Contingency Planning for Planetary Rovers

Richard Dearden[*], **Nicolas Meuleau**[†], **Sailesh Ramakrishnan**[‡], **David Smith and Rich Washington**[§]
NASA Ames Research Center
Mail stop 269–2
Moffet Field, CA 94035–1000, USA
{dearden, nmeuleau, sailesh,de2smith, richw}@email.arc.nasa.gov

Abstract

There has been considerable work in AI on planning under uncertainty. But this work generally assumes an extremely simple model of action that does not consider continuous time and resources. These assumptions are not reasonable for a Mars rover, which must cope with uncertainty about the duration of tasks, the power required, the data storage necessary, along with its position and orientation.

In this paper, we outline an approach to generating contingency plans when the sources of uncertainty involve continuous quantities such as time and resources. The approach involves first constructing a "seed" plan, and then incrementally adding contingent branches to this plan in order to improve utility. The challenge is to figure out the best places to insert contingency branches. This requires an estimate of how much utility could be gained by building a contingent branch at any given place in the seed plan. Computing this utility exactly is intractable, but we outline an approximation method that back propagates utility distributions through a graph structure similar to that of a plan graph.

1 Introduction

For a Mars rover, daily operation is rife with uncertainty. There is inherent uncertainty about the duration of tasks, the power required, the data storage necessary, position and orientation, and environmental factors such as soil characteristics, dust on the solar panels, ambient temperature, etc. For example, in driving from one location to another, the amount of time required depends on wheel slippage and sinkage, which varies depending on slope, terrain roughness, and soil characteristics. All of these factors also influence the amount of power that is consumed. The amount

[*] Research Institute for Advanced Computer Science (RIACS)
[†] QSS Group Inc.
[‡] QSS Group Inc.
[§] RIACS

of energy collected by the solar panels during a traverse depends on the length of the traverse, but also on the angle of the solar panels. This is dictated by the slope and roughness of the terrain.

Since rover operations are often highly constrained by time and power constraints, plans that do not take this uncertainty into account often fail miserably. In fact, it has been estimated that the Mars Pathfinder rover spent a substantial amount of its life doing nothing because of either plan failure or conservative action sequences constructed to avoid any possibility of plan failure. One way to attack this problem is to do on-board replanning when failures occur. While this capability is certainly desirable, there are several difficulties with exclusive reliance on this approach:

- Rovers have severely limited computational resources due to power limitations and radiation hardening requirements. As a result, it is not always feasible to do timely or significant onboard replanning.
- Many actions are potentially risky and require pre-approval by mission operations personnel. Because of the cost and difficulty of communication, the rover receives infrequent command uplinks (typically one per day). As a result, each daily plan must be constructed and checked for safety well in advance.
- Some contingencies require anticipation; e.g., switching to a backup system may require that the backup system be warmed up in advance. For time critical operations there is insufficient time to perform these setup operations once the contingency has occurred, no matter how fast the planning can be done.

For these reasons, it is sometimes necessary to plan in advance for potential contingencies; that is, anticipate unexpected outcomes and events and plan for them in advance. In this paper we will be concerned with ground-based contingency planning for rovers. More precisely, the problem is to produce a (concurrent) plan with maximal expected utility, given the following domain information:

- A set of possible goals that may be achievable, each of which has a value or reward associated with it.
- A set of initial conditions, which may involve uncertainty about continuous quantities like temperature, energy available, solar flux, and position. This uncertainty is characterized by probability distributions over the possible values.
- A set of possible actions, each of which is characterized by:
  - a set of conditions that must be true before the action can be performed. (These may include metric temporal constraints and constraints on resource availability.)
  - an uncertain duration characterized by a probability distribution.
  - a set of certain and uncertain effects that describe the world following the action. Uncertain effects on continuous variables are characterized by probability distributions.

Contingency planning is already known to be quite hard both in theory [5] and in practice. However, there are some characteristics of this domain, which make this planning problem different and even more difficult:

Time - actions take differing amounts of time and concurrency is often necessary.

Continuous outcomes - most of the uncertainty is associated with continuous quantities like time and power. In other words, actions do not have a small number of discrete outcomes.

