Journal of Artificial Intelligence Research volume (year) pages Submitted 3/08; published 3/08

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

Daniel Bryce BRYCE@AI.SRI.COM
SRI International, Artificial Intelligence Center
333 Ravenswood Ave., Menlo Park, CA 94025

William Cushing CUSHING@ASU.EDU
Subbarao Kambhampati RAO@ASU.EDU
Department of Computer Science and Engineering
Ira A. Fulton School of Engineering
Arizona State University, 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 an all-pairs shortest path problem by iterating a single-source algorithm over each source.

We introduce a structure, the state agnostic planning graph, that directly solves the all-pairs 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. A more prominent application of this technique is in belief state space planning, where an optimization to exploit state overlap between belief states results in drastically improved theoretical complexity. We describe another extension in probabilistic planning that uses common action outcome uncertainty to further improve theoretical complexity. 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.

1. Introduction

Heuristics derived from planning graphs (Blum & Furst, 1995) are widespread in planning (Gerevini, Saetti, & Serina, 2003; Hoffmann & Nebel, 2001; Bonet & Geffner, 1999; Younes & Simmons, 2003; Nguyen, Kambhampati, & Nigenda, 2002; Bryce & Kambhampati, 2007). A planning graph represents a relaxed look-ahead of the state space that identifies propositions reachable at different depths. Planning graphs are typically layered graphs of vertices $(\mathcal{P}_0, \mathcal{A}_0, \mathcal{P}_1, \mathcal{A}_1, ..., \mathcal{A}_{k-1}, \mathcal{P}_k)$, where each level t contains a proposition layer $\mathcal{P}_t$ and an action layer $\mathcal{A}_t$. In many cases, heuristics are derived from a *set* of planning graphs. In deterministic planning, progression planners typically compute a planning graph for every search 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 successor state is a sub-graph of the planning graph of the preceding state, left-shifted by one step (Zimmerman & Kambhampati, 2005). Computing a set of planning graphs by enumerating its members is, therefore, inherently redundant.

$P = \{$`at(L1), at(L2), have(I1), comm(I1)`$\}$
$A = \{$ `drive(L1, L2)` $= (\{$`at(L1)`$\}, \{$`at(L2)`$\}, \{$`at(L1)`$\}),$
`drive(L2, L1)` $= (\{$`at(L2)`$\}, \{$`at(L1)`$\}, \{$`at(L2)`$\}),$
`sample(I1, L2)` $= (\{$`at(L2)`$\}, \{$`have(I1)`$\}, \{\}),$
`commun(I1)` $= (\{$`have(I1)`$\}, \{$`comm(I1)`$\}, \{\})\}$
$I = \{$`at(L1)`$\}$
$G = \{$`comm(I1)`$\}$

Figure 1: Deterministic Planning Problem Example.

Consider progression planning in a deterministic 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 `comm(I1)`, having communicated the picture back to a lander. There are four actions `drive(L1, L2)`, `drive(L2, L1)`, `sample(I1, L2)`, and `commun(I1)`. The rover can use the plan: `drive(L1, L2)`, `sample(I1, L2)`, `commun(I1)` to achieve the goal. The sequence of states corresponding to the plan, where each state will be evaluated by the heuristic, is:
$s_I = \{$`at(L1)`$\}$
$s_1 = \{$`at(L2)`$\}$
$s_2 = \{$`at(L2), have(I1)`$\}$
$s_3 = \{$`at(L2), have(I1), commun(I1)`$\}$
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 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 and planning graphs for several states.

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 an elegant generalization of the planning graph called the State Agnostic Graph (SAG). The general technique (of which we will describe several variations) is to represent a single planning graph skeleton to capture action and proposition connectivity (for preconditions and effects) and use propositional formulas, called labels, to annotate which portions of the skeleton relate to which states. Our techniques are related to work on assumption based truth maintenance systems (de Kleer, 1986), 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 diverse set of 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 all-pairs shortest path problem. We develop the SAG under the following intuition: directly solving the all-pairs shortest path problem is more efficient than iterating a single source algorithm.

This intuition falls short in most planning problems, because the majority of states are unreachable. Reasoning about such states is useless, so instead, we develop the SAG as a solution to the

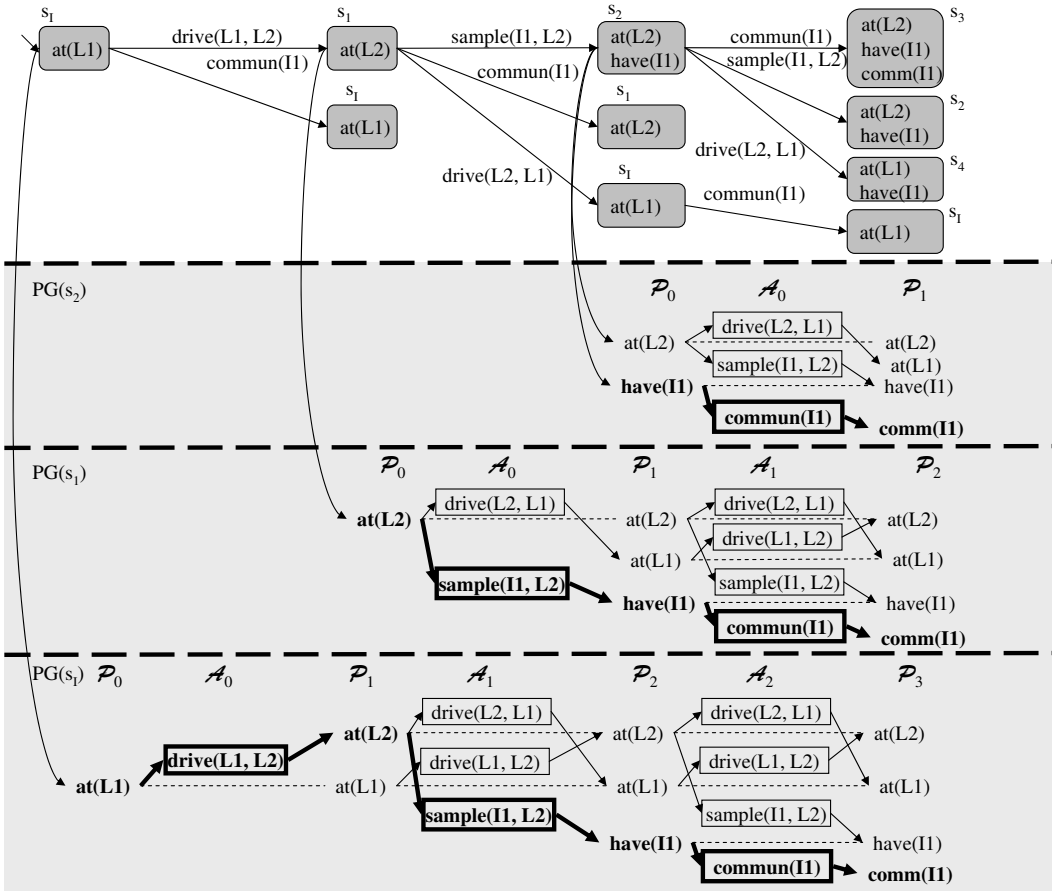

Figure 2: Planning graphs and state space projection tree.

multi-source shortest path problem. More precisely, the SAG is a representation of a set of planning graphs. The technique is to represent the propagation rules of the planning graph and a set of sources as boolean functions (labels) and to compose these functions instead of evaluating them. Composition via boolean algebra yields a symbolic approach for building the set of planning graphs without explicitly enumerating its elements. This boosts empirical performance by exploiting redundant sub-structure, in despite of the fact that the technique does not alter worst-case computational complexity.

Planning Graph	Deterministic	Non-Deterministic	Probabilistic
Traditional	PG	LUG	McLUG
State Agnostic	SAG	SLUG	CSSAG

Table 1: Planning Graph Taxonomy for Types of Planning.

The exact form of the label functions is specific to the type of planning graph being generalized (see Table 1 for a taxonomy of the planning graphs generalized in this work). The SAG represents a set of relaxed planning graphs in deterministic planning, a set of labeled planning graphs (*LUG*) (Bryce, Kambhampati, & Smith, 2006) in non-deterministic planning, and a set of Monte Carlo la-

beled planning graphs ($\mathcal{M}cLUG$) (Bryce, Kambhampati, & Smith, 2008) in probabilistic planning. (The LUG and $\mathcal{M}cLUG$ generalize planning graphs to compute heuristics for non-deterministic and probabilistic belief states and probabilistic actions). Labels represent sets of states in deterministic planning, sets of belief states in non-deterministic planning, and sets of particles (samples) in probabilistic planning. In non-deterministic planning we can also overload the labels to represent belief states in terms of their constituent states, arriving at an exponentially smaller set of label symbols in the $SLUG$. In probabilistic planning we can use the same overloading and additionally simulate each planning graph through a set of *common samples* in the $\mathcal{CS}SAG$, exponentially reducing the sampling complexity over a naive adaptation of the $\mathcal{M}cLUG$ to its SAG version.

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 5.1 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, we consider several strategies for reducing irrelevant heuristic computations in Section 5.2; 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 deterministic 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 (the false propositions are inferred by the closed world assumption) and the goal is a set of propositions $G \subseteq P$ that must be made true in a state s for s to be a goal state. Each action $a \in A$ is described by $(\rho_e(a), (\varepsilon^+(a), \varepsilon^-(a)))$, where the execution precondition $\rho_e(a)$ is a set of propositions, and $(\varepsilon^+(a), \varepsilon^-(a))$ is an effect where $\varepsilon^+(a)$ is a 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 $s' = exec(a, s) = s \backslash \varepsilon^-(a) \cup \varepsilon^+(a)$. A sequence of actions $\{a_1, ..., a_m\}$, executed in state s, results in a state s', where $s' = exec(\{a_1, ..., a_m\}, s) = exec(a_m, exec(a_{m-1}, ... \ exec(a_1, s) \ ...))$ 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

$s = \{p_1, ..., p_n\} \subseteq P$ represented in set notation is equivalent to a logical state $\hat{s} = p_1 \wedge ... \wedge p_n \wedge \neg p_{n+1} \wedge ... \wedge \neg p_m$, where $P \backslash s = \{p_{n+1}, ..., p_m\}$.

The 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 results in the state: $s_1 = exec(\texttt{drive(L1, L2)}, s_I) = \{\texttt{at(L2)}\}$. The state s_1 can be represented as $\hat{s}_1 = \neg\texttt{at(L1)} \wedge \texttt{at(L2)} \wedge \neg\texttt{have(I1)} \wedge \neg\texttt{comm(I1)}$. In the following, we drop the distinction between set (s) and logic notation ($\hat{s}$) because the context will dictate the appropriate representation.

The most popular heuristic search formulation, 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 $(\mathcal{P}_0(s), \mathcal{A}_0(s), \mathcal{P}_1(s), \mathcal{A}_1(s), ..., \mathcal{A}_{k-1}(s), \mathcal{P}_k(s))$, 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 for action layers $\mathcal{A}_t$ and proposition layers $\mathcal{P}_t$ also assumes that the state s is implicit.

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

1. If p holds in s then $p \in \mathcal{P}_0$
2. For any t such that $p \in \mathcal{P}_t$, for every $p \in \rho_e(a)$, then $a \in \mathcal{A}_t$
3. For any t such that $a \in \mathcal{A}_t$, then $p \in \mathcal{P}_{t+1}$ for all $p \in \varepsilon^+(a)$

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 made true by the 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 $\mathcal{P}_{t-1}$ persist to $\mathcal{P}_t$. 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). (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.

RPExtract($PG(s), G$)
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: $\quad \mathcal{P}_k^{RP} \leftarrow \mathcal{P}_k^{RP} \cup p$
4: **end for**
5: **for** $t = k...1$ **do**
6: $\quad$ **for all** $p \in \mathcal{P}_t^{RP}$ **do** {Find Supporting Actions}
7: $\quad\quad$ Find $a \in \mathcal{A}_{t-1}$ such that $p \in \varepsilon^+(a)$
8: $\quad\quad \mathcal{A}_{t-1}^{RP} \leftarrow \mathcal{A}_{t-1}^{RP} \cup a$
9: $\quad$ **end for**
10: $\quad$ **for all** $a \in \mathcal{A}_{t-1}^{RP}, p \in \rho_e(a)$ **do** {Insert Preconditions}
11: $\quad\quad \mathcal{P}_{t-1}^{RP} \leftarrow \mathcal{P}_{t-1}^{RP} \cup p$
12: $\quad$ **end for**
13: **end for**
14: **return** $(\mathcal{P}_0^{RP}, \mathcal{A}_0^{RP}, \mathcal{P}_1^{RP}, ..., \mathcal{A}_{k-1}^{RP}, \mathcal{P}_k^{RP})$

Figure 3: Relaxed Plan Extraction Algorithm.

Consider one of the most popular and effective heuristics, which is based on relaxed plans. Through a simple back-chaining algorithm (Figure 3) called *relaxed plan extraction*, it is possible to identify the actions in each level that can be used to support the goals. Relaxed plans are subgraphs $(\mathcal{P}_0^{RP}, \mathcal{A}_0^{RP}, \mathcal{P}_1^{RP}, ..., \mathcal{A}_{k-1}^{RP}, \mathcal{P}_k^{RP})$ 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 actions may conflict.

Figure 3 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 three 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 every vertex of the graph with a label $\ell_t(\cdot)$. The labels permit multiple source states by tracking the set of sources reaching the associated graph vertices. Each label describes a set of source states with a propositional sentence over the domain propositions. Intuitively, a source state s reaches a graph vertex x if $s \models \ell_t(x)$ (i.e., s is a model of the label formula $\ell_t(x)$). The set of possible sources is defined by the scope of the SAG,

denoted $\mathcal{S}$, also a propositional sentence. Each SAG vertex label $\ell_t(x)$ denotes a set of states that is a subset of the scope, meaning that $\ell_t(x) \models \mathcal{S}$.

The graph $SAG(\mathcal{S}) = \langle(\mathcal{P}_0, \mathcal{A}_0, ..., \mathcal{A}_{k-1}, \mathcal{P}_k), \ell\rangle$ 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 following holds:

1. If $p \in s$, then $s \models \ell_0(p)$ and $p \in \mathcal{P}_0$
2. If $s \models \ell_t(p)$ for every $p \in \rho_e(a)$, then:
   (a) $s \models \ell_t(a)$
   (b) $a \in \mathcal{A}_t$
3. If $p \in \varepsilon^+(a)$ and $s \models \ell_t(a)$, then $s \models \ell_{t+1}(p)$ and $p \in \mathcal{P}_{t+1}$

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. An explicit 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 (e.g., $\ell_t(p) = \bigvee_{p \in \mathcal{P}_t(s)} s$). Performing such an enumeration renders the SAG more or less pointless: no work is saved. Any practical approach avoids enumerating states (and their corresponding planning graphs) to construct the SAG. We note that actions appear in all action layers where *all of their preconditions hold* in the preceding proposition layer (a conjunction, see 2. below), and that propositions appear in proposition layers where *there exists an action giving it as an effect* in the previous action layer (a disjunction, see 3. below). We directly compute the label function, using the following propagation rules:[1]

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. $k =$ minimum level t such that $\ell_t(p) = \ell_{t+1}(p), \forall p \in \mathcal{P}_t$

The label of each initial proposition layer proposition indicates the source states in which it holds. The label of each action layer action is the conjunction of the action's preconditions' labels. The label of each later proposition layer proposition is the disjunction of supporting actions from the previous layer. The index of the last proposition layer is the first level where the succeeding proposition layer is identical.

