to be exponential). ENUM necessarily scales exponentially in the plan horizon as well, since all A^N 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 8 shows performance results for MAXPLAN, ENUM, and POMDP as the horizon increases in SAND-CASTLE-67. ENUM, as expected, scales exponentially, but so does MAXPLAN. We might expect POMDP, which can solve arbitrary partially observable Markov decision processes, to perform poorly here. But POMDP is able to take advantage of the unobservability in the domain and, remarkably, scales linearly as the horizon increases.

We see much different behavior in a problem that allows us to enlarge the state space without increasing the length of the optimal plan. DISARMING-MULTIPLE-BOMBS is a planning domain that presents us with some number of packages, any or all of which contain bombs. The disarm action allows us to disarm all the bombs at once, but it requires that we know where the bombs are. The scan action gives us this knowledge. Thus,

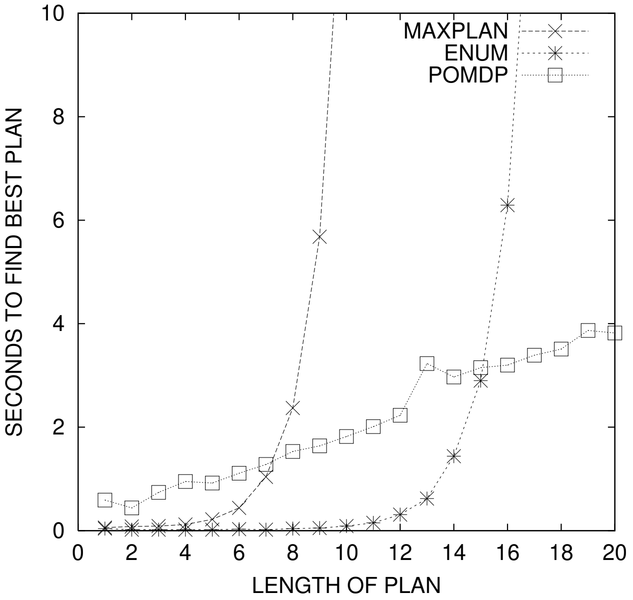

Figure 8: MAXPLAN solves SAND-CASTLE-67 more slowly than two dynamic programming-based approaches as plan length is increased.

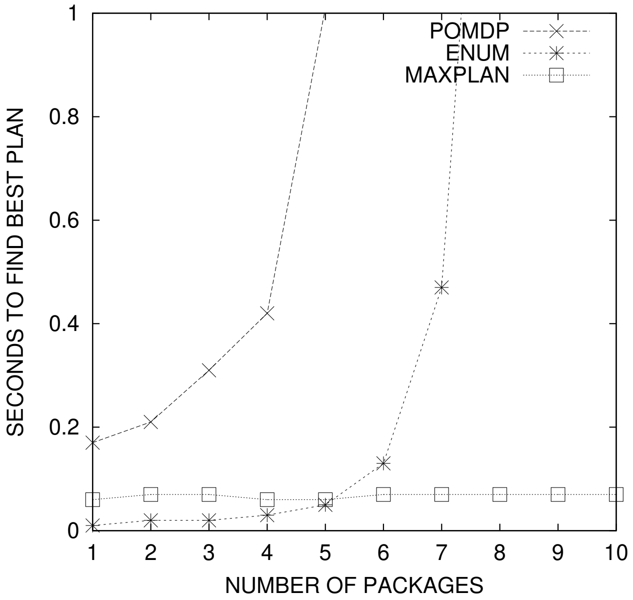

Figure 9: MAXPLAN solves DISARMING-MULTIPLE-BOMBS faster than two dynamic programming-based approaches as number of packages is increased.

even as the state space scales exponentially with the number of packages, there is always a successful 2-step plan. Figure 9 shows performance results for MAXPLAN, ENUM, and POMDP as the number of packages is increased. Both ENUM and POMDP scale exponentially, while MAXPLAN remains constant over the same range, solving each problem in less than 0.1 seconds.

FUTURE WORK

In addition to the problems described above, we have tested MAXPLAN on the slippery gripper problem described by Kushmerick, Hanks, & Weld (1995), and several variants of BOMB-IN-TOILET (with clogging and asymmetric actions). But the scaling behavior of MAXPLAN with respect to the size of the state space and the number of actions in the planing domain is largely unexplored. DISARMING-MULTIPLE-BOMBS shows that MAXPLAN can find *simple* solutions efficiently even as the state space grows, but we need to test our planner on more complicated domains.

