Utilizing Structured Representations and CSPs in Conformant Probabilistic Planning

Nathanael Hyafil and **Fahiem Bacchus**[1]

Abstract. A CSP based algorithm for the conformant probabilistic planning problem (CPP) has been presented by Hyafil & Bacchus. Although their algorithm displayed some interesting potential when compared with traditional POMDP algorithms, it was developed using a "flat" representation. In this work we revisit this algorithm and develop a version that utilizes a structured representation. The structured representation can be exponentially more efficient than the flat representation when dealing with the structured problems that are typical in AI. Our new structured version of their algorithm allows us to demonstrate that the CSP approach to CPP can be much more effective than traditional POMDP algorithms for an interesting range of problems. This is contrary to previously presented results, and makes the application of CSP techniques to decision theoretic planning a promising area for further work.

1 Introduction

We address the conformant probabilistic planning problem (CPP). Conformant planning problems are problems where there is no sensing: the system state cannot be tested after an action is executed. Since the plan is oblivious to the system state, the plan cannot be conditional—there would be no way to determine which branch to follow in a conditional plan. Instead conformant plans are linear sequences of actions. The aim of such a plan is to achieve the goal no matter what initial state the plan is applied to and no matter what happens during its execution. Conformant plans are useful in situations where sensing is too expensive or when a failure has occurred and sensing is no longer available.

Requiring a conformant plan to always succeed is a very strong condition, a condition that cannot always be satisfied. Probabilistic conformant plans are a useful generalization of conformant plans in the case when we can quantify the different possible initial states, and the different possible execution paths with probabilities. A probabilistic conformant plan is one that has a certain probability of success, and the conformant probabilistic planning problem, CPP, is the problem of finding a plan with maximum probability of success.

A planning algorithm for solving CPPs was presented by Hyafil & Bacchus in [9]. This algorithm employed a CSP encoding of the problem, and a backtracking tree search similar in spirit to the SAT based encoding used in the MAXPLAN planner [11] (which also solved CPPs). Although the CSP approach yielded consistently superior performance to the SAT based MAXPLAN, as demonstrated in [9], its performance was generally inferior to traditional POMDP decision theoretic algorithms. Such POMDP algorithms solve partially observable Markov decision processes, and can also solve CPP as a special case. One of the main problems with the approach presented in [9] is that the algorithm they specified required space exponential in the plan length. Nevertheless, the core ideas embodied in the algorithm did captured a number of useful intuitions for solving CPPs more effectively.

In this paper we recast the algorithm of [9] so that it can utilize structured representations, specifically decision diagrams [5]. Using structured representations in planning algorithms is a known technique, e.g., binary decision diagrams (BDDs) have been used to solve the non-probabilistic conformant planning problems [2], and algebraic decision diagrams (ADDs) have been used to solve probabilistic fully observable planning problems (MDPS) [8]. In this paper, we employ Algebraic Decision Diagrams (ADDs) [12] to represent the information required to solve CPPs in a more compact form. It is well known that decision diagrams are not always compact, but empirically they are often very effective. Our empirical results demonstrate that this is generally the case in our application as well.

In the rest of the paper we first present a more formal description of the CPP problem and describe the previous CSP based algorithm for solving CPPs. Then we demonstrate how ADDs can be employed to improve the algorithm's space requirements and often its runtime. These techniques can also be applied to other types of planing problems (encoded as CSPs). Then we present empirical data about the performance of the improved algorithm, and compare the approach with traditional POMDP algorithms. This data demonstrates that backtracking based CSP techniques, when augmented by caching, have considerable advantages over standard POMDP algorithms on various types of problems.

2 Background

CPP: A CPP planning domain is described by a set of m state variables, $v_1, \ldots, v_m$, where v_i can take on any of k_i different values. Every distinct setting of the m state variables defines a state, and every state is defined by some setting the state variables. Thus there can be $\prod_{j=1}^{m} k_j$ distinct states in $\mathcal{S}$, the set of all states (not all of them need be legal states). The different states in $\mathcal{S}$ represent all of the different possible configurations of the environment.