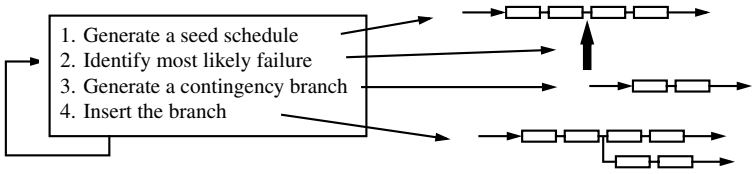

Figure 1: The JIC approach.

Problem size - a typical daily plan for a rover will involve on the order of a hundred actions.

As a result of these characteristics, it is not clear how to apply previous approaches to planning under uncertainty to this problem. In this paper, we outline a much different approach to this problem. At the top level, the approach involves 1) constructing a "seed" plan, and 2) incrementally adding contingent branches to this plan in order to improve utility. The challenge is to figure out the best places to insert contingency branches. In general, this requires an estimate of how much utility could be gained by building a contingent branch at any given place in the seed plan. Computing this utility exactly is intractable, but we outline an approximation method that involves back propagating utility distributions through a graph structure similar to that of a plan graph. In Section 2 we discuss *Just-in-Case Planning*, our incremental approach to contingency planning based on the *Just-in-Case Scheduling* work of Drummond, et al[3].We also argue that for planning, probability of failure is not a good heuristic for choosing branch points. In Section 3 we describe our plan graph method for estimating branch utility curves. In Section 4 we describe how this information is used in order to 1) choose branch points, 2) guide the planner in selecting goal sets, and 3) choose the correct branch condition.

2 Just-In-Case Planning

In the classical approach to contingency planning, each time an action with uncertain outcomes is added to a plan, the planner attempts to establish the goals for each different outcome of the action. Unless there are only a few discrete sources of uncertainty in a domain, this approach is completely impractical. For more complex domains, it is critical that the planner focus on those contingencies that will make a large difference in the overall value of the plan. To do this, we build upon the Just-In-Case(JIC) scheduling technique[3], that was initially developed for contingency scheduling of automated observatories. The basic idea in the JIC approach is to take a seed schedule, look for the place where it is most likely to fail, and augment the schedule with a contingent branch at that point. The process is repeated until the resulting contingent schedule is sufficiently robust, or until available time is exhausted. This process is illustrated in Figure 1.

Conceptually, it seems straightforward to apply the JIC approach to planning problems. Using a conventional planner, we first generate a seed plan assuming the expected behavior of each activity; in other words, we reason as if every action uses the expected amount of time and resources. This is the same approach taken in JIC scheduling. As with JIC scheduling, we then choose a place to insert a contingency branch. Once again, using a conventional planner, we generate a plan for the contingency branch and add it to the existing plan. [1]

[1]Just as with JIC scheduling, this process is not guaranteed to converge to an optimal contingent plan. However,

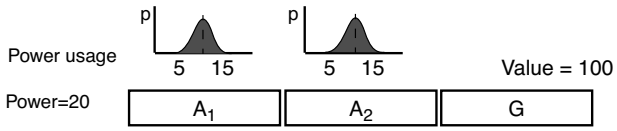

Figure 2: Example showing that the place where the plan is most likely to fail may not be the best branch point.

2.1 The JIC Branch Heuristic

For JIC planning, the tricky part is deciding where to insert contingency branches, and what the branch conditions should be. In Drummond et al.'s original implementation for automatic telescope scheduling, branches are added at the points *with the greatest probability of failure*. Given the distributions for time and resource usage this is relatively easy to calculate by statistical simulation of the plan. Unfortunately, the points most likely to fail are not necessarily good points for contingent branches. Consider the example in Figure 2 where we have a seed plan with two actions, A_1 and A_2, leading to a goal G that has positive value. Initially we have 20 units of some resource (say power) and each of the actions consumes somewhere between 5 and 15 units of the resource. Clearly, this plan is most likely to fail after (or during) action A_2. However, if the plan fails after (or during) action A_2, there will not be any resources left. If all the alternative activities require some of this resource, then there is clearly no point in putting a contingent branch after A_2.

