

State Agnostic Planning Graphs: Deterministic, Non-Deterministic, and Probabilistic Planning

Daniel Bryce,^a William Cushing,^b and Subbarao Kambhampati^b

^a *Utah State University, Department of Computer Science
4205 Old Main Hill, Logan, UT 84341*

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

Abstract

Planning graphs have been shown to be a rich source of heuristic information for many kinds of planners. In many cases, planners must compute a planning graph for each element of a set of states, and the naive technique enumerates the graphs individually. This is equivalent to solving a multiple-source shortest path problem by iterating a single-source algorithm over each source.

We introduce a data-structure, the state agnostic planning graph, that directly solves the multiple-source problem for the relaxation introduced by planning graphs. The technique can also be characterized as exploiting the overlap present in sets of planning graphs. For the purpose of exposition, we first present the technique in deterministic planning to capture a set of planning graphs used in forward chaining search. A more prominent application of this technique is in belief state space planning, where each search node utilizes a set of planning graphs; an optimization to exploit state overlap between belief states collapses the set of sets of planning graphs to a single set. We describe another extension in probabilistic planning that reuses planning graph samples of probabilistic action outcomes across search nodes to otherwise curb the inherent prediction cost associated with handling probabilistic actions. Our experimental evaluation (using many existing International Planning Competition problems) quantifies each of these performance boosts, and demonstrates that heuristic belief state space progression planning using our technique is competitive with the state of the art.

Key words: Planning, Heuristics

1 Introduction

Heuristics derived from planning graphs [4] are widespread in planning [19,22,43,33,8]. A planning graph represents a relaxed look-ahead of the state space that identifies

$$\begin{aligned}
P &= \{\text{at}(\text{L1}), \text{at}(\text{L2}), \text{have}(\text{I1}), \text{comm}(\text{I1})\} \\
A &= \{ \text{drive}(\text{L1}, \text{L2}) = (\{\text{at}(\text{L1})\}, \{\{\text{at}(\text{L2})\}, \{\text{at}(\text{L1})\}\}), \\
&\quad \text{drive}(\text{L2}, \text{L1}) = (\{\text{at}(\text{L2})\}, \{\{\text{at}(\text{L1})\}, \{\text{at}(\text{L2})\}\}), \\
&\quad \text{sample}(\text{I1}, \text{L2}) = (\{\text{at}(\text{L2})\}, \{\{\text{have}(\text{I1})\}, \{\}\}), \\
&\quad \text{commun}(\text{I1}) = (\{\text{have}(\text{I1})\}, \{\{\text{comm}(\text{I1})\}, \{\}\}) \} \\
I &= \{\text{at}(\text{L1})\} \\
G &= \{\text{comm}(\text{I1})\}
\end{aligned}$$

Fig. 1. Classical Planning Problem Example.

propositions reachable at different depths. Planning graphs are typically layered graphs of vertices $(\mathcal{P}_0, \mathcal{A}_0, \mathcal{P}_1, \mathcal{A}_1, \dots, \mathcal{A}_{k-1}, \mathcal{P}_k)$, where each level t contains a proposition layer \mathcal{P}_t and an action layer \mathcal{A}_t . Edges between the layers denote the propositions in action preconditions (from \mathcal{P}_t to \mathcal{A}_t) and effects (from \mathcal{A}_{t-1} to \mathcal{P}_t).

In many cases, heuristics are derived from a *set* of planning graphs. In deterministic (classical) planning, progression planners typically compute a planning graph for every search state in order to derive a heuristic cost to reach a goal state. (The same situation arises in planning under uncertainty when calculating the heuristic for a belief state.) A set of planning graphs for related states can be highly redundant. That is, any two planning graphs often overlap significantly. As an extreme example, the planning graph for a child state is a sub-graph of the planning graph of the parent, left-shifted by one step [44]. Computing a set of planning graphs by enumerating its members is, therefore, inherently redundant.

Consider progression planning in a classical planning formulation (P, A, I, G) of a Rovers domain, described in Figure 1. The formulation (discussed in more detail in the next section) defines sets P of propositions, A of actions, I of initial state propositions, G of goal propositions. In the problem, there are two locations, L1 and L2, and an image I1 can be taken at L2. The goal is to achieve $\text{comm}(\text{I1})$, having communicated the image back to a lander. There are four actions $\text{drive}(\text{L1}, \text{L2})$, $\text{drive}(\text{L2}, \text{L1})$, $\text{sample}(\text{I1}, \text{L2})$, and $\text{commun}(\text{I1})$. The rover can use the plan: $(\text{drive}(\text{L1}, \text{L2}), \text{sample}(\text{I1}, \text{L2}), \text{commun}(\text{I1}))$ to achieve the goal. The state sequence corresponding to this plan is:

$$\begin{aligned}
s_I &= \{\text{at}(\text{L1})\} \\
s_1 &= \{\text{at}(\text{L2})\} \\
s_2 &= \{\text{at}(\text{L2}), \text{have}(\text{I1})\} \\
s_3 &= \{\text{at}(\text{L2}), \text{have}(\text{I1}), \text{commun}(\text{I1})\}
\end{aligned}$$

Notice that $s_1 \subset s_2 \subset s_3$, meaning that the planning graphs for each state will have initial proposition layers where $\mathcal{P}_0(s_1) \subset \mathcal{P}_0(s_2) \subset \mathcal{P}_0(s_3)$. Further, many

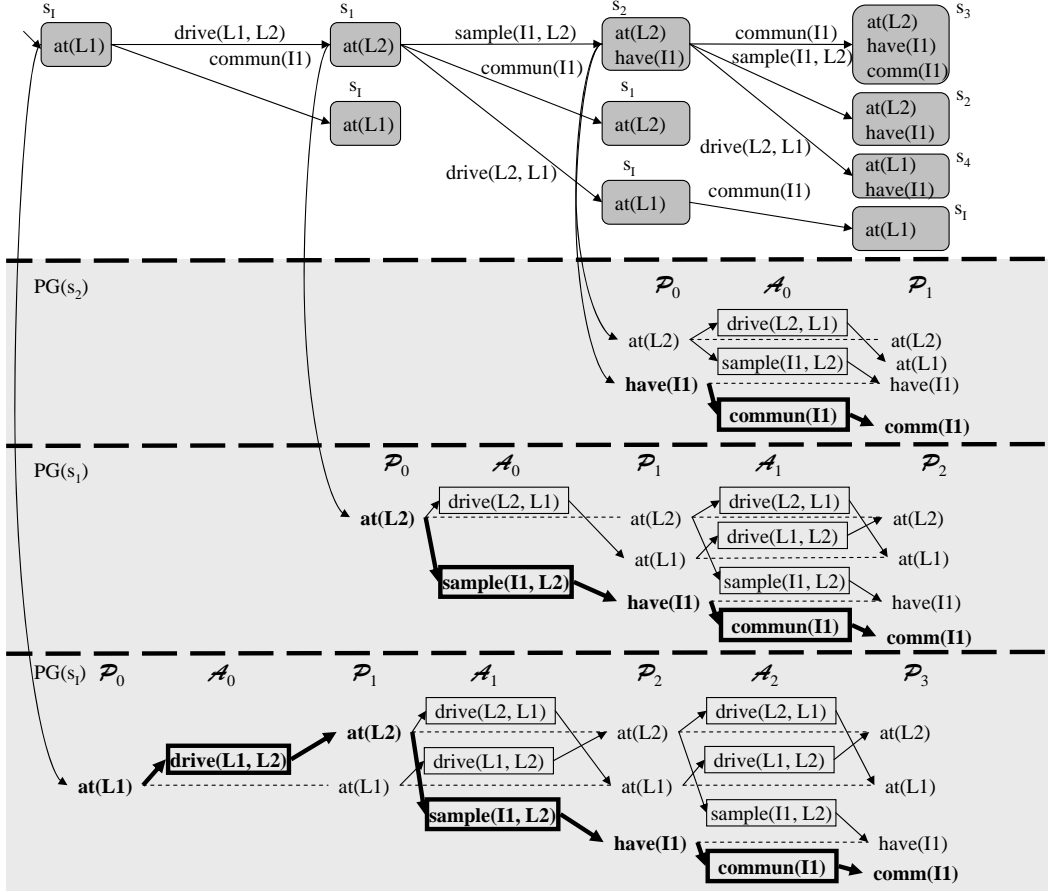


Fig. 2. Planning graphs and state space projection tree.

of the same actions appear in the first action layer of the planning graph for each state. Figure 2 (described in detail below) depicts the search tree (top) and planning graphs for several states (bottom).

State Agnostic Planning Graphs: Avoiding the redundant construction and representation of search heuristics as much as possible can improve planner scalability. Our answer to avoiding redundancy is a generalization of the planning graph called the State Agnostic Graph (SAG). The general technique (of which we will describe several variations) is to implicitly represent several planning graphs by a single planning graph skeleton that captures action and proposition connectivity (for preconditions and effects) and use propositional sentences, called labels, to annotate which portions of the skeleton relate to which of the explicit planning graphs. That is, any explicit planning graph from the set can be recovered by inspecting the labeled planning graph skeleton. Moreover, when the need to reason about sets of planning graphs arises, it is possible to perform the reasoning symbolically without materializing each of the explicit graphs. Our techniques are related to work on assumption based truth maintenance systems [15], where the intent is to capture common assumptions made in multiple contexts. The contributions of this work are to identify several extensions of this idea to reachability heuristics across a set of

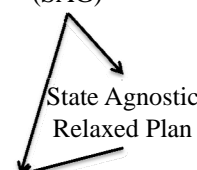
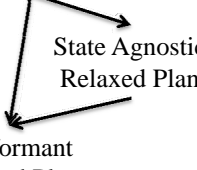
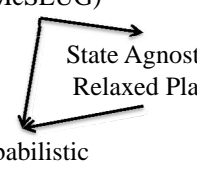
	Deterministic (Classical) Planning	Non-Deterministic Planning (Incomplete Initial State)	Probabilistic Planning (Probabilistic Initial State, Probabilistic Actions)
Traditional (Per Search Node) Planning Graph	Planning Graph (PG) ↓ Relaxed Plan	Labeled Planning Graph (LUG) ↓ Conformant Relaxed Plan	Monte Carlo Labeled Planning Graph (McLUG) ↓ Probabilistic Conformant Relaxed Plan
State Agnostic (Per Instance) Planning Graph	State Agnostic Planning Graph (SAG)  State Agnostic Relaxed Plan Relaxed Plan	State Agnostic Labeled Planning Graph (SLUG)  State Agnostic Relaxed Plan Conformant Relaxed Plan	Monte Carlo State Agnostic Labeled Planning Graph (McSLUG)  State Agnostic Relaxed Plan Probabilistic Conformant Relaxed Plan

Fig. 3. Taxonomy of Planning Graphs and Relaxed Plans.

planning problems.

From a graph-theoretic perspective, it is possible to view the planning graph as exactly solving a single-source shortest path problem, for a relaxed planning problem. The levels of the graph efficiently represent a breadth-first sweep from the single source. In the context of progression planning, the planner will end up calculating a heuristic for many different sources. Iterating a single-source algorithm over each source (building a planning graph per search node) is a naive solution to the multiple-source shortest path problem. We develop the SAG under the following intuition: directly solving the multiple-source shortest path problem is more efficient than iterating a single source algorithm.

The exact form of the SAG depends upon the underlying properties (mutexes, cost, time, ...) of the planning graphs being represented. The main insight to the technique is to identify the individual planning graphs by propositional models and represent the propagation rules of the planning graphs as the composition of propositional sentences (labels). Composing these sentences via boolean algebra yields a symbolic approach for building the set of planning graphs without explicitly enumerating its elements. The labels exploit redundant sub-structure, and can help boost empirical performance.

Figure 3 outlines the SAG techniques discussed in this work. The first row shows

the types of planning graphs and heuristics computed when using a traditional non-SAG approach that constructs a new planning graph or set of planning graphs at each search node. The second row shows the type of SAG and techniques for computing heuristics in each of the three problem classes. In each problem class, the corresponding version of the SAG represents a set of all traditional planning graphs required for a given problem instance. In deterministic (a.k.a. classical) planning the SAG captures a set of planning graphs; in non-deterministic (a.k.a. conformant) planning, a set of labeled planning graphs (*LUG*) [9]; and in probabilistic (a.k.a. probabilistic conformant) planning, a set of Monte Carlo labeled planning graphs (*McLUG*) [10].¹ We overload the term SAG to refer to both the specific generalization of the planning graph used in deterministic planning and the general technique of representing all planning graphs for a given instance with a single data structure. The SAG approach applied to non-deterministic planning results in a data-structure called the state agnostic *LUG* (*SLUG*) and applied to probabilistic planning, the Monte Carlo state agnostic *LUG* (*McSLUG*). In all types of problems there are two approaches to computing relaxed plan heuristics from the SAG: extracting a relaxed plan for each search node, or prior to search extracting a state agnostic relaxed plan (representing all relaxed plans) and then for each node extracting a relaxed plan from the state agnostic relaxed plan.

Labels are propositional sentences whose models refer to individual planning graphs. That is, a labeled planning graph element would be in the explicit planning graph corresponding to each model of the label. In deterministic planning, each planning graph is uniquely identified by the source state from which it is built; thus, each label model corresponds to a state. In non-deterministic planning, each *LUG* is uniquely identified by the source belief state from which it is built; however, the *LUG* itself is a set of planning graphs, one for each state in a belief state. Instead of representing a set of *LUG*, the *SLUG* represents the union of planning graphs present in each *LUG*. The *SLUG* labels, like the SAG for deterministic planning, have models that correspond to states. In probabilistic planning, where actions with probabilistic effects are the challenge, the *McLUG* represents a set of deterministic planning graphs, each obtained by sampling the action outcomes in each level. The *McSLUG* represents a set of *McLUG*, and each label model refers to both a state and a set of sampled action outcomes. The *McSLUG* uses an additional optimization that reuses action outcome samples among planning graphs built for different states to keep the number of action outcome samples independent of the number of states.

The idea to represent a set of planning graphs symbolically via labeling originally appears in our work on the *LUG* [9]. The idea of sampling a set of planning graphs

¹ Our discussion is mainly focussed on planning problems with sequential (non-conditional) plans, but the heuristics discussed have been successfully applied to the analogous non-deterministic and probabilistic conditional planning problems [7,6,9]. The focus of this paper is on how to compute the heuristics more efficiently with the SAG.

(and representing them with labels) in probabilistic planning originally appears in our work on the *McLUG* [11]. The work described herein reinterprets the use of labels to compute the planning graphs for all search nodes (states or belief states), not just a set of planning graphs needed to compute the heuristic for a single search node. The important issues addressed by this work (beyond those of the previous work) are: i) defining a semantics for labels that support heuristic computation for all search nodes and ii) evaluating whether precomputing all required planning graphs is more effective than computing the planning graphs for individual search nodes. The SAG was also previously described in a preliminary version of this work [13], and the primary contributions described herein relate to i) extending the SAG to the probabilistic setting, ii) introducing a new method to compute relaxed plans by symbolically pre-extracting a relaxed plan from each planning graph represented by a SAG (collectively called the SAG relaxed plan) and iii) presenting additional empirical evaluation, including International Planning Competition (IPC) results.

In addition to the IPC results, our empirical evaluation internally evaluates the performance of our planner *POND* while using traditional planning graphs, versus the SAG and the SAG relaxed plan to compute relaxed plan heuristics. Additional external evaluations compare *POND* to the following state-of-the-art planners: Conformant FF [21], t0 [35], BBSP [39], KACMBP [1], MBP [2], CPplan [25], and Probabilistic FF [16].

Layout: Our presentation describes traditional planning graphs and their generalization to state agnostic planning graphs for deterministic planning (Section 2), non-deterministic planning (Section 3), and probabilistic planning (Section 4). In Section 5 we explore a generalization of relaxed plan heuristics that follows directly from the SAG, namely, the state agnostic relaxed plan, which captures the relaxed plan for every state. From there, the experimental evaluation (Section 6.2) begins by comparing these strategies internally. We then conduct an external comparison in Section 6.3 with several belief state space planners to demonstrate that our planner *POND* is competitive with the state of the art in both non-deterministic planning and probabilistic planning. We finish with a discussion of related work in Section 7 and a conclusion in Section 8.

2 Deterministic Planning

This section provides a brief background on deterministic (classical) planning, an introduction to deterministic planning graphs, and a first discussion of state agnostic graphs.

2.1 Problem Definition

As previously stated, the classical planning problem defines the 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. 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 set of states S is the power set of P , such that $S = 2^P$. The initial state s_I is specified by a set of propositions $I \subseteq P$ known to be true and the goal is a set of propositions $G \subseteq P$ that must be made true in a goal state. Each action $a \in A$ is described by $(\rho_e(a), (\varepsilon^+(a), \varepsilon^-(a)))$, where the execution precondition $\rho_e(a)$ is the set of propositions, and $(\varepsilon^+(a), \varepsilon^-(a))$ is an effect where $\varepsilon^+(a)$ is the set of propositions that a causes to become true and $\varepsilon^-(a)$ is a set of propositions a 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 $s' = exec(a, s) = s \setminus \varepsilon^-(a) \cup \varepsilon^+(a)$. A sequence of actions (a_1, \dots, a_m) , executed in state s , results in a state s' , where $s' = exec((a_1, \dots, a_m), s) = exec(a_m, exec(a_{m-1}, \dots, exec(a_1, s) \dots))$ and each action is applicable in the appropriate state. A valid plan is a sequence of actions that is applicable in s_I and results in a goal state. The number of actions is the cost of the plan. Our discussion below will make use of the equivalence between set and propositional logic representations of states. Namely, a state $s = \{p_1, \dots, p_n\} \subseteq P$ represented in set notation is equivalent to a logical state $\hat{s} = p_1 \wedge \dots \wedge p_n \wedge \neg p_{n+1} \wedge \dots \wedge \neg p_m$, where $P \setminus s = \{p_{n+1}, \dots, p_m\}$.

The example problem description in Figure 1 lists four actions, in terms of their execution precondition and effects; the `drive(L1, L2)` action has the execution precondition `at(L1)`, causes `at(L2)` to become true, and causes `at(L1)` to become false. Executing `drive(L1, L2)` in the initial state (which is state s_I , in the example) results in the state: $s_1 = exec(drive(L1, L2), s_I) = \{at(L2)\}$. The state s_1 can be represented as the logical state $\hat{s}_1 = \neg at(L1) \wedge at(L2) \wedge \neg have(I1) \wedge \neg comm(I1)$. In the following, we drop the distinction between set (s) and logic notation (\hat{s}) because the context will dictate the appropriate representation.

One of the most popular state space search formulations, progression, creates a projection tree (Figure 2) 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 cost to *reach* a goal state from each state to focus effort on expanding the least cost leaf nodes.