The agent is in some initial belief state $\mathcal{B}$, which is a probability distribution over $\mathcal{S}$. For any state $s \in \mathcal{S}$, $\mathcal{B}[s]$ is the probability of s. The agent does not know the true initial state of its environment, but $\mathcal{B}[s]$ measures how likely it is that the true state is s. Thus if $\mathcal{B}[s] = 0$ then s is not a possible initial state. For a subset $S \subseteq \mathcal{S}$, we define $\mathcal{B}[S] = \sum_{s \in S} \mathcal{B}[s]$. The agent wants to achieve a goal G, which for now we take to be a subset of $\mathcal{S}$.

The agent has available some set of actions A. The effects of these actions are not guaranteed, rather they are only achieved with some probability. Each action $a \in \mathcal{A}$ is modeled by a transition probability. For each pair of states s and s', $Pr(s, a, s')$ is the probability that a

[1] Department of Computer Science, University of Toronto, Toronto, Canada. email: nhyafil@cs.utoronto.ca, fbacchus@cs.utoronto.ca

yields s' when executed in s. For example, with probability 0.25 a particular action a might fail to have any effect. In this case we would have $Pr(s, a, s) = 0.25$: with probability 0.25 the action yields the same state.

Our approach also allows a slight generalization where we allow actions to have standard *preconditions*. The action cannot be applied to states violating its precondition, which translates into $Pr(s, a, s') = 0$ for all states s failing to satisfy a's preconditions and for all states s'. Of course, in this case Pr no longer specifies a probability distribution since it no longer sums to 1. Note that this yields a different behavior than modeling failed preconditions as making the action a no-op.

A plan π in a CPP problem specifies a set of *execution paths*. If $\pi = \langle a_1, \ldots, a_k \rangle$, then each execution path for π is a length $k + 1$ sequence of states $s_0, s_1, \ldots, s_k$ where $\mathcal{B}[s_0] > 0$, and for each pair (s_{i-1}, s_i) we have that $Pr(s_{i-1}, a_i, s_i) > 0$. The first state s_0 is a possible initial state, and each subsequent state is a possible successor of the previous state under the action specified by π. The probability of an execution path is $\mathcal{B}[s_0] \prod_{i=1}^{k} Pr(s_{i-1}, a_i, s_i)$: the probability that executing π will yield this particular sequence of states. The *value* of π is equal to the sum of the probabilities of all of its execution paths whose final state is in G. In other words the value of π is the probability that π achieves the goal. Note that we are only interested in goal achievement in the final state. We could not terminate π earlier even if π achieves the goal before it has completed since we cannot observe whether or not the goal has been achieved.

From this point of view, standard preconditions correspond to assigning value (probability) zero to any execution path such that some s_{i-1} violates the preconditions of a_i. If instead preconditions were modeled as no-ops, an execution sequence containing a precondition violation might still achieve the goal and contribute to the plans value.[2]

It is not difficult to see that much more general notions of goals can be accommodated. For example, the goal could be to maximize some metric quantity in the final state. In this case we could define the value of a plan to be the expected value of the metric quantity over all of the plan's execution paths. Similarly, more complex temporal goals could be defined, as long as we can determine the value of each execution path with respect to such a goal we can determine the value of the plan. CSP based algorithms for CPP can easily accommodate general goal of this form.

CSP Encoding: Encoding planning problems as constraint satisfaction problems is a known technique in classical planning, e.g., [13, 7, 10]. Hyafil & Bacchus have adapted this technique to solve CPPs [9].

