ASU CSE TR-06-013; July 2006

Sequential Monte Carlo in Reachability Heuristics for Probabilistic Planning

Daniel Bryce, [a]Subbarao Kambhampati, [a] and David E. Smith [b]

[a] *Arizona State University, Department of Computer Science and Engineering Brickyard Suite 501, 699 South Mill Avenue, Tempe, AZ 85281*

[b] *NASA Ames Research Center, Intelligent Systems Division MS 269-2 Moffett Field, CA 94035-1000*

Abstract

The current best conformant probabilistic planners encode the problem as a bounded length CSP or SAT problem. While these approaches can find optimal solutions for given plan lengths, they often do not scale for large problems or plan lengths. As has been shown in classical planning, heuristic search outperforms CSP/SAT techniques (especially when a plan length is not given a priori). The problem with applying heuristic search in probabilistic planning is that effective heuristics are as yet lacking.

In this work, we apply heuristic search to conformant probabilistic planning by adapting planning graph heuristics developed for non-deterministic planning. We evaluate a straightforward application of these planning graph techniques, which amounts to exactly computing the distribution over reachable relaxed planning graph layers. Computing these distributions is costly, so we apply Sequential Monte Carlo (SMC) to approximate them. A key issue we explore within SMC is how to automatically determine the number of samples required for effective heuristic computation. We demonstrate on several domains how our approach enables our planner to far out-scale existing (optimal) probabilistic planners and still find reasonable quality solutions.

Key words: Planning, Heuristics

1 Introduction

Despite long standing interest [21,25,15,16], probabilistic plan synthesis algorithms have a terrible track record in terms of scalability. The current best conformant probabilistic planners are only able to handle very small problems. In contrast, there has been steady progress in scaling deterministic planning. Much of this progress has come from the use of sophisticated reachability heuristics. In this work, we show how to effectively use reachability heuristics [4] to solve conformant probabilistic planning (CPP) problems. We use work on planning graph heuristics for non-deterministic planning [5,12] as our starting point.

We investigate an extension of the work by Bryce, et. al. [5] that uses a planning graph generalization called the labelled uncertainty graph (*LUG*). The *LUG* is used to symbolically represent a set of relaxed planning graphs (much like the planning graphs used by Conformant GraphPlan, [30]), where each is associated with a possible state. While the *LUG* (as described by, [5]) works only with state uncertainty, it is necessary in CPP to handle action uncertainty. Extending the *LUG* to consider action uncertainty involves symbolically representing how at each level CGP explicitly splits the planning graph over all joint outcomes of uncertain actions. In order to capture all possible worlds, each time step has a set of planning graph layers defined by the cross product of an exponential set of joint action outcomes and an exponential number of proposition layers from the previous level. The planning graph becomes tree-like, where each branch corresponds to a possible world. A possible world corresponds to the joint outcome of a set of uncertain events (i.e., a path starting with a possible initial state and continuing through a joint outcome of actions at each planning graph level).

Without uncertain actions, the *LUG* worked well because while there were an exponential number of possible worlds at each time step the number was held constant. With uncertain actions, an explicit or symbolic representation of planning graphs for all possible worlds at each time step is *exactly* representing an exponentially increasing set. Since we are only interested in planning graphs to compute heuristics, it is both impractical and unnecessary to exactly represent all of the possible worlds. We turn to approximate methods for representing the possible worlds. Since we are applying planning graphs in a probabilistic setting, we can use Monte Carlo techniques to construct planning graphs.

There are a wealth of methods, that fall under the name sequential Monte Carlo (SMC) [9] for reasoning about a hidden random variable over time. SMC applied to "on-line" Bayesian filtering is often called particle filtering, however we use SMC for "off-line" prediction. The idea behind SMC is to represent a probability distribution as a set of samples (particles), which evolve recursively over time by sampling a transition function. In our application, each particle is a possible world in a conformant planning graph (i.e., a particle is a (simulated) deterministic planning graph). Using particles is much cheaper than splitting over all joint outcomes of uncertain actions to represent the true distribution over possible worlds in the planning graph. By using more particles, we capture more possible worlds, exploiting the natural affinity between SMC approximation and heuristic accuracy.

The SMC technique requires multiple planning graphs (each a particle), but their number is fixed. We could represent each planning graph explicitly, but they may have considerable redundant structure. Instead, we generalize the *LUG* to symbolically represent the set of planning graph particles in a planning graph we call the Monte Carlo *LUG* ($\mathcal{M}c$*LUG*). We show that by using the $\mathcal{M}c$*LUG* to extract a relaxed plan heuristic we are able to greatly out-scale the current best conformant probabilistic planner CPplan [16,15] in a number of domains, without giving up too much in terms of plan quality.

A natural question that accompanies most SMC approaches is deciding how many samples to use. As we will demonstrate, each planning problem requires a different number of samples in each $\mathcal{M}c$*LUG*. In order to support the ideal of domain-independent planning, we present a technique to automatically determine the number of samples. The automated technique, which relies on existing research [10] on particle filters, outperforms the best manually selected number of particles across many domains.

Our presentation starts by describing the relevant background of CPP and representation within our planner, and then gives a brief primer on SMC for prediction. We follow with a worked example of how to construct planning graphs that exactly compute the probability distribution over possible worlds versus using SMC, as well as how one would symbolically represent planning graph particles. After the intuitive example, we give the details of $\mathcal{M}c$*LUG*, the associated relaxed plan heuristic, and the technique for determining the number of samples. We also address the all important issue of finding an appropriate number of particles to use in each $\mathcal{M}c$*LUG*. Finally, we present an empirical analysis of our techniques by comparing with CPplan, analyzing the effect of using a different number of particles.We finish with a discussion of related work, and conclusions.

2 Background & Representation

In this section we give a brief introduction to planning graph heuristics, and then describe our action and belief state representation, the CPP problem, the semantics of conformant probabilistic plans, and our search algorithm.

Planning Graph Heuristics: Planning graphs have become the foundation for most modern heuristic search planners [4]. A planning graph relaxes the planning problem by ignoring some or all negative interactions between actions. The idea is to hypothesize which actions can be applied to the current search node, then determine what propositions are possible in successor states. We then find which actions can be supported by these propositions. This alternation of possible action and proposition layers continues until we find that all of the goal propositions are possible in a proposition layer. At this point it is possible to reason backwards to find the actions needed to support the goals while ignoring negative interactions. The resulting set of actions is termed a relaxed plan, and the number of actions can be used as a heuristic.

Definition 1 (Conformant Probabilistic Planning Problem) *A Conformant Probabilistic Planning Problem defines the tuple $CPP = \langle P, A, b_I, G, \tau \rangle$, where P is a set of propositions, A is a set of actions, b_I is an initial belief state, G is the goal description (a conjunctive set of propositions), and τ is a goal satisfaction threshold ($0 < \tau \leq 1$).*

Belief States: A state of the world s is a set of propositions $s \subseteq P$, where a proposition holds (does not hold) in s if $p \in s$ ($p \notin s$) . A belief state is a probability distribution over all states (or equivalently, the power set of propositions). The probability of a state s in a belief state b is denoted $b(s)$. We say that a state s is in b ($s \in b$) if $b(s) > 0$. The marginal probability of a set of propositions $P_t \subseteq P$ that hold and a set of propositions $P_f \subseteq P$ ($P_t \cap P_f = \emptyset$) which do not hold, is denoted $b(P_t, P_f)$, and computed as $b(P_t, P_f) = \sum_{s \in b, P_t \subseteq s, P_f \cap s = \emptyset} b(s)$.

Actions: An action a is a tuple $\langle \rho^e(a), \Phi(a) \rangle$, where $\rho^e(a)$ is an enabling precondition, and $\Phi(a)$ is a set of causative outcomes. The enabling precondition $\rho^e(a)$ is a conjunctive set of propositions that determines action applicability. For the ease of discussion our formalism allows only positive propositions in preconditions, but our planner implementation directly allows negative propositions in preconditions and goals. An action a is applicable $appl(a, b)$ to belief state b iff $\forall_{s \in b} \rho^e(a) \subseteq s$. The causative outcomes $\Phi(a)$ are a set of tuples $\langle w_i(a), \Phi_i(a) \rangle$ representing possible outcomes (indexed by i), where $w_i(a)$ is the probability of outcome i being realized, and $\Phi_i(a)$ is a mutually-exclusive and exhaustive set of conditional effects (indexed by j). Each conditional effect $\varphi_{ij}(a) \in \Phi_i(a)$ is of the form $\rho_{ij}(a) \rightarrow \varepsilon_{ij}(a)$, where both the antecedent (secondary precondition) $\rho_{ij}(a)$ and consequent $\varepsilon_{ij}(a)$ are a conjunctive set of propositions. This representation of effects follows the 1ND normal form presented by Rintanen [29]. As outlined in the PPDDL standard [32], for every action we can use $\Phi(a)$ to derive a state transition function $T(s, a, s')$ that defines a probability that executing a in state s will result in state s'. Thus, executing action a in belief state b, denoted $exec(a, b) = b_a$, defines the probability of each state in the successor belief state as $b_a(s') = \sum_{s \in b} b(s)T(s, a, s')$.

Definition 2 (Conformant Probabilistic Plan) *A conformant plan $\mathcal{P}$ is of the form $\mathcal{P} ::= \perp \mid a \mid a; \mathcal{P}$. Executing a plan in belief state b defines successor belief states as follows: $exec(\perp, b) = b$, $exec(a, b) = b_a$, and $exec(a; \mathcal{P}, b) = exec(\mathcal{P}, b_a)$. A plan $(a_0; a_1; ...; a_m)$ is executable with respect to b_I if each action a_i is applicable $appl(a_i, b_i)$ to a belief state b_i, where $b_i = exec(a_0; ...; a_{i-1}, b_0)$ and $b_I = b_0$. If plan $\mathcal{P}$ is executable for b_I, and $b_{\mathcal{P}} = exec(\mathcal{P}, b_I)$, then $\mathcal{P}$ satisfies G with probability $b_{\mathcal{P}}(G, \emptyset)$. If $b_{\mathcal{P}}(G, \emptyset) \geq \tau$, the plan solves the problem.*

Search: We use forward-chaining, weighted A* search to find solutions to CPP. The search graph is organized using nodes to represent belief states, and edges for actions. A solution is a path in the search graph from b_I to a terminal node. We define terminal nodes as belief states where $b(G, \emptyset) \geq \tau$. The g-value of a node is the length of the minimum length path to reach the node from b_I. The f-value of a node is $g(b) + 5h(b)$, using a weight of 5 for the heuristic. [1] In the remainder of the paper we concentrate on the very important issue of how to compute $h(b)$ using an extension of planning graph heuristics to CPP.

3 Sequential Monte Carlo

In many scientific disciplines it is necessary to track the distribution over values of a random variable X over time. This problem can be stated as a first-order stationary Markov process with an initial distribution $Pr(X_0)$ and transition equation $Pr(X_k|X_{k-1})$. It is possible to compute the probability distribution over the values of X after k steps as $Pr(X_k) = \int Pr(X_k|X_{k-1})Pr(X_{k-1})dX_{k-1}$. In general, $Pr(X_k)$ can be very difficult to compute exactly, even when it is a discrete distribution (as in our scenario).

We can approximate $Pr(X_k)$ as a set of N particles $\{x_k^n\}_{n=0}^{N-1}$, where the probability that X_k takes value x_k,

$$Pr(X_k = x_k) \approx \frac{|\{x_k^n | x_k^n = x_k\}|}{N}$$

is the proportion of particles taking on value x_k. At time $k = 0$, the set of particles is drawn from the initial distribution $Pr(X_0)$. At each time step $k > 0$, we simulate each particle from time $k - 1$ by sampling the transition equation $x_k^n \sim Pr(X_k|x_{k-1}^n)$. In our application of SMC to approximate conformant planning graphs, particles represent possible worlds (deterministic planning graphs), where at any time step the value of a particle denotes a specific joint outcome of uncertain events (e.g., an initial state and joint action outcomes for each time step). Our stochastic transition equation resembles the Conformant GraphPlan [30] construction semantics (i.e., modeling the probability of achieving one proposition layer from another, given the applicable actions).

We would like to point out that our SMC technique is inspired by, but different from the standard particle filter. The difference is that we are using SMC for prediction and not on-line filtering. We do not filter observations to weight our particles for re-sampling. Particles are assumed to be unit weight throughout simulation.

[1] Since our heuristic turns out to be inadmissible, the heuristic weight has no further bearing on admissibility. In practice, using five as a heuristic weight tends to improve search performance.

