# Sequential Monte Carlo in Probabilistic Planning Reachability Heuristics

**Daniel Bryce & Subbarao Kambhampati**
Department of Computer Science and Engineering
Arizona State University, Brickyard Suite 501
699 South Mill Avenue, Tempe, AZ 85281
{dan.bryce, rao}@asu.edu

**David E. Smith**
NASA Ames Research Center
Intelligent Systems Division, MS 269-2
Moffett Field, CA 94035-1000
de2smith@email.arc.nasa.gov

**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 straight-forward 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 to approximate them. 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.

## Introduction

Despite long standing interest (Kushmerick, Hanks, & Weld 1994; Majercik & Littman 1998; Hyafil & Bacchus 2003; 2004), 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 to solve conformant probabilistic planning (CPP) problems. We use work on planning graph heuristics for non-deterministic planning (Bryce, Kambhampati, & Smith 2006; Hoffmann & Brafman 2004) as our starting point.

We investigate an extension of the work by Bryce, Kambhampati, & Smith (2006) 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, Smith & Weld, 1998), where each is associated with a possible world. While the *LUG* (as described by, Bryce, Kambhampati, & Smith, 2006) 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 such a case, each time step has a set of planning graph layers (each a possible world) defined by the cross product of an exponential set of joint action outcomes and an exponential number of possible worlds from the previous level.

Copyright © 2006, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

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 reachable 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) (Doucet, de Freita, & Gordon 2001) 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 (simulated) deterministic planning graph and the probabilistic transition function resembles the Conformant GraphPlan (Smith & Weld 1998) construction semantics. 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 (Hyafil & Bacchus 2004; 2003) in a number of domains, without giving up too much in terms of plan quality.

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* and the associated relaxed plan heuristic. Finally, we present an empirical analysis of our technique compared to CPplan, and finish with a discussion of related work, and conclusions.

## 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 (Kambhampati 2003). 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 literals are possible in successor states. We then find which actions can be supported by these literals. This alternation of possible action and literal layers continues until we find that all of the goal literals are possible in a literal 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.

**Belief States:** A state of the world is an exhaustive assignment of truth values to every fluent $f \in F$. Fluent truth values $f = l$ and $f = \neg l$ are termed fluent literals (or just literals). Thus, a state is also a set of literals. A belief state is a joint probability distribution over all fluents. The probability of a set of literals, for instance a state $s$, in belief state $b$ is the marginal probability of the literals, denoted $b(s)$. A state is said to be in a belief state b ($s \in b$) if $b(s) > 0$.

**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 literals that determines action applicability. 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 $\langle w(a, i), \Phi(a, i) \rangle$ representing possible outcomes (indexed by $i$), where $\Phi(a, i)$ is a set of several conditional effects (indexed by $j$), and $w(a, i)$ is the probability that outcome $i$ is realized. Each conditional effect $\varphi(a, i, j)$ in outcome $\Phi(a, i)$ is of the form $\rho(a, i, j) \rightarrow \varepsilon(a, i, j)$, where both the antecedent (secondary precondition) $\rho(a, i, j)$ and consequent $\varepsilon(a, i, j)$ are a conjunctive set of literals. This representation of effects follows the 1ND normal form presented by Rintanen (2003). As outlined in the PPDDL standard (Younes & Littman 2004), for every action we can use $\Phi(a)$ to derive a state transition function $T(s, a, s')$ that defines the 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 1 (Conformant Probabilistic Planning).** *The Conformant Probabilistic Planning problem is the tuple* $CPP = \langle F, A, b_I, G, \tau \rangle$*, where* $F$ *is a set of boolean fluents,* $A$ *is a set of actions,* $b_I$ *is an initial belief state,* $G$ *is the goal description (a conjunctive set of literals), and* $\tau$ *is a goal satisfaction threshold (*$0 < \tau \leq 1$*).*

**Definition 2 (Conformant Probabilistic Plan).** *A conformant plan* $P$ *is of the form* $P ::= \perp | a | a; P$*. Executing a plan in belief state* $b$ *defines successor belief states as follows:* $exec(\perp, b) = b$*,* $exec(a, b) = b_a$*, and* $exec(a; P, b) = exec(P, b_a)$*. A plan* $(a_0; a_1; ...; a_n)$ *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* $P$ *is executable for* $b_I$*, and* $b_P = exec(P, b_I)$*, then* $P$ *satisfies* $G$ *with probability* $b_P(G)$*. If* $b_P(G) \geq \tau$*, 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. A belief state $b$ is a terminal node if $b(G) \geq \tau$. The g-value of a node is the length of the minimum cost 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 computing $h(b)$ using an extension of planning graph heuristics.