2.2 Planning Graphs

One effective technique to compute reachability heuristics is through planning graph analysis. Traditionally, progression search uses a different planning graph to compute the reachability heuristic for each state s (see Figure 2). A planning graph $PG(s, A)$ constructed for the state s (referred to as the source state) and the action set A is a leveled graph, captured by layers of vertices ($\mathcal{P}_0(s), \mathcal{A}_0(s), \mathcal{P}_1(s), \mathcal{A}_1(s), \dots, \mathcal{A}_{k-1}(s), \mathcal{P}_k(s)$), where each level t consists of a proposition layer $\mathcal{P}_t(s)$ and an action layer $\mathcal{A}_t(s)$. 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 (unless otherwise stated) for action layers \mathcal{A}_t and proposition layers \mathcal{P}_t also assumes that the state s is implicit. The specific type of planning graph that we discuss is the relaxed planning graph [22]; in the remainder of this work we drop the terminology “relaxed”, because all planning graphs discussed are relaxed.

A planning graph, $PG(s)$, built for a single source s , satisfies the following:

- (1) $p \in \mathcal{P}_0$ iff $p \in s$
- (2) $a \in \mathcal{A}_t$ iff $p \in \mathcal{P}_t$, for every $p \in \rho_e(a)$
- (3) $p \in \mathcal{P}_{t+1}$ iff $p \in \varepsilon^+(a)$ and $a \in \mathcal{A}_t$

The first proposition layer, \mathcal{P}_0 , is defined as the set of propositions in the state s . An action layer \mathcal{A}_t consists of all actions that have all of their precondition propositions in \mathcal{P}_t . A proposition layer \mathcal{P}_t , $t > 0$, is the set all propositions given by the positive effect of an action in \mathcal{A}_{t-1} . It is common to use implicit actions for proposition persistence (a.k.a. noop actions) to ensure that propositions in \mathcal{P}_t persist to \mathcal{P}_{t+1} . A noop action a_p for proposition p is defined as $\rho_e(a_p) = \varepsilon^+(a_p) = p$. Planning graph construction continues until the goal is reachable (i.e., every goal proposition is present in a proposition layer) or the graph reaches level-off (two proposition layers are identical). (The index of the level where the goal is reachable can be used as an admissible heuristic, called the level heuristic.)

Figure 2 shows three examples of planning graphs for different states encountered within the projection tree. For example, $PG(s_I)$ has `at (L1)` in its initial proposition layer. The `at (L1)` proposition is connected to the `i) drive (L1, L2)` action because it is a precondition, and ii) connected to a persistence action (shown as a dashed line). The `drive (L1, L2)` action is connected to `at (L2)` because it is a positive effect of the action.

Consider one of the most popular and effective heuristics, which is based on relaxed plans [22]. Through a simple back-chaining algorithm (Figure 4) called *relaxed plan extraction*, it is possible to identify actions in each level that are needed to causally support the goals. Relaxed plans are subgraphs $(\mathcal{P}_0^{RP}, \mathcal{A}_0^{RP}, \mathcal{P}_1^{RP}, \dots, \mathcal{A}_{k-1}^{RP}, \mathcal{P}_k^{RP})$ of the planning graph, where each layer corresponds to a set of vertices. A relaxed plan captures the causal chains involved in supporting the goals, but ignores how


```

RPEExtract( $PG(s), G$ )
1: Let  $k$  be the index of the last level of  $PG(s)$ 
2: for all  $p \in G \cap \mathcal{P}_k$  do {Initialize Goals}
3:    $\mathcal{P}_k^{RP} \leftarrow \mathcal{P}_k^{RP} \cup p$ 
4: end for
5: for  $t = k \dots 1$  do
6:   for all  $p \in \mathcal{P}_t^{RP}$  do {Find Supporting Actions}
7:     Find  $a \in \mathcal{A}_{t-1}$  such that  $p \in \varepsilon^+(a)$ 
8:      $\mathcal{A}_{t-1}^{RP} \leftarrow \mathcal{A}_{t-1}^{RP} \cup a$ 
9:   end for
10:  for all  $a \in \mathcal{A}_{t-1}^{RP}, p \in \rho_e(a)$  do {Insert Preconditions}
11:     $\mathcal{P}_{t-1}^{RP} \leftarrow \mathcal{P}_{t-1}^{RP} \cup p$ 
12:  end for
13: end for
14: return  $(\mathcal{P}_0^{RP}, \mathcal{A}_0^{RP}, \mathcal{P}_1^{RP}, \dots, \mathcal{A}_{k-1}^{RP}, \mathcal{P}_k^{RP})$ 

```

Fig. 4. Relaxed Plan Extraction Algorithm.

actions may conflict.

Figure 4 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 k 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 that selects an action to support a proposition. 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 also possible to select actions based on other criterion, such as the index of the first action layer where they appear. 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 a heuristic as the total number of non-noop actions in the action layers.

Figure 2 depicts relaxed plans in bold for each of the states. The relaxed plan for s_I has three actions, giving the state an h-value of three. Likewise, s_1 has a h-value of two, and s_2 , one.

2.3 State Agnostic Planning Graphs

We generalize the planning graph to the SAG, by associating a label $\ell_t(\cdot)$ with each action and proposition at each level of the graph. A label tracks the set of sources reaching the associated action or proposition at level t . That is, if the planning graph

built from a source includes a proposition at level t , then the SAG also includes the proposition at level t and labels it to denote it is reachable from the source. Each label is a propositional sentence over state propositions P whose models correspond to source states (i.e., exactly those source states reaching the labeled element). Intuitively, a source state s reaches a graph element x if $s \models \ell_t(x)$, the state is a model of the label at level t . The set of possible sources is defined by the scope of the SAG, denoted \mathcal{S} , that is also a propositional sentence. Each SAG element label $\ell_t(x)$ denotes a subset of the scope, meaning that $\ell_t(x) \models \mathcal{S}$. A conservative scope $\mathcal{S} = \top$ would result in a SAG built for all states (each state is a model of logical true).

The graph $SAG(\mathcal{S}) = \langle (\mathcal{P}_0, \mathcal{A}_0, \dots, \mathcal{A}_{k-1}, \mathcal{P}_k), \ell \rangle$ is defined similar to a planning graph, but additionally defines a label function ℓ and is constructed with respect to a scope \mathcal{S} . For each source state s where $s \models \mathcal{S}$, the SAG satisfies:

- (1) $s \models \ell_0(p)$ iff $p \in s$
- (2) $s \models \ell_t(a)$ iff $s \models \ell_t(p)$ for every $p \in \rho_e(a)$
- (3) $s \models \ell_{t+1}(p)$ iff $s \models \ell_t(a)$ and $p \in \varepsilon^+(a)$

This definition resembles that of the planning graph, with the exception that labels dictate which propositions and actions are included in various levels.

There are several ways to construct the SAG to satisfy the definition above. An explicit (naive) approach might enumerate the source states, build a planning graph for each, and define the label function for each graph vertex as the disjunction of all states whose planning graph contains the vertex (i.e., $\ell_t(p) = \bigvee_{p \in \mathcal{P}_t(s)} s$). Enumerating the states to construct the SAG is clearly worse than building a planning graph for each state. A more practical approach would not enumerate the states (and their corresponding planning graphs) to construct the SAG. We use the intuition that actions appear in all planning graph action layers where *all of their preconditions hold* in the preceding proposition layer (a conjunction, see 2. below), and that propositions appear in all planning graph proposition layers where *there exists an action giving it as an effect* in the previous action layer (a disjunction, see 3. below). It is possible to implicitly define the SAG, using the following rules:

- (1) $\ell_0(p) = \mathcal{S} \wedge p$
- (2) $\ell_t(a) = \bigwedge_{p \in \rho_e(a)} \ell_t(p)$
- (3) $\ell_t(p) = \bigvee_{a: p \in \varepsilon^+(a)} \ell_{t-1}(a)$,
- (4) $k = \text{minimum level } t \text{ such that } (\mathcal{P}_t = \mathcal{P}_{t+1}) \text{ and } \ell_t(p) = \ell_{t+1}(p), p \in \mathcal{P}_t$

Figure 5 depicts the SAG for the example (Figure 1), where $\mathcal{S} = \top$ (the set of all states is represented by the logical true \top). The figure denotes the labels by propositional formulas in italics above actions and propositions. By the third level, the goal proposition $\text{comm}(\text{I1})$ is labeled $\ell_3(\text{comm}(\text{I1})) = \text{at}(\text{L1}) \vee$

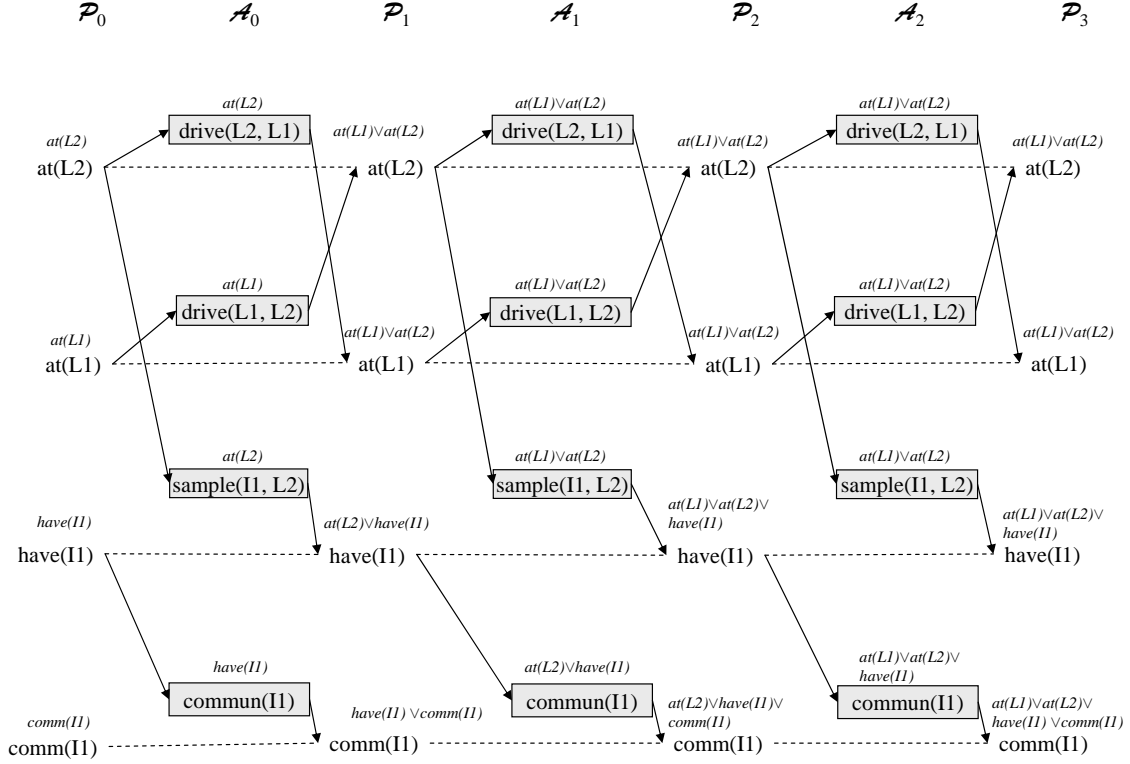


Fig. 5. SAG for rover example.

$at(L2) \vee have(I1) \vee comm(I1)$. The goal is reachable by every state, except $s = \neg at(L1) \wedge \neg at(L2) \wedge \neg have(I1) \wedge \neg comm(I1)$ because $s \not\models \ell_3(comm(I1))$. The state s will never reach the goal because level three is identical to level four (not shown) and $s \not\models \ell_4(comm(I1))$, meaning the heuristic value for s is provably ∞ . Otherwise, the heuristic value for each state s (where $s \models \mathcal{S}$) is at least $\min_{0 \leq t \leq k} t$ where $s \models \ell_t(p)$ for each $p \in G$. This lower bound is known as the level heuristic.

Extracting the relaxed plan for a state s from the SAG is almost identical to extracting a relaxed plan from a planning graph built for state s . The primary difference is that while a proposition in some SAG layer \mathcal{P}_t may have supporting actions in the preceding action layer \mathcal{A}_{t-1} , not just any action can be selected for the relaxed plan. We require that $s \models \ell_{t-1}(a)$ to guarantee that the action would appear in \mathcal{A}_{t-1} in the planning graph built for state s . For example, to evaluate the relaxed plan heuristic for state $s_1 = \neg at(L1) \wedge at(L2) \wedge \neg have(I1) \wedge \neg comm(I1)$, we may mistakenly try to support $comm(I1)$ in \mathcal{P}_1 with $commun(I1)$ in \mathcal{A}_0 without noticing that $commun(I1)$ does not appear until action layer \mathcal{A}_1 for state

s_1 (i.e., $s_1 \not\models \ell_0(\text{comm}(\text{I1}))$, but $s_1 \models \ell_1(\text{comm}(\text{I1}))$). Ensuring that supporting actions have the appropriate state represented by their label guarantees that the relaxed plan extracted from the SAG is identical to the relaxed plan extracted from a normal planning graph. The change to the relaxed plan extraction procedure (in Figure 4) replaces line 7 with:

“Find $a \in \mathcal{A}_{t-1}$ such that $p \in \varepsilon^+(a)$ and $s \models \ell_{t-1}(a)$ ”,

adding the underlined portion.

Sharing: The graph, $SAG(\mathcal{S})$, is built once for a set of states represented by \mathcal{S} . For any s such that $s \models \mathcal{S}$, computing the heuristic for s reuses the shared graph $SAG(\mathcal{S})$. For example, it is possible to compute the level heuristic for every state in the rover problem, by finding the first level t where the state is a model of $\ell_t(\text{comm}(\text{I1}))$. Any state s where $\text{comm}(\text{I1}) \in s$ has a level heuristic of zero because $\ell_0(\text{comm}(\text{I1})) = \text{comm}(\text{I1})$. Any state s , where $\text{comm}(\text{I1}) \in s$ or $\text{have}(\text{I1}) \in s$, has a level heuristic of one because $\ell_1(\text{comm}(\text{I1})) = \text{comm}(\text{I1}) \vee \text{have}(\text{I1})$, and so on for states modeling the labels of the goal proposition in levels two and three. It is possible to compute the heuristic values *a priori*, or on-demand during search. In Section 5 we will discuss various points along the continuum between computing all heuristic values before search and computing the heuristic at each search node.

The search tree for the rover problem has a total of six unique states. By constructing a planning graph for each state, the total number of planning graph vertices (for propositions and actions) that must be allocated is 56. Constructing the equivalent SAG, while representing the planning graphs for extra states, requires only 28 planning graph vertices. There are a total of 10 unique propositional sentences, with at most four propositions per function. To answer the question of whether the SAG is more efficient than building individual planning graphs, we must consider the cost of manipulating labels. Representing labels as boolean functions (e.g., with BDDs) significantly reduces both the size and cost of reasoning with labels, making the SAG a better choice in some cases. We must also consider whether the SAG is used enough by the search: it may be too costly to build the SAG for all states if we only evaluate the heuristic for relatively few states. We will address these issues empirically in the evaluation section, but first consider the SAG for non-deterministic planning.

3 Non-Deterministic Planning

This section extends the deterministic planning model to consider non-deterministic planning with an incomplete initial state, deterministic actions, and no observabil-

ity.² The section follows with an approach to planning graph heuristics for search in belief state space, and ends with a SAG generalization of the planning graph heuristics.

3.1 Problem Definition

The non-deterministic planning problem is given by (P, A, b_I, G) where, as in classical planning, P is a set of propositions, A is a set of actions, and G is a goal description. Extending the classical model, the initial state is replaced by an initial belief state b_I . Belief states capture incomplete information by representing a set of all states consistent with the information. A non-deterministic belief state describes a boolean function $b : S \rightarrow \{0, 1\}$, where $b(s) = 1$ if $s \in b$ and $b(s) = 0$ if $s \notin b$. For example, the problem in Figure 6 indicates that there are two states in b_I , denoting that it is unknown if $\text{have}(\text{I1})$ holds. We also make use of a logical representation of belief states, where a state \hat{s} is in a belief state \hat{b} if $\hat{s} \models \hat{b}$. From the example, $\hat{b}_I = \text{at}(\text{L1}) \wedge \neg \text{at}(\text{L2}) \wedge \neg \text{comm}(\text{I1})$. As with states, we drop the distinction between the set and logic representation because the context dictates the representation.

In classical planning it is often sufficient to describe actions by their execution precondition and positive and negative effects. With incomplete information, it is convenient to describe actions that have context-dependent (conditional) effects. (Our notation also allows for multiple action outcomes, which we will adopt when discussing probabilistic planning. We do not consider actions with uncertain effects in non-deterministic planning.) In non-deterministic planning, an action $a \in A$ is a tuple $(\rho_e(a), \Phi(a))$, where $\rho_e(a)$ is an enabling precondition and $\Phi(a)$ is a set of causative outcomes (in this case there is only one outcome). The enabling precondition $\rho_e(a)$ is a set of propositions that determines the states in which an action is applicable. An action a is applicable $\text{appl}(a, s)$ to state s if $\rho_e(a) \subseteq s$, and it is applicable $\text{appl}(a, b)$ to a belief state b if for each state $s \in b$ the action is applicable.

Each causative outcome $\Phi_i(a) \in \Phi(a)$ is a set of conditional effects. 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) $\rho_{ij}(a)$, the positive consequent $\varepsilon_{ij}^+(a)$, and the negative consequent $\varepsilon_{ij}^-(a)$ are a set of propositions. Actions are assumed to be consistent, meaning that for each $\Phi_i(a) \in \Phi(a)$ each pair of conditional effects $\varphi_{ij}(a)$ and $\varphi_{i'j'}(a)$ have consequents such that $\varepsilon_{ij}^+(a) \cap \varepsilon_{i'j'}^-(a) = \emptyset$ if there is a state s where both may execute (i.e., $\rho_{ij}(a) \subseteq s$ and $\rho_{i'j'}(a) \subseteq s$). In other words, no two

² With minor changes in notation, the heuristics described in this section apply to unrestricted non-deterministic planning (i.e., non-deterministic actions and partial observability), but only under the relaxation that all non-deterministic action outcomes occur and observations are ignored.

$$\begin{aligned}
P &= \{\text{at}(\text{L1}), \text{at}(\text{L2}), \text{have}(\text{I1}), \text{comm}(\text{I1})\} \\
A &= \{ \text{drive}(\text{L1}, \text{L2}) = (\{\text{at}(\text{L1})\}, \{(\{\{\} \rightarrow (\{\text{at}(\text{L2})\}, \{\text{at}(\text{L1})\})\})\}), \\
&\quad \text{drive}(\text{L2}, \text{L1}) = (\{\text{at}(\text{L2})\}, \{(\{\{\} \rightarrow (\{\text{at}(\text{L1})\}, \{\text{at}(\text{L2})\})\})\}), \\
&\quad \text{sample}(\text{I1}, \text{L2}) = (\{\text{at}(\text{L2})\}, \{(\{\{\} \rightarrow (\{\text{have}(\text{I1})\}, \{\})\})\}), \\
&\quad \text{commun}(\text{I1}) = (\{\}, \{(\{\{\text{have}(\text{I1})\} \rightarrow (\{\text{comm}(\text{I1})\}, \{\})\})\}) \\
b_I &= \{\{\text{at}(\text{L1}), \text{have}(\text{I1})\}, \{\text{at}(\text{L1})\}\} \\
G &= \{\text{comm}(\text{I1})\}
\end{aligned}$$

Fig. 6. Non-Deterministic Planning Problem Example.

conditional effects of the same outcome can have consequents that disagree on a proposition if both effects are applicable. This representation of effects follows the 1ND normal form [38]. For example, the `commun(I1)` action in Figure 6 has a single outcome with a single conditional effect $\{\text{have}(\text{I1})\} \rightarrow (\{\text{comm}(\text{I1})\}, \{\})$. The `commun(I1)` action is applicable in b_I , and its conditional effect occurs only if `have(I1)` is true.