Figure 4 depicts the SAG for the rover 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

1. The graph structure is implied by the label function: an element exists in the graph if and only if its label is non-false. Many labels are false, at least at early levels, so we retain the graph structure as an efficient means of accessing non-false labels.

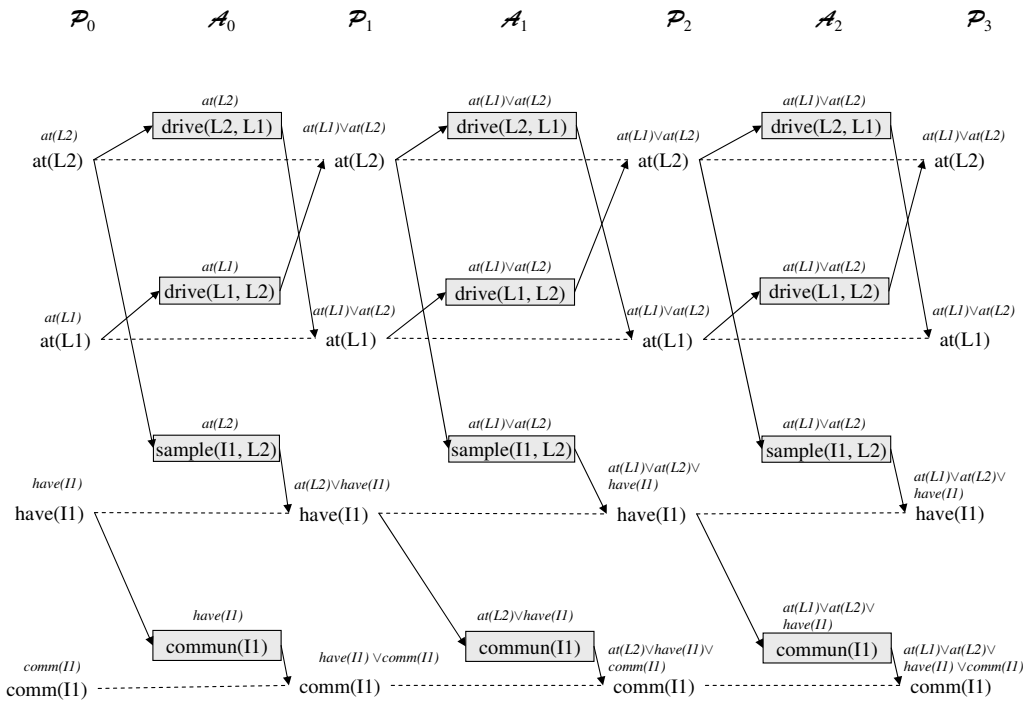

Figure 4: SAG for rover example.

italics above actions and propositions. By the third level the graph has leveled off, that is, the fourth and subsequent levels are identical to the third. At that point the goal is considered reachable by every state except $s_u = \neg\texttt{at(L1)} \wedge \neg\texttt{at(L2)} \wedge \neg\texttt{have(I1)} \wedge \neg\texttt{comm(I1)}$; the goal is labeled $\ell_3(\texttt{comm(I1)}) = \texttt{at(L1)} \vee \texttt{at(L2)} \vee \texttt{have(I1)} \vee \texttt{comm(I1)}$, the negation of which is s_u. Planning graphs over-approximate reachability, so it is certainly possible that some of the states satisfying $\ell_3(\texttt{comm(I1)}) = \texttt{at(L1)} \vee \texttt{at(L2)} \vee \texttt{have(I1)} \vee \texttt{comm(I1)}$ cannot, in fact, reach the goal.[2] What can be concluded with certainty is that states not entailing the goal's label, at some level, truly cannot reach the goal in that many parallel steps. In the case of s_u, this lower bound can be taken as ∞ since $\forall t, s_u \not\models \ell_t(\texttt{comm(I1)})$. For other states, this lower bound, the level heuristic, is given by the formula: $\min_t s \models (\bigwedge_{p \in G} \ell_t(p))$.

Of special interest are heuristics based on extracting relaxed plans; 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. Naturally our goal is to ensure that the relaxed plans extracted from the SAG are identical to the relaxed plans extracted from the planning graph for s. Guaranteeing such equivalence

2. Coincidentally the approximation of reachability of the goal happens to be exact for this tiny problem (all states besides s_u do reach the goal). However, the whole approximation is not exact: states satisfying $\texttt{at(L1)} \wedge \texttt{at(L2)}$ are not in fact reachable, but considered so.

amounts to ignoring every other planning graph represented by the SAG. The efficient approach (to ignoring planning graphs irrelevant to the current purpose) is to ensure that $s \models \ell_t(x)$ whenever considering some vertex x in the SAG. For the purpose of extracting relaxed plans, it is sufficient to enforce this constraint just when picking the support for subgoals (since actions demand that all preconditions hold). That is, when picking support for a subgoal p at level t, we (lazily) remove from consideration any action $a \in \mathcal{A}_{t-1}$ that fails to satisfy $s \models \ell_{t-1}(a)$.

For example, to evaluate the relaxed plan heuristic for state $s_1 = \{\texttt{at(L2)}\}$, one might 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 in the initial action layer of the planning graph for s_1: $s_1 \not\models \ell_0(\texttt{commun(I1)})$. By ensuring that $s_1 \models \ell_t(a)$ holds for each (a, t) in the relaxed plan (and leaving all other details of extraction unchanged), we guarantee that the result is identical to the relaxed plan extracted from a normal planning graph built for s_1. The change to the relaxed plan extraction procedure (in Figure 3) 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 $\ell_t(\texttt{comm(I1)})$. Any state s where $\texttt{comm(I1)} \in s$ has a level heuristic of zero because $\ell_0(\texttt{comm(I1)}) = \texttt{comm(I1)}$. Any state s, where $\texttt{comm(I1)} \in s$ or $\texttt{have(I1)} \in s$, has a level heuristic of one because $\ell_1(\texttt{comm(soil)}) = \texttt{comm(I1)} \vee \texttt{have(I1)}$, and so on for states modeled by 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. Later we will discuss various points along the continuum between computing all heuristic values before search and computing the heuristic at each search node.

Label Implementation: At the heart of the SAG is representation of and inference upon boolean formulas (labels). Very careful consideration needs to be given to these implementation details. Recall that the competing approach to SAG in forward state space search, is building one planning per search node. To apply SAG in this context one must take an up-front guess at what states the search will end up visiting (discussed in more detail in Section 5.2). This may be a gross overestimate, since the best available heuristic for estimating search behavior is the estimate given by planning graphs.

So while there is a great deal of redundant structure to exploit, a SAG approach starts with the handicap of constructing a potentially larger set of planning graphs. The question becomes: does symbolic planning graph representation and inference (via boolean functions) offset this handicap? To answer the question, the first observation to make is that satisfiability of boolean formulas (a key part of the inferences needed in SAG) is not NP-hard – satisfiability of boolean formulas *in Conjunctive Normal Form* is NP-hard. Considering the operations needed within the SAG, sheds light on the appropriate boolean function representation. In general, different normal forms allow

different sets of operations to be polynomial vs. NP-hard.[3] Specifically, we must consider which and how often different operations are needed by the SAG; these include:

- Conjunction (action labels)
- Disjunction (proposition labels)
- Satisfiability (reachability)
- Equivalence (level-off)

At every reachable graph vertex we need to perform either a conjunction or a disjunction to compute the label. In order to extract heuristics, all the labels of vertices included in the heuristic computation have to be checked for reachability. For example, for relaxed plans, the label of every element of the relaxed plan must indicate the element is reachable from the given state. Reachability checks are computed as entailment (the state must entail the label), which is equivalent to checking unsatisfiability of the conjunction of the state with the negation of the label. Finally, checking equivalence of every label at a given level with the corresponding label at the prior level is needed to detect level-off of the graph, if the structure is being built that far.

In short, satisfiability will be the dominating operation in terms of sheer number of invocations (many times per search node, as compared to once per potentially reachable planning graph vertex). So the ideal implementation pushes as much effort as possible away from satisfiability into conjunctions and disjunctions, with significant bonuses if equivalence ends up being cheap. While there are presumably more conjunctions than disjunctions (more actions than propositions, and entailment can be reduced through conjunction), there are still many disjunctions to perform, and there is a fair amount of nesting. Therefore, it might be a good idea to avoid a normal form that has a strong bias in favor of either conjunctions or disjunctions.

CNF, for example, does not seem like a good choice; satisfiability is NP-hard. DNF looks better on paper; satisfiability is trivial. However, DNF is very similar to reasoning with an explicit set of models, which is the representation of the competing approach (i.e., that of traditional planning graphs). Since the competing approach enjoys the significant advantage of only building the set of planning graphs that are needed, not some gross overestimate, DNF seems an unpromising route.

We represent and manipulate labels using Ordered Binary Decision Diagrams (OBDDs) (Bryant, 1986). In terms of structure, these are directed acyclic graphs (DAGs), and formulas are represented by pointers to the appropriate node. Specifically we utilize the CUDD package (Somenzi, 1998), which incorporates a great many optimizations. For our purposes, the desirable characteristics are:

- A formula and its negation are the same DAG; negation is a reserved bit in the pointers.
- Conjunction and disjunction are the same cost ($f \wedge g = \neg(\neg f \vee \neg g)$).
- Equivalence is a single operation; if two formulas are equivalent they point to the same place.
- Satisfiability is a special case of non-equivalence.

3. As a concrete example, satisfiability in Disjunctive Normal Form is trivial. Conjunction is hard within DNF, in particular, repeated conjunction is NP-hard. Consider a DNF on n clauses. Each clause, c_i, is in DNF, so evaluating $f = \bigwedge_{i=1}^{n} c_i$ entirely within DNF must be NP-hard. While conjunction is hard within DNF, disjunction (not surprisingly) is easy. In CNF the roles are reversed: conjunction is easy and disjunction is hard.

- Entailment is, at worst, two operations (negation and satisfiability) worse than a conjunction.
- Any DAG is stored at most once in memory. If the representation of f occurs within g, f points to the appropriate internal node of g.

That is, satisfiability, entailment, and equivalence are all much cheaper in an OBDD representation than in DNF or CNF. The price is that repeated conjunctions/disjunctions, in the worst case, result in diagrams of exponential size. But equivalence and satisfiability remain fast even on huge diagrams, and, many practical problems do not induce explosions in size. Further, this is the desired result: one cannot avoid paying a price somewhere, and the goal was to move as much of the expensive computation to the SAG construction phase as possible, that is, to shift computational difficulty to repeated conjunctions/disjunctions.

Consider the running example. The search tree for the rover problem has a total of six reachable states, five of which are actually generated during search. By constructing a planning graph for each state, the total number of planning graph vertices (for propositions and actions) that must be visited is 56. Constructing the SAG for the set of all states requires 28 planning graph vertices. There are a total of 10 unique boolean functions in this SAG, but many end up being sub-functions of one another under the first variable order we tried (`at(L1)`, `at(L2)`, `have(I1)`, and `comm(I1)`). Under that variable order, the total size of the shared OBDD representation is 10 vertices, not counting the special vertex denoting $\top$. So the SAG approach ends up with a 38+1 vertex structure as compared to the 45 or so vertices considered by the approach which builds planning graphs one by one. Counting vertices in this way is at best a weak measure of actual performance, but it is quite telling that even in this tiny problem the approach seems viable. Especially as we made no effort to exclude the 10 unreachable states from consideration, nor any effort to optimize the variable order (though perhaps we picked a great order by accident), and the entire technique is aimed at improving performance on large, not small, problems.

Ultimately we do improve the performance of our planner $POND$ on deterministic problems by employing the SAG approach. This is an exciting result, but $POND$ is designed as a belief space planner, not a state space planner. So the results are unlikely to be directly applicable to, say, improving FF (Hoffmann & Nebel, 2001).[4] Our aim within deterministic planning is primarily to explain the concept in a simpler context before moving on to the problems of real interest: belief-space planning problems.

3. Non-Deterministic Planning

This section extends the deterministic planning model to consider non-deterministic state uncertainty with no observability, 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 deterministic planning, P is a set of propositions, A is a set of actions, and G is a goal description. Extending the

4. Though there is some promise to the idea of further developing the idea specifically for deterministic planning. For example, one could aim for a problem agnostic formulation of the planning graph heuristic; that is, one could aim to amortize heuristic computation cost across problems.

$P = \{$`at(L1), at(L2), have(I1), comm(I1)`$\}$
$A = \{$ `drive(L1, L2)` $= (\{$`at(L1)`$\}, \{(\{\}$ $\rightarrow$ $(\{$`at(L2)`$\}, \{$`at(L1)`$\})\})\}),$
`drive(L2, L1)` $= (\{$`at(L2)`$\}, \{(\{\}$ $\rightarrow$ $(\{$`at(L1)`$\}, \{$`at(L2)`$\})\})\}),$
`sample(I1, L2)` $= (\{$`at(L2)`$\})\}, \{(\{\}$ $\rightarrow$ $(\{$`have(I1)`$\}, \{\})\})\}),$
`commun(I1)` $= (\{\}, \{(\{$`have(I1)`$\}$ $\rightarrow$ $(\{$`comm(I1)`$\}, \{\})\})\})\}$
$b_I = \{\{$`at(L1)`$,$`have(I1)`$\},\{$`at(L1)`$\}\}$
$G = \{$`comm(I1)`$\}$

Figure 5: Non-Deterministic Planning Problem Example.

deterministic model, the initial state is replaced by an initial belief state b_I. Belief states capture incomplete information by representing 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 5 indicates that there are two states in b_I, denoting that it is unknown if `have(I1)` holds. We also make use a logical representation of belief states, where a state $\hat{s}$ is in a belief state $\hat{b}$ if $\hat{s} \models \hat{b}$. For example, $\hat{b}_I =$ `at(L1)` $\wedge \neg$`at(L2)` $\wedge \neg$`comm(I1)`. As with states, we drop the distinction between the set and logic representation because the context dictates the representation.

