Conformant Probabilistic Planning via CSPs

Nathanael Hyafil
Department of Computer Science
University of Toronto
Toronto, Ontario
Canada M5S 1A4
nhyafil@cs.utoronto.ca

Fahiem Bacchus
Department of Computer Science
University of Toronto
Toronto, Ontario
Canada M5S 1A4
fbacchus@cs.utoronto.ca

Abstract

We present a new algorithm for the conformant probabilistic planning problem. This is a planning problem in which we have probabilistic actions and we want to optimize the probability of achieving the goal, but we have no observations available to us during the course of the plan's execution. Our algorithm is based on a CSP encoding of the problem, and a new more efficient caching scheme. The result is a gain in performance of several orders of magnitude over previous AI planners that have addressed the same problem.

We also compare our algorithm to algorithms for decision theoretic planning. There our algorithm is faster on small problems but does not scale as well. We identify the reasons for this, and show that the two types of algorithms are able to take advantage of distinct types of problem structure. Finding an algorithm that can lever both types of structure simultaneously is posed as an interesting open problem.

Introduction

In this paper we address what has been called in the AI planning community the probabilistic planning problem (Kushmerick, Hanks, & Weld 1995), but in modern terminology would more accurately be called the conformant probabilistic planning problem (*CPP*). Other work on probabilistic planning, e.g., (Blum & Langford 1999; Onder & Pollack 1999; Blythe 1998; Goldman & Boddy 1994) makes different assumptions about the observability of the environment.

Given a set of states S, we start with a *belief state*, which is a probability distribution over S, a set of actions with probabilistic effects, and a goal, which is a subset of S. The problem is to compute a single sequence of actions π that *maximizes* the probability that the goal will be true after executing π.

The basic assumption of the problem is that the system state cannot be observed while the plan is being executed, and that even when we start we do not know for certain what state we are in. This is what makes it a *conformant* planning problem. In fact, *CPP* is a generalization of the standard conformant planning problem studied in, e.g., (Cimatti & Roveri 2000). In particular, if our initial belief state assigns probability zero to every impossible initial state and we require the plan to achieve the goal with probability 1, then the problem reduces to that of standard conformant planning.

Conformant planning is useful for controlling systems with non-deterministic actions when sensing is too expensive, or when a fault has occurred and sensing is no longer reliable. In these cases, conditional plans (which have a branching structure) are not useful: executing a conditional plan requires sensing to know which branch to follow.

CPP can be applied in situations where standard conformant planning would fail (e.g., when a plan that guarantees that the goal is *always* achieved does not exist). In such cases we might still have some information about the likely distribution of states the system might be in, and then we can at least try to maximize the probability of achieving the goal state.

In this paper we present a new planning algorithm, implemented in a system called *CPplan*, for solving *CPP*. Our algorithm utilizes the standard technique of fixing the length of the plan and thus converting it to a problem over a finite set of states (this technique was first used in (Kautz & Selman 1996)). We then encode the resultant finite problem as an instance of a constraint satisfaction problem (CSP). Standard CSP backtracking algorithms are then utilized to compute a solution. Unlike the standard CSP problem, we are not just interested in whether or not a solution exists. Nevertheless, the answer we need can easily be computed from the search tree generated by these standard algorithms.

Our approach is very closely related to the MAXPLAN (Majercik & Littman 1998) planner which also solves *CPP*. However MAXPLAN encoded the problem as a CNF formula and used a variant of the DPLL (Davis, Logemann, & Loveland 1962) algorithm to compute the solution. As we will demonstrate, encoding in the higher-level formalism of CSPs has a number of advantages which ultimately yield orders of magnitude improvement in performance.

CPP can also be encoded as an instance of a partially observable Markov decision process (*POMDP*). Most *POMDP* algorithms are specified to solve *POMDP*s with discounted rewards. Our formulation, where we only care about achieving the goal after a finite length plan has been executed, requires a more involved encoding to convert it to a discounted reward problem. However, there is a class of *POMDP* algorithms that can solve the problem directly. We were able to run one of these algorithms on the same set of problems and thus compare its performance to our approach.

Our data shows that the *POMDP* algorithm performs extremely well on many problems. This leads to the other contribution of our paper, which is to clarify why our algorithm performs better on certain types of domains while the *POMDP* algorithm performs better on other types. Our conclusion is that finding a way to combine the positive features of these two algorithms would enhance our ability to solve both *CPP* and *POMDP*s.

Background

In this section we give a more precise description of *CPP* and the problem representation we use.

The input to *CPP* is a tuple $\langle S, \mathcal{B}, A, G, n \rangle$. S is a set of states. These are all of the states the system could possibly be in at any instance. $\mathcal{B}$ is a belief state, i.e., a probability distribution over S. We denote the probability of any particular state $s \in S$ under $\mathcal{B}$ by $\mathcal{B}[s]$. Similarly if $S' \subseteq S$ then $\mathcal{B}[S'] = \sum_{s' \in S'} \mathcal{B}[s']$. A is a set of actions. Each action is a function that maps states $s \in S$ to a distribution over S. If $a \in A$ we use $Pr(s, a, s')$ to denote the probability that action a when applied to state s yields state s'. G is a goal which is a subset of S. Finally n is an integer specifying the length of the plan to be computed.

One way of viewing probabilistic actions is to regard them as mapping belief states to new belief states (a point of view utilized in (Bonet & Geffner 2000)). For any action a, $a(\mathcal{B})$ is a new belief state such that for any state s',

$$a(\mathcal{B})[s'] = \sum_{s \in S} \mathcal{B}[s] Pr(s, a, s').$$

That is, the probability we arrive in s' from s is the probability we started off in s ($\mathcal{B}[s]$) times the probability a yields s' when applied in s ($Pr(s, a, s')$). Summing over all states s gives us the probability of being in s' after executing a.

A sequence of actions is also a mapping between belief states as follows. The empty sequence ϵ is the identity mapping $\epsilon(\mathcal{B}) = \mathcal{B}$, and action sequence $\langle a, \pi \rangle$, where a is a single action and π is an action sequence, is the mapping $\langle a, \pi \rangle(\mathcal{B}) = a(\pi(\mathcal{B}))$.

Under this view, *CPP* is the problem of finding the length n plan π such that $\pi(\mathcal{B})[G]$ is maximized: i.e., it maximizes the probability of the goal G when applied to the initial belief state. We call this probability of success (where the initial belief state $\mathcal{B}$ and goal G are fixed by the problem) the *value* of π. More generally, for any belief state $\mathcal{B}'$, or individual state s, and plan (of any length) π, the *value* of π in $\mathcal{B}'$ (in s) is the probability of reaching the goal when π is executed in $\mathcal{B}'$ (or s): i.e., $\pi(\mathcal{B}')[G]$ (or $\pi(s)[G]$).