A CSP consists of a set of variables and a set of constraints over these variables. To encode a length n CPP as a CSP, $n + 1$ copies of the state variables $v_1, \ldots, v_m$ are used, indexed by the plan step; each copy is used to represent one of the states in a possible execution path of a n-length plan. There are also n action variables, each one with domain equal to the set of all actions in the domain. Since the actions are probabilistic, knowing the i-th state is not sufficient to determine the effect of the i-th action. Hence an additional set of "random" variables are required, one set for each step of the plan. The different legal settings of the i-th random variables are in one-to-one correspondence with the different possible probabilistic effects of the i-th action when applied to the i-th state. Thus each setting of these variables has a particular probability—the probability the i-th action will have this particular effect–and once these variables have been set the i-th action's effects are determined. This technique was first used in [11].

There are three types of constraints: initial state constraints, goal state constraints, and action constraints. These constraints force the initial state s_0 (as specified by the 0-th group of state variables) to be one where $\mathcal{B}[s] > 0$, the goal state s_n (as specified by the n-th group of state variables) to be in G, and the transitions between s_i and s_{i+1} to be compatible with a_i given that a_i's random effects have been determined by the i-th random variables.

A solution to a CSP is a setting of the CSP variables that satisfies all of the constraints. The solutions of the CSP that encodes the CPP are in one to one correspondence with the execution paths of the length n plans that reach the goal (if we remove the goal constraint, then the solutions will be all execution paths irrespective of whether or not these paths reach the goal). In a particular CSP solution, ρ, the setting of the n action variables specifies the plan π, the setting of the $n + 1$ collections of state variables specifies an execution path for π, and the probability of this particular execution path can be obtained from the setting of the n collections of random variables and the setting of the variables specifying the initial state.

Solving CPP: Conceptually CPP can be solved by finding all solutions to its CSP encoding and for each plan summing the probability of the solutions corresponding to its execution paths. Standard backtracking search algorithms can be used to find all solutions. In this context, the CSP constraints serve to eliminate zero probability execution paths, and constraint propagation serves to prune some of the path prefixes that have probability zero of reaching the goal.

In Hyafil & Bacchus' planning system, *CPplan*, the CSP variables are instantiated in plan order during backtracking search. Starting with the initial state variables and proceeding in order of plan steps, for each step the state variables are first instantiated, thus determining the current state, then the action, and then the random variables, thus determining the next state.

Ignoring the goal constraint, the other constraints prune illegal paths in the tree. Each remaining full path corresponds to a particular execution path of some plan. Along each path a node that instantiates the last of the i-th group of state variables corresponds to the i-th state visited, and a node that instantiates the i-th action variable corresponds to the i-th action taken in the plan. Prefixes of full paths in the tree represent a particular execution path of a particular plan prefix. Below this prefix are all possible extensions of this prefix, i.e., the set of all execution paths ρ for all plans π such that (1) ρ extends the current execution path prefix, and (2) π extends the current plan prefix.

With the goal constraint, the solutions now correspond only to successful execution paths (paths reaching the goal). Constraint propagation can then be used to perform "goal" lookahead. That is, propagation allows the backtracking search to detect that certain path prefixes have no extension reaching the goal. This can have a significant effect in some domains. The ability to employ constraint propagation to perform goal reachability analysis is one of the main differences between the CSP approach and heuristic search in belief space [3].

Reusing Intermediate Computations: Finding and summing over all solutions to the CSP encoding is not practical. Hyafil & Bacchus showed how dynamic programming techniques could be employed within the backtracking search to cache and reuse interme-

[2] The choice of how to model preconditions thus becomes a modeling decision: in some domains preconditions are hard constraints on plan executability while in other domains preconditions are simply constraints on the action's effects.

diate results.[3] It is known that caching in backtracking search can have a profound effect on worst case complexity when examining all solutions, as is required when solving CPP [1].