In deterministic 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 $appl(a, s)$ to state s if $\rho_e(a) \subseteq s$, and it is applicable $appl(a, b)$ 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 $\varphi_{ij'}(a)$ have consequents such that $\varepsilon^+_{ij}(a) \cap \varepsilon^-_{ij'}(a) = \emptyset$ if there is a state s where both may execute (i.e., $\rho_{ij}(a) \subseteq s$ and $\rho_{ij'}(a) \subseteq s$). In other words, no two 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 presented by Rintanen (2003). For example, the `commun(I1)` action in Figure 5 has a single outcome with a single conditional effect $\{$`have(I1)`$\} \rightarrow (\{$`comm(I1)`$\}, \{\})$. The `commun(I1)` action is applicable to b_I, and its conditional effect occurs only in states where `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'. In non-deterministic planning, executing action a in state s will result in a single state s':

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

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 $s' = exec(\Phi_i(a), s)$, and $T(s, a, s') = 0$, otherwise.

Executing action a in belief state b, denoted $exec(a, b) = b_a$, 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 $\{\{$`at(L1), have(I1), comm(I1)`$\}, \{$`at(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 $\{a_1, ..., a_m\}$ in belief state b_I is defined as $b' = exec(\{a_1, ..., a_m\}, b_I) = exec(a_m, ...exec(a_2, exec(a_1, b_I))...)$. 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 $b' = exec(\{a_1, ..., a_m\}, \{s\})$, 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.

3.2 Planning Graphs

There are many ways to use planning graphs for non-deterministic conformant planning (Bryce et al., 2006; Hoffmann & Brafman, 2004). The most straightforward approach is to ignore state incompleteness. There is nothing about planning graphs that requires defining the first proposition layer as a state; every other proposition layer already represents a set of possibly reachable states. One could union the propositions in all states of the belief state to create an initial proposition layer. Alternatively, one could sample a single state from the belief state to use for the planning graph. Using either of these approaches results in a heuristic that measures the cost to reach the goal from one (or some intersection) of the states in a belief state. This typically leads to an under-estimate because a strong conformant plan must reach the goal from *all* of the states in a belief state.

A more systematic approach to using planning graphs, involves constructing a planning graph for each state in the belief state, extracting a relaxed plan from each planning graph, and aggregating the heuristic values (Bryce et al., 2006). For example, the top portion of Figure 6 shows two planning graphs, each built for a different state in b_I. 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 (deterministic) 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 this problem is addressed with the labeled (uncertainty) planning graph (*LUG*). The *LUG* represents multiple explicit planning graphs implicitly. The planning graph at the bottom of Figure 6 shows the *LUG* representation of the multiple planning graphs at the top. The *LUG* uses labels, 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 (Koehler, Nebel, Hoffmann, & Dimopoulos, 1997) planning graph, in order to explicitly capture conditional effects, and extends it to represent multiple state

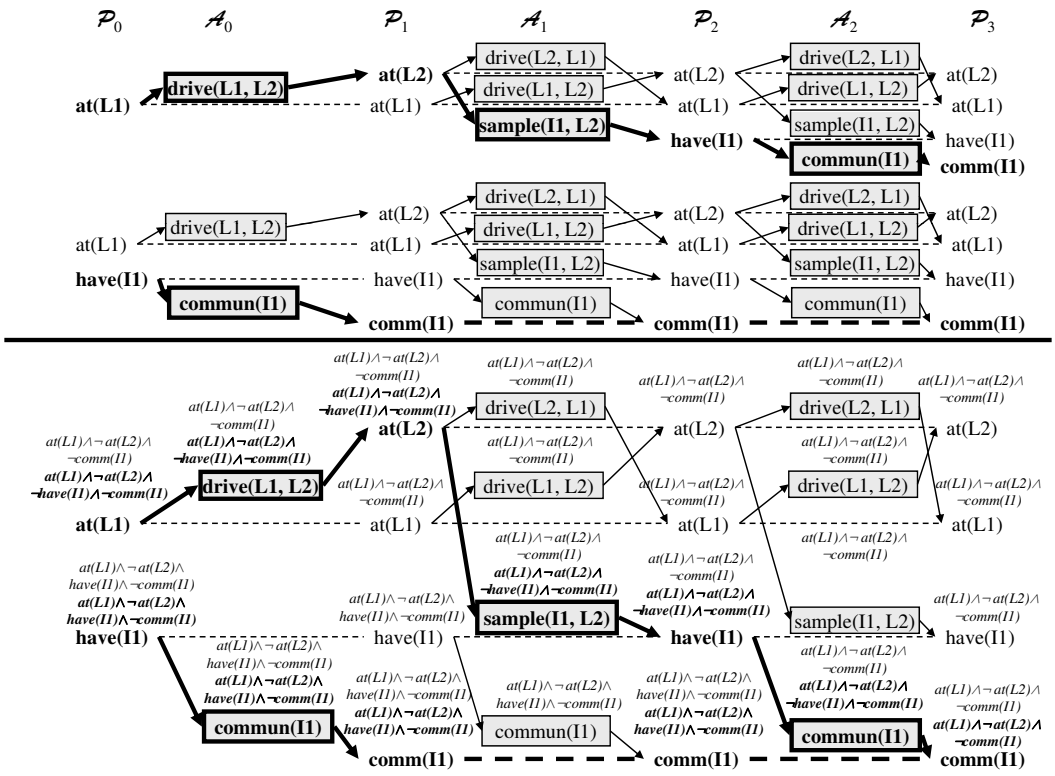

Figure 6: Multiple planning graphs and *LUG*.

causal support (as present in multiple graphs) by adding labels to the vertices of the action $\mathcal{A}$, effect $\mathcal{E}$, and proposition $\mathcal{P}$ layers.[5] 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: $LUG(b) = \langle(\mathcal{P}_0, \mathcal{A}_0, \mathcal{E}_0, ..., \mathcal{A}_{k-1}, \mathcal{E}_{k-1}, \mathcal{P}_k), \ell\rangle$. A label $\ell_t(\cdot)$ denotes a set of states (a subset of the state 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 level the same level. A proposition p is reachable from all states in b after t levels if $\ell_t(p) = b$.

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

1. If $p \in s$, then $s \models \ell_0(p)$ and $p \in \mathcal{P}_0$
2. If $s \models \ell_t(p)$ for every $p \in \rho_e(a)$, then:
   (a) $s \models \ell_t(a)$
   (b) $a \in \mathcal{A}_t$

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

3. If $a \in \mathcal{A}_t$ and $s \models \ell_t(a) \wedge \ell_t(p)$ for every $p \in \rho_{ij}(a)$, then:
   (a) $s \models \ell_t(\varphi_{ij}(a))$
   (b) $\varphi_{ij}(a) \in \mathcal{E}_t$
4. If $p \in \varepsilon^+_{ij}(a)$ and $s \models \ell_t(\varepsilon^+_{ij}(a))$, then $s \models \ell_{t+1}(p)$ and $p \in \mathcal{P}_{t+1}$

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

The initial layer propositions are labeled to denote all states in b where they hold. The actions are labeled to denote all states that reach all of their preconditions. The effects are labeled to denote all states where associated action is reachable and all antecedents are reachable. The propositions are labeled to denote all states that reach an action that gives the proposition as an effect. The goal is reachable by all states in b at the last level k when each state in b is a model of the conjunction of goal proposition labels. The level k is also the level heuristic for b.

For the sake of illustration, Figure 6 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 supported).

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. We can extract a labeled relaxed plan to avoid enumeration by manipulating labels. The labeled relaxed plan $\langle (\mathcal{P}^{RP}_0, \mathcal{A}^{RP}_0, \mathcal{E}^{RP}_0, \mathcal{P}^{RP}_1, ..., \mathcal{A}^{RP}_{k-1}, \mathcal{E}^{RP}_{k-1}, \mathcal{P}^{RP}_k), \ell^{RP} \rangle$ 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. For example, in Figure 6, to support `comm(I1)` in level three we use the labels to determine that persistence can support the goal from state {`at(L1)`, `have(I1)`} in level two and support the goal from state {`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 and each action's label denotes a set of states; the disjunction of chosen action labels denotes a set of states that must contain all states denoted by the supported proposition label.

The procedure for *LUG* relaxed plan extraction is shown in Figure 7. Much like the algorithm for relaxed plan extraction from deterministic planning graphs, *LUG* relaxed plan extraction supports

RPExtract(*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: $\quad \mathcal{P}_k^{RP} \leftarrow \mathcal{P}_k^{RP} \cup p$
4: $\quad \ell_k^{RP}(p) \leftarrow \bigwedge_{p' \in G} \ell_k(p')$
5: **end for**
6: **for** $t = k...1$ **do**
7: $\quad$ **for all** $p \in \mathcal{P}_t^{RP}$ **do** {Support Each Proposition}
8: $\quad\quad \ell \leftarrow \ell_t^{RP}(p)$ {Initialize Possible Worlds to Cover}
9: $\quad\quad$ **while** $\ell \neq \perp$ **do** {Cover Label}
10: $\quad\quad\quad$ 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: $\quad\quad\quad \mathcal{E}_{t-1}^{RP} \leftarrow \mathcal{E}_{t-1}^{RP} \cup \varphi_{ij}(a)$
12: $\quad\quad\quad \ell_t^{RP}(\varphi_{ij}(a)) \leftarrow \ell_t^{RP}(\varphi_{ij}(a)) \vee (\ell_t(\varphi_{ij}(a)) \wedge \ell)$
13: $\quad\quad\quad \mathcal{A}_{t-1}^{RP} \leftarrow \mathcal{A}_{t-1}^{RP} \cup a$
14: $\quad\quad\quad \ell_t^{RP}(a) \leftarrow \ell_t^{RP}(a) \vee (\ell_t(\varphi_{ij}(a)) \wedge \ell)$
15: $\quad\quad\quad \ell \leftarrow \ell \wedge \neg \ell_t(\varphi_{ij}(a))$
16: $\quad\quad$ **end while**
17: $\quad$ **end for**
18: $\quad$ **for all** $a \in \mathcal{A}_{t-1}^{RP}, p \in \rho_e(a)$ **do** {Insert Action Preconditions}
19: $\quad\quad \mathcal{P}_{t-1}^{RP} \leftarrow \mathcal{P}_{t-1}^{RP} \cup p$
20: $\quad\quad \ell_{t-1}^{RP}(p) \leftarrow \ell_{t-1}^{RP}(p) \vee \ell_{t-1}^{RP}(a)$
21: $\quad$ **end for**
22: $\quad$ **for all** $\varphi_{ij}(a) \in \mathcal{E}_{t-1}^{RP}, p \in \rho_{ij}(a)$ **do** {Insert Effect Preconditions}
23: $\quad\quad \mathcal{P}_{t-1}^{RP} \leftarrow \mathcal{P}_{t-1}^{RP} \cup p$
24: $\quad\quad \ell_{t-1}^{RP}(p) \leftarrow \ell_{t-1}^{RP}(p) \vee \ell_{t-1}^{RP}(\varphi_{ij}(a))$
25: $\quad$ **end for**
26: **end for**
27: **return** $\langle(\mathcal{P}_0^{RP}, \mathcal{A}_0^{RP}, \mathcal{E}_0^{RP}, \mathcal{P}_1^{RP}, ..., \mathcal{A}_{k-1}^{RP}, \mathcal{E}_{k-1}^{RP}, \mathcal{P}_k^{RP}), \ell^{RP}\rangle$

Figure 7: Labeled Relaxed Plan Extraction Algorithm.

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). Then for each time step (lines 6-26), the algorithm determines how to support propositions and what propositions must be supported at the preceding time step. Supporting an individual proposition at time t from the states represented by $\ell_t^{RP}(p)$ (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-persistence actions in each action layer of the relaxed plan. The *LUG* in the bottom of Figure 6 depicts the vertices and labels of a labeled relaxed plan in bold. The labeled relaxed plan supports the goal from both states, and the labels indicate which states support the goal using the indicated vertices.

3.3 State Agnostic Planning Graphs

Generalizing the *LUG* to its SAG version, at first glance, raises the worst case complexity by an exponential factor (by representing a set of sets of planning graphs) – there is no complexity-theoretic advantage to the technique. However, we describe an equivalent SAG generalization of the *LUG*, $SLUG$, whose worst-case complexity is identical to the *LUG*–an exponential reduction in the naive SAG generalization's complexity. The intuition is that the *LUG* operates on a set of states, and the naive SAG generalization would operate on a set of sets of states. However, by representing the union of these sets of states, the $SLUG$ manages to reduce the complexity of the representation.

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 4 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 4 is built for the belief state $b = S$ that contains every state in the rover example. The *LUG* is more than a representation of multiple planning graphs: the *LUG* is a full-fledged planning graph for belief-space planning, that can compute the heuristic for a belief state. However, *LUG* heuristics do not amortize graph construction effort across search nodes, even though it uses the deterministic SAG's labeling technique to efficiently build and reason about a set of planning graphs.

We generalize the *LUG* to its naive SAG version by analogy to the state agnostic generalization of the planning graph in deterministic planning. We introduce a label extension to track which sources reach vertices. In this case, sources are belief states. Because the *LUG* already defines its own labels over states, the SAG version of the *LUG* reasons about which states of which belief states reach vertices. In order to complete the generalization, we introduce additional symbols to the label functions to describe belief states (using the original propositional symbols to refer to states). A label represents sets of pairs of belief states and states (b, s) with a boolean formula over the problem propositions P and a new set P' for belief states. We denote to the boolean formula (defined over P and P') of a pair (b, s) by $z(b, a)$. Each pair (b, s) denoted by a label signifies that the planning graph for state s represented within *LUG*(b) reaches the graph vertex.