It is possible to use the effects of every action to derive a state transition function $T(s, a, s')$ that defines a possibility that executing a in state s will result in state s' . With deterministic actions, executing action a in state s will result in a single state s' :

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

This defines the possibility of transitioning from state s to s' by executing a as $T(s, a, s') = 1$ if there exists an outcome $\Phi_i(a)$ where $s' = \text{exec}(\Phi_i(a), s)$, and $T(s, a, s') = 0$, otherwise.

Executing action a in belief state b , denoted $\text{exec}(a, b) = b_a$, defines the successor belief state b_a as $b_a(s') = \max_{s \in b} b(s)T(s, a, s')$. Executing `commun(I1)` in b_I results in the belief state $\{\{\text{at}(\text{L1}), \text{have}(\text{I1}), \text{comm}(\text{I1})\}, \{\text{at}(\text{L1})\}\}$, indicating that the goal is satisfied in one of the states, assuming `have(I1)` was true before execution.

The result b' of executing a sequence of actions (a_1, \dots, a_m) in belief state b_I is defined as $b' = \text{exec}((a_1, \dots, a_m), b_I) = \text{exec}(a_m, \dots, \text{exec}(a_2, \text{exec}(a_1, b_I))\dots)$. A sequence of actions is a *strong* plan if every state in the resulting belief state is a goal state, $\forall_{s \in b'} G \subseteq s$. Another way to state the strong plan criterion is to say that the plan will guarantee goal satisfaction irrespective of the initial state (i.e., for each $s \in b_I$, let $b' = \text{exec}((a_1, \dots, a_m), \{s\})$, then $\forall_{s' \in b'} G \subseteq s'$). Under this second view of strong plans, it becomes apparent how one might derive planning graph heuristics: use a deterministic planning graph to compute the cost to reach the goal from each state in a belief state and then aggregate the costs [9]. As we will see, surmounting the possibly exponential number of states in a belief state is

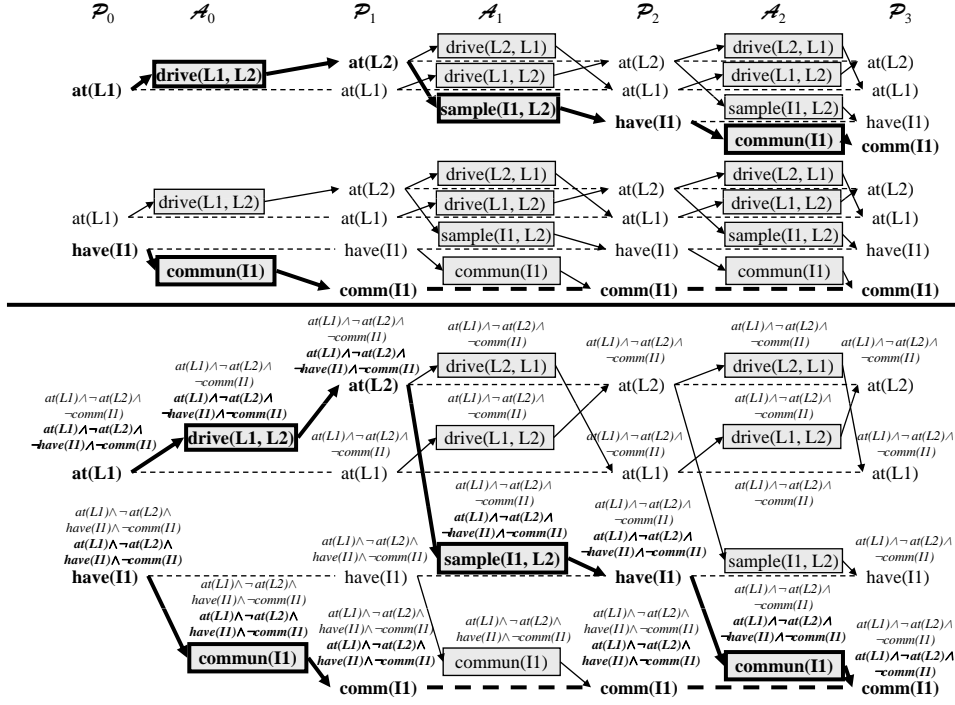


Fig. 7. Multiple planning graphs and LUG.

the challenge to deriving such a heuristic.

3.2 Planning Graphs

It is possible to compute a heuristic for a belief state by constructing a planning graph for each state in the belief state, extracting a relaxed plan from each planning graph, and aggregating the heuristic values [9]. For example, the top portion of Figure 7 shows two planning graphs, each built for a different state in b_I of our example. The bold subgraphs indicate the relaxed plans, which can be aggregated to compute a heuristic. While this multiple planning graph approach can provide informed heuristics, it can be quite costly when there are several states in the belief state; plus, there is a lot of repeated planning graph structure among the multiple planning graphs. Using multiple planning graphs for search in the belief state space exacerbates the problems faced in state space (classical) planning; not only is there planning graph structure repetition between search nodes, but also among the planning graphs used for a single search node.

The solution to repetition within a search node is addressed with the labeled (uncertainty) planning graph (LUG). The LUG represents a search node's multiple explicit planning graphs implicitly. The planning graph at the bottom of Figure 7

shows the labeled planning graph representation of the multiple planning graphs at the top. The *LUG* uses labels, much like the *SAG* in deterministic planning. The difference between the *LUG* and the *SAG* is that the *LUG* is used to compute the heuristic for a single search node (that has multiple states) and the *SAG* is used to compute the heuristics for multiple search nodes (each a state). The construction semantics is almost identical, but the heuristic computation is somewhat different.

The *LUG* is based on the *IPP* [27] planning graph, in order to explicitly capture conditional effects, and extends it to represent multiple state causal support (as present in multiple graphs) by adding labels to actions, effects, and propositions.³ The *LUG*, built for a belief state b (similar to a deterministic *SAG* with scope $\mathcal{S} = b$), is a set of vertices and a label function: $LUG(b) = \langle (\mathcal{P}_0, \mathcal{A}_0, \mathcal{E}_0, \dots, \mathcal{A}_{k-1}, \mathcal{E}_{k-1}, \mathcal{P}_k), \ell \rangle$. A label $\ell_t(\cdot)$ denotes a set of states (a subset of the states in belief state b) from which a graph vertex is *reachable*. In other words, the explicit planning graph for each state represented in the label would contain the vertex at the same level. A proposition p is reachable from all states in b after t levels if $b \models \ell_t(p)$ (i.e., each state model of the belief state is a model of the label).

For every $s \in b$, the following holds:

- (1) $s \models \ell_0(p)$ iff $p \in s$
- (2) $s \models \ell_t(a)$ iff $s \models \ell_t(p)$ for every $p \in \rho_e(a)$
- (3) $s \models \ell_t(\varphi_{ij}(a))$ iff $s \models \ell_t(a) \wedge \ell_t(p)$ for every $p \in \rho_{ij}(a)$
- (4) $s \models \ell_{t+1}(p)$ iff $p \in \varepsilon_{ij}^+(a)$ and $s \models \ell_t(\varphi_{ij}^+(a))$

Similar to the intuition for the *SAG* in deterministic planning, the following rules can be used to construct the *LUG*:

- (1) $\ell_0(p) = b \wedge p$
- (2) $\ell_t(a) = \bigwedge_{p \in \rho_e(a)} \ell_t(p)$
- (3) $\ell_t(\varphi_{ij}(a)) = \ell_t(a) \wedge \left(\bigwedge_{p \in \rho_{ij}(a)} \ell_t(p) \right)$
- (4) $\ell_t(p) = \bigvee_{a: p \in \varepsilon_{ij}^+(a)} \ell_{t-1}(\varphi_{ij}(a))$,
- (5) $k = \text{minimum level } t \text{ where } b \models \left(\bigwedge_{p \in G} \ell_t(p) \right)$

For the sake of illustration, Figure 7 depicts a *LUG* without the effect layers. Each of the actions in the example problem have only one effect, so the figure only depicts actions if they have an enabled effect (i.e., both the execution precondition and secondary precondition are reachable from some state $s \in b$). In the figure, there are

³ Like the deterministic planning graph, the *LUG* includes noop actions. Using the notation for conditional effects, the noop action a_p for a proposition p is defined as $\rho_e(a_p) = \rho_{00}(a_p) = \varepsilon_{00}^+(a_p) = p$.


```

RPEExtract( $LUG(b), G$ )
1: Let  $k$  be the index of the last level of  $LUG(b)$ 
2: for all  $p \in G \cap \mathcal{P}_k$  do {Initialize Goals}
3:    $\mathcal{P}_k^{RP} \leftarrow \mathcal{P}_k^{RP} \cup p$ 
4:    $\ell_k^{RP}(p) \leftarrow \bigwedge_{p' \in G} \ell_k(p')$ 
5: end for
6: for  $t = k \dots 1$  do
7:   for all  $p \in \mathcal{P}_t^{RP}$  do {Support Each Proposition}
8:      $\ell \leftarrow \ell_t^{RP}(p)$  {Initialize Possible Worlds to Cover}
9:     while  $\ell \neq \perp$  do {Cover Label}
10:      Find  $\varphi_{ij}(a) \in \mathcal{E}_{t-1}$  such that  $p \in \varepsilon_{ij}^+(a)$  and  $(\ell_k(\varphi_{ij}(a)) \wedge \ell) \neq \perp$ 
11:       $\mathcal{E}_{t-1}^{RP} \leftarrow \mathcal{E}_{t-1}^{RP} \cup \varphi_{ij}(a)$ 
12:       $\ell_t^{RP}(\varphi_{ij}(a)) \leftarrow \ell_t^{RP}(\varphi_{ij}(a)) \vee (\ell_t(\varphi_{ij}(a)) \wedge \ell)$ 
13:       $\mathcal{A}_{t-1}^{RP} \leftarrow \mathcal{A}_{t-1}^{RP} \cup a$ 
14:       $\ell_t^{RP}(a) \leftarrow \ell_t^{RP}(a) \vee (\ell_t(\varphi_{ij}(a)) \wedge \ell)$ 
15:       $\ell \leftarrow \ell \wedge \neg \ell_t(\varphi_{ij}(a))$ 
16:    end while
17:  end for
18:  for all  $a \in \mathcal{A}_{t-1}^{RP}, p \in \rho_e(a)$  do {Insert Action Preconditions}
19:     $\mathcal{P}_{t-1}^{RP} \leftarrow \mathcal{P}_{t-1}^{RP} \cup p$ 
20:     $\ell_{t-1}^{RP}(p) \leftarrow \ell_{t-1}^{RP}(p) \wedge \ell_{t-1}^{RP}(a)$ 
21:  end for
22:  for all  $\varphi_{ij}(a) \in \mathcal{E}_{t-1}^{RP}, p \in \rho_{ij}(a)$  do {Insert Effect Preconditions}
23:     $\mathcal{P}_{t-1}^{RP} \leftarrow \mathcal{P}_{t-1}^{RP} \cup p$ 
24:     $\ell_{t-1}^{RP}(p) \leftarrow \ell_{t-1}^{RP}(p) \wedge \ell_{t-1}^{RP}(\varphi_{ij}(a))$ 
25:  end for
26: end for
27: return  $\langle (\mathcal{P}_0^{RP}, \mathcal{A}_0^{RP}, \mathcal{E}_0^{RP}, \mathcal{P}_1^{RP}, \dots, \mathcal{A}_{k-1}^{RP}, \mathcal{E}_{k-1}^{RP}, \mathcal{P}_k^{RP}), \ell^{RP} \rangle$ 

```

Fig. 8. Labeled Relaxed Plan Extraction Algorithm.

potentially two labels for each part of the graph: the un-bolded label found during graph construction, and the bolded label associated with the relaxed plan (described below).

The heuristic value of a belief state is most informed if it accounts for all possible states, but the benefit of using the LUG is lost if we compute and then aggregate the relaxed plan for each state. Instead, we can extract a labeled relaxed plan to avoid enumeration by manipulating labels. The labeled relaxed plan $\langle (\mathcal{P}_0^{RP}, \mathcal{A}_0^{RP}, \mathcal{E}_0^{RP}, \dots, \mathcal{A}_{k-1}^{RP}, \mathcal{E}_{k-1}^{RP}, \mathcal{P}_k^{RP}), \ell^{RP} \rangle$ is a subgraph of the LUG that uses labels to ensure that chosen actions are used to support the goals from all states in the source belief state (a conformant relaxed plan). For example, in Figure 7, to support `comm (I1)` in level three we use the labels to determine that persistence can support the goal from $s_2 = \{\text{at (L1)}, \text{have (I1)}\}$ in level two and support the goal from $s_1 = \{\text{at (L1)}\}$

with `commun(I1)` in level two. The relaxed plan extraction is based on ensuring a goal proposition’s label is covered by the labels of chosen supporting actions. To cover a label, we use intuition from the set cover problem, and the fact that a label denotes a set of source states. That is, a proposition’s label denotes a set of states S_p and each action’s label denotes a set of states S_a ; the disjunction of labels of chosen supporting actions A_p denotes a set of states $(\cup_{a \in A_p} S_a)$ that must contain all states denoted by the supported proposition label ($S_p \subseteq \cup_{a \in A_p} S_a$).

The procedure for *LUG* relaxed plan extraction is shown in Figure 8. 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 deterministic 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}_k^{RP} at time k and associating a label $\ell_k^{RP}(p)$ with each to denote the states in b from which they must be supported (lines 2-5).⁴ 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 t from the states represented by $\ell_k^{RP}(p)$ (lines 7-17) is the key decision point of the algorithm, embodied in line 10. First, we initialize a variable ℓ with the remaining states in which to support the proposition (line 8). Until there are no remaining states, we choose effects and their associated actions (lines 9-16). Those effects that i) have the proposition as a positive effect and ii) support from states that need to be covered (i.e., $\ell_k(\varphi_{ij}(a)) \wedge \ell \neq \perp$) are potential choices. In line 10, one of these effects is chosen. We store the effect (line 11) and the states from which it supports (line 12), as well as the associated action (line 13) and the states where its effect is used (line 14). The states 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 states from which to support each proposition are simply the union of the states where an action or effect is needed (lines 20 and 24). The extraction ends by returning the labeled subgraph of the *LUG* that is needed to support the goals from all possible states (line 27). The heuristic is the sum of the number of non-noop actions in each action layer of the relaxed plan.

⁴ Notice that the relaxed plan label of each goal proposition is defined identically in terms of the labels of all goal propositions ($p' \in G$); we define relaxed plan labels in this fashion, even though (in the *LUG*) each goal proposition label should be equivalent to b at the last level, because later extensions of this algorithm for use in the SAG must deal with goal proposition with different labels.

3.3 State Agnostic Planning Graphs

A naive generalization of the *LUG* to its *SAG* version has a larger worst case complexity over the *SAG* version of the deterministic planing graph. Recall that the *SAG* represents a set of planning graphs, and in this case, a set of *LUG* (each representing a set of planning graphs). Each *LUG* may represent $O(2^{|P|})$ planning graphs, making a set of all *LUG* represent $O(2^{2^{|P|}})$ graphs. However, we describe an equivalent *SAG* generalization of the *LUG*, the State Agnostic Labeled Uncertainty Graph (*SLUG*), whose worst-case complexity is identical to the *LUG* and the deterministic *SAG*— an exponential savings over the naive *SAG* generalization. The intuition is that the *LUG* labels represent a set of states, and the naive *SAG* generalization labels would represent a set of sets of states. However, by representing the union of these sets of states and modifying the heuristic extraction, the *SLUG* manages to retain the complexity of the *LUG*.

As stated, the *LUG* is a kind of *SAG*. The *LUG* is an efficient representation of a set of planning graphs built for deterministic planning with conditional effects. We introduced Figure 5 as an example of the *SAG* for deterministic planning; it is possible to re-interpret it as an example of the *LUG*. The graph depicted in Figure 5 is built for the belief state $b = \top$ that contains every state in the rover example. It is possible to use this graph to compute a heuristic for $b = \top$, but it is not yet clear how to compute the heuristic for some other belief state using the same graph. The contribution made with the *SLUG*, described below, is to reuse the labeling technique described for the *LUG*, and provide a modification to the relaxed plan extraction algorithm to compute the relaxed plan for any belief state.

***SLUG*:** The $SLUG(B)$ represents each *LUG* required to compute the heuristic for any belief state in a given set B . Each *LUG* represents a set of planning graphs, and the *SLUG* simply represents the union of all planning graphs used in each *LUG*. Thus, we can construct a *SLUG* with scope B by constructing an equivalent *LUG* for the belief state $b^* = \bigvee_{b \in B} b = \bigvee_{b \in B} \bigvee_{s \in b} s$ (if B contains all belief states, then $b^* = S = \top$). Representing the union leads to an exponential savings because otherwise the $LUG(b)$ and $LUG(b')$ built for belief states b and b' represent redundant planning graphs if there is a state s that is in both b and b' . This is an additional savings not realized in the deterministic *SAG* because no two search nodes (states) use the same planning graph to compute a heuristic. However, like the deterministic *SAG*, the constituent planning graphs share the savings of using a common planning graph skeleton.

Computing the heuristic for a belief state using the *SLUG* involves identifying the planning graphs that would be present in $LUG(b)$. By constructing the $LUG(b)$, the appropriate planning graphs are readily available. However, with the *SLUG*, we need to modify heuristic extraction to “cut away” the irrelevant planning graphs; the same was true when we discussed the deterministic *SAG*. In the deterministic

SAG, it was sufficient to check that the state is a model of a label, $s \models \ell_t(\cdot)$, to determine if the element is in the planning graph for the state. In the *SLUG*, we can also check that each state in the belief state is a model of a label, or that all states in the belief state are models by the entailment check $b \models \ell_t(\cdot)$. For example, the level heuristic for a belief state is t if t is the minimum level where $b \models \bigwedge_{p \in G} \ell_t(p)$ – all goal propositions are in level t of the planning graph for each state $s \in b$.

Extending relaxed plan extraction for a belief state b to the *SLUG* is straightforward, given the existing labeled relaxed plan procedure in Figure 8. Recall that extracting a labeled relaxed plan for the *LUG* involves finding causal support for the goals from all states in a belief state. Each goal proposition is given a label in the relaxed plan that is equal to b , and actions are chosen to cover the label (find support from each state). In the *SLUG*, the labels of the goal propositions may include state models that are not relevant to computing the heuristic for b . The sole modification we make to the algorithm is to restrict which state models must support each goal. The change replaces line 4 of Figure 8 with:

$$“\ell_k^{RP}(p) \leftarrow \bigwedge_{p' \in G} \ell_k(p') \underline{\wedge b}”,$$

the conjunction of each goal proposition label with b (the underlined addition). By performing this conjunction, the relaxed plan extraction algorithm commits to supporting the goal from only states represented in b . Without the conjunction, the relaxed plan would support the goal from every state in some $b' \in B$, which would likely be a poor heuristic estimate (effectively computing the same value for each $b \in B$).

