Planning and Control in Artificial Intelligence: A Unifying Perspective

Blai Bonet and Héctor Geffner
Departamento de Computación
Universidad Simón Bolívar
Aptdo. 89000, Caracas, Venezuela

Abstract

The problem of selecting actions in environments that are dynamic and not completely predictable or observable is a central problem in intelligent behavior. In AI, this translates into the problem of designing controllers that can map sequences of observations into actions so that certain goals are achieved. Three main approaches have been used in AI for designing such controllers: the *programming* approach, where the controller is programmed by hand in a suitable high-level procedural language, the *planning* approach, where the control is automatically derived from a suitable description of actions and goals, and the *learning* approach, where the control is derived from a collection of experiences. The three approaches can exhibit successes and limitations. The focus of this paper is on the *planning approach*. More specifically, we present an approach to planning based on various *state models* that can handle various types of *action dynamics* (deterministic and probabilistic) and *sensor feedback* (null, partial, and complete). The approach combines *high-level representations languages* for describing actions, sensors, and goals, *mathematical models of sequential decisions* for making precise the various planning tasks and their solutions, and *heuristic search algorithms* for computing those solutions. The approach is supported by a computational tool we have developed that accepts high-level descriptions of actions, sensors, and goals and produces suitable controllers. We also present empirical results and discuss open challenges.

1 Introduction

The problem of selecting actions in environments that are dynamic and not completely predictable or observable is a central problem in intelligent behavior. In AI, this translates into the problem of designing controllers that can map sequences of observations into actions so that certain goals can be achieved (Fig. 1). Three main approaches have been used in AI for designing such controllers:

$$\text{Goals} \longrightarrow \boxed{\text{Controller}} \quad \underset{\longleftarrow}{\overset{actions}{\longrightarrow}} \atop observations \quad \boxed{\text{World}}$$

Figure 1: The *control problem:* designing controllers that can map sequences of observations into actions for achieving certain goals

- the *programming* approach, where the controller is programmed by hand in a suitable high-level procedural language,
- the *planning* approach, where the controller is derived automatically from a suitable description of the actions, sensors, and goals, and
- the *learning* approach, where the controller is derived from experiences.

The work of R. Brooks and others in mobile robotics is an example of the first approach [16, 1], Control Theory and AI planning are examples of the second approach [48, 25], and systems that learn by trial and error or generalization are instances of the third approach [58, 7].

The three approaches exhibit both strengths and limitations. In this paper we focus on the *planning approach.* More specifically, we present a general and operational approach to planning that combines elements from AI planning, dynamic programming, and logic, and is able to deal with systems that exhibit different types of *dynamics* (deterministic or probabilistic) and different types of *feedback* (null, partial, or complete). Plans are thus *open-loop* or *closed-loop* according to the type of sensors available (Fig. 2).

The main ingredients of the approach are the use of *high representations logical languages* for describing actions, sensors, and goals, *mathematical models of sequential decisions* for making precise the various planning tasks and their solutions, and *heuristic search algorithms* for obtaining those solutions.

The use of high-level representation languages is common in AI Planning [25, 50] and our approach draws insight from recent work on theories of action [29, 54, 28]. Similarly, the use of mathematical models of sequential decisions such as Markov Decision Processes (MDPs), and Partially Observable MDPs (POMDPs) borrows from the work in Dynamic Programming [51, 6]. Finally, the use heuristic search algorithms has long tradition in AI (e.g., [46]), even though the use of such algorithms for solving Strips planning problems, MDPs, and POMDPs is more recent [3, 13, 11, 32].

In this paper we provide a coherent integration of these various elements and argue that the result provides a natural framework for modeling and solving planning problems under a variety of conditions, including deterministic and probabilistic actions, and complete, partial or null sensing. We have actually developed a computational tool that supports this approach and given a convenient description of actions, sensors, and goals, computes the corresponding controllers. We report a number of experiments with this tool and discuss current limitations and future challenges.

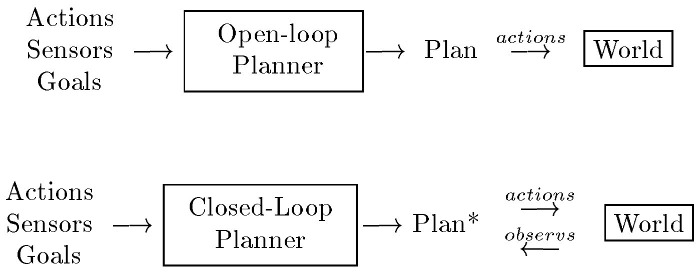

Figure 2: *Open and Closed-Loop Planning:* in open-loop planning the plan is a fixed sequence of actions; in closed-loop planning, the sequence depends on the observations

The paper is written in a tutorial style requiring only a basic knowledge of logic and probabilities. We have tried to keep the presentation coherent but have made no attempt to provide a survey of the area. For more comprehensive treatments on Planning and Control in AI, and Decision-Theoretic Planning, see [23, 53, 14, 35].

The paper is organized as follows. First, we consider the models of sequential decisions that make explicit the mathematical structures underlying various planning tasks (Models, Sect. 2). Second, we consider the class of search algorithms for solving these models (Algorithms, Sect. 3). Third, we discuss the use of a logical language for specifying such models in a convenient way (Representation, Sect. 4). And finally, we report results on a variety of experiments (Results, Sect. 5).

2 Models

We consider first the mathematical models that make various forms of planning precise according to the type of *action dynamics* and *sensor feedback*. In each case, the models determine what the planning task *is* and what is the *form* of the solution.

2.1 State Models

Deterministic state models [45] are the most basic action models in AI and consist of a finite set of states S, a finite set of actions A, and a state transition function f that describes how actions map one state into another. Deterministic state models provide useful models for the *dynamics* of systems in which the effects of actions are discrete and fully predictable.

A *Deterministic Control Problem* or DCP refers to the problem of finding the actions that would drive a system described by a deterministic state model, from a given initial state s_0 to a final set of goal states G. More precisely, a deterministic control problem is characterized by:

- a state space S
- an initial state $s_0 \in S$
- actions $A(s) \subseteq A$ applicable in each state $s \in S$
- a transition function $f(s, a)$ for $s \in S$ and $a \in A(s)$
- action costs $c(a, s) > 0$
- a non empty set $G \subseteq S$ of goal states

A *solution* of a deterministic control problem is a sequence of actions $a_0, a_1, \ldots, a_n$ that generates a state trajectory $s_0,\ s_1 = f(s_0),\ \ldots,\ s_{n+1} = f(s_i, a_i)$ such that each action a_i is applicable in s_i and s_{n+1} is a goal state, i.e., $a_i \in A(s_i)$ and $s_{n+1} \in G$. The solution is *optimal* when the total cost $\sum_{i=0}^{n} c(s_i, a_i)$ is minimal.

Classical planning, i.e., open-loop planning with deterministic actions and complete knowledge of the initial situation, is a deterministic control problem where states are normally represented by sets of atoms, action costs are equal, and the transition function f and the sets of executable actions $A(s)$ are implicitly defined in a high-level language such as Strips [25].

While solution methods in classical planning have traditionally been based on divide-and-conquer ideas [46, 53], methods based on explicit state space exploration and suitable domain-independent heuristic have recently been shown to be effective [12].