4 Monte Carlo Planning Graph Construction

We start with an example to give the intuition for Monte Carlo simulation in planning graph construction. Consider a simple Logistics domain where we wish to load a specific freight package into a truck and loading works probabilistically (because rain is making things slippery). There are two possible locations where we could pick up the package, but we are unsure of which location. There are three propositions, $P = \{$ atP1, atP2, inP $\}$, and our initial belief state b_I has two states $s0 =$ $\{$atP1$\}$ and $s1 = \{$atP2$\}$ where $b_I(s0) = b_I(s1) = 0.5$, and the goal is $G =$$\{inP\}$. The package is at location 1 (atP1) or location 2 (atP2) with equal probability, and is definitely not in the truck (inP). Our actions are LoadP1 and LoadP2 to load the package at locations 1 and 2, respectively. Both actions have an empty enabling precondition $\{\}$ (so they are always applicable) and have two outcomes. The first outcome, with probability 0.8, loads the package if it is at the location, and the second outcome, with probability 0.2, does nothing. We assume for the purpose of exposition that driving between locations in not necessary. The descriptions of the actions are:

LoadP1 = $\langle\{\}$, $\{\langle 0.8$, $\{$atP1$\rightarrow$inP$\}\rangle$, $\langle 0.2, \{\}\rangle\}\rangle$

LoadP2 = $\langle\{\}$, $\{\langle 0.8$, $\{$atP2$\rightarrow$inP$\}\rangle$, $\langle 0.2, \{\}\rangle\}\rangle$

Each action has two outcomes. The first outcome has a single conditional effect, and the second has no effects (which we denote by "Noop" in Figure 1, 2, and 3).

Figures 1, 2, and 3 illustrate several approaches to planning graph based reachability analysis for our simplified Logistics domain. (We assume we are evaluating the heuristic value $h(b_I)$ of reaching G from our initial belief state.) The first is in the spirit of Conformant GraphPlan, where uncertainty is handled by splitting the planning graph layers for all outcomes of uncertain events. CGP creates a planning graph that resembles a tree, where each branch corresponds to a deterministic planning graph.

CGP: In Figure 1, we see that there are two initial proposition layers (denoted by propositions in boxes), one for each possible world at time zero. We denote the uncertainty in the source belief state by X_0, which takes on values $s0, s1$ (for each state in our belief state). Both load actions are applicable in both possible worlds because their enabling preconditions are always satisfied. The edges leaving the actions denote the probabilistic outcomes (each a set of conditional effects). While it is possible for any outcome of an action to occur, the effects of the outcome may not have their secondary precondition supported. In world $s0$, if outcome Φ_0(LoadP1) occurs, then effect φ_{00}(LoadP1) (denoted by atP1$\rightarrow$inP) is enabled and will occur, however even if Φ_0(LoadP2) occurs φ_{00}(LoadP2) is not enabled and will not occur.

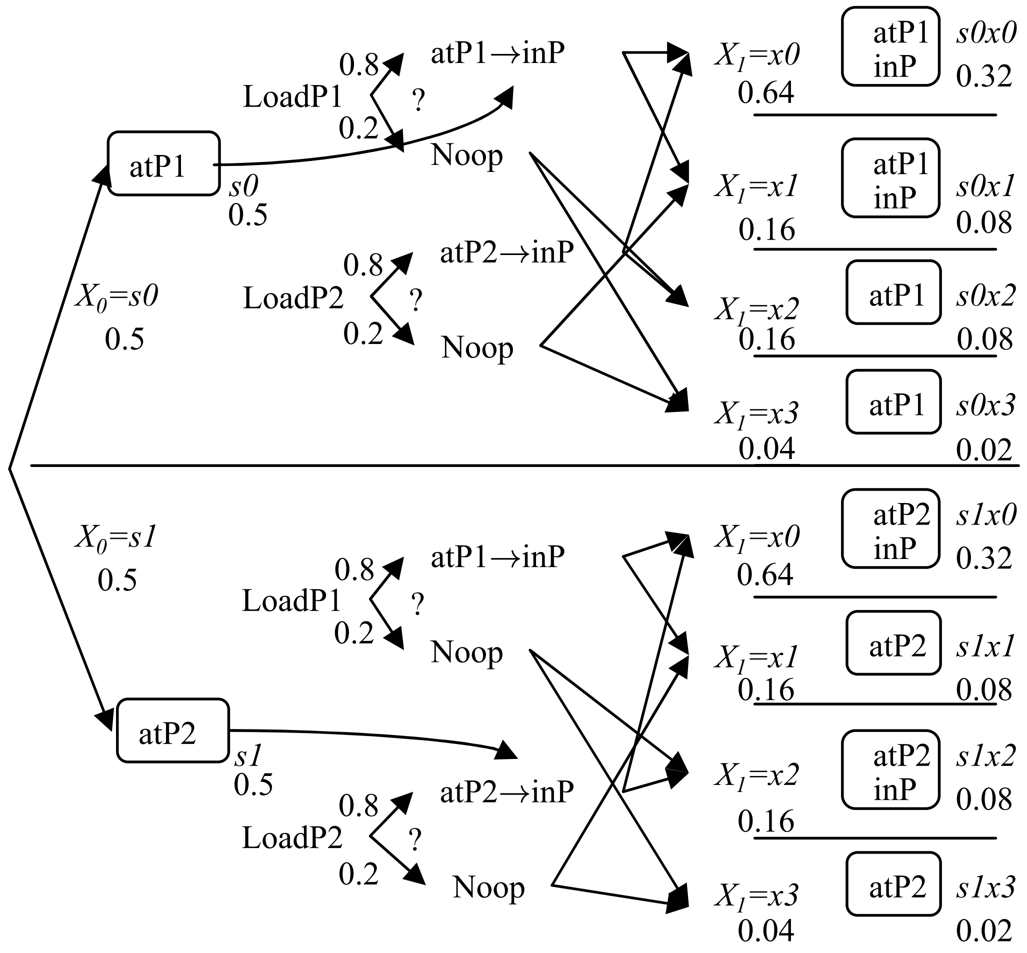


Fig. 1. *CGP representation.*

The set of possible worlds at time one is determined by the cross product of action outcomes in each world at time zero and the possible worlds at time zero. For instance, possible world $s0x0$ is formed from world $s0$ when outcomes $\Phi_0(\text{LoadP1})$ and $\Phi_0(\text{LoadP2})$ co-occur in $s0$. Likewise, world $s1x2$ is formed from world $s1$ when outcomes $\Phi_1(\text{LoadP1})$ and $\Phi_0(\text{LoadP2})$ occur in $s1$. (The edges from outcomes to possible worlds in Figure 1 denote which outcomes are used to form the worlds.)

CGP is exactly representing the reachable proposition layers for all possible worlds. In our example, CGP could determine the exact distribution over possible worlds at time one $Pr(X_0, X_1)$. We see that our goal is satisfied in half of the possible worlds at time 1, with a total probability of 0.8. It is possible to back-chain on this graph like CGP search to extract a relaxed plan (by ignoring mutexes) that satisfies the goal with 0.8 probability. However, we note that this is exactly representing all possible worlds (which can increase exponentially).

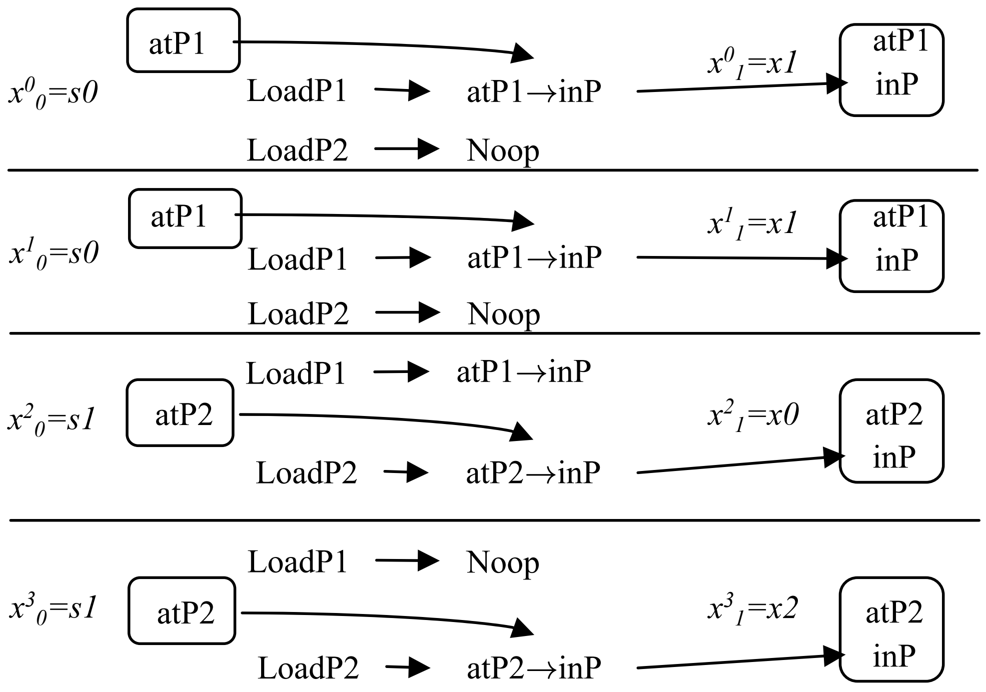


Fig. 2. *McCGP representation.*

McCGP: Next, we illustrate a Monte Carlo simulation approach we call Monte Carlo CGP (McCGP), in Figure 2. The idea is to represent a set of N deterministic planning graphs as particles. In our example, say we sample $N = 4$ states from b_I, denoted $\{x_0^n\}_{n=0}^{N-1} \sim Pr(X_0)$, where $Pr(X_0) = b_I$, and create an initial proposition layer for each. To simulate a particle we first insert the applicable actions. We then insert effects by sampling from the distribution of joint action outcomes (i.e. $x_k^n \sim Pr(X_k|x_{k-1}^n)$). (It is possible to sample the outcome of each action independently because their outcomes are independent.) Finally, we construct the subsequent proposition layer, given the sampled outcomes. Note that each particle is a deterministic planning graph.

In our example, the simulation was lucky and the proposition layer for each particle at time 1 satisfies the goal, so we may think the best one step plan achieves the goal with certainty. From each of these graphs it is possible to extract a relaxed plan, which can then be aggregated to give a heuristic as described by Bryce, et. al. [5].

While McCGP improves memory consumption by bounding the number of possible worlds, it still wastes quite a bit of memory. Many of the proposition layers in the resulting planning graphs are identical. Symbolic techniques can help us compactly represent the set of planning graph particles.

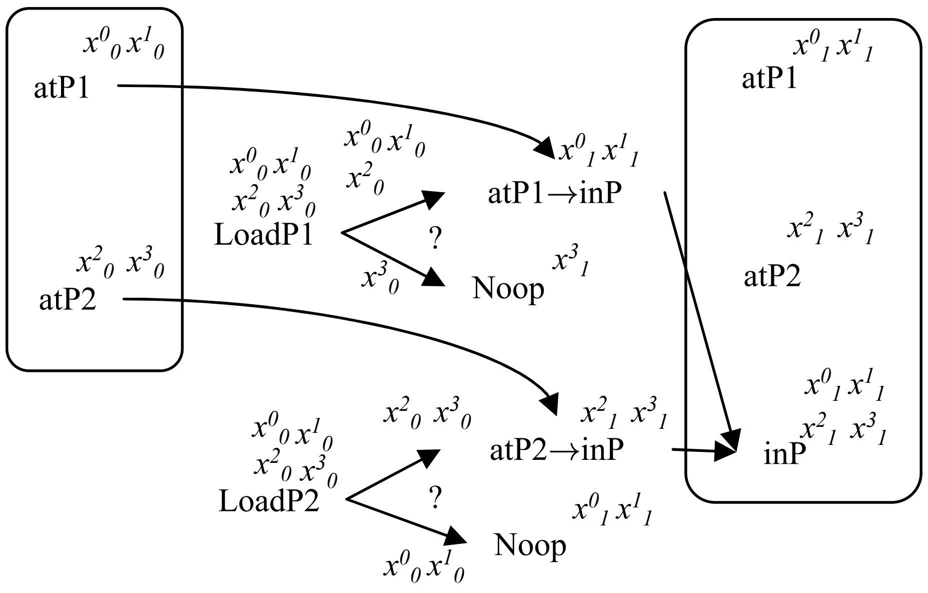


Fig. 3. $\mathcal{M}cLUG$ *representation.*

McLUG**:** Using ideas from Bryce, et. al. [5] , we can represent a single proposition layer at every time step for all particles in a planning graph called the Monte Carlo *LUG* ($\mathcal{M}cLUG$), in Figure 3. We associate a label with each proposition, action, and effect instead of generating multiple copies. The idea is to union the connectivity of multiple planning graphs into a single planning graph skeleton, and use labels on the actions and propositions to signify the original, explicit planning graphs in which an action or proposition belongs. The contribution in the $\mathcal{M}cLUG$ is to represent a set of particles symbolically and provide a relaxed plan extraction procedure that takes advantage of the symbolic representation.

