Learning sorting and decision trees with POMDPs

Blai Bonet
Departamento de Computación
Universidad Simón Bolívar
Aptdo. 89000, Caracas 1080-A
Venezuela
bonet@usb.ve

Héctor Geffner
Departamento de Computación
Universidad Simón Bolívar
Aptdo. 89000, Caracas 1080-A
Venezuela
hector@usb.ve

Abstract

POMDPs are general models of sequential decisions in which both actions and observations can be probabilistic. Many problems of interest can be formulated as POMDPs, yet the use of POMDPs has been limited by the lack of effective algorithms. Recently this has started to change and a number of problems such as robot navigation and planning are beginning to be formulated and solved as POMDPs. The advantage of the POMDP approach is its clean semantics and its ability to produce principled solutions that integrate physical and information gathering actions. In this paper we pursue this approach in the context of two learning tasks: learning to sort a vector of numbers and learning decision trees from data. Both problems are formulated as POMDPs and solved by a general POMDP algorithm. The main lessons and results are that 1) the use of suitable heuristics and representations allows for the solution of sorting and classification POMDPs of non-trivial sizes, 2) the quality of the resulting solutions are competitive with the best algorithms, and 3) problematic aspects in decision tree learning such as test and misclassification costs, noisy tests, and missing values are naturally accommodated.

1 INTRODUCTION

POMDPs are general models of sequential decisions in which both actions and observations can be probabilistic (Sondik 1971; Cassandra, Kaebling, & Littman 1994). Many problems of interest can be formulated as POMDPs yet the use of POMDPs has been limited by the lack of effective algorithms (Cassandra, Kaebling, & Littman 1995). Recently this has started to change and a number of problems such as robot navigation and planning are beginning to be formulated and solved as POMDPs (Cassandra, Kaebling, & Kurien 1996; Geffner & Bonet 1998a). The advantage of the POMDP approach is its clean semantics and its ability to produce principled solutions that integrate physical and information gathering actions. In this paper we pursue this approach in the context of two learning tasks: learning to sort a vector of numbers and learning decision trees from data. Both problems are formulated as POMDPs and solved by a general POMDP algorithm (Geffner & Bonet 1998b) based on the ideas of Real Time Dynamic Programming (Barto, Bradtke, & Singh 1995).

The choice of the two tasks requires an explanation. Both are sequential decision problems that can be naturally seen as POMDPs. Yet the difficulties and insights that result from modeling and solving each problem as a POMDP are different. Sorting involves finding a sequence of comparisons and swaps that would sort *any* vector of size n. This is a challenging *planning* problem and we are not aware of any contingent planner that can model and solve problems of this type. Modeling and solving the problem from the perspective of POMDPs is challenging too. For $n = 10$, the number of possible states in the problem is greater than 10^6. Until recently POMDPs with more than 20 states could not be reasonably solved, especially when they involved information-gathering actions. Here we provide solutions for POMDPs of size $n = 10$ that involve more than a million states. Moreover the solutions are good: on average they involve half the number of comparisons and swaps as Quicksort, one of the best sorting algorithms (Aho, Hopcroft, & Ullman 1983). The solution method relies on good heuristic functions, compact representations of beliefs, and suitable decompositions.

The sorting problem is difficult and we use it not to learn about sorting but to learn about POMDPs. The focus on decision tree induction is different as we expect that the POMDP approach may contribute to a better understanding of decision tree induction on aspects such as noisy data and tests, missing values, and tests and misclassification costs. All these aspects fit into the POMDP formulation of decision tree learning in a natural way. We evaluate this formulation over a number of datasets from (Murphy & Aha 1998). Our goal is to show that the POMDP approach may be competitive with the standard approaches and potentially more general. Indeed POMDPs provide a unifying framework for modeling and solving not only sorting and induction, but other AI tasks as well such as robot navigation, planning, control, diagnosis, etc. (Cassandra, Kaebling, & Littman 1994; Geffner & Bonet 1998a). On the other hand, the POMDPs algorithms we use do not scale up yet to learning problems over very large datasets.