2.2 MDPs

Markov Decision Processes (MDPs) [51, 5] differ from deterministic control problems in two ways: first, they accommodate *probabilistic actions*, second, they assume that the effect of actions, while no longer predictable, is *fully observable*. An MDP is thus given by:[1]

- a state space S
- actions $A(s) \subseteq A$ applicable in each state $s \in S$
- transition probabilities $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- action costs $c(a, s) > 0$
- a non empty set $G \subseteq S$ of goal states

The states s_{i+1} that result from an action a_i are *not predictable* but are *observable*, providing *feedback* for the selection of the next action a_{i+1}. As a result, a *solution* of an MDP is not an action sequence, but a function π mapping states s into actions $a \in A(s)$. Such a function is called a *policy*. A policy π assigns

[1] We are considering a subclass of MDPs, the so-called *stochastic shortest-path* MDPs [5]. For general treatments, see [5] and [51]. For uses of MDPs in Planning and AI, see [58, 14, 3, 53].

a probability to every state trajectory $s_0, s_1, s_2, \ldots$ starting in state s_0 which is given by the product of all transition probabilities $P_{a_i}(s_{i+1}|s_i)$ with $a_i = \pi(s_i)$. We assume that actions in goal states have no costs and produce no changes (i.e., $c(a,s) = 0$ and $P_a(s|s) = 1$ if $s \in G$). The *expected cost* associated with a policy π starting in state s is the weighted average of the probability of such trajectories times their cost $\sum_{i=0}^{\infty} c(\pi(s_i), s_i)$. An *optimal solution* is a control policy π^* that has a minimum expected cost for all states $s \in S$.

While Classical Planning can be formulated as a deterministic control problem, Closed-Loop Planning with Complete Information can be formulated as an MDP [21, 3]. The desired closed-loop plans are the optimal policies π^*. In AI, it is common to *represent* policies by lists of condition-action pairs or universal plans [55, 47].

2.3 POMDPs

POMDPs generalize MDPs allowing the state to be *partially observable* [57, 17]. Information about the state comes from observations o whose probabilities $P_a(o|s)$ depend on the action a performed and the unobserved but true resulting state s. In addition, a prior probability distribution over the states encodes a *prior belief* about the initial state of the world which is no longer assumed to be observable or known. A POMDP is thus characterized by the elements describing an MDP

- states $s \in S$
- actions $A(s) \subseteq A$ applicable in each state s
- transition probabilities $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- costs $c(a,s) > 0$ of performing action a in s
- a non-empty set $G \subseteq S$ of goal states

plus information about the initial state of uncertainty and the sensor model in the form of

- an initial belief state b_0, and
- a set O of observations o with probabilities $P_a(o|s)$

The probabilities $P_a(o|s)$ express the probability of getting the observation o in state s after having done action a. These probabilities must be defined for each state s and action $a \in A(s)$ and must add up to one, i.e. $\sum_{o \in O} P_a(o|s) = 1$.

Since feedback from the environment is only partial in POMDPs, the state of the system is normally not known, and therefore, policies that map states into actions are of no use. The *solution* of a POMDP takes the form of a function that maps *belief states* into actions, where belief states are probability distributions over the real states of the environment. The effect of actions on *belief states* is completely predictable, and the belief state b_a that results from performing action a in belief state b can be obtained as

$$b_a(s) = \sum_{s' \in S} P_a(s|s')b(s') \tag{1}$$

In the *absence of observations*, a POMDP reduces to a deterministic control problem in *belief space* where the task is to find a sequence of actions that maps the initial belief state b_0 into a final belief state b_F with actions a that map one belief b into a successor belief b_a as given in (1). We take the *final belief states* to be the beliefs that make the goal certain, i.e., the b_F's for which $b_F(s) = 0$ for all $s \notin G$, or more simply, $b_F(G) = 1$. Different sets of final belief states could also be used; e.g., the beliefs b_F that make the goal very likely ($b_F(G) \geq 0.9$), and so on.

In the *presence of observations,* an action a can map a belief state b into several belief states b_a^o according to the observation o that obtains. The probability $b_a(o)$ of observing o is given by

$$b_a(o) = \sum_{s \in S} P_a(o|s) b_a(s) \tag{2}$$

Similarly, the probability $b_a^o(s)$ that the state is s after doing action a in b and observing o is

$$b_a^o(s) = P_a(o|s) b_a(s) / b_a(o) \tag{3}$$

These expressions follow from the action and sensor models, and Bayes rule.

In the presence of observations thus actions have a probabilistic effects on belief states, and hence a POMDP with sensing is no longer a deterministic control problem in belief space, but an MDP over belief space [2, 57, 17]. The solution of the POMDP, is given by the solution of such *belief* MDP; namely, a policy mapping belief states into actions such that the expected cost for going from the initial state b_0 to a final belief state b_F is minimized.

The problem of planning with sensing [42] can be formulated as a POMDP whose solutions are policies mapping belief states into actions. In the AI literature, such policies are often *represented* by contingent plans, i.e., sequential plans extended with tests and branching [19, 42, 20].

2.4 Non-deterministic Control Problems

In MDPs and POMDPs, non-determinist transitions are modeled as probabilistic transitions. In certain contexts, however, we may want to describe the *possible* effects of an action without specifying the corresponding probabilities. This applies in particular to situations in which we are interested in minimizing the *worst possible cost* of a policy rather than its *expected cost* [34, 38]. Such sequential decision problems can be expressed with a *dynamic model* made up of a *non-deterministic transition function* $F(a, s)$ mapping actions and states into *sets* of states. Then, *non-deterministic* MDPs can be defined by simply replacing the transition probabilities with non-deterministic transition functions, and expected costs with worst possible costs. For defining *non-deterministic* POMDPs, we also need to replace the probabilistic sensor model $P_a(o|s)$ by a non-deterministic sensor model made up of *sets* $O(a, s)$ of *possible* observations o that may arise in state s after having done action a. As we will see, the algorithms for solving MDPs and POMDPs can easily be adapted for solving their non-deterministic counterparts.

Non-deterministic planning problems with and without sensing have been considered in the AI literature in [56, 20, 18] among other sources.

3 Algorithms

We have seen that planning problems can be formulated as either DCPs, MDPs, or POMDPs, or as their non-deterministic counterparts, according to the type of action dynamics and sensor feedback that is present. Techniques for solving deterministic control problems are reviewed in details in most AI texts; e.g., [53], and include algorithms such as A* which rely on heuristic functions $h(s)$ that estimate the cost from states s to the goal. Here we focus first on Greedy or Hill-Climbing Search, which is a simple search strategy that can be easily extended to deal with *all* the models we have considered.

3.1 Greedy Search

Greedy search is one of the simplest search strategies. In a state s, the best action according to some criterion is applied, and the same process is repeated until the goal is reached. In deterministic control problems, it's natural to define the best action in terms of a measure $Q(a, s)$ given by the sum of the cost $c(a, s)$ of applying the action a in s, and the estimated cost of reaching the goal from the resulting state as measured by an heuristic function h. This search procedure, that we call GREEDY, is shown in Fig. 3.

GREEDY is a simple search procedure that uses constant memory but is neither optimal nor complete. That is, GREEDY may return solutions with non-minimal costs or may loop and return no solution at all. Nonetheless, GREEDY can often be amended to become sufficiently practical. For a example, the planner HSP [12] is based on a greedy search procedure guided by an heuristic function $h(\cdot)$ that is extracted automatically from Strips representations. The greedy search is extended with memory to avoid past states, a limited number of plateu-moves (in which the heuristic h is not decremented), and multiple restarts. HSP was entered into the AIPS98 Planning Contest where it solved more problems than state-of-the-art Graphplan and SAT planners [44] .

