Conditional, Probabilistic Planning: A Unifying Algorithm and Effective Search Control Mechanisms

Nilufer Onder
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
nilufer@cs.pitt.edu

Martha E. Pollack
Department of Computer Science
and Intelligent Systems Program
University of Pittsburgh
Pittsburgh, PA 15260
pollack@cs.pitt.edu

Abstract

Several recent papers describe algorithms for generating conditional and/or probabilistic plans. In this paper, we synthesize this work, and present a unifying algorithm that incorporates and clarifies the main techniques that have been developed in the previous literature. Our algorithm decouples the search-control strategy for conditional and/or probabilistic planning from the underlying plan-refinement process. A similar decoupling has proven to be very useful in the analysis of classical planning algorithms, and we show that it can be at least as useful here, where the search-control decisions are even more crucial. Previous probabilistic/conditional planners have been severely limited by the fact that they do not know how to handle failure points to advantage. We show how a principled selection of failure points can be performed within the framework our algorithm. We also describe and show the effectiveness of additional heuristics. We describe our implemented system called Mahinur and experimentally demonstrate that our methods produce efficiency improvements of several orders of magnitude.

Introduction

Several recent papers describe algorithms for generating conditional and/or probabilistic plans. Unfortunately, these techniques have not been synthesized into a clear algorithm. In this paper, we present a unifying algorithm that incorporates and clarifies the main techniques that have been developed.

Our algorithm has three useful features. First, it decouples the search-control strategy for conditional probabilistic planning from the underlying plan-refinement process. A similar decoupling has proven to be very useful in the analysis of classical planning algorithms (Weld 1994), and we show that it can be at least as useful here, where search-control decisions are even more crucial. We achieve the decoupling by treating the possible failure points in a plan as flaws. By a *failure point*, we mean a part of the plan that (a) involves a branching action, i.e., one whose outcome is uncertain, and (b) relies on a particular outcome of that action. Where classical planning algorithms consider open conditions and threats to be flaws, we add possible failure points into this set. Decisions about whether and when to handle each failure point can then be encoded as part of the search-control strategy.

Second, we repair plan failures in a direct way, using three logically distinct techniques: (1) *corrective repair*, originally introduced in the work on conditional planning, which involves reasoning about what to do if the desired outcome of a branching action does not occur; (2) *preventive repair*, originally introduced in the work on probabilistic planning, which involves reasoning about how to help ensure that the desired outcome of a branching action will occur; and (3) *replacement*, implemented by backtracking in the planning literature, which involves removing the branching action and replacing it with an alternative.

Finally, our planner can generate conditional plans with merged branches: if two branches involve different steps at the beginning but the final steps are the same, the final part can be shared. This way the cost of generating the same part twice can be avoided.

Previous probabilistic/conditional planners have been severely limited by the fact that they do not know how to handle failure points to advantage. For all but very small domains, the search space explodes quickly if plan failures are considered indiscriminantly. We show how a principled selection of failure points can be performed within the framework our algorithm. We also describe and show the effectiveness of a few additional heuristics. We describe our implemented system, called Mahinur, and experimentally demonstrate that our methods produce efficiency improvements of several orders of magnitude.

Background and Related Research

When a planning agent does not have complete knowledge of the environment in which its plans will be executed, it may have to create a *conditional plan*, which includes *observation steps* to ascertain the unknown conditions. Using an example from (Dearden & Boutilier 1997), imagine a robot whose goal is to deliver coffee without getting wet; imagine further that the robot does not know whether it is raining outside. A reasonable plan is to go to the window, observe whether it is dry, and if so, go to the cafe. Conditional planning systems (Warren 1976; Peot & Smith 1992; Etzioni *et al.* 1992; Goldman & Boddy 1994a; Pryor & Collins 1996) generate plans that have *branching actions*, i.e., actions with multiple possible outcomes.[1] When a branching action is initially inserted into a plan, one of its outcomes (the *desired outcome*) will be linked to a later step on the path to the goal, while the other(s) (the *undesired outcomes*) will not. We will also refer to an unlinked outcome as a *dangling edge*. In the coffee example, the knowledge that it is dry outside is the desired outcome, while knowledge that it is raining outside is the undesired outcome. The plan is guaranteed to succeed if the desired outcomes of all its observation actions occur; there is no such guarantee otherwise.

