Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 172 (2008) 685–715
Artificial Intelligence
www.elsevier.com/locate/artint

Sequential Monte Carlo in reachability heuristics for probabilistic planning

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

[a] *SRI International, Artificial Intelligence Center, 333 Ravenswood Ave, Menlo Park, CA 94025, USA*
[b] *Arizona State University, Department of Computer Science and Engineering, Brickyard Suite 501, 699 South Mill Avenue, Tempe, AZ 85281, USA*
[c] *NASA Ames Research Center, Intelligent Systems Division, MS 269-2 Moffett Field, CA 94035-1000, USA*

Received 10 July 2006; received in revised form 31 August 2007; accepted 24 October 2007

Available online 6 November 2007

Abstract

Some of the current best conformant probabilistic planners focus on finding a fixed length plan with maximal probability. While these approaches can find optimal solutions, they often do not scale for large problems or plan lengths. As has been shown in classical planning, heuristic search outperforms bounded length search (especially when an appropriate 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 a distribution over many relaxed planning graphs (one planning graph for each joint outcome of uncertain actions at each time step). Computing this distribution is costly, so we apply Sequential Monte Carlo (SMC) to approximate it. One important issue that we explore in this work is how to automatically determine the number of samples required for effective heuristic computation. We empirically demonstrate on several domains how our efficient, but sometimes suboptimal, approach enables our planner to solve much larger problems than an existing optimal bounded length probabilistic planner and still find reasonable quality solutions.

Keywords: Planning; Heuristics

1. Introduction

Agents that execute risky, fault-prone actions cannot always guarantee plan success. In such cases, a viable alternative to abandoning their goals is to formulate high probability plans. With a probability distribution over the outcomes of their actions, an agent can assess the probability a plan satisfies its goals. Conformant probabilistic planning (CPP) is a special case where the agent cannot observe the outcomes of their actions and must find an unconditional plan.

* Corresponding author.
E-mail address: bryce@ai.sri.com (D. Bryce).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2007.10.018

Conditional probabilistic planning allows the agent to observe action outcomes and execute different sub-plans, *conditioned* on their observations.

Recent successes [7,12,22] in scaling-up non-deterministic conformant and conditional planning have formulated the problem as heuristic search in belief state space. Much of the improvements come from using novel reachability heuristics based on planning graphs to guide the search. The heuristics estimate the distance between a belief state (a set of states that are believed to be possible) and a goal state. Using this work as a starting point, we address conformant probabilistic planning as heuristic search in belief state space for several reasons:

- despite long standing interest [25,26,30,35], probabilistic plan synthesis algorithms have a terrible track record in terms of scalability,
- our search formulation is relatively straight-forward, allowing us to concentrate on our contribution—heuristic search guidance,
- conformant planning heuristics (with little or no modification) have proven useful in the, more general, conditional planning problem [12,22].

The current best conformant probabilistic planners are only able to handle very small problems. This is in part due to their focus on finding optimal solutions (i.e., finding the maximal probability of satisfying the goal with a k length plan). In contrast, there has been steady progress in scaling deterministic planning by finding sub-optimal, but feasible plans (i.e., satisfying the goal with a plan whose length is not necessarily minimal). Much of this progress has come from the use of sophisticated reachability heuristics. In this work, we show how to effectively use reachability heuristics [11] to solve conformant probabilistic planning problems (where a minimum probability of goal satisfaction is guaranteed, but the plan length is not necessarily minimal). We use our previous work on planning graph heuristics for non-deterministic planning [12] as our starting point. While we do not discuss conditional planning in this work, our previous work on heuristics for non-deterministic planning [12] and initial results for conditional probabilistic planning [9,10] indicate this work is relevant to conditional probabilistic planning.

We investigate an extension of our work [12] that uses a planning graph generalization called the Labeled 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, [44]), where each is associated with a possible world state. By analyzing the set of planning graphs, we compute a heuristic estimate of the cost to transition all possible current states to a goal state. The *LUG* (as described in [12]) handles only uncertainty about the initial state. In this work, we extend the *LUG* to consider action uncertainty and probabilistic uncertainty (i.e., probability distributions instead of sets of possibilities). This extension involves symbolically representing how at each level CGP explicitly splits the planning graph over all joint outcomes of uncertain actions to capture the resulting possible worlds and assigning probabilities to the possible worlds.

A possible world is a set of possible outcomes to all uncertain events (i.e., one starting state and one outcome for each action at each level of the planning graph). The number of possible worlds grows exponentially with the number of levels in the planning graph because in each level an action can have a different outcome. However, because the planning graph is a relaxation of the planning problem it represents the propositions reachable if all actions per level are executed in parallel and do not conflict. Therefore each possible world captures a joint set of action outcomes per level. CGP represents the set of planning graphs like a tree, where each path corresponds to a possible world. Two possible worlds may have partially overlapping paths in this tree, meaning they share the same action outcomes and start state up to the point that they diverge.

Without uncertain actions, CGP and the *LUG* represent an exponential[1] (yet, constant) number of possible worlds at each time step. The possible worlds are only attributed to state uncertainty, and deterministic actions keep their number constant. With uncertain actions, an explicit or symbolic representation of planning graphs is *exactly* representing an exponentially increasing set of possible worlds (corresponding to uncertainty in the source state and each action level). 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. The key contribution of this work is to use approximate methods for

[1] Exponential in the number of unknown propositions.

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) [17] 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 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, in our first contribution we generalize the *LUG* to symbolically represent the set of planning graph particles in a planning graph we call the Monte Carlo *LUG* ($\mathcal{M}cLUG$) [13]. We show that by using the $\mathcal{M}cLUG$ to extract a relaxed plan heuristic we are able to find much higher probability plans than one of the current best conformant probabilistic planners CPplan [25,26] in a number of domains. In the problems that both can solve (i.e., those that have the same minimum probability of goal satisfaction), our approach finds comparable quality plans to the optimal plans found by CPplan.

A natural question that accompanies most SMC approaches is deciding how many samples to use. In this article, we build upon our initial work on the $\mathcal{M}cLUG$[13] to answer this issue. As we will demonstrate, each planning problem is solved best with a different number of samples in each $\mathcal{M}cLUG$. 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 [19] on particle filters, outperforms the best manually selected number of particles across many domains.

Our presentation starts by describing the relevant background on planning graph heuristics, CPP, and SMC. We follow with a running 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 cover the details of $\mathcal{M}cLUG$, and the associated relaxed plan heuristic, which appeared in [13]. We then describe a new contribution that automatically determines the number of samples. Finally, we present an empirical analysis of our techniques by comparing with CPplan, and analyze 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 for classical planning, and then describe our action and belief state representation, the CPP problem, the semantics of conformant probabilistic plans, our search algorithm, and sequential Monte Carlo.

2.1. Classical planning graph heuristics

This section describes the foundations of planning graph heuristics used in classical planning. We start by formally discussing the classical planning problem, follow with a description of the most common heuristic search formulation (forward state space search), and finish with how to derive search heuristics from planning graphs.

Classical planning The classical planning problem is defined as a tuple (P, A, I, G), where P is a set of propositions, A is a set of actions, I is a set of initial state propositions, and G is a set of goal propositions. This description of classical planning uses many representational assumptions (described below) consistent with the STRIPS language [18]. While STRIPS is only one of many choices for action representation, it is very simple and most other action languages can be compiled down to STRIPS [37]. In STRIPS a state s is a proper subset of the propositions P, where every proposition $p \in s$ is said to be true (or to hold) in the state s. Any proposition $p \notin s$ is false in s. The initial state s_I is specified by a set of propositions $I \subseteq P$ known to be true (the false propositions are inferred by the closed world assumption) and the goal is a set of propositions $G \subseteq P$ that must be made true in a state s for s to be a goal state. Each action $a \in A$ is described by the pair $(\rho_e(a), (\varepsilon^+(a), \varepsilon^-(a)))$, where $\rho_e(a)$ is a set of propositions for

execution preconditions, and $(\varepsilon^+(a), \varepsilon^-(a))$ is an effect where $\varepsilon^+(a)$ describes a set of propositions that it causes to become true and $\varepsilon^-(a)$ is a set of propositions it causes to become false. An action a is applicable $appl(a, s)$ to a state s if each precondition proposition holds in the state, $\rho_e(a) \subseteq s$. The successor state s' is the result of executing an applicable action a in state s, where $s' = exec(a, s) = s \backslash \varepsilon^-(a) \cup \varepsilon^+(a)$. A sequence of actions $\{a_1, \ldots, a_m\}$, executed in state s, results in a state s', where $s' = exec(a_m, exec(a_{m-1}, \ldots, exec(a_1, s)\ldots))$ and each action is executable in the appropriate state. We will assume that a valid plan is a sequence of actions that can be executed from s_I and results in a goal state.[2] The number of actions m in the sequence is the length of the plan.

Forward state space search Classical planning can be viewed as finding a path from an initial state to a goal state in a state transition graph. This view suggests a simple algorithm that constructs the state transition graph and uses a shortest path algorithm to find a plan in $O(n \log n)$ time. However, practical problems have a very large number of states n. In many problems there are hundreds of propositions P, leading to an extremely large number of states $n = 2^{|P|}$, which makes the above algorithm impractical.

Instead of an explicit graph representation, it is possible to use a search algorithm and the propositional representation to construct regions of the state transition graph, as needed. However, in the worst case, it is possible to still construct the entire transition graph. Heuristic search algorithms, such as A* search, can "intelligently" search for plans and hopefully avoid visiting large regions of the transition graph. The critical concern of such heuristic search algorithms is the design of a good heuristic.

To illustrate heuristic search for plans, consider the most popular search formulation, progression (a.k.a. forward-chaining). The search creates a projection tree rooted at the initial state s_I by applying actions to leaf nodes (representing states) to generate child nodes. Each path from the root to a leaf node corresponds to a plan prefix, and expanding a leaf node generates all single step extensions of the prefix. A heuristic estimates the "goodness" of each leaf node, and in classical planning this can be done by estimating the cost to *reach* a goal state. With the heuristic estimate, search can focus effort on expanding the most promising leaf nodes.

Planning graph heuristics One way to compute exact reachability information is to compute the full projection tree rooted at the initial state. Within this tree, the exact reachability cost for each node is the minimal length path to reach a state satisfying the goal. Computing exact reachability information is impractical as it is no cheaper than the cost of solving the original problem! Instead, research on classical planning explores more efficient ways of computing reachability information approximately; one of the better ways is through planning graph analysis.

Planning graphs We start by formalizing planning graphs and follow with a planning graph based reachability heuristic (called a relaxed plan). The planning graph is a layered graph structure with alternating action and proposition layers. There are edges between layers: an action has its preconditions in the previous layer and its positive effects in the next layer. Unlike the projection tree, the planning graph structure can be computed in polynomial time. The planning graph can be viewed as the exact projection tree for a relaxation of the domain that ignores the interactions between and within action effects during graph expansion.

Traditionally, progression search uses a different planning graph to compute the reachability heuristic for each state s. A planning graph $PG(s, A)$, constructed for the state s (referred to as the projected state) and the action set A is a leveled graph, captured by layers of vertices $(\mathcal{P}_0(s), \mathcal{A}_0(s), \mathcal{P}_1(s), \mathcal{A}_1(s), \ldots, \mathcal{A}_k(s), \mathcal{P}_{k+1}(s))$, where each level i consists of a proposition layer $\mathcal{P}_i(s)$ and an action layer $\mathcal{A}_i(s)$. In the following, we simplify the notation for a planning graph to $PG(s)$, assuming that the entire set of actions A is always used. The notation for action layers $\mathcal{A}_i$ and proposition layers $\mathcal{P}_i$ also assumes that the state s is implicit.

The first proposition layer, $\mathcal{P}_0$, is defined as the set of propositions in the state s. An action layer $\mathcal{A}_i$ consists of all actions that have all of their precondition propositions in $\mathcal{P}_i$. A proposition layer $\mathcal{P}_i$, $i > 0$, is the set of all propositions given by the positive effect[3] of an action in $\mathcal{A}_{i-1}$. It is common to use implicit actions for proposition

[2] Plans can in theory contain parallel actions.