In the main looop of GREEDY in Fig. 3, the state s' resulting from the application of action a in s (Step 3) is taken to be the state s_a that is predicted by the (deterministic) model of the system. With full observability, however, GREEDY can be easily extended to handle 'perturbations' or behaviors not predicted by the model. Indeed, when the resulting state s' is not the same as the predicted state s_a in Step 3, GREEDY continues the search from s' as if nothing had happened. This can occur for instance when a block A that was supposed to be moved on top of block B unexpectedly falls on the table. In that case, provided that the new state is *observable* and s' is set to the resulting state, GREEDY can recover. In such case, the greedy search behaves as a closed-loop policy that we refer as the *greedy policy.*

1. **Evaluate** each action a applicable in current state s as

$$Q(a, s) = c(a, s) + h(s_a)$$

where s_a is state predicted after doing a in s

2. **Apply** action $\mathbf{a}$ that minimizes $Q(\mathbf{a}, s)$, breaking ties randomly

3. **Exit** if resulting state s' is a goal state, else set s to s' and go to 1

Figure 3: Greedy search

3.2 Learning Real-Time A*

Greedy search is simple and memory efficient but has two problems: it may return non-optimal solutions, or it may *loop* and return no solution at all. In [39], Korf proposed a simple change in GREEDY (that he calls *real-time search*) that solves both of these problems. The idea is to adjust the heuristic function h dynamically during the search. More precisely, Korf suggests to regard the heuristic function h as providing the *initial* value of a function V that estimates the *optimal cost* of reaching the goal from each state. Then every time an action a is selected and applied in a state s, this cost function V is updated as

$$V(s) := c(a, s) + V(s_a) \tag{4}$$

where $V(s_a)$ is the value of the cost function for the predicted state s_a. These updates aim to enforce the relation that has to hold between the cost of s and the cost of s_a when a is indeed the optimal action in s. For example, if initially $V(s) = h(s) = 5$, and action a with cost $c(a, s) = 1$ is applied in s leading to s_a with cost $V(s_a) = h(s_a) = 3$, the update changes $V(s)$ from 5 to 4.

Korf shows that the result of these updates is twofold: first they guarantee that the greedy search will not be trapped into a loop as long as there is a path from every state to the goal; second, they guarantee that *successive trials* of the algorithm, each trial beginning with the value function resulting from the previous trial, eventually deliver an *optimal* path to the goal. This is provided that the heuristic h used to initialize the value function V is *admissible* (non-overestimating).

Korf refers to the greedy search procedure extended with these updates as *learning real-time A** or LRTA*. For the implementation of LRTA*, the estimates $V(s)$ are stored in a hash table that initially contains the heuristic value of the starting state only. Then, when the value $V(s)$ of a state s that is not in the table is needed, a new entry with $V(s)$ set to $h(s)$ is allocated (Step 1, Fig. 4). These entries are updated following (4) when a move from s is performed. The main loop of a single trial of the LRTA* algorithm is shown in Fig. 4.

As an illustration, if LRTA* is given the non-informative but admissible heuristic $h(s) = 0$ in the 8-puzzle, the updates would guarantee that LRTA*

1. **Evaluate** each action a applicable in s as

$$Q(a, s) = c(a, s) + V(s_a)$$

initializing $V(s_a)$ to $h(s_a)$ when s_a is not in the table

2. **Apply** action $\mathbf{a}$ with minimum $Q(\mathbf{a}, s)$ value, breaking ties randomly
3. **Update** $V(s)$ to $Q(\mathbf{a}, s)$
4. **Exit:** if resulting state s' is goal, else set s to s' and go to 1

Figure 4: Single trial of LRTA*

will find a solution to the goal in every trial. Moreover, after a sufficiently large number of trials, LRTA* will reach a point in which the updates no longer change the value function. The solutions returned by LRTA* from that point on, can be shown to be *optimal.* The rate at which LRTA* converges to the optimal solution depends on the size of the state space and the quality of the heuristic function h. A better heuristic yields a more focused search, a higher ratio of updates on the relevant states, and faster convergence. However, if the state space is very large and the heuristic is not good enough, the time and space requirements for convergence may grow impractically large.

A common way to speed up LRTA* is by cutting off trials that take too many steps. This is normally done by selecting a cutoff value known to be above the optimal number of steps. For the use of LRTA* in planning, see [13].

3.3 Dynamic Programming

Barto *et al.* in [3] provide an explanation of LRTA* in terms of *dynamic programming.* In dynamic programming [4, 51, 5], a deterministic control problem is solved by finding the function V^* that assigns to each state s the optimal cost $V^*(s)$. This optimal value function satisfies the fixed point equation

$$V^*(s) = \min_{a \in A(s)} [c(a, s) + V^*(s_a)] \tag{5}$$

also called the Bellman equation. Given $V^*(s)$, the optimal actions in s are the ones that minimize the right-hand side of (5) [51, 6].

A common way for finding the function V^* is by an iterative method known as *value iteration* in which estimates V_i are plugged into the right-hand side of (5) so that an improved value function V_{i+1} is obtained on the left-hand side:

$$V_{i+1}(s) := \min_{a \in A(s)} [c(a, s) + V_i(s_a)] \tag{6}$$

These updates are done in parallel for all states $s \in S$. The new value function V_{i+1} can be plugged again into the right-hand side of (5) to yield still better estimates, until eventually the optimal value function is obtained. Namely, under suitable conditions, parallel value iteration yields a sequence of value functions V_0, V_1, ..., such that $\lim_{i \to \infty} V_i = V^*$ [6].

For the implementation of value iteration, two vectors V and V' of size $|S|$ are needed. One stores the current estimates; the other, the new estimates. Actually, a way for reducing space and speeding up convergence is to use a single vector V for storing current and new estimates. The updates then get the simpler form

$$V(s) := \min_{a \in A(s)} [c(a,s) + V(s_a)] \tag{7}$$

The difference with parallel-value iteration is that once the value $V(s)$ of an state s is updated according to (7), it immediately becomes available for computing the new value $V(s')$ of other states.

A second variation on parallel value iteration is to perform the updates of the form (7) over subsets S' of S or even over *single states* s in S:

$$V(s) := \min_{a \in A(s)} [c(a,s) + V(s_a)] \text{ for } \textbf{selected } s \in S \tag{8}$$

The order in which these states s are selected for updates can be arbitrary, and as long as all states are selected infinitely often, the function V converges to the optimal value function [5, 6].

The relation between this form of *asynchronous value iteration* and Korf's LRTA* must be clear by now. As noticed in [3], by combining a greedy search with updates, LRTA* is performing a form of value iteration in which *the states selected for update are the ones visited during the search.* The novelty in LRTA* in comparison with other forms of value iteration, is that LRTA* does not require to update *all* states infinitely often for delivering an optimal solution. Indeed, provided that the heuristic used to initialize the cost function V is admissible (non-overestimating), *the updates can be restricted to the states visited during the search;* all other states can be safely ignored. This set of states can be a small fraction of the total number of states if the heuristic h is sufficiently good. This is crucial and in many cases makes LRTA* applicable in problems where standard dynamic programming methods are not. Indeed, LRTA* unlike standard dynamic programming methods makes use of the initial state s_0, and does not compute the optimal value function over *all* states but over a fraction of *relevant* states that include those appearing in the optimal trajectories from s_0.

