

# Planning and Learning in Hybrid Discrete-Continuous Models

Richard Dearden

June 20, 2005

## Abstract

Many real-world problems require richer representations than are typically studied in planning and learning. For example, state estimation in complex systems such as vehicles or spacecraft often requires a representation that captures the rich continuous behaviour of these kinds of systems. Similarly, planning for such systems may require a representation of continuous resource usage, particularly when planning under uncertainty. In this talk I will discuss some commonly used representations of these systems as hybrid systems, examine some approaches to planning and state estimation in them, and finally discuss some first steps toward learning a hybrid model, or at least parameters of such a model, directly from data.

These notes are intended as a set of background information for an IJCAI tutorial talk. They are made up of pieces of text taken from a variety of places, without necessarily having any continuity

## 1 Introduction

In these notes I discuss classical AI problems such as planning, diagnosis, and learning but applied in a much richer space of models than is used in traditional AI applications. The motivation for this is applications such as Mars rovers, automated factories, chemical process plants, and spacecraft for which discrete models such as those traditionally used in planning and learning are inadequate. In particular, we will look at systems represented using hybrid models, in which a mixture of discrete and continuous variables are used. These models use differential equations (or simpler representations where appropriate) to represent the continuous dynamics of the system, but make the equations depend on the discrete state, often referred to as the *mode* of the system. We will use the term *state* to refer to the combination of a mode and a value for each continuous variable. For example, consider a motor. It can be idle or powered, and has a number of fault modes such as having a faulty encoder. These correspond to the discrete part of the model. It also has continuous state, such as the speed it is running at, the current powering it, and so on. In each discrete mode, there is a set of differential equations that describe the relationship between the various continuous values, and the way those values evolve over time. Transitions from mode to mode may occur as a result of actions, or in some models may be induced by the continuous system behaviour, for example when a very high current through the motor causes it to burn out, its mode changes from 'working' to 'failed'.

In many cases, not all of the hybrid system will be observable. Therefore, we also have an observation function that defines the likelihood of an observation given the mode and the values of the continuous variables. All these processes are inherently noisy, and the representation reflects this by explicitly including noise in the continuous values, and stochastic transitions between system modes. We describe this hybrid model in more detail in Section 2.

Since rovers are illustrative of many of the problems we encounter in diagnosing systems of this type, and since the primary testbed we have used in this work is the K-9 rover at NASA Ames, we will use rovers as a running example throughout this talk. From the point of view of planning this example exhibits hybrid behaviour as it moves between discrete locations and operational modes while also needing estimates of continuous quantities such as the amount of power currently available from the battery. From the point of view of state estimation there are discrete variables corresponding to commanded modes and the health or otherwise of components, plus continuous variables such as the currents through different subsystems, or the speed the wheels rotate.

The complex dynamics of a system such as a rover, along with its interaction with a potentially complex, poorly modeled and noisy environment—the surface of Mars in the rover’s case—makes it very difficult to determine the current true state with certainty, or to predict the effects of actions with certainty. The representation we will choose makes these sources of uncertainty explicit, allowing noisy observations of the underlying system state, and probabilistic outcomes for actions. By representing uncertainty explicitly we can reason about it when selecting actions to perform. This is extremely important as actions that appear good for the most likely state may be catastrophic in another of its possible states, or may have unlikely but catastrophic outcomes.

In Section 2 we describe two hybrid system models in detail. The first is a relatively simple model we use for planning, while the second is a richer model used for diagnosis and state estimation, and which makes an interesting target representation for algorithms trying to learn hybrid models of systems. In Section 4 we look at a classically-based approach to planning in these kinds of models, while in Section 5 we look at a Markov Decision Problem-based approach for relatively simple hybrid models. Finally, Section 6 provides an initial description of how such a model might be learned directly from data.

## 2 Hybrid Models

In this section we describe two different approaches to modelling hybrid systems. The first is a relatively simple model used for planning. It is based on the assumption that discrete mode changes are only the result of control actions or the previous discrete mode. The second representation has been used for diagnosis and state estimation. It allows the full richness of arbitrary differential equations, the values of continuous parameters influencing discrete mode changes, and partial observability. It is also the kind of model we would like to learn from data.

### 2.1 Hybrid MDPs

For our simple hybrid representation we adopt a standard MDP model with the addition of continuous state variables:  $\{(Z, \mathbf{X}), A, T, R\}$ . Where  $Z$  is a set of discrete states (which could equally be represented using a set of variables), and  $\mathbf{X}$  is a vector of continuous state variables  $\langle X_1, \dots, X_d \rangle$ . Without loss of generality, we will assume the value of the variables are all in the range  $[0, 1]$ , and thus the state space is the unit square  $[0, 1]^d$ . We use  $(z \in Z, \mathbf{x} \in [0, 1]^d)$  to refer to a particular state.  $A$  is a finite set of actions.

$T$  is the transition model:  $T_a((z', \mathbf{x}'), (z, \mathbf{x}))$  is the probability that the system moves from state  $(z, \mathbf{x})$  to  $(z', \mathbf{x}')$  if action  $a \in A$  is taken. Finally,  $R$  is the reward model:  $R_a^z(\mathbf{x})$  is the reward for taking action  $a$  in state  $\mathbf{x}$ .

We are interested in maximizing the expected total reward of a finite-horizon plan. The Bellman Equation for this model is:

$$V_z^{n+1}(\mathbf{x}) = \max_{a \in A} \{ R_a^z(\mathbf{x}) + \sum_{\mathbf{x}', z'} T_a((z', \mathbf{x}'), (z, \mathbf{x})) V_{z'}^n(\mathbf{x}) \} \quad (1)$$

where  $V_z^n(\mathbf{x})$  stands for the value function over the horizon of  $n$  time-steps.

We define the transition model by a marginal probability distribution on the arrival discrete state:  $T_a^d(z', (z, \mathbf{x}))$ , and a conditional distribution over the continuous space  $T_a^c(\mathbf{x}', z', (z, \mathbf{x}))$  given the arrival discrete state. We will frequently assume that one of the continuous variables in the model is time. This allows us to encode semi-MDPs in this framework. The factorization of the transition probability allows the following alternative form for a Bellman backup:

$$V_z^{n+1}(\mathbf{x}) = \max_{a \in A} \{R_a^z(\mathbf{x}) + \lambda \sum_{z'} T_a^d(z', (z, \mathbf{x})) \sigma_a(z')\} \quad (2)$$

$$\sigma_a(z') = \int_{x'} T_a^c(\mathbf{x}', z', (z, \mathbf{x})) V_{z'}^n(\mathbf{x}') \quad (3)$$

For reasons that will become clear when we use this model for planning, we will often assume that  $T_a^c$  is a discrete distribution, so the integral in Equation 3 can be replaced by a summation.