Finally we introduce some other useful pieces of notation. Given a belief state $\mathcal{B}$, we say that a state s *is in* $\mathcal{B}$ if $\mathcal{B}[s] > 0$. That is, the states in a belief state are those that are assigned non-zero probability. Second, for any action sequence π and belief state $\mathcal{B}$, we say that state s' *is reachable by* π *from* $\mathcal{B}$ if $\pi(\mathcal{B})[s'] > 0$. That is, π arrives at s' with non-zero probability when executed in belief state $\mathcal{B}$. We also say that s' is reachable by π from a particular state s if $\pi(s)[s'] > 0$. Finally, for a particular action a and state s we say that s' *is a successor state of* s *under* a if $Pr(s, a, s') > 0$.

Representing S, A and G

We utilize higher level representations of S, A and G as follows. Let $\{v_1, \ldots, v_m\}$ be a set of m state *variables*. Each v_i can take on any of $k(i)$ different values $\{d_1, \ldots, d_{k(i)}\}$. A state $s \in S$ is represented as a particular setting of these m state variables. Hence, there are $\prod_{i=0}^{m} k(i)$ different states in S. The set of goal states G is represented as a boolean expression over the state variables that is satisfied by all the, and only the, states in G.

Finally, the actions are represented using the sequential-effects decision tree formalism of (Littman 1997) (also used in the MAXPLAN planner). Each action is represented by a sequence of decision trees each of which is labeled by the state variable it changes. Given a state s, the decision trees of action a are applied sequentially to modify the state variables of s and so generate a set of successor states. Each change applied has a certain probability, given by the decision tree, and the probability of each successor state s' is the product of the probabilities of the changes made to s to convert it to s'.

Consider the SANDCASTLE-67 example given in (Majercik & Littman 1998). In this domain states are specified by two boolean variables *moat* and *castle*. There are two actions *dig-moat* and *erect-castle*, whose decision trees are given in Figure 1. The domain concerns building a sand castle. We can build a moat with the action *dig-moat*, but this action might not succeed. Alternately we can erect a castle with *erect-castle*. This is more likely to succeed if a moat has already been built, but it can also destroy the moat.

Say, for example, we apply the *erect-castle* action to the state $s = \{\textit{moat}, \neg\textit{castle}\}$ (for boolean variables we write x for $x =$ **true** and $\neg x$ for $x =$ **false**). First we apply the decision tree for *castle*, following the branch compatible with s. This brings us to the leaf node labeled r_2. This leaf indicates that *castle* is made true by the action (in this context) with probability 0.67. So we get the two intermediate states with probabilities $\big\{\{\textit{moat}, \neg\textit{castle}\}{:}0.33, \{\textit{moat}, \textit{castle}\}{:}0.67)\big\}$. Note that in these states the setting of *castle* is "new", and might be different from its original setting. Second, we apply the decision tree for *moat* to these two intermediate states, For $\{\textit{moat}, \neg\textit{castle}\}{:}0.33$ we arrive at the leaf labeled r_5 (the nodes *moat* and *castle* in the tree depend on the setting of these variables in the original state, while the node "*castle* :new" depends on the setting of *castle* in the intermediate state[1]). Hence *moat* is made true with probability 0.5. This yields two final states with probabilities $\big\{\{\neg\textit{moat}, \neg\textit{castle}\}{:}0.33 \times 0.5, \{\textit{moat}, \neg\textit{castle}\}{:}0.33 \times 0.5\big\}$. From the other intermediate state $\{\textit{moat}, \textit{castle}\}{:}0.67$ we arrive at leaf r_4. Hence *moat* is made true with probability 0.75. This yields two more final states with probabilities $\big\{\{\neg\textit{moat}, \textit{castle}\}{:}0.67 \times 0.25, \{\textit{moat}, \textit{castle}\}{:}0.67 \times 0.75\big\}$.

There are a few things worth noting. First, all other states are assigned zero probability (in this example all states are successor states). Second, the randomness of the action occurs only at the leaves. The branch followed is completely determined by the original state and by the changes gener-

[1] The decision trees are applied sequentially so that the settings created by previous trees can be branched on in subsequent trees.

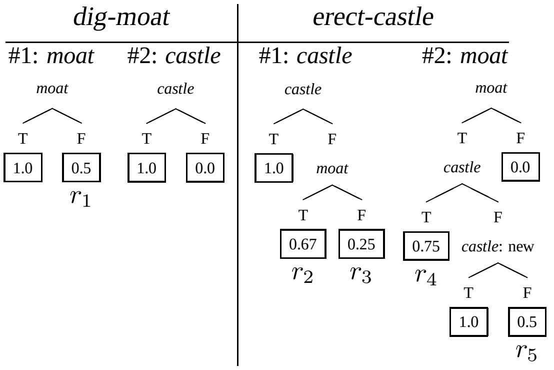

Figure 1: Sequential Decision trees for SANDCASTLE-67

ated by the previous trees. And third, the action does not provide any information about the state it actually generated. So even if we start in a known state, all we will have after the action is executed is a distribution over the states we could be in.

The CSP Encoding

The technique of translating planning problems to constraint satisfaction problems has been used before in classical planning, e.g., (van Beek & Chen 1999; Do & Kambhampati 2001). Here we apply this technique to *CPP*. In this section we describe our encoding and the manner in which standard CSP algorithms can be used to solve the problem.

A CSP consists of a set of variables and a set of constraints. Each variable has a finite domain of values and can be assigned any value from its domain. Each constraint is over some subset of the variables. It acts as a function from an assignment of values to those variables to **true**/**false**. We say that an assignment of values to the variables of a constraint *satisfies* the constraint if the constraint evaluates to **true** on that assignment. A solution to a CSP is an assignment of a value to each variable such that all constraints are satisfied.

The CSP variables We represent a *CPP* with a CSP containing three types of variables. For each step of the length n plan, we have m state variables whose values specify the state reached at that step of the plan; one action variable whose value specifies the action taken at that step of the plan; and R random variables whose values specify the particular random outcome of the action taken at that step.

As noted above, the random effects of an action occur at the leaves of the decision trees. We call leaves with probabilities that are not 0 or 1 random leaves. (Leaves with probabilities 0 or 1 do not generate random effects). We associate a random variable with each random leaf. For each random outcome at the leaf we have a value for its associated random variable, and the probability that the random variable takes on that value is the same as the probability that the leaf yields that random outcome. Assigning a value to the random variable corresponds to asserting that its leaf will generate the associated outcome. Thus once all of the random variables from the decision trees of an action have been set, the action becomes deterministic: at each leaf a fixed outcome will be generated. We can examine all possible random outcomes of an action by examining all possible settings of its random variables. This in fact is what occurs while solving the CSP—we iterate over all settings of these variables. This technique is exactly that used in (Majercik & Littman 1998).