The caching scheme employed was as follows. Let *value*(τ, s) be the value of a length $n-i$ plan suffix τ when executed in state s. This is the probability that τ reaches the goal when executed in state s with $n-i$ steps to go in the plan. During the backtracking search we will visit nodes ν corresponding to a particular state s_i at the i-th step of an execution path. (ν instantiates the last of the i-th group of state variables). The path to ν will correspond to some execution path prefix $s_0, s_1, \ldots, s_i$ (ending at ν with s_i) for some plan prefix $a_1, a_2, \ldots, a_i$, and the subtree under ν will contain all successful execution path suffixes $s_{i+1}, \ldots, s_n$ ($s_n \in G$) for all possible plan suffixes $\tau = a_{i+1}, \ldots, a_n$. The probability of any of these execution path suffixes is $\prod_{j=i+1}^{k} Pr(s_{j-1}, a_j, s_j)$, and summing up these probabilities for a fixed τ gives us *value*(τ, s_i).

On visiting node $\nu{=}s_i$ the quantity *value*(τ, s_i) can be computed and cached for all length $n{-}i$ plan suffixes τ. From this information, the value of *all* length $(n{-}i){+}1$ plan suffixes $\langle a_i; \tau\rangle$ in the previous state s_{i-1} can easily be computed. In particular, *value*$(\langle a_i; \tau\rangle, s_{i-1}) = \sum_{s_i} Pr(s_{i-1}, a_i, s_i)$*value*$(\tau, s_i)$: $\langle a_i; \tau\rangle$ can succeed in state s_{i-1} by transitioning to state s_i (with probability $Pr(s_{i-1}, a_i, s_i)$) and from there reaching the goal (*value*(τ, s_i)). Since in the subtree below (s_{i-1}, a_i) all states s_i with $Pr(s_{i-1}, a_i, s_i) > 0$ are visited and these are the only states we need sum over, by the time the search has backtracked out of this subtree, all of the values it needs to compute *value*$(\langle a_i; \tau\rangle, s_{i-1})$ will be available. Recursively, we can in this manner compute and cache the value of all length n plans π. The critical element is that by caching *value*(τ, s_i) for all length $n{-}i$ plan suffixes τ, we need visit s_i at step i only once. If some other path in the tree (corresponding to another plan/execution path prefix) visits s_i at step i, we need not explore the subtree below s_i again—we can reuse the cached *value*(τ, s_i) information.

Two pieces of information can be stored at every step i of the plan. The first is a list of visited states at step i—used to determine if the value of all plan suffixes from that state has already been cached. The second is a table for each visited state containing the value of every length $n-i$ plan suffix in that state. Hyafil & Bacchus utilized a simple tabular representation for this information, which required size exponential in $n{-}i$. The tables for the initial states thus required space exponential in the plan horizon n (a value had to be stored for each length n plan). Hence, although this work provided some useful insights about applying CSP techniques to solving CPP its space requirements do not allow it to scale.[4] Nevertheless, on particular problems the approach was still faster than traditional POMDP algorithms, and in terms of worst case complexity it was no worse than other algorithms for POMDPs.

Convinced that the CSP approach had more potential than could be demonstrated by this previous algorithm, we have investigated techniques for addressing the weaknesses of the previous approach. The main difficulty of the previous approach is that the tabular representation utilized *always* requires exponential space; traditional POMDP algorithms based on the idea of α-vectors need exponential space only in the *worst* case. Hence we investigated ways of improving the algorithm so that it needed exponential space only in the worst case rather than always. Of the techniques we investigated for saving space, the utilization of structured representations (which can save space by sharing information) was very successful, and we report on that technique in this paper. This technique allows us to empirically validate the potential of the CSP approach by showing that it can be much more effective than traditional POMDP algorithms on certain problems.

[3] The differences between searching the CSP with dynamic programming and the more traditional use of dynamic programming for solving POMDPs, e.g., [6], are explained in [9].

[4] Turning off the cache and recomputing *value*(s_i, τ) when need was so time expensive that even the smallest problems (that required reasonable sized tables) could not be solved. So this simple trading of time for space does not work.