This state agnostic *LUG* is a labeled graph built for a scope $\mathcal{S}$ containing the set of belief states $B = 2^S$. The following holds for every $b \in B$, and $s \in b$:

1. If $p \in s$, then $z(b, s) \models \ell_0(p)$ and $p \in \mathcal{P}_0$
2. If $z(b, s) \models \ell_t(p)$ for every $p \in \rho_e(a)$, then:
   (a) $z(b, s) \models \ell_t(a)$

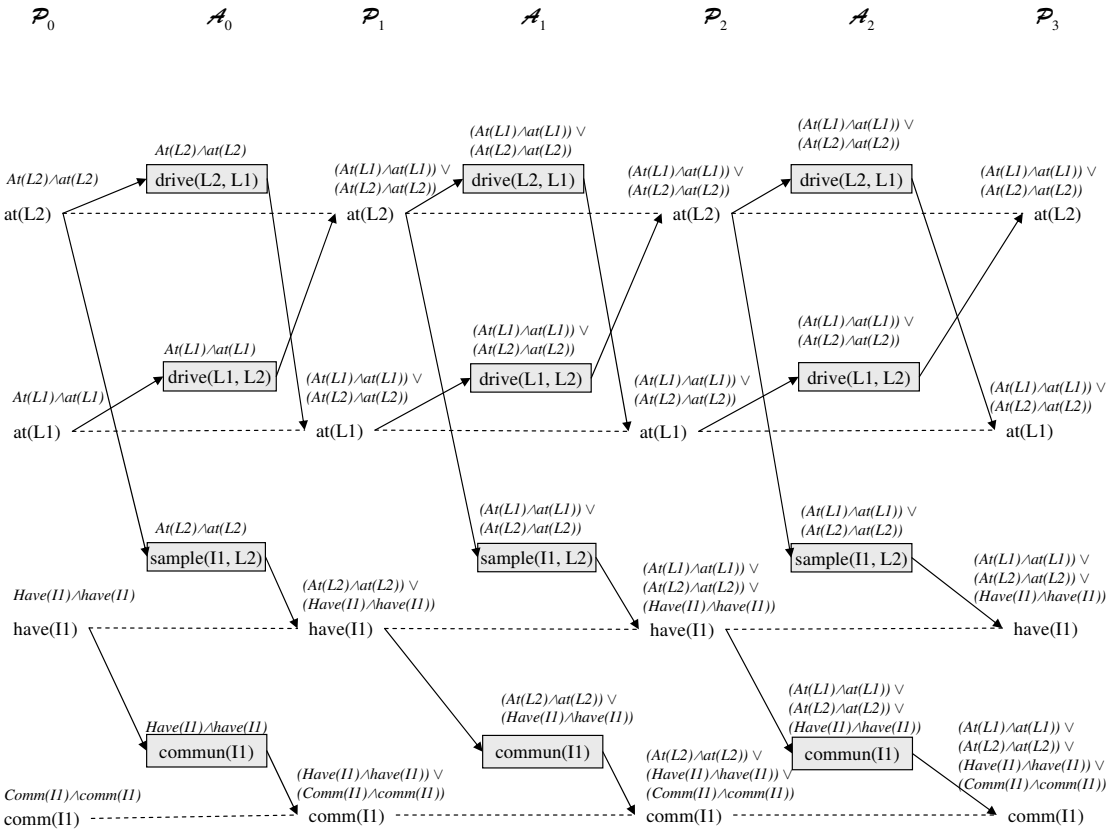

Figure 8: Graph structure of a naive SAG, representing a set of *LUG*.

(b) $a \in \mathcal{A}_t$

3. If $a \in \mathcal{A}_t$ and $z(b,s) \models \ell_t(a) \wedge \ell_t(p)$ for every $p \in \rho_{ij}(a)$, then:
   (a) $z(b,s) \models \ell_t(\varphi_{ij}(a))$
   (b) $\varphi_{ij}(a) \in \mathcal{E}_t$
4. If $p \in \varepsilon^+_{ij}(a)$ and $z(b,s) \models \ell_t(\varphi_{ij}(a))$, then $z(b,s) \models \ell_{t+1}(p)$ and $p \in \mathcal{P}_{t+1}$

Figure 8 depicts the state agnostic generalization of the *LUG* for the example. The labels (in italics) use uppercase propositions to denote elements of P' and lowercase propositions to denote elements of P.

$SLUG$**:** It is possible to optimize the state agnostic *LUG* by eliminating the distinction between belief states in labels. Introducing the new set of propositions is sufficient for representing arbitrary extensions of the planning graph to belief state space. The *LUG*, however, does not require this mechanical scheme. Intuitively, the propagation rules of the *LUG* depend only upon properties

$P = \{$`at(L1), at(L2), have(I1), comm(I1)`$\}$
$A = \{$ `drive(L1, L2)` $= (\{$`at(L1)`$\}, \{(1.0, \{\} \rightarrow (\{$`at(L2)`$\}, \{$`at(L1)`$\})\})\}),$
`drive(L2, L1)` $= (\{$`at(L2)`$\}, \{(1.0, \{\} \rightarrow (\{$`at(L1)`$\}, \{$`at(L2)`$\})\})\}),$
`sample(I1, L2)` $= (\{$`at(L2)`$\})\}, \{(0.9, \{\} \rightarrow (\{$`have(I1)`$\}, \{\})\})\}),$
`commun(I1)` $= (\{\}, \{(0.8, \{\{$`have(I1)`$\} \rightarrow (\{$`comm(I1)`$\}, \{\})\})\})\}$
$b_I = \{(0.9, \{$`at(L1)`$,$`have(I1)`$\}), (0.1, \{$`at(L1)`$\})\}$
$G = \{$`comm(I1)`$\}$

Figure 9: Probabilistic Planning Problem Example.

of world states (as opposed to properties of belief states). An, optimized, State Agnostic Labeled Uncertainty Graph ($SLUG$) exploits this: $SLUG(B)$ represents a set $\{LUG(b)|b{\in}B\}$ for the price of a single element $LUG(b^*), b^* = \bigvee_{b \in B} b$. Figure 4 also depicts the $SLUG$ for every belief state in the rover example because $b^* = \bigvee_{b \in B} b = \bigvee_{b \in B} \bigvee_{s \in b} s = S$ (i.e., the union of all belief states is the set of states).

Any query concerning $LUG(b)$, for any $b \in B$, can be answered using $SLUG(B)$ and the following rule for propositions (similarly for effects and actions): $p \in \mathcal{P}_t(b)$ iff $p \in \mathcal{P}_t(B)$ and $b \models \ell_t(p)$. By analogy with the level heuristic for the SAG, it is possible to compute a level heuristic for the $SLUG$. The level heuristic for belief state space planning, extracted from $SLUG(B)$ for belief state b is defined as minimum level t where $b \models \bigwedge_{p \in G} \ell_t(p)$. Like how the SAG is used in deterministic planning (performing entailment checks for *states* and labels), heuristics use entailment checks with *belief states* and $SLUG$ labels.

The algorithm to extract the relaxed plan for a belief state b from $SLUG(B)$, where $b \in B$, involves a minor change to the labeled relaxed plan extraction algorithm (Figure 7). The change replaces line 4 with:

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

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 the underlying planning model, planning graphs, and state agnostic planning graphs. The main extension in probabilistic planning arises from using actions with stochastic outcomes.

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 stochastic outcomes, b_I is a probability distribution over states, and τ is the minimum probability the plan must satisfy the goal.

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 $s' = exec(\Phi_i(a), s)$, such that:

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

Executing action a in belief state b, denoted $exec(a, b) = b_a$, defines the successor belief state 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 $\{a_1, a_2, ..., a_m\}$ as $b' = exec(\{a_1, a_2, ...a_m\}, b) = exec(a_m, ...exec(a_2, exec(a_1, b))...)$. A plan's probability of satisfying the goal is the probability of the goal in the resulting belief state b' and must exceed τ, such that $\sum_{s \in b': G \subseteq s} b'(s) \geq \tau$. The cost of the plan is equal to the number of actions in the plan.

4.2 Planning Graphs

It is possible to extend the *LUG* to handle probabilities by associating probabilities with each model of each label (Bryce et al., 2008). However, handling uncertain actions, whether non-deterministic or stochastic, is troublesome. With deterministic actions, labels only capture uncertainty about the source belief state 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* because every execution of an action may have a different result. That is, the *LUG* labels capture the joint distribution over random variables $(X_b, X_{a,0}, ..., X_{a',0}, ..., X_{a,k-1}, ..., X_{a',k-1})$, where X_b is distributed over the states in the source belief state b, and $(X_{a,t}, ..., X_{a',t})$ are distributed over the corresponding action outcomes in action layer t. An assignment of values to the random variables corresponds to a single deterministic planning graph, built for the given state and set of action outcomes. 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. 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* ($\mathcal{M}c$*LUG*) represents a set of planning graph samples using the labeling technique developed in the *LUG*. The $\mathcal{M}c$*LUG* is a set of vertices and a label function: M$cLUG(b) = \langle(\mathcal{P}_0, \mathcal{A}_0, \mathcal{E}_0, ..., \mathcal{A}_{k-1}, \mathcal{E}_{k-1}, \mathcal{P}_k), \ell\rangle$. The $\mathcal{M}c$*LUG* represents a set of N particles (joint sets of sampled values). Each particle x^n, $n = 0, ..., N-1$, is a set of pairs of random variables and their sampled values $\{(X_b, s), (X_{a,0}, \Phi_i(a)), ..., (X_{a',0}, \Phi_i(a'))$, ..., $(X_{a,k-1}, \Phi_i(a)), ...,$ $(X_{a',k-1}, \Phi_i(a'))\}$. In the following, we denote the n^{th} sampled value v from the probability distribution over random variable X by $P(X) \overset{n}{\sim} v$. To facilitate labels, each particle is also associated with a boolean formula $y(x^n)$ (a model), defined over the special propositions $(y_0, ..., y_{log_2(N)-1})$. For example, when $N = 4$ as in Figure 10, $y(x^0) = \neg y_0 \wedge \neg y_1$ and $y(x^3) = y_0 \wedge y_1$.

For each particle $x^n, n = 0...N-1$, the $\mathcal{M}c$*LUG* satisfies:

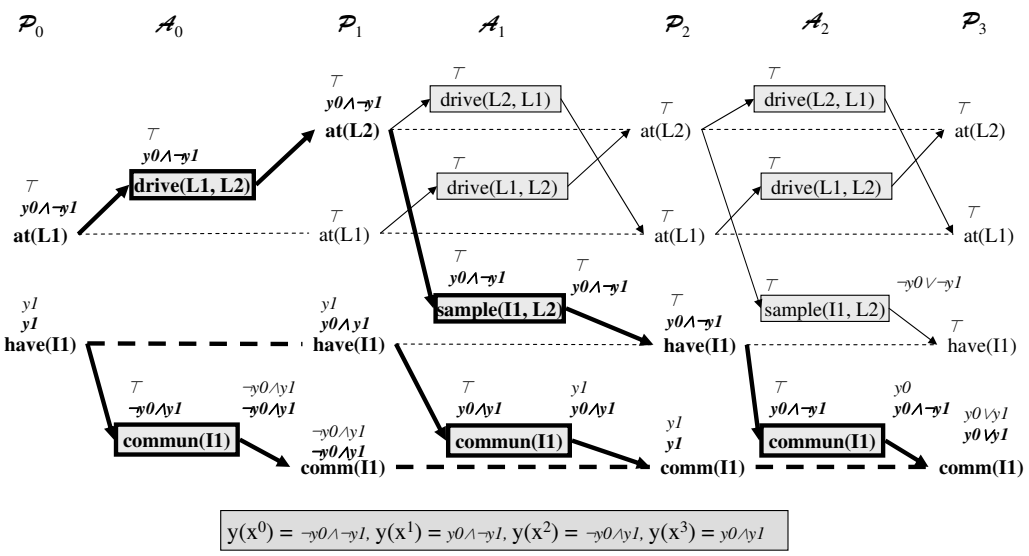

Figure 10: Monte Carlo labeled uncertainty graph.

1. If $P(X_b) \overset{n}{\sim} s$, and $p \in s$, then $(X_b, s) \in x^n$, $y(x^n) \models \ell_0(p)$, and $p \in \mathcal{P}_0$
2. If $y(x^n) \models \ell_t(p)$ for every $p \in \rho_e(a)$, then:
   (a) $y(x^n) \models \ell_t(a)$
   (b) $a \in \mathcal{A}_t$
3. If $a \in \mathcal{A}_t$, $y(x^n) \models \ell_t(a) \wedge \ell_t(p)$ for every $p \in \rho_{ij}(a)$, and $P(X_{a,t}) \overset{n}{\sim} \Phi_i(a)$ then:
   (a) $(X_{a,t}, \Phi_i(a)) \in x^n$
   (b) $y(x^n) \models \ell_t(\varphi_{ij}(a))$
   (c) $\varphi_{ij}(a) \in \mathcal{E}_t$
4. If $p \in \varepsilon^+_{ij}(a)$ and $y(x^n) \models \ell_t(\varphi_{ij}(a))$, then $y(x^n) \models \ell_{t+1}(p)$ and $p \in \mathcal{P}_{t+1}$

The following label rules can be used to construct the $\mathcal{M}cLUG$:

1. If $P(X_b) \overset{n}{\sim} s$, then $(X_b, s) \in x^n, n = 0, ..., N-1$
2. $\ell_0(p) = \bigvee_{x^n : p \in s, (X_b, s) \in x^n} y(x^n)$
3. $\ell_t(a) = \bigwedge_{p \in \rho_e(a)} \ell_t(p)$
4. If $P(X_{a,t}) \overset{n}{\sim} \Phi_i(a)$, then $(X_{a,t}, \Phi_i(a)) \in x^n, n = 0, ..., N-1$

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. $k =$ minimum t such that $\frac{\left| \left\{ x^n \middle| y(x_t^n) \models \left(\bigwedge_{p \in G} \ell_t(p) \right) \right\} \right|}{N} \geq \tau$

The $\mathcal{M}cLUG$ label construction starts by sampling N states from the belief state b and associating each state with a particle. Figure 10 depicts a $\mathcal{M}cLUG$ for the initial belief state of the example in Figure 10. Two particles sample state {`at(L1)`} and two sample {`at(L1)`, `have(I1)`}. The label of each initial proposition is the disjunction of the labels of the particles that sample states in which the proposition holds. Each action label is the conjunction of its precondition proposition labels (as in the *LUG*). The label of each effect is the conjunction of the associated action label, the conjunction of the secondary precondition labels, and the conjunction of the disjunction of the labels of particles that sample the outcome containing the effect. The last level of the $\mathcal{M}cLUG$ (and also the level heuristic) is defined by the first level where the proportion of particles reaching the goal propositions is no less than τ. A particle reaches the goal if its label is a model of the conjunction of goal proposition labels. In Figure 10 there are $N = 4$ particles. One particle reaches the goal `comm(I1)` at level one; two particles, at level two; and three particles, 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 particles.

Labeled relaxed plan extraction in the $\mathcal{M}cLUG$ is identical to the *LUG*, as described in the previous section. However, the interpretation of procedure's semantics does change slightly. We pass a $\mathcal{M}cLUG$ for a given belief state b to the procedure, instead of a *LUG*. The labels for goal propositions (line 4) represent particles, and via the $\mathcal{M}cLUG$ termination criterion, we do not require labels to contain all particles – only a proportion no less than τ.

In the example, the goal `comm(I1)` is labeled with three particles $\ell_3(\texttt{comm(I1)}) = y_0 \vee y_1 = y(x^1) \vee y(x^2) \vee y(x^3)$. Particles x^2, x^3 are supported by the persistence effect $\varphi_{00}(\texttt{comm(I1)}_p)$, and particle x^1 is supported by $\varphi_{00}(\texttt{commun(I1)})$, 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 execution precondition. The relaxed plan contains three invocations of `commun(I1)` (reflecting how action repetition may be needed when actions have uncertain outcomes), and the `drive(L1, L2)` and `sample(L1, L2)` actions. The value of the relaxed plan is five because it uses five non-persistence actions.