Intuitively, one way to improve such a plan is to figure out what to do if some step has an undesired outcome. We will call this a *corrective repair*, since it involves figuring out actions that can be taken to correct the situation that results after the undesired outcome occurs. For the above example, one corrective repair might be to pick up an umbrella if it is raining. In practice, conditional planners implement corrective repairs by duplicating the goal state, and attempting to find a plan that will achieve the (duplicated) goal state without relying on the assumption that the branching actions in the original plan have their desired outcomes.

A different approach is taken in probabilistic planners. Where conditional planners assume that agents have no information about the probability of alternative action outcomes but will be able to observe their environments during plan execution, probabilistic planners such as Buridan (Kushmerick, Hanks, & Weld 1995) make just the opposite assumption. They assume that planning agents have knowledge of the probabilities that their actions will have particular outcomes but that they will be unable to observe their environment. Typically, probabilistic planners model actions with a finite set of tuples $< t_i, p_{i,j}, e_{i,j} >$, where the t_i are a set of exhaustive and mutually exclusive *triggers*, and $p_{i,j}$ represents the probability that the action will have effect $e_{i,j}$ if t_i is true at the time of the action's execution. The triggers serve the role of

[1] To simplify presentation, we will focus here on actions with two possible outcomes; generalization to a larger number of outcomes is straightforward.

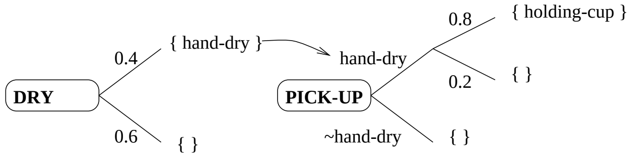

Figure 1: Plan for picking up a part.

preconditions in standard causal-link planners. Suppose that the robot's hand might get wet while closing the umbrella before entering the shop, and the coffee cup might slip if its hand is wet. In the example plan fragment shown in Fig. 1, the `PICK-UP` step has been inserted to achieve the goal of holding the cup. The trigger for `holding-cup` is `hand-dry`, and a `DRY` step has been inserted to probabilistically make that true.

As can be seen, this plan is not guaranteed to succeed. If the hand is not dry, the step will not achieve the desired outcome of holding the cup. To help prevent this undesired outcome, a planner may increase the probability that the hand is dry. One way to do this would be to add a second `DRY` step prior to the `PICK-UP`. We can call this a *preventive repair*, since it involves adding actions that help prevent the undesired outcome.

It is only natural to combine the ideas of conditional and probabilistic planning. The first combined conditional, probabilistic planning system was C-Buridan (Draper, Hanks, & Weld 1994). Interestingly, while C-Buridan uses preventive repair to increase the probability of success, it does not use corrective repair to generate conditional branches. Its branches are formed in a somewhat indirect fashion: in performing a preventive repair, it may add to the plan a step that conflicts with some other step already in the plan. To resolve this conflict, C-Buridan will split the plan into two branches, putting the conflicting steps on different branches. In effect, C-Buridan identifies a new branch only after it has been formed. Generating plans in this way has been shown to be very inefficient, involving a rapid explosion of the search space as branches are discovered haphazardly (Onder & Pollack 1997).

A more recent system that combines conditional and probabilistic planning is Weaver (Blythe & Veloso 1997) Weaver was built on top of a bidirectional planner (Prodigy 4.0), and therefore uses a different set of basic plan generation operations than those described in this paper. However, as in our approach, Weaver first reasons about which actions to choose in order to most quickly improve the likelihood of success (Blythe 1995) and then uses both preventive and corrective repair. Unlike most of the other planners, it also includes explicit mechanisms for dealing with external events.

The Plinth conditional-planning system was also expanded to perform probabilistic reasoning (Goldman & Boddy 1994b). The focus of the Plinth project was on using a belief network to reason about correlated probabilities in the plan.

A different approach to planning under uncertainty is called conformant planning, and involves generating plans that achieve the goals in all the possible cases without using observation actions(Goldman & Boddy 1996). The work on Markov Decision Process (MDP) based planners focuses on finding "policies," which are functions from states to actions. To do this in an efficient way, MDP-based planners rely on dynamic programming and abstraction techniques(Dearden & Boutilier 1997). The DRIPS system (Haddawy, Doan, & Goodwin 1995) interleaves plan expansion and decision theoretic assessment but uses a previously formed plan tree (HTN-style) rather than generating plans using operator descriptions. Recent work by Weld et al. extends Graphplan to handle uncertainty (1998). The Just-In-Case scheduling algorithm (Drummond, Bresina, & Swanson 1994) involves creating an initial schedule and building contingent schedules for the points that are most likely to fail.