Fundamentally, the problem is that in order to select the best place to insert a branch, we need to know whether or not it is possible to accomplish anything useful at the points under consideration. More precisely, we need to know how much utility could be gained by inserting a branch at each given point. In order to do this, we need to know the *value function* of the mainline plan and of each possible branch. The value function gives the expected future reward (utility) at each step of a plan, as a function of the resource levels.

Computing the value function for a completed plan (such as the seed plan) is relatively straightforward. It may be done analytically if the resource consumptions for activities are simple distributions. However, more typically, Monte Carlo simulation is required [1, 6]. Similarly, it is easy to get an estimate of the probability distribution over resources at each step of a plan. A crucial piece of information is then *the value function of the best branch plan that can be added at each point in the existing plan*. Given this information, we can easily determine the optimal branch point in the plan. We just have to compare the relative gain in utility obtained by considering the best possible branch plan at each point and pick the branch point where this gain is maximal. Unfortunately, there is no easy way to calculate the value function for the best possible branch plan at a given point. It requires actually doing the planning for the branch. Instead we must approximate this value function. In the next section, we present a procedure designed to estimate the value function of the best possible branch plan that could be generated at each point, without actually doing the planning.

JIC will always monotonically improve a plan until a local optimum is reached.

3 Estimation of Branch Utility

The main procedure of our algorithm computes an estimate of the value function of the best possible branch plan, at each point of the mainline plan. It is based on a representation of the planning problem as a graph identical to the *plan graph* of Blum and Furst's Graphplan [2, 4]. Graphplan is a classical planning algorithm that first performs a reachability analysis by constructing a plan graph, and then performs goal regression within this graph to find a plan. Our approach retains only the first of these stages, the plan graph construction. We then perform backpropagation of utility tables in the graph to produce estimates of utility functions (instead of plans). This section provides an outline of this mechanism.

3.1 The Plan Graph

The plan graph is a sequential graph that alternates propositional (fluent) levels and action levels. Each propositional level contains the set of propositions that can be made true at that level, and a set of mutual exclusion(mutex) constraints between pairs of these propositions. A mutex between two fluents indicates that these propositions cannot both be true at the same time at this level of the graph.[2] The first propositional level contains all the fluents that are true in the initial state of the problem (initial conditions). The action levels contain all the actions that can be applied given the previous propositional level. Each action has an arc from each fluent that it consumes and an arc to each fluent it produces.

Figure 3 shows a part of the plan graph obtained in a simple example where the only continuous variable is power. In this problem, the mainline plan(shown in bold) consists of two actions: A which takes the fluent p as precondition and produces q and r, and B which has q as precondition and g as effect. The fluent g represents a goal that provides a reward (utility) of 5. For each action, the expected consumption is 10 Ah, and it can be started only if the current level of resource is atleast 15 Ah. Three other actions, C, D, and E, are available in the domain, but they are not included in the mainline plan. The fluent g' represents a secondary goal with utility 1. Finally, both p and s are true and all the other fluents are false in the initial conditions. There are two points of the mainline plan that are candidate branch points: at the beginning of the plan, and between A and B. The latter is characterized by the following set of propositions: p, q, r and s (all other fluents being false). Our goal is to estimate the best utility gain we can get by branching at these points.

3.2 Utility Table Backpropagation

The basic principle of our algorithm is to backpropagate utility distribution tables in the plan graph. Each table is attached to a single (action or proposition) node and contains:

- a *condition*, that is, a list of fluents such that the table is valid if all the fluents are true.
- a piecewise constant function giving utility as a function of resource level. It represents an estimate of the expected reward we can get by performing this action, or by having this fluent true, as a function of current resource levels, if all fluents in the table condition are true.

[2]Note that the reciprocal is not true: since Graphplan takes into account only binary exclusion constraints, two fluents that are not mutex in the graph may in fact be unreachable simultaneously.