3 Saving Space

Branch and Bound: The first technique we investigated was the well known idea of branch and bound. We are trying to compute a maximum valued plan, so instead of computing and storing the value of all plans, we can eliminate from consideration plans as soon as we know that their value cannot be the maximum. In particular, by keeping track of the value of the current best solution, one can sometimes detect that a better solution cannot lie in the subtree below a node, and hence that that subtree need not be searched. In CPPs however, the subtrees do not contain solutions to the CPP. Rather they only contain incremental information about the value of the plans Π that extend the current plan prefix μ. To obtain the full value of any of the plans in Π we would have to search the subtrees below *all* of the states that could be visited by μ. Hence, in CPP branch and bound is only able to avoid subtrees in which every plan is already known to be sub-optimal.

Furthermore, although branch and bound only requires linear space when applied to planning under full observability [14], this is not the case when it is applied to solving CPP. In particular, in fully observable planning one need only keep bounds along the current path, and a depth first backtracking search can perform branch and bound in space linear in the plan horizon. With full observability the agent can choose what action to take at each state. Thus the value of choosing an action in a particular state is determined solely by the subtree below that state action pair. In CPP choosing an action at step i means that one is committed to executing that action in all states that could arise at step i. Hence the value of choosing that action in a particular state is not sufficient to determine the overall value of choosing that action. Once all of the details were addressed it was found that branch and bound had disappointing performance: because of its weak pruning power and the information that must be kept to perform the bounding, branch and bound neither saved time nor space over the simple tabular representation.

ADDs: The technique that did perform well was to utilize Algebraic Decision Diagrams (ADDs) [12] to represent the cached value of plan suffixes. ADDs are graphical representations of functions from a set of boolean variables $V_1, \ldots, V_m$ to the reals. Representing such a function f in tabular form would require 2^m table entries. However, if f takes on much fewer than 2^m different values, and there are compact propositional formulas representing the sets of inputs that generate each of these different values, the ADD representation of f can be vastly more compact. Furthermore, algorithms exists for performing various arithmetic operators on ADD represented functions. For example, computing the new ADD representing the function h such that $h(\vec{V}) = f(\vec{V}) * g(\vec{V})$ can be done in time linear in sizes of the ADDs representing f and g. If f and g have compact ADD representations, h can be computed much faster than the time it would take to compute it from tabular representations of f and g. However, ADDs are not always compact, and in the worst case they can can have size as large $O(m)2^m$ (e.g., when every input has a

different value). When their size approaches that of the tabular representation, one must pay a significant overhead for using them. ADDs where the range of the function is restricted to 0/1 are the same as BDDs [5]. BDDs are generally significantly more compact and more efficient to use, and are particularly useful for representing sets and for performing set operations. As mentioned in the introduction, both types of decision diagrams have been used before in planning.

Using ADDs with Caching: In the caching algorithm we utilize n "visited-state" 0/1 ADDs (BDDs). The i-th of these stores the states that have already been cached at step i. When the backtracking search visits a state at step i it first checks this ADD to see if the state has already been computed. If it has been seen before the cached values can be used and the subtree below can be avoided. In addition to the visited-state BDD we utilize n "state/plan-suffix" ADDs. The i-th of these, ADD_i, maps s_i, a state at step i and any length $n-i$ plan suffix τ to *value*(s_i, τ). That is, ADD_i stores the value tables for all states visited at step i. By combining the tables for all states into one ADD we facilitate the sharing of more structure. In particular, we might have that for two states s and s' visited at step i, *value*$(s, \tau) =$ *value*(s', τ) for many plan suffixes τ. In this case ADD_i can potentially exploit this structure to save space.