4 Probabilistic Planning

Probabilistic planning involves extensions to handle actions with stochastic outcomes, which affect the underlying planning model, planning graphs, and state agnostic planning graphs. We consider only conformant probabilistic planning in this section, but as in non-deterministic planning, conditional planning can be addressed by ignoring observations in the heuristics.

4.1 Problem Definition

The probabilistic planning problem is defined by (P, A, b_I, G, τ) , where everything is defined as the non-deterministic problem, except that each $a \in A$ has probabilistic outcomes, b_I is a probability distribution over states, and τ is the minimum probability that the plan must satisfy the goal.

$$\begin{aligned}
P &= \{\text{at}(\text{L1}), \text{at}(\text{L2}), \text{have}(\text{I1}), \text{comm}(\text{I1})\} \\
A &= \{ \text{drive}(\text{L1}, \text{L2}) = (\{\text{at}(\text{L1})\}, \{(1.0, \{\{\} \rightarrow (\{\text{at}(\text{L2})\}, \{\text{at}(\text{L1})\})\})\}), \\
&\quad \text{drive}(\text{L2}, \text{L1}) = (\{\text{at}(\text{L2})\}, \{(1.0, \{\{\} \rightarrow (\{\text{at}(\text{L1})\}, \{\text{at}(\text{L2})\})\})\}), \\
&\quad \text{sample}(\text{I1}, \text{L2}) = (\{\text{at}(\text{L2})\}, \{(0.9, \{\{\} \rightarrow (\{\text{have}(\text{I1})\}, \{\})\})\}), \\
&\quad \text{commun}(\text{I1}) = (\{\}, \{(0.8, \{\{\text{have}(\text{I1})\} \rightarrow (\{\text{comm}(\text{I1})\}, \{\})\})\}) \} \\
b_I &= \{(0.9, \{\text{at}(\text{L1}), \text{have}(\text{I1})\}), (0.1, \{\text{at}(\text{L1})\})\} \\
G &= \{\text{comm}(\text{I1})\}
\end{aligned}$$

Fig. 9. Probabilistic Planning Problem Example.

A probabilistic belief state b is a *probability distribution over states*, describing a function $b : S \rightarrow [0, 1]$, such that $\sum_{s \in S} b(s) = 1.0$. While every state is involved in the probability distribution, many are often assigned zero probability. To maintain consistency with non-deterministic belief states, those states with non-zero probability are referred to as states in the belief state, $s \in b$, if $b(s) > 0$.

Like non-deterministic planning, an action $a \in A$ is a tuple $(\rho_e(a), \Phi(a))$, where $\rho_e(a)$ is an enabling precondition and $\Phi(a)$ is a set of causative outcomes. Each causative outcome $\Phi_i(a) \in \Phi(a)$ is a set of conditional effects. In probabilistic models, there is a weight $0 < w_i(a) \leq 1$ indicating the probability of each outcome i being realized, such that $\sum_i w_i(a) = 1$. We redefine the transition relation $T(s, a, s')$ as the sum of the weight of each outcome where $s' = \text{exec}(\Phi_i(a), s)$, such that:

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

Executing action a in belief state b , denoted $\text{exec}(a, b) = b_a$, defines the successor belief state b_a such that $b_a(s') = \sum_{s \in b} b(s)T(s, a, s')$. We define the belief state b' reached by a sequence of actions (a_1, a_2, \dots, a_m) as $b' = \text{exec}((a_1, a_2, \dots, a_m), b) = \text{exec}(a_m, \dots, \text{exec}(a_2, \text{exec}(a_1, b))\dots)$. The cost of the plan is equal to the number of actions in the plan.

4.2 Planning Graphs

Recall that the *LUG* represents a planning graph for each state in a belief state – enumerating the possibilities. When actions have uncertain outcomes, it is also possible to enumerate the possibilities. Prior work on Conformant GraphPlan [40], enumerates both the possibilities due to belief state uncertainty and action outcome uncertainty by constructing a planning graph for each initial state and set of action outcomes. However, because each execution of an action may have a different outcome, the possible outcomes at each level of the planning graph must be enumerated. Thus, we can describe each of the enumerated planning graphs in terms of the random variables $(X_b, X_{a,0}, \dots, X_{a',0}, \dots, X_{a,k-1}, \dots, X_{a',k-1})$. Each planning graph

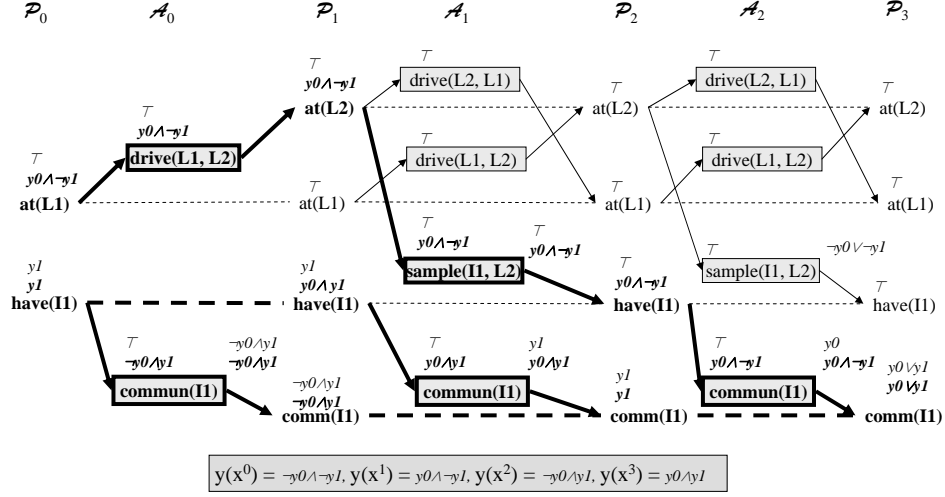


Fig. 10. Monte Carlo labeled uncertainty graph.

is a different assignment of values to the random variables, where X_b is distributed over the states in the source belief state b , and $(X_{a,t}, \dots, X_{a',t})$ are distributed over the action outcomes in action layer t .

In the probabilistic setting, each planning graph has a probability defined in terms of the source belief state and action outcome weights. It is possible to extend the *LUG* to handle probabilities and actions with uncertain outcomes by defining a unique label model for each planning graph and associating with it a probability [11]. However, the labels become quite large because each model is a unique assignment to $(X_b, X_{a,0}, \dots, X_{a',0}, \dots, X_{a,k-1}, \dots, X_{a',k-1})$. With deterministic actions, labels only capture uncertainty about the source belief state (X_b) and the size of the labels is bounded (there is a finite number of states in a belief state). With uncertain actions, labels must capture uncertainty about the belief state and *each uncertain action at each level of the planning graph*. Naturally, as the number of levels and actions increase, the labels used to exactly represent this distribution become exponentially larger and quite costly to propagate for the purpose of heuristics. We note that it does not make sense to *exactly* compute a probability distribution within a *relaxed* planning problem. Monte Carlo techniques are a viable option for approximating the distribution (amounting to sampling a set of planning graphs).

The Monte Carlo *LUG* (*McLUG*) represents a set of sampled planning graphs using the labeling technique developed in the *LUG*. The *McLUG* is a set of vertices and a label function: $\text{McLUG}(b) = \langle (\mathcal{P}_0, \mathcal{A}_0, \mathcal{E}_0, \dots, \mathcal{A}_{k-1}, \mathcal{E}_{k-1}, \mathcal{P}_k), \ell \rangle$. The *McLUG* represents a set of N planning graphs. The n^{th} planning graph, $n = 0, \dots, N - 1$, is identified by a set x^n of sampled values for the random variables $x^n = \{X_b = s, X_{a,0} = \Phi_i(a), \dots, X_{a',0} = \Phi_i(a'), \dots, X_{a,k-1} = \Phi_i(a), \dots, X_{a',k-1} = \Phi_i(a')\}$, corresponding to the state in the belief state and the action outcomes at

different levels. In the following, we denote the n^{th} sampled value v of random variable X by $P(X) \stackrel{n}{\sim} v$.

McLUG Labels: Recall that without probabilistic action outcomes, a single planning graph is uniquely identified by its source state, and the *LUG* or *SAG* define labels in terms of these states (each model of a label is a state). The *McLUG* is different, in that each unique planning graph is identified by a different set of sampled values of random variables (states and action outcomes). In defining the labels for the *McLUG*, we could require that each model of a label refers to a state and a set of level-specific action outcomes; the label would be expressed as a sentence over the state propositions P and a set of propositions denoting the level-specific action outcomes. However, we note that the number of propositions required to express the k levels of level-specific action outcomes is $O(\log_2(|\Phi(a)||A|^k))$. Under this scheme, a label could have $O(2^{|P|}|\Phi(a)||A|^k)$ models, of which only N are actually used in the *McLUG*. It often holds in practice that $N \ll O(2^{|P|}|\Phi(a)||A|^k)$.

The *McLUG* uses an alternative representation of labels where the number of label models is much closer to N . Each planning graph n is assigned a unique boolean formula $y(x^n)$ (a model, or more appropriately, a bit vector), defined over the propositions $(y_0, \dots, y_{\lceil \log_2(N) \rceil - 1})$. For example, when $N = 4$ two propositions y_0 and y_1 are needed, and we obtain the following models: $y(x^0) = \neg y_0 \wedge \neg y_1$, $y(x^1) = y_0 \wedge \neg y_1$, $y(x^2) = \neg y_0 \wedge y_1$, and $y(x^3) = y_0 \wedge y_1$. The *McLUG* depicted in Figure 10, uses $N = 4$ samples for the initial belief state of the probabilistic Rovers problem in Figure 9. The non-bold labels above each graph vertex are boolean formulas defined over y_0 and y_1 that denote which of the sampled planning graphs $n = 0, \dots, 3$ contain the vertex. The bolded labels (described below) denote the relaxed plan labels.

For each planning graph n and corresponding set of sampled values x^n , $n = 0 \dots N - 1$, the *McLUG* satisfies:

- (1) $y(x^n) \models \ell_0(p)$ and $X_b = s \in x^n$ iff $P(X_b) \stackrel{n}{\sim} s$ and $p \in s$
- (2) $y(x^n) \models \ell_t(a)$ iff $y(x^n) \models \ell_t(p)$ for every $p \in \rho_e(a)$
- (3) $y(x^n) \models \ell_t(\varphi_{ij}(a))$ and $X_{a,t} = \Phi_i(a) \in x^n$ iff $P(X_{a,t}) \stackrel{n}{\sim} \Phi_i(a)$ and $y(x^n) \models \ell_t(a) \wedge \ell_t(p)$ for every $p \in \rho_{ij}(a)$
- (4) $y(x^n) \models \ell_{t+1}(p)$ iff $p \in \varepsilon_{ij}^+(a)$ and $y(x^n) \models \ell_t(\varphi_{ij}(a))$

Notice that the primary difference between the *LUG* and *McLUG* is that the *McLUG* records a set of sampled random variable assignments for each planning graph in the set x^n and ensures the model $y(x^n)$ entails the corresponding graph element label. In reality, each set x^n is maintained implicitly through the *McLUG* labels.

The following label rules can be used to construct the *McLUG*:

- (1) If $P(X_b) \stackrel{n}{\sim} s$, then $X_b = s \in x^n$, $n = 0, \dots, N - 1$

- (2) $\ell_0(p) = \bigvee_{p \in s: X_b = s \in x^n} y(x^n)$
- (3) $\ell_t(a) = \bigwedge_{p \in \rho_e(a)} \ell_t(p)$
- (4) If $P(X_{a,t}) \stackrel{n}{\sim} \Phi_i(a)$, then $X_{a,t} = \Phi_i(a) \in x^n, n = 0, \dots, N - 1$
- (5) $\ell_t(\varphi_{ij}(a)) = \ell_t(a) \wedge \left(\bigwedge_{p \in \rho_{ij}(a)} \ell_t(p) \right) \wedge \left(\bigvee_{x^n: X_{a,t} = \Phi_i(a) \in x^n} y(x^n) \right)$
- (6) $\ell_t(p) = \bigvee_{\varphi_{ij}(a) \in \mathcal{E}_{t-1}: p \in \varepsilon_{ij}^+(a)} \ell_{t-1}(\varphi_{ij}(a))$
- (7) $k = \text{minimum } t \text{ such that } \sum_{n=0}^{N-1} \delta(n)/N \geq \tau$, where $\delta(n) = 1$ if $y(x^n) \models \bigwedge_{p \in G} \ell_t(p)$, and $\delta(n) = 0$, otherwise.

The *McLUG* label construction starts by sampling N states from the belief state b and associating each of the n^{th} sampled states with a planning graph n (1, above). In the example, the state $\{\text{at}(\text{L1})\}$ is sampled first and second, and associated with the sets x^0 and x^1 , and likewise, state $\{\text{at}(\text{L1}), \text{have}(\text{I1})\}$ is sampled and associated with x^2 and x^3 . Initial layer proposition labels denote all samples of states where the propositions hold (2, above). The labels in the example are calculated as follows, $\ell_0(\text{at}(\text{L1})) = y(x^0) \vee y(x^1) \vee y(x^2) \vee y(x^3) = \top$ because $\text{at}(\text{L1})$ holds in every state sample and $\ell_0(\text{have}(\text{I1})) = y(x^2) \vee y(x^3) = y_1$ because $\text{have}(\text{I1})$ holds in two of the state samples. Action labels denote all planning graphs where all of their preconditions are supported (3, above), as in the *LUG* and *SAG*. At each level, each action outcome is sampled by each planning graph n (4, above). In the example, the outcomes of $\text{commun}(\text{I1})$ in level zero are sampled as follows:

$$P(X_{\text{commun}(\text{I1}),0}) \stackrel{0}{\sim} \Phi_0(\text{commun}(\text{I1})),$$

$$P(X_{\text{commun}(\text{I1}),0}) \stackrel{1}{\sim} \Phi_0(\text{commun}(\text{I1})),$$

$$P(X_{\text{commun}(\text{I1}),0}) \stackrel{2}{\sim} \Phi_0(\text{commun}(\text{I1})),$$

$$P(X_{\text{commun}(\text{I1}),0}) \stackrel{3}{\sim} \Phi_1(\text{commun}(\text{I1})).$$

Each effect is labeled to denote the planning graphs where it is supported and its outcome is sampled (5, above). While the outcome $\Phi_0(\text{commun}(\text{I1}))$ is sampled the first three times, the effect $\varphi_{00}(\text{commun}(\text{I1}))$ is only supported when $n = 2$ because its antecedent $\text{have}(\text{I1})$ is only supported when $n = 2$ and $n = 3$. The label is calculated as follows (via rule 5, above): $\ell_0(\varphi_{00}(\text{commun}(\text{I1}))) = \ell_0(\text{commun}(\text{I1})) \wedge \ell_0(\text{have}(\text{I1})) \wedge (y(x^0) \vee y(x^1) \vee y(x^2)) = \top \wedge y_1 \wedge ((\neg y_1 \wedge \neg y_2) \vee (\neg y_1 \wedge y_2) \vee (y_1 \wedge \neg y_2)) = y_1 \wedge \neg y_2$. Each proposition is labeled to denote planning graphs where it is supported by some effect (6, above), as in the *LUG* and *SAG*. The last level k (which can be used as the level heuristic) is defined as the level where a proportion of planning graphs where all goal propositions are

reachable is no less than τ . A planning graph reaches the goal if its label is a model of the conjunction of goal proposition labels. In the example, one planning graph reaches the goal $\text{comm}(\text{I1})$ at level one (its label has one model); two, at level two; and three, at level three. The relaxed plan shown in bold, supports the goal (in the relaxed planning space) with probability 0.75 because it supports the goal in three of four sampled planning graphs.

Labeled relaxed plan extraction in the *McLUG* is identical to the *LUG*, as described in the previous section. However, the interpretation of procedure’s semantics does change slightly. We pass a *McLUG* for a given belief state b to the procedure, instead of a *LUG*. The labels for goal propositions (line 4) represent sampled planning graphs, and via the *McLUG* termination criterion, we do not require goal propositions to be reached by all planning graphs – only a proportion no less than τ .

In the example, the goal $\text{comm}(\text{I1})$ is reached in three planning graphs because its label at level three is $y_0 \vee y_1 = y(x^1) \vee y(x^2) \vee y(x^3)$. The planning graphs associated with sample sets x^2, x^3 support the goal by $\varphi_{00}(\text{comm}(\text{I1})_p)$, and the planning graph with sample set x^1 supports the goal by $\varphi_{00}(\text{commun}(\text{I1}))$, so we include both in the relaxed plan. For each action we subgoal on the antecedent of the chosen conditional effect as well as its enabling precondition. The relaxed plan contains three invocations of $\text{commun}(\text{I1})$ (reflecting how action repetition is needed when actions have uncertain outcomes), and the $\text{drive}(\text{L1}, \text{L2})$ and $\text{sample}(\text{L1}, \text{L2})$ actions. The value of the relaxed plan is five because it uses five non-persistence actions.

4.3 State Agnostic Planning Graphs

A state agnostic generalization of the *McLUG* should be able to compute the heuristic for all probabilistic belief states (of which there are an infinite number). Following our intuitions from the *McLUG*, we would sample N states from a belief state, and expect that the state agnostic *McLUG* already represents a planning graph for each. However, this is not enough, we should also expect that each of these N planning graphs uses a different set of sampled action outcomes. For example, if each of the N sampled states is identical and all planning graphs built for this state use identical action outcome samples, we might compute a poor heuristic (essentially ignoring the fact that effects are probabilistic).

In order to precompute N planning graphs (using different action outcome samples) for each sampled state, we could construct N copies of the *SLUG*, each built for a different set of sampled action outcomes $x^n = \{X_{a,0} = \Phi_i(a), \dots, X_{a',0} = \Phi_i(a'), \dots, X_{a,k-1} = \Phi_i(a), \dots, X_{a',k-1} = \Phi_i(a')\}$. We refer to the n^{th} *SLUG* by $SLUG(x^n)$. To compute a heuristic, we sample N states from a belief state and

lookup the planning graph for each of the n^{th} sampled states in the corresponding $SLUG(x^n)$. In this manner, we can obtain different sets of action outcome samples even if the same state is sampled twice. We note, however, that a set of $SLUG$ is highly redundant, and clashes with our initial motivations for studying state agnostic planning graphs. Since the set of $SLUG$ is essentially a set of planning graphs, the contribution of this section is showing how to extend the label semantics to capture a set of $SLUG$ built with different action outcomes in a single data structure, which we call the $McSLUG$. Prior to discussing the $McSLUG$ labels, we explore some important differences between the $McLUG$ and the $McSLUG$.