The rest of the paper is organized as follows. First we review MDPs, POMDPs, and the POMDP algorithm (Sections 2 and 3). Then we formulate the problems of sorting and decision tree induction as POMDPs, and report empirical results (Sections 4 and 5). Finally we summarize the main lessons and ideas (Section 6).

2 BACKGROUND

POMDPs are a generalization of a model of sequential decision making formulated by Richard Bellman in the 50's called *Markov Decision Processes* or MDPs, in which the state of the environment is assumed known (Bellman 1957). MDPs provide the basis for understanding POMDPs so we turn to them first.[1]

2.1 MDPs

The type of MDPs that we consider is a generalization of the standard search model used in AI in which actions can have *probabilistic* effects. Goal MDPs, as we call them, are characterized by:

1. a state space S
2. actions $A(s) \subseteq A$ applicable in each state s
3. positive costs $c(a, s)$ of performing action a in s
4. transition probabilities $P_a(s'|s)$ of ending up in state s' after doing action $a \in A(s)$ in state s
5. goal states $G \subseteq S$

Since the effect of actions is assumed to be *observable* but not *predictable*, the solution of an MDP is not an action sequence but a function that maps states s into actions $a \in A(s)$. Such a function is called a *policy*, and its effect is to assign a probability to each state trajectory. We assume that goal states are *absorbing* in the sense that actions in those states have no effects and zero costs. As a result, state trajectories that contain goal states have finite costs, while others have infinite costs. The *expected cost* of a policy from an initial state is the weighted average of the costs of all the state trajectories starting in that state times their probability. A policy is *optimal* when its expected cost from any state is minimal. General conditions for the existence of such policies can be found in (Puterman 1994; Bertsekas & Tsitsiklis 1996).

3 POMDPs

POMDPs generalize MDPs allowing the state to be *partially* observable (Sondik 1971; Cassandra, Kaebling, & Littman 1994; Russell & Norvig 1994). The solution of a POMDP is no longer a mapping from states into actions, but a mapping from *belief states* into actions, where belief states are probability distributions over the states. A POMDP agent or controller starts with a prior belief state that adjusts as a result of the actions it performs and the observations it gathers. It is assumed that the agent has a model of both the actions and the sensors. Formally, a *goal* POMDP is defined in terms of:

1. **states** $s \in S$
2. **actions** $A(s) \subseteq A$ applicable in each state s
3. **positive costs** $c(a, s)$ of performing action a in s
4. **transition probabilities** $P_a(s'|s)$ of ending up in state s' after doing action $a \in A(s)$ in state s
5. **initial belief state** b_0
6. **final belief states** b_F
7. **observations** o in state s after action a with probabilities $P_a(o|s)$

The first four components define an MDP that is extended with prior and final beliefs, and a sensor model.

[1] For some recent books on MDPs, see (Puterman 1994; Bertsekas & Tsitsiklis 1996); for an AI perspective, see (Boutilier, Dean, & Hanks 1995; Barto, Bradtke, & Singh 1995).

POMDPs can be formulated as *information* or *belief* MDPs in which *states* are replaced by *belief states* (Sondik 1971; Cassandra, Kaebling, & Littman 1994). The task is to find a mapping π from belief states to actions that will take us from the initial belief state b_0 to a final belief state b_F at a minimum expected cost. The way actions and observations affect the *belief* state is given by the equations (Cassandra, Kaebling, & Littman 1994):

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

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

$$b_a^o(s) = P_a(o|s) b_a(s)/b_a(o) \quad \text{if } b_a(o) \neq 0 \quad (3)$$

where b_a is the belief state that results after doing action a in b, $b_a(o)$ is the probability of observing o after doing a in b, and b_a^o is the belief state that results after doing action a in b and then observing o. The cost $c(a, b)$ of an action a in b is the weighted average $\sum_{s \in S} c(a, s) b(s)$. The exception are the final belief states b_F that are assumed to be absorbing; i.e., $c(a, b_F)$ is defined as 0, and b_a and b_a^o are defined as b, when b is a final belief state. Finally, the set of actions $A(b)$ applicable in b excludes the actions a that are not applicable in states s with $b(s) > 0$.