Example 1 Consider the SANDCASTLE-67 domain specified above. For a n step plan we will have the following variables

State Variables: *castle*^{i}, *moat*^{i}, $i = 0, \dots, n$.

Action Variables: A^i, $i = 0, \dots, n-1$. The domain of each action variable is the set $\{$*dig-moat*, *erect-castle*$\}$.

Random Variables: $r_1^i, \dots, r_5^i$, $i = 0, \dots, n-1$.

In most problems the state variables are boolean, but because we are encoding to a CSP, our formalism can deal with arbitrary domain sizes. This makes the representation of problems such as GRID-10X10 (described below) much simpler. Similarly, we need only n action variables with domains equal to all possible actions. In a SAT encoding one needs kn action variables where k is the number of possible actions, and $n \times k^2$ clauses to encode the constraint that only one action can be executed at each step. These exclusivity constraints are automatically satisfied in the CSP encoding by the fact that in a CSP a variable (in particular the action variables) can only be assigned a single value. As we will discuss later we could also use non-binary domains to optimize the processing of the random variables.

The CSP Constraints The CSP encoding of a *CPP* will contain constraints over the variables specified above. One constraint is used to encode the goal. It is a constraint over the state variables mentioned in the goal that is satisfied only by the settings to those variables that satisfy the goal. Since the goal is a condition on the final state, its constraint would only mention state variables from the n-th step.

The other constraints are used to model the action transitions. There is one constraint for every branch of every decision tree in the problem specification. In particular, for every step i in the plan and every leaf in the decision trees there will be a single constraint. This constraint will be over some subset of the state variables at step i, some subset of the state variables at step $i+1$, some subset of the random variables at step i, and the action variable at step i. The constraint encodes the setting of the step $i+1$ state variables that is compatible with the execution of a particular action with a fixed random outcome in the step i state.

Example 2 In the SANDCASTLE-67 domain at step i the leaf labeled r_5 in the second decision tree of the *erect-castle* action (see Figure 1) will generate a constraint over A^i, *moat*^{i}, *castle*^{i}, *castle*$^{i+1}$, *moat*$^{i+1}$ and r_5^i that encodes the following boolean condition

$$A^i = \textit{erect-castle} \wedge \textit{moat}^i \wedge \neg\textit{castle}^i \wedge \neg\textit{castle}^{i+1} \wedge r_5^i \\ \Rightarrow \textit{moat}^{i+1}$$

Initial Belief State Finally we must capture the initial belief state $\mathcal{B}$. This is most easily done by having, like partial order planning, a special initial action. Like the other actions, this action is specified by a collection of decision trees with an associated set of random leaves. Since the initial action is applied to the "null" state, its decision trees can only refer to state variables set by the action's previous decision trees. Sequentially applying these decision trees yields a set of states each with some associated probability: i.e., a belief state. Given any initial belief state a sequential set of decision trees that yield precisely this belief state can be easily constructed.

Once the initial action is specified we can encode its random leaves with corresponding step zero random variables, and the branches of its trees with constraints. Then every setting of the step zero random variables will generate, via the constraints, a setting of the step zero state variables (i.e., one of the initial states s such that $\mathcal{B}[s] > 0$), and the probability of that particular setting of the step zero random variables will be equal to $\mathcal{B}[s]$.

From CSP solutions to a solution to *CPP*

A solution to the CSP is an assignment of the state, action and random variables representing a valid sequence of transitions from an initial state to a goal state. The settings of the action variables represent the plan that is being executed, the setting of the state variables specify the execution path the plan induced, and the probability of the random variables assignments when multiplied together is equal to the probability of that solution. That is, it is the probability that this particular execution path was traversed by this particular plan.

To evaluate the value (probability of success) of a plan π, one must sum the probabilities of all the solutions to the CSP where the settings of the action variables are equal to those in π. Once this is done for every plan (i.e., every possible sequence of assignments to the action variables) the optimal plan will be the one with the highest value.

Reusing Intermediate Computations

A naive implementation in which all solutions are enumerated and the value of each plan evaluated, as described above, runs very slowly. To make our approach viable it is necessary to do a further analysis to identify redundancies in the computation that can be eliminated using dynamic programming techniques, i.e., caching (recording) and reusing intermediate results.

This analysis identifies two types of intermediate computation that can be cached and reused. The first arises from the Markov property of the problem. In particular, if we arrive at the belief state $\mathcal{B}_i$ with $n - i$ steps remaining, the optimal sequence of $n - i$ actions to execute is independent of how we arrived at $\mathcal{B}_i$. Thus, for each step i and each belief state we arrive at step i, we could cache the optimal subplan for that belief state once it has been computed. If we once again arrive at that belief state with $n - i$ steps to go, we can reuse the cached value rather than recomputing the $n-i$ step optimal plan for that belief state. This is the dynamic programming scheme used in value iteration POMDP algorithms (discussed below). It is also the dynamic programming scheme used in MAXPLAN.

Both MAXPLAN and our own system *CPplan* work by searching in a tree of variable instantiations. At each node n in the search tree a variable v is chosen that has not been assigned by any ancestor node. The children of n are generated by assigning v all possible values in its domain. The leaves of the tree are those where some constraint has been violated (or for MAXPLAN a clause falsified) or where all variables have been assigned. The latter leaves are solutions.[2] The tree is searched in a depth-first manner (and in fact it is constructed and deconstructed as it is searched, so that the only part of the tree that is actually materialized at any point in the search is the current path).

MAXPLAN uses a variable ordering that instantiates the action variables first, instantiating those in chronological order. Thus at any node it has already committed to a i-length plan prefix, and must compute the optimal $n - i$ length plan given the belief state produced by the prefix. MAXPLAN caches the subformula generated by fixing the i-length prefix. This subformula corresponds to an encoding of the belief state generated by the current plan prefix, along with all of the future possibilities for the next $n - i$ steps. Hence, when later on in the depth-first search it encounters the same subformula, it has in fact discovered two distinct i-length plan prefixes that map the initial state to the same belief state. In this case both have the same optimal completion, and that completion need only be computed once.[3]

Our planner *CPplan* uses a second more refined caching scheme. It instantiates variables in the sequence, $A^0, R^0, S^0, A^1, R^1, S^1, A^i, R^i, S^i, \ldots$, where A^i is the i-step action variable, R^i are the i-step random variables and S^i are the i-step state variables. That is, like MAXPLAN it builds up the plan chronologically, but after each action it branches on all of the settings of the random variables associated with the chosen action. The setting of the previous state variables, the action variable, and the random variables, is sufficient to determine the next state variables (i.e., these variables do not generate any branches—they each will have only one legal value).

