Probabilistic Planning in the Graphplan Framework

Avrim L. Blum and John C. Langford

School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213-3891
{avrim,jcl}@cs.cmu.edu

Abstract. The Graphplan planner has enjoyed considerable success as a planning algorithm for classical STRIPS domains. In this paper we explore the extent to which its representation can be used for *probabilistic* planning. In particular, we consider an MDP-style framework in which the state of the world is known but actions are probabilistic, and the objective is to produce a finite horizon contingent plan with highest probability of success within the horizon.

We describe two extensions of Graphplan in this direction. The first, PGraphplan, produces an optimal contingent plan. It typically suffers a performance hit compared to Graphplan but still appears to be fast compared with other approaches to probabilistic planning problems. The second, TGraphplan, runs at essentially the same speed as Graphplan, but produces potentially sub-optimal policies: TGraphplan's policy selects the first action on the highest probability *trajectory* from its current state to the goal. Ideally, we would like an optimal planner for probabilistic domains with the same speed that Graphplan would have if the domain were made deterministic. By comparing the speed and quality of these two planners to each other and to other existing planners, we are able to estimate how far off we are from our ideal.

PGraphplan is based on a forward-chaining search, unlike the backward-chaining search of the standard Graphplan algorithm. Thus, one focus of this paper is exploring the extent to which Graphplan's representation can be used to speed up forward search in addition to the backward search for which it was originally intended.

1 Introduction

The Graphplan planner is based on compiling a STRIPS-style planning problem into a compact (polynomial size) graph structure in which information can be quickly propagated to aid in the search for a plan [BF97]. Empirical results indicate that this approach is often quite fast compared to other traditional methods [BF97, Byl97, KNHD97]. Since its initial formulation, this basic algorithm has been extended and improved in a number of ways, such as allowing operators with contingent effects [GK97, KNHD97, ASW97], handling certain kinds of uncertainty [SW98, WAS98], and further speed improvements [KNHD97, KLP97].

In this paper, we explore the question: to what extent can the speed of Graphplan be extended to probabilistic domains? We consider the setting in which initial conditions and current state are known, but actions can be probabilistic, having several possible outcomes. This falls into the framework of Markov Decision Processes (MDPs). In particular, instead of looking for a plan that consists of an action sequence, we will be looking for a contingent plan that tells which action to take based on the history of outcomes so far. The specific kind of MDP we focus on is one in which, as with STRIPS planning, our objective is to satisfy all of a given set of conjunctive goals. Our aim will be to produce an optimal finite-horizon contingent plan, where by "optimal" we mean maximizing the probability of success within the time window (though much of the discussion applies to objectives such as minimizing the expected completion time as well).

Ideally, we would like an optimal planner for probabilistic domains with the same speed that Graphplan would have, had the domain been deterministic. Embedding probabilistic outcomes into Graphplan's graph structure is straightforward. However, how to search for a plan is less obvious. Planning in probabilistic domains is inherently more complex than in deterministic domains, in part because a complete policy can have size exponential in the horizon time whereas in deterministic domains, plan sizes are linear in the horizon time.

We instead present two planners: PGraphplan and TGraphplan. TGraphplan runs at essentially the same speed as Graphplan, but produces potentially suboptimal policies. Specifically, TGraphplan finds the highest probability *trajectory* from the start state to the goal, which can then be turned into a kind of greedy contingent plan in a natural way. In contrast, PGraphplan produces *optimal* contingent plans but typically suffers a performance hit. Unlike the backward-chaining search of Graphplan, PGraphplan is based on a *forward-chaining* search through the planning graph, which is much more natural for producing optimal plans in these probabilistic settings (we discuss this further in Section 3.3). PGraphplan is like a standard top-down Dynamic Programming algorithm, but uses information stored in the graph to prune its search. The difficulty here is that for forward-chaining search, Graphplan's mutual exclusion relations are not especially helpful, and instead PGraphplan uses other kinds of information propagated *backwards* from the goals. In this sense, our objectives are similar to those of Kambhampati and Parker [KP99].