Solving belief MDPs is difficult and until recently only very small problems could be solved reasonably well especially when they involved information-gathering actions. This has started to change (Cassandra, Kaebling, & Littman 1995) and here we use a POMDP algorithm introduced in (Geffner & Bonet 1998b) that is based on the ideas of Real Time Dynamic Programming (Barto, Bradtke, & Singh 1995).

RTDP-BEL is a *hill-climbing* algorithm that from any state b searches for the goal states b_F by performing actions a that lead to new states b_a^o with probability $b_a(o)$ (Figure 1). Estimates $V(b)$ of the expected costs to reach b_F guide the search. The main difference with standard hill-climbing is that these estimates are updated dynamically. Initially $V(b)$ is set to $h(b)$, where h is a suitable heuristic function, and every time the state b is visited $V(b)$ is updated to make it consistent with the values $V(b')$ of its possible successor states b' (Korf 1990). In the implementation, the estimates $V(b)$ are stored in a hash table that initially contains an estimate for $V(b_0)$ only. Then when the value $V(b')$ of a state b' that is not in the table is needed, a new entry with $V(b')$ set to $h(b')$ is created. Usually belief states need to be discretized (Geffner & Bonet 1998b)

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. **Observe** o
5. **Compute** b_a^o using Equations 1–3
6. **Exit** if b_a^o is a final belief state, else set b to b_a^o and go to 1

Figure 1: RTDP-BEL

but this is not needed in the tasks considered in this paper.

RTDP-BEL combines search and simulation, and in every trial selects a random initial state s with probability $b_0(s)$ on which the effects of the actions applied by RTDP-BEL (Step 2) are simulated. More precisely, when action a is chosen, the current state s in the simulation changes to s' with probability $P_a(s'|s)$ and then produces an observation o with probability $P_a(o|s')$. The complete RTDP-BEL algorithm is shown in Fig. 1.

4 SORTING

The sorting problem involves arranging a vector of numbers in increasing order. We simplify the problem slightly assuming that no two numbers in the vector are equal. There are two types of actions available: $swap(i, j)$ that exchanges the elements in positions i and j, and $cmp(i, j)$ that tests whether the element in position i is smaller than the element in position j. One of the best algorithms for sorting is Quicksort, which takes in the order of $n\,log(n)$ operations on average, where n is the size of the problem (the number of elements to be sorted).

4.1 FORMULATION

We formulate the problem as a goal POMDP in which we have to go from an initial belief state to a final belief state by means of a number of tests and swaps. The state s reflects the way in which the elements in the input vector may be ordered; for example, the *state* $s = [3, 1, 2]$ for $n = 3$ says that the first element in the input vector is the third smallest element, the second element is the smallest element of all, and the third element is the second smallest element. More generally, a state s will be a vector of size n such that $s[i] = j$, for $1 \leq i, j \leq n$ and $s[i] \neq s[j]$ for $i \neq j$. The meaning of $s[i] = j$ is that the i-th element in the input vector is the j-th smallest element.

Given an input vector, there is a single state that is the *true* state associated with the input vector and the swaps performed. The actions $cmp(i, j)$ yield information about such state and the actions $swap(i, j)$ mutate it. The resulting 'sorting' POMDP for a particular problem size n consists of:

1. **states** given by the vectors s of size n such that $s[i] = j$ for $0 \leq i, j \leq n$ and $s[i] \neq s[j]$ if $i \neq j$
2. **actions** $swap(i, j)$ and $cmp(i, j)$ for $0 \leq i < j \leq n$
3. **transition probabilities** $P_a(s'|s) = 1$ for $a = cmp(i, j)$ and $s' = s$, and $a = swap(i, j)$ and s' such that $s'[j] = s[i]$, $s'[i] = s[j]$, and $s'[k] = s[k]$ for $k \neq i$, $k \neq j$. Otherwise $P_a(s'|s) = 0$
4. **action costs** $c(a, s) = 1$ for all a and s
5. **initial belief state** b_0 uniform over all states
6. **final belief state** b_F for which $b_F(G) = 1$, where $s = G$ is the *sorted state* for which $s[i] = i$ for $i = 1, \ldots, n$
7. **observations** $o_1 = (i < j)$ or $o_2 = (j < i)$ from the actions $a = test(i, j)$ with probabilities $P_a(o_1|s)$ equal to 1 (0) when $s[i] < s[j]$ ($s[i] > s[j]$), and complementary probabilities for $P_a(o_2|s)$.

4.2 IMPLEMENTATION

Finding a policy to take us from b_0 to b_F at a nearly optimal expected cost is difficult, and for the RTDP-BEL algorithm to solve this problem for even small values of n, suitable belief representations and heuristic functions are needed.

4.2.1 Representation of Beliefs

The beliefs $b(s)$ encode the probability that state s represents the way the elements in the input are ordered. For a sorting problem of size n, the size of the state space is $n!$. For $n = 10$, this means 10^6 states. Such large state spaces introduce problems of memory and time in RTDP-BEL and other POMDP algorithms. *Memory* is a potential problem as in the worst case the size of the hash table grows with the size of the belief space which is in the order of $2^{n!}$. This problem, however, can be ameliorated by the use of good heuristic functions as discussed below.

The *time* complexity is more troublesome. The RTDP-BEL loop involves the computation of the belief states b_a and b_a^o from the original belief state b as dictated by Equations 1-3. In the worst case the time for these computations grows with $|S|^2$ and $|S||O|$ respectively. If belief states had few non-zero entries, a suitable sparse representation could be used, but this is not true in sorting where the initial belief state is uniform.

The representation that we use exploits features of the sorting problem that we expect would arise in other tasks as well.[2] First of all, since the prior is uniform and the 'sensors' (i.e., tests) are noiseless, belief states b can be represented by *sets* of states $S_b = \{s|b(s) > 0\}$. Indeed, from Bayes' rule it follows that $b(s) = 1/|S_b|$ if $s \in S_b$ and $b(s) = 0$ otherwise. Furthermore, in sorting such sets can be conveniently encoded by collection of 'links' of the form $i \rightarrow j$ for $0 \leq i, j \leq n$, where each link $i \rightarrow j$ is a constraint that excludes all states s for which $s[i] \not< s[j]$. The initial belief state b_0 is represented by an empty set of such links, while the representation of b_a^o is obtained from the representation of b_a by adding the link $i \rightarrow j$ if $o = (i < j)$, and $j \rightarrow i$ if $o = (j < i)$. The representation of b_a and b are equal for $a = cmp(i, j)$ and the first is obtained from the second by exchanging the occurrences of i and j when $a = swap(i, j)$. Our implementation extends this idea with a simple mechanism that removes redundant links after any observation (a link is redundant when it can be inferred by transitivity). The result of this representation is that we reduce the complexity of updating beliefs b into b_a^o from $|S|^2$ to $|O|$ which is significantly smaller.

4.2.2 Updating the values of belief states

The structures used to represent belief states need to be converted into numbers for computing the values

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

This expression involves a probability $b_a(o)$ that has to be obtained from the representation of b_a. One way to compute $b_a(o)$ is by computing the proportion of states s in b_a that satisfy o (s satisfies $(i < j)$ if

[2] In particular we expect similar ideas to apply to the problem of handling continuous attributes in decision tree learning, but we don't deal with such problems here.