At a node of its search tree where all of the i-step state variables have first been set, i.e., the node where state s has first been generated by i-steps of some plan prefix, *CPplan* computes for every length $n - i$ plan π_{n-i} the value (success probability) of π_{n-i} in state s. It then caches these values in a table indexed by the state s, and the step i. If later on in the depth-first search *CPplan* again encounters state s at step i it backtracks immediately without having to recompute these values. The variable ordering mentioned above can be altered as long as the i-step state variables S^i are instantiated *after* the actions, state and random variables for all steps $j < i$. For example, the ordering $A^0, A^1, A^i, \ldots, R^0, S^0, R^1, S^1, R^i, S^i, \ldots$ is also possible.

Note that whenever the first caching scheme is able to

[2] Both MAXPLAN and *CPplan* do additional constraint propagation to eliminate values from the domains of uninstantiated variables that would be bound to violate a constraint.

[3] Subformula caching also occurs when all n steps of the plan have been determined. In this case the subformula caching has to do with optimizing the computation of the value of the plan.

```
CPplan()
Action variable first; then random variables; then state variables
  Select next unassigned variable $V$
  If $V$ is the last state variable of a step:
    If this state/step is already cached return
  Else-if all variables are assigned
    Cache 1 as the value of the previous state/step
  Else
    For each value $d$ of $V$
      $V = d$
      if GACPropagation()
        CPplan()
      If $V$ is the action variable $A^i$
        Update the cached results for the previous state/step
            adding the value of all plans starting with $d$
```

Table 1: *CPplan* algorithm

avoid computing a subtree, the second is also. Furthermore, the second scheme can avoid computing some subtrees that the first must compute. First, if using the first scheme we had previously encountered belief state $\mathcal{B}_i$, then using the second scheme we would have previously encountered all of the states in $\mathcal{B}_i$ (i.e., those with non-zero probability). Second, we could have two different belief states $\mathcal{B}_i$ and $\mathcal{B}'_i$ which contain the same set of states (although the probabilities assigned to these states might be different). The first caching scheme would have to compute the subtree below $\mathcal{B}'_i$, but the second would have already encountered all of the states in $\mathcal{B}'_i$ and thus could avoid this subtree.

In terms of space, in the worst case, the first scheme can encounter as many different belief states as there are action sequences of length less than or equal to n. All of these would have to be stored in the cache. The second scheme, on the other hand, might reach every state in S at each level. Thus it might have to store for each state, information about every action sequence of length less than or equal to n. Hence, the second scheme can, in the worst case, require a factor equal to the size of the state space ($|S|$) more space.

However, in practice, what is relevant for the first scheme is the number of distinct belief states reached by a i-step plan, and for the second scheme the number of distinct states that are reached by a i-step plan. Although our current set of experiments do not demonstrate this conclusively, it seems intuitively reasonable that many more different belief states would be reached than distinct states.

Our caching scheme also has some other important practical advantages, which we will discuss later.

The *CPplan* algorithm

We use *value*(π, s) to denote the value of π in state s (i.e., the probability π reaches the goal when executed in s). The *CPplan* algorithm computes for every state s in the initial belief state $\mathcal{B}$ (i.e., $\mathcal{B}[s] > 0$), and every length n plan π, *value*(π, s). From these values, the value of any length n plan is simply computed by the expression

$$\sum_{s:\mathcal{B}[s]>0} \mathcal{B}[s] \times \mathit{value}(\pi, s).$$

That is, the probability that π reaches the goal state from $\mathcal{B}$ is the probability we start off in s ($\mathcal{B}[s]$) times the probability π reaches the goal when executed in s (*value*(π, s)). Thus these values provide us with sufficient information to find the length n plan with maximum value (success probability).

Note that we must have the value of all plans in each of the initial states. It is not sufficient to keep, e.g., only the plan with maximum value for each state. The plan with maximum value overall depends on the probabilities of these states. For example, the best plan for state s_1 may be very poor for another state s_2. If $\mathcal{B}[s_1]$ is much greater than $\mathcal{B}[s_2]$, then its best plan might be best overall, but if $\mathcal{B}[s_1]$ is much lower than $\mathcal{B}[s_2]$ it is unlikely to be best overall. Even more problematic is that the best plan overall might not be best for any single state.[4]

A sketch of the *CPplan* algorithm is given in Table 1. As mentioned it works by doing a depth-first search in a tree of variable instantiations. Given the order it instantiates the variables (action variable followed by the random variables followed by the state variables), its computation can be recursively decomposed as follows. To compute the value of any length i plan $\pi = \langle a, \pi^{i-1} \rangle$ in state s, where a is π's first action, we use the fact that

$$\mathit{value}(\pi, s) = \sum_{s':Pr(s,a,s')>0} Pr(s, a, s') \times \mathit{value}(\pi^{i-1}, s').$$

That is, π can reach the goal from s by making a transition to s', with probability $Pr(s, a, s')$, and then from there reach the goal, with probability *value*(π^{i-1}, s'). Summing the product of these probabilities over all of s's successor states under π gives the probability of π reaching the goal from s.

Hence, if we recursively compute the value of every length $i - 1$ plan in all states reachable from s by a single action, we can compute the value of every length i plan in s, that starts with this action, with a simple computation. After exploring a value a for the action variable below s, we can update s's cached value to include s's value on the plans starting with a. Subsequent actions a' might be able to reuse some of these computations (or previous computations). After all actions have been tried, we can backtrack from s with a complete cache for s. The recursion bottoms out at states generated at step n that satisfy the goal (constraint propagation of the goal constraint means that no step n state falsifying the goal will be visited). For these states we only need to compute the value of the empty action sequence. This has value 1 since the state must satisfy the goal. Finally, after

[4] This makes *CPP* inherently complex, and means that to obtain truly practical algorithms we will probably have to examine approximate versions of *CPP*. For example, (Kushmerick, Hanks, & Weld 1995) examine the approximate version where any plan with value greater than a fixed threshold was acceptable. This can be done in our framework as well. Another way of avoiding computing the value of all plans is the α-vector abstraction used in *POMDP* algorithms. We will discuss this form of abstraction later.

backtracking from the initial call we can compute the value of all length n plans from the caches for the initial states.

An important factor in the algorithm is its use of local constraint propagation, specifically the generalized arc consistency algorithm (GAC) (Mackworth 1977), before exploring the subtree below a value assignment. GAC allows the algorithm to avoid exploring some of states and actions that have zero probability of achieving the goal (given the variables assignments already committed to at this node of the tree). In particular, it prunes from the domains of the future variables, values that would make achieving the goal impossible. We do not affect the computed probabilities of success by rejecting such assigments since they would only contribute zero to the probability of success. GAC is a local consistency check, hence it cannot detect all inconsistent assignments. Nevertheless, it can eliminate some assignments thus reducing the size of the subtree that must be searched below the current node. Furthermore, it can sometimes detect that an unassigned variable has no consistent values remaining, in which it can avoid searching the subtree altogether (GACPropagation() will return false in this case). This is one of the important differences between our approach and approaches based simply on reachability, e.g., (Bonet & Geffner 2000).