Although our improvements have increased the efficiency of MAXPLAN by orders of magnitude over our initial implementation, more work needs to be done in many areas before MAXPLAN can be successfully applied to large-scale probabilistic planning problems. Efficiency could probably be improved by using a better CNF encoding of the planning problem (Ernst, Millstein, & Weld 1997; Kautz, McAllester, & Selman 1996) and by using more sophisticated data structures for storing the CNF formulas (Zhang & Stickel 1994).

Another promising area is splitting heuristics. Our time-ordered splitting heuristic does not specify an ordering for variables associated with the same time step. A heuristic that addresses this issue could provide a significant performance gain in real-world problems with a large number of variables at each time step. An entirely different splitting strategy could be developed based on the structure of the E-MAJSAT problems that result from planning problems. The clauses associated with each time step form a natural cluster whose variables appear only in those clauses and in the clauses associated with the immediately preceding and following time steps. A divide-and-conquer splitting strategy that forces a decomposition of the problem into these natural subformulas could yield performance gains.

Memoization could be more efficient. Memoizing every subformula solved is both impractical—we exhaust memory searching for plans with more than 15 steps in the SAND-CASTLE-67 domain—and probably unnecessary—memoizing every subformula is redundant. One approach is to treat a fixed amount of memory as a cache for subformulas and their values and create a good replacement policy for this cache. Such a policy might be based on many attributes of the cached subformulas, including size, difficulty, and frequency of use. Some initial results from caching experiments are reported by Majercik & Littman (1998).

Reinforcement learning could potentially be used to learn better splitting heuristics and cache replacement policies. With the exception of very large planning problems, however, a sufficiently long training period will not 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.

Approximation techniques need to be explored. Perhaps we can solve an approximate version of the problem quickly and then explore plan improvements in the remaining available time, sacrificing optimality for "anytime" planning and performance bounds. This does not improve worst-case complexity, but is likely to help for typical problems.

The variables generated by the current problem conversion process can be thought of as the nodes in a belief network. The planning problem, recast as a belief net problem, becomes one of finding the setting the choice nodes that maximizes the probability of the goal-condition nodes having the desired setting, and techniques for solving this belief network problem are likely to be applicable to our E-MAJSAT problem.

Finally, the current planner assumes complete unobservability and produces an optimal straight-line plan; a practical planner must be able to represent and reason about more complex plans, such as *conditional plans*, which can take advantage of circumstances as they evolve, and *looping plans*, which can express repeated actions compactly.

CONCLUSIONS

We have described MAXPLAN, a new approach to probabilistic planning that converts the planning problem to an equivalent E-MAJSAT problem—a type of Boolean satisfiability problem—and then solves that problem. We have shown that the conversion can be performed efficiently and we have described a solution method for the resulting E-MAJSAT problem. MAXPLAN performs significantly better on some standard stochastic test problems than the state-of-the-art probabilistic planner BURIDAN. Furthermore, although MAXPLAN's scaling behavior is problem dependent, MAXPLAN scales significantly better than two algorithms based on dynamic programming on one test problem.

Acknowledgments

This work was supported in part by grant NSF-IRI-97-02576-CAREER.

References

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1–2):279–298.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 1995. Exploiting structure in policy construction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1104–1113.

Cassandra, A.; Littman, M. L.; and Zhang, N. L. 1997. 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)*, 54–61. San Francisco, CA: Morgan Kaufmann Publishers.

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

Ernst, M. D.; Millstein, T. D.; and Weld, D. S. 1997. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1169–1176.

Goldsmith, J.; Littman, M. L.; and Mundhenk, M. 1997. The complexity of plan existence and evaluation in probabilistic domains. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97)*, 182–189. San Francisco, CA: Morgan Kaufmann Publishers.

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

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

Kautz, H.; McAllester, D.; and Selman, B. 1996. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*.

Kushmerick, N.; Hanks, S.; and Weld, D. S. 1995. An algorithm for probabilistic planning. *Artificial Intelligence* 76(1-2):239–286.

Littman, M. L.; Goldsmith, J.; and Mundhenk, M. 1998. The computational complexity of probabilistic plan existence and evaluation. *Journal of Artificial Intelligence Research*. To appear.

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

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

Zhang, H., and Stickel, M. E. 1994. Implementing the Davis-Putnam algorithm by tries. Technical report, Computer Science Department, University of Iowa, Iowa City, IA.