3.4 Real Time Dynamic Programming

Dynamic programming methods apply not only to deterministic problems but to probabilistic and non-deterministic problems as well. For MDPs the fixed point equation characterizing the optimal value function V^* becomes

$$V^*(s) = \min_{a \in A(s)} [c(a,s) + \sum_{s' \in S} P_a(s'|s) V^*(s')] \tag{9}$$

1. **Evaluate** each action a applicable in s as:

$$Q(a,s) = c(a,s) + \sum_{s' \in S} P_a(s'|s)V(s')$$

initializing $V(s')$ to $h(s')$ when s' not in the table.

2. **Apply** action $\mathbf{a}$ with minimum $Q(\mathbf{a}, s)$ value, breaking ties randomly
3. **Update** $V(s)$ to $Q(\mathbf{a}, s)$
4. **Generate** successor state s' with probability $P_{\mathbf{a}}(s'|s)$
5. **Exit** if s' is a goal, else set s to s' and go to 1

Figure 5: Trial of RTDP algorithm

Provided that the goal is reachable from every state with positive probability, this value function is well-defined and can be computed by performing updates of the form (see [6])

$$V(s) := \min_{a \in A(s)} [c(a,s) + \sum_{s' \in S} P_a(s'|s)V(s')] \tag{10}$$

If the updates are performed over all states in parallel, the result is the familiar value iteration algorithm. On the other hand, if the updates are performed on the states visited by a greedy search guided by the value function V, the result is the probabilistic version of LRTA*, called *real time dynamic programming* (RTDP) in [3]. The resulting algorithm is shown in Fig. 5. Note that due to the inclusion of probabilistic actions, the state s' that follows a given action a in state s is generated with probability $P_a(s'|s)$.[2]

RTDP is an 'anytime' algorithm for solving MDPs. At any one point, the hash table and the heuristic function encode a value function $V(s)$[3] that determines a *greedy policy*. This greedy policy is given by the first two steps in Fig. 5. For RTDP, the same LRTA* properties apply; if the heuristic h is non-overestimating and there is a path (with positive probability) from every state to the goal, RTDP will eventually find the goal in every trial, and after successive trials, it will eventually deliver an optimal policy [3, 6]. Up to that point, the greedy policy normally improves monotonically at a rate that has to do with the size of the state space and the quality of the heuristic.

[2] If the algorithm is applied to a real or simulated system, the successor state s' is not generated but is *observed*.

[3] This value function is assumed equal to $h(s)$ when there is no entry for s in the hash table.

3.5 RTDP for POMDPs

The most common way to solve POMDPs is by formulating them as completely observable MDPs over *belief states* [57, 17], where belief states are probability distributions over the real states s in S. Given that the current belief state is b and action a is performed resulting in the observation o, the *revised* belief state b_a^o can be computed using the action and sensor models, and Bayes' rule as [17]:

$$b_a^o(s) = P_a(o|s)b_a(s)/b_a(o) \quad \text{provided } b_a(o) \neq 0 \tag{11}$$

where $b_a(s)$ and $b_a(o)$ refer to the probabilities that the next state is s and the next observation is o. These probabilities are given by

$$b_a(s) = \sum_{s' \in S} P_a(s|s')b(s') \tag{12}$$

$$b_a(o) = \sum_{s \in S} P_a(o|s)b_a(s) \tag{13}$$

The transformation of a POMDP into a belief MDP maps the *partially observable* problem of going from an initial state to a goal state into the *completely observable* problem of going from one *initial belief state* to a *final belief state*. The Bellman equation for the resulting *belief* MDP is

$$V^*(b) = \min_{a \in A(b)} [c(a,b) + \sum_{o \in O} b_a(o)V^*(b_a^o)] \tag{14}$$

where $c(a,b)$ is the average cost of doing action a in b

$$c(a,b) = \sum_{s \in S} c(a,s)b(s) \tag{15}$$

and $A(b)$ is the set of actions a that are applicable in the belief state b, defined as

$$A(b) = \{a \in A \mid a \in A(s) \text{ for all } \ s \text{ such that } b(s) > 0\} \tag{16}$$

Computationally, the belief MDP is difficult to solve optimally as the resulting state space, given the probability distributions over S, is infinite and continuous (unlike the finite-state MDPs we have considered so far). The known *optimal* POMDP algorithms can thus solve very small problems only (e.g., see [17]). Non-optimal methods, on the other hand, scale up better and often produce reasonable results for non-trivial problems [33, 31, 59, 11]. Here we present an algorithm that is the adaptation of RTDP for solving belief MDPs. We call such algorithm RTDP-BEL and show it in Fig. 6.

Details on RTDP-BEL can be found in [11, 9, 10] where the algorithm is used to solve a variety of problems: planning problems with sensing, robot navigation problems, and problems of sorting and classification.

A key idea in RTDP-BEL, as in general heuristic search algorithms, is the use of an heuristic function $h(b)$ to guide the search in belief space, focusing the

1. **Evaluate** each action a applicable in b as

$$Q(a,b) = c(a,b) + \sum_{o \in O} b_a(o) V(b_a^o)$$

initializing $V(b_a^o)$ to $h(b_a^o)$ when b_a^o not in table
2. **Apply** action **a** that minimizes $Q(a,b)$ breaking ties randomly
3. **Update** $V(b)$ to $Q(\mathbf{a}, b)$
4. **Generate** o with probability $b_a(o)$
5. **Compute** b_a^o
6. **Exit** if b_a^o is a goal (belief) state, else set b to b_a^o and go to 1

Figure 6: RTDP-BEL: RTDP algorithm for POMDPs

updates on the (belief) states that are most relevant. A common choice for $h(b)$ is the heuristic h_{mdp} defined in terms of the optimal cost function V^*_{mdp} of the underlying MDP

$$h_{mdp}(b) \stackrel{\text{def}}{=} \sum_{s \in S} V^*_{mdp}(s)\ b(s) \tag{17}$$

This heuristic presumes that the problem becomes fully observable after performing the next action. While this assumption is false, the resulting heuristic is non-overestimating and in certain problems can be quite informative.

When sensing is noisy, the belief space needs to be discretized. The discretization converts the continuous and infinite belief space into a discrete and finite grid of beliefs [43, 33]. While traditional methods were restricted to coarse grids which often produce poor approximations, algorithms like RTDP-BEL can deal with finer grids due to their ability to focus the updates on the relevant parts of the grid. See [11] for details.

3.6 RTDP for Non-Deterministic Control Problems

The RTDP algorithm for MDPs and POMDPs can easily be extended to non-deterministic MDPs and POMDPs where transition probabilities are replaced by non-deterministic transition functions and expected costs are replaced by worst case costs (Sect. 2.4 and [34, 38]). For non-deterministic *fully observable* problems, i.e., non-deterministic MDPs, the Bellman equation becomes:

$$V^*(s) = \min_{a \in A(s)} [c(a,s) + \max_{s' \in F(a,s)} V^*(s')] \tag{18}$$

where $F(a,s)$ is the non-deterministic function mapping action and states into *sets* of states.

For non-deterministic *partially observable* problems, i.e., non-deterministic POMDPs, the Bellman equation becomes

$$V^*(b) = \min_{a \in A(b)} [c(a,b) + \max_{o \in O} V^*(b_a^o)] \tag{19}$$

where the belief states b and b_a^o are now *sets* of states, and b_a^o is the set of states b_a resulting from doing action a in b filtered with the observation o:

$$\begin{aligned} b_a &= \{s' \in S \mid s' \in F(a,s) \text{ for } s \in b\} && (20) \\ b_a^o &= \{s' \in b_a \mid o \in O(a,s')\} && (21) \end{aligned}$$

Here $O(a,s)$ stands for the non-deterministic sensor model which takes the place of the probabilistic sensor model $P_a(o|s)$. For each action a and state s, $O(a,s)$ stands for the observations that are possible in state s after having done action a.