4.3 State Agnostic Planning Graphs

The SAG for non-deterministic and deterministic planning captures a single planning graph for each state because the actions are deterministic. In probabilistic planning, the $\mathcal{M}cLUG$ deals with actions with uncertain outcomes, meaning there is potentially several planning graphs per state. The $\mathcal{M}cLUG$ shows how it is possible to compute a heuristic for a belief state by first sampling a set of states from the belief state, and for each state sampling from its set of potential planning graphs.

Developing a state agnostic generalization of the $\mathcal{M}cLUG$ requires a data structure that can compute the heuristic for all probabilistic belief states (of which there an infinite number). A com-

plication is that the $\mathcal{M}cLUG$ generates new and potentially different planning graphs for each state sampled from each belief state. As a first step toward a state agnostic $\mathcal{M}cLUG$, it seems reasonable to sample a fixed pool of planning graphs for each state and re-sample from this pool during search. It is possible to ensure that the pool of planning graphs can compute the heuristic for any set of state samples by generating N planning graphs for each state (where N is the maximum number of samples per belief state). This means that, using ideas from the $SLUG$ (where labels describe sets of states, instead of pairs of belief states and states), the state agnostic $\mathcal{M}cLUG$ captures $O(N2^{|P|})$ planning graphs.

Consider the number of random numbers, used in this scheme, to construct a state agnostic $\mathcal{M}cLUG$. Each of the $N2^{|P|}$ planning graphs represented by the state agnostic $\mathcal{M}cLUG$ will have at most k levels (determined below), with $O(A)$ actions per level. If each action has an uncertain outcome, there are on the order of $O(N2^{|P|}|A|^k)$ samples needed to construct the state agnostic $\mathcal{M}cLUG$. Under this scheme, each label defines a set of pairs $(s, x^n(s))$, defined as a boolean function over state propositions and the particle propositions. Because the particle propositions are state dependent, there is a set of propositions $y_0(s), ..., y_{log_2(N)-1}(s)$ for each state s. Each state dependent particle $x^n(s)$ samples the value v of each random variable X, denoted $P(X) \overset{n,s}{\sim} v$.

The state agnostic version of the $\mathcal{M}cLUG$ for all $s \in \mathcal{S}$ and all $n = 0...N-1$ satisfies:

1. If $p \in s$, then $s \wedge y(x^n(s)) \models \ell_0(p)$ and $p \in \mathcal{P}_0$
2. If $s \wedge y(x^n(s)) \models \ell_t(p)$ for every $p \in \rho_e(a)$, then:
   (a) $s \wedge y(x^n(s)) \models \ell_t(a)$
   (b) $a \in \mathcal{A}_t$
3. If $a \in \mathcal{A}_t$, $s \wedge y(x^n(s)) \models \ell_t(a) \wedge \ell_t(p)$ for every $p \in \rho_{ij}(a)$, and $P(X_{a,t}) \overset{n,s}{\sim} \Phi_i(a)$ then:
   (a) $(X_{a,t}, \Phi_i(a)) \in x^n(s)$
   (b) $s \wedge y(x^n(s)) \models \ell_t(\varphi_{ij}(a))$
   (c) $\varphi_{ij}(a) \in \mathcal{E}_t$
4. If $p \in \varepsilon^+_{ij}(a)$ and $s \wedge y(x^n(s)) \models \ell_t(\varphi_{ij}(a))$, then $s \wedge y(x^n(s)) \models \ell_{t+1}(p)$ and $p \in \mathcal{P}_{t+1}$

Since each label denotes a set of pairs $(s, x^n(s))$, the label representation is dependent both on the number of states and the number of samples per state. Using the notion of "common samples", it is possible to reduce the size of the label representation and the cost of computing the SAG generalization of the $\mathcal{M}cLUG$.

$\mathcal{C}SSAG$: We consider a planning graph called the common sample SAG ($\mathcal{C}SSAG$) to share action outcome samples between states. The $\mathcal{C}SSAG$ is a set of vertices and a label function, defined: $\mathcal{C}SSAG(\mathcal{S}, N) = \langle(\mathcal{P}_0, \mathcal{A}_0, \mathcal{E}_0, ..., \mathcal{A}_{k-1}, \mathcal{E}_{k-1}, \mathcal{P}_k), \ell\rangle$. In the $\mathcal{C}SSAG$, labels define sets of pairs (s, x^n), $s \models \mathcal{S}, n = 0...N-1$, where x^n refers to a joint set of action outcome samples that is independent of the starting state. A label denoting the pairs (s, x^n) and (s', x^n), signifies that whether starting in state s or s' and fixing the same outcomes to uncertain actions in the planning graph, the labeled vertex is reachable. This reduces the number of samples needed from $O(N2^{|P|}|A|^k)$ to $O(N|A|^k)$. The complexity of the $\mathcal{C}SSAG$ is the same order of magnitude as the $\mathcal{M}cLUG$ but

the actual complexity is often higher in the state agnostic version because there are usually more actions per level.

Notice the effect that using common samples has upon the number of propositions needed to represent particles. Without common samples, the SAG version of the $\mathcal{M}cLUG$ uses state specific propositions $y_0(s), ..., y_{log_2(N)-1}(s)$ for each state s, and with common samples we use propositions $y_0, ..., y_{log_2(N)-1}$ that are independent of the state (an exponential reduction). Furthermore, compared to the LUG, the $\mathcal{C}SSAG$ only increases the number of label propositions by a factor logarithmic in N (where the naive SAG generalization of the $\mathcal{M}cLUG$ increases the number of label propositions by a factor exponential in P, and logarithmic in N).

The $\mathcal{C}SSAG$ for all $s \in \mathcal{S}$ and $n = 0...N-1$ satisfies:

1. If $p \in s$, then $s \wedge y(x^n) \models \ell_0(p)$ and $p \in \mathcal{P}_0$
2. If $s \wedge y(x^n) \models \ell_t(p)$ for every $p \in \rho_e(a)$, then:
   (a) $s \wedge y(x^n) \models \ell_t(a)$
   (b) $a \in \mathcal{A}_t$
3. If $a \in \mathcal{A}_t$, $s \wedge y(x^n) \models \ell_t(a) \wedge \ell_t(p)$ for every $p \in \rho_{ij}(a)$, and $P(X_{a,t}) \overset{n}{\sim} \Phi_i(a)$ then:
   (a) $(X_{a,t}, \Phi_i(a)) \in x^n$
   (b) $s \wedge y(x^n) \models \ell_t(\varphi_{ij}(a))$
   (c) $\varphi_{ij}(a) \in \mathcal{E}_t$
4. If $p \in \varepsilon^+_{ij}(a)$ and $s \wedge y(x^n) \models \ell_t(\varphi_{ij}(a))$, then $s \wedge y(x^n) \models \ell_{t+1}(p)$ and $p \in \mathcal{P}_{t+1}$

The following rules can be used to construct the $\mathcal{C}SSAG$:

1. $\ell_0(p) = \mathcal{S} \wedge p$
2. $\ell_t(a) = \bigwedge_{p \in \rho_e(a)} \ell_t(p)$
3. If $P(X_{a,t}) \overset{n}{\sim} \Phi_i(a)$, then $(X_{a,t}, \Phi_i(a)) \in x^n, n = 0, ..., N-1$
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. $k =$ minimum t such that $\mathcal{P}_t = \mathcal{P}_{t+1}$, $\ell_t(p) = \ell_{t+1}(p), p \in P$, and $\frac{\left| \left\{ x^n \middle| y(x^n) \models \left(\bigwedge_{p \in G} \ell_t(p) \right) \right\} \right|}{N} = 1.0$

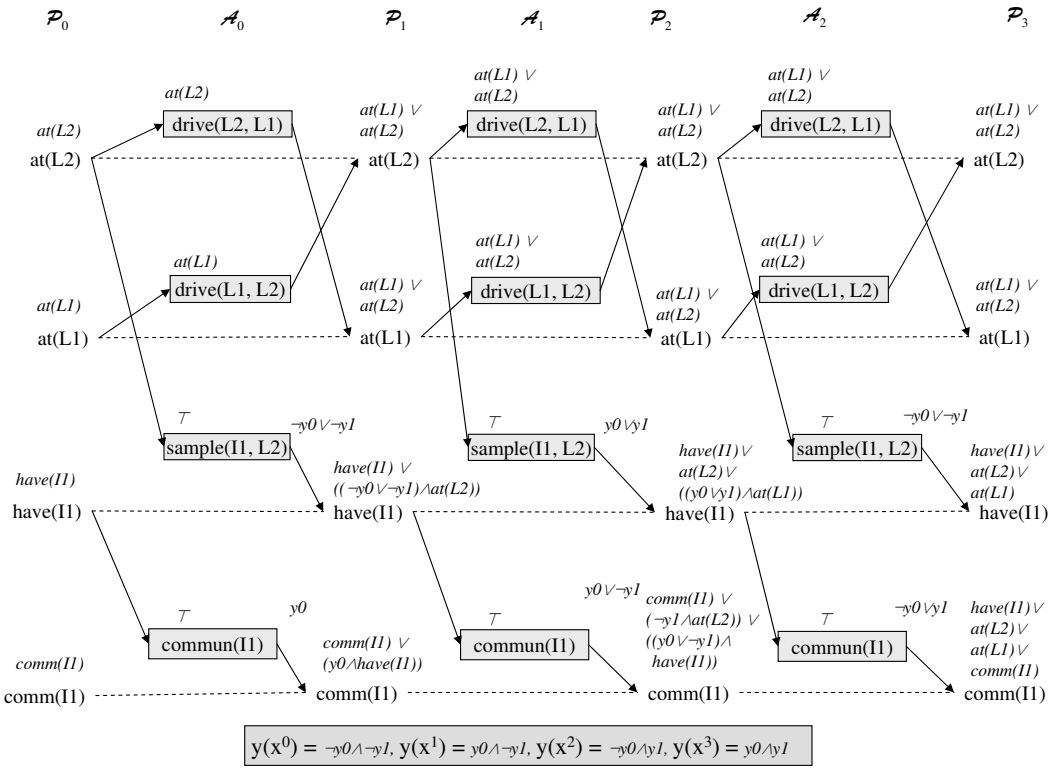

Figure 11: Graph structure of a $\mathcal{CSSAG}$.

The initial layer propositions are labeled to denote the states in the scope where they hold.[6] Action labels denote the states and particles where their preconditions are all reachable. Each particle samples each action outcome at each level. Each effect label denotes the particles that sample an outcome with the effect and where the effect is reachable (i.e., its action and secondary preconditions are reachable). Each proposition label denotes the states and particles where it is given by some effect. The last level k is defined by the level where proposition layers and labels are identical and all particles satisfy the goal.

Using the $\mathcal{CSSAG}$ to evaluate a heuristic for a belief state b involves sampling a set of N states from the belief state to identify a label formula, denoted $\ell(b)$, where

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

The level heuristic is the first level t where the proportion of state and particle pairs (sampled from the belief state) that reach the goal exceeds the goal satisfaction threshold,

$$\frac{|\{x^n | y(x^n) \models \ell(b) \wedge \bigwedge_{p \in G} \ell_k(p)\}|}{N} \geq \tau$$

6. Because labels are defined over state propositions and particle label propositions and particles are independent of the state, proposition labels denote the proposition is reachable for all N particles (i.e., $(\mathcal{S} \wedge p) \models \bigvee_{n=1 \ldots N-1} y(x^n) = \top$).

For example, the level heuristic for the initial belief state $b_I = \{(0.9, \{\texttt{at(L1)}, \texttt{have(I1)}\}), (0.1, \{\texttt{at(L1)}\})\}$ is computed as follows. If $N = 4$, we may draw the following sequence of state samples from the initial belief state $(\{\texttt{at(L1)}\}, \{\texttt{at(L1)}, \texttt{have(I1)}\}, \{\texttt{at(L1)}, \texttt{have(I1)}\}, \{\texttt{at(L1)}, \texttt{have(I1)}\})$ to define

$$\begin{aligned}\ell(b_I) \;=\; & (\texttt{at(L1)} \wedge \neg\texttt{at(L2)} \wedge \neg\texttt{have(I1)} \wedge \neg\texttt{comm(I1)} \wedge \neg y0 \wedge \neg y1) \vee \\ & (\texttt{at(L1)} \wedge \neg\texttt{at(L2)} \wedge \texttt{have(I1)} \wedge \neg\texttt{comm(I1)} \wedge (y0 \vee y1))\end{aligned}$$

The conjunction of the label of the goal proposition $\texttt{comm(I1)}$ and $\ell(b_I)$ at level zero is

$$\ell_0(\text{comm(I1)}) \wedge \ell(b_I) = \texttt{comm(I1)} \wedge \ell(b_I) = \bot$$

meaning zero particles reach the goal. At level one the conjunction of the goal label with $\ell(b_I)$ is

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

meaning that two particles reach the goal because both $y(x^1) = y_0 \wedge \neg y_1$ and $y(x^3) = y_0 \wedge y_1$ are entailed by the conjunction above. At level two, the conjunction of the goal label with $\ell(b_I)$ is

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

meaning that the same two particles reach the goal at level two as in level one. At level three, the conjunction of the goal label with $\ell(b_I)$ is

$$\ell_2(\texttt{comm(I1)}) \wedge \ell(b_I) \;=\; (\texttt{have(I1)} \vee \texttt{at(L2)} \vee \texttt{at(L1)} \vee \texttt{comm(I1)}) \wedge \ell(b_I)$$

meaning that all particles reach the goal because each particle label entails 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. Without expanding the $\mathcal{CSSAG}$ further, it is possible to say the level heuristic is at least three, when $\tau = 1.0$.

Relaxed plan extraction for the common sample SAG is identical to relaxed plan extraction for the $SLUG$, with the exception that the intersection in line 4 of Figure 7 replaces the belief state b with the formula representing the set of state-particle sample pairs, such that line 4 becomes:

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

adding the underlined portion. The addition to relaxed plan extraction enforces that the relaxed plan supports the goal in only those state and particle pairs sampled to compute the heuristic.

5. Advanced SAG Techniques

This section covers two extensions to the SAG techniques described in the previous section. The first extension carries the analogy between planning graphs and state agnostic planning graphs through to relaxed plans to extract a state agnostic relaxed plan, which captures all relaxed plans. The second extension more closely examines the choice of scope used to construct the SAG, defining a continuum between planning graphs and state agnostic planning graphs.

5.1 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, extracts the relaxed plan for each state

(or state and sample pair) before search, and during search combines the necessary relaxed plans to evaluate specific search nodes.

Much like how the SAG can be used to represent a set of planning graphs with labels indicating which states are relevant to which planning graph vertices, it is possible to do the same for relaxed plans. The idea is to extract a labeled relaxed plan to support the goals from the entire scope of the SAG. Each action in this state agnostic relaxed plan is labeled to denote the states that use it to support the goal. Then, to evaluate the heuristic for a specific search node, we count the number of actions in the state agnostic relaxed plan that are used for the states denoted by the search node.