We compare PGraphplan and TGraphplan to each other and to several other probabilistic planners such as Buridan [KHW95], SPI [BDG95], and vanilla dynamic programming, on a variety of domains. Comparing PGraphplan and TGraphplan to each other allows us to estimate how far off we are from our ideal objective. In particular, from the perspective of Graphplan, two key questions are: "to what extent can the planning graph representation be used to speed up *forward search*?" and "Can Graphplan's backward-chaining search produce a near-enough-optimal solution?"

1.1 Other related work

There is a long history of work in probabilistic planning. One of the first planners designed specifically for probabilistic STRIPS-style domains is Buridan [KHW95], extended to the contingent planner C-Buridan by [DHW94]. Unlike our work, these planners also considered partially observability, but in general were quite slow. Closer to our motivations is Structured Policy Iteration (SPI) of [BDG95]. SPI is designed for the fully observable MDP setting, and attempts to use the propositional representation of states and actions to find an optimal policy without expanding out the entire state space. Zander [ML99] and Maxplan [ML98] by Majercik and Littman compile a planning problem onto a stochastic satisfiability problem, solving the problem in that representation. Maxplan is a blind planner, while Zander produces contingent plans.

There has also been work on probabilistic planning explicitly using representations motivated by Graphplan. In particular, Boutilier et al. [BBG98] generalize Graphplan's pairwise mutual-exclusion constraints to k-wise constraints, and examine the reduction in the size of the MDP that is implicitly represented by the layer at which the planning graph levels off.

The work of Dean et al. [DKKN95] has a close connection to the goals of TGraphplan. That work finds an initial trajectory using a forward search, and then attempts to interpolate (in an anytime fashion) towards an optimal solution.

2 Graphplan and its representation

Graphplan [BF97] is a planner for STRIPS domains. These domains consist of *initial conditions* that describe the starting state of the world, *operators* which describe the legal actions that may be performed, and *goals* representing those facts that we wish to be true at the end of a plan. Operators have conjunctive preconditions, add-lists, and delete-lists. For instance, in a blocks-world, the initial conditions specify the starting configuration of the blocks, the goals specify what we wish to be true about the blocks at the end of the plan, and the operators specify our legal moves.

Graphplan is based on compiling such a planning problem into a polynomial-size structure called a *planning graph*. A planning graph is a directed, leveled graph. The first level has one node for each proposition in the initial conditions. The next level has a node for each action (fully-instantiated operator) that might possibly be performed at time 1: that is, one node for each action whose preconditions all exist in the initial conditions. The next level of the graph lists all propositions that might be true at time 2: namely, the union of the add-effects of all actions in the previous action level, including the no-ops (for consistency, Graphplan also includes a *no-op* action for each proposition that simply propagates the proposition forward in time). The next level consists of actions that might possibly be performed at time 2, and so forth. Edges in a planning graph connect actions to their preconditions and their add and delete effects.

Graphplan begins by creating a planning graph forward from the initial conditions until all of the goals appear in the graph. It then searches in a backward-chaining fashion. If the recursive search does not find a plan, then the graph is extended one more time step and the process is repeated. The planner also has a mechanism for eventually halting if, in fact, the problem has no solution.

The planning graph allows for several important optimizations including:

1. Propagating pairwise *mutual exclusion* relations between propositions and between actions forward through the graph while it is being created. These relations tell the planner that, for instance, propositions P and Q cannot be both made true by time step 4, and are used to prune the backward search.
2. Allowing the plan to perform several operators in parallel so long as they do not conflict with each other.

Memoizing the results of unsuccessful searches is also used, and improvements to Graphplan use a number of other optimizations as well. For more details see [BF97, KNHD97, KLP97].

Representing probabilistic actions

In this paper, we consider probabilistic actions. Specifically, we consider operators with conjunctive preconditions and with several sets of add and delete effects, each set having an associated probability. For example, an "open door" action might require that the door be unlocked, and with 88% probability actually open the door, with a 10% probability do nothing, and with 2% probability pull off the handle. We assume that the outcome that actually occurred will always be known when executed (the MDP not the POMDP setting).