## 2.2 Probabilistic Hybrid Automata

As we said above, the hybrid MDP representation described above is restricted in terms of the kinds of hybrid system it can represent. For problems such as diagnosis where a richer representation is needed, we use the probabilistic hybrid automaton (PHA) model of [24] and [28]. A PHA is a tuple  $\langle Z, X, Y, F, G, T, P \rangle$ , where:

- $Z = z_1, \dots, z_n$  is the set of discrete modes the system can be in.
- $X = x_1, \dots, x_m$  is the set of continuous state variables which capture the dynamic evolution of the automaton.
- $Y = Y_c \cup Y_d$  is the set of observable variables.  $Y_c$  is the set of continuous observable variables, while  $Y_d$  is the set of discrete observable variables, typically commands sent to the system.
- $F = F_1, \dots, F_n$  is, for each mode  $z_i$  the set of discrete time difference equations  $F_i$  that describe the evolution of the continuous variables  $X$  in that mode. We write  $P(X_t | z_{t-1}, x_{t-1})$  for the distribution over  $X$  at time  $t$  given that the system is in state  $(z, x)$  at  $t-1$ .
- $G = G_1, \dots, G_n$  is, for each mode, the set of equations governing the relationship between the observational variables  $Y$  and the state variables  $X$  and  $Z$ . We write  $P(Y_t | z_t, x_t)$  for the distribution of observations in state  $(z_t, x_t)$ .
- $T$  is a probabilistic transition function over the discrete modes that specifies  $P(Z_t | z_{t-1}, x_{t-1})$ , the conditional probability distribution over modes at time  $t$  given that the system is in state  $(z, x)$  at  $t-1$ . In some systems, this is independent of the continuous variables:  $P(Z_t | z_{t-1}, x_{t-1}) = P(Z_t | z_{t-1})$ .
- $P$  is the prior distribution  $P(Z_0, X_0)$  over states of the system.

As before, we denote a hybrid state of the system by  $s = (z, x)$ , which consists of a discrete mode  $z$ , and an assignment to the state variables  $x$ .

As an example, consider the model shown in Figure 1. This is part of a model for a single wheel of a Mars rover, with four discrete modes (*drive-flat*, *drive-uphill*, *drive-downhill*, and *stuck-wheel*), one continuous variable, *speed*, and one observable variable *obs-speed*. The difference equation for the *drive-downhill* state is shown, as is the relationship between *speed* and *obs-speed* for that mode. The transition function is represented by the arrows connecting the modes.

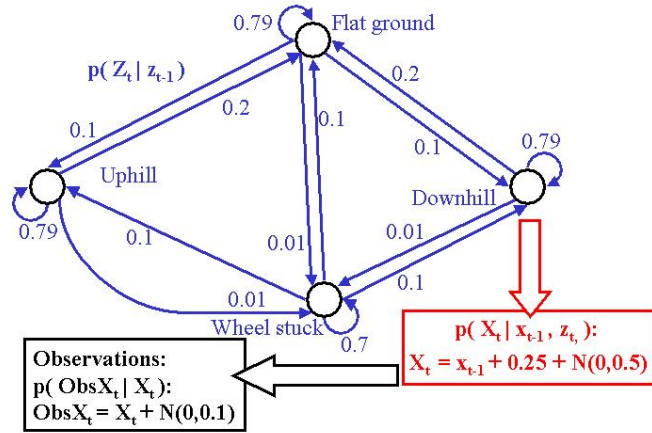


Figure 1: A small example PHA with four discrete states and one continuous variable.

Note that the PHA model we have described assumes discrete decision/observation epochs and summarises the possible discrete transitions of the system—what could happen in a single epoch—by a simple transition function. Sometimes even this model is not expressive enough and we need a continuous-time model such as that presented in [?].

### 3 Planning

The key features of our representation which make it challenging from a planning perspective are the continuous variables and the uncertainty in the model. In this section we briefly examine existing approaches to planning under uncertainty, and discuss whether they could be applied in this hybrid system representation. Table 1 classifies much of this work along the following two dimensions:

Representation of uncertainty: whether uncertainty is modeled strictly logically, using disjunctions, or is modeled numerically, with probabilities.

Observability assumptions: whether the uncertain outcomes of actions are not observable, partially observable, or fully observable.

There are a number of difficulties in attempting to apply existing work on planning under uncertainty to hybrid models of systems such as spacecraft or rovers. First of all, the work listed in Table 1 assumes a very simple model of action in which explicit time constraints are not allowed, and actions are considered to be instantaneous. These characteristics are not as much of an obstacle for Partial-Order Planning frameworks such as SENSP [16], PUCCINI [19], WARPLAN-C [45], CNLP [35], Buridan [26], UDTPOP [34], C-Buridan [14], DTPOP [34], Mahinur [33] and Weaver [6]. In theory, these systems could represent plans in these models. The requirements for a rich model of time and action are more problematic for planning techniques that are based on the MDP or POMDP representations, satisfiability encodings, the graphplan representation, or state- space encodings. These techniques rely heavily on a discrete model of time and action. (See [40] for a more detailed discussion of this issue.) Semi-Markov decision processes (SMDPs) [37] and temporal

	<b>Disjunction</b>	<b>Probability</b>
<b>Non Observable</b>	CGP [41] CMBP [13, 1] C-PLAN [12, 17] Fragplan [25]	Buridan [26] UDTPOP [34]
<b>Partially Observable</b>	SENSp [16] Cassandra [36] PUCCINI [19] SGP [46] QBF-Plan [38] GPT [7] MBP [2]	C-Buridan [14] DTPOP [34] C-MAXPLAN [29] ZANDER [29] Mahinur [33] POMDP [8]
<b>Fully Observable</b>	WARPLAN-C [45] CNLP [35]	JIC [15] Plinth [20] Weaver [6] PGP [5] MDP [8]

Table 1: A classification of planners that deal with uncertainty. Planners in the top row are often referred to as conformant planners, while those in the other two rows are often referred to as contingency planners

MDPs (TMDP) [9] can be used to represent actions with uncertain durations, but as we will see in Section 5 they can only operate on a restricted version of our hybrid representation.