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 states. In the traditional relaxed plan, this set of states contains exactly those states that are needed to evaluate a search node. In the state agnostic relaxed plan, the set of states (e.g., any subset of S) 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, the 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).

5.2 Choosing the SAG Scope

The choice of scope $\mathcal{S}$ has a great impact on the performance of any approach based on the SAG. Using fewer states in the scope almost always requires less computation per graph. While not every label function becomes smaller, the aggregate size almost always decreases. Restricting the scope, however, prevents the SAG from representing the planning graphs for states so excluded. If such states are visited in search, then a new SAG will need to be generated to cover them. All of the approaches based on SAG can be seen as a particular strategy for covering the set of all states with shared graphs. We define four points in that spectrum: Global-SAG, Reachable-SAG, Child-SAG, and the PG (Node-SAG).

Global-SAG: A Global-SAG is a single graph for an entire planning episode. The scope is taken to be the set of all states (or state and sample pairs); that is, Global-SAG uses the degenerate partition containing only the set of all states (or state and sample pairs).

Reachable-SAG: A Reachable-SAG is also a single graph for an entire planning episode. A normal planning graph is constructed from the initial belief state of the problem; all states consistent with the last level form the set from which the Reachable-SAG is built. That is, states are partitioned into two groups: definitely unreachable, and possibly reachable. A graph is generated only for the latter set of states.

Child-SAG: A Child-SAG is a graph built for the set of children of the current node. This results in one graph per non-leaf search node. That is, Child-SAG partitions states into sets of siblings. While this is still an exponential number of graphs, the reduction in computation time, relative to Node-SAG, is still significant.

Node-SAG: A Node-SAG is built for the smallest conceivable scope: the current search node. In some situations, label propagation is useless, and therefore skipped. That is, Node-SAG is just a name for the traditional approach of building a planning graph for every search node.

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?
- What choices for SAG scope are most appropriate?
- Will using state agnostic relaxed plans improve planner performance over using 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 (domains, planners, and environments) used for evaluation. The last two subsections discuss the first three 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 domains, planners, and environments that we use to evaluate our approach.

Domains: We use deterministic, non-deterministic, and probabilistic planning domains that appear either in the literature or past International Planning Competitions (IPC). In deterministic planning, we use domains from the first three IPCs, including logistics, rovers, blocksworld, zenotravel, driverlog, towers of hanoi, and satellite. In non-deterministic planning, we use rovers, logistics, ring, and cube from (Cushing & Bryce, 2005), and blocksworld, coins, communication, universal traversal sequences, adder circuits, and sorting networks from the Fifth IPC conformant planning track. In probabilistic planning we use the logistics, grid, slippery gripper, and sand castle (Bryce et al., 2008) domains. Each of the domains contains several problem instances.

Planners: $POND$ is implemented in C++ and makes use of some notable existing technologies: the CUDD BDD package (Somenzi, 1998) and the IPP planning graph (Koehler et al., 1997). $POND$ 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 (Hoffmann & Nebel, 2001) and A* search (with a heuristic weight of five). Briefly, enforced hill-climbing interleaves local search with systematic search by locally committing to action that lead to

search nodes with a decreased heuristic value and using breadth first search if it cannot immediately find a better search node.

We compare our planner $POND$ with several other planners. In non-deterministic planning, we compare with Conformant FF (CFF) (Hoffmann & Brafman, 2004) and t0 (Palacios & Geffner, 2006) in the IPC results, and CFF, KACMBP (Bertoli & Cimatti, 2002), BBSP (Rintanen, 2005), and MBP (Bertoli, Cimatti, Roveri, & Traverso, 2001) in additional non-deterministic planning results. CFF is an extension of the FF planner 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 deterministic planning, where it uses the FF planner (Hoffmann & Nebel, 2001). KACMBP uses a combination of cardinality-based and reachability based heuristics to measure both the size of belief states and number of actions needed to achieve goals. BBSP uses regression search, guided by a belief state cardinality heuristic. MBP uses a forward chaining search with a belief state cardinality heuristic.

In probabilistic planning, we compare with Probabilistic FF (PFF) (Domshlak & Hoffmann, 2006) and CPplan (Hyafil & Bacchus, 2003). 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.

Environments: In all models, the measurement for each problem is the total run time of the planner, 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 deterministic 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 twenty minute time limit. There are several hundred problems in the deterministic planning IPC test sets, so we imposed relatively tight limits on the execution (five 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 the results 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.

6.2 Internal Evaluation

We describe several sets of results that address which scope is appropriate, whether using the SAG is reasonable, and how computing different types of relaxed plans effects planner performance. The

results include deterministic, non-deterministic, and probabilistic planning problems. Due to the high number of combinations between the models and issues under evaluation, we describe the results in the following manner. We evaluate every scope in deterministic and non-deterministic models and present results for two scopes in each model: Node-SAG and Reachable-SAG. With respect to these two scopes, 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 *a priori*) affects performance.

Scope: We do not present exhaustive results for the range of choices of SAG scope because of the following observations:

- The Global-SAG is always dominated by Reachable-SAG because it wastes a significant amount of time projecting reachability for unreachable states. While the Reachable-SAG's dominance may seem obvious, the phase-transition inherent to BDDs (which we use to represent labels) is a factor that is not well understood without experimentation. That is, the Global-SAG uses labels (boolean functions) that represent more states (assignments to variables), and the Reachable-SAG generally represents much fewer. It is well known that the BDD representation of boolean functions is typically simple when there are very many or very few satisfying assignments to the function variables. It appears that there is not enough simplification in the boolean functions to make the Global-SAG viable. The Reachable-SAG performs well despite having potentially less boolean function simplification because it is more likely that fewer unreachable actions and propositions enter the planning graph.

- The Child-SAG improves upon the Node-SAG in virtually all problems. However, that margin is relatively small. The Child-SAG typically improves upon the Reachable-SAG in deterministic planning problems in cases where the number of evaluated states is very small compared to the number of reachable states. In almost all non-deterministic planning problems, the Reachable-SAG outperforms the Child-SAG because the same states often appear in a search node's descendants. If the same state appears within different belief states multiple times, then the Reachable-SAG will compute the planning graph for this state once, but the Child-SAG may compute the planning graph multiple times.

In the remainder of the empirical evaluation, we limit discussion to the Reachable-SAG and the Node-SAG (also referred to as a planning graph).

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 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 Reachable-SAG with the planning graph (PG) in deterministic planning. The top graph is a scatter-plot of the total running times. 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 Reachable-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.

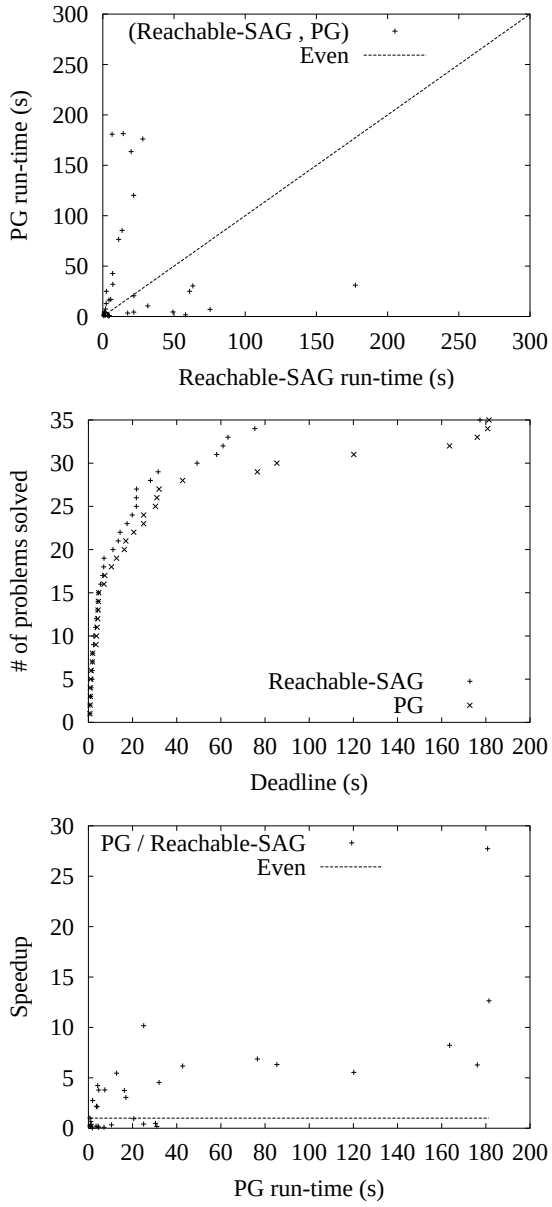

Figure 12: Reachable-SAG vs. PG, Deterministic Problems

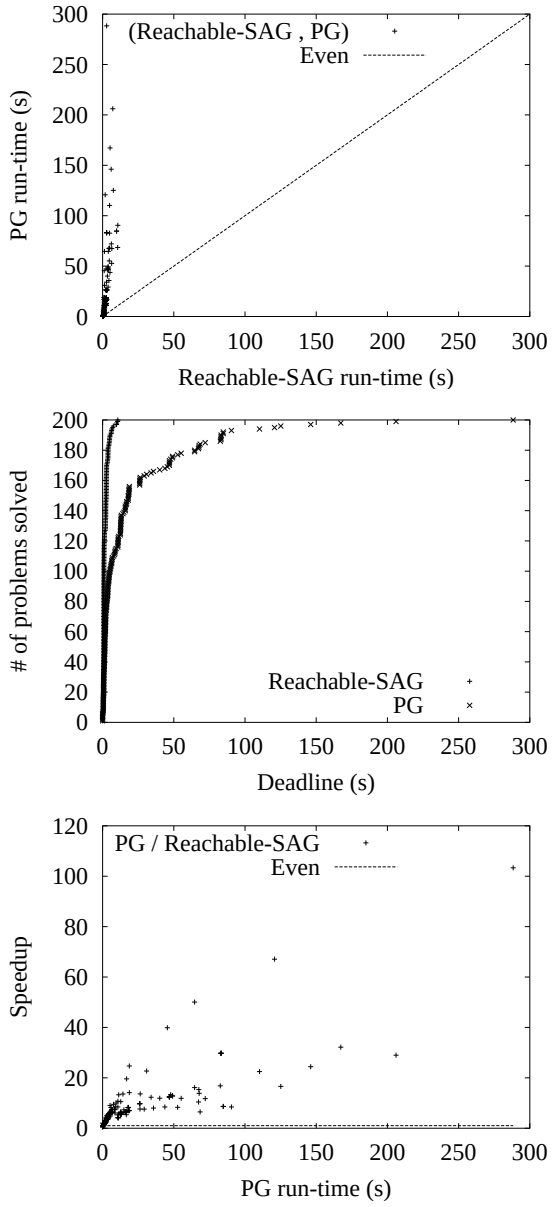

Figure 13: *SLUG* vs. *LUG*, Non-Deterministic Problems

In light of this, we have made a investigation in comparing SAG to state of the art implementations of deterministic planning graphs. In particular, Hoffmann and Nebel (2001) go to great lengths to build deterministic 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 Reachable-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 Reachable-SAG.

We see that in non-deterministic planning problems, depicted in Figure 13 in the same format as the deterministic planning problems, the scatter-plots reveal that Reachable-SAG always outperforms 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 complexity: rotating the axes reveals the telltale exponential trend. However, it is difficult to measure complexity 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, Reachable-SAG solves a much greater number of planning problems. Most importantly, Reachable-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.

Figures 14 to 20 show total time in seconds and plan length results for the probabilistic planning problems. The figures compare the relaxed plan extracted from the $\mathcal{CSSAG}$ (denoted NRP), the state agnostic relaxed plan (denoted $SAGRP$), the $\mathcal{McLUG}$ (denoted by the number of particles), PFF, 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 16 or 64 particles per $\mathcal{McLUG}$ or $\mathcal{CSSAG}$ as indicated in the legend of the figures for each domain, because these numbers proved best in our evaluation.

Figure 14 shows that the $\mathcal{CSSAG}$ improves, if only slightly, upon the $\mathcal{McLUG}$ in Logistics p2-2-2. Figure 15 shows similar results for Logistics p4-2-2, with the $\mathcal{CSSAG}$ performing considerably better than the $\mathcal{McLUG}$ – solving the problem where $\tau = 0.95$. Figure 16 shows results for Logistics p2-2-4 that demonstrate the improvements of using the $\mathcal{CSSAG}$, finding plans for $\tau = 0.95$, where the $\mathcal{McLUG}$ could only solve instances where $\tau \leq 0.25$. Overall, using the $\mathcal{CSSAG}$ is better than the $\mathcal{McLUG}$.

Figure 17 shows results for the Grid-0.8 domain that indicate the $\mathcal{CSSAG}$ greatly improves total planning time with the $\mathcal{CSSAG}$ over the $\mathcal{McLUG}$. However, Figure 18 shows the results are different for Grid-0.5. The $\mathcal{CSSAG}$ performs much worse than the $\mathcal{McLUG}$. A potential explanation is the way in which the $\mathcal{CSSAG}$ chooses a pool of common action outcome samples to use in the planning graphs. The $\mathcal{McLUG}$ is more robust to the sampling because it re-samples the action outcomes for each belief state, where the $\mathcal{CSSAG}$ re-samples the action outcomes from the pre-sampled pool.