After backtracking from a state s_i we are recursively assured that ADD_i contains *value*(s_i, τ) for all length $n-i$ plan suffixes τ. As we visit the successor states s_i generated by action a_{i-1}, we accumulate an ADD representing the function $T = \sum_{s_i} Pr(s_{i-1}, a_{i-1}, s_i) \times s_i$. This function maps a set of states at step i to the probability that the set is visited by a_{i-1} when executed in s_{i-1}. After we have visited all of a_{i-1}'s successor states, i.e., after we have backtracked to a_{i-1}, we compute $T \times ADD_i$, followed by the existential abstraction of all variables used to represent states at step i. This new ADD T' represents a function mapping every length $n-i$ plan suffix τ to the expectation of *value*(s_i, τ), where the expectation is taken with respect to the distribution over s_i states generated by executing a_{i-1} in state s_{i-1}. Finally we conjoin T' with the state s_{i-1} and the action a_{i-1} and add this into ADD_{i-1}. This operation updates ADD_{i-1} so that it now contains *value*(s_{i-1}, τ) for all length $n-i+1$ plan suffixes τ that *start* with a_{i-1}. After visiting all actions that can be taken from s_{i-1}, i.e., after backtracking to s_{i-1}, we have augmented ADD_{i-1} so that it has complete information about state s_{i-1}. We can then add s_{i-1} to the set of visited states at step $i-1$. Recursively, at the top of the tree ADD_0 will contain the values of all n length plans for all initial states. From this information, computing the plans that have maximal expected value (with respect to the probability distribution over the initial states) is straight forward.

4 Empirical Results

In order to evaluate our modified CSP algorithm we utilized two test domains, GRID-10X10 and a probabilistic version of the standard AI planning benchmark LOGISTICS. We also tried two other problems reported on in [9], SANDCASTLE-67 and SLIPPERY-GRIPPER. However on these problems, (both of which have very small state spaces) the CSP approach was still not competitive with a traditional POMDP algorithm. This and our results on the other two domains indicates that traditional POMDP algorithms and the CSP approach have distinct areas of coverage.

The POMDP algorithm we tested against was the witness algorithm of [6]. The advantage of this algorithm is that it can easily be adapted to solve CPP without having to resort to specialized encodings (see [9] for more details). All experiments were performed on a 2.4 GHz Xeon machine with 3GB of RAM.

GRID-10X10: GRID-10X10 has a robot starting in some fixed initial (x, y) location with the goal of reaching the upper right corner $(9, 9)$ of a 10X10 grid. There are 4 move actions, *right, left, up, down,* each moving the robot in the right direction with probability 0.8, in the opposite direction with probability 0, and in either of the other two directions with probability 0.1.

GRID-10X10 has a state space size of 100 (the robot can be in any location on the grid), but each action can only reach 3 states. Hence, GRID-10X10 has significant deterministic structure in its transition probabilities. The CSP approach, with its constraint propagation, is able to take advantage of this deterministic structure. The results for GRID-10X10 are shown in Table 1. The results compare the previous tabular representation, the ADD representation, and the witness algorithm. We see that ADDs offer a significant improvement over tables, and that the CSP approach is significantly better than the POMDP algorithm.

Init.	n	C+Table	C+ADD	POMDP
(7,6)	5	0.01	0.06	1.21
(6,6)	6	0.01	0.10	1.44
(6,5)	7	0.01	0.16	2.15
(3,3)	12	2.97	1.12	237.0
(3,2)	13	11.72	1.72	508.8
(2,2)	14	oom	2.31	949.5
(1,1)	16	-	4.79	2282.3
(0,0)	18	-	11.44	3098.6
(0,0)	19	-	129.12	-

Table 1. Time on GRID-10X10 in CPU sec. Init. (initial state), n (plan length), C+Table (caching with tables), C+ADD (caching with ADDs), POMDP (witness POMDP algorithm), omm (out of memory), - ($>$ 3,600 sec).