A second, and equally serious, problem with existing planning techniques is that they all assume that uncertain actions have a small number of discrete outcomes. To characterize where a rover could end up after a move operation, or in fact the effects of an action on any continuous parameter in a hybrid model, we would typically have to list all the different possible discrete locations. For many problems this kind of discrete representation is impractical since most of the uncertainty involves continuous quantities, such as the amount of time and power an activity requires. Action outcomes are distributions over these continuous quantities. There is some recent work using models with continuous states and/or action outcomes in both the MDP [3, 30, 31, 39] and POMDP [43] literature, but this has primarily been applied to reinforcement learning rather than planning problems. We will come back to this point in Section 5.

## 4 Just In Case Contingency Planning

In this section we outline a planning approach that has successfully been applied to hybrid models for activity and route planning on a Mars rover. The approach is based on contingency planning, in which a branching plan is built so that as uncertainty is resolved at execution time, the best branch for the actual outcome that occurred can be selected.

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. Of the planning systems in table 1, only Just-In-Case (JIC) contingency scheduling [15] and Mahinur [33] exhibit a principled approach to choosing what contingencies to focus on. The approach we describe here builds upon JIC. 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

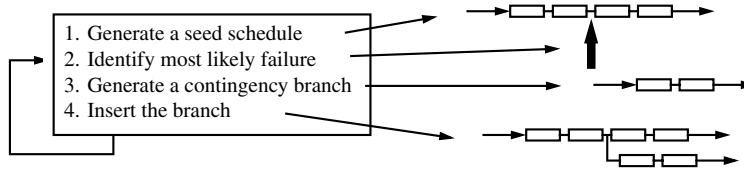


Figure 2: The JIC approach.

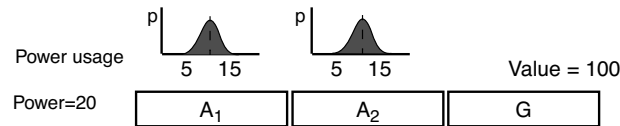


Figure 3: Example showing that the place where the plan is most likely to fail may not be the best branch point.

the resulting contingent schedule is sufficiently robust, or until available time is exhausted. This process is illustrated in Figure 2.

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.<sup>1</sup>

#### 4.0.1 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 3 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

<sup>1</sup>Just as with JIC scheduling, this process is not guaranteed to converge to an optimal contingent plan. However, JIC will always monotonically improve a plan until a local optimum is reached.

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 [3, 42]. 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*. If we had this information, we could easily determine the optimal branch point in the plan. We would 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.

Our extended JIC algorithm finds the best branch point using the following approach. For each possible branch point in the current plan:

1. Calculate the value function (as a function of available resources) of the remaining plan as well as the probability distribution of resource availability at that point (using Monte Carlo simulation).
2. Estimate the value function of the best branch that can be added to the plan at this point, using the procedure described in Section 4.1. This procedure also partitions the value function for a branch according to the set of goals contributing to the expected value; we can thus determine the set of goals responsible for the branch's estimated value.
3. Calculate the net utility gain for adding the best branch plan, as described in Section ???. The best overall branch point is the one with the maximum net utility gain. The branch condition is the condition that defines the region where the value function of the branch is greater than that of the current plan.

We generate the contingency branch using the same planner as for the seed plan, setting the initial conditions equal to the branch condition, and providing the set of goals pursued by the optimal branch. Note that the steps for estimating the value of a branch do not actually construct the branch plan.

Unfortunately, there is no easy way to calculate the exact value function for the best possible branch. The problem is that we are trying to simultaneously optimise the actual branch to be added, the position to insert it into the existing plan, and the branch conditions—the test for whether the branch should be taken or not. Computing all of this would require actually doing the planning for all possible branches. Instead we must approximate this value function.

## 4.1 Estimation of Branch Utility

A critical part of our algorithm is to compute an estimate of the value function of possible branch plans, at each candidate branch-point of the mainline plan. It is based on a representation of the planning problem as a plan graph [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 back-propagation of utility tables in the graph to produce estimates of utility functions (instead of plans). This section provides an outline of this mechanism.

### 4.1.1 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

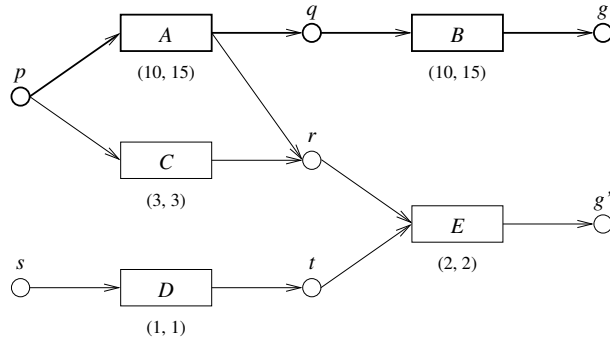


Figure 4: An example of plan graph (partial). The two numbers below each action represent, first, its expected consumption, and second, the minimum power required to be allowed to start this action.

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. 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 4 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 actions *A* and *B*, the expected consumption is 10 Ah, and they can be started only if the current level of resource is at least 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.

#### 4.1.2 Utility Table Back-propagation

The basic principle of our algorithm is to back-propagate utility distribution tables in the plan graph back to the initial state. Each table is attached to a single (action or proposition) node and contains a piecewise constant function giving utility as a function of resource level  $e$  (e.g. energy). 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.

The process is initialized by creating utility tables for the goals. In our example, we start with a table for *g* 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.