[3] The reason that actions do not contribute their negative effects to proposition layers (which contain only positive propositions) is a syntactic convenience of using STRIPS. Since action preconditions and the goal are defined only by positive propositions, it is not necessary to reason about reachable negative propositions. In general, action languages (such as ADL [40]) allow negative propositions in preconditions and goals, requiring the planning graph to maintain "literal" layers that record all reachable values of propositions [29].

```
RPExtract(PG(s), G)
 1: Let k be the index of the last level of PG(s)
 2: for all p ∈ G ∩ P_k do {Initialize Goals}
 3:    P_k^RP ← P_k^RP ∪ p
 4: end for
 5: for i = k ... 1 do
 6:    for all p ∈ P_i^RP do {Find Supporting Actions}
 7:       Find a ∈ A_{i−1} such that p ∈ ε+(a)
 8:       A_{i−1}^RP ← A_{i−1}^RP ∪ a
 9:    end for
10:    for all a ∈ A_{i−1}^RP, p ∈ ρe(a) {Insert Preconditions}
11:       P_{i−1}^RP ← P_{i−1}^RP ∪ p
12:    end for
13: end for
14: return (P_0^RP, A_0^RP, P_1^RP, ..., A_{k−1}^RP, P_k^RP)
```

Fig. 1. Relaxed plan extraction algorithm.

persistence (a.k.a. noop actions) to ensure that propositions in $\mathcal{P}_{i-1}$ persist to $\mathcal{P}_i$. A noop action a_p for proposition p is defined as $\rho_e(a_p) = \varepsilon^+(a_p) = p$. Planning graph construction can terminate when the goal is reachable (i.e., every goal proposition is present in a proposition layer).

Relaxed plans Planning graph heuristics are used to estimate the plan cost to transition between two states, a source and a destination state. The source state is always the state that defines $\mathcal{P}_0$, and the destination state is one of potentially many goal states. Heuristics typically do not distinguish between the goal states, computing the cost to achieve the goal propositions. One way to do this is to compute a relaxed plan.

Relaxed plans identify the actions needed to causally support all goals (while ignoring conflicting actions). Through a simple back-chaining algorithm (Fig. 1) called *relaxed plan extraction*, it is possible to identify the actions in each level that are needed to support the goals or other actions. Relaxed plans are subgraphs $(\mathcal{P}_0^{\mathrm{RP}}, \mathcal{A}_0^{\mathrm{RP}}, \mathcal{P}_1^{\mathrm{RP}}, \ldots, \mathcal{A}_{n-1}^{\mathrm{RP}}, \mathcal{P}_n^{\mathrm{RP}})$ of the planning graph, where each layer corresponds to a set of vertices. A relaxed plan satisfies the following properties: (i) every proposition $p \in \mathcal{P}_i^{\mathrm{RP}}$, $i > 0$, in the relaxed plan is supported by an action $a \in \mathcal{A}_{i-1}^{\mathrm{RP}}$ in the relaxed plan, and (ii) every action $a \in \mathcal{A}_i^{\mathrm{RP}}$ in the relaxed plan has its preconditions $\rho_e(a) \subseteq \mathcal{P}_i^{\mathrm{RP}}$ in the relaxed plan. A relaxed plan captures the causal chains involved in supporting the goals, but ignores how actions may conflict.

Fig. 1 lists the algorithm used to extract relaxed plans. Lines 2–4 initialize the relaxed plan with the goal propositions. Lines 5–13 are the main extraction algorithm that starts at the last level of the planning graph n and proceeds to level 1. Lines 6–9 find an action to support each proposition in a level. Line 7 is the most critical step in the algorithm where we select actions. It is common to prefer noop actions for supporting a proposition (if possible) because the relaxed plan is likely to include fewer extraneous actions. For instance, a proposition may support actions in multiple levels of the relaxed plan; by supporting the proposition at the earliest possible level, it can persist to later levels. It is also possible to select actions based on other criterion, such as probability, which we will describe later. Lines 10–12 insert the preconditions of chosen actions into the relaxed plan. The algorithm ends by returning the relaxed plan, which is used to compute the heuristic as the total number of non-noop actions in the action layers.

2.2. Conformant probabilistic planning

The conformant probabilistic planning problem generalizes the classical planning problem in several ways: there is a probability distribution over initial states, there is a probability distribution over the outcomes of each action, and the goal may be satisfied with less than 1.0 probability. A conformant probabilistic planning problem is given by the tuple $CPP = (P, A, b_I, G, \tau)$, where P is a set of propositions, A is a set of actions, b_I is an initial belief state (probability distribution over initial states), G is the goal description (a conjunctive set of propositions), and τ is a goal probability satisfaction threshold ($0 < \tau \leqslant 1$).

Belief states 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 $P_f \subseteq P$ ($P_t \cap P_f = \emptyset$) that 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 $(\rho_e(a), \Phi(a))$, where $\rho_e(a)$ is an enabling precondition, and $\Phi(a)$ is a set of outcomes. The enabling precondition $\rho_e(a)$ is a conjunctive set of propositions that determines action applicability. An action a is applicable $appl(a, b)$ in belief state b if it is applicable in each state in the belief state, $\forall_{s \in b} \rho_e(a) \subseteq s$. The causative outcomes $\Phi(a)$ are a set of tuples $(w_i(a), \Phi_i(a))$ 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 $\rho_{ij}(a) \rightarrow (\varepsilon^+_{ij}(a), \varepsilon^-_{ij}(a))$, where both the antecedent (secondary precondition) $\rho_{ij}(a)$ and positive $\varepsilon^+_{ij}(a)$ and negative $\varepsilon^-_{ij}(a)$ consequents are conjunctive sets of propositions. This representation of effects follows the 1ND normal form presented by Rintanen [43]. As outlined in the probabilistic PDDL (PPDDL) standard [47], it is possible to use the effects of every action 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'. Executing action a in state s will result in a single state s' for each outcome $\Phi_i(a)$:

$$s' = exec(\Phi_i(a), s) = s \cup \left(\bigcup_{j:\ \rho_{ij}(a) \subseteq s} \varepsilon^+_{ij}(a) \right) \setminus \left(\bigcup_{j:\ \rho_{ij}(a) \subseteq s} \varepsilon^-_{ij}(a) \right)$$

The value for $T(s, a, s')$ is the sum of the weight of each outcome where $s' = exec(\Phi_i(a), s)$, such that:

$$T(s, a, s') = \sum_{i:\ s' = exec(\Phi_i(a), s)} w_i(a)$$

Executing action a in belief state b, denoted $exec(a, b) = b_a$, defines the successor belief state such that $b_a(s') = \sum_{s \in b} b(s) T(s, a, s')$.

A sequence of actions $\{a_1, \ldots, a_m\}$, executed in belief state b, results in a state b', where $b' = exec(a_m, exec(a_{m-1}, \ldots, exec(a_1, b) \ldots))$ and each action is executable in the appropriate state. A valid conformant plan is a sequence of actions that can be executed from b_I and results in a belief state b' where $b'(G, \emptyset) \geqslant \tau$. The number of actions m in the sequence is the length of the plan.

Belief space search Like our description of classical planning, we use a forward-chaining weighted A* search to find solutions to CPP. The search graph is organized using nodes to represent belief states (instead of 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 $b(G, \emptyset) \geqslant \tau$. 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.[4] In the remainder of the paper we concentrate on the central issue of how to compute $h(b)$ using an extension of planning graph heuristics to CPP.

2.3. 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 $Pr(X_k) = \int Pr(X_k|X_{k-1}) Pr(X_{k-1}) \, \mathrm{d}X_{k-1}$. 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 $\{x^n_k\}^{N-1}_{n=0}$, where the probability that X_k takes value x_k,

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

[4] 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.

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 [44] 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 use of SMC 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.

3. 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 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, $P = \{\text{atP1}, \text{atP2}, \text{inP}\}$, and our initial belief state b_I has two states $s0 = \{\text{atP1}\}$ and $s1 = \{\text{atP2}\}$ where $b_I(s0) = b_I(s1) = 0.5$, and the goal is $G = \{\text{inP}\}$. 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 LP1 and LP2 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. In the following, we provide a key to our notation for action descriptions and the actual descriptions for the two actions:

$$
\begin{aligned}
a = (\ & \rho_e(a), \\
& \Phi(a) = \{\ \ldots, \\
& \qquad (w_i(a), \Phi_i(a) = \{\ldots, \varphi_{ij}(a) = (\rho_{ij}(a), (\varepsilon_{ij}^{+}(a), \varepsilon_{ij}^{-}(a))), \ldots\}), \\
& \qquad \ldots\})
\end{aligned}
$$

$$
\begin{aligned}
\text{LP1} = (\ & \{\}, \\
& \{(0.8, \{(\{\text{atP1}\}, (\{\text{inP}\}, \{\}))\}), \\
& \ (0.2, \{\})\}
\end{aligned}
$$

$$
\begin{aligned}
\text{LP2} = (\ & \{\}, \\
& \{(0.8, \{(\{\text{atP2}\}, (\{\text{inP}\}, \{\}))\}), \\
& \ (0.2, \{\})\}
\end{aligned}
$$

Each action has two outcomes. The first outcome has a single conditional effect, and the second has no effects.

Figs. 2, 3, and 4 illustrate several approaches to planning graph based reachability analysis for our simplified logistics domain. (We assume that 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 Fig. 2, 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 $\Phi_0(\text{LP1})$ occurs, then effect $\varphi_{00}(\text{LP1})$ is enabled and will occur, however even if $\Phi_0(\text{LP2})$ occurs $\varphi_{00}(\text{LP2})$ 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 and the possible worlds at time zero. For instance, the possible world $s0, \{\Phi_0(\text{LP1}), \Phi_0(\text{LP2})\}$ is formed from

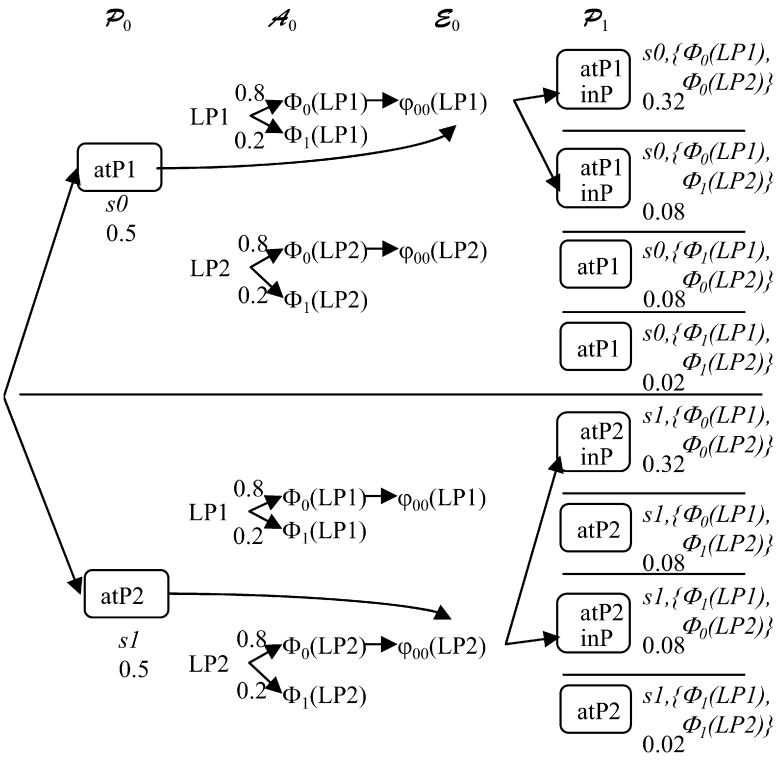

Fig. 2. CGP representation.

world $s0$ when outcomes Φ_0(LP1) and Φ_0(LP2) co-occur in $s0$. Likewise, possible world $s1$, $\{\Phi_1(\text{LP1}), \Phi_0(\text{LP2})\}$ is formed from possible world $s1$ when outcomes Φ_1(LP1) and Φ_0(LP2) occur in $s1$.