Figures 19 and 20 show results for the respective SandCastle-67 and Slippery Gripper domains, where the $\mathcal{McLUG}$ outperforms the $\mathcal{CSSAG}$. Similar to the Grid-0.5 domain, $\mathcal{CSSAG}$ has an impoverished pool of action outcome samples that does not plague the $\mathcal{McLUG}$. Since these problems are relatively small, the cost of computing the $\mathcal{McLUG}$ pales in comparison to the quality of the heuristic it provides. Overall, the $\mathcal{CSSAG}$ is useful when it is too costly to compute a $\mathcal{McLUG}$ 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. The heuristic that we use to extract state agnostic relaxed plans is typically very poor because it selects actions that help achieve the goal from more states. 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

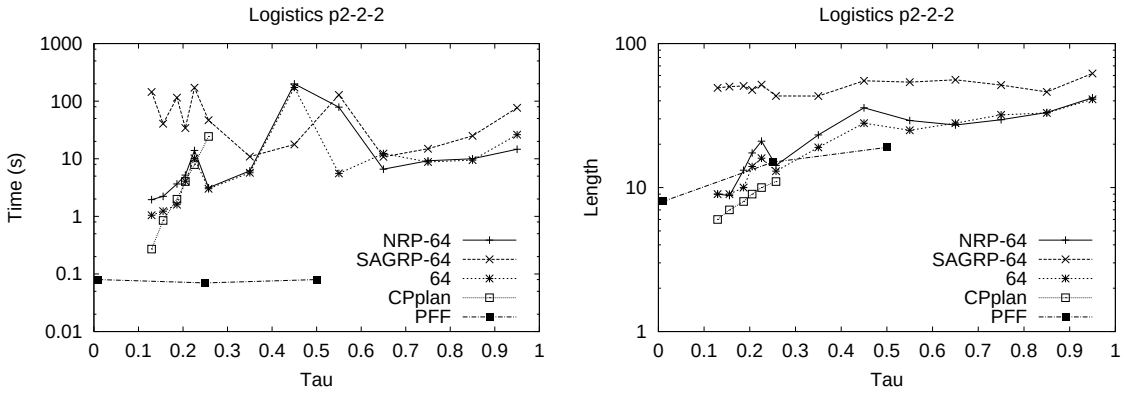

Figure 14: Run times (s) and Plan lengths vs. τ for Logistics p2-2-2

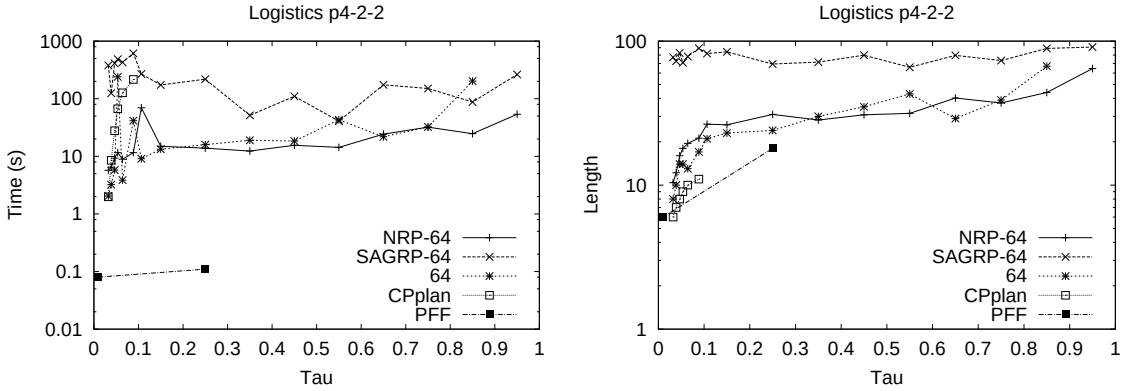

Figure 15: Run times (s) and Plan lengths vs. τ for Logistics p4-2-2

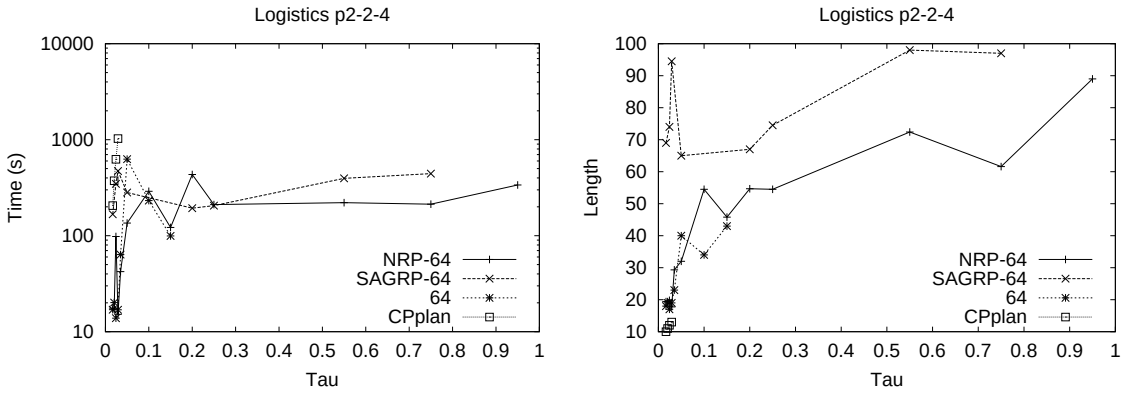

Figure 16: Run times (s) and Plan lengths vs. τ for Logistics p2-2-4

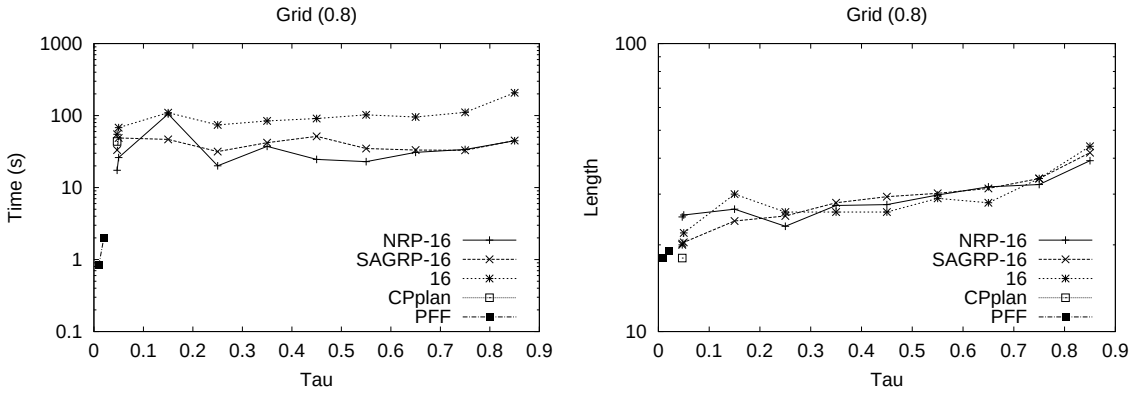

Figure 17: Run times (s) and Plan lengths vs. τ for Grid-0.8

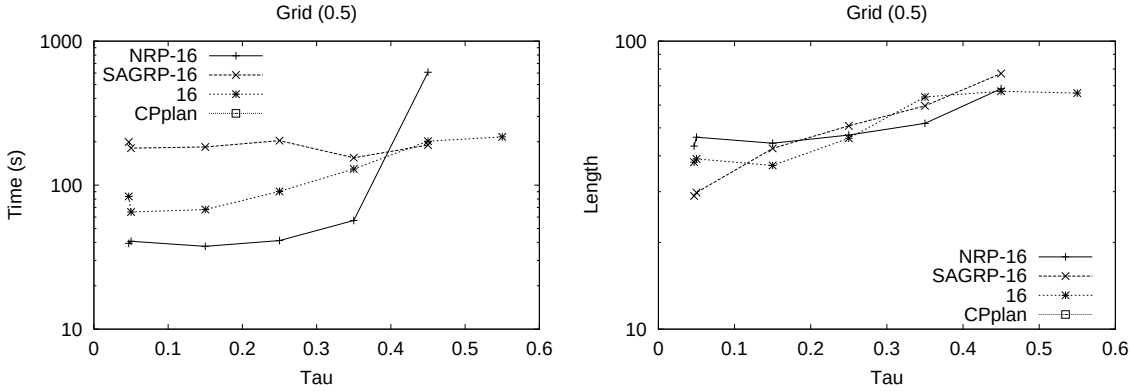

Figure 18: Run times (s) and Plan lengths vs. τ for Grid-0.5

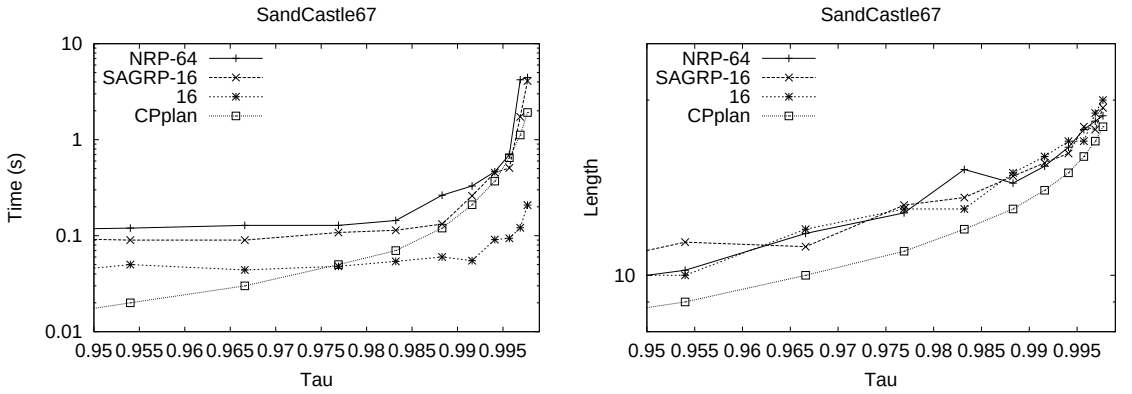

Figure 19: Run times (s) and Plan lengths vs. τ for SandCastle-67.

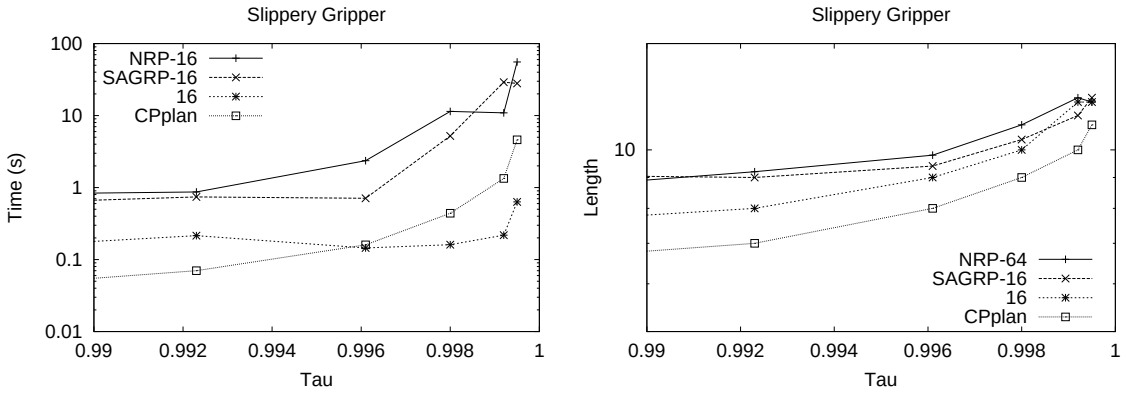

Figure 20: Run times (s) and Plan lengths vs. τ for Slippery Gripper.

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.

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 IPC: We entered a version of $POND$ in the non-deterministic track of the IPC that uses an enforced hill-climbing search algorithm (Hoffmann & Nebel, 2001), and the Reachable-SAG to extract a relaxed plan at each search node. The other planners entered in the competition are Conformant FF (CFF) (Hoffmann & Brafman, 2004) and t0 (Palacios & Geffner, 2006). 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.

Figures 21 to 26 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 blocksworld and sorting network 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.

Additional Non-Deterministic Domains: In addition to the results for the non-deterministic track of the IPC, we conducted our own external comparison of $POND$ with several of the best conformant: KACMBP (Bertoli & Cimatti, 2002) and CFF (Hoffmann & Brafman, 2004), and conditional planners: MBP (Bertoli et al., 2001) and BBSP (Rintanen, 2005). Based on the results of the internal analysis, we used relaxed plans extracted from a common $SLUG$, using the Reachable-SAG strategy. We denote this mode of $POND$ as "SLUG" in Figure 27.

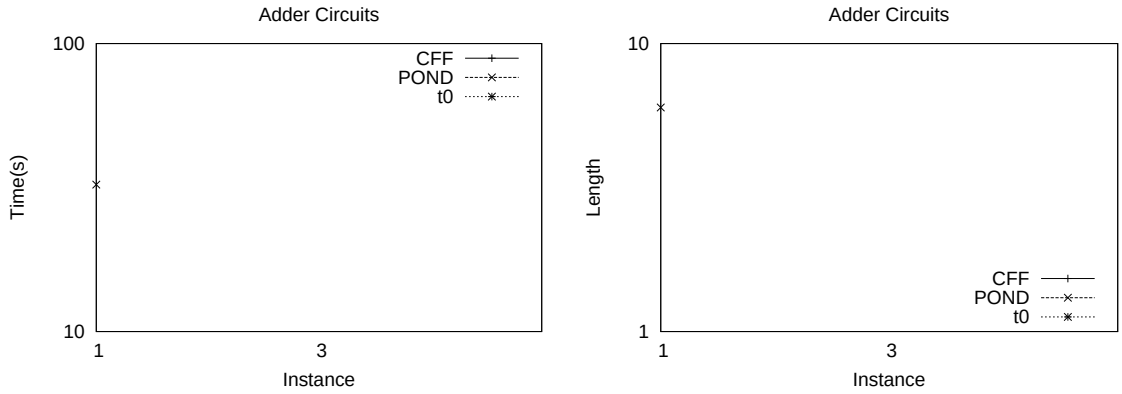

Figure 21: Run times (s) and Plan lengths IPC5 Adder Instances.

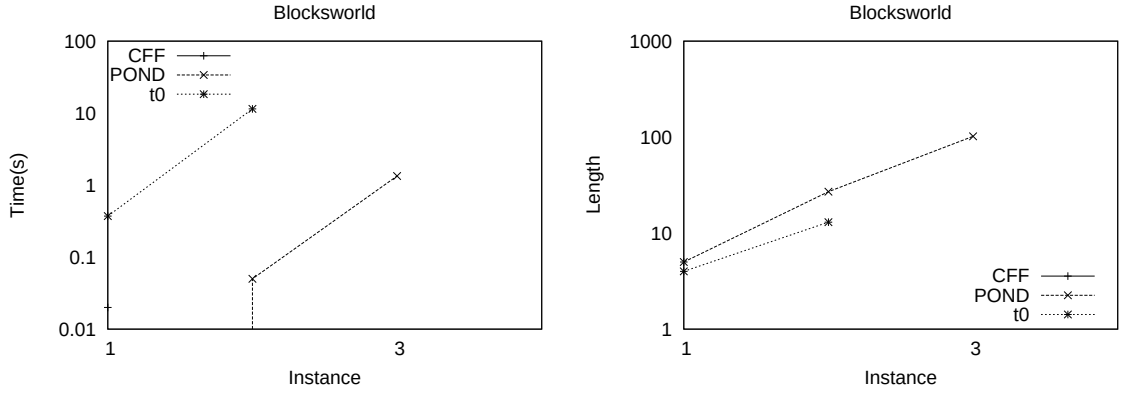

Figure 22: Run times (s) and Plan lengths IPC5 Blocksworld Instances.