Comparison with $McLUG$: Both the $McLUG$ and $McSLUG$ sample N states from a belief state, and use a planning graph with a different set of action outcome samples for each state. However, the $McLUG$ generates a new set of action outcomes for each state sampled from each belief state, where, instead, the $McSLUG$ reuses an existing set of action outcomes for each state (depending on which $SLUG$ is used for the state). The $McSLUG$ introduces potential correlation between the heuristics computed for different belief states because it is possible that the same state is sampled from each belief state and that state’s planning graph is obtained from the same $SLUG$. The $McLUG$ is often more robust because even if the same state is sampled twice from different belief states it is unlikely that the set of sampled action outcomes is identical. As discussed in the empirical results, the $McSLUG$ can degrade the heuristic informedness, and hence planner performance, in some problems. However, in other problems the heuristic degradation is offset by improved speed of computing the heuristic with the $McSLUG$.

$McSLUG$: The $McSLUG$ is a set of action, effect, and proposition vertex layers and a label function: $McSLUG(\mathcal{S}, N) = \langle (\mathcal{P}_0, \mathcal{A}_0, \mathcal{E}_0, \dots, \mathcal{A}_{k-1}, \mathcal{E}_{k-1}, \mathcal{P}_k), \ell \rangle$. The $McSLUG$ label semantics involves combining $McLUG$ labels and $SLUG$ labels. Recall that the $McLUG$ uses a unique model $y(x^n)$ for each set of sampled outcomes x^n (which correspond to a deterministic planning graph) and that the $SLUG$ uses models of the state propositions. The $McSLUG$ represents a set $\{SLUG(x^n) \mid n = 0 \dots N - 1\}$, where we distinguish each $SLUG(x^n)$ by a unique formula $y(x^n)$ and distinguish the planning graph for state s in $SLUG(x^n)$ by the formula $s \wedge y(x^n)$. Thus, $McSLUG$ labels have models of the form $s \wedge y(x^n)$, defined over the propositions in P , referring to states, and $y_0, \dots, y_{\log_2(N)-1}$, referring to a particular $SLUG$ built for a set of action outcome samples.

For each $s \models \mathcal{S}$ and $n = 0 \dots N - 1$, the $McSLUG$ satisfies:

- (1) $s \wedge y(x^n) \models \ell_0(p)$ iff $p \in s$
- (2) $s \wedge y(x^n) \models \ell_t(a)$ iff $s \wedge y(x^n) \models \ell_t(p)$ for every $p \in \rho_e(a)$
- (3) $s \wedge y(x^n) \models \ell_t(\varphi_{ij}(a))$ and $X_{a,t} = \Phi_i(a) \in x^n$ iff $P(X_{a,t}) \stackrel{n}{\sim} \Phi_i(a)$ and $s \wedge y(x^n) \models \ell_t(p)$ for every $p \in \rho_{ij}(a)$
- (4) $s \wedge y(x^n) \models \ell_{t+1}(p)$ iff $p \in \varepsilon_{ij}^+(a)$ and $s \wedge y(x^n) \models \ell_t(\varphi_{ij}(a))$

The following rules can be used to construct the \mathcal{McSLUG} :

- (1) $\ell_0(p) = \mathcal{S} \wedge p \wedge \left(\bigvee_{n=0..N-1} y(x^n) \right)$
- (2) $\ell_t(a) = \bigwedge_{p \in \rho_e(a)} \ell_t(p)$
- (3) If $P(X_{a,t}) \stackrel{n}{\sim} \Phi_i(a)$, then $X_{a,t} = \Phi_i(a) \in x^n, n = 0, \dots, N - 1$
- (4) $\ell_t(\varphi_{ij}(a)) = \ell_t(a) \wedge \left(\bigwedge_{p \in \rho_{ij}(a)} \ell_t(p) \right) \wedge \left(\bigvee_{x^n: X_{a,t} = \Phi_i(a) \in x^n} y(x^n) \right)$
- (5) $\ell_t(p) = \bigvee_{\varphi_{ij}(a) \in \mathcal{E}_{t-1}: p \in \varepsilon_{ij}^+(a)} \ell_{t-1}(\varphi_{ij}(a))$
- (6) $k = \text{minimum } t \text{ such that } \mathcal{P}_{t+1} = \mathcal{P}_t, \ell_{t+1}(p) = \ell_t(p), p \in P, \text{ and } \sum_{n=0}^{N-1} \delta(n)/N = 1.0, \text{ where } \delta(n) = 1 \text{ if } \mathcal{S} \wedge y(x^n) \models \bigwedge_{p \in G} \ell_t(p), \text{ and } \delta(n) = 0, \text{ otherwise.}$

The initial layer propositions are labeled (1, above) to denote the states in the scope where they hold ($\mathcal{S} \wedge p$) and the sets of action outcome samples where they hold ($\bigvee_{n=0..N-1} y(x^n)$) – if an initial layer proposition holds in a planning graph, it holds regardless of which action outcomes are sampled. Figure 11 depicts the \mathcal{McSLUG} for the rovers example, where the scope is $\mathcal{S} = \top$ and $N = 4$. The initial proposition layer label for $\text{comm}(\text{I1})$ is computed as follows: $\ell_0(\text{comm}(\text{I1})) = \top \wedge \text{comm}(\text{I1}) \wedge (\bigvee_{n=0..3} y(x^n)) = \top \wedge \text{comm}(\text{I1}) \wedge \top = \text{comm}(\text{I1})$. Action labels denote the planning graphs where their preconditions are all reachable (2, above), as in the \mathcal{McLUG} . N samples of each action’s outcome (one for each \mathcal{SLUG}) are drawn in each level and the outcome’s effects are labeled (3, above). Each effect label (4, above) denotes the planning graphs i) that contain the effect’s outcome and ii) where the effect is reachable (its action and secondary preconditions are reachable), as in the \mathcal{McLUG} . Each proposition label denotes the planning graphs where it is given by some effect (5, above), as in the \mathcal{McLUG} . The last level k is defined by the level where proposition layers and labels are identical and all planning graphs satisfy the goal (6, above).

The termination criterion for the \mathcal{McSLUG} requires some additional discussion. Unlike, the \mathcal{McLUG} , where at least τ proportion of the planning graphs must satisfy the goal, we require that *all* planning graphs satisfy the goal in the \mathcal{McSLUG} – which has several implications. First, it may not be possible to satisfy the goal in all of the planning graphs; however, i) this problem also affects the \mathcal{McLUG} , and because iia) the planning graphs are relaxed and iib) most problems contain actions whose outcome distributions are skewed towards favorable outcomes, it is often possible to reach the goal in every planning graph. Second, because the \mathcal{McSLUG} heuristics use N planning graphs where the goal is reachable, the heuristic estimates the cost to achieve the goal with 1.0 probability (which may be an overestimate when $\tau \ll 1.0$). Third, despite the first two points, it is unclear in which planning graphs it is acceptable to not achieve the goal; until states are sampled from belief states during search, we will not know which planning graphs will be

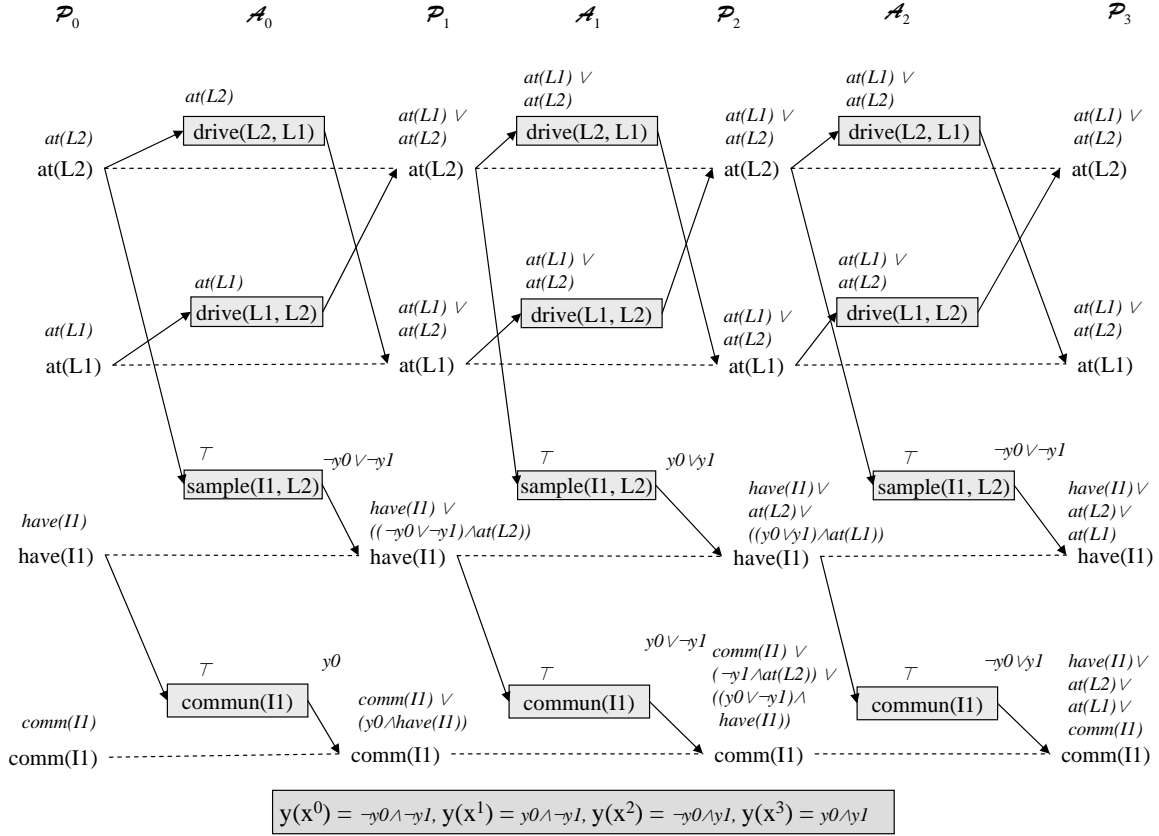


Fig. 11. Graph structure of a $\mathcal{M}cSLUG$.

used. The net effect is that search will seek out highly probable plans, which still solve the problem of exceeding the goal satisfaction threshold.

Computing Heuristics: Using the $\mathcal{M}cSLUG$ to evaluate the heuristic for a belief state b is relatively similar to using the $\mathcal{M}cLUG$. We sample N states from b , and associate each state with a $SLUG(x^n)$. If s is the n^{th} state sample, $P(X_b) \stackrel{n}{\sim} s$, then we use the planning graph whose label model is $s \wedge y(x^n)$. The set of N planning graphs thus sampled is denoted by the boolean formula $\ell^N(b)$, where

$$\ell^N(b) = \bigvee_{n: P(X_b) \stackrel{n}{\sim} s, n=0 \dots N-1} s \wedge y(x^n)$$

The level heuristic is the first level t where the proportion of the N planning graphs that satisfy the goal exceeds the threshold τ . The n^{th} planning graph satisfies the goal in level t if $s \wedge y(x^n) \models \bigwedge_{p \in G} \ell_t(p)$. We compute the proportion of the N planning graphs satisfying the goal by counting the number of models of the formula $\ell^N(b) \wedge \bigwedge_{p \in G} \ell_t(p)$, denoting the subset of the N sampled planning graphs

that satisfy the goal in level t .

For example, the level heuristic for the initial belief state $\{0.1 : s_I, 0.9 : s_4\}$ to reach the goal of the example is computed as follows. If $N = 4$, we may draw the following sequence of state samples from the initial belief state (s_I, s_4, s_4, s_4) to define

$$\begin{aligned}
\ell^N(b_I) &= (s_I \wedge y(x^0)) \vee (s_4 \wedge y(x^1)) \vee (s_4 \wedge y(x^2)) \vee (s_4 \wedge y(x^3)) \\
&= (s_I \wedge \neg y_0 \wedge \neg y_1) \vee (s_4 \wedge y_0 \wedge \neg y_1) \vee (s_4 \wedge \neg y_0 \wedge y_1) \vee (s_4 \wedge y_0 \wedge y_1) \\
&= (s_I \wedge \neg y_0 \wedge \neg y_1) \vee (s_4 \wedge (y_0 \vee y_1)) \\
&= (\text{at(L1)} \wedge \neg \text{at(L2)} \wedge \neg \text{have(I1)} \wedge \neg \text{comm(I1)} \wedge \neg y_0 \wedge \neg y_1) \vee \\
&\quad (\text{at(L1)} \wedge \neg \text{at(L2)} \wedge \text{have(I1)} \wedge \neg \text{comm(I1)} \wedge (y_0 \vee y_1))
\end{aligned}$$

We compute the level heuristic by computing the conjunction $\ell^N(b) \wedge \bigwedge_{p \in G} \ell_t(p)$, noted above, at each level t until the proportion of models is no less than τ . At level zero, the conjunction of $\ell^N(b_I)$ and the label of the goal proposition comm(I1) is

$$\ell^N(b_I) \wedge \ell_0(\text{comm(I1)}) = \ell^N(b) \wedge \text{comm(I1)} = \perp,$$

meaning that the goal is satisfied with zero probability at level zero. At level one we obtain

$$\begin{aligned}
\ell^N(b_I) \wedge \ell_1(\text{comm(I1)}) &= \ell^N(b_I) \wedge (\text{comm(I1)} \vee (y_0 \wedge \text{have(I1)})) \\
&= \text{at(L1)} \wedge \neg \text{at(L2)} \wedge \text{have(I1)} \wedge \neg \text{comm(I1)} \wedge y_0,
\end{aligned}$$

meaning that the goal is satisfied with 0.5 probability at level one, because two of the sampled planning graph label models $\text{at(L1)} \wedge \neg \text{at(L2)} \wedge \text{have(I1)} \wedge \neg \text{comm(I1)} \wedge y_0 \wedge \neg y_1$ and $\text{at(L1)} \wedge \neg \text{at(L2)} \wedge \text{have(I1)} \wedge \neg \text{comm(I1)} \wedge y_0 \wedge y_1$ are models of the conjunction above. At level two we obtain

$$\begin{aligned}
\ell^N(b_I) \wedge \ell_2(\text{comm(I1)}) &= \ell^N(b_I) \wedge (\text{comm(I1)} \vee (\neg y_1 \wedge \text{at(L2)}) \vee ((y_0 \vee \neg y_1) \wedge \text{have(I1)})) \\
&= \text{at(L1)} \wedge \neg \text{at(L2)} \wedge \text{have(I1)} \wedge \neg \text{comm(I1)} \wedge y_0
\end{aligned}$$

meaning that the goal is satisfied with 0.5 probability at level two because we obtain the same two label models as level one. At level three we obtain

$$\ell^N(b_I) \wedge \ell_3(\text{comm(I1)}) = \ell^N(b_I) \wedge (\text{have(I1)} \vee \text{at(L2)} \vee \text{at(L1)} \vee \text{comm(I1)})$$

meaning that the goal is satisfied with probability 1.0 because there are four models of the formula above.

Thus, the probability of reaching the goal is $0/4 = 0$ at level zero, $2/4 = 0.5$ at level

one, $2/4 = 0.5$ at level two, and $4/4 = 1.0$ at level three.

Relaxed plan extraction for the *McSLUG* is identical to relaxed plan extraction for the *SLUG*, with the exception that the conjunction in line 4 of Figure 8 replaces the belief state b with the formula representing the set of sampled label models, such that line 4 becomes:

$$“\ell_k^{RP}(p) \leftarrow \bigwedge_{p' \in G} \ell_k(p') \wedge \underline{\ell^N(b)}”$$

adding the underlined portion. The addition to relaxed plan extraction enforces that the relaxed plan supports the goal in only those planning graphs sampled to compute the heuristic.

5 State Agnostic Relaxed Plans

There two fundamental techniques that we identify for extracting relaxed plans from state agnostic planning graphs to guide *POND*'s search. As previously described, the first, simply called a relaxed plan, extracts relaxed plans from the state agnostic graph during search upon generating each search node. The second, called a state agnostic relaxed plan (SAGRP), extracts a relaxed plan from every planning graph represented by the SAG. The set of relaxed plans are represented implicitly by a single labeled relaxed plan, and extracted as such.

For example, the state agnostic relaxed plan extracted from the *SLUG* would correspond to the labeled relaxed plan for a belief state containing all states. Then, to compute the heuristic for a single belief state, we check the label of each action of the state agnostic relaxed plan. If any state in the belief state is a model of the label, then the action is in the labeled relaxed plan for the belief state. The number of such non-noop actions is used as the conformant relaxed plan heuristic. In this manner, evaluating the heuristic for a search node (aside from the one-time-per-problem-instance state agnostic relaxed plan extraction) involves no non-deterministic choices and only simple model checking.

The trade-off between traditional relaxed plans and the state agnostic relaxed plan, is very similar to the trade-off between using a planning graph and a SAG: *a priori* construction cost must be amortized over search node expansions. However, there is another subtle difference between the relaxed plans computed between the two techniques. Relaxed plan extraction is guided by a simple heuristic that prefers to support propositions with supporters that contribute support in more planning graphs. In the traditional relaxed plan, this set of planning graphs contains exactly those planning graphs that are needed to evaluate a search node. In the state agnostic relaxed plan, the set of planning graphs used to choose supporters may be much larger than the set used to evaluate any given search node. By delaying the relaxed

plan extraction to customize it to each search node (e.g., by exploiting positive interactions), the traditional relaxed plan can be more informed. By committing to a state agnostic relaxed plan before search (without knowledge of the actual search nodes), the heuristic may make poor choices in relaxed plan extraction and be less informed. Despite the possibility of poor relaxed plans, the state agnostic relaxed plan is quite efficient to compute for every search node (modulo the higher extraction cost).

6 Empirical Evaluation

In this section, we evaluate several aspects of state agnostic planning graphs to answer the following questions:

- Is the cost of pre-computing all planning graphs with the SAG effectively amortized over a search episode in order to improve planner scalability?
- Will using state agnostic relaxed plans improve planner performance over using traditional relaxed plans?
- How will using the SAG compare to other state of the art approaches?

To answer these questions, this section is divided into three subsections. The first subsection describes the setup (*POND* implementation, other planners, domains, and environments) used for evaluation. The last two subsections discuss the first two questions in an internal evaluation and the last question in an external evaluation, respectively. We use a number of planning domains and problems that span deterministic, non-deterministic, and probabilistic planning. Where possible, we supplement domains from the literature with domains used in several International Planning Competitions (i.e., deterministic and non-deterministic tracks). In the case of non-deterministic planning, we discuss actual competition results (where our planner competed using SAG techniques) in addition to our own experiments.

6.1 Evaluation Setup

This subsection describes the implementation of our planner, other planners, domains, and environments that we use to evaluate our approach.

Implementation: *POND* is implemented in C++ and makes use of some notable existing technologies: the CUDD BDD package [41] and the IPP planning graph [27]. *POND* uses ADDs and BDDs to represent belief states, actions, and planning graph labels. In this work, we describe two of the search algorithms implemented in *POND*: enforced hill-climbing [22] and weighted A* search (with a heuristic weight of five). Briefly, enforced hill-climbing interleaves local search with system-

atic search by locally committing to action that lead to search nodes with decreased the heuristic value and using breadth first search if it cannot immediately find a better search node.