## 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 $P(X_0)$ and transition equation $P(X_k|X_{k-1})$. It is possible to compute the probability distribution over the values of $X$ after $k$ steps as $P(X_k) = \int P(X_k|X_{k-1})P(X_{k-1})dX_{k-1}$. In general, $P(X_k)$ can be very difficult to compute exactly, even when it is a discrete distribution (as in our scenario).

We can approximate $P(X_k)$ as a set of $N$ samples $\{x_k^n\}_{n=0}^{N-1}$, 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 samples is drawn from the initial distribution $P(X_0)$. At each time step $k > 0$, we simulate each particle from time $k - 1$ by sampling the transition equation

[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.

$x_k^n \sim P(X_k|x_{k-1}^n)$. In our application of SMC to planning graphs, samples represent possible worlds and our transition equation resembles the Conformant GraphPlan (Smith & Weld 1998) construction semantics.

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.

## 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 fluents, $F = \{$ atP1, atP2, inP $\}$, our initial belief state $b_I$ is 0.5: $s0 = \{$atP1, ¬atP2, ¬inP $\}$, 0.5: $s1 = \{$¬atP1, atP2, ¬inP $\}$, 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 = ⟨{}, {⟨0.8, {atP1→inP}⟩, ⟨0.2, {}⟩}⟩
LoadP2 = ⟨{}, {⟨0.8, {atP2→inP}⟩, ⟨0.2, {}⟩}⟩

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).

Figure 1 illustrates 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 1a, we see that there are two initial literal layers (denoted by literals 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 $\Phi(\text{LoadP1}, 0)$ occurs, then effect $\varphi(\text{LoadP1}, 0, 0)$ (denoted by atP1→inP)

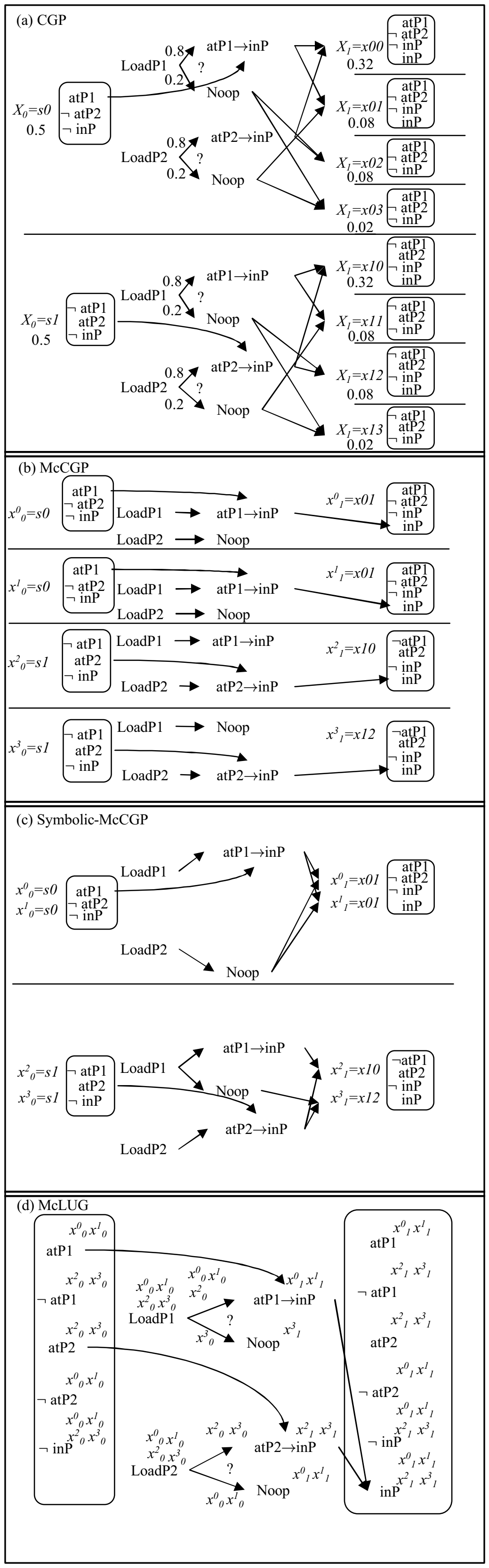

Figure 1: *Variations on planning graph representations.*

is enabled and will occur, however even if $\Phi(\text{LoadP2}, 0)$ occurs $\varphi(\text{LoadP2}, 0, 0)$ is not enabled and will not occur.

The set of possible worlds at time one is determined by the cross product of action outcomes in each world at time zero. For instance, possible world $x00$ is formed from world $s0$ when outcomes $\Phi(\text{LoadP1}, 0)$ and $\Phi(\text{LoadP2}, 0)$ co-occur. Likewise, world $x12$ is formed from world $s1$ when outcomes $\Phi(\text{LoadP1}, 1)$ and $\Phi(\text{LoadP2}, 0)$ occur. (The edges from outcomes to possible worlds in Figure 1a denote which outcomes are used to form the worlds.)