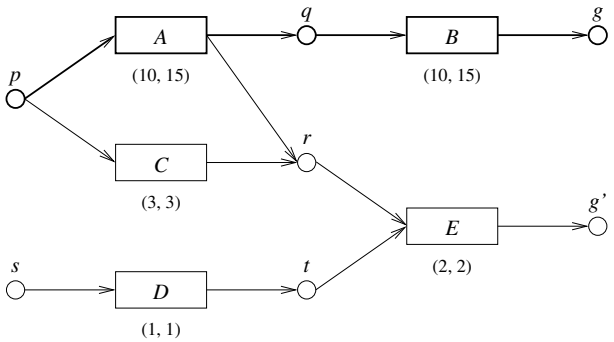

Figure 3: An example of plan graph (partial). The two numbers below each action represent, first, its expected consumption, and second, the minimum power required to be allowed to start this action.

The process is initialized by creating utility tables for the goals. In our example, we start with a table for g with an empty condition, and an expected return of 5 for positive resource levels (and 0 otherwise), indicating that we obtain a reward of 5 if we can get to g with some power remaining. Similarly, g' has a table with an empty condition and reward 1.

We then backpropagate these tables in the plan graph, until all the tables have reached the initial conditions. First, a table is created for action B, based on the table in g. Its condition is set to the empty set (the condition of the table in g), and its utility function is defined by:

$$V_B(e) = \begin{cases} 0 & \text{if } e < 15 \ ; \\ V_g(e-10) & \text{otherwise} \ ; \end{cases} \quad (1)$$

where $V_B(e)$ and $V_g(e)$ are the (piecewise constant) utility estimates encoded by the tables in B and g respectively. The first line expresses the fact that we are not allowed to start B if the current energy is at or below 15 Ah. The second says that B consumes 10 Ah and leads to g, from where we can get the reward encoded by V_g. A similar table is created for E as shown in Figure 4.

3.3 Conjunctive Preconditions

Since E has two fluents as preconditions, r and t, two copies of its table are created, one for each fluent node. The value functions encoded by these tables are both equal to the function of the table for B ($V_r(e) = V_t(e) = V_B(e)$). However, their conditions are different. The condition of the table for r is $\{t\}$, while the condition of the table for t is $\{r\}$.

The table in t will be backpropagated through action D to a table in s.This table has condition $\{r\}$ and predicts 0 reward if $e \leq 3$ (the consumption of D plus the consumption of E), and 1 otherwise. When this new table is created in s, we need to change its condition from $\{r\}$ to $\{p\}$. To do this we apply the consumption of action C to the table. The result is a table attached to s with $\{p\}$ as condition and predicting 0 reward if $e \leq 6$ (the sum of the consumptions of C, D and E) and 1 otherwise. Similarly, the table in r with $\{t\}$ as condition is backpropagated to fluent p through action C.

We then have two tables, one for p with $\{s\}$ as condition, and one for s with $\{p\}$ as condition, that both encode the same plan: (C, D), then E. However, they represent different orderings of C and D.

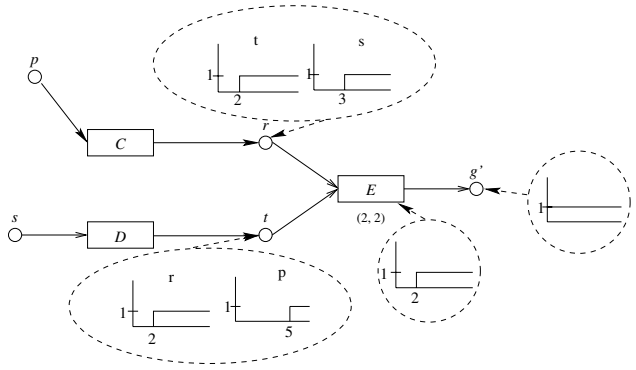

Figure 4: Utility table propagation for conjunctive preconditions

In general, this process may lead to several tables attached to the same node, since there may be several ways to support a fluent. The total number of tables is limited by merging all the tables that have the same condition at some node: they are replaced by a single table that encodes the maximum of all their value functions.

3.4 Conjunctive Effects

The most interesting step of the backpropagation mechanism is illustrated by action A in Figure 5. Since this action has two effects, it will receive utility tables from both nodes.

