Probabilistic Planning with Constraint Satisfaction techniques

Nathanael Hyafil
Department of Computer Science
University of Toronto
Toronto, Ontario
Canada M5S 1A4
1-416-978-4488
nhyafil@cs.utoronto.ca

Our research's aim is to explore the use of constraint satisfaction techniques in probabilistic planning. We first focus on two special cases that make different assumptions on the observability of the domain: the conformant probabilistic planning problem (*CfPP*), where the agent's environment is *not* observable, and the contingent probabilistic planning problem (*CtPP*), where the environment is *fully* observable. A paper describing some of our work on the first case has been accepted to the technical program of ICAPS 2003 under the title "Conformant Probabilistic Planning via CSPs". We are currently working on applying similar techniques to *CtPP*. So far, our research has resulted in exhibiting two independent types of structure that probabilistic planning problems tend to show. Decision theoretic techniques take advantage of state abstraction while our approach, and AI planning techniques in general, rely on reachability properties. Our ultimate goal is to design algorithms that can take advantage of both of these properties, not only in the special cases of *CfPP* and *CtPP* but in the general framework of probabilistic planning in partially observable domains.

The Conformant Probabilistic Planning Problem

The input to a *CfPP* is a tuple $\langle S, \mathcal{B}, A, G, n\rangle$. S, A and n are a set of states, a set of actions, and an integer specifying the horizon of the problem. $\mathcal{B}$ 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 $\mathcal{B}[S'] = \sum_{s' \in S'} \mathcal{B}[s']$. 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'. Finally, G is a goal which is a subset of S.

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

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

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, *CfPP* 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 $\pi(s)[s'] > 0$, i.e., π arrives at s' with non-zero probability when executed in the state s. Finally, for a particular action a and state s we say that s' *is a successor state of* s *under* a if $Pr(s, a, s') > 0$.

We use a factored representation of the state space S. The actions are represented using the sequential-effects decision tree formalism of (Littman 1997). And the goal is represented by a boolean expression over the state variables, that is satisfied only by the states in G.

The CSP Encoding

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 *CfPP* 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.

In most problems the state variables are boolean, but because we are encoding to a CSP, our formalism can deal with arbitrary domain sizes. For example, 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.

The CSP Constraints The CSP encoding of a *CfPP* 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. 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.

Reusing Intermediate Computations

A naive implementation in which all solutions are enumerated and the value of each plan evaluated, 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 can 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 ((Majercik & Littman 1998)).

Both MAXPLAN and our own system *CfPplan* 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. 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).

Our planner *CfPplan* uses a more refined caching scheme than MAXPLAN's. 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, *CfPplan* 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 *CfPplan* again encounters state s at step i it backtracks immediately without having to recompute these values. Note that storing the value of π on s is more general than storing its value on individual belief sets, since for any belief set $\mathcal{B}$ we can compute $\pi(\mathcal{B})$ from $\pi(s)$: $\pi(\mathcal{B}) = \sum_{s \in \mathcal{B}} \pi(s)$.

The *CfPplan* 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 *CfPplan* 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

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

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 the 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.

As mentioned above the *CfPplan* algorithm works by doing a depth-first search in a tree of variable instantiations.

CfPplan()
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
      $V = d$
      CfPplan()
      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: *CfPplan* algorithm

But 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 $\mathit{value}(\pi^{i-1}, s')$. 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 each 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 (the goal constraint does not allow any step n state that falsifies the goal to 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 backtracking from the initial call we can compute the value of all length n plans from the caches for the initial states.

The algorithm used is given in Table 1.

We have compared *CfPplan* to MAXPLAN on SandCastle-67 and on the Slippery Gripper problem. These results show that *CfPplan*'s caching mechanism is much faster (between 2 and 3 orders of magnitude) than MAXPLAN, which itself was faster than previous planners in the AI community. *CfPplan* also uses much less memory.

*POMDP*s

CfPP 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 *CfPP*). 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 *CfPP* in precisely this formalism requires a specialized encoding to handle the fact that in a *CfPP* the rewards are only given after n steps of the plan have been executed. But such an encoding would result in an important blow up of the state space. 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 *CfPP*. 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 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 than ϵ difference to the value of the plan.

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 that is optimal for some region of the belief space. 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 space, and we use this to compute 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. Therefore, we compute the α-vectors associated with all one step extensions of the plans in Φ_k and then prune those that are nowhere optimal to obtain Φ_{k+1}.

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 $\mathcal{B}$.

The key factor 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.

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.

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 *CfPplan* (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 analysis. In *CfPplan*, 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 *CfPplan* 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) and will end up instantiating all the state nodes that are reachable from the previous one in one step. Therefore only these reachable nodes will be expanded in the future search and the effects of actions on all other (non-reachable) states will not be unnecessarily considered.

Comparison with *POMDP*s

We were able to solve *CfPP* problems using a VI based *POMDP* solver written by Cassandra (Cassandra 1999). Cassandra's solver implements (among others) the incremental pruning algorithm described in (Cassandra, Littman, & Zhang 1997).

Our experiment demonstrate a common structure: *CfPplan* is faster for shorter plans but *POMDP* eventually "catches up". The position of the intersection point (where *POMDP* becomes faster than *CfPplan*) depends on three factors

1. The ratio of dynamically reachable states at any step to the total number of states. The lower this ratio the better *CfPplan* 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, *CfPplan*'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 *CfPplan* and *POMDP* curves.

Current and Future Work

We have recently extended our research to both finite and infinite horizon *CtPP*s. The two resulting algorithms are very similar to that of *CfPplan* in that they exploit the same type of redundancies. The main difference is that the solution of a fully observable problem is a mapping from states to actions, indicating how to act in each state that can be encountered from the initial state(s). The resulting caching mechanism for *CtPplan*, our *CtPP* solver, requires very little memory use, since we are storing values for each action instead of each sequence of actions. Contingent probabilistic planning problems are very similar to fully-observable MDPs. We therefore evaluate *CtPplan* against SPUDD, an MDP solver described in (Hoey *et al.* 1999).

From empirical results, we are able to draw similar conclusions to those from *CfPP*. As α-vectors provide a power of abstraction over the belief state space to *POMDP* algorithms, SPUDD uses Algebraic Decision Diagrams to efficiently abstract the state space of an MDP.

In general, techniques from the decision-theoretic planning community rely very much on abstraction mechanisms, whereas AI planning algorithms utilize dynamic reachability properties. We believe an efficient algorithm for solving large probabilistic planning problems must be able to take advantage of both of these types of structure. We are currently exploring such techniques for *CfPP* ad *CtPP* and plan to expand our work to the more general framework of partially observable probabilistic planning.

References

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.

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

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Uncertainty in Artificial Intelligence, Proceedings of Annual Conference*, 279–288.

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

Majercik, S. M., and Littman, M. L. 1998. MAXPLAN: A New Approach to Probabilistic Planning. In *The Fourth International Conference on Artificial Intelligence Planning Systems*.