Partial Caching

Our caching mechanism provides an important gain in time but can be very costly in terms of space: it is necessary to store results for all possible plan suffixes rather than just the best one. For an n-horizon problem with A actions, the space required is proportional to A^n. However, there is a very good space-time tradeoff that can be utilized. The largest caches exist at the top of the tree, where each cache contains the value of all length n plans at the state. If we avoid caching states at the first i steps, we will only require caches containing information about all length $n - i$ plans, at the expense of always having to recompute every state reached in the first i steps (once we reach step $i + 1$ in these computations, however, we can again utilize the cached results). Thus we increase computation time by a factor $O(2^i)$ to save space of order $O(2^{n-i})$. When i is small this is a good tradeoff. However, we have not yet implemented this partial caching scheme, so our empirical results are all reported with full caching.

Another technique that can be utilized when partial caching is used (the technique is redundant when full caching is used), is to optimize the random variables. Notice, that the random variables are set after the action has been chosen. This means that the only random variables that can influence the next state are those in the branches of the decision trees of that action, all other random variables are irrelevant, Furthermore, even among those decision trees only the branches compatible with the previous state (which has also been previously set) are relevant.

We can stop the algorithm from branching on irrelevant random variables (which would generate multiple copies of the same state), by adding an extra value to the domain of all random variables. Furthermore, we can add constraints to force the random variables to take on this extra value whenever they become irrelevant. This makes irrelevant random variables single valued, and they no longer generate extra branches in the search tree.

These extra constraints involve minimal memory cost and can therefore be very effective when only partial caching is being performed.

Cache Keys

A final benefit of our scheme is that the cache key is very short—the state and the time step. Thus we can quite efficiently access the cache. This is in sharp contrast with the caching scheme used in MAXPLAN, where the cache keys are large CNF formulas. MAXPLAN's cache only stores a single value for each key, while we store an entire table. Nevertheless, empirically the net space required for the cache is significantly lower for *CPplan*.

Comparison with MAXPLAN

We have compared *CPplan* to MAXPLAN on SANDCASTLE-67 and on the SLIPPERY-GRIPPER problem (Kushmerick, Hanks, & Weld 1995). All experiments were conducted on a 2.4 GHz Xeon machine with 3GB of RAM. The description of SLIPPERY-GRIPPER is as follows. There are 4 boolean state variables, *grip-dry*, *grip-dirty*, *block-painted*, and *block-held*; and 4 actions, *dry*, *clean*, *paint* and *pick-up*. The initial belief state is the set of states with associated probabilities $\big\{\{\neg\textit{block-painted}, \neg\textit{grip-dirty}, \textit{clean}, \textit{grip-dry}\}{:}0.7, \{\neg\textit{block-painted}, \neg\textit{grip-dirty}, \textit{clean}, \neg\textit{grip-dry}\}{:}0.3\big\}$. The goal is $\neg$*grip-dirty* $\wedge$ *block-held* $\wedge$ *block-painted.* Action *dry* dries a wet gripper with probability 0.8; *clean* cleans a dirty gripper with probability 0.85; *paint* paints the block with probability 1 but makes the gripper dirty with probability 1 if the block was held and probability 0.1 if it was not; and *pick-up* picks up the block with probability 0.95 if the gripper is dry and with probability 0.5 if it is wet. The actions do not affect any other state variables. Figures 2 and 3 show the results on a logarithmic scale for plans of length 10 to 20 for SANDCASTLE-67 and 5 to 12 for SLIPPERY-GRIPPER. In both cases, *CPplan* was run with full caching. Results were also generated for the BombToilet problem from (Majercik & Littman 1998) (where MAXPLAN was faster than all other algorithms) but both *CPplan* and MAXPLAN are able to solve the problem in less than the precision of the timer.

These results show that *CPplan*'s caching mechanism is much faster (between 2 and 3 orders of magnitude) on these problems. The rate of growth seems to be similar, so the results do not tell us much about the effectiveness of *CPplan*'s more refined caching scheme. *CPplan* also uses much less memory. On SANDCASTLE-67, MAXPLAN cannot solve the 20 step problem due to lack of memory while *CPplan* goes up to 28 steps. On SLIPPERY-GRIPPER MAXPLAN stops at 12 steps while *CPplan* can go to 14. These differences are very significant since *CPplan* is operating with full caching, and hence its memory requirements are growing exponentially with plan length.

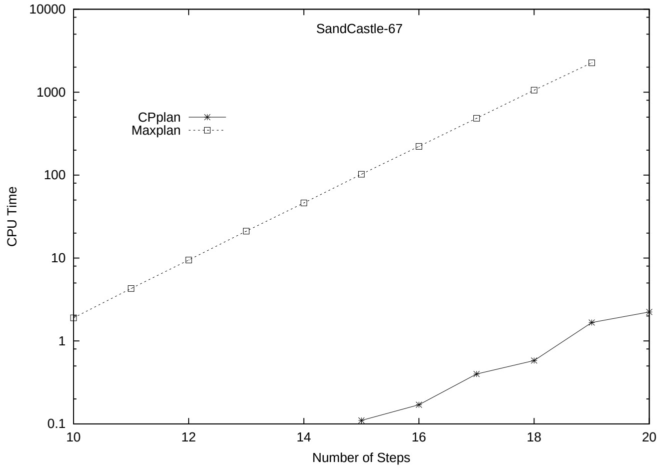

Figure 2: *CPplan* vs Maxplan on SandCastle-67

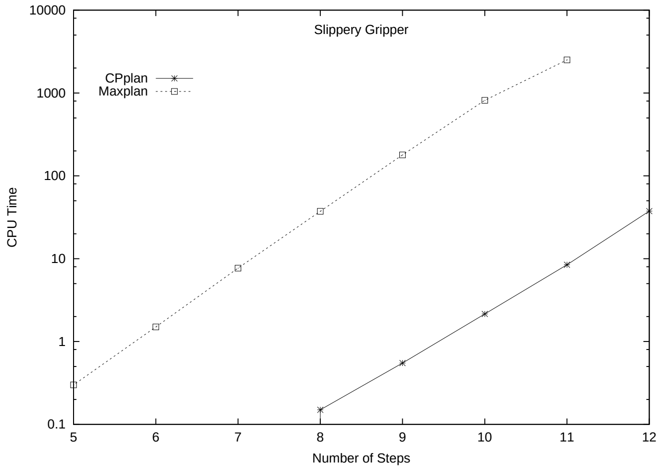

Figure 3: *CPplan* vs Maxplan on Slippery Gripper

*POMDP*s

CPP can also be seen as a special case of Partially Observable Markov Decision Processes (*POMDP*s). A general *POMDP* model has probabilistic transitions but also allows for partial observability (as compared to the complete unobservability case of *CPP*). In this setting, a solution is a mapping from history (past actions and observations) to actions. Decision theoretic planning techniques for solving *POMDP*s usually assume a fixed reward for every state and an infinitely executing plan. The plan specifies the action to take in each belief state, and each belief state visited yields a reward equal to the expected reward under that distribution. The plan's infinite horizon is handled by discounting future rewards exponentially.