Algorithm

Our algorithm (Fig. 2) rests on the observation that conditional, probabilistic planning involves repairing plan flaws (closing an open precondition or resolving a threat) and repairing dangling edges (corrective repair or preventive repair). The input is a set of initial conditions, a set of goal conditions, and a time limit T. The output is a set of plans. The algorithm is a plan-space search, where, as usual, the nodes in the search space represent partial plans. We assume that actions are encoded using the probabilistic action representation described in the previous section.

Normal flaws—threats and open conditions—are repaired in the usual way. To achieve an open condition c, the planner will find an action that includes a branch $< t_i, p_{i,j}, e_{i,j} >$, such that one of the elements of $e_{i,j}$ unifies with c. The relevant trigger t_i will then become a new open condition. Note that a condition c remains "open" only so long as it has no incoming causal link; once an action α has been inserted to (probabilistically) produce c, it is no longer open, even if α has only a small chance of actually achieving c.[2] A threat is resolved by step ordering (*demote, promote*), committing to desired outcomes (*confront*), or separating the steps into branches (*constrain-to-branch*). For "dangling-edge flaws", we assume that preventive

[2]A sensible heuristic is to select actions where the relevant $p_{i,j}$ is as high as possible.

PLAN (init, goal, T)
```
plans ← { make-init-plan ( init, goal ) }
while plan-time < T and plans is not empty do
 CHOOSE (and remove) a plan P from plans
 SELECT a flaw f from P.
 add all refinements of P to plans:
   plans ← plans ∪ new-step(P, f) ∪
         step-reuse(P, f)
                   if f is an open condition,
   plans ← plans ∪ demote(P, f) ∪ promote(P, f) ∪
         confront(P, f) ∪ constrain-to-branch(P, f)
                   if f is a threat.
   plans ← plans ∪ corrective-repair(P, f) ∪
         preventive-repair (P, f)
                   if f is a dangling-edge.
return (plans)
```

preventive-repair (plan, f)
```
open-conditions-of-plan ← open-conditions-of-plan ∪
 triggers for the desired outcomes of the
 action in f.
return (plan)
```

corrective-repair (plan, f)
```
top-level-goal-nodes-of-plan ← top-level-goal-nodes-of-plan
 ∪ new-top-level-goal-node labeled not to depend
 on the desired outcomes of the action in f.
return (plan)
```

Figure 2: Conditional probabilistic planning algorithm.

repair is achieved by reintroducing the triggers for desired effects into the set of open conditions, as done in Buridan; we assume corrective repair is achieved by adding new, labeled copies of the goal node as in CNLP. Corrective repairs form new branches in the plan that indicate alternative responses to different observational results. Preventive repairs do not introduce new branches.

Consistent with the prior literature, we use SELECT to denote a non-deterministic choice that is *not* a backtrack point, and CHOOSE for a backtrack point. As usual, node selection, but not flaw selection, is subject to backtracking.

It is important to note that in prior algorithms such as CNLP and C-Buridan, observation actions do not differentiate between the conditions that are true in the world (*state conditions*) and the conditions that represent the agent's state of knowledge (*information conditions*). In CNLP, the outcomes of an observation action are state conditions, thus observation actions are inserted through standard backchaining. In C-Buridan, the outcomes of actions are information conditions. C-Buridan has no concept of corrective repair; instead it inserts observation actions only during threat resolution, when conflicting actions are constrained to be in different branches. During this process, C-Buridan will

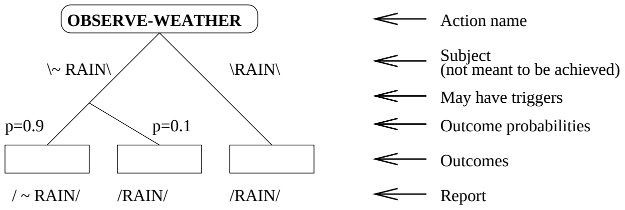

Figure 3: Observing whether it is raining.

consider every possible observation action. It is thus complete in the sense that it will find a relevant observation action whenever one exists, even if the correlation between the observation and the condition in question has not been made explicit. However, C-Buridan's indirect method of inserting observation actions is very inefficient: it has no notion of the source and potential impact of any plan failure, and thus cannot prioritize the failures it chooses to work on (Onder & Pollack 1997).