5 Symbolic Planning Graph Representation

Bryce, et. al. [5] describe a planning graph generalization called the Labelled Uncertainty Graph (*LUG*), used in non-deterministic conformant planning, that symbolically represents the exponential number of planning graphs used by Conformant GraphPlan [30]. Bryce et. al. [5] construct multiple planning graphs symbolically by propagating "labels" over a single planning graph skeleton. The skeleton serves to represent the connectivity between actions and propositions in their preconditions and effects. The labels on actions and propositions capture non-determinism by indicating the outcomes of random events that support the actions and propositions. In the problems considered by Bryce et. al. [5] there is only a single random event X_0 captured by labels because the actions are deterministic. Where CGP would build a planning graph for each possible state, the *LUG* is able to use labels to denote which of the explicit planning graphs would contain a given proposition or action in a level. For instance, if CGP built a planning graph for possible worlds $s1, ..., sn$ (each a state in a source belief state) and the planning graphs for $s1, ..., sm$ each had proposition p in level k, then the *LUG* would have p in level k labelled with a propositional formula $\ell_k(p)$ whose models are $\{s1, ..., sm\}$. In the worst case, the random event X_0 captured by the labels has $2^{|P|}$ outcomes (i.e., all states are in the belief state), characterized by a logical formula over $\log_2(2^{|P|})$ = $|P|$ boolean variables.

Bryce et. al. [5] construct the *LUG* until all goal propositions are labelled with all states in the source belief state, meaning the goal is strongly reachable in the relaxed plan space. The authors defined a strong relaxed plan procedure that back-chains on the *LUG* to support the goal propositions in all possible worlds. This relaxed plan proved effective for search in both conformant and conditional non-deterministic planning.

5.1 Exact Symbolic Representation

Despite the utility of the *LUG*, it has a major limitation in that it does not reason with actions that have uncertain effects, an essential feature of probabilistic planning. We would like to complete the analogy between the *LUG* and CGP by symbolically representing uncertain effects. However, as we argue, exactly representing all possible worlds is still too costly even with symbolic techniques.

We previously noted that the *LUG* symbolically represents $Pr(X_0)$ using labels with $|P|$ boolean variables. When we have uncertain actions, the distribution $Pr(X_1)$ requires additional boolean variables. For example, if the action layer contains $|A|$ actions, each with m probabilistic outcomes, then we would require an additional $\log_2(m^{|A|}) = |A|\log_2(m)$ boolean variables (for a total of $|P|$ + $|A|\log_2(m)$ boolean variables to exactly represent the distribution $Pr(X_1)$). For the distribution after k steps, we would need $|P|$ + $k|A|\log_2(m)$ boolean variables. In a reasonable sized domain, where $|P| = 20$, $|A| = 30$, and $m = 2$, a *LUG* with $k = 3$ steps could require 20+(3)30$\log_2(2)$ = 110 boolean variables, and for $k = 5$ it needs 170. Currently, a label function with this many boolean variables is feasible to construct, but is too costly for use in heuristics. We implemented this approach (representing labels as BDDs [31]) and it performed very poorly; in particular it ran out of memory constructing the first planning graph for the p2-2-2 Logistics problem, described in our empirical evaluation.

We could potentially compile all action uncertainty into state uncertainty to alleviate the need for additional label variables. This technique, mentioned in [30], involves making the uncertain outcome of each action conditional on a unique, random, and unknown state variable for each possible time step the action can execute. While this compilation would allow us to restrict the growth of *LUG* labels (to a constant sized, but exponentially larger representation), there is a problem. We are solving indefinite horizon planning problems, meaning that the number of possible time points for an action to execute is unbounded. This further means that the size of the compilation is unbounded. Consequently, we shift our focus to approximating the distribution using particles.

5.2 Symbolic Particle Representation ($\mathcal{M}c$LUG)

We describe how to construct a *$\mathcal{M}c$LUG*, a symbolic version of McCGP that we use to extract relaxed plan heuristics. There are noticeable similarities to the *LUG*, but by using a fixed number of particles we avoid adding boolean variables to the label function at each level of the planning graph. We implement labels as boolean formulas, but find it convenient in this context to describe them as sets of particles (where each particle is in reality a model of a boolean formula). The *$\mathcal{M}c$LUG* is constructed with respect to a belief state encountered in search which we call the source belief state. The algorithm to construct the *$\mathcal{M}c$LUG* starts by forming an initial proposition layer $\mathcal{L}_0$ and an inductive step to generate a graph level $\{\mathcal{A}_k, \mathcal{E}_k, \mathcal{L}_k\}$ consisting of an action, effect, and proposition layer. We describe each part of this procedure in detail, then follow with a description of relaxed plan extraction, and how to select the number of particles.

Initial Proposition Layer: The initial proposition layer is constructed with a set of N particles $\{x_0^n\}_{n=0}^{N-1}$ drawn from the source belief state. Each particle x_0^n corresponds to a state $s \in b$ in the source belief state. (The super-script of a particle denotes its identity, and the sub-script denotes its time index.)

In the example (assuming N=4), the particles map to the states: $x_0^0 = s0, x_0^1 = s0, x_0^2 = s1, x_0^3 = s1$.

The initial proposition layer $\mathcal{L}_0$ is a set of labelled propositions $\mathcal{L}_0 = \{p | \ell_0(p) \neq \emptyset\}$, where each proposition must be labelled with at least one particle. A proposition is labelled $\ell_0(p) = \{x_0^n | p \in s, x_0^n = s\}$ to denote particles that correspond to states where the proposition holds.

In the example, the initial proposition layer is $\mathcal{L}_0 = \{\text{atP1, atP2}\}$, and the labels are:

$$\ell_0(\text{atP1}) = \{x_0^0, x_0^1\}$$
$$\ell_0(\text{atP2}) = \{x_0^2, x_0^3\}$$

Action Layer: The action layer at time k consists of all actions whose enabling precondition is enabled, meaning all of the enabling preconditions hold together in at least one particle. The action layer is defined as all enabled actions $\mathcal{A}_k = \{a|\ell_k(a) \neq \emptyset\}$, where the label of each action is the set of particles where it is enabled $\ell_k(a) = \bigcap_{p \in \rho^e(a)} \ell_k(p)$. When the enabling precondition is empty the label contains all particles.

In the example, the zeroth action layer is $\mathcal{A}_0 = \{\text{LoadP1, LoadP2}\}$, and the labels are:

$$\ell_0(\text{LoadP1}) = \ell_0(\text{LoadP2}) = \{x_0^0, x_0^1, x_0^2, x_0^3\}$$

Both actions are enabled for all particles because their enabling preconditions are empty, thus always enabled.

Effect Layer: The effect layer contains all effects that are labelled with a particle $\mathcal{E}_k = \{\varphi_{ij}(a)|\ \ell_k(\varphi_{ij}(a)) \neq \emptyset\}$. Determining which effects get labelled requires simulating the path of each particle. The path of a particle is simulated by sampling from the distribution over the joint outcomes of all enabled actions, $x_{k+1}^n \sim P(X_{k+1}|x_k^n)$. We sample by first identifying the actions that are applicable for a particle x_k^n. An action is applicable for particle x_k^n if $x_k^n \in \ell_k(a)$. For each applicable action we sample from the distribution of its outcomes. The set of sampled outcomes identifies the path of x_k^n to x_{k+1}^n. We record the path by adding x_{k+1}^n to the labels $\ell_k(\varphi_{ij}(a))$ of applicable effects of sampled outcomes. Note that even though an outcome is sampled for a particle, some of its effects may not be applicable because their antecedents are not supported by the particle (i.e., $x_k^n \notin \bigcap_{p \in \rho_{ij}(a)} \ell_k(p)$).

In the example, we first simulate x_0^0 by sampling the outcomes of all actions applicable in x_0^0, which is both Load actions. Suppose we get outcome 0 for LoadP1 and outcome 1 for LoadP2, which are then labelled with x_1^0. Particle x_0^1 happens to sample the same outcomes as x_0^0, and we treat it similarly. Particle x_0^2 samples outcome 0 of both actions. Note that we do not label the effect of outcome 0 for LoadP1 with x_1^2 because the effect is not enabled in x_0^2. Finally, for particle x_0^3 we sample outcome 1 of LoadP1 and outcome 0 of LoadP2. Thus, the effect layer is $\mathcal{E}_0 = \{\varphi_{00}(\text{LoadP1}), \varphi_{10}(\text{LoadP1}), \varphi_{00}(\text{LoadP2}), \varphi_{10}(\text{LoadP2})\}$, labelled as:

$$\begin{aligned}
\ell_0(\varphi_{00}(\text{LoadP1})) &= \{x_1^0, x_1^1\} \\
\ell_0(\varphi_{10}(\text{LoadP1})) &= \{x_1^3\} \\
\ell_0(\varphi_{00}(\text{LoadP2})) &= \{x_1^2, x_1^3\} \\
\ell_0(\varphi_{10}(\text{LoadP2})) &= \{x_1^0, x_1^1\}
\end{aligned}$$

Proposition Layer: Proposition layer $\mathcal{L}_k$ contains all propositions that are given by an effect in $\mathcal{E}_{k-1}$. Each proposition is labelled by the particles of every effect that give it support. The proposition layer is defined as $\mathcal{L}_k = \{p | \ell_k(p) \neq \emptyset\}$, where the label of a proposition is $\ell_k(p) = \bigcup_{\varphi_{ij}(a) \in \mathcal{E}_{k-1} : p \in \varepsilon_{ij}(a)} \ell_{k-1}(\varphi_{ij}(a))$.

In the example, the level one proposition layer is $\mathcal{L}_1 = \mathcal{L}_0 \cup \{\text{inP}\}$. The propositions are labelled as:

$$\begin{aligned} \ell_1(\text{atP1}) &= \{x_1^0, x_1^1\} \\ \ell_1(\text{atP2}) &= \{x_1^2, x_1^3\} \\ \ell_1(\text{inP}) &= \{x_1^0, x_1^1, x_1^2, x_1^3\} \end{aligned}$$

The propositions from the previous proposition layer $\mathcal{L}_0$ persist through implicit noop actions, allowing them to be labelled as in the previous level – in addition to particles from any new supporters. The inP proposition is supported by two effects, and the union of their particles define the label.

Termination: $\mathcal{M}c$*LUG* construction continues until a proposition layer supports the goal with probability no less than τ. We assess the probability of the goal at level k by finding the set of particles where the goal is supported and taking the ratio of its size with N. Formally,

$$Pr(G|X_k) \approx \frac{|\bigcap_{p \in G} \ell_k(p)|}{N}$$

We also define level off for the $\mathcal{M}c$*LUG* as the condition when every proposition in a proposition layer is labelled with the same number of particles as in the previous level. If level off is reached without $Pr(G|X_k) \geq \tau$, then we set the heuristic value of the source belief state to ∞.

We note that whether we use the estimated probability of goal satisfaction or level off to terminate $\mathcal{M}c$*LUG* expansion, it is possible for the $\mathcal{M}c$*LUG* to continue changing (if we were to further expand). It should be clear that goal satisfaction will increase monotonically as the number of levels grows. It is less obvious that level off is not a sufficient fix point criterion. Since we are sampling action outcomes, it is possible to reach level off without sampling an outcome that will add new particles that label the goal. However, it is possible that expansion after level off will sample the outcome and change the particles that label the goal.

Another issue that affects our relaxed plan heuristic (described next) is the choice of the level to support the goals and/or terminate expansion. We obviously want to support the goals no earlier than the level where the achievement probability is greater or equal to τ. It is not clear which of the later levels to use. In each extra level it is presumably more costly to support the goal, but with a potentially higher probability. This issue reveals the multi-objective (decision-theoretic) nature of CPP. Since the CPP problem itself is not defined as multi-objective, we make the following assumption about desirable plans. *We wish to minimize the cost of a plan that achieves the goal with probability no less than* τ. As such, we terminate $\mathcal{M}c$*LUG* expansion at the first proposition layer that satisfies the goal with probability no less than τ. It should be straight forward to adapt the *$\mathcal{M}c$LUG* to the more general multi-objective setting, however we do not do so here.

5.3 Heuristics

We just defined how to terminate construction of the *$\mathcal{M}c$LUG* at level k, and we can use k as a measure of the number of steps needed to achieve the goal with probability no less than τ. This heuristic is similar to the level heuristic defined for the *LUG* [5]. As has been shown in non-deterministic and classical planning, relaxed plan heuristics are often much more effective, despite being inadmissible. Since we are already approximating the possible world distribution of the planning graph and losing admissibility, we decide to use relaxed plans as our heuristic.