To cast the n-step *CPP* in precisely this formalism requires a specialized encoding to handle the fact that in a *CPP* the rewards are only given after n steps of the plan have been executed. One way to encode a *CPP* as a standard infinite horizon *POMDP* would be to generate $n+1$ copies of each state, each indexed by the time step $i = 0, \dots, n$. The initial belief state would be a distribution over the 0-step states, with all other states assigned zero probability. The transitions would always map step-i states to step-$i+1$ states, giving zero probability to any other transition. Only n-step states would have a non-zero reward, with those n-step states satisfying the goal having reward 1. All actions would map the n-step states to an absorbing terminal state that has zero reward. Finally, there would be no observations. The solution to this infinite horizon *POMDP* would be precisely the solution to the corresponding *CPP*. Unfortunately, the factor n blow up in the state space makes the resulting problems impossible to solve with current *POMDP* algorithms.

There is however, a class of *POMDP* algorithm that although designed to solve the general infinite horizon problem, in fact does all the computations required to solve *CPP*. These algorithms are called value iteration (VI) algorithms, and we will now describe their basic operation. In our description we ignore observations, so that all plans are simple action sequences rather than conditional plans.

Brief VI overview

VI algorithms use the first of the dynamic programming scheme presented above. Specifically, they compute the optimal k step plan for *every* belief state starting at $k = 0$ and increasing k until they reach a k such that adding one more step to any of the plans makes less that ϵ difference to the value of the plan.[5]

The reason this approach works is that there exists compact representations for the function that maps any belief state to its optimal k step plan. There are of course an infinite number of belief states, but since there are only a finite number of different k step plans it must be the case that the same plan is optimal for an entire region of the belief space. More importantly, it turns out that many of the k step plans are nowhere optimal, and those that are optimal for some belief state are optimal for a linear region of the belief space surrounding that belief state.

For any k step plan, π, the value of π in any belief state is a linear function of its value in the individual states. That is,

$$\mathit{value}(\pi, \mathcal{B}) = \sum_{s \in S} \mathcal{B}[s] \times \mathit{value}(\pi, s).$$

Thus by storing π's value for every state, we can easily compute its value for every belief state. If there are ℓ states in S, then we need only store an ℓ dimensional vector of values for π. This vector is called an α-vector.

Abstractly VI algorithms start with $k = 1$ and with a set Φ_1^+ of α-vectors that contains an α-vector for every one-step plan. Then they prune from Φ_1^+ all α-vectors that are

[5] This is where the assumption of discounted future rewards comes into play. k step plans only differ from $k+1$ step plans by the belief states they visit at step $k+1$. Since the reward achieved by visiting these belief states has been discounted by a factor of size $1/O(2^{k+1})$ the value of the optimal k step plan will not be much different from the value of the optimal $k+1$ step plan when k is large.

nowhere optimal; i.e., those whose value on any belief state is always dominated by some other α-vector in Φ_1^+. This yields a reduced set of one-step α-vectors Φ_1, each of which represents a one-step plan that is optimal for some region of the belief state. At stage k we have a set Φ_k of α-vectors each corresponding to some k-step plan that is optimal for some region of the belief state, and we use this to compute Φ_{k+1}. We first compute Φ_{k+1}^+ which is a superset of Φ_{k+1}. The dynamic programming scheme is based on the fact that any optimal $k+1$ plan must be of the form $\langle a, \pi_k \rangle$ where π_k is an optimal k step plan. Thus π_k must be one of the plans already represented in Φ_k. So one simple way to compute Φ_{k+1}^+ is to include in it all one step extensions of the plans represented in Φ_k. That is, we compute the α-vectors associated with all one step extensions of the plans in Φ_k and place these α-vectors in Φ_{k+1}^+. Then we prune from Φ_{k+1}^+ all nowhere optimal α-vectors, to obtain Φ_{k+1} the set of α-vectors representing somewhere optimal $k+1$ plans.

Once we have the set Φ_n we can find the optimal n step plan for a particular belief state $\mathcal{B}$, by computing the value of all of the plans in Φ_n at $\mathcal{B}$ (using the α-vectors as shown above) and identifying the plan with maximal value. The value of this maximum value plan at $\mathcal{B}$ is also called $\mathcal{B}$'s value. Thus, the set Φ_n also represents a value function that maps every belief state $\mathcal{B}$ to the value of the best plan for $\mathcal{B}$.

Modern *POMDP* VI algorithms attempt to optimize the construction of Φ_{k+1} in various ways. For example, they might generate this set directly without first generating Φ_{k+1}^+ (e.g., the linear support algorithm of (Cheng 1988) or the witness algorithm of (Kaelbling, Littman, & Cassandra 1998)); or they might prune Φ_{k+1}^+ while it is being constructed (e.g., the incremental pruning algorithm of (Cassandra, Littman, & Zhang 1997)); or they might wait until the end to do the pruning (e.g. the DP-Backup algorithm of (Monahan 1982)). In all cases much of the computation is performed by setting up and solving various linear programming problems.

The key factor, however, in the complexity of VI *POMDP* algorithms is the number of somewhere optimal length k plans and how this number grows with k. These algorithms scale well if this number grows slowly. As we will see in the next section, this is the case for many of the problems we have experimented with.

But before we present some of these empirical results, there is one more source of efficiency that *POMDP* algorithms take advantage of, that should be mentioned. Implementations of *POMDP* algorithms utilize linear programming packages which work with finite precision. This means that if two α-vectors are within ϵ of the optimal for some region of the belief space, these algorithms will only keep one of them. Thus although the run times of these algorithms grow exponentially with the number of steps, we tend to see the exponent shrinking as the number of steps increases. In other words, the rate at which the sets of somewhere optimal α-vectors grows slows down as the number of steps increases, since more and more of these α-vectors become approximately equal.

α-vector abstraction

α-vector abstraction refers to the fact that each α-vector specifies a plan that is optimal for a (linear) region of the belief space. Thus it can be that a relatively small number of plans are in fact sufficient to cover the entire belief space.