In our example, CGP could determine the exact distribution over possible worlds at time one (i.e., $Pr(X_{0:1})$). Because each time step k introduces a new random variable X_k to capture joint action outcomes, the probability distribution $Pr(X_{0:k})$ over level k proposition layers is exponentially larger than the previous level.

Fig. 2 identifies that the 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 as done in CGP, to extract a relaxed plan that satisfies the goal with 0.8 probability. We choose actions needed to support the goal in each possible world and prefer actions already chosen for other possible worlds. The resulting actions form the basis for heuristic computation, described in more detail later.

McCGP Next, we illustrate a Monte Carlo simulation approach we call Monte Carlo CGP (McCGP), in Fig. 3. The idea is to represent a set of N deterministic planning graphs as particles. In our example, suppose 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. The first two samples correspond to $s0$ and the second two correspond to $s1$. To simulate a particle we first insert the applicable actions into $\mathcal{A}_0$. We then insert effects into $\mathcal{E}_0$ by sampling from the distribution of joint action outcomes (i.e. $x_k^n \sim Pr(X_k|x_{k-1}^n)$). The first particle x^0 samples outcomes Φ_0(LP1) and Φ_1(LP2), identifying the effects that construct the subsequent proposition layer. Note that by sampling the action outcomes, each particle is a deterministic planning graph.

While it may seem strange to sample the outcomes of several actions at each time step, where in a particle filter only one action occurs at each time step, we must capture the relaxed semantics of the planning graph. That is, several actions execute in parallel at each time step, and we assume they do not conflict. Despite executing actions in parallel, we must sample each action outcome independently because their outcomes are independent. Thus, simulating each particle $x_k^n \sim Pr(X_k|x_{k-1}^n)$ involves sampling each action outcome.

In our example, the simulation was lucky and the proposition layer for each particle at time 1 satisfies the goal. 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. [12]. We can union the sets

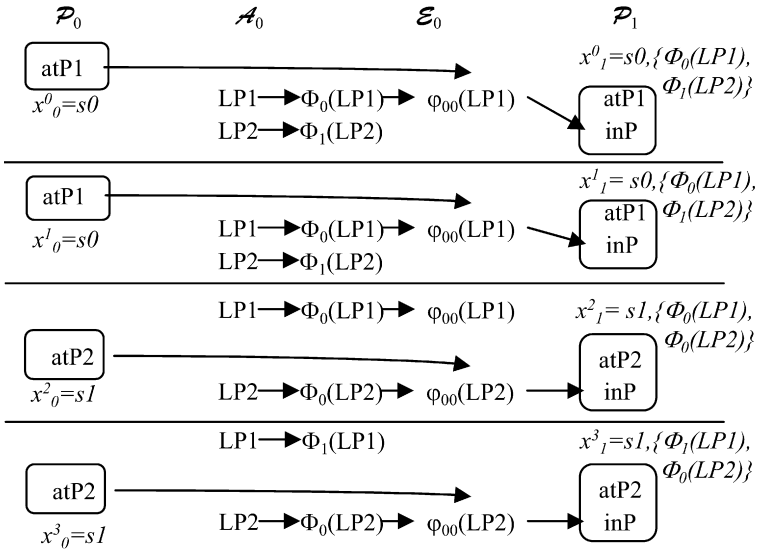

Fig. 3. McCGP representation.

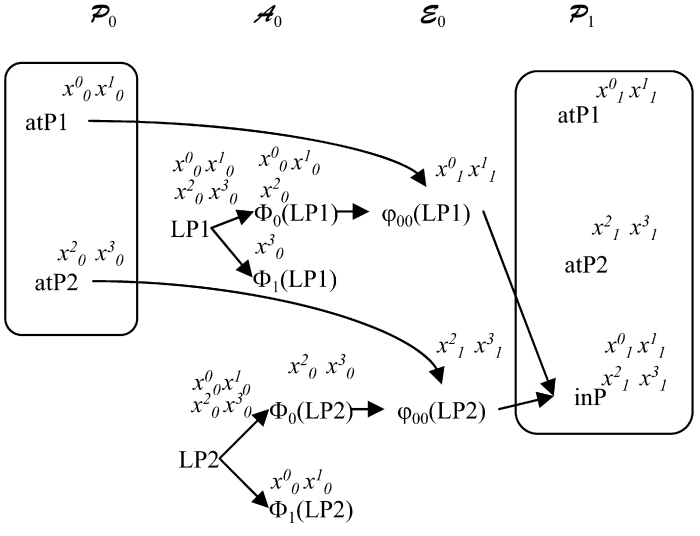

Fig. 4. $\mathcal{M}cLUG$ representation.

of actions appearing in each level of the relaxed plans. For example, if we use LP1 in the zeroth level of two relaxed plans, the unioned relaxed plan will have LP1 once in its zeroth action layer. Unioning the action layers step-by-step captures how actions can be used once to satisfy the goal in multiple possible worlds (similar to how CGP prefers to reuse actions from other possible worlds).

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.

McLUG Using our ideas from Bryce et al. [12] , 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 Fig. 4. 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. In the following section we work through the example in Fig. 4 and provide definitions needed to construct a $\mathcal{M}cLUG$.

4. Symbolic planning graph representation

In our previous work [12] we describe a planning graph generalization called the Labeled Uncertainty Graph (*LUG*). In non-deterministic conformant planning, the *LUG* symbolically represents the exponential number of planning graphs used by Conformant GraphPlan [44]. In [12], we 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 that we consider in [12] there is only a single random event X_0 captured by labels because the actions are deterministic and only the source state is uncertain. Where CGP would build a planning graph for each possible state, the *LUG* uses labels to denote which of the explicit planning graphs would contain a given proposition or action in a level. For instance, if CGP builds a planning graph for possible worlds $s1, \ldots, sn$ (each a state in a source belief state) and the planning graphs for $s1, \ldots, sm$ each have proposition p in level k, then the *LUG* has p in level k labeled with a propositional formula $\ell_k(p)$ whose models are $\{s1, \ldots, sm\}$. In the worst case, the random event X_0 captured by the labels has $2^{|P|}$ outcomes (i.e., all states are possible in the belief state), characterized by a logical formula over $\log_2(2^{|P|}) = |P|$ boolean variables.

We [12] expand the *LUG* until a level where all goal propositions are labeled with all states in the source belief state, meaning the goal is strongly reachable[5] in the relaxed plan space. We define a strong relaxed plan procedure that back-chains on the *LUG* to support the goal propositions in all possible worlds. This relaxed plan proves effective for search in both conformant and conditional non-deterministic planning.

4.1. 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 boolean function labels with $|P|$ 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 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 $Pr(X_{0:1})$). For the distribution $Pr(X_{0:k})$, we need $|P| + k|A|\log_2(m)$ boolean variables. In a reasonable size 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 [45]) 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 initial state uncertainty to alleviate the need for additional label variables. This technique, mentioned in [44], 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. If each action has m outcomes, then the planning graph has k steps, and the belief state has size $|b| = |\{s|s \in b\}|$, then the transformed belief state would describe $O(|b||A|mk)$ states. While this compilation would allow us to restrict the growth of *LUG* labels, there is still a problem. We are solving indefinite horizon planning problems, meaning that the number of possible time points for an action to execute is unbounded (i.e., k is unknown). This further means that the size of the compilation (for any large k) is unbounded. Consequently, we shift our focus to approximating the distribution using particles.

[5] The goal is guaranteed to be reachable despite all uncertainty.

4.2. Sampled symbolic 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 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 $\mathcal{M}cLUG$ 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 proposition layer $\mathcal{P}_0$ and an inductive step to generate a graph level $\{\mathcal{A}_k, \mathcal{E}_k, \mathcal{P}_k\}$ 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 $\mathcal{P}_0$ captures a set of N particles $\{x_0^n\}_{n=0}^{N-1}$ drawn from the source belief state.[6] Each particle x_0^n corresponds to a state $s \in b$ in the source belief state, and, equivalently, the initial proposition layer representing the state. (The super-script of a particle denotes its assigned possible world index, and the sub-script denotes its time index.) Later particles ($k > 0$) correspond to sets of action outcomes sampled at or before time k.

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

In the example, the initial proposition layer is $\mathcal{P}_0 = \{\text{atP1}, \text{atP2}\}$, 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 $\mathcal{A}_0 = \{\text{LP1}, \text{LP2}\}$, and the labels are:

$$\ell_0(\text{LP1}) = \ell_0(\text{LP2}) = \{x_0^0, x_0^1, x_0^2, x_0^3\}$$

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 labeled with a particle $\mathcal{E}_k = \{\varphi_{ij}(a) | \ell_k(\varphi_{ij}(a)) \neq \emptyset\}$. Determining which effects get labeled requires simulating the path of each particle, $x_{k+1}^n \sim P(X_{k+1} | x_k^n)$. The path of a particle is simulated by sampling from the distribution over the joint outcomes of all enabled actions. 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, adding each sampled outcome $\Phi_i(a)$ to the particle at the next time step x_{k+1}^n. 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, such that $\ell_k(\varphi_{ij}(a)) = \{x_k^n | x_k^n \in \ell_k(a), \Phi_i(a) \in x_k^n, \text{ and } x_k^n \in \bigcap_{p \in \rho_{ij}(a)} \ell_k(p)\}$. 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.

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 LP1 and outcome 1 for LP2, which are then labeled 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 LP1 with x_1^2 because the effect is not enabled in

[6] While we represent belief states with ADDs, our current implementation samples belief states by enumerating the states represented by the ADD, generating a random probability, and selecting the corresponding state via the cumulative density function.

x_0^2. Finally, for particle x_0^3 we sample outcome 1 of LP1 and outcome 0 of LP2. Thus, the effect layer is $\mathcal{E}_0 = \{\varphi_{00}(\text{LP1}), \varphi_{10}(\text{LP1}), \varphi_{00}(\text{LP2}), \varphi_{10}(\text{LP2})\}$, labeled as:

$$\ell_0(\varphi_{00}(\text{LP1})) = \{x_1^0, x_1^1\}$$
$$\ell_0(\varphi_{10}(\text{LP1})) = \{x_1^3\}$$
$$\ell_0(\varphi_{00}(\text{LP2})) = \{x_1^2, x_1^3\}$$
$$\ell_0(\varphi_{10}(\text{LP2})) = \{x_1^0, x_1^1\}$$

Proposition layer Proposition layer $\mathcal{P}_k$ contains all propositions that are made positive by an effect in $\mathcal{E}_{k-1}$. Each proposition is labeled by the particles of every effect that give it support. The proposition layer is defined as $\mathcal{P}_k = \{p|\ell_k(p) \neq \emptyset\}$, 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 $\mathcal{P}_1 = \mathcal{P}_0 \cup \{\text{inP}\}$. The propositions are labeled as:

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

The propositions from the previous proposition layer $\mathcal{P}_0$ persist through implicit noop actions, allowing them to be labeled 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 $\mathcal{M}cLUG$ 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 $\mathcal{M}cLUG$ as the condition when every proposition in a proposition layer is labeled with the same number of particles as in the previous level. If level off is reached without $Pr(G|X_k) \geqslant \tau$, 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 $\mathcal{M}cLUG$ expansion, it is possible for the $\mathcal{M}cLUG$ 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 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 $\mathcal{M}cLUG$ expansion at the first proposition layer that satisfies the goal with probability no less than τ. It should be straight forward to adapt the $\mathcal{M}cLUG$ to the more general multi-objective setting.

4.3. Heuristics

By terminating construction of the $\mathcal{M}cLUG$ at level k, 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* [12]. 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.