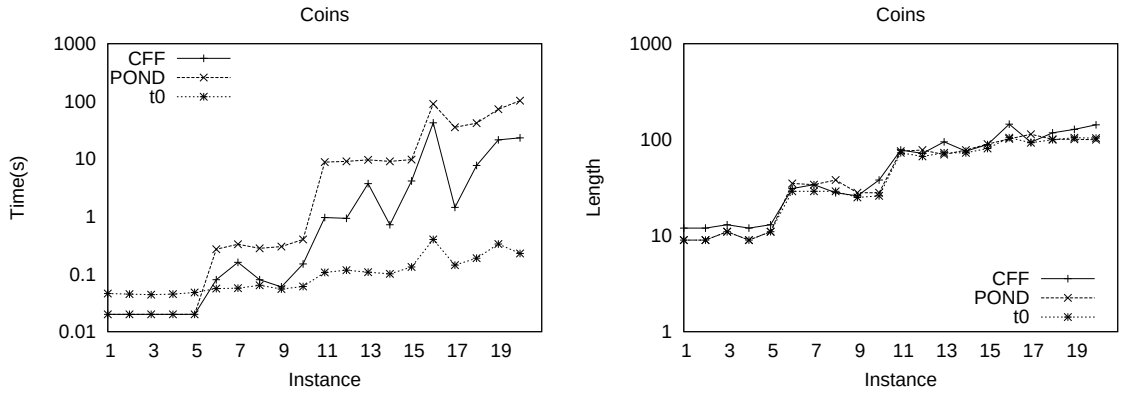

Figure 23: Run times (s) and Plan lengths IPC5 Coins Instances.

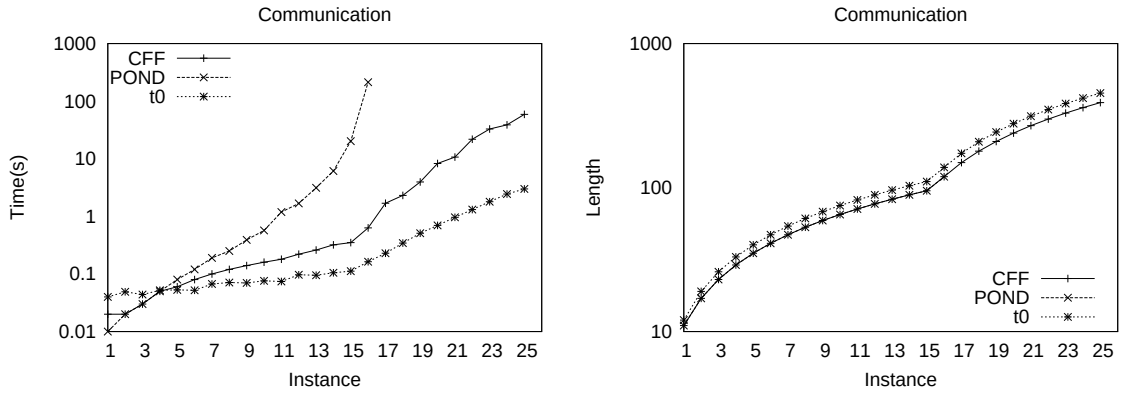

Figure 24: Run times (s) and Plan lengths IPC5 Comm Instances.

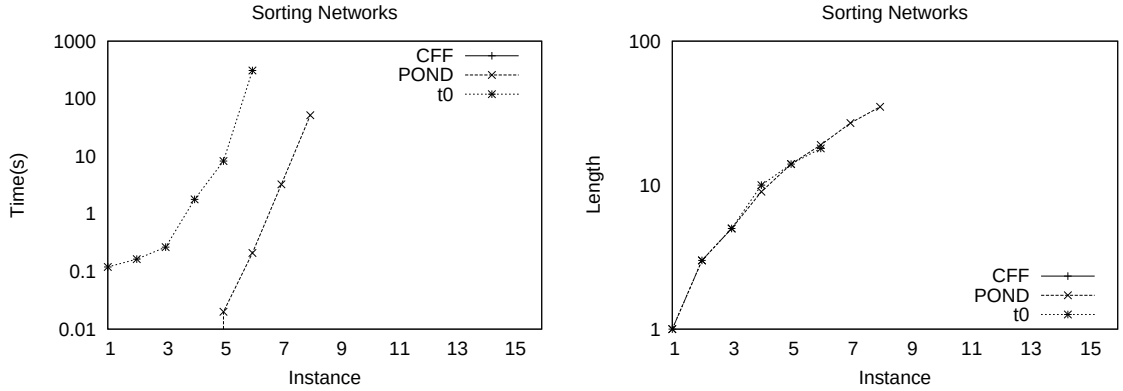

Figure 25: Run times (s) and Plan lengths IPC5 Sortnet Instances.

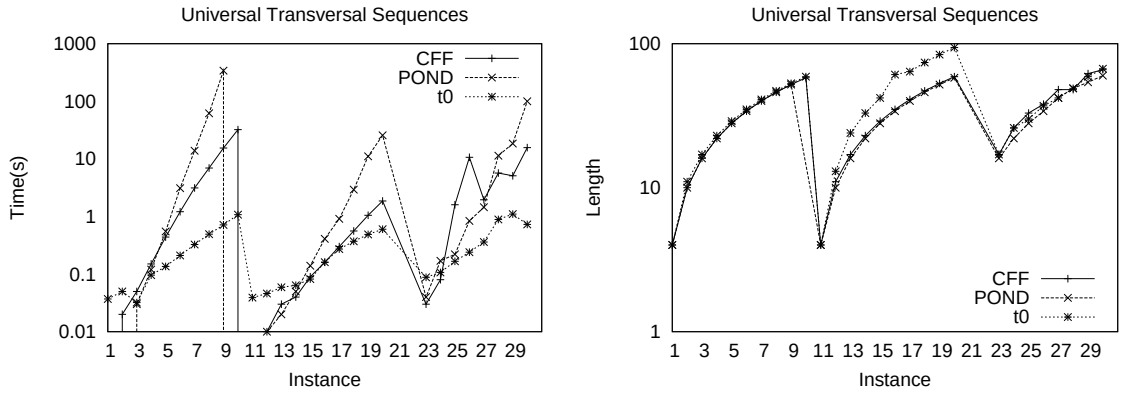

Figure 26: Run times (s) and Plan lengths IPC5 UTS Instances.

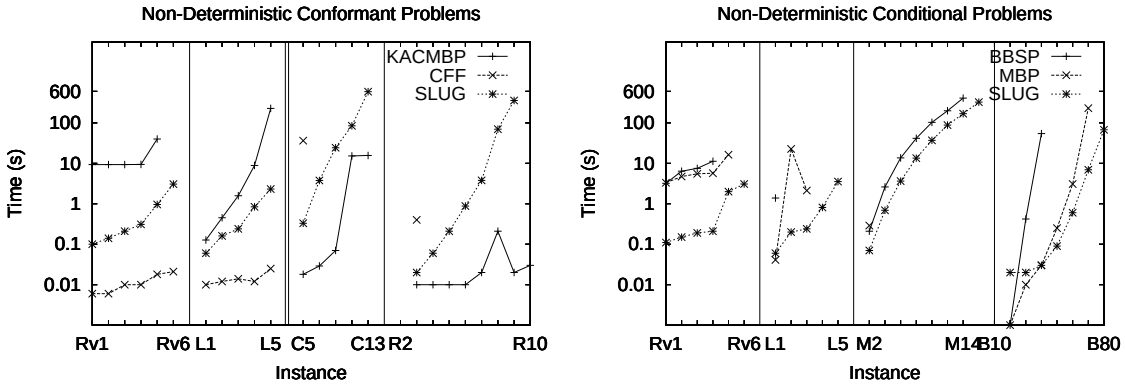

Figure 27: Comparison of planners on conformant (left) and conditional (right) domains. Four domains appear in each plot. The conformant domains include Rovers (Rv1-Rv6), Logistics (L1-L5), Cube Center (C5-C13), and Ring (R2-R10). The conditional domains are Rovers (Rv1-Rv6), Logistics (L1-L5), Medical (M2-M14), and BTCS (B10-B80).

In this external comparison, we chose domains that exhibit two distinct dimensions of difficulty. The Rovers and Logistics problems require significant planning effort to *cause* the goal. The Cube Center and Ring domains, on the other hand, require effort to *know* 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 also devised conditional versions of Logistics and Rovers domains by introducing sensory actions. We also drew conditional domains from the literature: BTCS (Weld, Anderson, & Smith, 1998) and a variant of Medical (Petrick & Bacchus, 2002). Our variant of Medical splits the multi-valued stain type sensor into several boolean sensors.

The results (Figure 27) show that $POND$ (using an AO* search) 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 $SLUG$, in Reachable-SAG mode). 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 19 identify, $POND$ generally out scales CPplan in every domain, but sacrifices quality.

Due to some unresolved issues with the PFF planner, we are only able to present results in the Logistics and Grid domains. The issues are of low impact because the Logistics and Grid domains are the most revealing in terms of planner scalability. We see that PFF scales reasonably well in 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 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 p2-2-4 (Figure 15) and Grid-0.8 (Figure 17). 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 $\mathcal{CSSAG}$, 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 Harwood and Warren (1974) 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. Another approach to guiding forward chaining planners is to use relaxed goal regression to compute the heuristic; work on greedy regression graphs (McDermott, 1999) as well as the GRT system (Refanidis & Vlahavas, 2001), 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 (Kambhampati, Parker, & Lambrecht, 1997). The main difficulty in applying such backwards planning graph approaches is the relative low quality of the derived heuristics.

Pattern databases (Culberson & Schaeffer, 1998) have been used in heuristic search and planning 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* (Bryce et al., 2006), which efficiently exploits the overlap in the planning graphs of members of a belief state.

Other works on improving planning graph construction, include Liu, Koenig, and Furcy (2002) where the authors explore issues in speeding up heuristic calculation in HSP. Their approach utilizes the prior planning graph to improve the performance of building the current planning graph (the rules which express the dynamic program of HSP's heuristic correspond to the structure of a

planning graph). 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, Long and Fox (1999) 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. Liu et al. (2002) seek to exploit the overlap between different graphs as well, but limit the scope to graphs adjacent in time.

The Prottle planner (Little, Aberdeen, & Theibaux, 2005) 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

We described a generalization of the traditional, *single source*, planning graph to the state agnostic, *multi source*, planning graph. Motivated by the fact that some of the best performing planners use forward chaining search and construct a planning graph for each search node, we developed the state agnostic planning graph as a way to avoid this mostly redundant computation and amortize the cost over multiple heuristic computations. We found that the state agnostic planning graph is most beneficial in cases where multiple heuristic computations overlap significantly, as in belief state space planning. Through extensive empirical evaluation, we showed that the state agnostic planning graph techniques are beneficial in three problem classes, and that our planner is competitive with the state of the art in non-deterministic and probabilistic planning.

An exciting future direction for state agnostic techniques includes that of problem agnostic planning graphs. It should be possible to extend the SAG techniques to a first-order representation where the planning graph is problem independent for a given domain of interest. Specifically, by using skolem objects with the domain's predicates and the action schemas, the problem agnostic planning graph could be constructed offline and used for any problem instance in the domain. The primary challenge is in the propagation and efficient inference over first-order label functions. Using advances in first-order decision diagrams may be a promising direction in which to pursue problem agnostic planning graphs.

Acknowledgements: This research is supported in part by the NSF grant IIS-0308139 and an IBM Faculty Award to Subbarao Kambhampati. Daniel Bryce was also partially supported by SRI International, and the ARCS Foundation. We thank the members of the Yochan research group 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

Bertoli, P., & Cimatti, A. (2002). Improving heuristics for planning as search in belief space. In *Proceedings of AIPS'02*, pp. 143–152.

Bertoli, P., Cimatti, A., Roveri, M., & Traverso, P. (2001). Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of IJCAI'01*, pp. 473–478.

Blum, A., & Furst, M. (1995). Fast planning through planning graph analysis. In *Proceedings of IJCAI'95*, pp. 1636–1642.

Bonet, B., & Geffner, H. (1999). Planning as heuristic search: New results. In *Proceedings of ECP'99*, pp. 360–372.

Bryant, R. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, *C-35*(8), 677–691.

Bryce, D., & Kambhampati, S. (2007). A tutorial on planning graph based reachability heuristics. *AI Magazine*, *28*(1), 47–83.

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

Bryce, D., Kambhampati, S., & Smith, D. (2008). Sequential monte carlo in probabilistic planning reachability heuristics. *Artificial Intelligence*, *172(6-7)*, 685–715.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(3), 318–334.

Cushing, W., & Bryce, D. (2005). State agnostic planning graphs. In *Proceedings of AAAI'05*, pp. 1131–1138.

de Kleer, J. (1986). An Assumption-Based TMS. *Artificial Intelligence*, *28*(2), 127–162.

Domshlak, C., & Hoffmann, J. (2006). Fast probabilistic planning through weighted model counting. In *Proceedings of ICAPS'06*, pp. 243–251.

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

Harwood, E., & Warren, D. (1974). Warplan: A system for generating plans.. Tech. rep. memo 76, Computational Logic Dept., School of AI, Univ. of Edinburgh.

Hoffmann, J., & Brafman, R. (2004). Conformant planning via heuristic forward search: A new approach. In *Proceedings of ICAPS'04*, pp. 355–364.

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

Hyafil, N., & Bacchus, F. (2003). Conformant probabilistic planning via CSPs. In *Proceedings of ICAPS' 03*, pp. 205–214.

Kambhampati, S., Parker, E., & Lambrecht, E. (1997). Understanding and extending graphplan.. In *Proceedings of Fourth European Conference on Planning*, pp. 260–272.

Koehler, J., Nebel, B., Hoffmann, J., & Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In *Proceedings of Fourth European Conference on Planning*, pp. 273–285.

Little, I., Aberdeen, D., & Theibaux, S. (2005). Prottle: A probabilistic temporal planner. In *Proc. of AAAI'05*, pp. 1181–1186.

Liu, Y., Koenig, S., & Furcy, D. (2002). Speeding up the calculation of heuristics for heuristic search-based planning.. In *Proceedings of the National Conference on Artificial Intelligence*, pp. 484–491.

Long, D., & Fox, M. (1999). Efficient implementation of the plan graph in stan.. *Journal of AI Research*, *10*, 87–115.

McDermott, D. V. (1999). Using regression-match graphs to control search in planning. *Artificial Intelligence*, *109*(1-2), 111–159.

Nguyen, X., Kambhampati, S., & Nigenda, R. S. (2002). Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence*, *135*(1-2), 73–123.

Palacios, H., & Geffner, H. (2006). Compiling uncertainty away: Solving conformant planning problems using a classical planner (sometimes). In *Proceedings of the National Conference on Artificial Intelligence*. AAAI Press.

Petrick, R., & Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of AIPS'02*, pp. 212–222.

Refanidis, I., & Vlahavas, I. (2001). The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research*, *15*, 115–161.

Rintanen, J. (2003). Expressive equivalence of formalisms for planning with sensing. In *Proceedings of ICAPS'03*, pp. 185–194.

Rintanen, J. (2005). Conditional planning in the discrete belief space. In *Proceedings of IJCAI'05*, pp. 1260–1265.

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

Weld, D., Anderson, C., & Smith, D. (1998). Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of AAAI'98*, pp. 897–904.

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

Zimmerman, T., & Kambhampati, S. (2005). Using memory to transform search on the planning graph. *Journal of AI Research*, *23*, 533–585.