The choice of OBDDs to represent labels (aside from parsimony with the action and belief state representation) requires some explanation, as there are a number of alternatives. The primary operations required for label reasoning are model checking and entailment, which take polynomial time in the worst case when using OBDDs [14]. While the size of the BDDs can become exponentially large through the repeated disjunctions and conjunctions during label propagation, the size does not become prohibitive in practice.

A final implementation consideration is the choice of scope (set of states) used to construct the SAG. In preliminary evaluations, we found that using a scope consisting of all $2^{|P|}$ states is always dominated by an approach using the estimated reachable states. We estimate the reachable states by constructing a planning graph whose initial proposition layer consists of propositions holding in some initial state (or state in the initial belief state) and assuming that all outcomes of probabilistic actions occur. The estimated reachable states are those containing propositions in the last proposition layer of this “unioned” planning graph.

Planners: We compare our planner *POND* with several other planners. In non-deterministic planning, we compare with Conformant FF (CFF) [21] and t0 [35]. CFF is an extension of the FF planner [22] to handle initial state uncertainty. CFF is similar to *POND* in terms of using forward chaining search with a relaxed plan heuristic; however, CFF differs in how it implicitly represents belief states and computes relaxed plan heuristics by using a SAT solver. The t0 planner is based on a translation from conformant planning to classical planning, where it uses the FF planner [22].

In probabilistic planning, we compare with Probabilistic FF (PFF) [16] and CPplan [24]. PFF further generalizes CFF to use a weighted SAT solver and techniques to encode probabilistic effects of actions. PFF computes relaxed plan heuristics by encoding them as a type of Bayesian inference, where the weighted SAT solver computes the answer. CPplan is based on a CSP encoding of the entire planning problem. CPplan is an optimal planner that finds the maximum probability of satisfying the goal in a k-step plan; by increasing k incrementally, it is possible to find a plan that exceeds a given probability of goal satisfaction threshold. We omit comparison with COMPlan, a successor to CPplan that also finds optimal bounded length plans, because the results presented by [23] are not as competitive as with PFF. However, we do note that while COMPlan is over an order of magnitude slower than *POND* or PFF, it does have better scalability than CPplan.

Domains: We use deterministic, non-deterministic, and probabilistic planning do-

mains that appear either in the literature or past IPCs. In classical planning, we use domains from the first three IPCs, including Logistics, Rovers, Blocksworld, Zenotravel, Driverlog, Towers of Hanoi, and Satellite [31]. In non-deterministic planning, we use Rovers [9], Logistics [9], Ring [2], and Cube [2], and Blocksworld, Coins, Communication, Universal Traversal sequences, Adder Circuits, and Sorting Networks from the Fifth IPC conformant planning track [18]. All non-deterministic domains use only deterministic actions and an incomplete initial state, and all probabilistic domains use probabilistic actions. In probabilistic planning we use the Logistics, Grid, Slippery Gripper, and Sand Castle [25] domains. We include two versions of each probabilistic Logistics instance that have different initial belief states and goals in order to maintain consistency with results previously presented [25,16]. We do not use probabilistic problems with deterministic actions [16], because these problems can be seen as *easier* forms of the corresponding non-deterministic problems with deterministic actions where the plan need not satisfy the goal with 1.0 probability.

Environments: In all models, the measurement for each problem is the total run time of the planner (including SAG construction in *POND*), from invocation to exit, and the resulting plan length. We have only modified the manner in which the heuristic is computed; despite this, we report total time to motivate the importance of optimizing heuristic computation. It should be clear, given that we achieve large factors of improvement, that time spent calculating heuristics is dominating time spent searching.

All tests, with the exception of the classical planning domains, the Fifth IPC domains, and PFF solving the probabilistic planning domains, were run on a 1.86GHz P4 Linux machine with 1GB memory and a 20 minute time limit. There are several hundred problems in the classical planning IPC test sets, so we imposed relatively tight limits on the execution (5 minutes on a P4 at 3.06 GHz with 900 MB of RAM) of any single problem. We exclude failures due to these limits from the figures. In addition, we sparsely sampled these failures with relaxed limits to ensure that my conclusions were not overly sensitive to the choice of limits. Up until the point where physical memory is exhausted, the trends remain the same. The Fifth IPC was run on a single Linux machine, where competitors were given 24 hours to attempt all problems in all domains, but no limit was placed on any single problem. We were unable to run PFF on the same machine used for other probabilistic planning problems. We used a significantly faster machine, but report the results without scaling because the speed of the machine did not give PFF an unfair advantage in scalability. In the probabilistic domains, the results presented are the average of five runs, with only the successful runs reported in the average.

6.2 Internal Evaluation

We describe several sets of results that address whether using the SAG is reasonable and how computing different types of relaxed plans affects planner performance. The results include deterministic, non-deterministic, and probabilistic planning problems. In every model we present results to describe scalability improvements due to the SAG, where we extract a relaxed plan at each search node. Finally, upon noticing that the SAG is most beneficial in probabilistic models, we concentrate on how computing the relaxed plan differently (either at each search node or via the SAGRP) affects performance.

Amortization: The primary issue that we address is whether it makes sense to use the SAG at all. The trade-off is between efficiently capturing common structure among a set of planning graphs and potentially wasting effort on computing unnecessary, albeit implicit, planning graphs. If the heuristic computation cost per search node is decreased (without degrading the heuristic), then the SAG is beneficial.

We start by presenting results in Figure 12 that compare the SAG with the planning graph (PG) in classical planning in the classical IPC domains. The top graph is a scatter-plot of the total running times for approximately 500 instances for the first three IPCs. The line “ $y=x$ ” is plotted, which plots identical performance. The middle graph in each figure plots the number of problems that each approach has solved by a given deadline. The bottom graph in each figure offers one final perspective, plotting the ratio of the total running times. We see that the SAG produces an improvement on average. While the scatter-plots reveal that performance can degrade, it is still the case that average time is improved: mostly due to the fact that as problems become more difficult, the savings become larger.

In light of this, we have made an investigation in comparing SAG to state of the art implementations of classical planning graphs. In particular, FF [22] goes to great lengths to build classical planning graphs as quickly as possible, and subsequently extract a relaxed plan. We compete against that implementation with a straightforward implementation of SAG within FF. We ran trials of greedy best-first search using the FF relaxed plan heuristic against using the same heuristic as the SAG strategy. Total performance was improved, sometimes doubled, for the Rovers domain; however, in most other benchmark problems, the relative closeness of the goal and the poor estimate of reachability prohibited any improvement. Of course, per-node heuristic extraction time (i.e. ignoring the time it takes to build the SAG) was always improved, which motivates an investigation into more sophisticated graph-building strategies than SAG.

We see that in non-deterministic planning domains (Bomb in Toilet, Cube, Ring [2], Logistics, and Rovers [9]), depicted in Figure 13 in the same format as the deterministic planning problems, the scatter-plots reveal that SAG always outper-

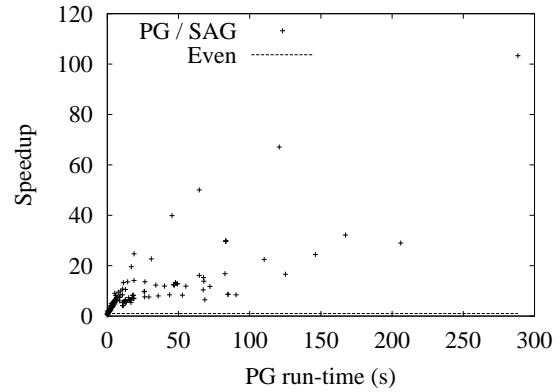
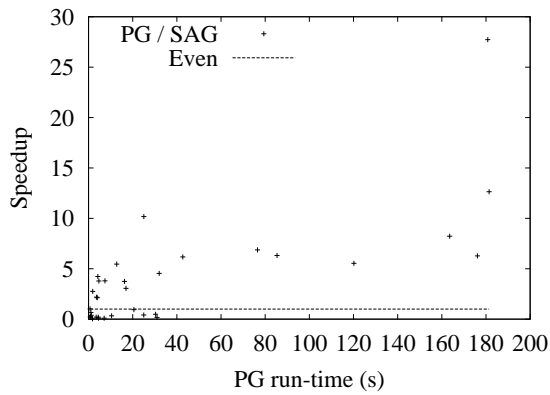
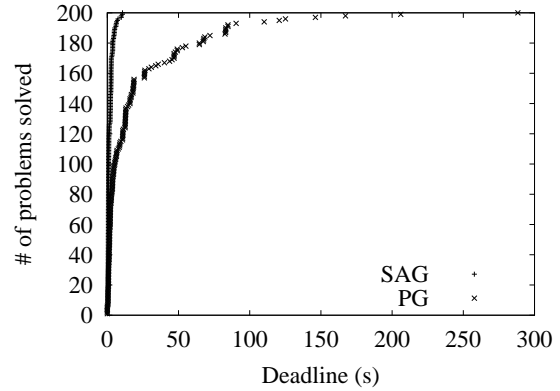
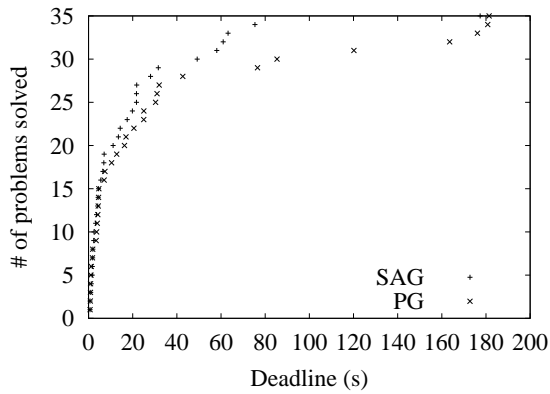
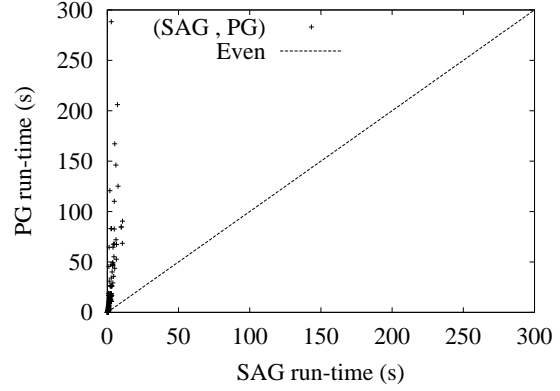
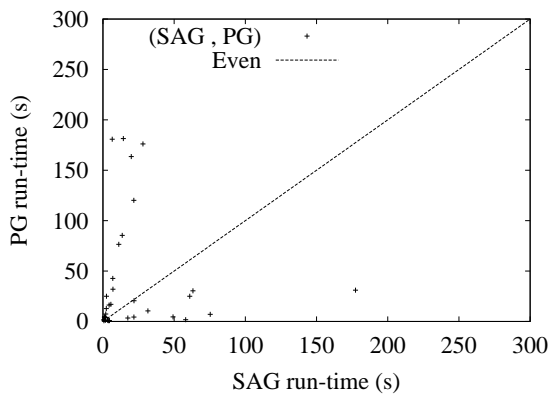


Fig. 12. SAG vs. PG, Classical Problems

Fig. 13. *SLUG* vs. *LUG*, Non-Deterministic Problems

forms the PG approach. Moreover, the boost in performance is well-removed from the break-even point. The deadline graphs are similar in purpose to plotting time as a function of difficulty: rotating the axes reveals the telltale exponential trend. However, it is difficult to measure difficulty across domains. This method corrects for that at the cost of losing the ability to compare performance on the same problem. We observe that, with respect to any deadline, SAG solves a much greater number of planning problems. Most importantly, the SAG out-scales the PG approach. When we examine the speedup graphs, we see that the savings grow larger as the problems become more difficult.

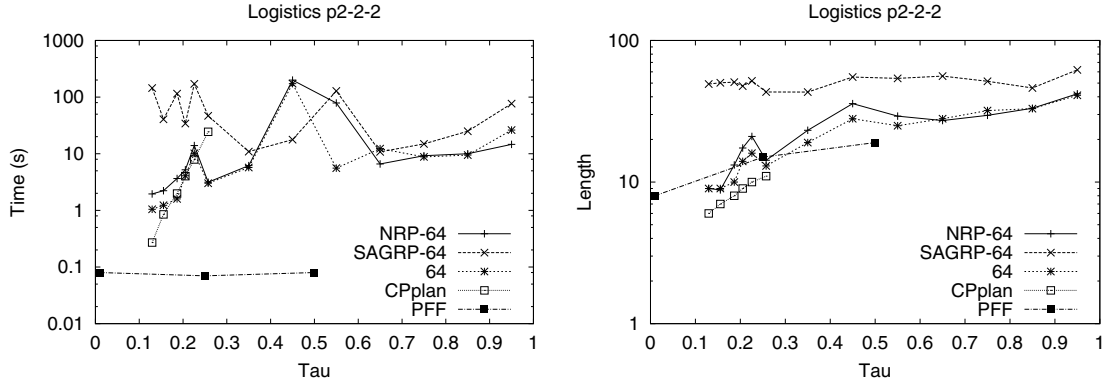


Fig. 14. Run times (s) and Plan lengths vs. τ for CPPlan Logistics p2-2-2

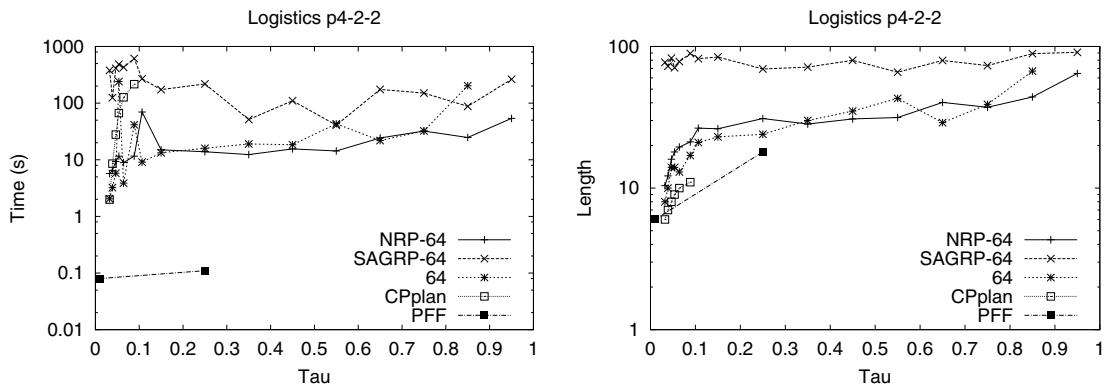


Fig. 15. Run times (s) and Plan lengths vs. τ for CPPlan Logistics p4-2-2

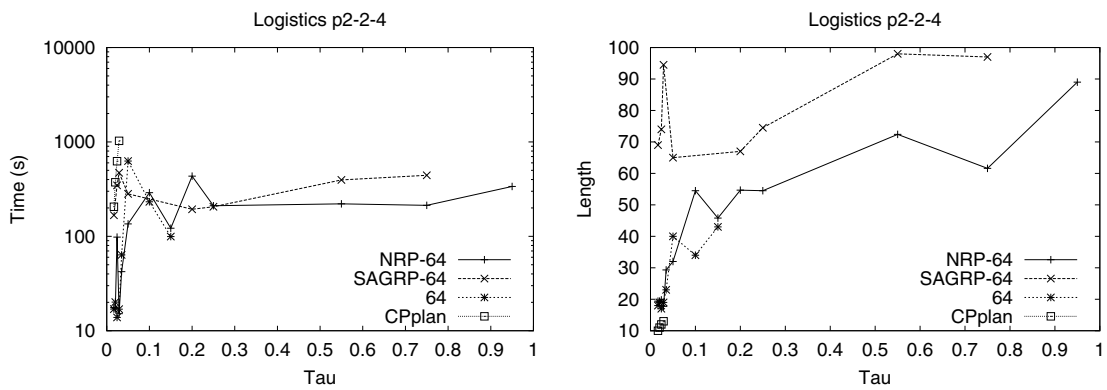


Fig. 16. Run times (s) and Plan lengths vs. τ for CPPlan Logistics p2-2-4

Figures 14 to 22 show total time in seconds and plan length results for the probabilistic planning problems. The figures compare the relaxed plan extracted from the *McSLUG* (denoted *NRP* – *N*, for per-node relaxed plan), the state agnostic relaxed plan from Section 5 (denoted *SAGRP* – *N*), the *McLUG* (denoted by *N*), and CPplan. We discuss the state agnostic relaxed plan later in this subsection and comparisons to PFF and CPplan in the next subsection. We use $N = 16$ or 64

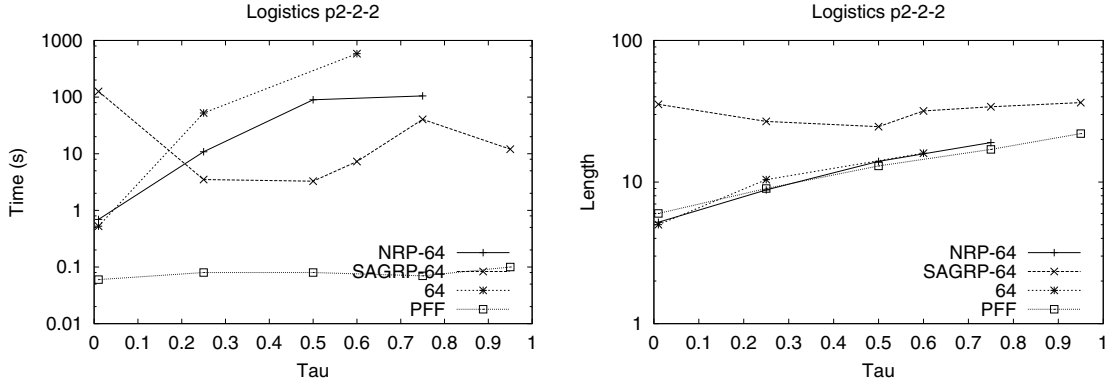


Fig. 17. Run times (s) and Plan lengths vs. τ for PFF Logistics p2-2-2

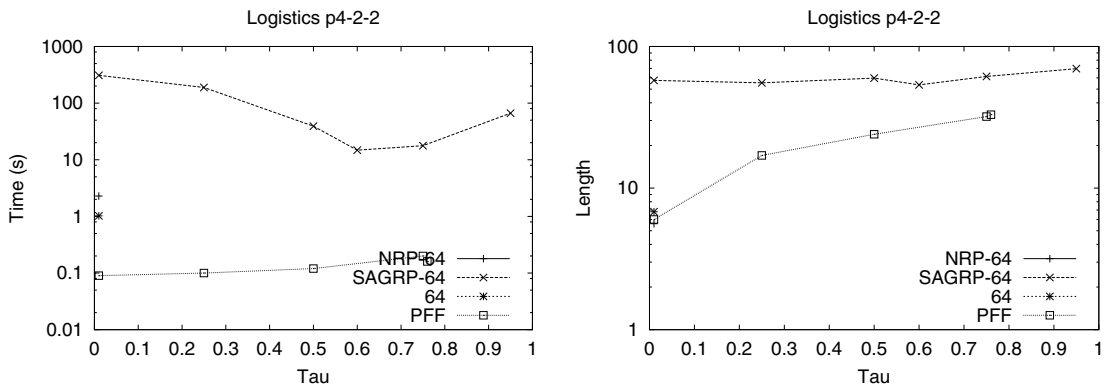


Fig. 18. Run times (s) and Plan lengths vs. τ for PFF Logistics p4-2-2

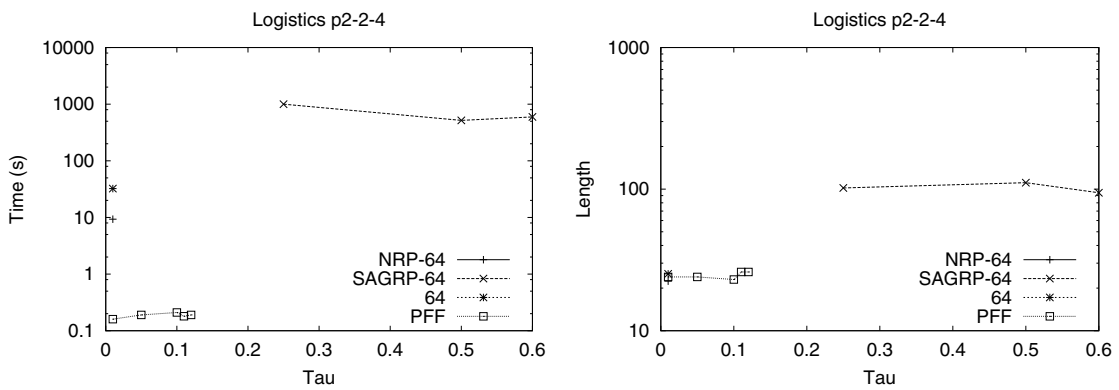


Fig. 19. Run times (s) and Plan lengths vs. τ for PFF Logistics p2-2-4

in each *McLUG* or *McSLUG* as indicated in the legend of the figures for each domain, because these numbers proved best in our evaluation.