RPExtract($LUG(b)$, G)
1: Let n be the index of the last level of $LUG(b)$
2: **for all** $p \in G$ **do** {Initialize Goals}
3: $\quad \mathcal{P}_n^{\mathrm{RP}} \leftarrow \mathcal{P}_n^{\mathrm{RP}} \cup p$
4: $\quad \ell_n^{\mathrm{RP}}(p) \leftarrow \bigcap_{p' \in G} \ell_n(p')$
5: **end for**
6: **for** $k = n \ldots 1$
7: $\quad$ **for all** $p \in \mathcal{P}_k^{\mathrm{RP}}$ **do** {Support Each Proposition}
8: $\quad\quad \ell \leftarrow \ell_k^{\mathrm{RP}}(p)$ {Initialize Possible Worlds to Cover}
9: $\quad\quad$ **while** $\ell \neq \emptyset$ **do** {Cover Label}
10: $\quad\quad\quad$ Find $\varphi_{ij}(a) \in \mathcal{E}_{k-1}$ such that $p \in \varepsilon_{ij}^{+}(a)$ and $(\ell_k(\varphi_{ij}(a)) \cap \ell) \neq \emptyset$
11: $\quad\quad\quad \mathcal{E}_{k-1}^{\mathrm{RP}} \leftarrow \mathcal{E}_{k-1}^{\mathrm{RP}} \cup \varphi_{ij}(a)$
12: $\quad\quad\quad \ell_k^{\mathrm{RP}}(\varphi_{ij}(a)) \leftarrow \ell_k^{\mathrm{RP}}(\varphi_{ij}(a)) \cup (\ell_k(\varphi_{ij}(a)) \cap \ell)$
13: $\quad\quad\quad \mathcal{A}_{k-1}^{\mathrm{RP}} \leftarrow \mathcal{A}_{k-1}^{\mathrm{RP}} \cup a$
14: $\quad\quad\quad \ell_k^{\mathrm{RP}}(a) \leftarrow \ell_k^{\mathrm{RP}}(a) \cup (\ell_k(\varphi_{ij}(a)) \cap \ell)$
15: $\quad\quad\quad \ell \leftarrow \ell \backslash \ell_k(\varphi_{ij}(a))$
16: $\quad\quad$ **end while**
17: $\quad$ **end for**
18: $\quad$ **for all** $a \in \mathcal{A}_{k-1}^{\mathrm{RP}}$, $p \in \rho_e(a)$ **do** {Insert Action Preconditions}
19: $\quad\quad \mathcal{P}_{k-1}^{\mathrm{RP}} \leftarrow \mathcal{P}_{k-1}^{\mathrm{RP}} \cup p$
20: $\quad\quad \ell_{k-1}^{\mathrm{RP}}(p) \leftarrow \ell_{k-1}^{\mathrm{RP}}(p) \cup \ell_{k-1}^{\mathrm{RP}}(a)$
21: $\quad$ **end for**
22: $\quad$ **for all** $\varphi_{ij}(a) \in \mathcal{E}_{k-1}^{\mathrm{RP}}$, $p \in \rho_{ij}(a)$ **do** {Insert Effect Preconditions}
23: $\quad\quad \mathcal{P}_{k-1}^{\mathrm{RP}} \leftarrow \mathcal{P}_{k-1}^{\mathrm{RP}} \cup p$
24: $\quad\quad \ell_{k-1}^{\mathrm{RP}}(p) \leftarrow \ell_{k-1}^{\mathrm{RP}}(p) \cup \ell_{k-1}^{\mathrm{RP}}(\varphi_{ij}(a))$
25: $\quad$ **end for**
26: **end for**
27: **return** $(\mathcal{P}_0^{\mathrm{RP}}, \mathcal{E}_0^{\mathrm{RP}}, \mathcal{A}_0^{\mathrm{RP}}, \mathcal{P}_1^{\mathrm{RP}}, \ldots, \mathcal{A}_{n-1}^{\mathrm{RP}}, \mathcal{E}_{n-1}^{\mathrm{RP}}, \mathcal{P}_n^{\mathrm{RP}})$

Fig. 5. *LUG* relaxed plan extraction algorithm.

The intuition behind a conformant relaxed plan is to capture both the positive interaction and independence between possible worlds as each achieves the goals [12]. In [12] we explore multiple ways to compute relaxed plan heuristics from explicit and symbolic sets of planning graphs. The most simple approach is to extract a relaxed plan from each planning graph and somehow aggregate the relaxed plans to reflect the conformant (aggregate) cost of achieving the goals in every planning graph. We can either take the summation or maximization of the number of actions in each relaxed plan, or merge the relaxed plans. We merge relaxed plans by unioning the actions appearing in the first step, second step, and so on of each relaxed plan. The unioned relaxed plan then reflects the actions needed in each of the planning graphs and summing the number of resulting actions avoids counting the common actions more than once.

In the *LUG* (and the $\mathcal{M}cLUG$), we can find this unioned relaxed plan symbolically, meaning the unioning is implicit. The procedure for *LUG* relaxed plan extraction is shown in Fig. 5. Much like the algorithm for relaxed plan extraction from classical planning graphs, *LUG* relaxed plan extraction supports propositions at each time step (lines 7–17), and includes the supporting actions in the relaxed plan (lines 18–25). The significant difference with classical planning is with respect to the required label manipulation, and to a lesser extent, reasoning about actions and their effects separately. The algorithm starts by initializing the set of goal propositions $\mathcal{P}_n^{\mathrm{RP}}$ at time n and associating a label $\ell_n^{\mathrm{RP}}(p)$ with each to denote the possible worlds where they must be supported (lines 2–5). Then for each time step (lines 6–26), the algorithm determines how to support propositions and what propositions must be supported at the preceding time step. Supporting an individual proposition at time k in possible worlds $\ell_k^{\mathrm{RP}}(p)$ (lines 7–17) is the key decision point of the algorithm, embodied in line 10. First, we initialize a variable ℓ with the remaining possible worlds to support the proposition (line 8). While there are remaining possible worlds to support, we choose effects and their associated actions (lines 9–16). Those effects that i) have the proposition as a positive effect and ii) support it in possible worlds that need to be covered (i.e., $\ell_k(\varphi_{ij}(a)) \cap \ell \neq \emptyset$) are potential choices. In line 10, one of these effects is chosen (later we describe some heuristics for making this choice). We store the effect (line 11) and the possible worlds it supports (line 12), as well as the associated action (line 13) and the possible worlds where its

effect is used (line 14). The possible worlds left to support are those not covered by the chosen effect (line 15). After selecting the necessary actions and effects in a level, we examine their preconditions and antecedents to determine the propositions we must support next (lines 18–25); the possible worlds to support each proposition are simply the union of the possible worlds where they are needed to support an action or effect (lines 20 and 24). The extraction ends by returning the labeled subgraph of the *LUG* that is needed to support the goals in all possible worlds (line 27).

The same procedure will work, without modification, to extract $\mathcal{M}cLUG$ relaxed plans. However, our interpretation of the semantics does change slightly. We pass a $\mathcal{M}cLUG$ for a given belief state b to the procedure, instead of a *LUG*. The labels for goal propositions (line 4) represent particles, and using the $\mathcal{M}cLUG$ termination criterion, we do not require labels to contain all particles—only a proportion no less than τ.

In our example, the goal inP is labeled 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}(LP1), and particles x_1^2, x_1^3 are supported by φ_{00}(LP2), so we include both LP1 and LP2 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 LP1 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 LP2. 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.

There is one important aspect of the relaxed plan extraction that we have not discussed: how to select the supporting effects (line 10). Lines 8 through 16 are actually a greedy set cover algorithm: the set of possible worlds ℓ must be covered with subsets $\{\ell_k(\varphi_{ij}(a))|\varphi_{ij}(a) \in \mathcal{E}_{k-1}, (\ell_k(\varphi_{ij}(a)) \cap \ell) \neq \emptyset\}$. The typical greedy set cover algorithm will choose the subset that covers the most remaining elements (possible worlds) at each iteration. We use this strategy, with the only caveat that we prefer noop actions (described in more detail below). Aside from noops, we prefer effects that cover more possible worlds because they contribute more probability mass; hopefully by concentrating the probability mass within fewer effects, we will include fewer actions and have lower cost relaxed plans.

Our strategy for preferring noop actions follows a potentially non-intuitive line of reasoning that differs very much from the reasoning used in classical relaxed plan extraction. In classical planning, preferring noops ensures that propositions are supported at the earliest level possible, meaning they can potentially support more actions at later levels. The net effect is that fewer actions will be included in the relaxed plan. Including fewer actions in a $\mathcal{M}cLUG$ relaxed plan is not necessarily ideal. We would like to capture how probabilistic actions must sometimes be executed multiple times to accumulate probability for supported propositions and avoid counting extra actions that do not increase probability.

We argue that preferring noops in $\mathcal{M}cLUG$ relaxed plan extraction does in fact achieve this desirable behavior, under the following assumption: the particles supporting a proposition accumulate over the levels of the planning graph. Except in pathological cases where we use very few particles and action outcome distributions are very skewed (or deterministic), it is not possible to support a proposition with all particles in one level of the $\mathcal{M}cLUG$. In general, a proposition accumulates support over levels, meaning the supporting actions are staggered over the levels of the $\mathcal{M}cLUG$. Preferring noops ensures that, with respect to each individual particle, an appropriate supporting action is chosen as early as possible in the planning graph. However, due to the staggered support, the relaxed plan includes these actions (that may be different instances of the same action) across several levels of the planning graph. The relaxed plan heuristic counts these actions separately, capturing how repetition is necessary for supporting a proposition with high probability.

4.4. 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. We determine N by the number of state samples needed to approximate representative belief states.

It is well known that classical planning graphs approximate state transition graphs, reasoning about the propositions (union of states) reachable within k steps of a state. Similarly, the $\mathcal{M}cLUG$ approximates the belief state transition graph, reasoning about the probability distribution over proposition layers within k steps from a belief state. The belief states that we 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 [19] 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 [15], it is possible to derive a value for the belief state approximation $N(b,\epsilon)$ 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 [27] 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 a number of state samples $N(b,\epsilon)$ 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 a poor indicator of stochastic belief states reached after a few steps. It seems reasonable to consider several belief states and use the average number of particles needed for approximation.

Since the $\mathcal{M}cLUG$ 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 single random walk (sampling action choices) in the belief state space to identify reachable belief states.[7] 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. We define the value for N as the average of the number of samples per belief state,

$$N = \frac{\sum_{b\in B} N(b,\epsilon)}{|B|}$$

A final consideration for selecting N is the error ϵ and the probability of making at most ϵ error, δ. In our empirical evaluation we will identify a good value for ϵ for a fixed δ. While we are trading one free parameter (N) for another (ϵ), our intent is to find a reasonable *domain-independent* parameter setting value for ϵ.

[7] Multiple random walks may improve our knowledge of the reachable belief state space, but in informal experiments additional random walks did not change the results.

5. 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 $\mathcal{M}cLUG$ by comparing with one of the leading approaches to CPP that was available at the time of writing, CPplan [25,26]. We also internally evaluate our approach by adjusting (both manually and automatically) the number of particles N that we use in each $\mathcal{M}cLUG$. We refrain from comparing with POMDP solvers (such as POMDP-solve [14]), as did Hyafil and Bacchus [26], 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 [41] may make a better comparison, but there are many implementation-level details preventing a thorough comparison at this time.

5.1. 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 at the time of this writing other 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 [47] for input, the IPP planning graph construction code [29] for the $\mathcal{M}cLUG$, and the CUDD BDD package [45] for representing belief states, actions, and labels.

We use ADD operations to compute successor belief states, but sample from belief states by enumerating their states.[8] Because we use ADDs, in the worst case it is possible for the representation of belief states to have exponential size (i.e., the ADD is a tree where each path is length $|P|$ and each leaf is mapped to a different probability).

Fig. 6 depicts our planner architecture. The two inputs to the planner are the problem specification, and the method to select the number of particles N. The problem is grounded, pre-processed, and compiled into ADDs. If we choose to automatically determine number of particles, then we determine the length of our random walk, take the random walk, analyze the belief states in the random walk, and finally compute N. The search commences by expanding search nodes and computing heuristics. Each heuristic computation involves constructing a $\mathcal{M}cLUG$ with the chosen number of particles until the goal is reached with enough probability. From the $\mathcal{M}cLUG$, 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, [32]), but *POND*, like Buridan [30], finds plans that satisfy the goal with probability no less than τ (an undecidable problem, [34]). CPplan could be used to find an optimal length plan that exceeds τ by iterating over increasing plan lengths (similarly to BlackBox for classical planning [28]).