CGP is exactly representing the reachable literal layers for all possible worlds. In our example, CGP could determine the exact distribution over $X_1$ for every value of $X_0$. 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).

**McCGP:** Next, we illustrate a Monte Carlo simulation approach we call Monte Carlo CGP (McCGP), in Figure 1b. The idea is to represent a set of $N$ planning graph particles. In our example we sample $N = 4$ states from $b_I$, denoted $\{x_0^n\}_{n=0}^{N-1} \sim P(X_0)$, where $P(X_0) = b_I$, and create an initial literal 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. (It is possible to sample the outcome of each action independently because their outcomes are independent.) Finally, we construct the subsequent literal layer, given the sampled outcomes. Note that each particle is a deterministic planning graph.

In our example, the simulation was lucky and the literal 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, Kambhampati, & Smith (2006).

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

**Symbolic-McCGP:** To see the intuition for symbolic representation of planning graphs and why it is useful for our Monte Carlo techniques, consider our third example, symbolic-McCGP, in Figure 1c. In McCGP our sampling gave us two copies of the initial literal layers for each initial literal layer in CGP. We can capture the same notion of samples by representing the unique literal layers once and associating a label with each. The label signifies which samples use the literal layer. By labelling entire literal layers we are improving our use of memory, but we can do better.

***McLUG*:** Using ideas from Bryce, Kambhampati, & Smith (2006) , we can represent a single literal layer at every time step for all samples in a planning graph called the Monte Carlo *LUG* ($\mathcal{M}cLUG$), in Figure 1d. By analogy with the symbolic-McCGP planning graph, we associate a label with each literal instead of each literal layer. The idea is to union the connectivity of multiple planning graphs into a single planning graph skeleton, and use labels on the actions and literals to signify the original, explicit planning graphs in which an action or literal 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.

## Symbolic Representation

Bryce, Kambhampati, & Smith (2006) 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 (Smith & Weld 1998). Bryce, Kambhampati, & Smith (2006) construct multiple planning graphs symbolically by propagating "labels" over a single planning graph skeleton. The skeleton serves to represent the connectivity between actions and literals in their preconditions and effects. The labels on actions and literals capture non-determinism by indicating the outcomes of random events that support the actions and literals. In the problems considered by Bryce, Kambhampati, & Smith (2006) 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 literal 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 literal $p$ 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 $2^{|F|}$ outcomes (i.e., all states are in the belief state), characterized by a logical formula over $\log_2(2^{|F|}) = |F|$ boolean variables.

Bryce, Kambhampati, & Smith (2006) construct the *LUG* until all goal literals 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 literals in all possible worlds. This relaxed plan proved effective for search in both conformant and conditional non-deterministic planning.

### 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 $P(X_0)$ using labels with $|F|$ boolean variables. When we have uncertain actions, the distribution $P(X_1)$ 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 $|F|$ + $|A|\log_2(m)$ boolean variables to exactly represent the distribution $P(X_1)$). For the distribution after $k$ steps, we would need $|F|$ + $k|A|\log_2(m)$ boolean variables. In a reasonable sized domain, where $|F| = 20$, $|A| = 30$, and $m = 2$, a *LUG* with $k = 3$ steps could require 20+(3)30log$_2$(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, Somenzi, 1998) 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 the next section.

We could potentially compile all action uncertainty into state uncertainty to alleviate the need for additional label variables. This technique, mentioned in (Smith & Weld 1998), 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.

## Symbolic Particle Representation ($\mathcal{M}cLUG$)

We describe how to construct a $\mathcal{M}cLUG$, 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 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 $\mathcal{M}cLUG$ is constructed with respect to a belief state encountered in search which we call the source belief state. The algorithm to construct the $\mathcal{M}cLUG$ starts by forming an initial literal 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 literal layer. We describe each part of this procedure in detail and follow with a description of relaxed plan extraction.

**Initial Literal Layer:** The initial literal 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 samples map to the states: $x_0^0 = s0, x_0^1 = s0, x_0^2 = s1, x_0^3 = s1$.

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

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

$$\begin{aligned}\ell_0(\text{atP1}) = \ell_0(\neg\text{atP2}) &= \{x_0^0, x_0^1\}\\ \ell_0(\neg\text{atP1}) = \ell_0(\text{atP2}) &= \{x_0^2, x_0^3\}\\ \ell_0(\neg\text{inP}) &= \{x_0^0, x_0^1, x_0^2, x_0^3\}\end{aligned}$$

**Action Layer:** The action layer at time $k$ consists of all actions whose enabling precondition is enabled, meaning all of the enabling precondition literals 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 $\ell_k(a) = \bigcap_{l \in \rho^e(a)} \ell_k(l)$. When the enabling precondition is empty the label contains all particles.