We then back-propagate this table in the plan graph, until it has reached the initial conditions. First, a table is created for action *B*, based on the table in *g*. Its utility function is defined by:

$$V_B(e) = \left\{ \begin{array}{ll} 0 & \text{if } e < 15 ; \\ V_g(e - 10) & \text{otherwise ;} \end{array} \right\} \quad (4)$$

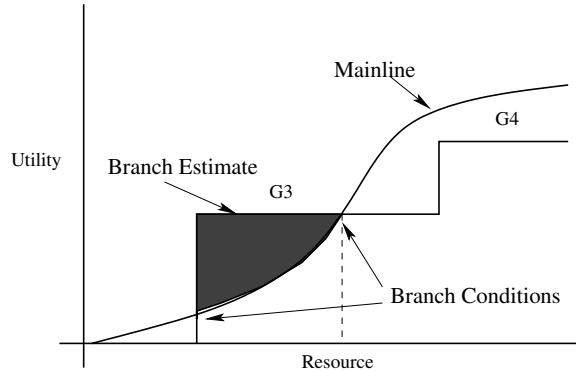


Figure 5: Selecting the branch point, branch condition and goals

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  $V_g$ . Next, the table attached to  $B$  is back-propagated to the previous propositional level. Since  $B$  has only one fluent,  $q$ , as precondition, the table in  $B$  is copied *as is* in  $q$ . Similarly, we back-propagate the table at  $q$  to  $A$  and to  $p$ . The resulting value function is

$$V_p(e) = V_A(e) = \begin{cases} 0 & \text{if } e < 25 ; \\ V_g(e - 20) & \text{otherwise ;} \end{cases} \quad (5)$$

The process described above—that is, regressing the goal  $g$  down to fluent  $p$ —is relatively simple because there is no action with multiple preconditions in the path, and we ignored the fact that action  $A$  makes  $r$  true as well. The more complex cases of actions with multiple preconditions, or multiple effects can be handled by an extension of this approach. The details can be found in [10].

### 4.1.3 Extracting Utility Estimates

Once the utility tables have been back-propagated down to the fluents representing initial conditions of the problem, we extract the utility estimates for the candidate branch points from the graph. Consider a point between two actions  $A$  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 by taking the max of all the functions. The resulting table is the value function estimate that we need.

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 5, 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.

A number of authors working in probabilistic planning have looked at the problem of estimating the utility of a plan. Haddawy et al. [22, 21] have developed the DRIPS planner, which attempts to optimize the utility of the plan it returns. However, DRIPS plans in a very restricted domain represented as an abstraction/decomposition network that constrains the plans that can be created. Comparing abstractions of plans built using this network allows DRIPS to discard plans without having to estimate the utility of a partial plan.

Another related system is Blythe’s Weaver [6]. Weaver builds contingency plans in an incremental fashion very similar to the approach described here, although it only considers actions with discrete outcomes (and uncontrolled external actions). Weaver is concerned with adding contingencies to increase the probability of success of the plan, rather than the utility. It translates the current plan into a Bayesian network to compute the probability of success of the plan, and identifies points where an action can lead to a failure as candidates for new contingent branches. Unlike with our approach, it does not need to estimate the value of the contingent branch before adding it because the discrete domain means that by necessity the new branch must handle an outcome that wasn’t in the original plan.

## 5 Hybrid MDP Planning

Markov Decision Processes (MDPs) have been adopted as a framework for much recent research in decision-theoretic planning. Classic dynamic programming algorithms solve MDPs in time polynomial in the size of the state space. However, the size of the state space is usually very large in practice. For systems modeled with a set of propositional state variables, the state space grows exponentially with the number of variables. This problem becomes even more important for MDPs with continuous state-spaces. If the continuous space is discretized to find a solution, the discretization causes yet another level of exponential blow up. This “curse of dimensionality” has limited the use of the MDP framework, and overcoming it has become an important topic of research.

In discrete MDPs, model-minimization [?] techniques have been used with considerable success to avoid this state explosion problem. Algorithms such as Structured Policy Iteration [?] and SPUDD [23] operate by identifying regions of the state-space that have the same value under the optimal policy, and treating those regions as a single state for the purposes of dynamic programming. In this paper we propose an approach to extend this state aggregation to continuous problems. A particular type of structure appears in some continuous domains such as the Mars rover domain [10], which motivated this work. Figure 6 represents the optimal value function from the initial state of a simple Mars rover problem as a function of two continuous variables: the time and energy remaining. The plan for this state was computed using the algorithm described in the previous section. The shape of this value function is characteristic of the rover domain, as well as other domains featuring a finite set of goals with positive utility and resource constraints. Noticeably, it includes vast plateau regions where the expected reward is nearly constant. These represent regions of the state space where the optimal policy is the same, and the probability distribution on future history induced by this optimal policy is nearly constant. The goal of this work is to exploit this structure by grouping together states belonging to the same plateau, while reserving a fine discretization for the regions of the state space where it is the most useful (such as the curved hump where there is more time and energy available).

We will show that for certain subclasses of Semi-MDPs, we can compute the optimal value function exactly, while exploiting the structure in the problem to perform dynamic programming at far fewer points than a naive approach would. The approach we will describe is restricted to MDPs with piecewise constant or piecewise linear reward functions, and more significantly, to MDPs with discrete transition functions. This means that for any state and action, a finite set of states can be reached with non-zero probability.

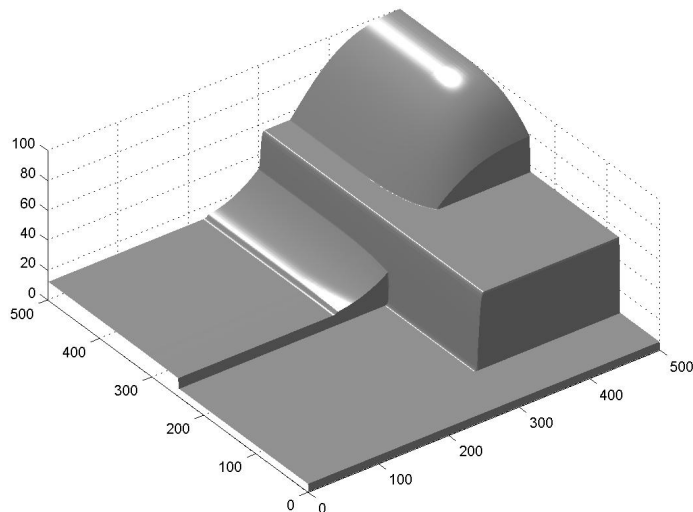


Figure 6: The value function for a rover plan computed using the contingency planning algorithm.

These restrictions ensure that the final optimal value function found by our algorithms belong to well-behaved families. In the case of discrete transition functions and piecewise-constant reward function, the optimal value function is guaranteed to be piecewise constant as well. Similarly, the use of piecewise-linear reward functions ensures a piecewise-linear value function.

The restriction to discrete transition functions is a particularly strong one. However, we can approximate MDPs with continuous transition functions by an appropriately fine discretization of the transition function. This provides an attractive alternative to function approximation approaches [?, ?] in that it approximates the model but then solves the approximate model exactly, rather than finding an approximate value function for the original model. This has the advantage that the effect of the approximation can be much more easily quantified. It also contrasts with the naive approach that consists of discretizing the state space regardless of the relevance of the partition introduced. In our approach, we discretize the action outcomes and deduce from it a partition of the state space. For ease of exposition, we will assume the problems we are solving are piecewise-constant or linear, and that any approximation has already been performed. For this reason we will refer to finding optimal policies and value functions below, even when the model has been approximated.