To represent these kinds of operators, the graph is constructed in the normal manner except that each action contains a list of possible "outcomes", each with its associated probability, and then each outgoing edge of the action indicates which outcome produced that edge. For instance, suppose that we have an operator that with probability 0.7 deletes $G0$ and adds $G1$, and with probability 0.3 just adds $G1$. Then, there would be two outcomes, one with probability 0.7 and one with probability 0.3. There would also be three outgoing edges. One edge would be a delete edge leading to $G0$ and associated with the first outcome, one would be an add edge leading to $G1$ and associated with the first outcome, and one would be an add edge leading to $G1$ and associated with the second outcome. Since each outcome contains its probability, this representation is sufficient to reconstruct the definition of the operator.

3 PGraphplan

A standard algorithm for solving a finite-horizon MDP is dynamic programming. One begins by computing the value of each state for a time horizon of 0, then uses that to compute values for a time horizon of 1, and so on up to the given horizon t_{max}. In propositional planning, because we have an initial state and the state space is not explicitly enumerated, it is more natural to do this in a recursive top-down fashion, as described in Figure 1. Notice that top-down DP, because it stores the result of each computation, explores each state at most once per time step just like bottom-up DP.

DPsolve(state s, time t): Compute $value(s,t)$ = best possible probability of success within the time window for a contingent plan starting from state s at time-step t.

1. If $t = t_{max}$ then return 0 if goals not satisfied, else return 1.
2. If already-visited(s, t), then return the previously-computed value.
3. For each possible action a,
   (a) For each possible state s' that could result from taking action a in state s, recursively call DPsolve($s', t+1$).
   (b) Let $value(s,a,t)$ be the probability-weighted average of the results.
4. Let $value(s,t) = \max_a value(s,a,t)$. Return this quantity, after first storing it in case we ever visit s at time t in a later recursive call. Also, return $\text{argmax}_a value(s,a,t)$ as the optimal action.

Fig. 1. Standard top-down Dynamic Programming

PGraphplan begins with this vanilla top-down Dynamic Programming as its starting point,[1] but uses the planning graph to prune its search. In particular, PGraphplan propagates two distinct kinds of information through the graph. The first kind tells the planner how various nodes in the graph contribute to solving the problem goals, and the second focuses more on how near or remote that contribution is. Both kinds of information are used in the same way: to tell the planner when the path it is currently exploring provably cannot reach the goals within the given time horizon and therefore it may safely return failure in its recursive call. The two types of information are given below.

3.1 Unary and pairwise "needed" nodes

One very simple optimization is to delete nodes from the planning graph that do not have any paths to the goal literals, by performing a backward sweep through the graph. This simple form of relevance analysis effectively collapses multiple states together, and is roughly equivalent to the notion of relevance used by Knoblock [Kno94]. For an even greater savings, we assign to each node not removed a vector indicating which goal literals *are* reachable from this node. The planner uses this information when at some state S by looking at the vectors assigned to each literal in S, and checking to see if any goal is missing from all the vectors; if so, then it knows it may backtrack immediately. In fact, if there is only one non-noop action that creates some goal, then the preconditions to this action can be used instead of that goal for the purposes of defining these vectors. This can be viewed as a crude but computationally cheap version of the fact-generation trees of Nebel et al. [NDK97].

[1] In particular, this is a forward-chaining search, and allows only one non-noop action per time step. It would be interesting to consider parallel actions as in Graphplan, but the computation becomes quite a bit hairier in the probabilistic setting.

A more interesting version of this information is a pairwise notion of "neededness". Suppose node P *does* have one or more paths to goal literals, but all of these paths use node Q as well (e.g., all potentially useful actions that have P as a precondition also have Q as a precondition); in that case, we can add a "P needs Q" edge to the graph, allowing the planner to drop literal P from its state if literal Q is not also present. Specifically, we say that proposition P *needs* proposition Q if each action with P as a precondition either (a) has Q as a precondition too, or (b) needs some action that has Q as a precondition. Similarly, a non-noop action A needs noop B if all add-effects of A need the result of B (and none are equal to it); a noop action A needs action B if the result of A needs a proposition that can be created only through B.[2]

For example, consider a domain in which a robot, initially with key in hand, must move to the end of the hallway, unlock a door at the end, and then drop the key in order to empty its hand for some subsequent task. In this case, the fact that the move-forward action needs the noop-have-key action gets propagated back through the entire graph, ensuring that the robot does not drop the key prematurely.