In the example, the zeroth action layer is $\mathcal{A}_0 = \{$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 $\mathcal{E}_k = \{\varphi(a,i,j) | \ell_k(\varphi(a,i,j)) \neq \emptyset\}$. 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 $\ell_k(\varphi(a,i,j))$ 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. $x_k^n \notin \bigcap_{l \in \rho(a,i,j)} \ell_k(l)$).

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 $\mathcal{E}_0 = \{\varphi(\text{LoadP1}, 0, 0), \varphi(\text{LoadP1}, 1, 0), \varphi(\text{LoadP2}, 0, 0), \varphi(\text{LoadP2}, 1, 0)\}$, labelled as:

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

**Literal Layer:** Literal layer $\mathcal{L}_k$ contains all literals that are given by an effect in $\mathcal{E}_{k-1}$. Each literal is labelled by the particles of every effect that gives it support. The literal layer is defined as $\mathcal{L}_k = \{l | \ell_k(l) \neq \emptyset\}$, where the label of a literal is $\ell_k(l) = \bigcup_{l \in \varepsilon(a,i,j), \varphi(a,i,j) \in \mathcal{E}_{k-1}} \ell_{k-1}(\varphi(a,i,j))$.

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

$$\ell_1(\text{atP1}) = \ell_1(\neg\text{atP2}) = \{x_1^0, x_1^1\}$$
$$\ell_1(\neg\text{atP1}) = \ell_1(\text{atP2}) = \{x_1^2, x_1^3\}$$
$$\ell_1(\text{inP}) = \ell_1(\neg\text{inP}) = \{x_1^0, x_1^1, x_1^2, x_1^3\}$$

The literals from the previous literal 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 literal is supported by two effects, and the union of their particles define the label.

**Termination:** $\mathcal{M}cLUG$ construction continues until a literal layer supports the goal with probability no less than $\tau$. 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,

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

We also define level off for the $\mathcal{M}cLUG$ as the condition when every literal in a literal 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 $\infty$.

## Heuristics

We just defined how to terminate construction of the $\mathcal{M}cLUG$ 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 $\tau$. This heuristic is similar to the level heuristic defined for the *LUG* (Bryce, Kambhampati, & Smith 2006). 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. Our relaxed plan extraction is almost identical to the relaxed plan extraction in the *LUG*. The extraction is very fast because it makes use of the symbolic representation to obtain a relaxed plan for all particles at once, rather than each individually and aggregating them. The intuition behind the relaxed plan is that we know which particles support the goals and which paths the particles took through the $\mathcal{M}cLUG$, so we can pick actions, labelled with these particles, that support the goal.

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 $\varphi(\text{LoadP1}, 0, 0)$, and particles $x_1^2, x_1^3$ are supported by $\varphi(\text{LoadP2}, 0, 0)$, 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 $\varphi$ in particles $x_k^1$ and $x_k^2$, $\varphi'$ in particles $x_k^2$ and $x_k^3$, and $\varphi''$ 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 literal 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 literal. Say we first pick $\varphi$ because it covers two new particles, then $\varphi'$ can cover one new particle, and $\varphi''$ covers no new particles. We finish the cover by selecting $\varphi'$ for particle $x_k^3$. Notice that even though $\varphi'$ can support two particles we use it to support one. When we subgoal to support $\varphi'$ we only support it in particle $x_k^3$ to avoid "bloating" the relaxed plan.

## Empirical Analysis

We externally evaluate our planner and its heuristic based on the $\mathcal{M}cLUG$ by comparing with the leading approach to CPP, CPplan (Hyafil & Bacchus 2003; 2004). We also internally evaluate our approach by adjusting the number of particles $N$ that we use in each $\mathcal{M}cLUG$. We refrain from comparing with POMDP solvers, as did (Hyafil & Bacchus 2004), because they were shown to be effective only on problems with very small state spaces (e.g., slippery gripper and sandcastle-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.

Our planner is implemented in C and uses several existing technologies. It employs the PPDDL parser (Younes & Littman 2004) for input, the IPP planning graph construction code (Koehler *et al.* 1997) for the $\mathcal{M}cLUG$, and the CUDD BDD package (Somenzi 1998) for representing belief states, actions, and labels. We use four test domains for our evaluation: logistics, grid, slippery gripper, and sandcastle-67. 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. We note that CPplan performs marginally worse than previously reported because our machine has one third the memory of the machine Hyafil & Bacchus (2004) used for their experiments.

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 $\text{NP}^{\text{PP}}$-complete problem, Littman, Goldsmith, & Mundhenk, 1998), but our planner, like Buridan (Kushmerick, Hanks, & Weld 1994), finds plans that satisfy the goal with probability no less than $\tau$ (an undecidable problem, Madani, Hanks, & Condon, 1999). CPplan could be used to find an optimal length plan that exceeds $\tau$ by iterating over increasing plan lengths (similar to BlackBox, Kautz, McAllester, & Selman, 1996).