Given these assumptions, the algorithm we describe will produce as its output a partition of the state-space in which each element of the partition consists of a region where the optimal value function is constant (for a piecewise-constant problem) or is the maximum of a set of linear functions (for a piecewise linear problem). On problems that exhibit structure, the algorithm computes the optimal value function substantially faster than a naive discretization of the state space, even when it must discretize more finely than the naive approach in some regions. The reduction in the number of Bellman backups performed to compute the optimal value function more than offsets the additional cost of maintaining the structured representation.

This work is a generalization of Boyan and Littman’s model of time dependent MDP (TDMDP) that features a single continuous state variable representing time [9], to multiple continuous dimension. Moving a the multi-dimensional framework raises numerous problems relative to storing and manipulating state

partitions. This paper presents our solutions to these problems.

## 5.1 Structural Assumption and Representation

The structure that we exploit consists of partitioning the continuous state space into discrete regions, each of which can be treated as a single entity. In particular, we consider (hyper-)rectangular partitions of the state space  $[0, 1]^d$ . We will use the term “rectangle” instead of “hyper-rectangle” for brevity, and we will discuss examples from a 2-dimensional state space, but the formalism holds for any dimension state space.

The important property of the models is that they are closed under the Bellman backup in Equation 1. There are many models that satisfy this property; in this work we consider piecewise constant and piecewise linear models.

The state-space partitioning applies to both the transition model and the reward function, in turn inducing a similar partitioning for the value function. The transition model partitions the state space into regions for which the set of outcomes  $\Delta_x^a$  and the probability distribution over the set of outcomes are identical. Following Boyan and Littman [9], both *relative* and *absolute* transitions are supported. A relative outcome is expressed as a *delta* from the current state: for a region  $\square$ ,

$$\forall \mathbf{x}, \mathbf{y} \in \square T_a(\mathbf{x} + \delta x, \mathbf{x}) = T_a(\mathbf{y} + \delta x, \mathbf{y}).$$

Thus a relative outcome can be seen as *shifting* a region. An absolute outcome maps all states in a region to a single state:

$$\forall \mathbf{x}, \mathbf{y} \in \square T_a(\mathbf{x}', \mathbf{x}) = T_a(\mathbf{x}', \mathbf{y}).$$

We will concentrate on the relative transition models, since they are more interesting from a formal and algorithmic standpoint. We will mention implications of absolute models where necessary.

The reward model partitions the state space as well, although the partition is not necessarily the same as for the transition model. For each state in a region, the reward is described by the same constant value (piecewise constant case) or the same linear function (piecewise linear case).

The induced value function will be itself partitioned, based on the partitions of the transition and reward models. For each state in a region, the value is described by the same constant value (piecewise constant case) or the max over the same set of linear functions (piecewise linear case). Each of these cases is described in more detail in the following subsections.

### 5.1.1 Piecewise Constant Structure

In this section, we assume the transition and reward model are *rectangular piecewise constant* (RPWC).

**Definition 1 Rectangular Partition** A *rectangular partition* of the state space  $[0, 1]^d$  is a finite set of rectangles  $\boxplus = \{\square_1, \square_2, \dots, \square_k\}$ , where  $\square_i = \prod[\mathbf{x}_i^{low}, \mathbf{x}_i^{high}]$ , such that  $\bigcup_{1 \leq i \leq k} \square_i = [0, 1]^d$ , and  $\square_i \cap \square_j = \emptyset$  iff  $i \neq j$ .

**Definition 2 RPWC** A function  $f : [0, 1]^d \rightarrow \mathbb{R}$  is RPWC, if there exists a rectangular partition  $\boxplus = \{\square_1, \square_2, \dots, \square_k\}$  such that for  $\forall i, 1 \leq i \leq k$  and  $\forall \mathbf{x}, \mathbf{y} \in \square_i$ ,  $f(\mathbf{x}) = f(\mathbf{y})$ .

As shown in Figure 7, the state space is partitioned into rectangular regions. For our implementation, we use  $k$ -dimensional (kd-) trees [18] to store and manipulate the rectangular partitions. A kd-tree is a multidimensional generalization of the binary tree in which space is recursively split by hyper-planes orthogonal to one of the  $k$  axis [].

For the transition model, the relative outcome set, as well as the probability distribution over it, are the same for all states inside an rectangle (for a given action  $a$ ). We will use  $\Delta_{\square}^a$  to refer to the outcome set

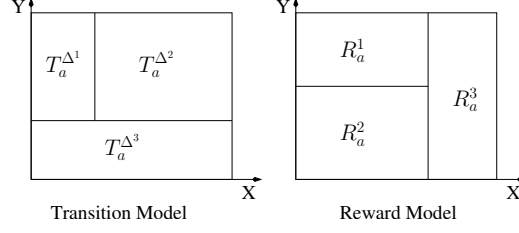


Figure 7: Rectangular piecewise constant models

associated with a rectangle  $\square$  and an action  $a$ . Similarly for the reward model, the rewards  $R_a$  are a constant in each region. For a specific action  $a$ , the transition model is represented by a partition  $\boxplus_T^a$ , and for each rectangle  $\square \in \boxplus_T^a$ , a set of relative outcomes  $\Delta_\square^a$  together with a probability distribution over it. We will use the short form  $T_\square(\mathbf{x})$  to refer to the probability distribution over  $\Delta$  in the rectangle  $\square$ . Similarly, the reward model is represented by a rectangular partition  $\boxplus_R^a$  and for each rectangle  $\square$  a numerical value  $R_\square$  representing the reward. Note that the partitions for the transition and reward model of an action need not be the same.

Applying RPWC assumptions to the standard model described in the previous section results in an MDP  $\mathbf{M1} = \{\mathbf{X}, A, T_\boxplus, R_\boxplus\}$ , where  $T_\boxplus$  and  $R_\boxplus$  are RPWC transition and reward models as described above. We can show that

**Theorem 1** *For MDP  $\mathbf{M1}$ , if  $V^n$  is RPWC, then  $V^{n+1}$  as computed by the Bellman backup in Equation 1, is also RPWC.*

States belonging to the same piece of value function (i) have the same optimal policy; (ii) generate the same probability distribution on future history, in terms of actions performed, rewards received, and pieces of value functions traversed under this optimal policy; and thus, (iii) have the same value. Since we can represent a RPWC function exactly using a set of rectangles, this theorem enables us to carry out the Bellman backup exactly, assuming the initial value function is RPWC.