Using the updates corresponding to the above Bellman equations, the RTDP and RTDP-BEL algorithms can be used for solving non-deterministic control problems with full or partial observability.

4 Language

Deterministic control problems, MDPs, and POMDPs, as well as their non-deterministic counterparts, are useful *models* for making explicit the mathematical structure of a wide class of planning problems. Often, however, they are are not good *languages* for describing them. This is due the number and size of the relations and parameters involved. In AI, it has been common to describe planning problems compactly in terms of *modular* and *high-level* languages such as Strips [25]. In recent years similar languages have been defined for describing probabilistic actions [41, 22] and general POMDPs [9]. We'll illustrate the latter with a problem of planning with incomplete information from [42].

4.1 Example

The problem involves an agent that has a large supply of eggs and whose goal is to get three good eggs and no bad ones into one of two bowls. The eggs can be either good or bad, and at any time the agent can find out whether a bowl contains a bad egg by inspecting the bowl. In [9] this problem is encoded by expressions such as the ones in Figs. 7 and 8, which are compiled into a POMDP and solved by the RTDP-BEL algorithm.

The language illustrated in Figs. 7 and 8 extends Strips in several ways: states are not associated with sets of atoms but with assignments to arbitrary fluents; probabilities, costs and primitive operations like '+' are included, and a special predicate **obs** is used to indicate observability. The fluents in this problem are the number of good and bad eggs in each bowl (*ngood*, *nbad*), and the boolean variables *holding*? and *good*? that represent whether the agent is

Domain:	$BOWL: small, large$
Types:	$ngood(BOWL)$, $nbad(BOWL) : Int$
	$holding$, $good? : Bool$
Init:	$ngood(small) = 0$, $nbad(small) = 0$
	$ngood(large) = 0$, $nbad(large) = 0$
Goal:	$ngood(large) = 3$, $nbad(large) = 0$

Figure 7: Representation of Omelette Problem (part 1)

holding an egg and whether such an egg is good or bad. The fluent *holding* is always observable, but the value of the expression $nbad(bowl) > 0$ is observable after doing the action $inspect(bowl)$ only. The formal syntax and semantics of the language, can be found in [9]. We provide the main ideas below.

4.2 Language and States

The *language* is a typed logical language that involves a number of constant, function, and predicate symbols from which atoms, terms, and formulas are defined in the standard way. For example, $(ngood(bowl) + nbad(bowl)) < 4$ is a formula expressing that the total number of eggs in *bowl* is less than 4.

Given a language with the relevant type and domain declarations, the *states* are the *logical interpretations* over such language. That is, a state s assigns a denotation x^s to any symbol x from which the denotation of all terms, atoms, and formulas is obtained following the standard composition rules. Symbols like '$<$', 4, and others, have a denotation that is fixed and is independent of the state. States thus have to assign a denotation to *fluent* symbols only; symbols like *ngood*, *nbad*, etc. *Type* and *domain* declarations for these symbols define their possible set of denotations (values) and all together implicitly define the *state space.* Then action *preconditions* define the set $A(s)$ of actions applicable in each state s (the actions whose preconditions have a true denotation in s) and action *effects* define the state *transition function* or transition *probabilities.*

Assuming that the cost of all actions is 1, such a language can be used to define state models and MDPs in a compact way. For describing POMDPs, it's necessary to describe also what is *observable.* That's the role of the special expressions like $\mathbf{obs}(nbad(bowl) > 0)$ in Fig. 8. An action a that makes the expression $\mathbf{obs}(x)$ true for a term or formula x produces observations $o : (x = v)$ for each possible denotation v of x with probabilities $b_a(o)$ that depend on the belief state b where the action a was done (see Equation 11).

The language also provides facilities for expressing deterministic or probabilistic *ramification* rules. While action rules express the value of a variable in terms of the value of variables at the *previous* time, ramification rules express the value of a variable as a function of the value of variables *at the same time point.* Ramification rules are useful in a number of circumstances. For example, it's possible to express that a sensing action reports the true value of a boolean

Action:	**grab-egg**()
Precond:	$\neg holding$
Effects:	$holding :=$ **true**
	$good? :=$ (**true** 0.5 ; **false** 0.5)
Action:	**break-egg**($bowl : BOWL$)
Precond:	$holding \wedge (ngood(bowl) + nbad(bowl)) < 4$
Effects:	$holding :=$ **false**
	$good? \;\rightarrow\; ngood(bowl) := ngood(bowl) + 1$
	$\neg good? \;\rightarrow\; nbad(bowl) := nbad(bowl) + 1$
Action:	**pour**($b1 : BOWL, b2 : BOWL$)
Precond:	$(b1 \neq b2) \wedge \neg holding$
	$ngood(b1) + nbad(b1) + ngood(b2) + nbad(b2) < 4$
Effects:	$ngood(b1) := 0 \;\;,\;\; nbad(b1) := 0$
	$ngood(b2) := ngood(b2) + ngood(b1)$
	$nbad(b2) := nbad(b2) + nbad(b1)$
Action:	**clean**(bowl:BOWL)
Precond:	$\neg holding$
Effects:	$ngood(bowl) := 0 \;\;, nbad(bowl) := 0$
Action:	**inspect**($bowl : BOWL$)
Effect:	**obs**($nbad(bowl) > 0$)
Action:	**all** (all actions)
Effect:	**obs**($holding$)

Figure 8: Representation of Omelette Problem (part 2)

variable x with 0.9 probability by making the sensor report the true value of a dummy variable y that with 0.9 probability has the same value as x. This is a general way for encoding arbitrary sensor models in the language.

4.3 Language and Models

The logical representation language plays two roles in this setting. One is as a convenient *front-end* for describing state models, MDPs, and POMDPs. This is essential in complex domains where providing the state space and the action and sensor models *explicitly* is often infeasible. However, there is a second role for representation languages that is not as widely recognized. It's a way for making explicit the structure of the problem and the relations among the variables of interest so that they can be exploited *computationally.* For example, [13] describes an LRTA* planner that, given the Strips encoding of a problem, automatically extracts an heuristic function h and uses such function to guide the search. The same idea is used in the HSP planner entered into the AIPS98 Planning Contest [12]. In neither case, this would have been possible if the Strips description of the actions had been replaced by an equivalent state-transition function. High-level descriptions of planning problems are not thus just a modeling convenience; they can be very useful computationally. A similar point has been made about the use of factored models (like Bayesian Networks) for representing decision-theoretic planning problems [14, 15].

5 Results

We have developed a computational tool that accepts high-level description of planning problems with various types of actions dynamics and feedback, and computes the resulting controllers. The controllers are obtained by solving the corresponding mathematical model with a suitable version of the RTDP algorithm and a heuristic function extracted from the description of the problem. We have modeled and solved a number of problems with this tool: problems of classical planning, problems of planning with sensing, robot navigation problems, and sorting and classification problems. Many of these results are reported in [13, 9, 11, 10]. Some of these results are summarized below.

5.1 Classical Planning

Tables 9 and 10 show results from [13] comparing RTDP with two recent planners, GRAPHPLAN [8] and SATPLAN [36].[4] The heuristic used is crucial to the performance of RTDP in this setting. The heuristic is extracted automatically from the Strips representation of the planning problems. The heuristic is very informative but is *not* admissible. For this reason the algorithm does not improve much after successive trials, and hence only a *single* RTDP *trial* is considered.