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 our planner a series of prob-

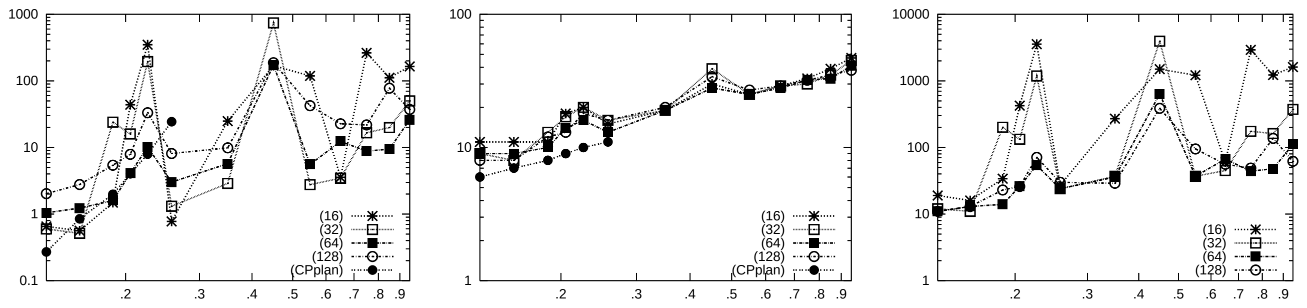

Figure 2: *Run times (s), Plan lengths, and Expanded Nodes vs. $\tau$ (log scale) for Logistics p2-2-2*

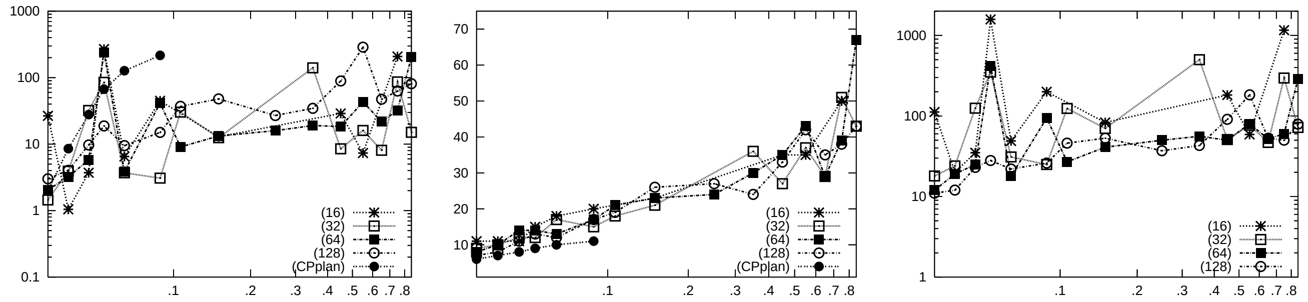

Figure 3: *Run times (s), Plan lengths, and Expanded Nodes vs. $\tau$ (log scale) for Logistics p4-2-2*

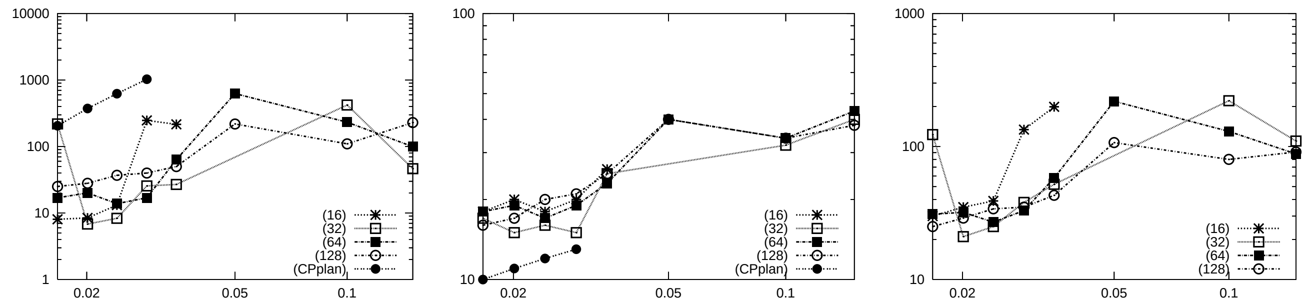

Figure 4: *Run times (s), Plan lengths, and Expanded Nodes vs. $\tau$ (log scale) for Logistics p2-2-4*

lems with increasing values for $\tau$ (which match the values found by CPplan). If our planner can solve the problem for all values of $\tau$ solved by CPplan, then we increase $\tau$ by fixed increments thereafter. We ran our planner five times on each problem and present the average run time, plan length, and expanded search nodes. Comparing the planners in this fashion allows us to compare the plan lengths found by our planner to the optimal plan lengths found by CPplan for the same value of $\tau$. Our planner often finds plans that exceed $\tau$ (sometimes quite a bit) and includes more actions, whereas CPplan meets $\tau$ 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. We intend to show that using CPplan to iterate over increasing plan lengths to solve our problem limits scalability. We choose CPplan for comparison because planners that directly solve our problem do not scale nearly as well.