It is interesting to compare the information propagated here to the "backward mutex" constraints used by Kambhampati and Parker [KP99]. They propagate the notion that "P and Q are redundant" in the sense that one can confidently remove one of them from any state that contains both. That information seems more difficult to propagate appropriately in a probabilistic setting and we have not attempted to do so.

3.2 Value propagation

The idea for this second kind of information is to store values on nodes of the graph that allow one to compute a "permissible heuristic" for an A^*-style search. This method empirically produces an even greater savings than the one above.

Imagine that reaching a goal state (one in which all the problem goals are satisfied) is worth t_{max} dollars, but performing a non-noop action costs \$1. In this case, the *true value* of a state that can reach the goals in i steps is $\$(t_{max} - i)$. Notice that if a state S at time step t is worth less than $\$t$, then this means it cannot possibly reach the goals by time t_{max}. Thus, a magic oracle that returned the true value of any given state would be quite useful. The idea of *value propagation* is to propagate heuristic values (hvalues) through the nodes of the graph such that the heuristic value of any state S, defined to be the *sum* of the hvalues of the nodes of the state, is guaranteed to be greater than or equal to the true value of S. If the planner finds that the heuristic value of its current state at time t is less than $\$t$, then it can confidently backtrack immediately.

Specifically, we begin by dividing the final value t_{max} evenly among the problem goals at time t_{max}, breaking ties arbitrarily (all hvalues will be integral). We propagate hvalues on propositions at time $t+1$ backwards to actions at time

[2] This asymmetry is the result of the planner only allowing one non-noop action per time step.

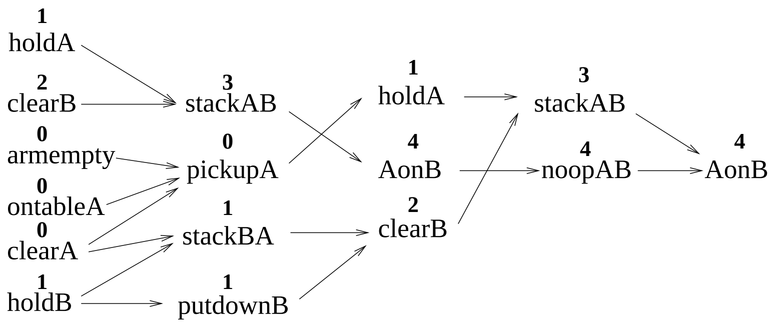

Fig. 2. RHS of graph for simple blocksworld domain, ending at $t = 4$. Some noops are not shown. Numbers indicate hvalues. Notice that the planner will return from any state at time $t = 2$ that doesn't have `clearB` (it will never reach the impossible state in which it is holding A and B).

t as follows: the hvalue of a noop is the hvalue of its effect; the hvalue of a non-noop action a is $-1 + \sum_{e \in \mathrm{addeffect}(a)} hvalue(e)$, where preconditions that are not deleted are viewed as add-effects for this purpose. We view probabilistic actions (which may have several possible outcomes) as if they were user-controllable for this computation: that is, a probabilistic action with k possible outcomes is treated as k separate deterministic actions, one for each possibility, each with its own value.

Values on actions at time t are then propagated to propositions at time t as follows. We begin with the noops: each gives its value to its precondition. We then consider each non-noop action in turn.[3] For each action, we compare its value to the sum of the hvalues of its preconditions. If its value is larger, we give the difference to the preconditions; the semantics is that the total value of the preconditions is the maximum of performing or not performing this action. As a heuristic, we distribute the value evenly among just the preconditions deleted by the action (breaking ties arbitrarily) if any exist; if none exist, we distribute the value evenly among all preconditions. (Actions with no preconditions are given a fake "always-true" precondition that is true in the initial conditions and never deleted.) An example is given in Figure 2.