**Dynamic Programming** We can now describe the Bellman backup procedure for the RPWC model. We first describe how to compute the summation in Equation 1, which we denote as  $\sigma_a$ :

$$\sigma_a := \sum_{\mathbf{x}' \in \Delta_x^a} T_a(\mathbf{x}', \mathbf{x}) V^n(\mathbf{x}')$$

We construct a partition for  $\sigma_a$  by projecting the partition defined by the transition model of action  $a$ , namely  $\boxplus_T^a$ , onto the partition defined by  $V^n$ , using the following procedure:

1. For each region  $\square_j$  in  $\boxplus_T^a$ 
  - (a) For each outcome  $\Delta_i \in \Delta_{\square_j}^a$ 
    - i. Compute the region  $\square_j^{\Delta_i}$  resulting from shifting  $\square_j$  by the relative outcome  $\Delta_i$ .
    - ii. Intersect the shifted region  $\square_j^{\Delta_i}$  with the partition of  $V^n$ , producing sub-regions  $\square_{j,k}^{\Delta_i}$
    - iii. Assign to each sub-region  $\square_{j,k}^{\Delta_i}$  the value of the corresponding region of  $V^n$  multiplied by the probability of the outcome  $\Delta_i$ .

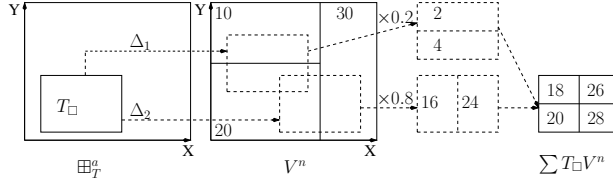


Figure 8: Dynamic Programming for **M1**

- (b) Intersect all the shifted regions from all of the outcomes, producing partition  $\boxplus_{\sigma_a}^j$ .
  - (c) Assign to each of the regions in partition  $\boxplus_{\sigma_a}^j$  the sum of the values of the corresponding sub-regions  $\square_{j,k}^{\Delta_i}$ .
2. Assemble the final partition:  $\boxplus_{\sigma_a} = \cup_j \boxplus_{\sigma_a}^j$ .

As an example, Figure 8 illustrates the sub-dividing process for a single rectangle  $\square \in \boxplus_T^a$ . There are two relative outcomes for this action when executed in the region  $\square$ , namely  $\Delta_1$  with probability 0.2 and  $\Delta_2$  with probability 0.8. For each outcome, we compute the new position of the rectangle  $\square$ , and intersect it with the partition of  $V^n$ . The result of the intersection is then multiplied by the probability of the outcome. Finally, the results of all outcomes are intersected and the summation is computed within each sub-region of the intersection.

With an absolute transition model, the entire outcome for each region takes on a single value, so that the outcome rectangles (shown with dotted lines in Figure 8) would be single-valued throughout. This would eliminate step (1)(a)(ii) and change (1)(a)(iii) to assign the value of the single element of  $V^n$  to the region  $\square_j^{\Delta_i}$ .

The remainder of the Bellman backup involves adding the reward and performing the max over all possible actions:

1. For each action  $a$ 
  - (a) The partition  $\boxplus_{\sigma_a}$  is intersected with the partition of the reward function  $R_a$ , producing partition  $\boxplus_{Q_a}$ .
  - (b) The value of each sub-region in  $\boxplus_{Q_a}$  is computed by summing the values of the corresponding regions of  $\boxplus_{\sigma_a}$  and the partition of the reward function.
2. The partitions  $\boxplus_{Q_a}$  of all actions are intersected, producing  $\boxplus_{V^{n+1}}$ .
3. The value of each region in  $\boxplus_{V^{n+1}}$  is computed as the max of each of the corresponding regions in all the partitions  $\boxplus_{Q_a}$ .

In the above process, a rectangle is further sub-divided only when necessary during the process of intersecting two partitions.

Note that if some actions have a relative effect on some variable, then the partition of the value function gets finer as the horizon increases, which can affect the efficiency of the algorithm. For this reason, it can be necessary to implement a merging mechanism to unify neighboring pieces with the same value. Value-based piece merging breaks some of the properties of the model. Notably, states belonging to the same

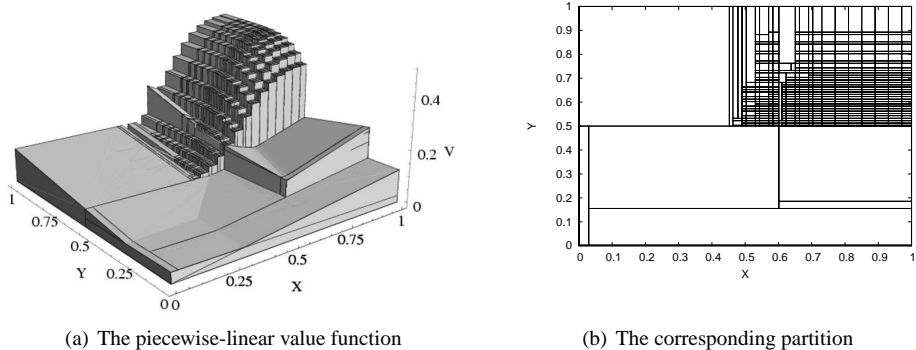


Figure 9: The piecewise-linear value function and the corresponding space partition for the discrete start state in the 2D problem. The resolution is 25 per continuous variable, to show the discretization more clearly.

piece do not have to generate the same probability distribution over trajectories anymore. However, it does not affect the outcome of the algorithm, while possibly allowing tremendous computational savings.

A similar approach can be taken for piecewise linear models. See [?] for details.

The results of this algorithm are shown in Figure 9. The figure shows the resulting value function on a specific discrete state of a problem in the 2D set with RPWLC rewards, and an input discretization of 25 on each dimension. The left side shows the actual function, and the right side shows the corresponding partition over the continuous space. As we can see, fine discretization is only applied to the upper right region. Approximately 70% of the space is treated exactly with only a small number of regions and linear vectors. In contrast, a naive approach would discretize the entire space evenly, expending a large amount of computation on areas that are, in fact, from the same linear function. Our algorithm avoids these computation by treating large regions as a single state.