For the sake of practicality, we have taken a middle road. We require that the connection be made explicit between an observation action and any information and state conditions it affects. Fig. 3 illustrates the act of directly observing whether it is raining outside. The observation may be inaccurate: with 0.10 probability, it will provide a false negative. The connection between the belief that it is not raining (the "report") and the fact of the matter of rain (the "subject") is explicit in the representation.

Consequently, our algorithm does need to insert observation actions by backchaining or for threat resolution. Instead, we can directly reason about which step S will have the greatest impact if it fails—i.e., does not achieve desired outcome c. Corrective repair can then be performed, directly inserting after S an observation action that reports on c, along with a *subject link* from S to the new observation action. The trade-off is that the algorithm is less complete than C-Buridan, because it will not discover observation actions whose connection to some condition are not explicitly encoded. Because the in-principle complete algorithms are too inefficient to be in-practice complete, we are willing to make this trade-off, and rely on the operator designer to encode explicitly the connections between information conditions and state conditions.

We implemented this algorithm in a planning system called Mahinur. In refining a plan, Mahinur first repairs the normal flaws until the plan is *complete*, i.e., has no open conditions or threats. It then selects a dangling edge and then works only on normal flaws until the plan is once again complete. This strategy reduces the amount of bookkeeping required to keep track of nested contexts when multiple corrective repairs are being performed. It also allows Mahinur to readily produce intermediate solutions throughout the planning process, because complete plans are potential solutions to the planning problem.

We assume that the top-level goals have additive scalar values assigned to them. Thus, the *expected value of a plan* is the sum of the products of the final probability of each top-level goal and its scalar value. We approximate the final probability of any goal by a process of simulation, in which we start with an initial state distribution, and simulate the execution of each step in the plan, updating the state distribution accordingly. We refer to the state distribution after the execution of step i as sd_i. Because the focus of our work is on search control, we finesse the issue of efficient plan assessment (i.e., calculation of the expected value) and use a random total ordering of the steps.

Efficient Corrective Repair

While the algorithm above captures the ideas inherent in the prior work on both conditional and the probabilistic planning, it also inherits a major problem: if applied without strong heuristics, it can be extremely inefficient. In particular, the time required to generate a plan with two branches can be exponentially greater than the sum of the times required to generate two separate plans, each identical to one of the branches. We illustrate this with the running example. If we ignore the possibility of rain, a simple solution is a plan with four steps, `GO-CAFE; BUY-COFFEE; GO-OFFICE; DELIVER-COFFEE`, which can be generated by Mahinur in 0.24 CPU seconds using 40 plan nodes. When we re-introduce the possibility of rain, the solution has two branches because if is is raining, the robot needs to get an umbrella: `SEE-IF-RAINING; if raining (GET-UMBRELLA; GO-CAFE; BUY-COFFEE; GO-OFFICE; DELIVER-COFFEE); if not raining (GO-CAFE; BUY-COFFEE; GO-OFFICE; DELIVER-COFFEE)`. The two branches are similar to one another, but it takes Mahinur 27.66 CPU seconds and 6902 plan nodes to generate the branching plan.[3]

The problem results from the backwards-chaining approach taken by Mahinur and other planning systems. When a new step is inserted into the plan, it is not part of any branch; it is put into a branch (by the *constrain-to-branch* procedure in Mahinur) only after a threat is detected between two steps that should

[3]Unfortunately, benchmark problems and implemented systems for comparison of probabilistic conditional planners are not available. However, the examples given in the literature suggest that other conditional planners suffer from similar exponential explosion of the search space.

Problem	Run time (sec.)		Plans created	
	with	without	with	without
coffee	1.35	27.66	105	6902
coffee,cream	1.81	687.13	178	57912
coffee,cream,sugar	2.73	—	398	>350000

Figure 4: Effects of the threat resolution heuristic.

be in separate branches. However, in addition to constraining, the planner needs to consider several other methods of threat resolution, resulting in exponential explosion of the search space. The intuitive solution is to prefer constraining a new step to be in a new branch when a conditional branch is being formed, i.e., corrective repairs are being performed. This can be implemented easily as a heuristic in the framework of our algorithm: while generating a new branch following observe step `I`, if a newly inserted step threatens a step in the first branch, prefer to resolve the threat by placing the new step into the new branch of `I` if it is consistent to do so.