The intuition behind a conformant relaxed plan is to measure both the positive interaction and independence between particles as they co-achieve the goals [5]. Rather than finding a relaxed plan for each particle and taking the maximum or summation of their respective numbers of actions, we attempt to align the relaxed plans to maximize overlap. We count the actions used in common between particles only once to account for positive interaction. Actions not used in common contribute to measuring independence between particles. The total number of actions in the aligned relaxed plan becomes the heuristic. Extraction from the *$\mathcal{M}c$LUG* is very fast because the symbolic representation lets us obtain this conformant relaxed plan for all particles at once, rather than each individually and merging them. Since we know which particles support the goals and which paths the particles took through the *$\mathcal{M}c$LUG*, we can pick actions labelled with these particles to support the goal. Picking actions that support in many particles at once helps maximize positive interaction between particles (which is equivalent to selecting actions that will work in high probability plans).

In our example, the goal inP is labelled with four particles $\{x_1^0, x_1^1, x_1^2, x_1^3\}$. Particles x_1^0, x_1^1 are supported by φ_{00}(LoadP1), and particles x_1^2, x_1^3 are supported by φ_{00}(LoadP2), so we include both LoadP1 and LoadP2 in the relaxed plan. For each action we subgoal on the antecedent of the chosen conditional effect as well as its enabling precondition. By including LoadP1 in the relaxed plan to support particles x_0^0, x_0^1, we have to support atP1 for the particles. We similarly subgoal for the particles supported by LoadP2. Fortunately, we have already reached level 0 and do not need to support the subgoals further. The value of the relaxed plan is two because we use two actions.

Often there are many choices for supporting a subgoal in a set of particles. Consider a subgoal g that must be supported in a set of particles $\{x_k^1, x_k^2, x_k^3\}$ and is supported by effect φ in particles x_k^1 and x_k^2, φ' in particles x_k^2 and x_k^3, and φ'' in

x_k^2. Choosing support in the wrong order may lead us to include more actions than needed, especially if the effects are of different actions. This problem is actually a set cover, which we solve greedily. For example, until the set of particles for g is covered, we select supporting effects based on the number of new particles they cover (except for proposition persistence actions, which we prefer over all others). The number of particles an effect can support is proportional to the probability with which the effect supports the proposition. Say we first pick φ because it covers two new particles, then φ' can cover one new particle, and φ'' covers no new particles. We finish the cover by selecting φ' for particle x_k^3. Notice that even though φ' can support two particles we use it to support one. When we subgoal to support φ' we only support it in particle x_k^3 to avoid "bloating" the relaxed plan.

5.4 Selecting the number of particles N

Up to this point, we have avoided the issue of selecting N, the number of particles to use in each $\mathcal{M}cLUG$. As we will demonstrate in the empirical evaluation, good choices for N are distinct to each problem. In this section we investigate an automated (domain-independent) method to find a good N. Our objective is to find a value for N that is large enough to provide informed heuristics, but small enough to keep heuristic computation cost low.

In order to pick a good N, we must understand how it affects the approximations made by the $\mathcal{M}cLUG$, as well as how the cost of building the $\mathcal{M}cLUG$ interplays with search. There are three factors that we will use to determine N: the number of state samples needed to approximate representative belief states, the estimated search depth, and the estimated search branching factor. In the following, we examine the role of each factor and end by describing a principled way of combining these features to estimate N.

5.4.1 Sample-Based Belief State Approximation

It is well known that classical planning graphs approximate state transition graphs. Similarly, the $\mathcal{M}cLUG$ approximates the belief state transition graph. The types of belief states we intend to approximate with the $\mathcal{M}cLUG$ play a role in selecting N. A natural characterization of these belief states (in our setting) is the number of state samples needed for their approximation. This raises two concerns:

- How do we approximate a belief state?
- Which belief states do we approximate?

The answer to the first concern readily exists in the particle filtering literature. For the second, we use a random walk in the belief state space to find reachable belief states.

Approximating a Belief State: Fox [10] addresses the quality of sample-based approximation for the purpose of dynamically adjusting the number of particles used in a particle filter. Fox presents an algorithm for determining the number of particles required to approximate a multinomial distribution, such as a finite state belief state. The algorithm guarantees with probability $1 - \delta$ that the error between the approximation $\hat{b}$ and the true belief state b is less than ϵ. By measuring error with KL-distance [7], it is possible to derive a value for N as

$$N(b, \epsilon) = \frac{1}{2\epsilon}\chi^2_{k_b-1,1-\delta},$$

where $\chi^2_{k_b-1,1-\delta}$ is the upper $1-\delta$ quantile of the chi-squared distribution with k_b-1 degrees of freedom. The value of k_b is the number of unique states represented in the approximation $\hat{b}$. The term for the number of samples can be approximated by the Wilson-Hilferty transformation [17] to obtain

$$N(b, \epsilon) = \frac{1}{2\epsilon}\chi^2_{k_b-1,1-\delta} \approx \frac{k_b - 1}{2\epsilon}\left\{1 - \frac{2}{9(k_b-1)} + \sqrt{\frac{2}{9(k-1)}}z_{1-\delta}\right\}^3,$$

where $z_{1-\delta}$ is the upper $1 - \delta$ quantile of the normal distribution. Since we have not approximated the belief state yet, we do not know k_b and hence we do not know $N(b, \epsilon)$. Thus, we must iteratively compute $N(b, \epsilon)$ by drawing state samples from the belief state, computing k_b (by counting the number of unique sampled states), and then computing $N(b, \epsilon)$. Once we have sampled the same number of states as our current value of $N(b, \epsilon)$, then we have an $N(b, \epsilon)$ that with probability $1 - \delta$ will approximate the belief state with error less than ϵ.

Finding Belief States to Approximate: With a method to determine the number of particles to approximate a given belief state, we must determine which belief states to approximate. Many problems start with a belief state containing a single state, which could serve as poor indicator of stochastic belief states reached after a few steps. It seems reasonable to consider several belief states and perhaps use the maximum number of particles needed for approximation.

Since the $\mathcal{M}c$*LUG* is approximating all reachable belief states, we need not focus solely on finding belief states reached by a feasible plan. Ideally we would like to characterize the belief states which are going to affect search decisions. However, finding these belief states is difficult without a heuristic to guide the search (the very same heuristic for which we are determining an N). Instead we use a random walk (sampling action choices) in belief space to identify reachable belief states. We determine the length of the random walk by computing a heuristic value $h(b_I)$ of the initial belief state (using a small number of particles). With a set of belief states B (expanded in our random walk), we can compute the number of state samples

$N(b, \epsilon)$ needed to approximate each $b \in B$ with error less than ϵ. In summary, N should be proportional to the max of the number of samples per belief state, $N \propto \max_{b \in B} N(b, \epsilon)$.

5.4.2 Search Depth

The search depth, which can be estimated by $h(b_I)$ (as in the random walk above), indicates how many times search will compute the *$\mathcal{M}cLUG$* heuristic. The number of particles should be inversely proportional to the estimated search depth for two reasons. As stated, a deeper search means more heuristic computation, meaning the heuristic should be less expensive. A more subtle point in favor of fewer particles is that the heuristic re-affirms as search deepens. If particles do a poor job of estimating the merit of a search node (e.g., by using too few), then evaluating each child of the node provides more particles that serve witness to the merit of the parent node. Thus, a deeper search may require fewer particles to offset the cost of the search and search may not be harmed. In summary, $N \propto 1/depth \propto 1/h(b_I)$.

5.4.3 Branching Factor

The search branching factor indicates the potential for making a bad search choice. When faced with more decisions, search should spend more effort during node evaluation. As branching factor increases more particles provide a more informed heuristic for node evaluation, meaning the number of particles is proportional to branching factor. The random walk (above) can easily estimate the average branching factor. In summary, $N \propto breadth$.

5.4.4 Estimating the Number of Particles

We compute the number of particles N by combining the previously mentioned features: the number of samples to approximate belief states, estimated search depth, and the estimated average branching factor. The belief state approximation factor $\max_{b \in B} N(b, \epsilon)$ is generally too large for estimating N. The approximation factor estimates the number of particles needed for a probability distribution over states, whereas the *$\mathcal{M}cLUG$* describes probabilities for propositions. Since the set of states is exponential in the set of propositions, we assume that their approximation factors are related by an exponential factor. Thus we use the logarithm of the approximation factor to determine N for the *$\mathcal{M}cLUG$*. We take the product of this with the estimated average branching factor ($breadth$). Finally, we divide this product by the estimated search depth ($h(b_I)$). This term is then multiplied a scaling factor of ten to get the resulting number of particles used for each *$\mathcal{M}cLUG$*:

$$N = \left\lceil 10 \left(\frac{log(\max_{b \in B} N(b, \epsilon))(breadth)}{h(b_I)} \right) \right\rceil$$

A final consideration for selecting N is the error ϵ we are willing to accept in our approximations. In our empirical evaluation we will identify a good value for ϵ. While we are trading one free parameter (N) for another (ϵ), our intent is to find a *domain-independent* parameter setting.

6 Empirical Analysis: Setup

In this section we describe the setup of the empirical analysis by describing the planners, domains, and testing environment. We externally evaluate our planner $POND$ and its heuristic based on the $\mathcal{M}c$*LUG* by comparing with the leading approach to CPP, CPplan [15,16]. We also internally evaluate our approach by adjusting (both manually and automatically) the number of particles N that we use in each $\mathcal{M}c$*LUG*. We refrain from comparing with POMDP solvers (such as POMDP-solve [6]), as did Hyafil and Bacchus [16], because they were shown to be effective only on problems with very small state spaces (e.g., Slippery Gripper and Sand Castle-67) and we care about problems with large state spaces. Our approach does only slightly better than CPplan on the small state space problems and we doubt we are superior to the POMDP solvers on these problems. Recent work in approximate POMDP solving [28] may make a better comparison, but there are many implementation-level details preventing a thorough comparison at this time.

6.1 Planners

In the following we describe the implementation of our planner $POND$ and the basic principles of the CPplan planner, as well as the way we compare the planners. We choose CPplan for comparison because existing planners that directly solve our problem do not scale nearly as well.

POND: Our planner is implemented in C++ and uses several existing technologies. It employs the PPDDL parser [32] for input, the IPP planning graph construction code [20] for the $\mathcal{M}c$*LUG*, and the CUDD BDD package [31] for representing belief states, actions, and labels. Figure 4 depicts our planner architecture. The two inputs to the planner are the problem specification, and the method to select particle size. The problem is grounded, pre-processed, and compiled into ADDs. If we choose to automatically determine particle size, then we determine the length of our random walk, take the random walk, analyze the random walk, and finally

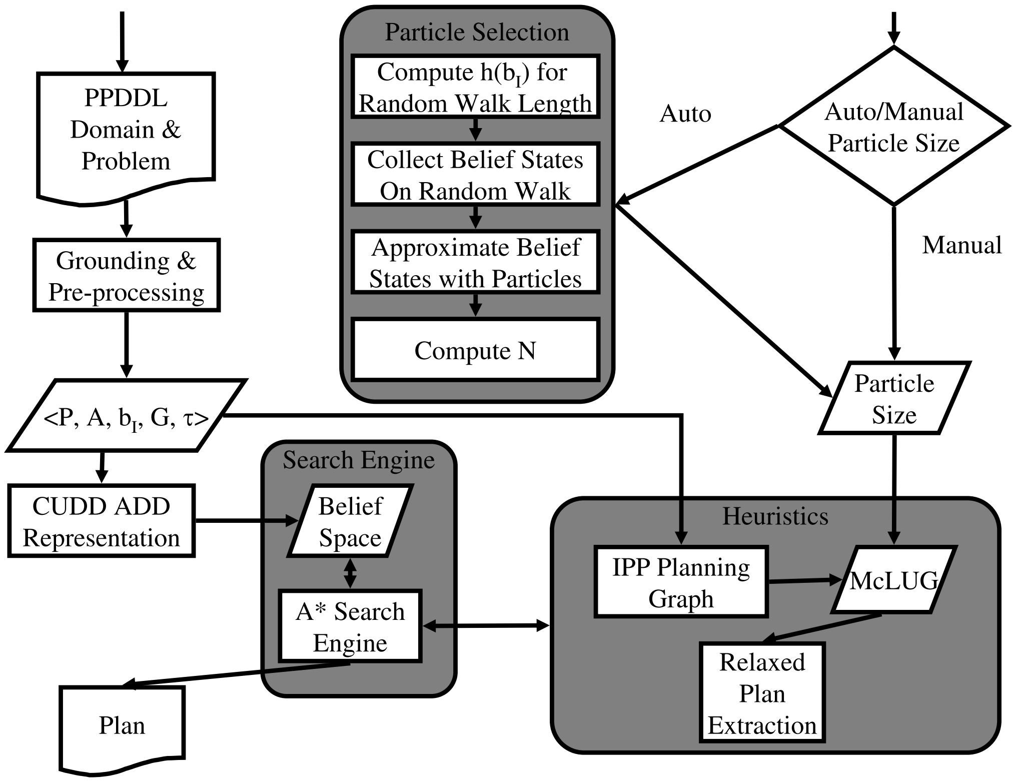


Fig. 4. *$POND$ Architecture.*

compute N. The search commences by expanding search nodes and computing heuristics. Each heuristic computation involves constructing a $\mathcal{M}c$*LUG* with the chosen number of particles until the goal is reached with enough probability. From the $\mathcal{M}c$*LUG*, we extract a relaxed plan whose number of actions is used for the h-value of a search node. Upon finding a plan, search ends and returns the plan.