As it can be seen from the tables, RTDP reaches the goal very fast with times that are competitive with GRAPHPLAN and SATPLAN(Fig. 9). On the other hand, the length of plans is sometimes far from optimal (Fig. 10). One way to decrease this length is to run the algorithm several times keeping only the best run. Other methods considered in [13] are the addition of 'noise' in the selection of actions and increased lookahead. More recent work along the line of the RTDP planner can be found in [12].

5.2 Planning with Incomplete Information

5.2.1 Assembly

We consider now a problem from [24] which deals with a robot that has to decide whether to dispatch or reject pieces that come on an assembly line. The pieces are supposed to be dispatched or rejected according to whether they are believed to be flawed or not. Dispatched pieces have to be painted first. The robot cannot observe directly whether a piece is flawed or not, but can sense

[4] The algorithm in [13] is referred to as ASP for Action Selection for Planning, and is presented as a variation of Korf's LRTA* [39] which is the deterministic version of RTDP. The results for RTDP presented here are slightly different from those in [13] which involve some lookahead before moves.

Problem	GRAPHPLAN	SATPLAN	RTDP
rocket_ext.a	268	0.17	1.3
logistics.a	5,942	22	7
logistics.b	2,538	6	6
12 blocks	1,119	18	1
15 blocks	—	524	5
19 blocks	—	4,220	19

Figure 9: Time performance in seconds for RTDP in comparison with GRAPHPLAN and SATPLAN from [13]

Problem	GRAPHPLAN	SATPLAN	RTDP
rocket_ext.a	34	34	35/28
logistics.a	54	54	64/57
logistics.b	47	47	58/48
12 blocks	9	9	16/12
15 blocks	—	14	24/19
19 blocks	—	18	32/25

Figure 10: Quality performance. Average and minimal plan lengths over 25 runs from [13]

with a probability of error p whether the piece is blemished. The robot also knows that a flawed piece will appear as blemished as long as it's not painted, and once painted, the piece will appear as not blemished.

This problem can be expressed compactly in the language described above, resulting in a small POMDP with 16 states and 4 actions (*paint*, *dispatch*, *reject*, and *inspect*). The performance of the controller obtained by the RTDP-BEL algorithm is shown in Fig. 11, where the vertical axis displays the average cost to the goal over 10 simulations as a function of the number of trials. The different curves correspond to different values of the parameter p that measures the accuracy of the sensor. The more noisy the sensor, the longer it takes RTDP-BEL to converge, and the more costly the resulting policy. This is because with more noise it is necessary to sense more times the piece before making the accept/reject decision. Sensor readings are assumed independent of each other, but a different assumption could be accommodated as well (using a different action description). The average time to complete 50 trials is in the order of 0.5 seconds.

5.2.2 Information Gathering

This is a navigation problem over the grid showed in Fig. 12 suggested by Sebastian Thrun. An agent starts in position 6 in the grid and has to reach a *goal* that is at position 0 or 4. The position that is not the goal is a high penalty

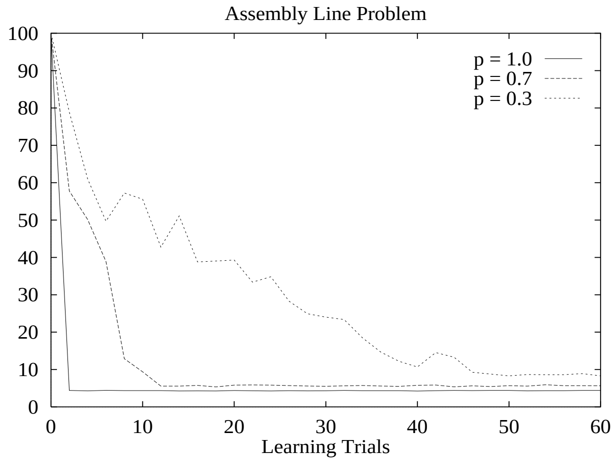

Figure 11: Assembly problem: performance of controllers derived by RTDP. Curve shows average cost to goal as a function of the accuracy of the sensor and the number of trials

state. At position 9 there is a sensor that reports the true position of the goal with probability p. When $p = 1$, the optimal solution is to go to position 9, 'read' the sensor once, and head up for the goal. We call this the 'reference policy' for the problem. Its performance for the case in which $p = 1$ is shown by the the flat curve in Fig. 13. The other curves show the performance of the controllers derived by RTDP-BEL for different values of p. When $p < 1$, the agent has to stay longer in 9 accumulating information from the sensor, thus the expected cost of the optimal policy in that case is higher.

The problem is encoded in the language described above and is compiled into a POMDP that contains 20 states and 4 actions.

Figure 13 shows the performance of the resulting RTDP-BEL controllers as a function of the number of trials and the level of noise in the sensor. The average costs were obtained by running 10 controllers in 100 simulations every 5 trials. The average time to compute 60 trials is in the order of 1 second for the different values of p.

5.2.3 Omelette Problem

Figures 14 displays the performance of the controller derived for the Omelette Problem discussed above. The resulting POMDP involves 356 states, 11 actions, and 6 observations. The curve that is flat shows the average number of actions to solve the problem for the 'reference' policy in which an egg is grabbed and broken it into one of the two bowls, and after inspecting the bowl, it's either passed to the other bowl or discarded, until three eggs have been passed. The other curve shows the performance of the greedy controller resulting from RTDP-BEL as a function of the number of trials.

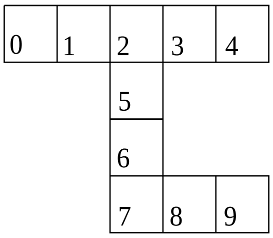

Figure 12: Information Gathering problem: agent is originally at position 6. The goal is either at position 0 or 4, and the other state carries a high-penalty. A sensor at position 9 provides information about the position of the goal.

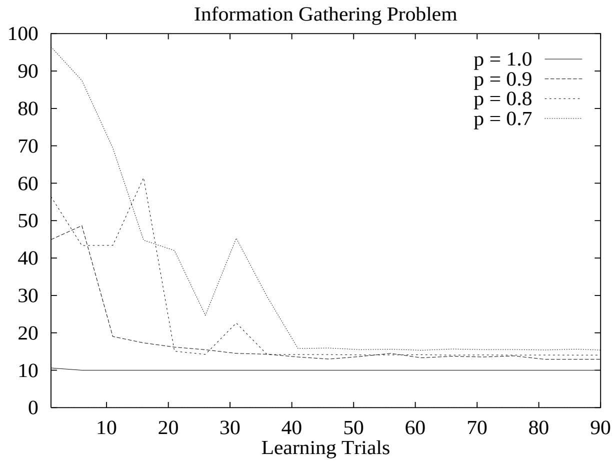

Figure 13: Information Gathering problem: performance of controllers derived by RTDP-BEL as a function of the number of trials and accuracy of the sensor

The convergence takes more than 1000 trials as the algorithm has to 'learn' the value of the action 'inspect', which as all information-gathering actions, appears useless to the h_{mdp} heuristic. The time for 2000 trials is in the order of 192 seconds on an UltraSparc running at 143Mhz.

5.3 Sorting Problems

The last test domain we discuss is the problem of sorting a vector of n numbers in increasing order. This problem, and the related problem of inferring decision trees from data, are formulated as POMDPs and solved by the RTDP-BEL algorithm in [10].