This method is *legal* in the sense that if a state at time t has a sum of hvalues less than t, then it cannot possibly reach the goals. Unfortunately, it can at times be an over-estimate, because of a double-counting that may occur as values are propagated.

The two kinds of information described above each have a different purpose. The first kind asks *is* a node useful, and *for what*? The second focuses more on *how long* the path is from some node to its eventual use.

[3] The order in which the actions are examined can, in principle, affect the values, but we have not attempted to optimize the ordering in the planner.

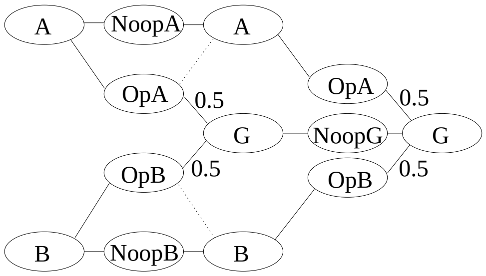

Fig. 3. Planning graph for the simple domain of section 3.3. Dotted lines represent deletes, and irrelevant nodes have been removed.

3.3 Why forward-chaining?

Finding an optimal policy with a planning graph appears to be considerably more difficult for a backwards search than a forwards one. Consider, for example, a domain in which the initial conditions contain the literals A and B, and the goal is to achieve G. There are 2 operators: OpA requires A, deletes A and with probability 0.5 adds the goal G; OpB requires B, deletes B and with probability 0.5 adds G. See Figure 3.

For this domain, the plan OpA, OpB has a 75% chance of success. However, to find this plan by reasoning backwards in the planning graph appears to require combining seemingly unrelated goal sets. In particular, to produce the optimal plan, we need at time 2 to consider the goal set $\{G\}$ (which achieves our goal via a noop) and $\{B\}$ (which has probability 0.5 of achieving the goal via OpB) together. We then need to realize that if OpA is performed from $\{A, B\}$ at time 1, then each of the two outcomes of the action will lead to one of these goal sets at time 2. (Note: these are two distinct *goal sets*, not just two goals in the same set.) This kind of reasoning seems possible, but it also seems that it would require time quadratic in the number of goal sets at any given time step (or cubic if some action has three possible outcomes). The problem stems from the fact that we are dealing with *goal sets* (which are only subsets of states) rather than the states themselves in our backward-chaining search, unlike in bottom-up DP. Perhaps some way can be found around these difficulties, or some way to make this not too expensive in "typical" domains. In any case, this is the reason we choose to use forward chaining.

Smith and Weld [SW98] and Weld et al. [WAS98] use a different approach that allows for backward-chaining search on goal-sets in the presence of uncertainty. They handle uncertainty in the framework of Graphplan by essentially creating one graph for each "possible world". For instance, if one views the outcome of a probabilistic action as being determined by the flip of an associated coin, then there would be one graph for each possible sequence of coin flips. Ideally, one would like to use the same kind of search but at the same time handling the (possibly exponentially many) possible worlds within the context of a single planning graph.

4 TGraphplan

TGraphplan uses a backward chaining search (essentially the same as the original Graphplan) and finds an optimal trajectory from the initial state to the goals. A *trajectory* is a sequence of actions and outcomes leading from one state to another, such as "I turn the key in the ignition, the car starts, I drive to the airport, get there in time for my plane, I catch my flight, and arrive at my destination." An optimal trajectory is the highest probability sequence of states and actions leading to the goal. When two trajectories have the same utility, trajectories which do noops later are preferred as in Graphplan. This bias is especially important in a probabilistic setting because it maximizes the amount of time available for recovery in case the trajectory experiences a failure.