Logistics: The other domain we report on with was a conformant version of the standard Logistics domain due to [4]. In Hoffmann's version everything in the domain is deterministic, except for some non-determinism in the initial locations of the packages. We changed this initial non-determinism into an initial probability distribution, and added a probabilistic effect to the *load*, and *unload* actions. If the package is in the same location as the vehicle *load* loads it into the vehicle with some probability *lprob* (otherwise the probability of the package becoming loaded is zero). Similarly if the package is in a vehicle *unload* loads it with some probability *uprob* (zero if the package is not in the vehicle).

The other actions (involving moving the vehicles) remain deterministic, with their standard preconditions; however all preconditions involving the packages were removed. (In this domain we do not know with certainty where the packages are). We also treated the preconditions as being strict. So for example any execution path that tried to drive the truck from location a to location b would fail (have zero probability) if the truck was not at location a.

Intuitively, in a logistics domain it makes sense to assume that vehicles don't become lost (to a first approximation), but that packages might. Adding uncertainty to *load* and *unload* (and not to vehicle movements) realizes this intuition.

The results on four different problems from this domain are shown in Table 2. The probabilities of success for *load* are 0.875 for trucks and 0.9 for airplanes and for *unload* 0.75 and 0.8 respectively. *CPplan* in its tabular representation could not solve problems with so many actions and states because of the memory required. As the data shows *CPplan* with ADDs can generate at least 12 step plans in all of these domains whereas POMDP algorithms can solve at best 9-step problems and no more than 2 steps for the biggest instance within the $15,000$ seconds allowed (raising this limit to $100,000$ seconds only

Name	p2-2-2		p3-2-2		p4-2-2		p2-2-4	
l-t-p-a	4-2-2-1		6-2-2-2		8-2-2-1		3-2-4-1	
$\|S\|$	392		1458		3872		19208	
$\|A\|$	30		46		66		54	
n	CPplan	Pomdp	CPplan	Pomdp	CPplan	Pomdp	CPplan	Pomdp
1	0	0.2	0	5.8	0	73.7	0	648
2	0	1.8	0	48.8	0	1123	0	13974
3	0	6.5	0	116	0	3184	0	–
4	0	21.5	0	230	0.1	9119	0	–
5	0.1	66.6	0	451	0.5	–	0.3	–
6	0.6	212	0.3	877	2.4	–	2.1	–
7	1.7	737	1b.9	1796	10.9	–	11.4	–
8	3.7	3495	5.7	3871	34.9	–	39.9	–
9	7.6	–	15.6	9378	83.0	–	185	–
10	14.8	–	28	–	208	–	391	–
11	42.6	–	45.8	–	304	–	722	–
12	281	–	68.8	–	648	–	1221	–
13	oom	–	122	–	6398	–	1445	–
14		–	414	–	oom	–	3725	–
15		–	oom	–	–	–	oom	–

Table 2. Solution Time on various Logistics problems in CPU seconds using Caching with ADDs. l-t-p-a: number of locations, trucks, packages and airplanes. $|S|$: number of states, $|A|$: number of actions. n (plan length). "–": more than 15000 seconds; oom: out of memory.

allows for one additional step to be solved!). Also on problems that both techniques can tackle, *CPplan* can be up to 3 orders of magnitude faster.

Finally in order to evaluate how the amount uncertainty affects the relative performance of *CPplan* and POMDP we ran the problem suites using entirely 0/1 probabilities (so that the problem becomes non-deterministic rather than probabilistic) and probabilities of 0.5 (maximal uncertainty). For reasons of space we report only on problem p-3-2-2 this problem was somewhat more tractable for POMDP.

The results show that the relative performance does not change much: CPplan remains orders of magnitude faster than POMDPs. However, both algorithms find the deterministic problem significantly easier. (This also helps illustrate how much harder CPP is than non-probabilistic conformant planning.) Interesting, in the full uncertainty case CPplan is a bit faster than the results given in Table 2—with equal probability values the ADDs tend to be more compact as the functions tend to take on fewer distinct values.