5.2. 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 [26] enriched the domain developed by Hoffmann and Brafman [7] to not only include

[8] One could improve belief state sampling by sampling directly from the ADD, subject to the probability distribution represented by the ADD.

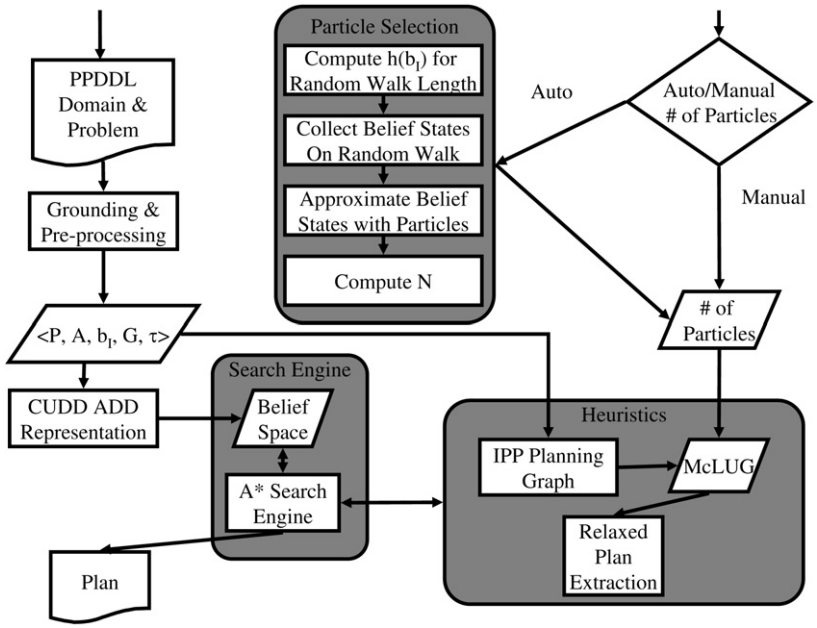

Fig. 6. *POND* architecture.

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 [26] for more details.

Grid The Grid domain, as described by Hyafil and Bacchus [26], is a 10×10 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 (5×5 or 15×15) 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. [30]. 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 [35]. 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.

5.3. Environment

In our test setup, we use a 2.66 GHz P4 Linux machine with 1 GB 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 non-optimal heuristic planner.

6. Empirical analysis: External evaluation & particle set size

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

6.1. Comparison with CPplan

We compare with CPplan 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 [26] used for their experiments. One major limitation on CPplan scalability is memory consumption.

Logistics The plots in Figs. 7, 8, and 9 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 prune search with helpful actions from the relaxed plan [23]. We notice that CPplan is able to at best find solutions where $\tau \leqslant 0.26$ in p2-2-2, $\tau \leqslant 0.09$ in p4-2-2, and $\tau \leqslant 0.03$ 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 $\tau \leqslant 0.95$ in p2-2-2, $\tau \leqslant 0.75$ in p4-2-2, and $\tau \leqslant 0.035$ 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 of plan quality, the average

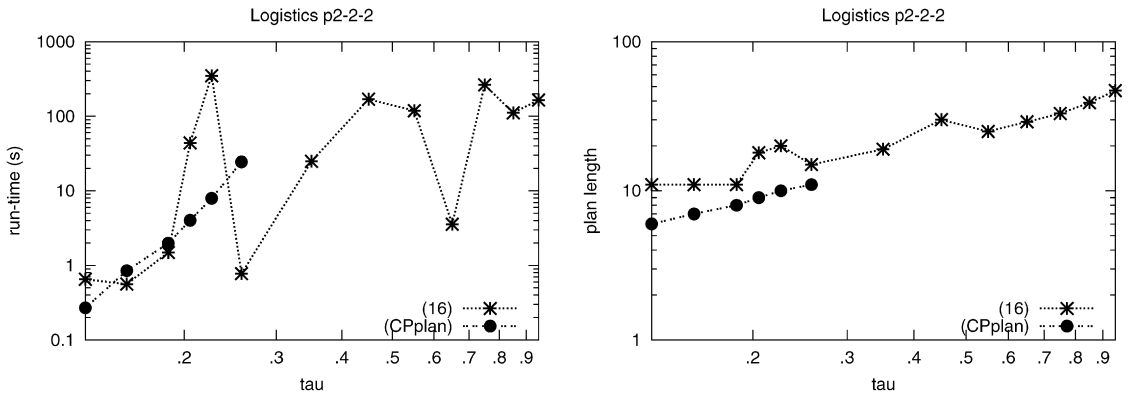

Fig. 7. Run times (s) and plan lengths vs. τ (log scale) for Logistics p2-2-2.

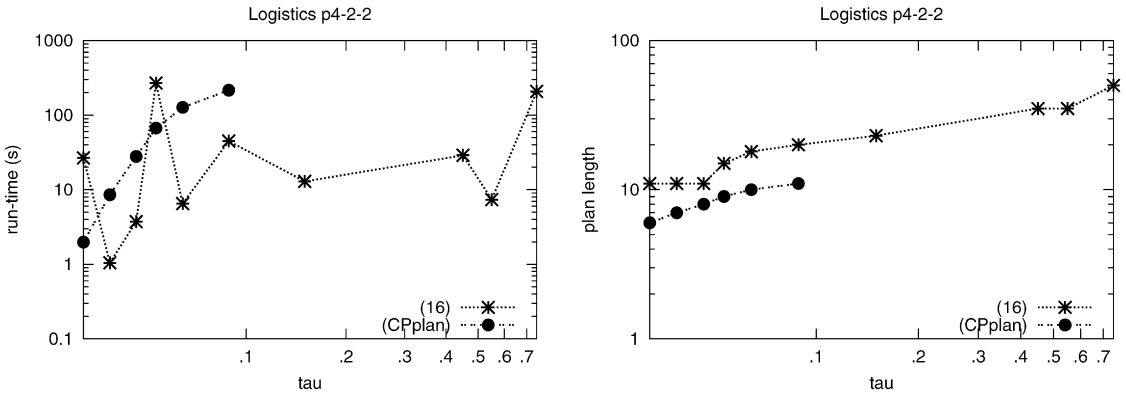

Fig. 8. Run times (s) and plan lengths vs. τ (log scale) for Logistics p4-2-2.

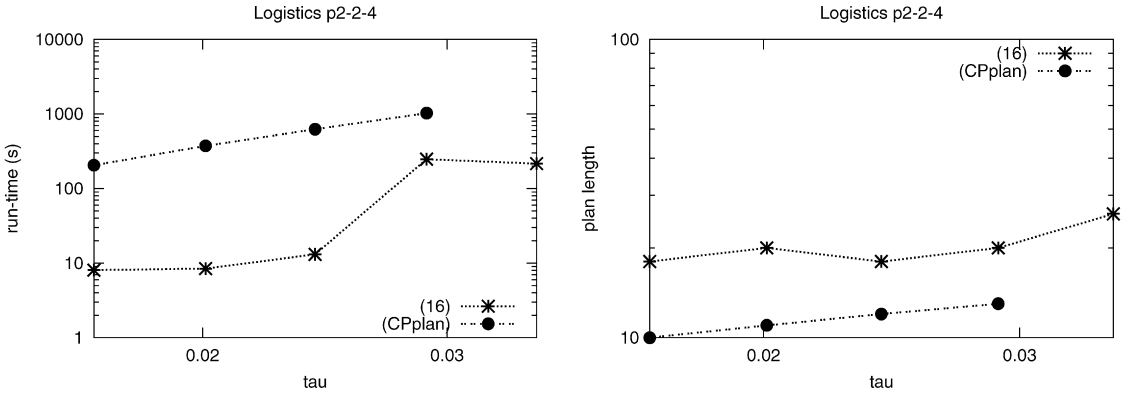

Fig. 9. Run times (s) and plan lengths vs. τ (log scale) for Logistics p2-2-4.

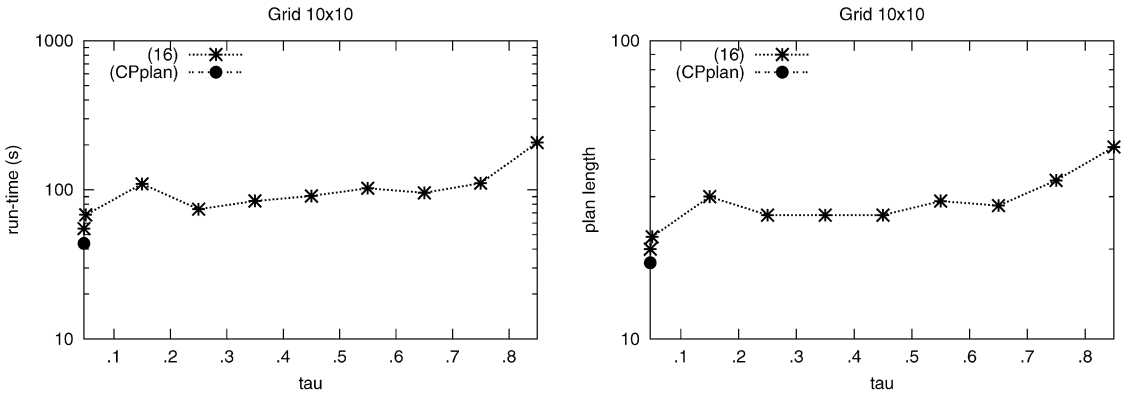

Fig. 10. Run times (s) and plan lengths vs. τ for Grid-0.8.

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.

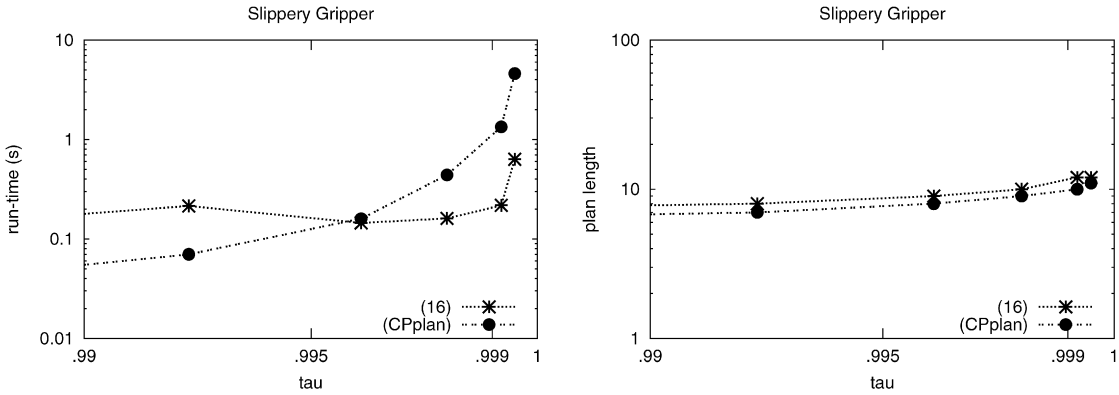

Fig. 11. Run times (s), and plan lengths vs. τ for Slippery Gripper.