TGraphplan can be used as a subroutine to manufacture a complete policy by simulating execution of the trajectory to detect the state and time of unexpected outcomes not on the trajectory. Optimal trajectories for these unexpected outcomes can then be found and simulated forward to find new unexpected outcomes. The recursion terminates when there are no more unexpected outcomes. More naturally, TGraphplan can be used in an online fashion after the optimal trajectory is discovered. While the first action is executing (in the real world), TGraphplan can be planning for subsidiary trajectories. Thus, the important time quantity to measure for TGraphplan is the time to find the optimal trajectory rather than the time to find a complete policy.

TGraphplan starts by building a planning graph with probabilistic outcomes included. The backward-chaining search is done exactly as in Graphplan, except instead of the recursion returning a binary value (success/fail), it returns a real-valued success probability for the best sub-trajectory. In the TGraphplan search algorithm, the probability of the trajectory is determined by recursively multiplying the probability of a step succeeding by the probability of the partial trajectory already explored. The same set of optimizations that Graphplan uses are used by TGraphplan. Mutual exclusions are more complicated because it is possible for an operator to interfere with one outcome of another probabilistic operator but not with a second outcome. In TGraphplan, a pair of operators is made exclusive when any pair of outcomes interfere. Instead, we could lose the notion of "exclusive operators" and replace it with a notion of "exclusive outcomes" but we do not do that. TGraphplan can also be run in an iterative deepening mode as for Graphplan. In order to do this, the desired trajectory probability must be given in advance. The algorithm terminates as soon as a trajectory with at least the desired probability is found.

The TGraphplan algorithm is fast in comparison to PGraphplan because it only outputs *partial policies*, solves an inherently less complex problem and uses a backward chaining search which can take advantage of the mutual exclusions propagated forward in the building the graph. It is interesting to consider when TGraphplan will produce a (near) optimal policy and when the choices it makes will be substandard. This will be discussed in the following examples.

5 Examples and Empirical Results

We now describe several example domains and give results of running PGraphplan, TGraphplan, and other comparison planners on them. The purpose of these experiments is twofold: to examine the speed of the proposed planners, and to explore the extent to which the plans produced by TGraphplan are optimal or close to it. PGraphplan and TGraphplan are written in C. The planners we compare to are

- Top-down Dynamic Programming (Figure 1).
- Buridan [KHW95]: in compiled Lisp.
- Blackbox [KS99] (on deterministic domains): written in C.
- SPI [BDG95]: an infinite-horizon discounted MDP solver, written in C.

Moats and Castles: This simple domain is an adaptation of one by Majercik and Littman [ML98], in which the goal is to build a sand castle on the beach. In our version, there are two operators. Dig-moat is a deterministic action that increases the depth of a protective moat (there are 5 discrete depths), and build-castle is a probabilistic action for creating the castle, whose success probability increases with the depth of the moat. The optimal finite-horizon plan for this problem will consist of some number of dig actions, followed by a remainder of builds, where the number of digs depends on the time horizon and the specific success probabilities. The problem as stated has a very small state space; to make things more interesting, we consider having multiple castles, each with its own moat.

We consider this domain in part because it gives a simple illustration of when TGraphplan does or does not produce optimal plans. In particular, for the case of one castle, the optimal *trajectory* is to dig as many times as possible, followed by one final build operation (and then noops up to the time horizon). This may or may not be a trajectory of the optimal policy; in particular, the optimal policy may have fewer dig operations, if, for instance, deepening the moat has only a small effect on the success probability of the build operation.

We describe performance results in Table 1. For this domain, when the time horizon is large, the information propagated by PGraphplan does not provide much of a gain. That is because in this case, almost all states *can*, in principle, lead to solving the goals, and the information propagated is only intended to prune states with no chance of success. TGraphplan scales better with time horizon but worse with number of castles, compared to PGraphplan; the latter effect appears to occur because of interaction between parallel and probabilistic actions.

Probabilistic Blocks: In the standard blocks-world domain, we can `pickup` a block from the table, `putdown` a block onto the table, `unstack` a block from another block, and `stack` a block on another block. So, for instance, to pick up a block from the table and place it on another block takes two time steps. Imagine we augment this domain with a probabilistic operator `faststack` that can move a block from the table onto the top of another block in one time step, but it only succeeds with a 70% probability; with a 30% probability it has no effect. This setting is interesting because the optimal action depends on the time-to-go.