$s[i] < s[j]$). This operation, however, is very costly as it grows linearly with $|S|$. For this reason we pursue a different approach **approximating** $b_a(o)$ for $o = (i < j)$ as:

$$b_a(o) = \begin{cases} 1 & \text{if } i \rightarrow j \text{ in } b_a \\ 0 & \text{if } j \rightarrow i \text{ in } b_a \\ 1/2 & \text{otherwise} \end{cases} \tag{4}$$

where $i \rightarrow j$ is in b_a when the link forms part of the representation of b_a or can be derived from such links by transitivity. The *approximation* here is that probabilities that are neither 0 nor 1 are mapped into 1/2. This amounts to assuming that a test $cmp(i, j)$ whose outcome is not predictable can go either way with equal probability. This assumption is not true in general but speeds up the computation and does not appear to do harm, as it is approximately correct for the tests that are optimal. We'll discuss later a similar approximation in the context of decision tree learning.

4.2.3 Heuristic Functions

The representation of beliefs reduces the complexity of updating beliefs b into b_a^o, while the approximation eliminates the cost of computing the probability $b_a(o)$. Both optimizations together speed up considerably the inner loop of the RTDP-BEL algorithm that selects and applies actions. To speed up the solution of problems we need also to consider and apply as few actions as possible. We do this by means of an heuristic function $h(b)$ that provides an estimate of the minimal expected number of actions needed to go from b to the final belief state b_F. We consider the combination of two heuristics:

1. the *longest chain heuristic* $h_l(b)$ is based on the longest sequence of links $i_1 < i_2 < i_3 < \ldots i_m$ that appear explicitly in the representation of b, with $h_l(b)$ defined as $n - m$

2. the *number of misplaced elements heuristic* $h_m(b)$ applies to *definite* belief states only; i.e., those b's such that $b(s) = 1$ for some state s. In such a case $h_m(b)$ is defined as the number of positions $i = 1, \ldots, n$, for which $s[i] \neq i$

These heuristics are not admissible in the sense that they may overestimate the minimum expected cost to the goal, and as a result may prevent the estimates $V(b)$ to approach the optimal values.[3] Yet the admissible heuristics we have tried were not as informative,

[3] See (Barto, Bradtke, & Singh 1995) for the relation between admissibility and optimality in RTDP algorithms.

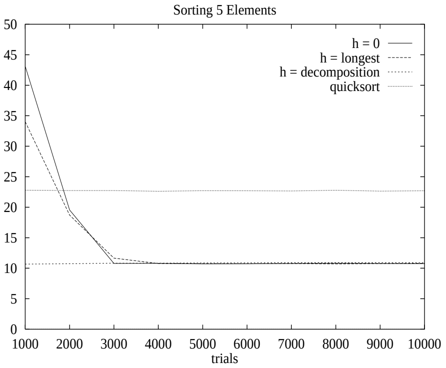

Figure 2: Average number of actions vs Number of Trials for sorting problems of sizes $n = 5$ and $n = 10$. Top line is the curve for Quicksort.

led the algorithm to visit too many belief states, and in general resulted in memory problems.

A final point about the implementation is that we impose the precondition that the ordering between the elements at positions i and j be *known* before considering a swap between them. This is done by making an action $swap(i, j)$ applicable in b only when a link $i \rightarrow j$ or $j \rightarrow i$ is in the representation of b. This condition tends to reduce the branching factor of the problem which is still large as it grows linearly with n.

4.3 EVALUATION

We tried the above implementation of the RTDP-BEL algorithm on sorting problems of two sizes. Figure 2 shows the performance of the sorting policies computed by RTDP-BEL for problems of size $n = 5$ and compares them with the ones obtained by Quicksort. The y-axis measures the average number of actions performed and the x-axis the number of trials. For $n = 5$, there are $5! = 120$ states, 20 actions, and 40 observations. The curves for RTDP-BEL correspond to the heuristic $h = 0$, $h = h_l$ and the decomposition method to be explained below. The point at trial i for $i = 1000, 2000, 3000, \ldots 10000$, indicates the average cost to reach the goal over 1000 simulations using the greedy policy determined by the estimates in the table at trial i. RTDP-BEL shows improvement with the heuristics $h = 0$ and h_l but no improvement with the decomposition method. In all cases they arrive to an expected cost that is slightly below 11 which is half the expect cost incurred by Quicksort (which is the top line in the figure). A run of 10000 trials with $h = 0$ takes in the order of 1.36 minutes and leaves 4230 entries in the hash table. The heuristic h_l and the decomposition method are slightly faster.