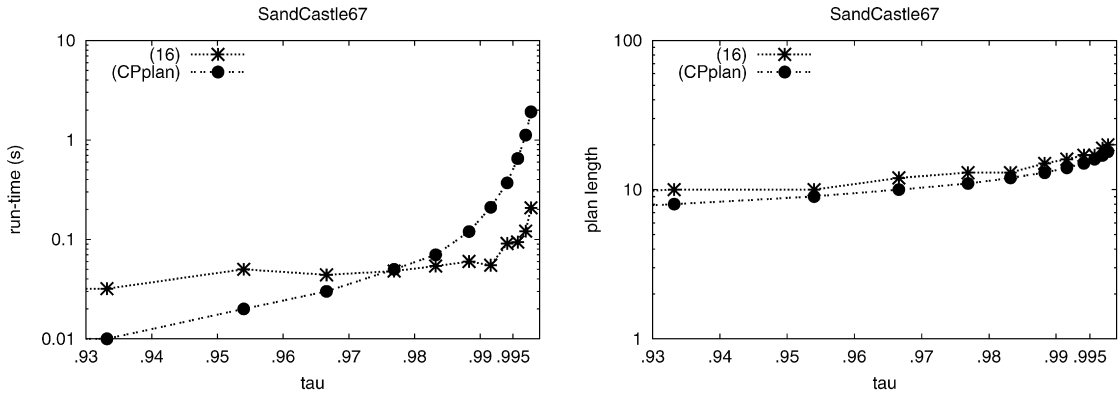

Fig. 12. Run times (s), and plan lengths vs. τ for Sand Castle-67.

Grid Fig. 10 shows total run times and plan lengths for the 10×10 Grid problem. We notice that CPplan can solve the problem for only the smallest value of τ, whereas *POND* finds plans for much larger values of τ. For the single problem we both solve, we found a solution with 6 more actions (26% longer).

Slippery Gripper Fig. 11 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 $\tau > 0.995$. In terms of plan quality, our solutions include on average 1.6 more actions (20% longer).

Sand Castle-67 The plots for run time and plan length in Fig. 12 show that the run time for CPplan has an exponential growth with τ, whereas our method performs much better. As τ increases, we are eventually able to outperform CPplan. 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.

Since POND may find suboptimal plans, CPplan may be able to find higher probability solutions in the extra steps taken by POND. In fact, CPplan may only need to repeat a few of the key actions to increase the probability of goal satisfaction. This suggests an approach that takes a small seed plan and repeats it sufficiently many times to guarantee a certain probability of goal satisfaction. Using such sequences could amount to anything from macro-actions to plan merging. Buridan [30] provides a similar functionality within the context of partial order planning, but fails to scale for

the lack of effective heuristics. Buridan includes specific search operations that add several invocations of an action to increase the probability of an open precondition. Directly facilitating repetition in state based search could complicate the search algorithm unnecessarily. Our approach does in fact reason about repeating actions within the heuristic, but not explicitly in the search. That is, when an action is needed to support the goal in multiple planning graphs (each for a different sample), then it is possible for the action to appear multiple times in the relaxed plans. This repetition in the relaxed plan models the repetition needed to accumulate greater probability.

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.

6.2. 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 0.15, 0.1, 0.05 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 is fixed at 0.1 (informal experiments that varied δ did not significantly impact results).

In the following, 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). (For colors see the web version of this article.) 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 average values of N chosen by each ϵ compared with the best manual value of N for each problem.

Logistics Figs. 13, 14, and 15 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, as one might expect. Without the right number of particles the heuristic is either too weak or too costly.

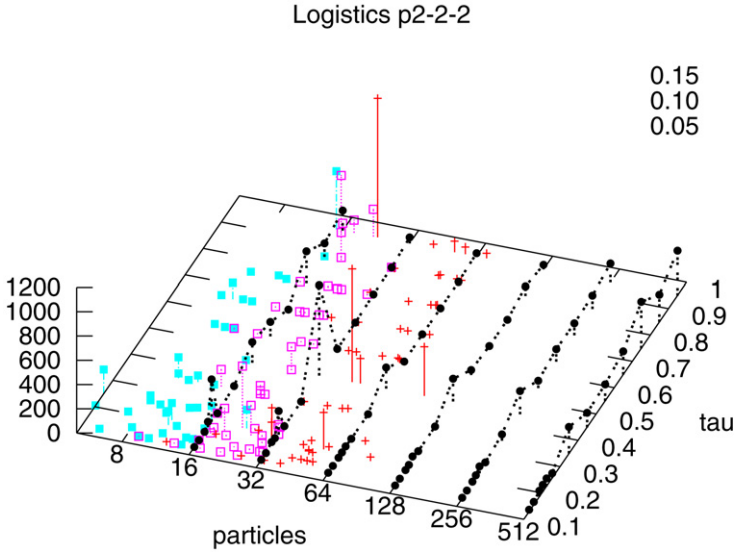

Fig. 13. Run times (s) vs. τ vs. N for Logistics p2-2-2.

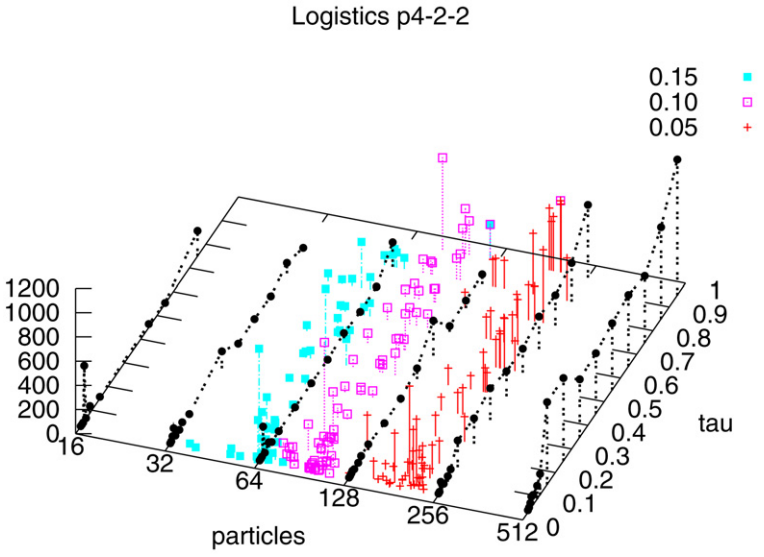

Fig. 14. Run times (s) vs. τ vs. N for Logistics p4-2-2.

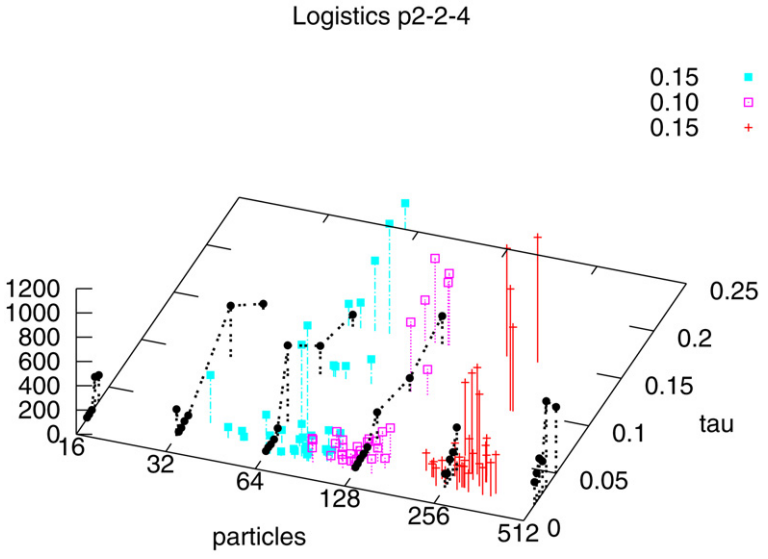

Fig. 15. Run times (s) vs. τ vs. N for Logistics p2-2-4.

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 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, p2-2-4 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

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

Problem	#	$N = 16$		$N = 32$		$N = 64$		$N = 128$	
		$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	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

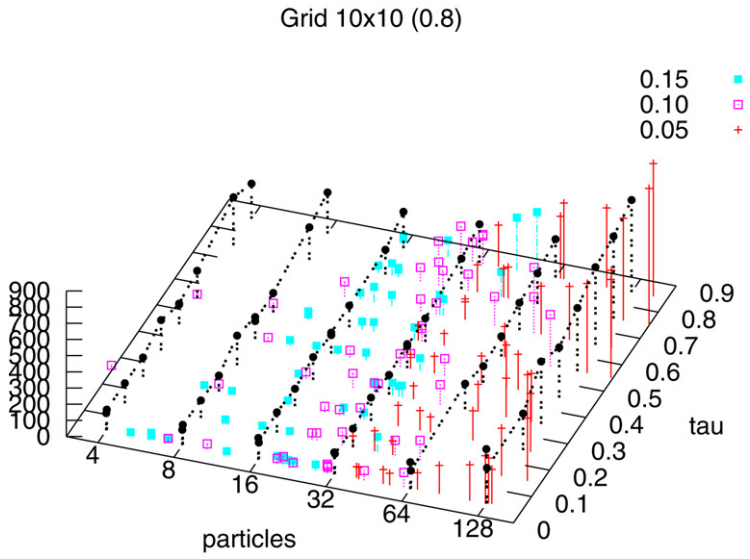

Fig. 16. Run times (s) vs. τ vs. N for Grid 10×10 with 0.8 correct transitions.

to compare with CPplan. The automated particle selection does best with $\epsilon = 0.15$ and $\epsilon = 0.1$. With $\epsilon = 0.05$, the number of particles can be too large. The automated selection typically selects values for N between 32 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.

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 10×10 Grid problem from our previous analysis as the base domain and extend it in two orthogonal directions. First, we keep the 10×10 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 5×5 and 15×15 (while keeping 0.8 for transitions as in the base domain).

Fig. 16 depicts results for the base domain. As the number of particles falls too low or grows too large, the 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 (around 16–32) than in Logistics (around 64–128). This difference between Grid and Logistics may be due to the relaxed plan heuristic giving better estimates in this simpler (more regular) domain structure. As we will see in the other versions of the Grid domain, the effective number of particles was not very different when we changed size, but are different when changing uncertainty.

Fig. 17 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

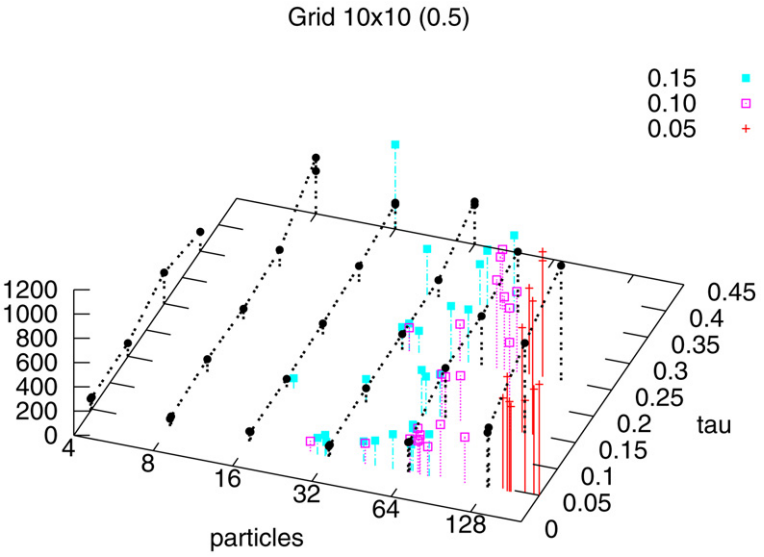

Fig. 17. Run times (s) vs. τ vs. N for Grid 10×10 with 0.5 correct transitions.

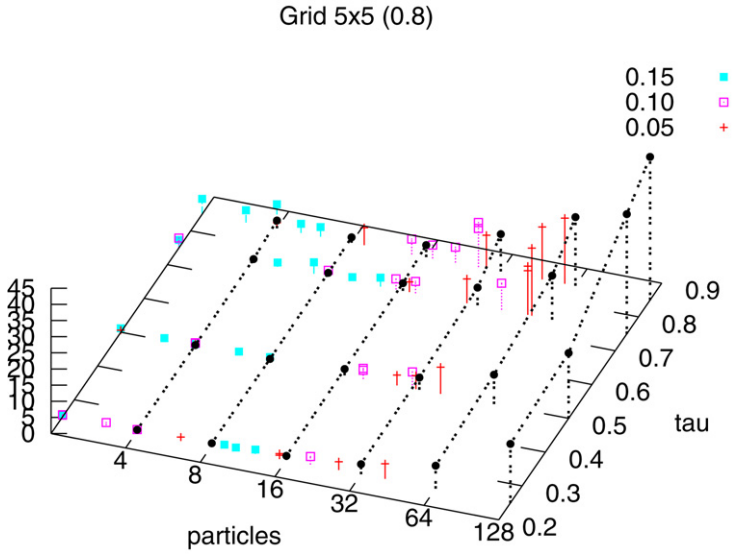

Fig. 18. Run times (s) vs. τ vs. N for Grid 5×5 with 0.8 correct transitions.