This is illustrated in the SLIPPERY-GRIPPER problem, described above. The 0 stages-to-go value function is obvious: the value of each belief state is the sum of the probabilities of the two goal states in that belief state, $V^0(b) = b(s_G^1) + b(s_G^2)$. With 1 stage-to-go, out of the 4 possible actions 2 are sufficient to represent the optimal value function (*clean* and *pick-up*). For the *paint* action, if the block was held the gripper will become dirty and if it was not *paint* will not pick it up, so in both cases a goal state cannot be reached. The *dry* action will only reach a goal state from a goal state, but *clean* and *pick-up* also maintain goal states and furthermore they transition some non-goal states to goal states. Thus they dominate *dry* in all belief states. Now with 2 stages-to-go, working from the two 1-stage plan, *clean* and *pick-up*, and pruning dominated plans, there are only six α-vectors (plans) necessary to represent the optimal value function, instead of $4^2 = 16$. These plan are $\langle \textit{clean}, \textit{clean} \rangle$, $\langle \textit{clean}, \textit{pick-up} \rangle$, $\langle \textit{pick-up}, \textit{pick-up} \rangle$, $\langle \textit{dry}, \textit{pick-up} \rangle$, $\langle \textit{paint}, \textit{clean} \rangle$, and $\langle \textit{paint}, \textit{pick-up} \rangle$. Any other length two plan is dominated by one of these six in all belief states. For three stages-to go there are ten α-vectors representing the optimal value function (instead of $4^3 = 64$) and for ten stages-to-go there are only 40 α-vectors (plans) that are somewhere optimal instead of more than a million.

Thanks to the power of this abstraction it might only be necessary to evaluate a small portion of all possible plans at every step. However to evaluate one plan, one must consider the whole ($|S|$-dimensional, continuous) belief state space, even though potentially large regions of that space are not reachable from the initial belief state. When the number of states in the problem is large, solving the resulting linear programs can take time.

Dynamic Reachability

To sum up, *POMDP* algorithms are able to evaluate only the necessary plans but over an unnecessarily large space. On the other hand, combinatorial probabilistic conformant planners like *CPplan* (or MAXPLAN) must evaluate all $|A|^n$ possible plans, but the tree-search approach leads to a significant advantage in that it performs a dynamic reachability and local constraints analysis. In *CPplan*, an assignment to all the state variables at a particular step can be considered to be a state “node”. Once such a node has been reached the *CPplan* algorithm will branch over all possible actions (through the A^k variable) and all possible probabilistic effects of these actions (through the random variables R_i^k) that are locally consistent. It will thus end up instantiating only the state nodes that are reachable and from the previous one in one step and that are locally consistent. Therefore only these consistent reachable nodes will be expanded in the future search and the effects of actions on all other (non-reachable or inconsistent) states will not be considered.

To illustrate this, consider the following GRID-10X10 problem. A robot is navigating in a 10x10 grid starting in some fixed initial location and the goal is to reach the upper right corner of the grid (i.e., coordinate (9,9)). There are 4 (probabilistic) actions, *right*, *left*, *up*, and *down*. *up* and *down* move the robot in the designated direction with probability 0.9 and to the left or right with probability 0.05 each. *left* and *right* "work" with probability 0.8 and move vertically with probability 0.1 in each direction. If the robot tries to move into a wall it bounces back and remains in its original location.

If the robot is in a particular location, irrespective of what the next executed action is, there are only 4 possible states that it can end up in (corresponding to the 4 directions or 3 directions and the current state if there is a wall). We are only interested in the values of future actions or plans in those 4 states and not in all 100 states. The search tree in *CPplan* does exactly that: starting in an initial state, it branches on all the 1-step reachable states. With only 4 reachable states per step, a k-step plan search will "only" visit 4^k state nodes (in the case of full caching). Note that this is very different, and much more useful, than global reachability analysis. In particular, at each stage we consider only the states reachable from the state we are currently in, rather than always considering all reachable states.

Comparison with *POMDP*s

We were able to solve *CPP* problems using a VI based *POMDP* solver written by Cassandra (Cassandra 1999). Cassandra's code allows one to specify a finite horizon (this fixes the number of iterations the VI algorithm performs), give a reward of 0 to every stage except the final one, and to set the discount factor to 1. Cassandra's solver implements the incremental pruning algorithm described in (Cassandra, Littman, & Zhang 1997).

We ran *CPplan* and Cassandra's implementation on SANDCASTLE-67 (Figure 4), SLIPPERY-GRIPPER (Figure 5) and GRID-10X10 (Figure 6) in order to compare the relative advantages of α-vector abstraction and dynamic reachability.

In SLIPPERY-GRIPPER, *CPplan* must evaluate all $|A|^n = 4^n$ plans by visiting only $S_{reach} = 8^n$ nodes (where S_{reach} is the maximum number of reachable states in one step). *POMDP* is able to prune many of the possible plans but must evaluate the remaining ones on a 16-dimensional continuous space. On GRID-10X10, *CPplan* must evaluate all $|A|^n = 4^n$ plans by visiting only $S^n_{reach} = 4^n$ nodes, while *POMDP* evaluates much fewer plans but on a 100-dimensional continuous space. Note that *CPplan* runs out of memory on SANDCASTLE-67 at 28 steps, and on SLIPPERY-GRIPPER and GRID-10X10 after 14 steps when executed with maximum caching capacity.

The graphs exhibit a common structure: *CPplan* is faster for shorter plans but *POMDP* eventually "catches up" (SANDCASTLE-67 exhibits the same structure but the intersection point is not plotted on Figure 4 because both algorithms can compute short plans very quickly in this domain). The position of the intersection point (where *POMDP* becomes faster than *CPplan*) depends on three factors