There are two features in Figure 9 that deserve further analysis. Firstly, although the input is only discretized with a resolution of 25, the resulting partition has considerably more discretization points, albeit all concentrated at the upper right region. This is because the initial partitions defining the transition and reward models are not necessarily aligned with the input discretization, and the dynamic programming process will propagate this dis-alignment along the backups to keep the value function accurate. The naive approach, with a fixed discretization before doing dynamic programming, will not be able to represent such dis-alignment.

Secondly, note the region from around point  $(0.6, 0.2)$  to  $(1.0, 0.5)$ . The value function as can be seen on the left of Figure 9 over this region has a curved shape. It is in fact composed of 13 linear functions. This is typical when the reward model is RPWLC. Again, the dynamic programming is keeping the discretization minimal by automatically grouping states whose value function can be represented in a single PWLC form into an abstract state.

## 5.2 Related Work

The most common approach to continuous state variables in MDPs is either to use function approximators such as artificial neural networks [3, 42], or to discretize the continuous state space more or less naively, which does not scale well to multiple dimensions. Munos and Moore [31, 32] propose a formal model

of a continuous MDP, and the theoretical foundations and algorithms for discretizing it. However, their approach involves solving the MDP at one level of discretization, then locally refining the discretization, and repeating until the approximation is good enough. By taking advantage of the known structure of the problem, we can find the correct level of discretization and solve the MDP only once.

We have already mentioned the time dependent MDP (TMDP) [9], which is probably the closest approach to this one. However, they are restricted to only a single continuous variable.

In the RL literature we find approaches such as U-trees [?] which learns a tree-based representation of a continuous value function similar to ours. However, since this and similar approaches assume an unknown model, they must infer the value function’s structure from observations rather than being able to compute it from the model.

## 6 Learning Differential Equation Models: A Few Thoughts

Building accurate hybrid models of complex systems is extremely difficult. The process could be dramatically improved if we could assist a domain expert by learning models of this kind from data. Even just learning the parameters of the differential equations given information about their structure would be very useful. In this section we sketch the outline of an approach to this problem. The idea is to use techniques from the computational scientific discovery [44, 27] community to learn models of the differential equations operating in each mode. This might require supervised learning to label training data with the mode it represented, but this is a far easier task than trying to build an entire model by hand, and requires much less knowledge on the part of the domain expert.

Traditionally, system models are constructed by a domain expert using their detailed understanding of the system to specify from first principles a dynamical model. This requires a great deal of domain-specific knowledge, and is labour intensive. There is a danger that the expert will neglect aspects of the dynamics of the system that they consider unimportant, but that may later be found to have a significant effect on the behaviour of the system. The models must also be adapted to changing operating conditions— for example, for space operation rather than earth-based testing.

However, the expert does know a great deal about the system they have designed, and this knowledge should be used when learning system models. We propose a Bayesian framework for achieving this. The model that the expert has determined is used as a *prior* for the model structure. The expert can also specify the prior probabilities of each type of addition to the structure of the model. These prior probabilities are specified as augmentations to a context-free grammar that defines the space of models that are being considered. The Bayesian Information Criterion (BIC) is used to rank models, and the priors that augment the model-space grammar are used to guide the search towards models that the expert considers to be a-priori more likely.

### 6.1 Priors over Grammars

We base our approach on Todorovski and Dzeroski’s LAGRAMGE [44] system. LAGRAMGE discovers a model of some system given data from that system and a context-free grammar that represents the space of possible equations that might govern the system’s behaviour. It searches through the space defined by the grammar to find candidate equations, uses the data to fit the parameters of the candidate, and then returns the equations that most closely fit the data.

The grammar LAGRAMGE uses is designed to capture an expert’s knowledge of a domain. For example, the expert may know that a particular domain is governed by a polynomial equation of unknown degree, and would therefore write a grammar that allows polynomials, but not exponentials. However, LAGRAMGE

E	←	E + F	⟨0.3⟩		E - F	⟨0.3⟩		F	⟨0.4⟩
F	←	F * T	⟨0.4⟩		F / T	⟨0.1⟩		T	⟨0.5⟩
T	←	const	⟨0.3⟩		var	⟨0.5⟩		(E)	⟨0.2⟩

Table 2: An example augmented grammar.

provides no method for the expert to specify the likelihood of different constructs in the grammar, and it is this that we address in this work.

Our approach is based on the idea that the expert has some probability information about the different rules in the grammar. For example, the expert may think that a polynomial is more likely than an exponential in the equation, but that the exponential is still possible. To represent this, the grammar rules are augmented by adding conditional probability terms that specify the likelihood of each possible production in the rule. Figure 10 shows a simple example of a grammar with conditional probabilities, and a parse tree from the grammar. The prior probability of the equation derived from the tree is computed as the product of the probabilities of each production in the tree.

## 6.2 Model Search and Evaluation

Given a set of observations  $\mathbf{y}$  of a system, consider two models  $M_1$  and  $M_2$  parameterised by  $\Theta_1$  and  $\Theta_2$ . We are interested in the *Bayes Factor* [11], which tells us how much more probable model  $M_1$  is compared to model  $M_2$ . This is given by

$$BF = \frac{l(\mathbf{y}|\Theta_1)p(M_1)}{l(\mathbf{y}|\Theta_2)p(M_2)} \quad (6)$$

where  $l()$  is the likelihood function, and  $p(M_1)$  is the prior probability computed from the structure of  $M_1$  using the expert-defined prior in the augmented grammar. If  $\mathbf{y}$  contains many data points, then the likelihood ratio in Equation 6 can be approximated. Taking logs, the BIC approximation gives

$$\Delta BIC = W - (p_2 - p_1) \log n + \log(p(M_1)) - \log(p(M_2)) \quad (7)$$

where  $p_1, p_2$  are the number of parameters in the two models,

$$W = -2 \log \left[ \frac{\sup_{M_1} l(\mathbf{y}|\Theta)}{\sup_{M_2} l(\mathbf{y}|\Theta)} \right] \quad (8)$$

and  $\Delta BIC$  is the change from model 1 to model 2. This allows us to evaluate the relative probabilities of two models, and can be generalised trivially to compare  $N$  models.

During model search we also use the priors over the model structure defined by the augmented grammar to determine which models to consider next. Using a beam search, we generate the next set of models as those that are a-priori most likely, and then retain those whose posterior probability is highest. Thus augmenting the grammar with prior probabilities enables both model comparison and a principled search through model space.