The probabilistic Logistics instances are named with the convention $px-y-z$, where x is the number of possible locations of a package, y is the number of cities, and z is the number of packages. Figure 14 shows that the *McSLUG* improves, if

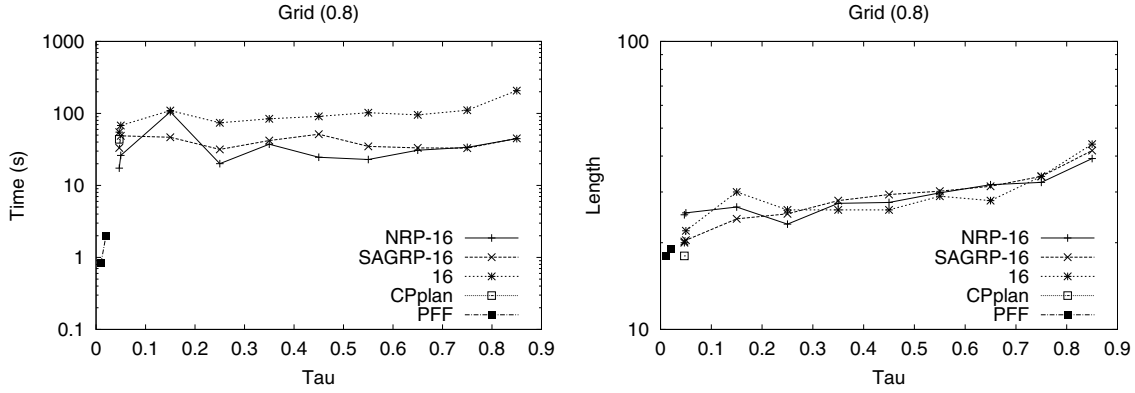


Fig. 20. Run times (s) and Plan lengths vs. τ for Grid-0.8

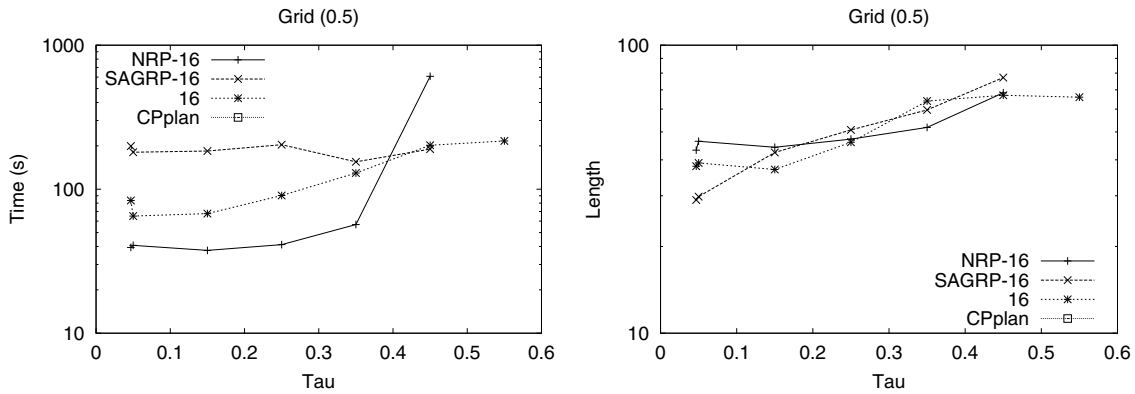


Fig. 21. Run times (s) and Plan lengths vs. τ for Grid-0.5

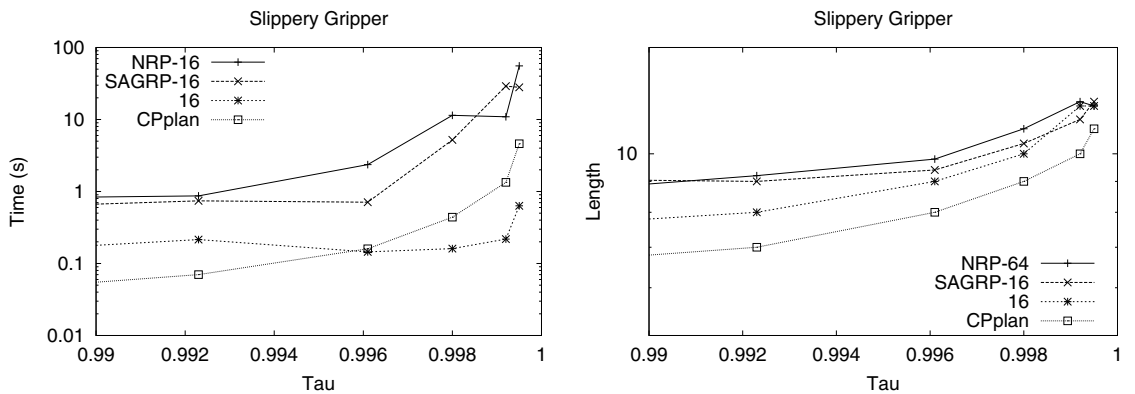


Fig. 22. Run times (s) and Plan lengths vs. τ for Slippery Gripper.

only slightly, upon the *McLUG* in the CPPlan version of Logistics p2-2-2. Figure 15 shows similar results for the CPPlan version of Logistics p4-2-2, with the *McSLUG* performing considerably better than the *McLUG* – solving the problem where $\tau = 0.95$. Figure 16 shows results for the CPPlan version of Logistics p2-2-4 that demonstrate the improvements of using the *McSLUG*, finding plans for $\tau = 0.95$, where the *McLUG* could only solve instances where $\tau \leq 0.25$. Overall,

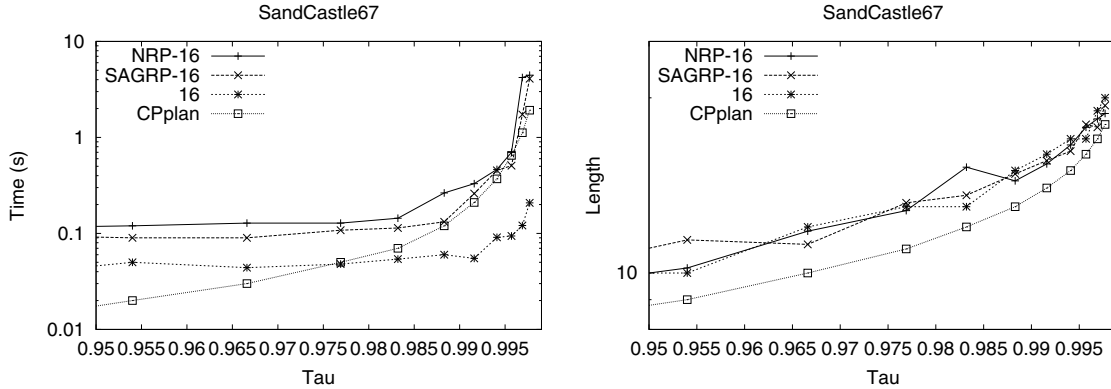


Fig. 23. Run times (s) and Plan lengths vs. τ for SandCastle-67.

using the *McSLUG* is better than the *McLUG*. With respect to the PFF versions of the Logistics instances, we see that using the *McSLUG*, regardless of how the heuristic is computed, is preferable to using the *McLUG* in these instances.

Figure 20 shows results for the Grid-0.8 domain that indicate the *McSLUG* greatly improves total planning time over the *McLUG*. However, Figure 21 shows the results are different for Grid-0.5. The *McSLUG* performs much worse than the *McLUG*. A potential explanation is the way in which the *McSLUG* shares action outcome samples among the planning graphs. The *McLUG* is more robust to the sampling because it re-samples the action outcomes for each belief state, where the *McSLUG* re-samples the action outcomes from the pre-sampled pool.

Figures 23 and 22 show results for the respective SandCastle-67 and Slippery Gripper domains, where the *McLUG* outperforms the *McSLUG*. Similar to the Grid-0.5 domain, *McSLUG* has an impoverished pool of action outcome samples that does not plague the *McLUG*. Since these problems are relatively small, the cost of computing the *McLUG* pales in comparison to the quality of the heuristic it provides. Overall, the *McSLUG* is useful when it is too costly to compute a *McLUG* for every search node, but it seems to provide less informed heuristics.

Relaxed Plans: In general, the state agnostic relaxed plan performs worse than the traditional relaxed plan both in terms of time and quality (with the exception of the PFF versions of Logistics). The heuristic that we use to extract state agnostic relaxed plans is typically very poor because it selects actions that help achieve the goal in more planning graphs. The problem is that with respect to a given set of states (needed to evaluate the heuristic for a single belief state), the chosen actions may be very poor choices. This suggests that an alternative action selection mechanism for state agnostic relaxed plans could be better, or that individualizing relaxed plan extraction to each search node is preferred. An alternative, that we do not explore, examines the continuum between state agnostic relaxed plans and node-based relaxed plans by extracting a *flexible* state agnostic relaxed plan that allows some choice to customize the relaxed plan to a specific search node.

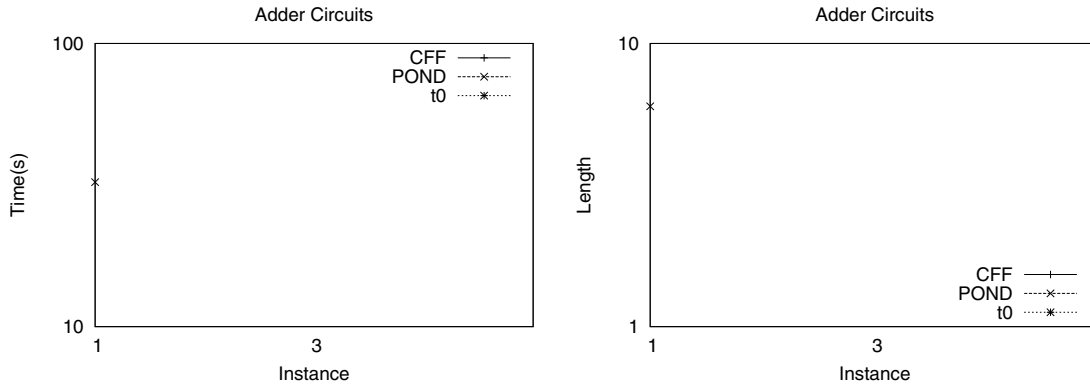


Fig. 24. Run times (s) and Plan lengths IPC5 Adder Instances.

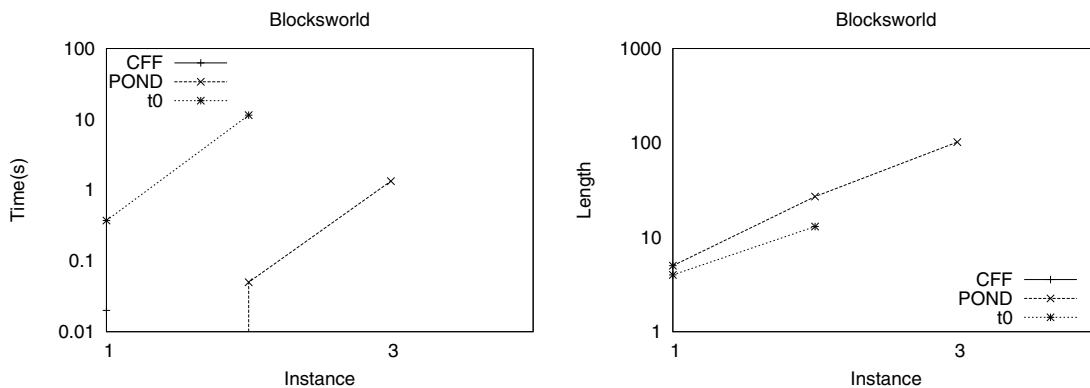


Fig. 25. Run times (s) and Plan lengths IPC5 Blocksworld Instances.

6.3 External Evaluation

In addition to internally evaluating the SAG, we evaluate how using the SAG helps *POND* compete with contemporary planners. We discuss three groups of results, the non-deterministic track of the Fifth IPC, a comparison with non-deterministic planners from the literature, and a comparison with CPplan and PFF in probabilistic planning.

Non-Deterministic Track of the Fifth IPC: We entered a version of *POND* in the non-deterministic track of the IPC that uses an enforced hill-climbing search algorithm [22], and the SAG to extract a relaxed plan at each search node. The other planners entered in the competition are Conformant FF (CFF) [21] and t0 [35]. All planners use a variation of relaxed plan heuristics, but the other planners compute a type of planning graph at each search node, rather than a SAG. To be precise, CFF computes a relaxed plan heuristic similar to that described in this work by taking into account uncertainty in the initial state, whereas t0 transforms the problem to a classical planning problem solved by FF (which computes relaxed plans at each search node).

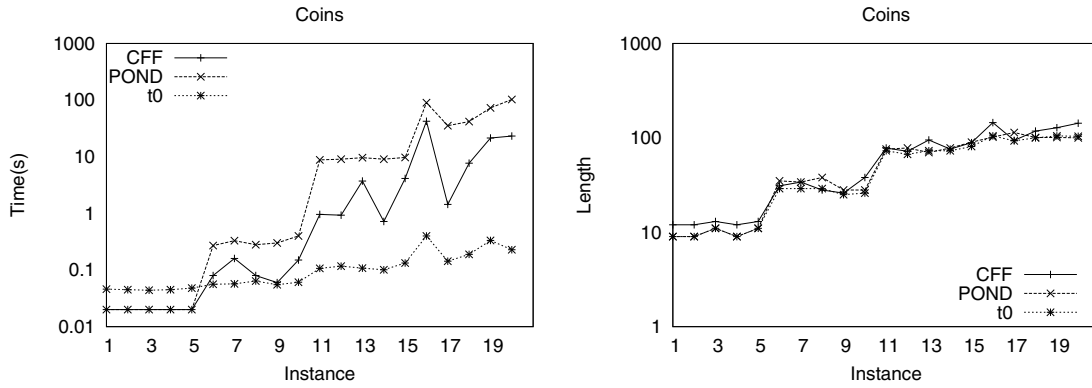


Fig. 26. Run times (s) and Plan lengths IPC5 Coins Instances.

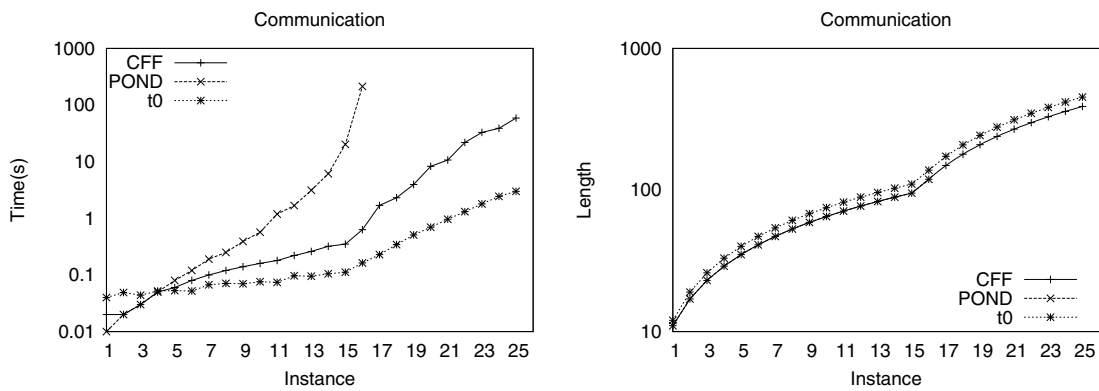


Fig. 27. Run times (s) and Plan lengths IPC5 Comm Instances.

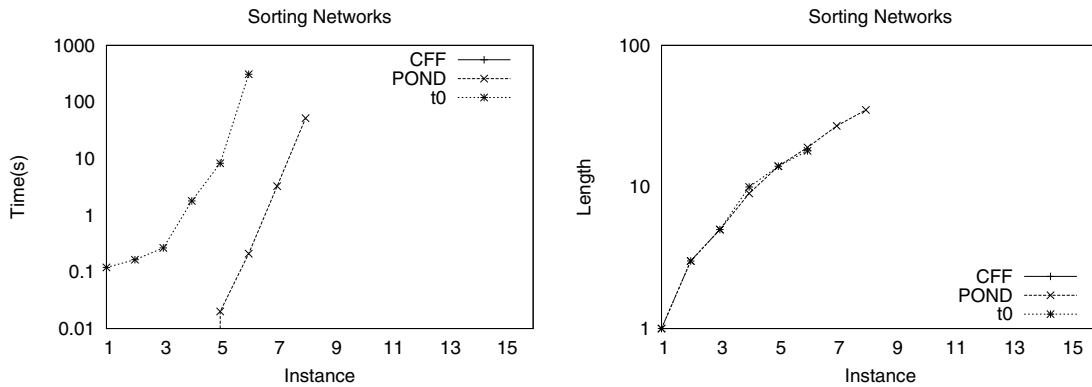


Fig. 28. Run times (s) and Plan lengths IPC5 Sortnet Instances.