We illustrate the proportionality of our heuristic's effect on variations of the `BUY COFFEE` problem. The first problem is the basic problem, including the possibility of rain; there are five steps in the new branch. In the second problem, we added a new step (`GET-CREAM`) to the problem, increasing the number of the steps in the new branch to 6. In the third problem, we also added the `GET-SUGAR` step. In Fig. 4, we tabulate the run time and the number of plans created while generating conditional plans with and without the threat resolution heuristic. (We terminated the experiment marked with "—" after 24 hours and 350000 plan nodes.) As expected, the search space is reduced significantly when the heuristic is used to control the search, and the reduction is proportional to the number of steps in the new branch. This heuristic proved to be very effective enabling us to generate plans with tens of steps and conditional branches in just a few seconds.

Selecting Contingencies

A plan is composed of many steps that establish conditions to support the goals and subgoals, but repairs for the failure of conditions are usually not expected to have equal contributions to the overall success of the plan. With the notable exception of (Feldman & Sproul 1977), the decision theoretic prioritization of repair points has not been a focus of recent systems. To identify the best repair points, we focus on two facts: first, contingencies in a plan may have unequal probability of occurrence; second, a plan may have multiple goals, each of which has some associated value. Let us leave the probability of failure aside for a moment, and consider the robot in the the previous examples and a plan involving two goals: mailing a document at the post office and delivering coffee to the user. The value of achieving the former goal may be significantly higher than the latter. Consequently, the conditions that support only the former goal (e.g., having envelopes) contribute more to the overall success of the plan than the conditions that support only the latter (e.g., having sugar). Conditions that support both goals (e.g., keeping the robot dry) will have the greatest importance. This suggests that it may be most important to have a contingency plan to handle the possibility of rain; almost as important to have a contingency plan in case there are no envelopes in the office; and less important to have a contingency plan in case there is no sugar. While performing this reasoning, we can fold the probability of failure back in as a weighing factor.

Of course, in reality the importance of having contingency plans will also depend on the likely difficulty of replanning "on-the-fly" for particular contingencies; it may be worth reasoning in advance about contingencies that are difficult to plan for. Another factor influencing the choice of a contingency is the difficulty of executing a contingent plan if not considered in advance. These types of information concern the plan that has not yet been generated, which suggests that they might have to be coded as part of the domain information, based on past experience.

Even if this type of domain information is not available, we can use the upper bound of the expected value of repairing a failure point as a good estimate for selecting contingencies. Suppose that a step S_i is used to establish a condition c, and the probability of that c will be false right after S_i is executed is $p > 0$. Then, the best the planner can do is to add a new branch that will make the top-level goals true even when c is not true. What is the value of adding such a branch? In computing this, we need to consider only the probability that immediately after executing S_i, both c and the top-level goals will be false: if a top-level goal is true anyway after executing S_i, then there is no benefit to establishing it. We also need to factor in the probability that S_i will be executed in the first place. (If it is downstream from another branch point, it may never be executed.) The upper bound on the expected value of performing a corrective repair on S_i in the case in which desired outcome c may fail is defined as follows:

$$EVCR(S_i, c) = \sum_{g \in G} P[\{\tilde{}c, \tilde{}g\} \cup S_{Ci} | sd_0, < S_1, \ldots, S_i >] \times V(G),$$

where $P[\{x_1, \ldots, x_n\} | sd_0, < S_1, \ldots, S_i >]$ denotes the probability that $\{x_1, \ldots, x_n\}$ all hold in the state distribution resulting from the execution of the steps through S_i starting with the initial state distribution sd_0. S_{Ci} is the *context* of the step, i.e., the conditions under which the step will be executed.[4] That is, for each top-level goal g in G, we compute the probability that immediately after performing step S_i, c and g will both fail to hold while all the effects of S_i except c hold. We then weight the value of G ($V(G)$) by this probability, and finally sum all the weighted values. As noted, this is an upper bound; the actual value of corrective repair of s_i might be less if the repair only probabilistically establishes G, or if there are other steps in the original plan that achieve G without using c in sd_i. Our strategy for selecting contingencies is to use the formula above to compute $EVCR$ for each failure point and then to select the one with the highest value.