CPplan: CPplan is an optimal bounded length planner that uses a CSP solver for CPP. Part of the reason CPplan works so well is its efficient caching scheme that re-uses optimal plan suffixes to prune possible solutions. In comparison, our work computes a relaxation of plan suffixes to heuristically rank partial solutions. CPplan finds the optimal probability of goal satisfaction for a given plan length (an NP^{PP}-complete problem, [23]), but $POND$, like Buridan [21], finds plans that satisfy the goal with probability no less than τ (an undecidable problem, [24]). CPplan could be used to find an optimal length plan that exceeds τ by iterating over increasing plan lengths (similar to BlackBox, [19]).

6.2 Domains

In the following we describe four domains used in the empirical evaluation. Within each domain we describe several problems instances and domain variations. The first two domains are considerably more difficult than the last two domains, but all exhibit a difference in scalability between $POND$ and CPplan.

Logistics: The Logistics domain has the standard Logistics actions of un/loading, driving, and flying, but adds uncertainty. Hyafil and Bacchus [16] enriched the domain developed by Hoffmann and Brafman [12] to not only include initial state uncertainty, but also effect uncertainty. In each problem there are some number of packages whose probability of initial location is uniformly distributed over some locations and un/loading is only probabilistically successful. Plans require several loads and unloads for a single package at several locations, making a relatively simple deterministic problem a very difficult stochastic problem. We compare on three problems p2-2-2, p4-2-2, and p2-2-4, where each problem is indexed by the number of possible initial locations for a package, the number of cities, and the number of packages. See [16] for more details.

Grid: The Grid domain, as described by Hyafil and Bacchus [16], is a 10x10 grid where a robot can move one of four directions to adjacent grid points. The robot has imperfect effectors and moves in the intended direction with high probability (0.8), and in one of the two perpendicular directions with a low probability (0.1). As the robot moves, its belief state grows and it becomes difficult to localize itself. However since the grid borders provide a barrier, moves that would put the robot through the barrier leave the robot in its original position. Thus, by bumping the barrier, it is possible for the robot to localize. The goal is to reach the upper corner of the grid. The initial belief state is a single state where the robot is at a known grid point. We test on the most difficult instance where the robot starts in the lower opposite corner. We also discuss instances of the domain where the grid is different sizes (5x5 or 15x15) or the transitions are more stochastic (the probability of intended moves becomes 0.5 instead of 0.8).

Slippery Gripper: Slippery Gripper is a well known problem that was originally presented by Kushmerick, et. al. [21]. There are four probabilistic actions that clean and dry a gripper and paint and pick-up a block. The goal is to paint the block and hold it with a clean gripper. Many of the lower values of τ require very short plans and take very little run time, so we focus on the higher values of τ where we see interesting scaling behavior.

Sand Castle-67: Sand Castle-67 is another well known probabilistic planning problem, presented by Majercik and Littman [25]. The task is to build a sand castle with high probability by using two actions: erect-castle and dig-moat. Having a moat improves the probability of successfully erecting a castle, but erecting a castle may destroy the moat. Again, scaling behavior is interesting when τ is high.

6.3 Environment

In our test setup, we used a 2.66 GHz P4 Linux machine with 1GB of memory, with a timeout of 20 minutes for each problem. To compare with CPplan, we run CPplan on a problem for each plan length until it exceeds our time or memory limit. We record the probability that CPplan satisfies the goal for each plan length. We then give $POND$ a series of problems with increasing values for τ (which match the values found by CPplan). If $POND$ can solve the problem for all values of τ solved by CPplan, then we increase τ by fixed increments thereafter. We ran $POND$ five times on each problem and present the average run time and plan length.

Comparing the planners in this fashion allows us to measure the plan lengths found by $POND$ to the optimal plan lengths found by CPplan for the same value of τ. Our planner often finds plans that exceed τ (sometimes quite a bit) and includes more actions, whereas CPplan meets τ with the optimal number of actions. Nevertheless, we feel the comparison is fair and illustrates the pros/cons of an optimal planner with respect to a heuristic planner.

7 Empirical Analysis: External Evaluation & Particle Set Size

In this section we evaluate our approach by first externally comparing with CPplan, then internally adjusting the number of particles used in each $\mathcal{M}cLUG$. The internal study uses both manual and automatic particle sizes. With manual particle selection we seek to identify useful ranges of particles sizes for each problem to evaluate the automated particle sizes. Within the automated approach we characterize the effect of adjusting our approximation error ϵ in order to find good values of N.

7.1 Comparison with CPplan

We compare with CPlan on each of the domains mentioned in the previous section using a version of $POND$ where N = 16. As we will see later, 16 is not necessarily the best value for N across all problems, but it does allow us to show the difference in scalability between CPplan and $POND$. We note that CPplan performs marginally worse than previously reported because our machine has one third the memory of the machine Hyafil and Bacchus [16] used for their experiments. The major limitation on CPplan scalability is memory consumption.

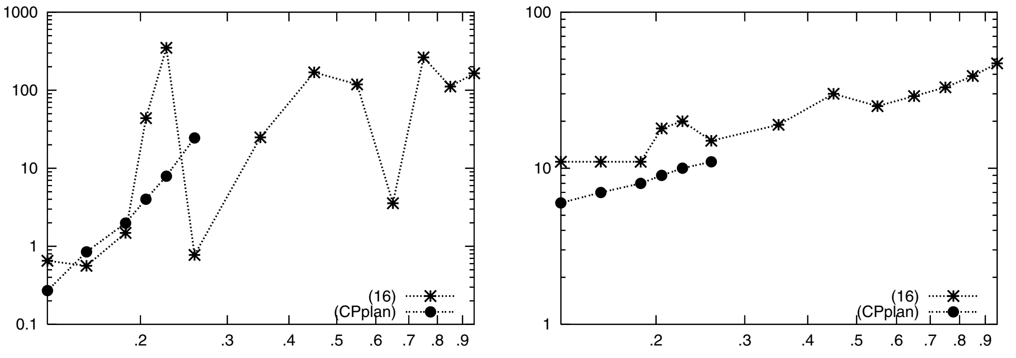


Fig. 5. *Run times (s) and Plan lengths vs. τ (log scale) for Logistics p2-2-2*

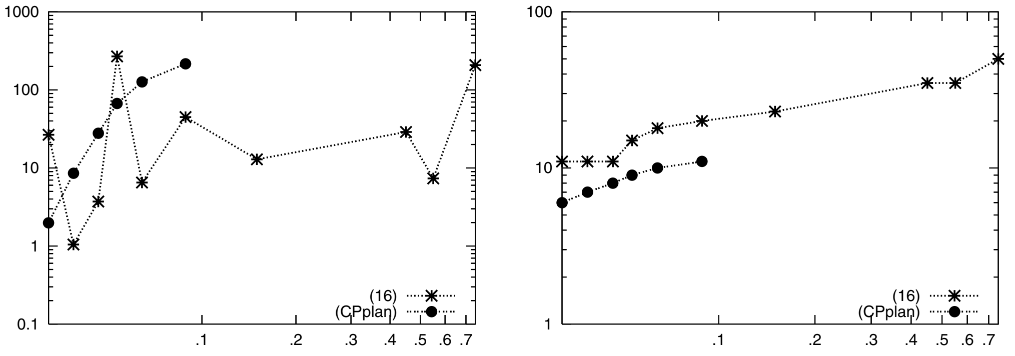


Fig. 6. *Run times (s) and Plan lengths vs. τ (log scale) for Logistics p4-2-2*

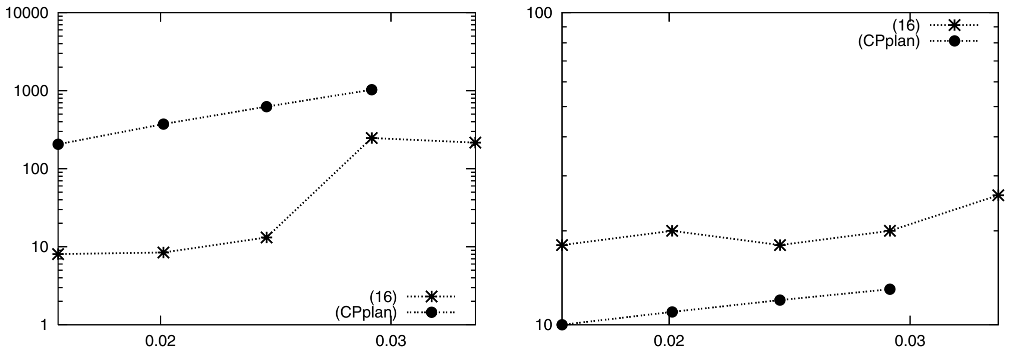


Fig. 7. *Run times (s) and Plan lengths vs. τ (log scale) for Logistics p2-2-4*

Logistics: The plots in Figures 5, 6, and 7 compare the total run time in seconds (left) and the plan lengths (right) of $POND$ with 16 particles in the $\mathcal{M}cLUG$ versus CPplan. In this domain we also use helpful actions from the relaxed plan [13]. We notice that CPplan is able to at best find solutions where $\tau \leq 0.26$ in p2-2-2, $\tau \leq 0.09$ in p4-2-2, and $\tau \leq 0.03$ in p2-2-4. In most cases $POND$ is able to find plans much faster than CPplan for the problems they both solve. It is more interesting that $POND$ is able to solve problems for *much larger* values of τ. With 16 particles in each $\mathcal{M}cLUG$, $POND$ finds solutions where $\tau \leq 0.95$ in p2-2-2, $\tau \leq 0.75$ in p4-2-2, and $\tau \leq 0.035$ in p2-2-4, which is respectively 3.7, 8.3, 1.2 times the maximum values of τ solved by CPplan. As we will see later, we can solve for much larger values of τ by using different numbers of particles. In terms

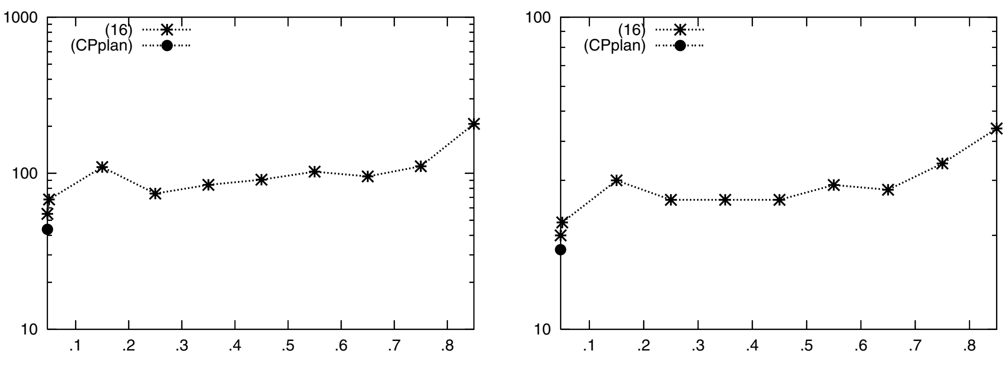


Fig. 8. *Run times (s) and Plan lengths vs. τ for Grid-0.8*

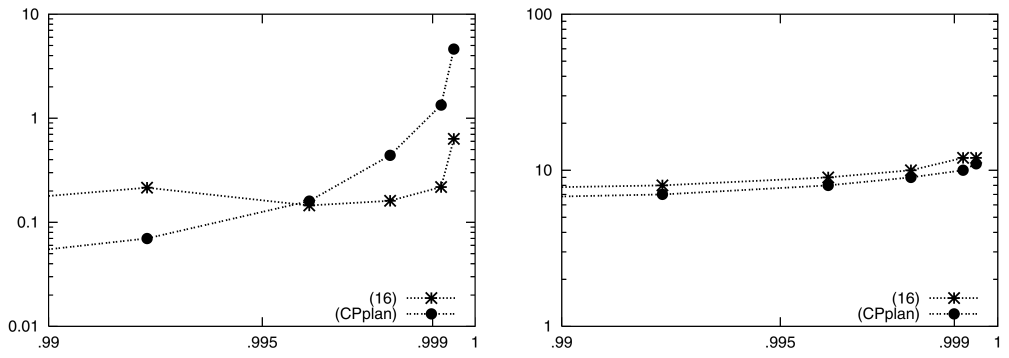
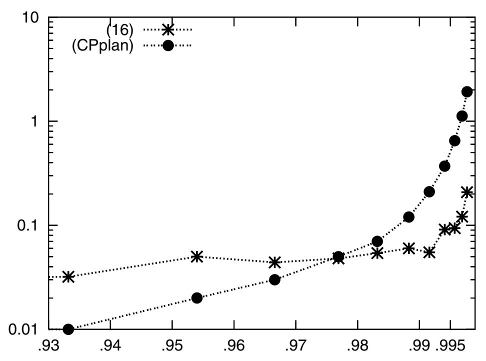


Fig. 9. *Run times (s), and Plan lengths vs. τ for Slippery Gripper.*

of plan quality, the average increase in plan length for the problems we both solved was 5.83 actions in p2-2-2 (43% longer), 5.83 actions in p4-2-2 (46% longer), and 7.5 actions in p2-2-4 (42% longer). The plot of plan lengths gives some intuition for why CPplan has trouble finding plans for greater values of τ. The plan lengths for the larger values of τ approach 40-50 actions and CPplan is limited to plans of around 10-15 actions.