Each time a table is backpropagated to A, we merge it with all other tables at A. In our example, when we want to backpropagate the table from q, we first test if the table in r can be merged with it. The test is successful if and only if the condition of the table for q implies that of the table for r, which is not true. Therefore, the test fails and we backpropagate the q table independently. However, when we consider backpropagating the table for r, the test is successful (since $t \Longrightarrow \mathrm{true}$) and we merge the two tables. The table in A inherited from r has condition $\{t\}$, and encodes the value function defined by

$$V_A(e) = \begin{cases} 0 & \text{if } e < 15 \ ; \\ \max\{V_q(e-10), V_r(e-10)\} & \text{otherwise} \ . \end{cases} \tag{2}$$

The use of the max operator in equation (2) corresponds to a pessimistic view where we assume that we can never get the rewards of two different goals in the same execution run.

To deal with situations where several goals are reachable, we use a more complex operator that requires augmenting the utility tables. We add: (i) the sum of the expected consumptions of the actions performed to get the utility encoded by the table, and (ii) the goals that are responsible for this utility. They both are a function of the resource level e. These are piecewise constant like the utility function. In the case of action A, we have

$$V_A(e) = \begin{cases} 0 & \text{if } e < 15 \ ; \\ \max\{V_q(e-10) + V_r(e-10-C_q(e-10)), & \\ \quad V_r(e-10) + V_q(e-10-C_r(e-10))\} & \text{otherwise} \ . \end{cases} \tag{3}$$

The first of the two alternatives represents performing A, pursuing the goals beyond q, and then the goals beyond r. The second alternative follows the same reasoning, but pursuing r before q.

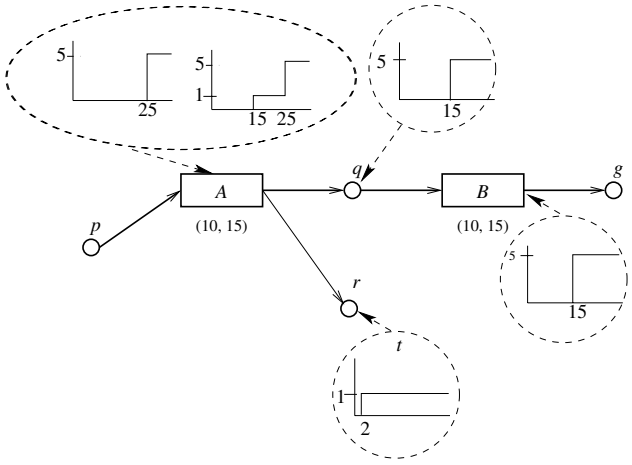

Figure 5: Utility table propagation for conjunctive effects

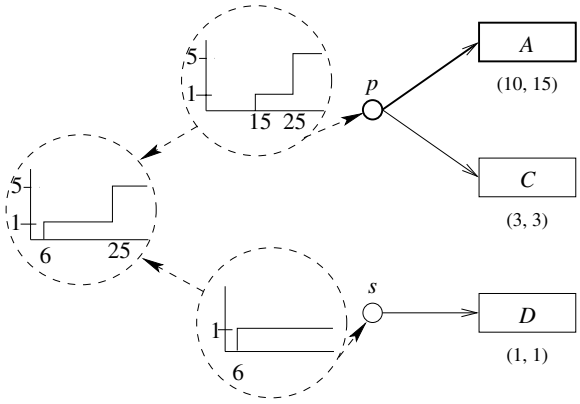

Figure 6: Extracting utility estimates(using the MAX operator)

The information about the goals pursued is used to avoid counting the same goal twice, which is a potential flaw of the previous rule. If the goals pursued in the two tables (for a given resource level) intersect, then we use a simple max rule as in (2). In the case of action A in our example, the goals in the tables attached to q and r do not intersect, so we use rule (3).

3.5 Extracting Utility Estimates

Once the utility tables have been backpropagated down to the fluents representing initial conditions of the problem, we extract the utility estimates for the candidate branch points from the graph. We start with the point between A and B, characterized by the set of fluents $\{p, q, r, s\}$. We build a single utility table for this branch point by merging all utility tables attached to p, q , r and s nodes whose condition is included in $\{p, q, r, s\}$ (that is, whose condition is true when we are at the point between A and B). This is all the tables that represent utility apparently reachable when p, q, r and s are true simultaneously. These tables are merged using a simple max operator as in equation (2) or the more complex operator of equation (3), depending on the context. The resulting table is the value function estimate that we need.