In order to illustrate the importance of this process, we have designed a synthetic domain with several top-level goals all of which have unit value (Fig. 5). We then ran experiments in which we started with an initial plan where each goal is achieved by a single step that has no preconditions, and achieves the goal with some probability (Fig 5a). In addition to the operators in the initial plan, we designed a set of alternative operators each of which achieves a goal with certainty. (Imagine that the probabilistic operator in the initial plan is cheaper to execute, and it is preferable to try it first and use the alternative only if it fails). In order to establish a baseline case, we designed the alternative operators to have the minimal planning effort, i.e., performing a corrective repair involves forming a branch with one step to establish the goal. This also made the corrective repair effort to be uniform for each failure point: each can be repaired by generating a one-step plan. We implemented our strategy for selecting contingencies and ran several experiments by increasing the user-specified expected value threshold. In one condition—ordered selection—we selected the contingency with the highest expected value for corrective repairs; and in another condition—random selection—we selected a contingency randomly (ordered selection always selects steps A1, B1 ...I1). In Fig. 6, we plot the time required to meet the threshold with and without ordering contingencies. As expected, by ordering contingencies, the planner can produce better plans in shorter time. Note that, the two lines converge to the same point because once all the failures are repaired, the total expected value of the plan is the same and the total cost of repairing all the failures is the same.

[4] Actually, the probability of S_{Ci} should be computed for the state that obtains just prior to the execution of S_i. However, for simplicity in the formula we compute it in the state that obtains after S_i is executed; this does not affect the result because a step cannot change its own context.

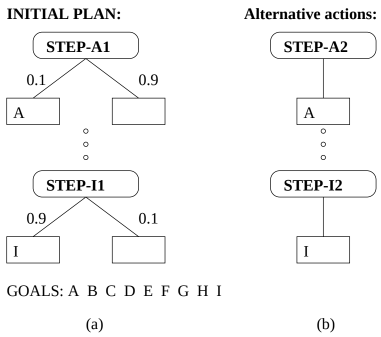

Figure 5: A synthetic domain for experiments.

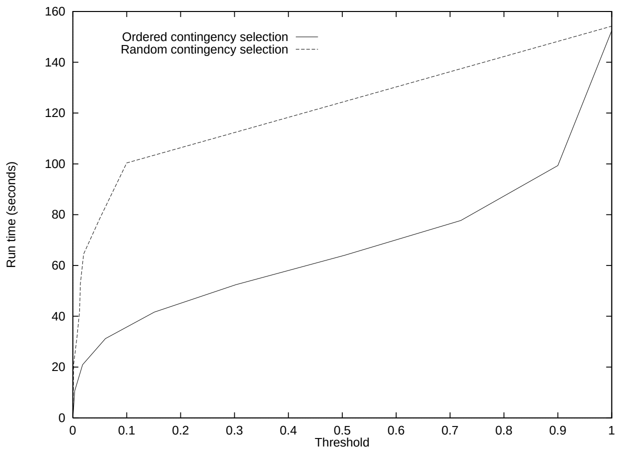

Figure 6: Time required to generate a plan that meets the threshold.

A similar strategy can be used for estimating the expected value of performing preventive repairs, but we omit discussion due to space limitations.

Generating Plans with Joined Branches

The efficiency of performing corrective repairs can be further improved by sharing the final parts of two branches if they are the same. CNLP-style conditional planners cannot generate plans with joined branches because they duplicate the top-level goals and regenerate every step even if they are the same as the existing branch. However, branches can be joined by nondeterministically choosing and duplicating a subgoal rather than a top-level goal. Suppose that the delivery robot has a detailed plan to go back to the office after picking up the coffee, and is generating contingency plans for the possible failures regarding coffee pick up. In such a situation, it might be more efficient to focus on the subgoal of getting coffee rather than revising the whole plan.