Figures 24 to 29 show total time and plan length results for the six competition domains. *POND* is the only planner to solve instances in the adder domain, and it outperforms all other planners in the blockworld and sortnet domains. *POND* is competitive, but slower in the coins, communication, and universal traversal sequences domains. In most domains *POND* finds the best quality plans. Overall, *POND* exhibited good performance across all domains, as a domain-independent planner should.

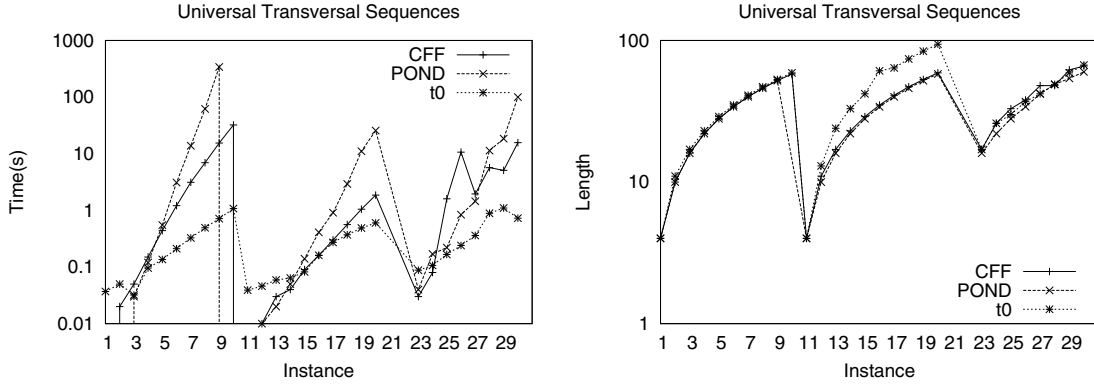


Fig. 29. Run times (s) and Plan lengths IPC5 UTS Instances.

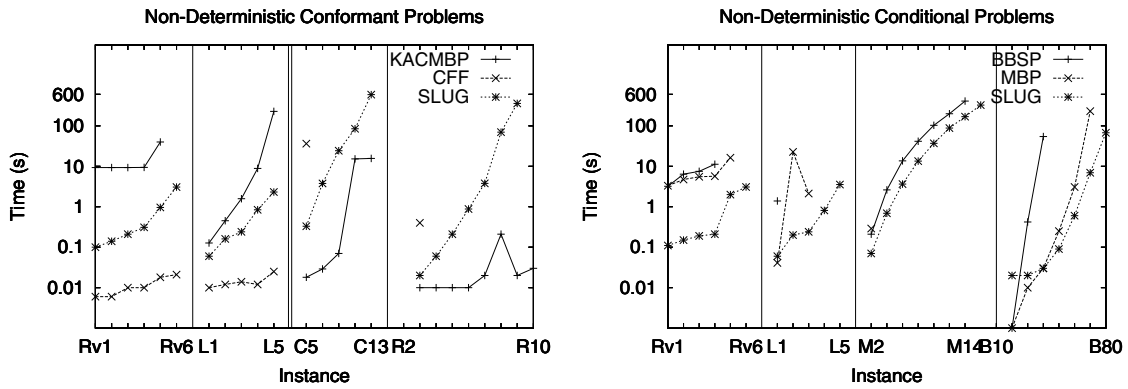


Fig. 30. Comparison of planners on conformant (left) and conditional (right) domains. Four domains appear in each plot. The conformant domains include Rovers (Rv1-Rv6), Logistics (L1-L5), Cube Center (C5-C13), and Ring (R2-R10). The conditional domains are Rovers (Rv1-Rv6), Logistics (L1-L5), Medical (M2-M14), and BTCS (B10-B80).

Additional Non-Deterministic Domains: We made an additional external comparison of *POND* with several non-deterministic conformant: KACMBP [1] and CFF [21], and conditional planners: MBP [3] and BBSP [39]. We previously mentioned that the conformant relaxed plan heuristic can be used to guide a conditional planner, which we demonstrate here using weighted AO* [34] search in belief state space. Based on the results of the internal analysis, we used relaxed plans extracted from a common *SLUG*, using the SAG strategy. We denote this mode of *POND* as “SLUG” in Figure 30. The tests depicted in Figure 30 were allowed 10 minutes on a P4 at 2.8 GHz with 1GB of memory. The planners we used for these comparisons require descriptions in differing languages. We ensured that each encoding had an identical state space; this required us to use only boolean propositions in our encodings.

We used the conformant Rovers and Logistics domains as well as the Cube Center and Ring domains [1] for the conformant planner comparison in Figure 30. These domains exhibit two distinct dimensions of difficulty. The primary difficulty in Rovers and Logistics problems centers around *causing* the goal. The Cube Cen-

ter and Ring domains, on the other hand, revolve around *knowing* the goal. The distinction is made clearer if we consider the presence of an oracle. The former pair, given complete information, remains difficult. The latter pair, given complete information, becomes trivial, relatively speaking.

We see the *SLUG* as a middle-ground between KACMBP’s cardinality based heuristic and CFF’s approximate relaxed plan heuristic. In the Logistics and Rovers domains, CFF dominates, while KACMBP becomes lost. The situation reverses in Cube Center and Ring: KACMBP easily discovers solutions, while CFF wanders. Meanwhile, by avoiding approximation and eschewing cardinality in favor of reachability, POND achieves middle-ground performance on all of the problems.

We devised conditional versions of Logistics and Rovers domains by introducing sensory actions. We also drew conditional domains from the literature: BTCS [42] and a variant of Medical [36]. Our variant of Medical splits the multi-valued stain type sensor into several boolean sensors.

The results (Figure 30) show that POND dominates the other conditional planners. This is not surprising: MBP’s and BBSP’s heuristic is belief state cardinality. Meanwhile, POND employs a strong, yet cheap, estimate of reachability (relaxed plans extracted from *SLUG*). MBP employs greedy depth-first search, so the quality of plans returned can be drastically poor. The best example of this in our results is instance Rv4 of Rovers, where the max length branch of MBP requires 146 actions compared to 10 actions for POND.

Probabilistic Planning: In the probabilistic planning problems that we previously used for internal comparison, we also compare with the PFF and CPplan planners. As Figures 14 to 23 identify, *POND* generally out scales CPplan in every domain, but sacrifices quality. CPplan is an optimal planner that exactly evaluates plan suffixes, where *POND* estimates the plan suffixes by a heuristic to guide its search. Moreover, CPplan is a bounded length planner that must be used in an iterative deepening manner to find plans that exceed the goal satisfaction threshold – leading to much redundant search.

Due to some unresolved implementation issues with the PFF planner relating to stability, we are only able to present results in the Logistics and Grid domains. Fortunately, the Logistics and Grid domains are the most revealing in terms of planner scalability. We see that PFF scales reasonably well in the CPPlan version of Logistics p2-2-2 (Figure 14), solving instances up to a probability of goal satisfaction threshold of 0.5 an order of magnitude faster than any other planner. PFF also solves instances in the CPPlan version of p4-2-2 (Figure 15) much faster, but fails to find plans for higher goal probability thresholds ($\tau > 0.25$). PFF scales even worse in the CPPlan version of p2-2-4 (Figure 15) and Grid-0.8 (Figure 20). PFF tends to scale better in the PFF version of the Logistics instances (Figures 17 to 19), but in p4-2-2 and p2-2-4 is outscaled by *POND* using the state agnostic relaxed plan

heuristic. When running PFF, it appears to expand search nodes very quickly, indicating it spends relatively little time on heuristic computation. Perhaps, PFF's poor scalability in these domains can be attributed to computing too weak of a heuristic to provide effective search guidance. *POND*, using the *McSLUG*, spends relatively more time computing its heuristic and can provide better guidance. It is not always true that spending more time on heuristic computation will lead to better scalability, but in this case it appears that the time is well spent.

7 Related Work

The state agnostic planning graph is similar to many previous works that use common structure of planning problems within planning algorithms, use precomputed heuristics during search, or speed up the construction or use of planning graphs.

Planning Algorithms: The SAG represents an all pairs relaxation of the planning problem. The work of [20] describes an all pairs solution to planning, called a universal plan. The idea implemented in the Warplan planner is to encode the current goal into the state space so that a universal plan, much like a policy, prescribes the action to perform in every world state for any current goal. The SAG can be viewed as solving a related reachability problem (in the relaxed planning space) to determine which states reach which goals.

Planning Heuristics: As previously noted, forward chaining planners often suffer from the problem of computing a reachability analysis forward from each search node, and the SAG is one way to mitigate this cost [13]. Another approach to guiding forward chaining planners is to use relaxed goal regression to compute the heuristic; work on greedy regression graphs [32] as well as the GRT system [37], can be understood this way. This backwards reachability analysis (i.e., relevance analysis) can be framed within planning graphs, avoiding the inefficiencies in repeated construction of planning graphs [26]. The main difficulty in applying such backwards planning graph approaches is the relative low quality of the derived heuristics. In addition to planning graphs, dynamic programming can be used to compute similar heuristics, but at each search node, as in HSP [5].

Pattern databases [12] have been used in heuristic search and planning [17] to store pre-computed heuristic values instead of computing them during search. The SAG can be thought of as a type of pattern database, where most of the heuristic computation cost is in building the SAG and per search node evaluation is much less expensive.

Planning Graphs: We have already noted that the SAG is a generalization of the *LUG* [9], which efficiently exploits the overlap in the planning graphs of members

of a belief state. The SAG inherits many of the properties of the *LUG*, and one option, while not explored in this work (for lack of a heuristic that can make use of mutexes), is the ability to use labels to compute mutexes that exist in common among multiple planning graphs [9].

Other works on improving heuristic computation, include [29] where the authors explore issues in speeding up heuristic calculation in HSP. Their approach utilizes the prior heuristic computation to improve the performance of building the current heuristic. We set out to perform work ahead of time in order to save computation later; their approach demonstrates how to boost performance by skipping re-initialization. Also in that vein, [30] demonstrate techniques for representing a planning graph that take full advantage of the properties of the planning graph. We seek to exploit the overlap between different graphs, not different levels. [29] seek to exploit the overlap between different graphs as well, but limit the scope to graphs adjacent in time.

The Prottle planner [28] makes use of a single planning graph to compute heuristics in a forward chaining probabilistic temporal planner. Prottle constructs a planning graph layer for every time step of the problem, up to a bounded horizon, and then back propagates numeric reachability estimates from the goals to every action and proposition in the planning graph. To evaluate a state, Prottle indexes the propositions and actions active in the state at the current time step, and aggregates their back-propagated estimates to compute a heuristic. Prottle combines forward reachability analysis with backwards relevance propagation to help to avoid recomputing the planning graph multiple times.

8 Conclusion

A common task in many planners is to compute a set of planning graphs. The naive approach fails to take account of the redundant sub-structure of planning graphs. We developed the SAG as an extension of prior work on the *LUG*. The SAG employs a labeling technique which exploits the redundant sub-structure, if any, of arbitrary sets of planning graphs.

We developed a belief-space progression planner called *POND* to evaluate the technique. We improve the use of the *LUG* within *POND* by applying our SAG technique. We found an optimized form, *SLUG*, of the state agnostic version of the *LUG*, and the *McSLUG* for the *McLUG*. These savings associated with these optimized forms carry through to the experimental results.

We also compared *POND* to state of the art planners in non-deterministic and probabilistic planning. We demonstrated that, by using *SLUG* and *McSLUG*, *POND* is highly competitive with the state of the art in belief-space planning. Given the pos-

itive results in applying SAG, we see promise in applying SAG to other planning formalisms.

Acknowledgements: This research is supported in part by the NSF grant IIS-0308139 and an IBM Faculty Award to Subbarao Kambhampati. We thank David Smith for his contributions to the foundations of our work, in addition, we thank the members of Yochan for many helpful suggestions. We also thank Nathaniel Hyafil and Fahiem Bacchus for their support in CPplan comparisons, Carmel Domshlak and Joerg Hoffmann for their support in PFF comparisons, Jussi Rintanen for help with BBSP, and Piergiorgio Bertoli for help with MBP.

References

- [1] P. Bertoli, A. Cimatti, Improving heuristics for planning as search in belief space, in: Proceedings of AIPS'02, 2002, pp. 143–152.
- [2] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Planning in nondeterministic domains under partial observability via symbolic model checking, in: IJCAI, 2001, pp. 473–486.
- [3] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Planning in nondeterministic domains under partial observability via symbolic model checking, in: Proceedings of IJCAI'01, 2001, pp. 473–478.
- [4] A. Blum, M. Furst, Fast planning through planning graph analysis, in: Proceedings of IJCAI'95, 1995, pp. 1636–1642.
- [5] B. Bonet, H. Geffner, Planning as heuristic search: New results, in: Proceedings of ECP'99, 1999, pp. 360–372.
- [6] R. Brafman, J. Hoffmann, Contingent planning via heuristic forward search with implicit belief states, in: Proceedings of ICAPS'05, 2005.
- [7] D. Bryce, W. Cushing, S. Kambhampati, Model-lite planning: Diverse mult-option plans and dynamic objective functions, in: Proceedings of the 3rd Workshop on Planning and Plan Execution for Real-World Systems, 2007.
- [8] D. Bryce, S. Kambhampati, A tutorial on planning graph based reachability heuristics, *AI Magazine* 28 (1) (2007) 47–83.
- [9] D. Bryce, S. Kambhampati, D. Smith, Planning graph heuristics for belief space search, *Journal of AI Research* 26 (2006) 35–99.
- [10] D. Bryce, S. Kambhampati, D. Smith, Sequential monte carlo in probabilistic planning reachability heuristics, in: Proceedings of ICAPS'06, 2006.
- [11] D. Bryce, S. Kambhampati, D. Smith, Sequential monte carlo in probabilistic planning reachability heuristics, *Artificial Intelligence* 172(6-7) (2008) 685–715.

- [12] J. C. Culberson, J. Schaeffer, Pattern databases, *Computational Intelligence* 14 (3) (1998) 318–334.
- [13] W. Cushing, D. Bryce, State agnostic planning graphs, in: *Proceedings of AAAI'05*, 2005, pp. 1131–1138.
- [14] A. Darwiche, P. Marquis, A knowledge compilation map, *Journal of Artificial Intelligence Research* 17 (2002) 229–264.
- [15] J. de Kleer, An Assumption-Based TMS, *Artificial Intelligence* 28 (2) (1986) 127–162.
- [16] C. Domshlak, J. Hoffmann, Fast probabilistic planning through weighted model counting, in: *Proceedings of ICAPS'06*, 2006, pp. 243–251.
- [17] S. Edelkamp, Planning with pattern databases, in: *Proceedings of the 6th European Conference on Planning (ECP-01)*, 2001, pp. 13–24.
- [18] A. Gerevini, B. Bonet, R. Givan (Eds.), *5th International Planning Competition*, Cumbria, UK., 2006.
- [19] 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.
- [20] E. Harwood, D. Warren, *Warplan: A system for generating plans.*, Tech. Rep. Memo 76, Computational Logic Dept., School of AI, Univ. of Edinburgh. (1974).
- [21] J. Hoffmann, R. Brafman, Conformant planning via heuristic forward search: A new approach, in: *Proceedings of ICAPS'04*, 2004, pp. 355–364.
- [22] J. Hoffmann, B. Nebel, The FF planning system: Fast plan generation through heuristic search, *Journal of Artificial Intelligence Research* 14 (2001) 253–302.
- [23] J. Huang, Combining knowledge compilation and search for efficient conformant probabilistic planning, in: *Proceedings of ICAPS'06*, 2006, pp. 253–262.
- [24] N. Hyafil, F. Bacchus, Conformant probabilistic planning via CSPs, in: *Proceedings of ICAPS' 03*, 2003, pp. 205–214.
- [25] N. Hyafil, F. Bacchus, Utilizing structured representations and CSPs in conformant probabilistic planning, in: *Proceedings of ECAI'04*, 2004, pp. 1033–1034.
- [26] S. Kambhampati, E. Parker, E. Lambrecht, Understanding and extending graphplan., in: *Proceedings of Fourth European Conference on Planning*, 1997, pp. 260–272.
- [27] J. Koehler, B. Nebel, J. Hoffmann, Y. Dimopoulos, Extending planning graphs to an ADL subset, in: *Proceedings of Fourth European Conference on Planning*, 1997, pp. 273–285.
- [28] I. Little, D. Aberdeen, S. Theibaux, Prottle: A probabilistic temporal planner, in: *Proc. of AAAI'05*, 2005, pp. 1181–1186.
- [29] Y. Liu, S. Koenig, D. Furcy, Speeding up the calculation of heuristics for heuristic search-based planning., in: *Proceedings of the National Conference on Artificial Intelligence*, 2002, pp. 484–491.

- [30] D. Long, M. Fox, Efficient implementation of the plan graph in stan., *Journal of AI Research* 10 (1999) 87–115.
- [31] D. Long, M. Fox, The 3rd international planning competition: Results and analysis, *Journal of Artificial Intelligence Research* 20 (2003) 1–59.
- [32] D. V. McDermott, Using regression-match graphs to control search in planning, *Artificial Intelligence* 109 (1-2) (1999) 111–159.
URL citeseer.ist.psu.edu/mcdermott99using.html
- [33] X. Nguyen, S. Kambhampati, R. S. Nigenda, Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search, *Artificial Intelligence* 135 (1-2) (2002) 73–123.
- [34] N. Nilsson, *Principles of Artificial Intelligence*, Morgan Kaufmann, 1980.
- [35] H. Palacios, H. Geffner, Compiling uncertainty away: Solving conformant planning problems using a classical planner (sometimes), in: *Proceedings of the National Conference on Artificial Intelligence*, AAAI Press, 2006.
- [36] R. Petrick, F. Bacchus, A knowledge-based approach to planning with incomplete information and sensing, in: *Proceedings of AIPS'02*, 2002, pp. 212–222.
- [37] I. Refanidis, I. Vlahavas, The GRT planning system: Backward heuristic construction in forward state-space planning, *Journal of Artificial Intelligence Research* 15 (2001) 115–161.
URL citeseer.ist.psu.edu/refanidis01grt.html
- [38] J. Rintanen, Expressive equivalence of formalisms for planning with sensing, in: *Proceedings of ICAPS'03*, 2003, pp. 185–194.
- [39] J. Rintanen, Conditional planning in the discrete belief space, in: *Proceedings of IJCAI'05*, 2005, pp. 1260–1265.
- [40] D. Smith, D. Weld, Conformant graphplan, in: *Proceedings of AAAI'98*, 1998, pp. 889–896.
- [41] F. Somenzi, CUDD: CU Decision Diagram Package Release 2.3.0, University of Colorado at Boulder (1998).
- [42] D. Weld, C. Anderson, D. Smith, Extending graphplan to handle uncertainty and sensing actions, in: *Proceedings of AAAI'98*, 1998, pp. 897–904.
- [43] H. Younes, R. Simmons, Vhpop: Versatile heuristic partial order planner, *Journal of Artificial Intelligence Research* 20 (2003) 405–430.
- [44] T. Zimmerman, S. Kambhampati, Using memory to transform search on the planning graph, *Journal of AI Research* 23 (2005) 533–585.