**Logistics:** The logistics domain has the standard logistics actions of un/loading, driving, and flying, but adds uncertainty. Hyafil & Bacchus (2004) enriched the domain developed by Hoffmann & Brafman (2004) to not only include initial state uncertainty, but also action 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 (Hyafil & Bacchus 2004) for more details.

The plots in Figures 2, 3, and 4 compare the total run time in seconds (left), the plan lengths (center), and number of expanded search nodes (right) of our planner with 16/32/64/128 particles in the $\mathcal{M}cLUG$ versus CPplan. In

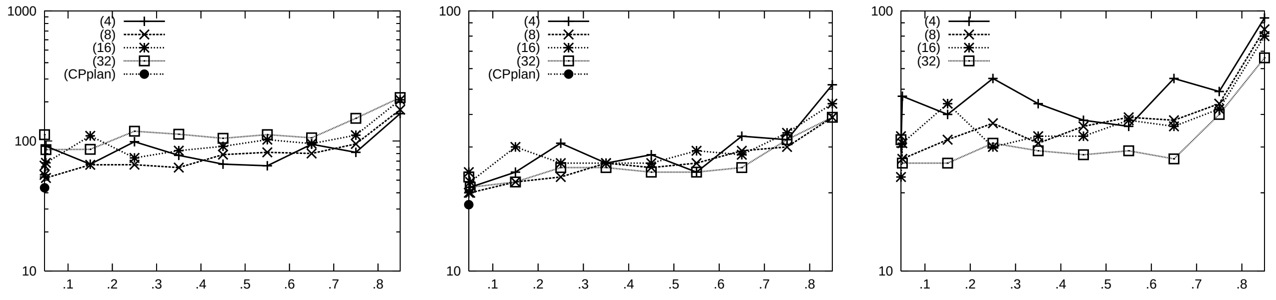

Figure 5: *Run times (s), Plan lengths, and Expanded Nodes vs. $\tau$ for Grid-0.8*

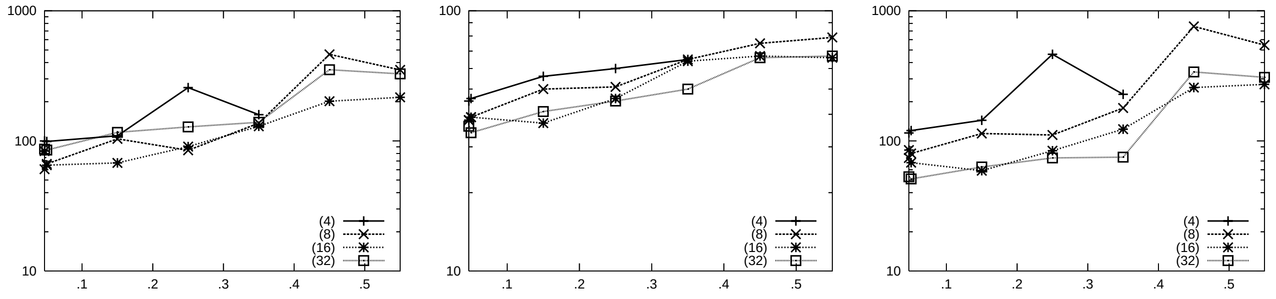

Figure 6: *Run times (s), Plan lengths, and Expanded Nodes vs. $\tau$ for Grid-0.5*

this domain we also use helpful actions from the relaxed plan (Hoffmann & Nebel 2001). 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 our planner is able to find plans much faster than CPplan for the problems they both solve. It is more interesting that our planner is able to solve problems for *much larger* values of $\tau$. Our planner finds solutions where $\tau \leq 0.95$ in p2-2-2, $\tau \leq 0.85$ in p4-2-2, and $\tau \leq 0.15$ in p2-2-4, which is respectively 3.7, 9.6, 5.2 times the maximum values of $\tau$ solved by CPplan. In terms of plan quality, the average increase in plan length for the problems we both solved was 4.6 actions in p2-2-2 (34% longer), 4.2 actions in p4-2-2 (33% longer), and 6.3 actions in p2-2-4 (56% longer).

The plot of plan lengths gives some intuition for why CPplan has trouble finding plans for greater values of $\tau$. The plan lengths for the larger values of $\tau$ approach 40-50 actions and CPplan is limited to plans of around 10-15 actions. For our planner we notice that plan length and total time scale roughly linearly as $\tau$ increases. Combined with the results in the plot showing the number of search node expansions we can see that the $\mathcal{M}cLUG$ relaxed plan heuristic directs search very well.