We have implemented this method in Mahinur in the

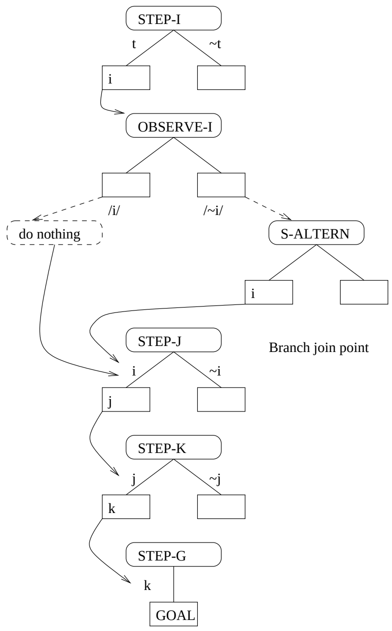

Figure 7: Corrective repairs with branch joining.

following way: consider the plan in Fig. 7 and suppose that `STEP-I` can fail to establish i for `STEP-J` and this failure point has been selected for corrective repairs. Then, rather than duplicating the top level goal, the planner duplicates just `STEP-J`'s triggers—i—and tries to find a conditional branch that establishes i without using support from `STEP-I`. The remainder of the plan (`STEP-J` and `STEP-K`) remains the same, and does not have to be regenerated. If the planner fails to find a plan for i, it backtracks and tries to duplicate the triggers of the next step that is connected by a causal link to the step it has just tried—the next step in the causal link path is `STEP-K` in this example. Imagine that the planner first tries to find alternative ways of getting coffee and then tries another beverage if this fails. Backtracking stops when a top-level goal needs to be duplicated. Note that our work focuses on generating plans with joined branches rather than merging already formed branches.

The step whose triggers are duplicated is called a *branch join point* (e.g., `STEP-J` in Fig. 7). No steps are necessary if `STEP-I` succeeds, and alternative step (`S-ALTERN`) will be executed if `STEP-I` fails. The remainder of the plan is shared by the two branches.

We conducted a set of experiments to show the possible benefits of branch joining. In these experiments, we used a set of coffee domain problems analogous to the ship-reject problem in C-Buridan (Draper, Hanks, & Weld 1994). In these problems, the robot asks whether decaffeinated coffee is available and gets it. If not available, it gets regular coffee. Both

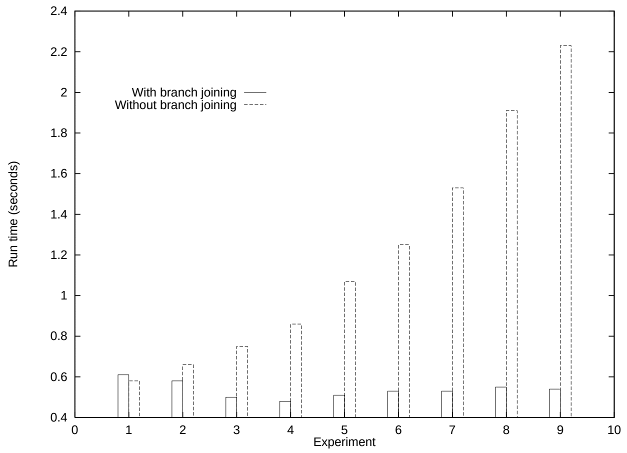

Figure 8: The CPU time required with and without branch joining to solve planning problems of increasing complexity.

have the same price, so the steps to pay for the coffee and to go back to the office can be shared. Without branch joining, the solution conditional plan is: `ASK; if available (GET-DECAF; PAY; GO-OFFICE; DELIVER-COFFEE); if not available (GET-REGULAR; PAY; GO-OFFICE; DELIVER-COFFEE)`. If branch joining is performed during corrective repairs, the last three steps of each branch can be shared.

If branch joining is used, the planner saves some of the effort of generating the sequence of steps after the branch join point. In order to demonstrate this, we designed a set of 9 problems based on the above problem. We made the plan generation process harder by putting more alternative steps into the domain description: In the first problem, there are no alternatives to the final three steps; in the second problem, each can be performed in two ways; and in the ninth problem, each can be performed in nine ways. For each problem, we plotted the CPU time required to generate the conditional branch with and without branch joining in Fig. 8. As expected, the planning effort does not increase when branch joining is used because the plan after the branch join point is reused while forming the new branch. When branch joining is not used, it takes the planner longer to generate the same plan because the part after the branch join point needs to be generated from scratch.