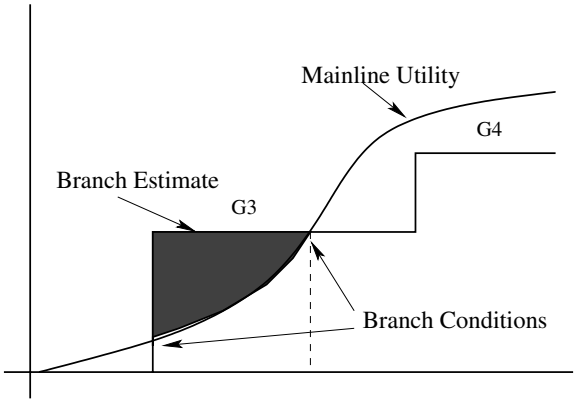

Figure 7: Selecting the branch point, branch condition and goals

As shown in Figure 6, the calculation for the branch point at the beginning of the mainline plan uses two tables:

- the table attached to p with $\{s\}$ as condition and showing that a reward of 1 may be reach if $15 < e \leq 25$, and a reward of 5 may be reached if $e > 25$;
- the table attached to s with $\{p\}$ as condition and showing a reward of 1 may be obtained if $e > 6$ (the sum of the consumptions of C, D and E).

The resulting table, which characterizes this branch point, shows that no reward can be obtained from here if $e < 6$, that a reward of 1 is available if $6 < e \leq 25$, and that a reward of 5 may be obtained if $e > 25$. Using equation (3) instead of (2), we would also have identified the possibility of reaching both g and g' if there are sufficient initial resources.

4 Using Utility Estimates

Given the utility estimates at the various branch points, we can now use this information to select the branch point, the branch condition and the set of goals to pursue. For a particular branch point, we compute the gain in area for the branch utility estimate over the mainline utility. This represents the net utility gain of the branch. The branch condition is composed of the points where the utility curves cross. The goals for the contingent branch correspond to the portion of the utility estimate that is greater than the utility curve of the mainline plan.

For example, in Figure 7, we show the mainline utility curve and the branch estimate curve for a branch point. The shaded area represents the utility gain for the branch. The branch conditions are shown and the goal corresponding to the utility gain is G3.

The JIC Planning algorithm is summarized below:

1. Generate a "seed" plan.
2. Find the best branch
   (a) Estimate the branch utility curves
   (b) Compute the net utility gain
   (c) Identify the branch conditions and associated goals to pursue

3. Generate the contingency branch
4. Insert the branch

5 Conclusions

For a Mars rover, uncertainty is absolutely pervasive in the domain. There is uncertainty in the duration of many activities, in the amount of power that will be used, in the amount of data storage that will be required, and in the location and orientation of the rover. Unfortunately, current techniques for planning under uncertainty are limited to simple models of time, and actions with discrete outcomes. In the rover domain there are concurrent actions, actions of differing duration, and much of the uncertainty is associated with continuous quantities like time, power, position and orientation.

For any non-trivial problem, it seems unlikely that exact or optimal solutions will be possible. In this paper, we have outlined an incremental technique for building up contingent plans. It uses a novel method for estimating the utility of possible branches. We are currently implementing this algorithm for the Mars Smart Lander Technology Demonstration Effort using the EUROPA planning system to generate seed and branch plans.

References

[1] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-dynamic Programming*. Athena, Belmont, MA, 1996.

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

[3] M. Drummond, J. Bresina, and K. Swanson. Just-In-Case scheduling. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1098–1104, 1994.

[4] S. Kambhampati. Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP search techniques in Graphplan. *Journal of AI Research*, 12:1–34, 2000.

[5] M. Littman, J. Goldsmith, and M. Mundhenk. The computational complexity of probabilistic planning. *Journal of AI Research*, 9:1–36, 1998.

[6] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.