We would also like to point out some differences in how our planner performs when the number of particles changes. As $\tau$ increases, using more particles makes the search more consistent (i.e., fluctuation in terms of run time, plan length, and expanded nodes is minimized). Total time generally increases as the number of particles increases because the number of generated search nodes is roughly the same and the heuristic is costlier.

**Grid:** The Grid domain, as described by (Hyafil & Bacchus 2004), 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. 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.

Figure 5 shows total run time, plan lengths, and expanded search nodes for the problem. We notice that CPplan can solve the problem for only the smallest values of $\tau$, whereas our planner scales much better. For the single problem we both solve, we were on average finding solutions with 4.75 more actions (26% longer). Again, our planner scales roughly linearly because the $\mathcal{M}cLUG$ heuristic is very informed. In this problem, we are able to do very well with only 4-8 particles, leading us to believe that there are only a few very important regions of the distribution over possible worlds and we actually capture them.

Doing so well with only a few particles made us question whether the $\mathcal{M}cLUG$ is really needed. As a sanity check, we show results for a variation of the grid problem in Figure 6. This problem defines the probability that the robot moves in the intended direction to 0.5 and to 0.25 for the adjacent directions. The result is that as the robot moves, the belief state will be much less peaked and harder to capture with few particles. We see that our doubts are quieted by the results. More particles are required to get good quality plans and make search more effective.

**Slippery Gripper:** Slippery Gripper is a well known prob-

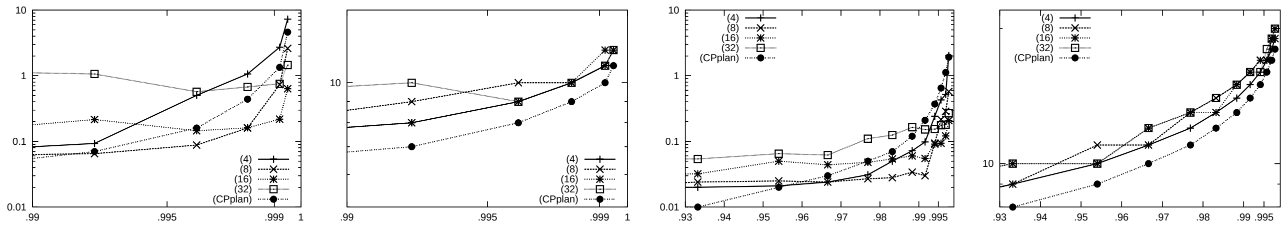

Figure 7: *Run times (s), and Plan lengths vs. $\tau$ for Slippery Gripper (left) and SandCastle-67 (right).*

lem that was originally presented by Kushmerick, Hanks, & Weld (1994). 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 $\tau$ find very short plans and take very little run time, so we focus on the higher values of $\tau$ where we see interesting scaling behavior.

Figure 7 shows the total time and plan length results for this problem in the two left-most plots. For short plans, CPplan is faster because the $\mathcal{M}cLUG$ has some additional overhead, but as $\tau$ increases and plans have to be longer the $\mathcal{M}cLUG$ proves useful. Using 8 particles, we are able to find solutions faster than CPplan in the problems where $\tau > .99$. Using more particles, we are able to find solutions faster for most problems where $\tau \geq .998$. In terms of plan quality, our solutions include on average 1.8 more actions (33% longer).

**SandCastle-67:** SandCastle-67 is another well known probabilistic planning problem, presented by Majercik & Littman (1998). 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 $\tau$ is high.

In the two right-most plots for run time and plan length in Figure 7, we see that the run time for CPplan has an exponential growth with $\tau$, whereas our methods scale roughly linearly. As $\tau$ increases, we are eventually able to outperform CPplan. In terms of plan quality, our plans included an average of 1.4 more actions (14% 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. It turns out that our estimates require us to evaluate very few possible plans (as evidenced by expanded nodes). As plans become longer, it is more difficult for CPplan to exactly evaluate plan suffixes because there are so many and they are large.

We have a couple of general observations about how the number of particles affects performance and plan length. If $\tau$ is well below $1.0$, increasing the number of particles may lead to shorter plans. However, if $\tau$ is very close to $1.0$, additional particles might be necessary in order to find a plan because low probability events may be essential to get enough probability.

As can be expected, when belief states or distributions over possible worlds are fairly non-peaked distributions, using more particles guides search better. However, without understanding the domain, it is difficult to pick the right number of particles. Fortunately, the number of particles is an easily tunable parameter that cleanly adjusts the all-too-common cost/effectiveness tradeoff in heuristics. We are currently developing an acceptance sampling method for adjusting the number of particles automatically.

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 by relaxing our grip on finding optimal plans.

## Related Work

Buridan (Kushmerick, Hanks, & Weld 1994) 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 $\mathcal{M}cLUG$. Probapop (Onder, Whelan, & Li 2006), which is built on top of Vhpop (Younes & Simmons 2003), 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 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.