	1 castle	2 castles	3 castles	4 castles
DP	.01 [.01] sec 44 [114] states	.01 [.01] sec 448 [2300] states	.01 [.09] sec 1977 [22113] states	.03 [0.54] sec 5996 [138308] states
PGP	.01 [.01] sec 40 [110] states	.01 [.01] sec 256 [1900] states	.01 [.07] sec 909 [16389] states	.01 [0.36] sec 1776 [89276] states
TGP	.01 [.01] sec 37 [97] sets	.01 [.01] sec 259 [954] steps	.01 [.05] sec 1959 [9974] sets	.07 [0.45] sec 13855 [102569] steps
SPI	.05 sec	0.21 sec	2.73 sec	36.3 sec
Bur	1 sec [> 5 min] 427 [>60k] plans	6 sec [> 5 min] 4823 [>60k] plans	> 5 min >60k plans	> 5 min >60k plans

Table 1. PGraphplan (PGP), TGraphplan (TGP), top-down DP (DP), SPI, and Buridan(Bur) on the moat and castles problem with varying numbers of castles. SPI is run with discount factor 0.9. The other planners are given time horizon of 5 or 10 (for Buridan, this is done by providing a desired success probability); values for horizon of 10 are given in square brackets. Running times are given on a PII-450 xeon with 512 MBytes of memory.

Note that the optimal trajectory will always choose the deterministic operators if there is sufficient time, otherwise it will choose `faststack`. Thus, in this domain, TGraphplan does, in fact, lead to an optimal policy. Results for this domain are given in Table 2.

8-puzzle and Flat-tire: In order to compare to deterministic planners, we also considered several deterministic domains, in particular the flat-tire problem of Russell, and the 8-puzzle problem. The 8-puzzle problem is interesting because there is no special advantage to backward-chaining on this problem. We consider the goal of achieving board state **ABCDEFGH_** (reading left to right, top to bottom) from two different initial states: one in which a solution requires 18 steps and one in which a solution requires 30 steps (this is the case of initial board **HGFEDCBA_**). Results are given in Table 3. Note that PGraphplan is the fastest of all planners tested (even the deterministic ones) on this problem.

6 Discussion and Conclusions

PGraphplan performs a forward search to find an optimal contingent plan, using information stored in the graph to collapse and prune away unnecessary states. On problems such as blocks-world, flat-tire, and 8-puzzle, this pruning provides a substantial speedup. The value information seems to provide the greatest savings in general, with the information on the eventual purpose, if any, of each node in the graph providing a smaller but still significant help. PGraphplan is still in general slower than Graphplan, in part we believe because we have not yet found the best way to propagate information backwards through the planning graph. One possible avenue would be to better integrate the two kinds of information discussed above, for instance by separating the hvalues into different "flavors" depending on which goal they are intended for. It would also be helpful to propagate more probabilistic information, such as an upper-bound on the probability of reaching the goals, which could then be used to prune search. Finally, it would be interesting to explore whether knowledge of the TGraphplan's more quickly-constructed policy could guide PGraphplan's search.

	2 blocks ($t = 4$)	3 blocks ($t = 8$)	7 blocks ($t = 18$)	7 blocks ($t = 20$)	7 blocks ($t = 22$)	7 blocks ($t = 24$)
DP	.01 sec 18 states	.01 sec 194 states	8.27 sec 974k states	14.53 sec 1607k states	20.88 sec 2239k states	25.43 sec 2646k states
PGP	.01 sec 6 states	.01 sec 111 states	1.34 sec 163k states	4.97 sec 599k states	10.83 sec 1211k states	14.49 sec 1549k states
TGP	.01 sec 11 sets	.01 sec 29 sets	.25 sec 126 sets	.42 sec 718 sets	.53 sec 1280 sets	.61 sec 1323 sets
SPI	1.17 sec	28.54 sec	–	–	–	–
Bur	1 sec 783 plans	– >100k plans	– >100k plans	– >100k plans	– >100k plans	– >100k plans