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. The automatic particle selection notices this distinction, and chooses more particles for this domain than the base domain.

Fig. 18 shows results for the 5×5 Grid domain. *POND* needs very few particles to do well in this domain because the plans are relatively short. Belief states do not become very flat because they contain relatively fewer states. The automated particle selection performs best when ϵ is 0.15 or 0.1.

Fig. 19 shows results for the 15×15 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. The automated particle selection tends to overestimate N, increasing planning time due to the cost of computing the heuristic.

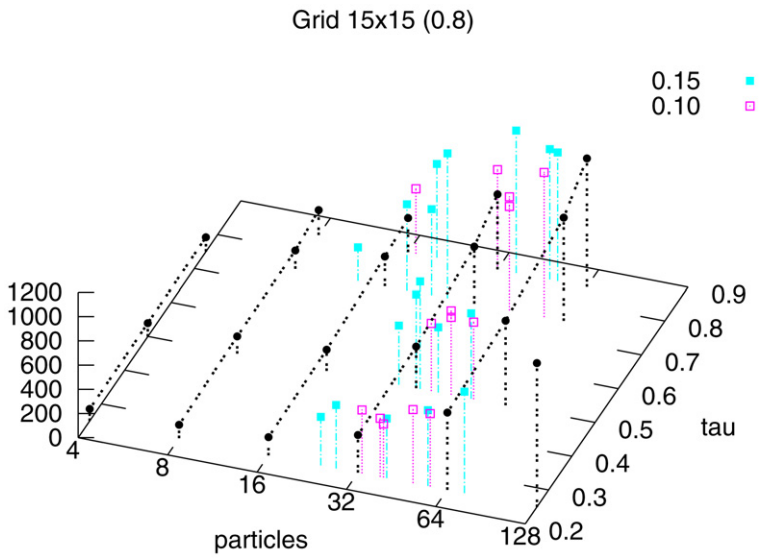

Fig. 19. Run times (s) vs. τ vs. N for Grid 15×15 with 0.8 correct transitions.

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

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

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 5×5 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, except in the 15×15 grid where the cost of computing the heuristic for low values of ϵ is prohibitive.

Slippery Gripper Fig. 20 shows results for the Slippery Gripper problem. The results indicate few particles are needed, and extra particles just increase 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 values of N. Table 3 lists a summary of results for the manual particle selection that show using 16 particles does best overall.

Sand Castle-67 Fig. 21 shows results for the Sand Castle-67 problem. Again, choosing the right number of particles is important, in this case 16–32. The automated particle selection is able to find good values for N in this domain. There are relatively few states and plans are short and the sample-based approximation 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.

6.3. Summary of automating the size of particle sets

Selecting the right number of particles for a domain is not easy. Uncertainty, as well as the cost and effectiveness of the relaxed plan heuristic affect the number of particles needed. While automatically estimating the right number of particles does not reason about the cost of computing the heuristic, only the accuracy, our automated particle selection

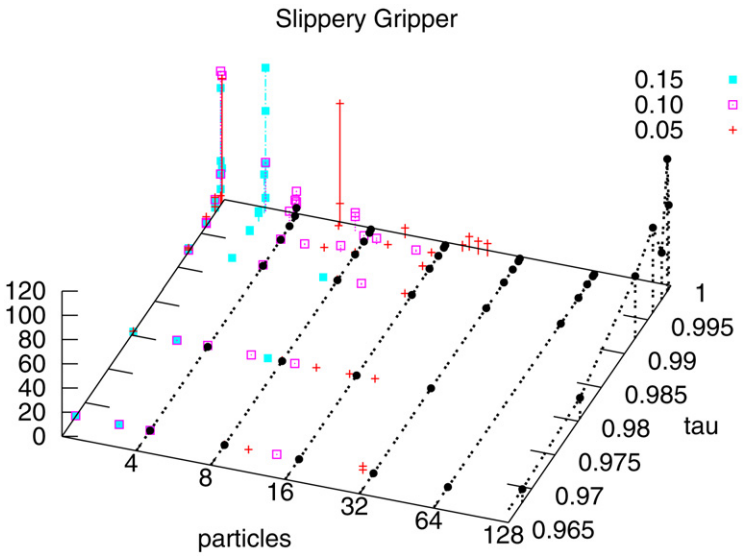

Fig. 20. Run times (s) vs. τ vs. N for Slippery Gripper.

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

Problem	#	$N = 4$		$N = 8$		$N = 16$		$N = 32$	
		$\overline{T}$	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

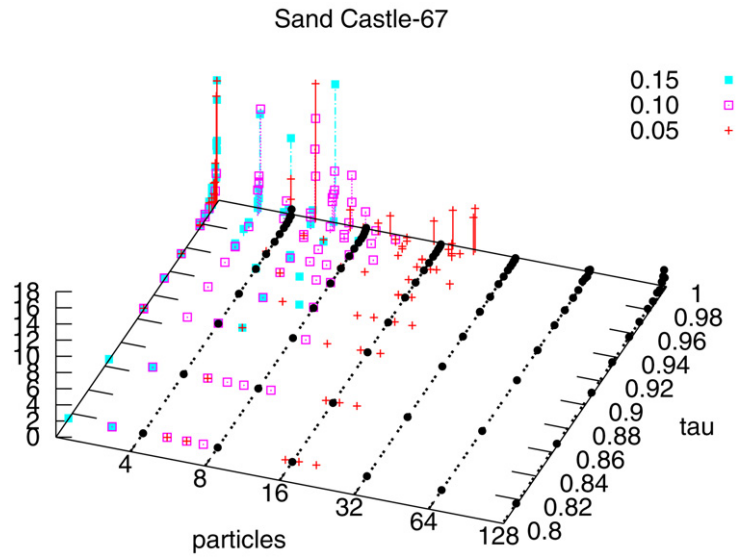

Fig. 21. Run times (s) vs. τ vs. N for Sand Castle-67.

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

Problem	#	$\epsilon = 0.1$		$\epsilon = 0.01$		$\epsilon = 0.005$		Best N		
		$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S	$\overline{T}$	S	N
Logistics p2-2-2	65	38.65	52	74.43	60	103.40	63	20.16	61	64
Logistics p4-2-2	80	105.57	69	115.09	76	198.27	79	44.82	49	64
Logistics p2-2-4	50	200.18	34	182.97	31	369.16	30	136.31	31	64
Grid 10×10 (0.8)	50	64.45	49	87.87	49	250.22	50	136.67	50	16
Grid 10×10 (0.5)	30	311.52	26	407.99	22	861.44	12	106.35	30	16
Grid 5×5 (0.8)	20	2.13	20	3.88	20	7.10	20	1.03	20	8
Grid 15×15 (0.8)	20	735.85	18	680.49	14	0	0	212.53	18	16
Slippery Gripper	50	11.08	50	8.59	50	6.83	50	0.18	50	16
Sand Castle	85	1.68	85	1.40	85	1.44	85	0.06	85	16
	450	1471.11	403	1562.70	407	1797.85	389	658.11	394	

Table 5
Summary of the average automated N found by $\epsilon = 0.15, 0.1$, and 0.05, and the best manual N

Problem	$\epsilon = 0.15$	$\epsilon = 0.1$	$\epsilon = 0.05$	Best N
Logistics p2-2-2	11.13	21.60	47.21	64
Logistics p4-2-2	60.27	95.08	194.94	64
Logistics p2-2-4	76.74	123.35	284.73	64
Grid 10×10 (0.8)	20.84	30.00	81.40	16
Grid 10×10 (0.5)	45.69	69.95	145.92	16
Grid 5×5 (0.8)	5.75	14.70	27.55	8
Grid 15×15 (0.8)	35.33	40.93	n.a.	16
Slippery Gripper	2.86	4.54	11.70	16
Sand Castle	3.07	4.86	11.04	16

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. Grid 10×10 (0.8) is one domain where the automated selection took less time than the best manual selection. The automated particle selection generally takes more time than the best manual particle selection by over or under estimating the right number of particles by a small amount. This suggests that other factors may be important, such as the shape of the search space (e.g., search depth and branching factor). However, the relative number of particles across the domains does reflect differences in the domains.

The automated selection with ϵ equal to 0.15, 0.1, 0.05, respectively solves 403, 407, and 389 of 450 instances, where the best manual selection solves 394. The total respective average times for the automated selection are 1471.11, 1562.70, and 1797.65 seconds, compared to 658.11 seconds for the best manual selection.

Table 5 shows the average number of particles chosen by each value of ϵ within automated method compared with the manual method. As we expect, the number of particles increases inversely with ϵ. In many cases, the number of selected particles is very close to the best manually selected number of particles.

7. Related work

Probabilistic planning has been a problem of interest over the last few decades, with much activity in recent years. Much of the early work explored different search space representations and search algorithms, which we summarize in the first subsection below. Only recently have planning graph techniques been applied to such problems, first as a search substrate, and later as the basis for heuristics; we summarize these in the second subsection. Lastly, we relate our approach to other work that also use Monte Carlo in planning.

7.1. Alternative approaches to probabilistic planning

The first approaches to probabilistic planning come from the operations research community work on controlling Markov chains, i.e. Markov Decision Processes [42]. The CPP problem, as we have stated it, can be solved with partially observable Markov decision process (POMDP) algorithms, such as [14]. The work on CPplan [25,26] shows that a POMDP algorithm [14] 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 more general problem by finding plans for all possible initial belief states.

From an AI perspective, Buridan [30] is the first symbolic planner 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 search heuristics.

The previously mentioned CPplan planner improves upon the original bounded length formulation for CPP used by MaxPlan [35]. Where CPplan is based on CSP, MaxPlan uses a variation of satisfiability. Both planners rely on storing previously computed plan suffixes to prune their search space. CPplan happens to use a superior representation and it has been shown to be more efficient for several problems.

In a similar formulation to that of CPplan, the recent Complan planner [24] improves approaches to optimal bounded length CPP problems. Complan improves upon CPplan by removing the heavy memory requirements due to storing plan suffixes. 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. While we do not provide extensive empirical comparisons with Complan, the results presented in [24] show that Complan takes on the order of hours to find solutions for larger instances of the 10×10 Grid problem, where we take minutes. While Complan can find plans that are high probability in fewer plan steps (because it is optimal), our heuristic approach is competitive from the standpoint that planning time is vastly reduced.

7.2. Planning graphs in planning under uncertainty

Planning graphs are one of the more important data-structures used within the planning community [11]. The initial use as a search substrate has transitioned to extensive use in heuristic computation. Originally developed for classical planning, they have been extended in a number of ways to handle uncertainty.

PGraphPlan [4] and CGP [44] are the first two planners to use generalizations of GraphPlan [3] for planning under uncertainty. PGraphPlan 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 by constructing a planning graph for each possible world.

One of the first works [6] to use planning graphs for heuristics, more specifically pruning, was in Markov decision processes. By analyzing the propositions reachable in the PGraphPlan planning graph, it is possible to prune some states from a given Markov decision process and improve any number of policy or value iteration algorithms.

Guiding search with planning graph heuristics is a relatively recent line of work. The simplest approach, embodied in Probapop [39], is to treat each outcome of each action as a new deterministic action, and compute classical planning heuristics. Probapop [39], which is built on top of Vhpop [48], guides the Buridan planner with these heuristics. 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.

The next planner to use planning graph heuristics, the Prottle planner [31], uses a variation of PGraphPlan and temporal planning graphs for fully-observable probabilistic temporal planning. In this planning graph, Prottle can explicitly reason about actions with probabilistic outcomes by adding an outcome layer and defining a cost propagation procedure. The authors do not extract relaxed plans, rather they compute an upper bound on goal satisfaction probability.