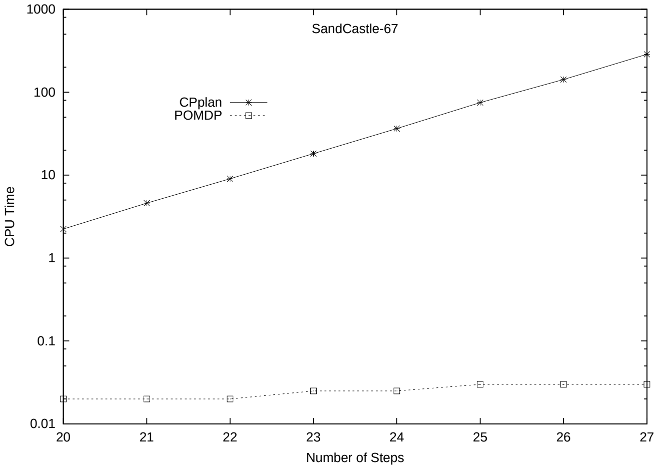

Figure 4: *CPplan* vs *POMDP* on SandCastle-67

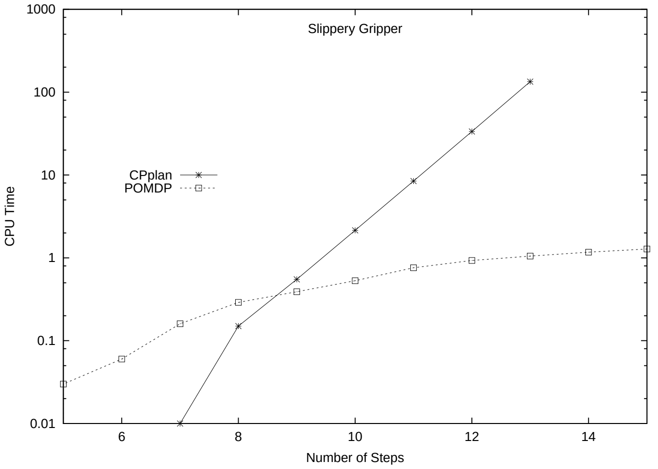

Figure 5: *CPplan* vs *POMDP* on Slippery Gripper

1. The ratio of dynamically reachable states at any step to the total number of states. This ratio is 1 for SANDCASTLE-67, 1/2 for SLIPPERY-GRIPPER, and 1/25 for GRID-10X10. The lower this ratio the better *CPplan* works.
2. The ratio of somewhere optimal α-vectors to the total number of α-vectors. The lower this ratio the better it is for *POMDP* algorithms.
3. How these two ratios compare.

Both algorithms have an exponential worst case complexity. However, *CPplan*'s complexity is always exponential in the plan length, with the base of the exponent being determined by S_{reach}, whereas, as noted above, the rate of growth in the number of α-vectors in the *POMDP* algorithms tends to slow down as plan length increases. This is partly due to the finite precision with which the value of these vectors is compared, in the *POMDP* implementation. It is because of this slow down in growth rate that we eventually see an intersection between the *CPplan* and *POMDP* curves.

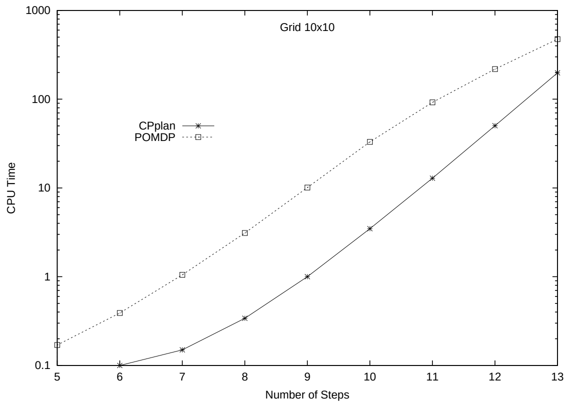

Figure 6: *CPplan* vs *POMDP* on Grid 10x10

Conclusions and Future work

We have presented a new AI planning technique for conformant probabilistic planning problems. Our algorithm works much better than previous approaches developed in the AI planning community. In particular, it is orders of magnitude faster than MAXPLAN (Majercik & Littman 1998) on all tested problems, which in turn is much faster than previous algorithms like Buridan (Kushmerick, Hanks, & Weld 1995).

We have also compared our approach with algorithms from decision theoretic planning, in particular value iteration algorithms for *POMDP*s. Here the performance of our system is less encouraging. Nevertheless, by analyzing the behavior of these two different approaches we have identified two quite different types of structure that each algorithm takes advantage of. Each algorithm has the advantage when the input problem has more of one type of structure than the other.

Most importantly, however, this analysis points out that there may be much to gain from finding algorithmic techniques that can take advantage of both types of structure simultaneously. In fact, it seems likely that there could be a significant synergy between these two types of structure, as the number of plans that are somewhere optimal on some *reachable* belief state, might be considerably smaller than either the number of reachable belief states or the number of somewhere optimal plans (when considering all possible belief states). We are currently working on developing such techniques.

Finally, an area we have not discussed is the use of branch and bound in solving *CPP*. Inspired by the MAXPLAN SAT encoding as we were, Walsh has developed a generic framework for CSPs involving random variables called stochastic constraint programming (Walsh 2002). He suggests algorithms based on branch and bound for solving such problems. An important future investigation would be to examine the effectiveness of these algorithms for solving *CPP*. However, it is not difficult to see that without caching the same computation can be performed exponentially many times during the tree search even in the presence of bounds pruning. Thus any technique for using bounds pruning will have to integrate well with a caching scheme.

References

Blum, A. L., and Langford, J. C. 1999. Probabilistic planning in the Graphplan framework. In *European Conference on Planning*, 319–332.

Blythe, J. 1998. *Planning under uncertainty in dynamic domains*. Ph.D. Dissertation, Carnegie Mellon University.

Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *AIPS-2000*, 52–61.

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 Conference on Uncertainty in Artificial Intelligence (UAI–97)*, 54–61. San Francisco, CA: Morgan Kaufmann Publishers.

Cassandra, A. 1999. POMDP-solve.
http://www.cs.brown.edu/research/ai/pomdp/code/index.html.

Cheng, H.-T. 1988. *Algorithms for Partially Observable Markov Decision Processes*. Ph.D. Dissertation, University of British Columbia.

Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research* 13:305–338.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem-proving. *Communications of the ACM* 4:394–397.

Do, M. B., and Kambhampati, S. 2001. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence* 132(2):151–182.

Goldman, R., and Boddy, M. 1994. Conditional linear planning. In *AIPS-1994*.

Kaelbling, L. P.; Littman, M. M.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101:99–134.

Kautz, H., and Selman, B. 1996. Pushing the envelope: planning, propositional logic, and stochastic search. In *AAAI-1996*, 1194–1201.

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

Littman, M. M. 1997. Probabilistic propositional planning: Representations and complexity. In *AAAI-1997*, 748–754. AAAI Press / The MIT Press.

Mackworth, A. K. 1977. On reading sketch maps. In *IJCAI-1977*, 598–606.

Majercik, S. M., and Littman, M. L. 1998. MAXPLAN: A New Approach to Probabilistic Planning. In *AIPS-1998*.

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

Onder, N., and Pollack, M. E. 1999. Conditional, probabilistic planning: a unifying algorithm and effective search control mechanisms. In *AAAI-1999*, 577–584.

van Beek, P., and Chen, X. 1999. CPlan: A constraint programming approach to planning. In *AAAI-1999*, 585–590.

Walsh, T. 2002 Stochastic Constraint Programming In *European Conference on Planning*.