In sorting, there are two types of actions: 'physical' actions such as $swap(i, j)$ that exchange the elements in positions i and j, and 'information gathering'

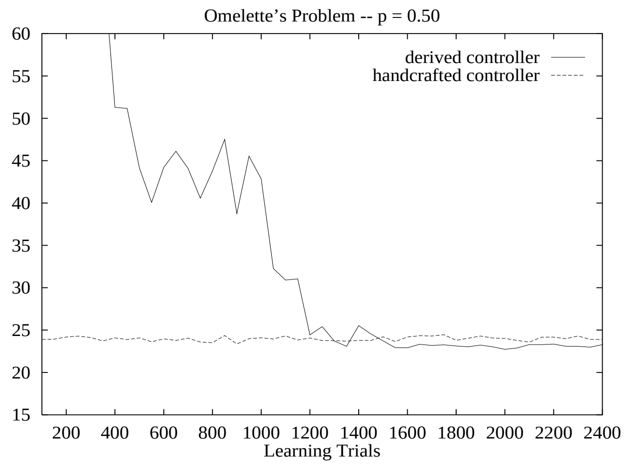

Figure 14: Comparison of RTDP-BEL policy vs. handcrafted policiy for the Omelette Problem

actions such as $cmp(i, j)$ that test whether the element in position i is smaller than the element in position j. The problem is a convenient benchmark for algorithms that perform planning with sensing as it is non-trivial and there are a number of well known sorting algorithms for assessing the quality of the solutions.

In the POMDP formulation, the state s is taken to be a vector of size n with $s[i] = j$ meaning that the i-th element of the input vector is the j-th smallest element. We assume that all elements in the input vector are different, and as a result, there is a single goal state s_G for which $s_G[i] = i$. The problem is to devise a policy of swaps and compares that takes an arbitrary and *unknown* input state s_0 into the goal state s_G. The number of states in the problem is $n!$. The initial belief state b_0 is *uniform* over all such states, and the goal belief state b_F is such that $b(s_G) = 1$.

The rest of the formulation is straightforward. Fig. 15 from [10] shows the performance of the RTDP-BEL algorithm for the sorting problem with $n = 5$. This means a POMDP with $5! = 120$ states, 20 actions, and 40 observations. The figure shows the average number of swaps and comparisons in the policies computed by RTDP-BEL with different heuristics, and compares them with the results obtained with Quicksort (top flat curve). In all cases, and in particular for the non-informative heuristic $h = 0$, RTDP-BEL produces better policies than Quicksort after a sufficiently large number of trials (3000) that take in the order of 45 seconds. On the other hand, the policies computed by RTDP-BEL are fixed for $n = 5$ and do not scale up for large values of n. Indeed, for $n = 10$ suitable heuristics and belief representations are needed (see [10]). Even then, scaling for larger values of n remains hard as the branching factor of the problem grows with n. In any case, if optimal policies could be derived for values of n as low

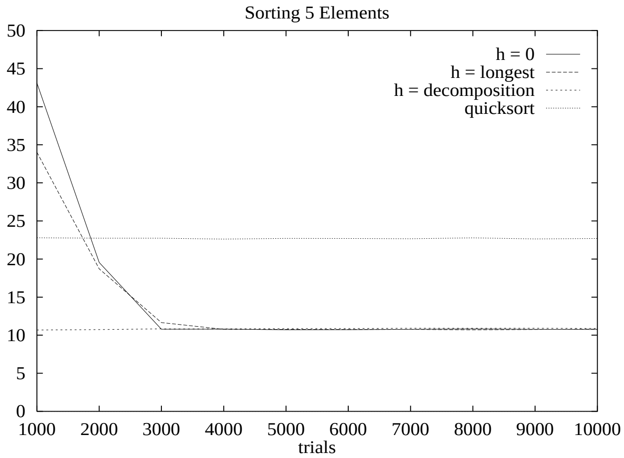

Figure 15: Performance of RTDP-BEL controller for Sorting problem with $n = 5$ using various heuristics. Top flat curve is for Quicksort. From [10]

as 13 some of the open conjectures in lower-bound theory could be settled (see [37]).

6 Discussion

We have presented a unified approach for modeling and solving planning problems that is based on *state models* that handle various types of *dynamics* (deterministic, non-deterministic, and probabilistic) and *sensor feedback* (null, partial, and complete). The approach combines *logical representations languages* for describing actions, sensors, and goals, *mathematical models of sequential decisions* for making precise the various planning tasks and their solutions, and heuristic search algorithms in the form of *real-time dynamic programming* procedures for computing those solutions. The approach is supported by a computational tool that accepts high-level descriptions of actions, sensors and goals and computes the resulting controllers. We have also presented results over a number of domains that illustrate the scope of the approach and the capabilities of the tool.

The planning approach described is a natural integration of a number of ideas from AI and Dynamic Programming: State models, MDPs, and POMDPs, [4, 57, 17], RTDP algorithms [39, 3], and action representation languages [29, 52, 54]. At the same time, it's related to a number of *decision-theoretic approaches to planning* such as [41, 24, 14], while being distinct in the use of a more expressive *logical* language for describing actions and sensors, and the use of the RTDP algorithm for solving a variety of decision models.

The challenge remains to scale up these methods to larger problems. As for any heuristic search algorithm, four key factors have a strong influence on the performance of the RTDP algorithm:

1. the node generation rate
2. the quality of the heuristic function
3. the use of memory
4. the exploitation of symmetries

The node generation rate is particularly critical in POMDPs where going from one state b to the next b_a has complexity $|S|^2$, where $|S|$ is the size of the state space. For this, specialized representations can help. For example, in [10] belief states are represented as graphs and the mapping from b to b_a takes roughly constant time. Similarly, in non-deterministic domains, OBDDs (ordered binary decision diagrams) have been used quite successfully for reducing both time and space [18, 30].

For keeping time and memory requirements under control, the use of *good heuristics* is critical. The domain-independent heuristic in [13] for Strips planning is quite informative but is not admissible. On the other hand, the heuristic h_{mdp} for POMDPs derived from the underlying MDPs is admissible but is not sufficiently informative. In both cases, it's likely that better general heuristics are to be found.

Finally, the exploitation of symmetries is a familiar theme in heuristic search (e.g., [40]) but not so much in planning (although see [26]). Yet symmetries abound, and if not detected, they can make the state spaces blow. Indeed, while the presence of more resources should make planning problems simpler (e.g., more trucks in the 'logistics' domains), for most planners, they make it more complex. Many times these symmetries can be exploited at modeling time (e.g., by representing resources by numbers and not by individual names), yet other times they have to be detected at run time. For example, in the 12-coin problem [49], where a heavier or lighter coin is to be identified from a set of 12-coins using a two-pan scale, initially all coins are symmetrical, yet they are not symmetrical after the first weighting. This is a typical problem of planning with sensing, yet without recognizing these changing symmetries it's unlikely to solved with domain-independent planning tools.

The techniques discussed for making RTDP algorithms scale to larger problems arise form viewing RTDP as a *heuristic search* algorithm. RTDP, however, can also be viewed as a *dynamic programming* algorithm. From that perspective, it makes sense to focus on the *value function* and ways for making the RTDP updates more effective. One way to do this is by *collapsing* states that have similar costs. This is a form of *ideal* state aggregation in which the dimensionality of the problem is reduced, gaining both in time and space. Since normally the cluster of states with similar costs is not known a priori, an alternative that is often used in Reinforcement Learning [58, 6] is to represent the value function in parametric form with a number of parameters that is smaller than the number of states (e.g., linear functions, neural networks, etc). The value function is then updated by adjusting those parameters. Under suitable conditions, if the parametric form is adequate, compact representations can approximate the effect of the ideal form of state aggregation (see [6]). Other approaches attempt to exploit the structure of problems through suitable modifications of the dynamic programming algorithms [15, 14]. Representing and using the structure of problems for extracting heuristics and speeding up the computation of plans is soon becoming a central problem in planning. See the forthcoming Workshop on *Analyzing and Exploiting Domain Knowledge for Efficient Planning*, at the AIPS'2000 Conference.