Most similar to our approach, Domshlak and Hoffmann [16] use a relaxed plan heuristic in the Probabilistic FF (PFF) planner to solve CPP. PFF uses a different representation of the belief state space where it may never completely compute a belief state. Instead, PFF uses a dynamic Bayesian network representation of the plan and performs probabilistic inference with weighted CNFs. Because PFF represents belief states in this fashion, computing its heuristic involves a similar type of reasoning in weighted CNFs. The heuristic computation is relaxed in the same sense as a planning graph; conflicts between action effects are ignored and actions execute in parallel. The reasoning is carried out in weighted CNFs. The heuristic computation involves extra relaxations beyond those made by the $\mathcal{M}cLUG$, such as ignoring all but one antecedent of conditional effects. At the time of writing, the PFF planner was not able to handle probabilistic effects, making empirical comparison impossible.

7.3. Monte Carlo in probabilistic planning

Monte Carlo techniques have long played a significant role in all types of probabilistic reasoning, and probabilistic planning is no different. To our knowledge our approach is the first to use Monte Carlo in heuristic computation, but many have used it within search algorithms and even to determine a representation.

RTDP [1] is a Monte Carlo search algorithm, used in recent work (e.g., Mausam and Weld [36] and Bonet and Geffner [5]). RTDP performs random walks to evaluate the state space, in contrast we use a deterministic search algorithm (albeit guided by a stochastic heuristic).

The Pegasus planner [38] and MCPOMDP [46] use Monte Carlo in POMDP problems to help define a representation. In Pegasus, the approach samples several deterministic versions of a problem's transition system, solves the resulting problems, and uses the results to define an approximate policy for the original problem. The MCPOMDP approach defines policies for large or continuous state spaces by generating reachable approximate belief states through Monte Carlo sampling of the transition and observation functions. With the sampled belief states, MCPOMDP uses policy iteration to define an approximate policy.

8. 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. At a broader level, our work shows one fruitful way of exploiting the recent success in deterministic planning to scale stochastic planners.

While the heuristics are inadmissible, precluding guarantees on plan optimality, we found empirically that the plan quality does not suffer tremendously. By using the heuristics, our planner is able to far out-scale one of the currently best optimal approaches to conformant probabilistic planning. Optimal approaches can sometimes improve the probability of goal satisfaction within the extra steps taken by our plans, but in many domains the optimal planner cannot scale to find such plans.

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) [8]. 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 particle filtering methods 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 [49]. 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 [1,20], 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, an IBM Faculty Award, and the NASA Exploration Technology Development Program. 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] S. Biundo, K.L. Myers, K. Rajan (Eds.), Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5–10, 2005, Monterey, California, USA, AAAI, 2005.
[3] A. Blum, M.L. Furst, Fast planning through planning graph analysis, in: C.S. Mellish (Ed.), Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal, Québec, Canada, August 20–25, 1995, Morgan Kaufmann, 1995, pp. 1636–1642.
[4] A. Blum, J. Langford, Probabilistic planning in the Graphplan framework, in: S. Biundo, M. Fox (Eds.), Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Proceedings, Durham, UK, September 8–10, 1999, in: Lecture Notes in Computer Science, vol. 1809, Springer, 1999, pp. 319–332.
[5] B. Bonet, H. Geffner, Labeled RTDP: Improving the convergence of real-time dynamic programming, in: Giunchiglia et al. [21], pp. 12–31.
[6] C. Boutilier, R.I. Brafman, C.W. Geib, Structured reachability analysis for Markov Decision Processes, in: G.F. Cooper, S. Moral (Eds.), UAI'98: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, July 24–26, 1998, University of Wisconsin Business School, Madison, Wisconsin, USA, Morgan Kaufmann, 1998, pp. 24–32.
[7] R.I. Brafman, J. Hoffmann, Conformant planning via heuristic forward search: A new approach, in: S. Zilberstein, J. Koehler, S. Koenig (Eds.), Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3–7, 2004, Whistler, British Columbia, Canada, AAAI, 2004, pp. 355–364.
[8] J.L. Bresina, R. Dearden, N. Meuleau, S. Ramkrishnan, D.E. Smith, R. Washington, Planning under continuous time and resource uncertainty: A challenge for AI, in: A. Darwiche, N. Friedman (Eds.), UAI '02, Proceedings of the 18th Conference in Uncertainty in Artificial Intelligence, University of Alberta, Edmonton, Alberta, Canada, August 1–4, 2002, Morgan Kaufmann, 2002, pp. 77–84.
[9] D. Bryce, W. Cushing, S. Kambhampati, Model-lite planning: Diverse multi-option plans and dynamic objective functions, in: Proceedings of the 3rd Workshop on Planning and Plan Execution for Real-World Systems, Providence, RI, 2007.
[10] D. Bryce, W. Cushing, S. Kambhampati, Probabilistic planning is multi-objective!, Tech. rep., Arizona State University, ASU CSE TR-07-006 (2007).
[11] D. Bryce, S. Kambhampati, A tutorial on planning graph based reachability heuristics, AI Magazine 28 (1) (2007) 47–83.
[12] D. Bryce, S. Kambhampati, D. Smith, Planning graph heuristics for belief space search, Journal of Artificial Intelligence Research 26 (2006) 35–99.
[13] D. Bryce, S. Kambhampati, D. Smith, Sequential Monte Carlo in probabilistic planning reachability heuristics, in: Long and Smith [33], pp. 233–242.
[14] A. Cassandra, M. Littman, N. Zhang, Incremental pruning: A simple, fast, exact method for Partially Observable Markov Decision Processes, in: Proceedings of the 13th Annual Conference on Uncertainty in Artificial Intelligence (UAI-97), Morgan Kaufmann, San Francisco, CA, 1997, pp. 54–61.
[15] T. Cover, J. Thomas, Elements of Information Theory, Wiley-Interscience, New York, 1991.
[16] C. Domshlak, J. Hoffmann, Fast probabilistic planning through weighted model counting, in: Long and Smith [33], pp. 243–251.
[17] A. Doucet, N. de Freitas, N. Gordon, Sequential Monte Carlo Methods in Practice, Springer, New York, 2001.
[18] R. Fikes, N.J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, in: D.C. Cooper (Ed.), Proceedings of the 2nd International Joint Conference on Artificial Intelligence, London, UK, September 1971, William Kaufmann, 1971, pp. 608–620.
[19] D. Fox, Adapting the sample size in particle filters through KLD-sampling, International Journal of Robotics Research 22 (12) (2003) 985–1003.
[20] 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.
[21] E. Giunchiglia, N. Muscettola, D.S. Nau (Eds.), Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9–13, 2003, Trento, Italy, AAAI, 2003.
[22] J. Hoffmann, R.I. Brafman, Contingent planning via heuristic forward search with implicit belief states, in: Biundo et al. [2], pp. 71–80.
[23] J. Hoffmann, B. Nebel, The FF planning system: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.
[24] J. Huang, Combining knowledge compilation and search for efficient conformant probabilistic planning, in: Long and Smith [33], pp. 253–262.
[25] N. Hyafil, F. Bacchus, Conformant probabilistic planning via CSPs, in: Giunchiglia et al. [21], pp. 205–214.
[26] N. Hyafil, F. Bacchus, Utilizing structured representations and CSP's in conformant probabilistic planning, in: R.L. de Mántaras, L. Saitta (Eds.), Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22–27, 2004, IOS Press, 2004, pp. 1033–1034.
[27] N. Johnson, S. Kotz, N. Balakrishnan, Continuous Univariate Distributions, John Wiley and Sons, New York, 1994.
[28] H.A. Kautz, D.A. McAllester, B. Selman, Encoding plans in propositional logic, in: L.C. Aiello, J. Doyle, S.C. Shapiro (Eds.), Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5–8, 1996, Morgan Kaufmann, 1996, pp. 374–384.

[29] J. Koehler, B. Nebel, J. Hoffmann, Y. Dimopoulos, Extending planning graphs to an ADL subset, in: S. Steel, R. Alami (Eds.), Recent Advances in AI Planning, 4th European Conference on Planning, ECP'97, Toulouse, France, September 24–26, 1997, in: Lecture Notes in Computer Science, vol. 1348, Springer, 1997, pp. 273–285.
[30] N. Kushmerick, S. Hanks, D.S. Weld, An algorithm for probabilistic least-commitment planning, in: B. Hayes-Roth, R. Korf (Eds.), Proceedings of the 12th National Conference on Artificial Intelligence, vol. 2, Seattle, WA, USA, July 31–August 4, 1994, AAAI Press, 1994, pp. 1073–1078.
[31] I. Little, D. Aberdeen, S. Thiébaux, Prottle: A probabilistic temporal planner, in: M.M. Veloso, S. Kambhampati (Eds.), Proceedings of The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9–13, 2005, Pittsburgh, Pennsylvania, USA, AAAI Press/The MIT Press, 2005, pp. 1181–1186.
[32] M. Littman, J. Goldsmith, M. Mundhenk, The computational complexity of probabilistic planning, Journal of Artificial Intelligence Research 9 (1998) 1–36.
[33] D. Long, S. Smith (Eds.), Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006), June 6–10, 2006, The English Lake District, Cumbria, UK, AAAI, 2006.
[34] O. Madani, S. Hanks, A. Condon, On the undecidability of probabilistic planning and infinite-horizon Partially Observable Markov Decision Problems, in: J. Hendler, D. Subramanian (Eds.), Proceedings of the Sixteenth National Conference on Artificial Intelligence, July 18–22, 1999, Orlando, Florida, USA, AAAI Press/The MIT Press, 1999, pp. 541–548.
[35] S.M. Majercik, M.L. Littman, Maxplan: A new approach to probabilistic planning, in: R.G. Simmons, M.M. Veloso, S. Smith (Eds.), Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, Pittsburgh Pennsylvania, USA, AAAI, 1998, pp. 86–93.
[36] Mausam, D.S. Weld, Concurrent probabilistic temporal planning, in: Biundo et al. [2], pp. 120–129.
[37] B. Nebel, On the compilability and expressive power of propositional planning formalisms, Journal of Artificial Intelligence Research 12 (2000) 271–315.
[38] A. Ng, M. Jordan, Pegasus: A policy search method for large MDPs and POMDPs, in: Proceedings of the 16th Annual Conference on Uncertainty in Artificial Intelligence (UAI-00), Morgan Kaufmann, San Francisco, CA, 2000, pp. 406–415.
[39] N. Onder, G. Whelan, L. Li, Engineering a conformant probabilistic planner, Journal of Artificial Intelligence Research 25 (2006) 1–15.
[40] E.P.D. Pednault, ADL and the state-transition model of action, Journal of Logic and Computation 4 (1994) 467–512.
[41] P. Poupart, C. Boutilier, VDCBPI: An approximate scalable algorithm for large scale pomdps, in: Proceedings of NIPS'04, 2004.
[42] M.L. Puterman, Markov Decision Processes—Discrete Stochastic Dynamic Programming, John Wiley & Sons, Inc., New York, 1994.
[43] J. Rintanen, Expressive equivalence of formalisms for planning with sensing, in: Giunchiglia et al. [21], pp. 185–194.
[44] D.E. Smith, D.S. Weld, Conformant Graphplan, in: J. Mostow, C. Rich (Eds.), Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26–30, 1998, Madison, Wisconsin, USA, AAAI Press/The MIT Press, 1998, pp. 889–896.
[45] F. Somenzi, CUDD: CU Decision Diagram Package Release 2.3.0, University of Colorado at Boulder, 1998.
[46] S. Thrun, Monte Carlo POMDPs, in: Advances in Neural Information Processing 12, 2000, pp. 1064–1070.
[47] 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.
[48] H. Younes, R. Simmons, VHPOP: Versatile heuristic partial order planner, Journal of Artificial Intelligence Research 20 (2003) 405–430.
[49] Y. Zemali, P. Fabiani, Search space splitting in order to compute admissible heuristics in planning, in: Workshop on Planen, Scheduling und Konfigurieren, Entwerfen, 2003.