## 7 Conclusions

Hybrid models present a considerable challenge for many areas of AI, but particularly for planning, learning, and diagnosis or state estimation. The goal is to reason about models that are much more closely



- [8] C. Boutilier, T.L. Dean, and S. Hanks. Decision theoretic planning: structural assumptions and computational leverage. *Journal of AI Research*, 11:1–94, 1999.
- [9] J.A. Boyan and M.L. Littman. Exact solutions to time-dependent MDPs. In *Advances in Neural Information Processing Systems 13*, pages 1–7. MIT Press, Cambridge, 2000.
- [10] J. Bresina, R. Dearden, N. Meuleau, S. Ramakrishnan, D. Smith, and R. Washigton. Planning under continuous time and resource uncertainty: A challenge for ai. In *Proc. of the Eighteenth Conf. on Uncertainty in Artificial Intelligence*, 2002.
- [11] Bradley P. Carlin and Thomas A. Louis. *Bayes and Empirical Bayes Methods for Data Analysis*. Chapman and Hall, New York, 2nd edition edition, 2000.
- [12] C. Castellini, E. Giunchiglia, and A. Tacchella. Improvements to sat-based conformant planning. In *Proc. of the 6th European Conf. on Planning*, 2001.
- [13] A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *Journal of AI Research*, 11:305–338, 2000.
- [14] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. of the Second Intl. Conf. on Artificial Intelligence Planning and Scheduling*, pages 31–36, 1994.
- [15] M. Drummond, J. Bresina, and K. Swanson. Just-In-Case scheduling. pages 1098–1104, 1994.
- [16] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In *Proc. of the Third Intl. Conf. on Principles of Knowledge Representation and Reasoning*, pages 115–125, 1992.
- [17] E. Ferraris and E. Giunchiglia. Planning as satisfiability in nondeterministic domains. In *Proc. of the 17th National Conf. on Artificial Intelligence*, 2000.
- [18] J.H. Friedman, J.L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.
- [19] K. Golden. Leap before you look: information gathering in the PUCINI planner. In *Proc. of the Fourth Intl. Conf. on Artificial Intelligence Planning and Scheduling*, pages 70–77, 1998.
- [20] R. Goldman and M. Boddy. Conditional linear planning. In *Proc. of the Second Intl. Conf. on Artificial Intelligence Planning and Scheduling*, pages 80–85, 1994.
- [21] Richard Goodwin. Using loops in decision-theoretic refinement planners. In B. Drabble, editor, *Proc. of the Third Intl. Conf. on Artificial Intelligence Planning and Scheduling*, pages 118–124. AAAI Press, 1996.
- [22] Peter Haddawy, AnHai Doan, and Richard Goodwin. Efficient decision-theoretic planning: Techniques and empirical analysis. In *Proc. of the Eleventh Conf. on Uncertainty in Artificial Intelligence*, pages 229–236, Montreal, 1995.
- [23] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. Stockholm, Sweden, 1999.

- [24] Michael W. Hofbaur and Brian C. Williams. Hybrid diagnosis with unknown behaviour modes. In *Proceedings of the Thirteenth International Workshop on Principles of Diagnosis*, pages 97–105, Semmering, Austria, 2002.
- [25] J. Kurien, P. Nayak, and D. Smith. Fragment-based conformant planning. In *Proc. of the 6th Intl. Conf. on Artificial Intelligence Planning and Scheduling*, 2002.
- [26] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, 1995.
- [27] P. Langley, J. Sanchez, L. Todorovski, and S. Dzeroski. Inducing process models from continuous data. In *Proceedings of Machine Learning 2002*, pages 336–349, 2002.
- [28] Uri Lerner, Ron Parr, Daphne Koller, and Gautam Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, 2000.
- [29] S. Majercik and M. Littman. Contingent planning under uncertainty via stochastic satisfiability. In *Proc. of the 16th National Conf. on Artificial Intelligence*, 1999.
- [30] R. Munos. Variable resolution discretization for high-accuracy solutions of optimal control problems. In *Proc. of the 16th Intl. Joint Conf. on Artificial Intelligence*, 1999.
- [31] R. Munos. A study of reinforcement learning in the continuous case by the means of viscosity solutions. *Machine Learning*, 40:265–299, 2000.
- [32] R. Munos and A. Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49:291–323, 2002.
- [33] N. Onder and M. Pollack. Conditional, probabilistic planning: A unifying algorithm and effective search control mechanisms. In *Proc. of the 16th National Conf. on Artificial Intelligence*, pages 577–584, 1999.
- [34] M. Peot. *Decision-Theoretic Planning*. PhD thesis, Dept. of Engineering-Economic Systems, Stanford University, 1998.
- [35] M. Peot and D. Smith. Conditional nonlinear planning. In *Proc. of the First Intl. Conf. on Artificial Intelligence Planning and Scheduling*, pages 189–197, 1992.
- [36] L. Pryor and G. Collins. Planning for contingencies: a decision-based approach. *Journal of AI Research*, 4:287–339, 1996.
- [37] M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, NY, 1994.
- [38] J. Rintanen. Constructing conditional plans by a theorem prover. *Journal of AI Research*, 10:323–352, 1999.
- [39] W. Smart and L. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proc. of the 17th Intl. Conf. on Machine Learning*, 2000.
- [40] D. Smith, J. Frank, and A. Jónsson. Bridging the gap between planning and scheduling. *The Knowledge Engineering Review*, 15(1), 2000.

- [41] D. Smith and D. Weld. Conformant graphplan. In *Proc. of the Fifteenth National Conf. on Artificial Intelligence*, pages 889–896, 1998.
- [42] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.
- [43] S. Thrun. Monte Carlo POMDPs. In *Advances in Neural Information Processing Systems 12*, pages 1064–1070. MIT Press, Cambridge, 1999.
- [44] Ljupčo Todorovski and Sašo Džeroski. Declarative bias in equation discovery. In *Proc. 14th International Conference on Machine Learning*, pages 376–384. Morgan Kaufmann, 1997.
- [45] D. Warren. Generating conditional plans and programs. In *Proc. of the Summer Conf. on AI and Simulation of Behavior.*, 1976.
- [46] D. Weld, C. Anderson, and D. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proc. of the Fifteenth National Conf. on Artificial Intelligence*, pages 897–904, 1998.