Partially observable Markov decision process (POMDP) algorithms, such as (Cassandra, Littman, & Zhang 1997) to name one, are also able to solve CPP. The work on CPplan (Hyafil & Bacchus 2003; 2004) 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). CPplan also compares with MaxPlan (Majercik & Littman 1998), 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.

More closely related to our approach is the work on the CFF planner (Hoffmann & Brafman 2004), where the focus is on deriving planning graph heuristics for non-deterministic conformant planning. Like our heuristic, CFF estimates the distance between belief states, but unlike us, CFF uses satisfiability to find relaxed plans.

RTDP (Barto, Bradtke, & Singh 1995) is a popular search algorithm, used in many recent works (e.g., Mausam & Weld, 2005), 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.

## 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.

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.

Because our heuristics are inadmissible, we often return plans that are longer than optimal. We intend to investigate methods, similar to (Do & Kambhampati 2003), for post-processing our plans to improve quality. We believe that by equipping a local search planner, like LPG (Gerevini, Saetti, & Serina 2003), with $\mathcal{M}cLUG$ reachability heuristics and probabilistic plan specific repairs we could be very successful in improving seed plans generated by our planner.

We also 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 (Zemali & Fabiani 2003). 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 (Barto, Bradtke, & Singh 1995; Gerevini, Saetti, & Serina 2003), 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 MCT/NASA summer fellowship program, the ARCS foundation, 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

Barto, A. G.; Bradtke, S.; and Singh, S. 1995. Learning to act using real-time dynamic programming. *Artificial Intelligence* 72:81–138.

Bryce, D.; Kambhampati, S.; and Smith, D. 2006. Planning graph heuristics for belief space search. *Journal of Artificial Intelligence Research*. (To appear).

Cassandra, A.; Littman, M.; and Zhang, N. 1997. Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In *Proceedings of UAI'97*.

Do, M., and Kambhampati, S. 2003. Improving temporal flexibility of position constrained metric temporal plans. In *Proceedings of ICAPS'03*.

Doucet, A.; de Freita, N.; and Gordon, N. 2001. *Sequential Monte Carlo Methods in Practice*. New York, New York: Springer.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs in lpg. *Journal of Artificial Intelligence Research* 20:239–290.

Hoffmann, J., and Brafman, R. 2004. Conformant planning via heuristic forward search: A new approach. In *Proceedings of ICAPS'04*.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hyafil, N., and Bacchus, F. 2003. Conformant probabilistic planning via CSPs. In *Proceedings of ICAPS' 03*.

Hyafil, N., and Bacchus, F. 2004. Utilizing structured representations and CSPs in conformant probabilistic planning. In *Proceedings of ECAI'04*.

Kambhampati, S. 2003. 1001 ways to skin a planning graph for heuristic fun and profit. Invited Talk at ICAPS'03.

Kautz, H.; McAllester, D.; and Selman, B. 1996. Encoding plans in propositional logic. In *Proceedings of KR'96*.

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997. Extending planning graphs to an adl subset. In *Proceedings of ECP'97*.

Kushmerick, N.; Hanks, S.; and Weld, D. 1994. An algorithm for probabilistic least-commitment planning. In *Proceedings of AAAI-94*.

Littman, M.; Goldsmith, J.; and Mundhenk, M. 1998. The computational complexity of probabilistic planning. *Journal of Artificial Intelligence Research* 9:1–36.

Madani, O.; Hanks, S.; and Condon, A. 1999. On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In *Proceedings of AAAI'99*, 541–548.

Majercik, S., and Littman, M. 1998. MAXPLAN: A new approach to probabilistic planning. In *Proceedings of AIPS'98*.

Mausam, and Weld, D. 2005. Concurrent probabilistic temporal planning. In *Proceedings of ICAPS'05*.

Onder, N.; Whelan, G.; and Li, L. 2006. Engineering a conformant probabilistic planner. *Journal of Artificial Intelligence Research* 25:1–15.

Rintanen, J. 2003. Expressive equivalence of formalisms for planning with sensing. In *Proceedings of ICAPS'03*.

Smith, D., and Weld, D. 1998. Conformant graphplan. In *Proceedings of AAAI'98*.

Somenzi, F. 1998. *CUDD: CU Decision Diagram Package Release 2.3.0*. University of Colorado at Boulder.

Younes, H., and Littman, M. 2004. PPDDL1.0: An extension to PDDL for expressing planning domains with probabilistic effects. Technical report, CMU-CS-04-167, Carnegie Mellon University.

Younes, H., and Simmons, R. 2003. Vhpop: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research* 20:405–430.

Zemali, Y., and Fabiani, P. 2003. Search space splitting in order to compute admissible heuristics in planning. In *Workshop on Planen, Scheduling und Konfigurieren, Entwerfen*.