Acknowledgements. H. Geffner wants to thank Jerome Lang and Helder Coelho for inviting him to deliver talks on this topic at the ECAI'98 Workshop on Decision-Theoretic Planning in Brighton and at the 1998 Iberoamerican Conference on AI in Lisbon. This paper is based on those talks and is an extended version of [27]. Partial support for this work is due to Conicit, Grant S1-96001365.

References

[1] P. Agre and D. Chapman. What are plans for? *Robotics and Autonomous Systems*, 6:17–34, 1990.

[2] K. Astrom. Optimal control of markov decision processes with incomplete state estimation. *J. Math. Anal. Appl.*, 10:174–205, 1965.

[3] A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[4] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[5] D. Bertsekas. *Dynamic Programming and Optimal Control, Vols 1 and 2.* Athena Scientific, 1995.

[6] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, 1996.

[7] C. Bishop. *Neural Networks and Pattern Recognition.* Oxford University Press, 1995.

[8] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of IJCAI-95*, Montreal, Canada, 1995.

[9] B. Bonet and H. Geffner. High-level planning and control with incomplete information using POMDPs. In *Proceedings AAAI Fall Symp. on Cognitive Robotics*, 1998.

[10] B. Bonet and H. Geffner. Learning sorting and decision trees with POMDPs. In *Proceedings ICML-98*, 1998.

[11] B. Bonet and H. Geffner. Solving large POMDPs using real time dynamic programming. In *Proc. AAAI Fall Symp. on POMDPs*, 1998.

[12] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proceedings of ECP-99*. Springer, 1999.

[13] B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pages 714–719. MIT Press, 1997.

[14] C. Boutilier, T. Dean, and S. Hanks. Planning under uncertainty: structural assumptions and computational leverage. In *Proceedings of EWSP-95*, 1995.

[15] C. Boutilier, R. Dearden, and M. Goldszmidt. Exploiting structure in policy construction. In *Proceedings of IJCAI-95*, 1995.

[16] R. Brooks. A robust layered control system for a mobile robot. *IEEE J. of Robotics and Automation*, 2:14–27, 1987.

[17] A. Cassandra, L. Kaebling, and M. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings AAAI94*, pages 1023–1028, 1994.

[18] A. Cimatti and M. Roveri. Conformant planning via model checking. In *Proceedings of ECP-99*. Springer, 1999.

[19] G. Collins and L. Pryor. Planning under uncertainty: Some key issues. In *Proceedings IJCAI95*, 1995.

[20] C. Anderson D. Weld and D. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proc. AAAI-98*, pages 897–904. AAAI Press, 1998.

[21] T. Dean, L. Kaebling, J. Kirman, and A. Nicholson. Planning with deadlines in stochastic domains. In *Proceedings AAAI93*, pages 574–579. MIT Press, 1993.

[22] T. Dean and K. Kanazawa. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(3):142–150, 1989.

[23] T. Dean and M. Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

[24] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proceedings of the Second Int. Conference on Artificial Intelligence Planning Systems*, pages 31–36. AAAI Press, 1994.

[25] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 1:27–120, 1971.

[26] M. Fox and D. Long. The detection and exploitation of symmetry in planning domains. In *Proc. IJCAI-99*, 1999.

[27] H. Geffner. Modelling intelligent behaviour: The MDP approach. In H. Coelho, editor, *Lecture Notes in AI*, volume 1484. Springer, 1998.

[28] H. Geffner and J. Wainer. Modeling action, knowledge and control. In *Proceedings ECAI-98*, 1998.

[29] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *J. of Logic Programming*, 17:301–322, 1993.

[30] F. Giunchiglia and P. Traverso. Planning as model checking. In *Proceedings of ECP-99*. Springer, 1999.

[31] E. Hansen. Solving pomdps by searching in policy space. In *Proceedings UAI-98*. Morgan Kauffman, 1998.

[32] E. Hansen and S. Zilberstein. Heuristic search in cyclic AND/OR graphs. In *Proc. AAAI-98*, pages 412–418, 1998.

[33] M. Hauskrecht. *Planning and Control in Stochastic Domains with Incomplete Information*. PhD thesis, MIT, 1997.

[34] M. Heger. Consideration of risk in reinforcement learning. In *Proceedings of the Int. Conf. on Machine Learning*, pages 105–111, 1994.

[35] L. Kaebling, M. Littman, and T. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99–134, 1998.

[36] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*, pages 1194–1201, 1996.

[37] D. Knuth. *The Art of Computer Programming, Vol. III: Sorting and Searching*. Addison-Wesley, 1973.

[38] S. Koenig and R. Simmons. Real-time search in non-deterministic domains. In *Proceedings IJCAI-95*, pages 1660–1667. Morgan Kaufmann, 1995.

[39] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

[40] R. Korf. Finding optimal solutions to to Rubik's cube using pattern databases. In *Proceedings of AAAI-98*, pages 1202–1207, 1998.

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

[42] H. Levesque. What is planning in the presence of sensing. In *Proceedings AAAI-96*, pages 1139–1146, Portland, Oregon, 1996. MIT Press.

[43] W. Lovejoy. Computationally feasible bounds for partially observed markov decision processes. *Operations Research*, pages 162–175, 1991.

[44] D. McDermott. AIPS-98 Planning Competition Results. `http://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html`, 1998.

[45] A. Newell and H. Simon. *Human Problem Solving.* Prentice–Hall, Englewood Cliffs, NJ, 1972.

[46] N. Nilsson. *Principles of Artificial Intelligence.* Tioga, 1980.

[47] N. Nilsson. Teleo-reactive programs for agent control. *JAIR*, 1:139–158, 1994.

[48] L. Padulo and M. Arbib. *System Theory.* Hemisphere Publishing Co., 1974.

[49] J. Pearl. *Heuristics.* Morgan Kaufmann, 1983.

[50] E. Pednault. ADL: Exploring the middle ground between Strips and the situation calcules. In *Proc. KR-89*, pages 324–332, 1989.

[51] M. Puterman. *Markov Decision Processes – Discrete Stochastic Dynamic Programming.* John Wiley and Sons, Inc., 1994.

[52] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[53] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1994.

[54] E. Sandewall. *Features and Fluents. The Representation of Knowledge about Dynamical Systems.* Oxford Univ. Press, 1994.

[55] M. J. Shoppers. Universal plans for reactive robots in unpredictable environments. In *Proc. IJCAI-87*, pages 1039–1046, 1987.

[56] D. Smith and D. Weld. Conformant graphplan. In *Proceedings AAAI-98*, pages 889–896. AAAI Press, 1998.

[57] E. Sondik. *The Optimal Control of Partially Observable Markov Processes.* PhD thesis, Stanford University, 1971.

[58] R. Sutton and A. Barto. *Introduction to Reinforcement Learning.* MIT Press, 1998.

[59] R. Washington. BI-POMDP: Bounded, incremental partially-observable Markov model planning. In *Proc. 4th European Conf. on Planning*, volume LNAI 1248. Springer, 1997.