Grid: Figure 8 shows total run times and plan lengths for the 10x10 Grid problem. We notice that CPplan can solve the problem for only the smallest value of τ, whereas $POND$ scales much better. For the single problem we both solve, we found solution with 6 more actions (26% longer).

Slippery Gripper: Figure 9 shows the total time and plan length results for Slippery Gripper. For short plans, CPplan is faster because the $\mathcal{M}cLUG$ has some additional overhead, but as τ increases and plans have to be longer the $\mathcal{M}cLUG$ proves useful. Using 16 particles, we are able to find solutions faster than CPplan in the problems where $\tau > .995$. In terms of plan quality, our solutions include on average 1.6 more actions (20% longer).



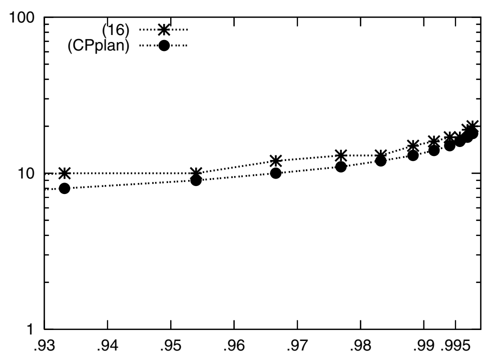


Fig. 10. *Run times (s), and Plan lengths vs. τ for Sand Castle-67.*

Sand Castle-67: The plots for run time and plan length in Figure 10 show that the run time for CPplan has an exponential growth with τ, whereas our method scales roughly linearly. As τ increases, we are eventually able to outperform CP-plan. In terms of plan quality, our plans included an average of 0.88 more actions (7% longer).

Discussion: In comparison with CPplan, the major difference with our heuristic approach is the way that plan suffixes are evaluated. CPplan must exactly compute plan suffixes to prune solutions, whereas we estimate plan suffixes. As plans become longer, it is more difficult for CPplan to exactly evaluate plan suffixes because there are so many and they are large.

Overall, our method is very effective in the CPP problems we evaluated, with the only drawback being longer plans in some cases. To compensate, we believe it should be reasonably straight-forward to post-process our plans to cut cost by removing actions. Nevertheless, it is a valuable lesson to see the size of problems that we can solve (in very little time) by relaxing our grip on finding optimal plans.

7.2 Particle Set Size

In this section we further analyze the effect of N on planner performance. We present results for both the manual and automatic selection of N. In the manual approach, we pick values of N between 4 and 512 (increasing exponentially). In the automated approach, we use 0.1, 0.01, 0.005 for ϵ (the KL-distance approximation error). The heuristic that determines the length of the random walk uses $N = 4$ in the $\mathcal{M}cLUG$. The probability of approximation error δ fixed at 0.01.

We show plots of the time to find solutions with varying values of τ and N in every domain. The height of each point (denoted by a vertical line) indicates the total time for the test. The planar orientation of the point indicates the values of τ and N. Each plot shows the results for manual particle selection as black points connected by lines, where each point is the average of 5 runs for the same value of τ and N. The automated particle selection results are shown as colored points that are not connected by lines (depicted in the legends by the value of ϵ). Each instance solved by the automated method can use a different number of particles, so we do not average over the automated runs. We show both automated and manual particle selection results in one plot to identify the base-line performance expected for fixed values of N and how the automated selection performs by picking varying values of N. We discuss results for each problem in detail and conclude with an analysis of the best value for ϵ compared with the best manual value of N for each problem.

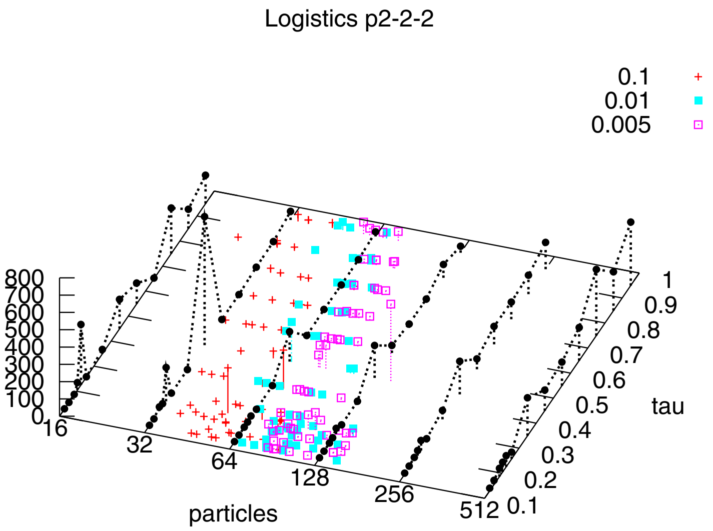


Fig. 11. *Run times (s) vs. τ vs. N for Logistics p2-2-2*

Logistics: Figures 11, 12, and 13 show the time to solve instances in the Logistics domain. In p2-2-2, we are able to scale well with any number of particles, but see that planning time increases when we use too few or too many particles. Without the right number of particles the heuristic is either too weak or too costly.

In p4-2-2 we need more particles than p2-2-2 to perform well (16 particles does not scale). This is due to starting with a more stochastic initial belief state (there are more packages we are uncertain about). It is interesting to notice that manual particle selection can only find plans for large τ when N is large, yet the automatic particle selection finds plans when N is much smaller. This is an artifact of using a stochastic heuristic, rather than finding a special number of particles that works well. We see the opposite behavior in other problems, where the same number of particles may or may not solve the same instances.

In p2-2-4, we again see that too few or too many particles harms performance. Compared with the p2-2-2 and p4-2-2, using more particles here is also helpful. Even though p4-2-2 and p2-2-4 have the same number of possible initial states, p2-

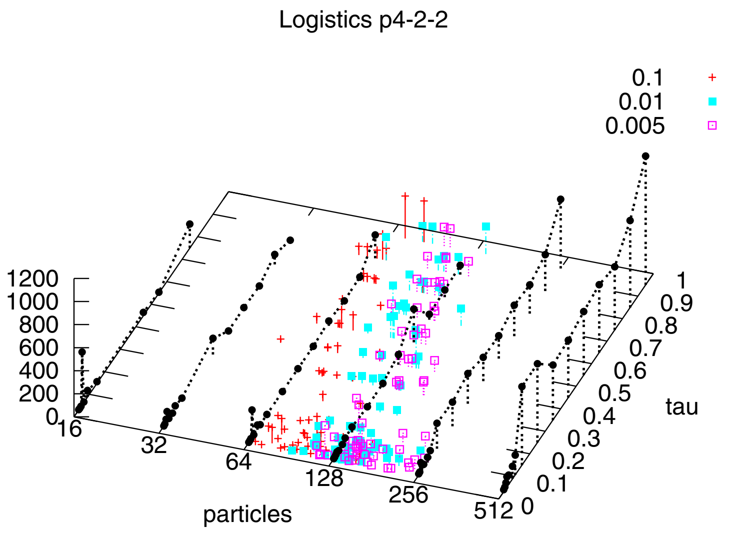


Fig. 12. *Run times (s) vs. τ vs. N for Logistics p4-2-2*

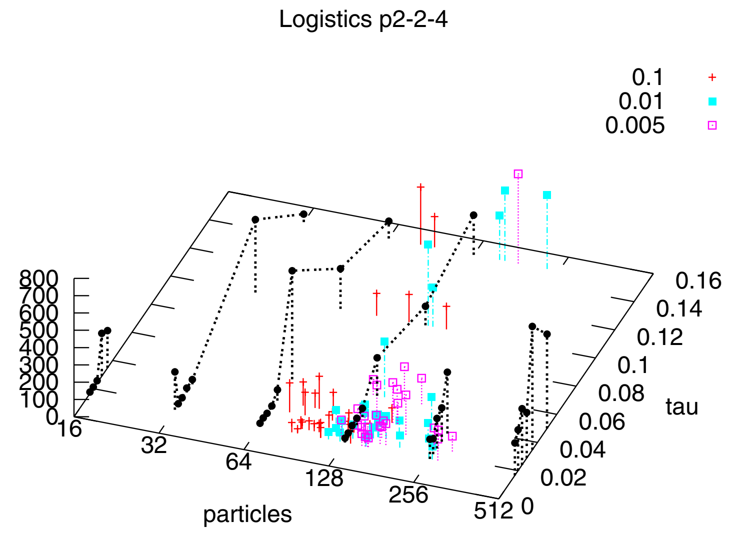


Fig. 13. *Run times (s) vs. τ vs. N for Logistics p2-2-4*

2-4 has more actions (which are stochastic) and requires longer plans for the same values of τ.

Table 1 summarizes the average time in seconds ($\overline{T}$) and number of solutions (S) results for manual particle selection in the Logistics problems, where each problem had (#) instances. With N = 64, $POND$ solves the most instances in the least amount of time for all problems. This is much better than the results for N = 16, which we used to compare with CPplan. The automated particle selection does best with $\epsilon = 0.1$ and $\epsilon = 0.01$. With $\epsilon = 0.005$, the number of particles grows too large.

Problem	#	N = 16 $\overline{T}$	S	N = 32 $\overline{T}$	S	N = 64 $\overline{T}$	S	N = 128 $\overline{T}$	S
Logistics p2-2-2	65	96.36	59	82.73	60	20.16	61	35.26	36
Logistics p4-2-2	80	78.04	35	31.91	42	44.82	49	51.60	47
Logistics p2-2-4	40	98.55	24	107.74	27	136.31	31	92.26	29

Table 1
Summary of results for manual N in Logistics domains, where # is the total number of instances, $\overline{T}$ is the average solution time (s) and S is the number of solved instances.

The automated selection typically selects values for N between 64 and 256. As we will see later, the automated particle selection is able to outperform the manual selection in some problems, despite using more than 64 particles. We will discuss the automated particle selection results summary (in Table 4) in more detail later.

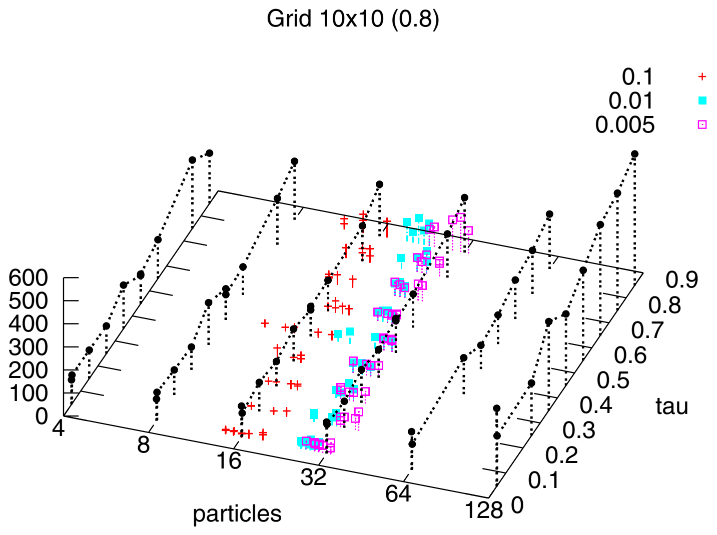


Fig. 14. *Run times (s) vs. τ vs. N for Grid 10x10 with 0.8 correct transitions.*

Grid: We use four versions of the Grid domain in this analysis to characterize how differences in the length of plans and uncertainty in action effects changes performance with the number of particles. We use the 10x10 Grid problem from our previous analysis as the base domain and extend it in two orthogonal directions. First, we keep the 10x10 Grid but change the probability of moving in the intended direction to 0.5 from 0.8 to get belief states that are less peaked. Second, we change the size of the Grid to 5x5 and 15x15 (while keeping 0.8 for transitions as in the base domain).