Table 2. Results for the probabilistic blocks problem described in the text. The initial state is a tower (`ABCD...`) and the goal is the same tower except that the block that used to be on top is now on the bottom (`BCD...A`). SPI finds a policy that applies from all possible starting states, so in a sense it is unfairly penalized in this experiment (to partially alleviate this, for SPI we discarded actions not used in any reasonable trajectory from our initial state from the domain description).

	flat-tire (19 step)	8-puzzle (18 step)	8-puzzle (30 step)
DP	0.3 sec 48851 states	0.63 sec 41242 states	27.61 sec 1777759 states
PGP	.05 sec 3661 states	0.14 sec 2069 states	6.56 sec 437722 states
TGP	.04 sec 411 sets	0.8 sec 461 sets	43.56 sec 98946 sets
Blackbox	.08 sec	0.93 sec	timeout
BlackboxGP	.08 sec	0.93 sec	45.56 sec
BlackboxWS	0.18 sec	timeout	timeout

Table 3. Results on the flat-tire domain, and the easy and hard 8-puzzle problems. Blackbox was run in its default mode, with `-solver graphplan` (BlackboxGP), and with `-solver walksat` (BlackboxWS).

The planners implemented apply to problems for which the objective is to maximize the probability of satsifying the problem goals within the given time window, or the related goal of minimizing expected completion time. More general MDP problems are often given by specifying rewards for performing certain (noop or non-noop) actions. It appears that some of the kinds of information propagated here should be useful in those more general settings, and it would be interesting to see if this generalization could be made without a sacrifice in performance.

Acknowledgements. This research is sponsored in part by NSF National Young Investigator grant CCR-9357793, NSF grant CCR-9732705 and an AT&T / Lucent Special Purpose Grant in Science and Technology. We would like to thank the anonymous reviewers for their detailed, thoughtful, and helpful comments.

Source code is available at `http://www.cs.cmu.edu/~avrim/pgp.html`.

References

[ASW97] C. R. Anderson, D. E. Smith, and D. S. Weld. Conditional effects in Graphplan. unpublished manuscript, 1997.

[BBG98] C. Boutilier, R. I. Brafman, and C. Geib. Structured reachability analysis for Markov Decision Processes. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–98)*, pages 24–32, 1998.

[BDG95] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *IJCAI95*, pages 1104–1111, Montreal, 1995.

[BF97] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[Byl97] T. Bylander. Linear programming heuristic for optimal planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 1997.

[DHW94] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, pages 31–37, 1994.

[DKKN95] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76, 1995.

[GK97] B. C. Gazen and C. A. Knoblock. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In *Proceedings of the Fourth European Conference on Planning*, 1997.

[KHW95] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76:239–286, 1995.

[KLP97] S. Kambhampati, E. Lambrecht, and E. Parker. Understanding and extending Graphplan. In *Proceedings of the Fourth European Conference on Planning*, September 1997.

[KNHD97] J. Koehler, B. Nebel, J. Hoffman, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proceedings of the Fourth European Conference on Planning*, September 1997.

[Kno94] C. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[KP99] S. Kambhampati and E. Parker. Making graphplan goal-directed. In *ECP99*, September 1999.

[KS99] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *IJCAI99*, 1999.

[ML98] S. M. Majercik and M. L. Littman. MAXPLAN: a new approach to probabilistic planning. In *AIPS*, pages 86–93, 1998.

[ML99] S. M. Majercik and M. L. Littman. Contingent planning under uncertainty via stochastic satisfiability. In *AAAI*, 1999.

[NDK97] B. Nebel, Y. Dimopoulos, and J. Koehler. Ignoring irrelevant facts and operations in plan generation. In *Proceedings of the Fourth European Conference on Planning*, pages 338–350, September 1997.

[SW98] D. E. Smith and D. S. Weld. Conformant graphplan. In *AAAI98*, 1998.

[WAS98] D. S. Weld, C.R Anderson, and D. E. Smith. Extending graphplan to handle uncertainty and sensing actions. In *AAAI98*, 1998.