On the other hand, the planner needs to backtrack if it cannot find a plan with joined branches. In our implementation, when branch joining is enabled, the planner first duplicates the triggers of the first action that is supported by the condition which may fail. If no plan is found, it tries the next step in the path of causal links and continues until a top-level goal is reached. (We do not consider re-opening every set of possible subgoals because once the top-level goals are re-opened, an entirely new plan can be found). Obviously, if the planner spends too much time trying to find a plan with joined branches when none exists, its performance will be worse than directly duplicating the top-level goals. As a result, domain-dependent tuning may be required to determine whether branch joining will be attempted and to select the step to be used as a branch join point. Nonetheless, the results are promising and we are optimistic about the effectiveness of our method because the savings obtained by branch joining can be significant and a strategy for branch joining can be determined by compiling typical problem instances in a domain. Branch joining is useful because it lets the planner focus on the steps that are in the vicinity of the possible failure rather than the top-level goals.

Conclusion

In real-world environments, planners must deal with the fact that actions do not always have certain outcomes, and that the state of the world will not always be completely known. Good plans can nonetheless be formed if the agent has knowledge of the probabilities of action outcomes and/or can observe the world. Intuitively, if an agent does not know what the world will be like at some point in its plan, there are two things it can do: (i) it can take steps to increase the likelihood that the world will be a certain way, and (ii) it can plan to observe the world, and then take corrective action if things are not the way the should be. These basic ideas have been included, in different ways, in the prior literature on conditional and probabilistic planning. The focus of this paper has been to synthesize this prior work in a unifying algorithm that cleanly separates the control process from the plan refinement process. Using our framework, contingencies can be handled selectively and heuristics that depend on the type of repair being performed can be used. This control is an important condition for applying conditional probabilistic planning to real world problems. We have obtained promising early results in a realistic domain(Desimone & Agosta 1994), and we will make the Mahinur system and the domain encoding publicly available.

Acknowledgments

This work has been supported by a scholarship from the Scientific and Technical Research Council of Turkey, by the Air Force Office of Scientific Research (F49620-98-1-0436), and by the National Science Foundation (IRI-9619579). We thank the anonymous reviewers for their comments.

References

Blythe, J., and Veloso, M. 1997. Analogical replay for efficient conditional planning. In *Proc. 15th Nat. Conf. on AI*, 668–673.

Blythe, J. 1995. The footprint principle for heuristics for probabilistic planners. In *Proc. European Workshop on Planning*.

Dearden, R., and Boutilier, C. 1997. Abstraction and approximate decision theoretic planning. *Artificial Intelligence* 89(1):219–283.

Desimone, R. V., and Agosta, J. M. 1994. Spill response system configuration study—final report. Technical Report ITAD-4368-FR-94-236, SRI International.

Draper, D.; Hanks, S.; and Weld, D. 1994. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Int. Conf. on AI Planning Systems*, 31–36.

Drummond, M.; Bresina, J.; and Swanson, K. 1994. Just-in-case scheduling. In *Proc. 12th Nat. Conf. on AI*.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Repr. and Reasoning*, 115–125.

Feldman, J. A., and Sproul, R. F. 1977. Decision theory and AI II: The hungry monkey. *Cognitive Science* 1:158–192.

Goldman, R. P., and Boddy, M. S. 1994a. Conditional linear planning. In *Proc. 2nd Int. Conf. on AI Planning Systems*, 80–85.

Goldman, R. P., and Boddy, M. S. 1994b. Epsilon-safe planning. In *Proc. 10th Conf. on Uncertainty in AI*, 253–261.

Goldman, R. P., and Boddy, M. S. 1996. Expressive planning and explicit knowledge. In *Proc. 3rd Int. Conf. on AI Planning Systems*, 110–117.

Haddawy, P.; Doan, A.; and Goodwin, R. 1995. Efficient decision-theoretic planning: Techniques and empirical analysis. In *Proc. 11th Conf. on Uncertainty in AI*.

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

Onder, N., and Pollack, M. E. 1997. Contingency selection in plan generation. In *Proc. European Conf. on Planning*, 364–376.

Peot, M. A., and Smith, D. E. 1992. Conditional nonlinear planning. In *Proc. 1st Int. Conf. on AI Planning Systems*, 189–197.

Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision based approach. *Journal of AI Research* 4:287–339.

Warren, D. H. 1976. Generating conditional plans and programs. In *Proc. AISB Summer Conference*, 344–354.

Weld, D. S.; Anderson, C. R.; and Smith, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In *Proc. 16th Nat. Conf. on AI*, 897–904.

Weld, D. S. 1994. An introduction to least commitment planning. *AI Magazine* 15(4):27–61.