Figure 14 depicts results for the base domain. As the number of particles falls too low or grows too large total time increases and is variable across values of τ, similar to Logistics. We note however that we are able to do well with much fewer particles

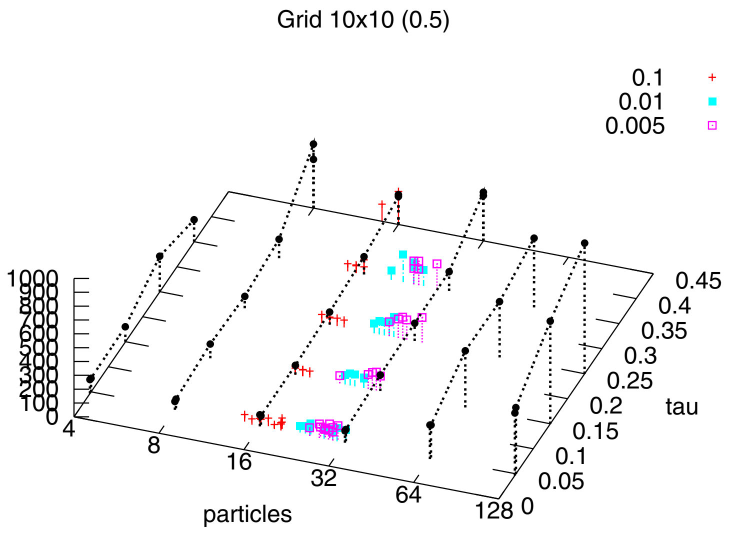


Fig. 15. *Run times (s) vs. τ vs. N for Grid 10x10 with 0.5 correct transitions.*

(around 16-32) than in Logistics (around 64-128). This difference between Grid and Logistics is due potentially to the difference in branching factor, which is 4 in Grid and much higher in Logistics. As we will see in the other versions of the Grid domain, effective particle sizes were not very different when we changed size and uncertainty.

Figure 15 depicts results for making transitions work with probability 0.5 instead of 0.8. It is much more difficult to reach the goal with high probability in this version. Every attempt to increase goal probability will also decrease goal probability quite a bit. In order to transition some probability mass to goal states, the same action will transition probability mass away from goal states. It is unclear if it is impossible to reach the goal with high probability, or if the heuristic is poor. Where in the base domain it was possible to perform well with just about any number of particles, it is apparent that too few particles is insufficient in this version. This is because more particles are needed to capture the significantly "flat" belief state distributions induced by the increased uncertainty. Despite this difference, the automatic particle selection chooses approximately the same number of particles for this domain and the base domain. This may be due to an interplay between the number of particles needed to approximate belief states and the estimated depth of the search tree. While belief states need more particles for approximation, the plans are typically longer.

Figure 16 shows results for the 5x5 Grid domain. $POND$ needs very few particles to do well in this domain because the plans are relatively short. This has two implications, first belief states do not become very flat because they contain relatively fewer states, and second the automated particle selection is confused. Using only four particles to estimate the length of the random walk to compute N is likely to

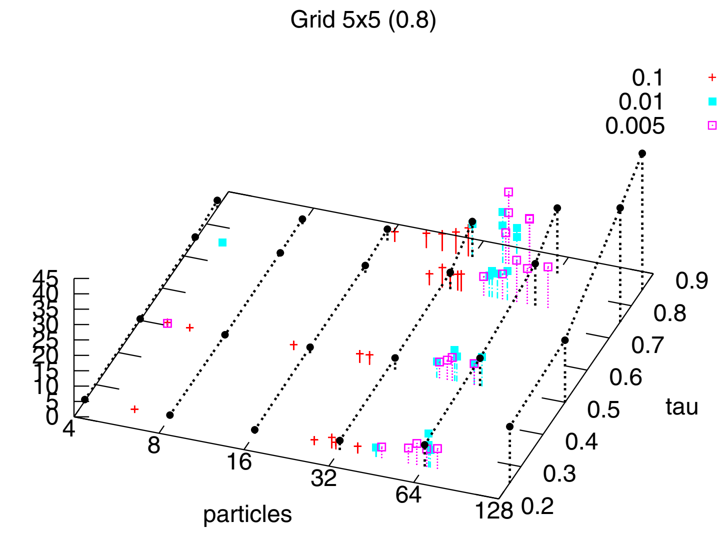


Fig. 16. *Run times (s) vs. τ vs. N for Grid 5x5 with 0.8 correct transitions.*

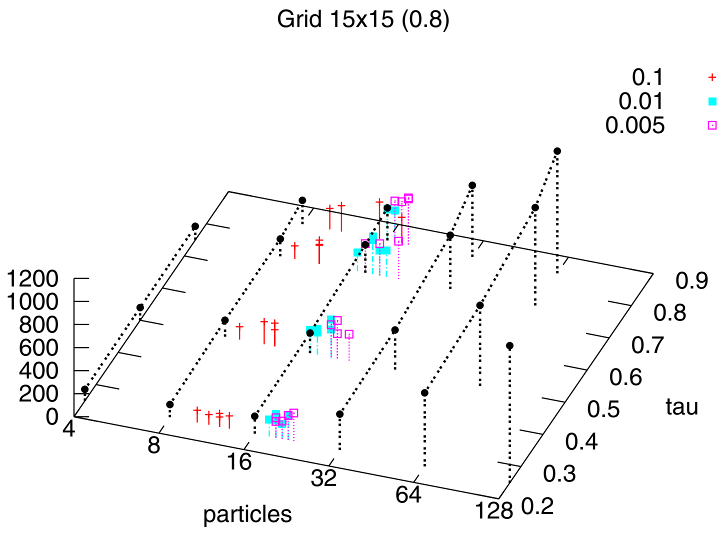


Fig. 17. *Run times (s) vs. τ vs. N for Grid 15x15 with 0.8 correct transitions.*

underestimate the search depth (and hence overestimate N) because there are less planning graph levels where low probability outcomes are sampled (making the relaxed plan smaller).

Figure 17 shows results for the 15x15 Grid domain. Like the domain with 0.5 probability transitions, using too few particles leads to problems. Due to longer plans (not more uncertainty) it is possible to have a flat belief state. While the automated particle selection has the potential to underestimate N due to longer plans, the random walk is able to identify very stochastic belief states and compensate through the sample-based approximation factor. It seems it is lucky and able to compensate just enough because the automated N is in fact much smaller in this version of the domain than the other versions.

		$N = 4$		$N = 8$		$N = 16$		$N = 32$	
Problem	#	$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S
Grid 10x10 (0.8)	50	159.36	50	142.79	50	136.67	50	150.06	49
Grid 10x10 (0.5)	30	144.81	25	152.45	30	106.35	30	151.42	30
Grid 5x5 (0.8)	20	1.07	20	1.03	20	2.31	20	5.95	20
Grid 15x15 (0.8)	20	108.90	10	152.28	12	212.53	18	435.04	20

Table 2
Summary for results for manual N in Grid domains, where # is the total number of instances, $\overline{T}$ is the average solution time (s) and S is the number of solved instances.

Table 2 summarizes the results for manual particle selection in the four versions of the Grid domain. The best performer in most versions is 16 particles, with the only exception of 8 particles in the 5x5 Grid. The reason more particles are needed in larger grids is that belief states can get potentially very flat over longer plans. The automated particle selection is able to perform comparably to the best manual particle selection.

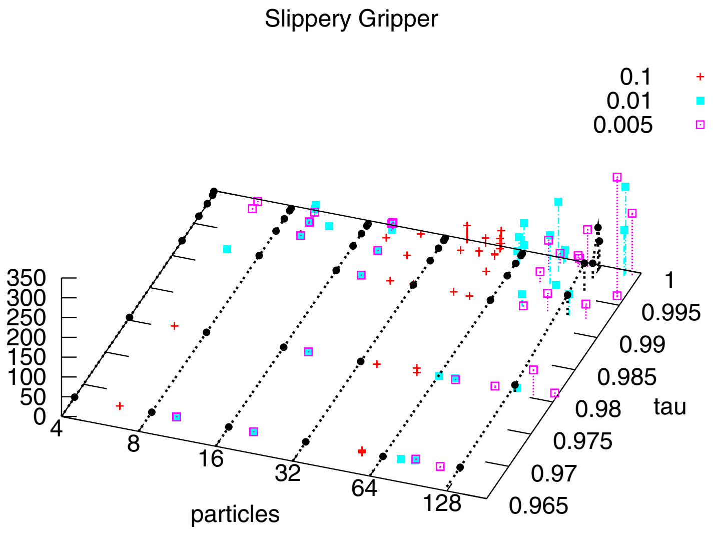


Fig. 18. *Run times (s) vs. τ vs. N for Slippery Gripper.*

Slippery Gripper: Figure 18 shows results for the slippery gripper problem. The results indicate few particles are needed, and extra particles just increases planning time. Using only four particles may be perhaps too few because time does start to increase only slightly for high values of τ. The automated particle selection performs reasonably well on average, with a few instances where it selects larger

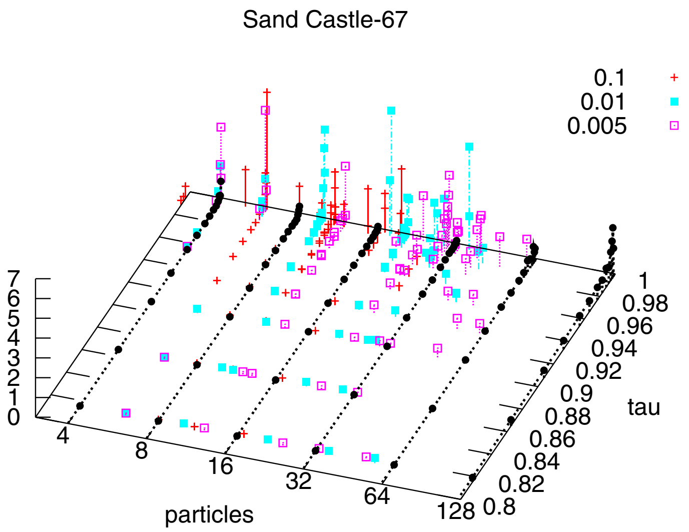


Fig. 19. *Run times (s) vs. τ vs. N for Sand Castle-67.*

values of N when $\epsilon \leq 0.01$. This may be due to similar reasons described in the 5x5 Grid domain. With very few particles estimating the length of the random walk, it is possible to select a very short random walk and not see enough stochastic belief states to offset the value of N. Table 3 lists a summary of results for the manual particle selection that show using 16 particles does best overall.

		N = 4		N = 8		N = 16		N = 32	
Problem	#	$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S
Slippery Gripper	50	1.21	50	0.31	50	0.18	50	0.70	50
Sand Castle-67	85	0.13	85	0.08	85	0.06	85	0.11	85

Table 3
Summary for results for manual N in the Slippery Gripper and Sand Castle-67 domains, where # is the total number of instances, $\overline{T}$ is the average solution time (s) and S is the number of solved instances.

Sand Castle-67: Figure 19 shows results for the Sand Castle-67 problem. Again, choosing the right number of particles is important, using 4-8 or 64-128 is insufficient. The automated particle selection is able to find good values for N in this domain. There are relatively few states and plans are short, so the play between the sample-based approximation and estimated search depth is less sensitive. Table 3 shows a summary of results for manual particle selection that identifies 16 particles as the best choice for this domain.

Discussion: Selecting the right number of particles for a domain is not easy. Several factors such as uncertainty, search depth, and search branching factor affect the number of particles needed. While uncertainty and search depth are important, it

		ϵ = 0.1		ϵ = 0.01		ϵ = 0.005		Best N		
Problem	#	$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S	N
Logistics p2-2-2	65	19.79	59	18.97	60	32.95	62	20.16	61	64
Logistics p4-2-2	80	59.14	67	79.56	72	78.54	65	44.82	49	64
Logistics p2-2-4	40	96.48	29	137.69	29	125.56	28	136.31	31	64
Grid 10x10 (0.8)	50	29.49	50	48.41	50	52.98	50	136.67	50	16
Grid 10x10 (0.5)	30	63.70	27	89.09	25	110.16	25	106.35	30	16
Grid 5x5 (0.8)	20	3.79	20	8.25	20	9.49	20	1.03	20	8
Grid 15x15 (0.8)	20	162.27	17	225.16	18	272.15	16	212.53	18	16
Slippery Gripper	50	6.57	50	21.82	50	21.38	50	0.18	50	16
Sand Castle-67	85	0.55	85	0.74	85	0.72	85	0.06	85	16
Totals	450	**441.78**	**404**	629.68	409	703.94	401	**658.11**	**394**	

Table 4

Summary of results for ϵ compared with best manual N, where # is the total number of instances, $\overline{T}$ is the average solution time (s) and S is the number of solved instances.

seems that the branching factor affects which number of particles works best. The Logistics domain, with the highest branching factor, required the most particles. The other domains, with relatively low branching factors, sufficed with much fewer particles. Within each domain, variations on uncertainty and search depth affected which number of particles is needed.