	Deterministic		Full Uncertainty	
n	CPplan	Pomdp	CPplan	Pomdp
1	0	5.0	0	6.4
2	0	49.7	0	53.6
3	0	104.0	0	127.7
4	0	170.4	0	254.1
5	0.1	279.7	0	498.3
6	0.5	427.7	0.3	967.8
7	2.5	674.8	1.5	1934
8	8.9	1054	5.3	4031
9	23.6	1614	13.6	9163
10	31.4	2686	26.0	–
11	40.7	3907	32.7	
12	35.0	5776	40.7	
13	41.9	8536	51.7	
14	69.9	11777	128.8	
15	68.2	–	1124	
16	91.7		oom	
17	99.0			
18	103.6			
19	91.6			
20	126.6			
21	137.1			
22	oom			

Table 3. Solution Time on P-3-2-2 with "extreme" probability settings (CPU seconds) using Caching with ADDs. n: plan length. "–": more than 15000 seconds; oom: out of memory.

5 Conclusions

We have shown that ADDs can significantly improve the capability of the CSP approach to CPP to the point where it consistently outperforms all other algorithms we are aware of some interesting problems. A number of open questions remain, including that of how different domain representations might affect the size and efficiency of the ADDs, how domain specific knowledge might be employed, and how computational leverage can be obtained from trying to find approximate solutions to CPP rather than optimal ones. The two domains we have experimented with here appear to work well with the CSP approach because they have transitions with restrictive reachability. A final open question is whether this intuition of can be formalized to provide deeper answers to the question of characterizing the domains on which the CSP approach is superior to traditional POMDP algorithms.

REFERENCES

[1] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi, 'Algorithms and complexity results for #sat and bayesian inference', in *Symposium on Foundations of Computer Science (FOCS)*, pp. 340–351, (2003).

[2] Piergiorgio Bertoli, Alessandro Cimatti, and Marco Roveri, 'Heuristic search + symbolic model checking = eficient conformant planning', in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 467–472, (2001).

[3] Blai Bonet and Héctor Geffner, 'Planning with incomplete information as heuristic search in belief space', in *Proceedings of the International Conference on Artificial Intelligence Planning (AIPS)*, pp. 52–61, (2000).

[4] Ronen Brafman and Joerg Hoffmann, 'Conformant planning via heuristic forward search', in *Workshop on Planning Under Uncertainty and Incomplete Information*, (2003).

[5] R. E. Bryant, 'Symbolic boolean manipulation with ordered binary decision diagrams', *ACM Computing Surveys*, **24**(3), 293–318, (September 1992).

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

[7] Minh Binh Do and Subbarao Kambhampati, 'Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP', *Artificial Intelligence*, **132**(2), 151–182, (2001).

[8] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier, 'Spudd: Stochastic planning using decision diagrams', in *Uncertainty in Artificial Intelligence, Proceedings of Annual Conference (UAI)*, pp. 279–288, (1999).

[9] Nathanael Hyafil and Fahiem Bacchus, 'Conformant probabilistic planning via csps', in *International Conference on Automated Planning and Scheduling (ICAPS 2003)*, pp. 205–214, (2003).

[10] Adriana Lopez and Fahiem Bacchus, 'Generalizing graphplan by formulating planning as a csp', in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, (2003).

[11] Stephen M. Majercik and Michael L. Littman, 'MAXPLAN: A New Approach to Probabilistic Planning', in *The Fourth International Conference on Artificial Intelligence Planning Systems*, pp. 86–93, (1998).

[12] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, 'Algebraic Decision Diagrams and Their Applications', in *IEEE /ACM International Conference on CAD*, pp. 188–191, Santa Clara, California, (1993). IEEE Computer Society Press.

[13] P. van Beek and X. Chen, 'CPlan: A constraint programming approach to planning', in *Proceedings of the AAAI National Conference*, pp. 585–590, Orlando, Florida, (1999).
[14] Toby Walsh, 'Stochastic constraint programming', in *Proceedings of the European Conference on Artificial Intelligence*, (2002).