While automatically estimating and combining these factors is not easy, our automated particle selection technique did well across all problems. Table 4 shows a summary of the results for the automated particle selection with different values of ϵ along with the best manual N for each problem. Of the three values of ϵ that were tried, 0.1 performed best, solving the most problems in the least time. By inspecting the individual problems, there were instances where the automated selection performed better than the best manual selection (e.g., Logistics p4-2-2 and Grid 10x10 (0.8)). The automated particle selection never performs much worse than the best manual particle selection. As a result, the automated selection solves 404 of 450 instances, where the best manual selection solves 394. The total average time for the automated selection is 441.78 seconds, compared to 658.11 seconds for the best manual selection.

Table 5 shows the average number of particles chosen by the best automated method ($\epsilon = 0.1$) compared with the manual method. In most cases, the number of selected particles is very close.

Problem	$\epsilon = 0.1$	Best N
Logistics p2-2-2	50.83	64
Logistics p4-2-2	89.73	64
Logistics p2-2-4	104.93	64
Grid 10x10 (0.8)	15.92	16
Grid 10x10 (0.5)	16.44	16
Grid 5x5 (0.8)	24.30	8
Grid 15x15 (0.8)	11.65	16
Slippery Gripper	35.66	16
Sand Castle-67	11.45	16

Table 5
Summary of the average automated N found by $\epsilon = 0.1$ and the best manual N.

8 Related Work

Buridan [21] was one of the first planners to solve CPP. Buridan is a partial order casual link (POCL) planner that allows multiple supporters for an open condition, much like our relaxed plans in the $\mathcal{M}cLUG$. Unfortunately, Buridan does not scale very well because it lacks effective heuristics. Probapop [27], which is built on top of Vhpop [33], extends Buridan by using heuristics. Probapop uses the classical planning graph heuristics implemented by Vhpop by translating every outcome of probabilistic actions to a deterministic action. In theory, POCL planners are a nice framework for probabilistic planning because it is easy to add actions to support a low probability condition without backtracking (as may be necessary in state based search). In practice, POCL planners can be hard to work with because it is often difficult to assess the probability of a partially ordered plan. At the time of publication we have not made extensive comparisons with Probapop, except on the Grid problem where it cannot find a solution.

Recently, two approaches [8,14] to CPP have also improved scalability considerably. Similar to our approach, Domshlak and Hoffmann [8] use a planning graph heuristic approach to CPP that computes relaxed plans in Probabilistic FF (PFF). PFF computes relaxed plans through a Bayesian network that resembles a planning graph. While current implementation prevents comparison on domains with probabilistic effects, the theory allows for random variables in the planning graph to denote the outcomes of probabilistic effects. This is very similar to our approach that extends the *LUG* to directly handle probabilistic effects through extended labels (which did not scale). It is unclear whether the alternative relaxations (e.g., ignoring all but one effect condition of every action) will offset the cost of considering all outcomes of uncertain actions.

Huang [14] improves approaches to optimal bounded length CPP problems in the Complan planner. Similar to our approach, Complan computes a heuristic estimate of the plan suffix, but unlike us, the heuristic gives an admissible over-estimate of the probability of goal satisfaction for a finite number of actions. Complan improves upon CPplan by removing the heavy memory requirements due to storing plan suffixes. While we do not provide extensive empirical comparisons with Complan, the results presented in [14] show that Complan takes on the order of hours to find solutions for larger instances of the 10x10 Grid problem, where we take minutes. We expect similar behavior on other domains.

Partially observable Markov decision process (POMDP) algorithms, such as [6] to name one, are also able to solve CPP. The work on CPplan [15,16] makes extensive comparisons with the mentioned POMDP algorithm and shows it is inferior for solving CPP problems with large state spaces (like Logistics and Grid). This disparity may be partly due to the fact that the POMDP algorithms solve a slightly different problem by finding plans for all possible initial belief states. CPplan also compares with MaxPlan [25], showing that it too is inferior for several problems. MaxPlan is similar to CPplan, in that it encodes CPP as a bounded length planning problem using a variant of satisfiability. The main difference is in the way they cache optimal plan suffixes used for pruning.

The Prottle planner [22] uses a variation of temporal planning graphs for fully-observable probabilistic temporal planning. In their planning graph they explicitly reason about actions with probabilistic actions by adding an outcome layer and defining a cost propagation procedure. The authors do not extract relaxed plans, nor reason about possible worlds.

PGraphPlan [3] and CGP [30] are two planners that use generalizations of GraphPlan [2] for planning under uncertainty. PGraphPlan and is used for fully-observable probabilistic planning (similar to Markov decision processes). The key idea in PGraphPlan is to forward chain in the planning graph, using dynamic programming, to find an optimal probabilistic plan for a given finite horizon. Alternatively, TGraphPlan greedily back-chains in the planning graph to find a solution that satisfies the goal, without guaranteeing optimality. CGP solves non-observable (conformant) non-deterministic planning problems.

RTDP [1] is a popular search algorithm, used in many recent works (e.g., Mausam and Weld [26]), that also uses Monte Carlo. RTDP samples a single plan suffix to evaluate, whereas we estimate the plan suffix with a relaxed plan. Because we are reasoning about non-observable problems we sample several suffixes and aggregate them to reflect that we are planning in belief space.

9 Conclusion & Future Work

We have presented an approach called $\mathcal{M}cLUG$ to integrate Monte Carlo simulation into heuristic computation on planning graphs. The $\mathcal{M}cLUG$ enables us to quickly compute effective heuristics for conformant probabilistic planning. By using the heuristics, our planner is able to far out-scale the current best approach to conformant probabilistic planning. At a broader level, our work shows one fruitful way of exploiting the recent success in deterministic planning to scale stochastic planners.

The $\mathcal{M}cLUG$ suggests a general technique for handling uncertain actions in planning graphs. A potential application of the $\mathcal{M}cLUG$ is in planning with uncertainty about continuous quantities (e.g., the resource usage of an action). In such cases, actions can have an infinite number of outcomes. Explicitly keeping track of possible worlds is out of the question, but sampling could be useful in reachability heuristics.

We have also presented a domain-independent technique for automatically determining the number of particles to use in the $\mathcal{M}cLUG$. The technique demonstrates a successful integration of existing methods in particle filtering with planning. In the future, we hope to incorporate additional such approximation techniques to further scale planning in stochastic environments. We intend to understand how we can more fully integrate MC into heuristic computation, as there are numerous possibilities for relaxation through randomization. One possibility is to sample the actions to place in the planning graph to simulate splitting the planning graph [34]. More importantly, we would like to use knowledge gained through search to refine our sampling distributions for importance sampling. For instance, we may be able to bias sampling of mutexes by learning the actions that are critical to the planning task. Overall, randomization has played an important role in search [1,11], and we have presented only a glimpse of its benefit in heuristic computation.

Acknowledgements: This work was supported by the NSF grant IIS-0308139, the ONR Grant N000140610058, the ARCS foundation, Honeywell Labs, and an IBM Faculty Award. We would like to thank the members of the Yochan planning group, William Cushing, and David Musliner for helpful suggestions.

References

[1] A. G. Barto, S. Bradtke, S. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1995) 81–138.

[2] A. Blum, M. Furst, Fast planning through planning graph analysis, in: Proceedings of IJCAI'95, 1995, pp. 1636–1642.

[3] A. Blum, J. Langford, Probabilistic planning in the graphplan framework, in: Proceedings of ECP'99, 1999, pp. 319–322.

[4] D. Bryce, S. Kambhampati, How to skin a planning graph for fun and profit (a tutorial on planning graph based reachability heuristics), Tech. rep., ASU CSE TR-06-007 (2006).

[5] D. Bryce, S. Kambhampati, D. Smith, Planning graph heuristics for belief space search, Journal of AI Research 26 (2006) 35–99.

[6] A. Cassandra, M. Littman, N. Zhang, Incremental pruning: A simple, fast, exact method for partially observable markov decision processes, in: Proceedings of UAI'97, 1997, pp. 54–61.

[7] T. Cover, J. Thomas, Elements of information theory, Wiley-Interscience, New York, NY, USA, 1991.

[8] C. Domshlak, J. Hoffmann, Fast probabilistic planning through weighted model counting, in: Proceedings of ICAPS'06, 2006, pp. 243–251.

[9] A. Doucet, N. de Freitas, N. Gordon, Sequential Monte Carlo Methods in Practice, Springer, New York, New York, 2001.

[10] D. Fox, Adapting the sample size in particle filters through kld-sampling, International Journal of Robotics Research 22.

[11] A. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs in lpg, Journal of Artificial Intelligence Research 20 (2003) 239–290.

[12] J. Hoffmann, R. Brafman, Conformant planning via heuristic forward search: A new approach, in: Proceedings of ICAPS'04, 2004, pp. 355–364.

[13] J. Hoffmann, B. Nebel, The FF planning system: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.

[14] J. Huang, Combining knowledge compilation and search for efficient conformant probabilistic planning, in: Proceedings of ICAPS'06, 2006, pp. 253–262.

[15] N. Hyafil, F. Bacchus, Conformant probabilistic planning via CSPs, in: Proceedings of ICAPS' 03, 2003, pp. 205–214.

[16] N. Hyafil, F. Bacchus, Utilizing structured representations and CSPs in conformant probabilistic planning, in: Proceedings of ECAI'04, 2004, pp. 1033–1034.

[17] N. Johnson, S. Kotz, N. Balakrishnan, Continuous Univariate Distributions, John Wiley and Sons, New York, 1994.

[18] S. Kambhampati, L. Ihrig, B. Srivastava, A candidate set based analysis of subgoal interactions in conjunctive goal planning, in: Proceedings of AIPS'96, 1996, pp. 123–133.

[19] H. Kautz, D. McAllester, B. Selman, Encoding plans in propositional logic, in: Proceedings of KR'96, 1996, pp. 374–384.

[20] J. Koehler, B. Nebel, J. Hoffmann, Y. Dimopoulos, Extending planning graphs to an adl subset, in: Proceedings of ECP'97, 1997, pp. 273–285.

[21] N. Kushmerick, S. Hanks, D. Weld, An algorithm for probabilistic least-commitment planning, in: Proceedings of AAAI'94, 1994, pp. 1073–1078.

[22] I. Little, D. Aberdeen, S. Theibaux, Prottle: A probabilistic temporal planner, in: Proceedings of AAAI'05, 2005, pp. 1181–1186.

[23] M. Littman, J. Goldsmith, M. Mundhenk, The computational complexity of probabilistic planning, Journal of Artificial Intelligence Research 9 (1998) 1–36.

[24] O. Madani, S. Hanks, A. Condon, On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems, in: Proceedings of AAAI'99, 1999, pp. 541–548.

[25] S. Majercik, M. Littman, MAXPLAN: A new approach to probabilistic planning, in: Proceedings of AIPS'98, 1998, pp. 86–93.

[26] Mausam, D. Weld, Concurrent probabilistic temporal planning, in: Proceedings of ICAPS'05, 2005, pp. 120–129.

[27] N. Onder, G. Whelan, L. Li, Engineering a conformant probabilistic planner, Journal of Artificial Intelligence Research 25 (2006) 1–15.

[28] P. Poupart, C. Boutilier, VDCBPI: an approximate scalable algorithm for large scale pomdps, in: Proceedings of NIPS'04, 2004.

[29] J. Rintanen, Expressive equivalence of formalisms for planning with sensing, in: Proceedings of ICAPS'03, 2003, pp. 185–194.

[30] D. Smith, D. Weld, Conformant graphplan, in: Proceedings of AAAI'98, 1998, pp. 889–896.

[31] F. Somenzi, CUDD: CU Decision Diagram Package Release 2.3.0, University of Colorado at Boulder (1998).

[32] H. Younes, M. Littman, PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects, Tech. rep., CMU-CS-04-167, Carnegie Mellon University (2004).

[33] H. Younes, R. Simmons, Vhpop: Versatile heuristic partial order planner, Journal of Artificial Intelligence Research 20 (2003) 405–430.

[34] Y. Zemali, P. Fabiani, Search space splitting in order to compute admissible heuristics in planning, in: Workshop on Planen, Scheduling und Konfigurieren, Entwerfen, 2003.