For larger sizes, neither of the two heuristics $h = 0$ nor $h = h_l$ scale up, and only the decomposition method works. We tried this method for $n = 10$ that generates a POMDP with several million states, 45 actions and 90 observations. The resulting curve is flat with a cost of 37. The average curve for Quicksort is also flat with an average cost of 64. The idea of the decomposition method is the following: the sorting problem is divided into two subproblems by introducing the *definite* belief states b'_F as subgoals, where the b'_F's are such that $b'_F(s) = 1$ for some s. We deal with the problem of going from b_0 to some b'_F, and from b'_F to b_F separately. That is, each subproblem has its own heuristic function and its own hash table. The second subproblem is triggered after a belief b'_F is obtained with b'_F as the initial belief state. For the first subproblem, the heuristic h_l is used, while for the second subproblem, h_m is used.

For both $n = 5$ and $n = 10$ the resulting curves for the decomposition method are practically flat. This means that the resulting algorithm starts off well but then does not improve. As mentioned above this is the result of the non-admissibility of the heuristics h_l and h_m for each of the two subproblems. We actually ran the same algorithm for both values of n eliminating the *update* step in RTDP-BEL. The resulting algorithm is a purely *greedy* algorithm and produced the same results while consuming constant memory (the table with the estimates is not needed). However even this simplification is not good for very large values of n as the branching factor (the number of actions) grows linearly with n. For such problems other optimizations are needed. An alternative that we have considered is the use of 'indexicals' to control the actions that can be considered at any given point. The indexicals in this problem can be just a pair of vector subscripts so that only comparisons and swaps of elements with those subscripts can be considered, in addition to the operation of incrementing and decrementing those indices. Schemes such as these reduce the branching factor of the problem but push the solutions deeper in search space. Whether and when such tradeoff speeds up computation remains an open question.

4.4 SUMMARY

Sorting is a challenging problem that can be effectively modeled and solved as a POMDP provided suitable heuristics, representations and decompositions are used. In this way we have solved a POMDP that involves millions of states and have obtained solutions that compare favorably with Quicksort in terms of the number of steps. The obvious weakness of the resulting sorting policy is that it applies to a particular problem size. An interesting challenge is the extraction of a concise and generalized representation of the policy that could be applied to problems of any size.

5 DECISION TREES

Decision trees are classifiers that map instances into classes by sequentially testing the value of a finite set of attributes (Mitchell 1997). The standard way to learn decision trees from data is by a top-down greedy strategy in which the attribute that is most informative for classification according to the data is used to split the data first, and for each possible outcome, the attribute that is most informative according to the remaining data is used second and so on, until either there are no more data or no more uncertainty regarding the classification (Breiman *et al.* 1984; Quinlan 1993). The generalization power of decision tree algorithms is measured by the classification error over part of the data that is left aside for testing. Decision tree learning algorithms have been applied to a number of domains (Murthy 1998) and a number of variations and extensions have been considered (Dietriech 1997).

5.1 FORMULATION

The problem of learning decision trees can be seen as a sequential decision problem that involves two types of actions: $report(i)$ by which the current instance s is classified in class c_i, and $test(j)$ by which the attribute t_j of s is observed. The goal is to have the instance s classified, and this can be achieved by any of the actions $report(i)$, $i = 1, \ldots, n$ where n is the number of classes. The expected cost associated with such actions depends on the *true* class of s. The actions $test(j)$ provide information about s. The 'classification' POMDP consists thus of:

1. **states** s that are *the instances in the training set* supplemented by a separate goal state G
2. **actions** $report(i)$ for each of the classes c_i, and $test(j)$, for each of the attributes t_j
3. **transition probabilities** $P_a(s'|s) = 1$ for $a = test(j)$ and $s' = s$, and $a = report(i)$ and $s' = G$. Otherwise $P_a(s'|s) = 0$
4. **action costs** $c(report(i), s) = C_{ij}$ for $class(s) = c_j$ and $c(test(j), s) = C_j$ for all s

5. **initial belief state** b_0 uniform over the non-goal states and zero over the goal state

6. **final belief state** b_F for which $b^F(G) = 1$

7. **observations** o after action $a = test(j)$ with probabilities $P_a(o|s) = 1$ if $o = v_j(s)$ and 0 otherwise, where $v_j(s)$ stands for the value of s over the attribute t_j

The POMDP formulation suggests generalizations of the standard decision tree learning setting such as different test and misclassification costs C_j and C_{ij}, noisy tests with $P_a(o|s) \in [0,1]$, etc. By default we assume here that the cost of tests and correct classifications is 1, while the cost C_{ij} of misclassifications for $i \neq j$, is some constant $C > 1$.

5.2 IMPLEMENTATION

We represent belief states as sets of states (training set instances), taking advantage of the the uniform prior over the instances and the noiseless 'sensors'. With this representation, the complexity of a single RTDP-BEL cycle reduces from $|S|^2$ to $|S|$. The value $b_a(o)$ for $a = test(j)$ in Equation 2 is obtained as the proportion of states s in b for which $v_j(s) = o$, a proportion that is computed as $|b_a^o|/|b|$.

We use the *non-informative* heuristic $h = 0$. Heuristics based on measures such as information gain (Quinlan 1990) could be used as well but they only make a difference in the first trials of RTDP-BEL as they are not calibrated with the expected classification costs. It may be possible to calibrate such heuristics to accelerate convergence but we don't know how to do that yet.

5.3 EVALUATION

Table 1 compares RTDP-BEL with two standard decision tree learning algorithms, ID3 and C4.5 (Quinlan 1990; 1993) over some small datasets obtained from the UCI Repository (Murphy & Aha 1998) for two different misclassification costs C.[4] For each dataset, we constructed the corresponding POMDP and ran the RTDP-BEL algorithm with the non-informative heuristics $h = 0$ for 10000 trials. The curve in Figure 3 shows the average classification accuracy as a function of the number of trials in the Monk-1 and Monk-2 datasets. A run of 10000 trials over the Monk datasets takes a few minutes on average and leaves a few thousand entries in the hash table. For the larger Votes dataset, the run takes 24 minutes on average and leaves around 16000 entries in the hash table. During testing, whenever a new belief state b_a^o was generated that was not in the hash table, b_a^o was approximated to b. This means that unexpected values in the test set are regarded as 'missing' values. This is not too different from the approach taken in decision tree learning when test instances get to a node with no compatible branches, and are classified by the distribution of instances in that node.

5.3.1 Missing Values

In the presence of missing values in the training set, the sum of the beliefs $b_a(o)$ over the *real* observations o may fail to add up to 1 due to the mass $b_a(m) \neq 0$ over the *missing* values. In such cases, the beliefs $b_a(o)$ are normalized by dividing them by the sum $\sum_i b_a(o_i)$ taken over the *real* observations o_i. This amounts to assuming that having 'observed' a missing value m is like having observed a real observation o_i with probability $b_a(o_i)$. This implies that $b_a^m = b_a$, in agreement with the interpretation of missing values as *missing observations.* The dataset $Votes$ in Table 1 has missing values.

5.3.2 Misclassification Costs and Overfitting

As expected, misclassification costs have an influence on the level of overfitting in noisy datasets. Very high misclassification costs induce the algorithm to fit the training data as much as possible, which in those cases may increment the error rate on the test set. This can be seen in the last row in Table 1, where the error rate in the Votes data set goes up by almost 10 points when the misclassification costs are increased from $C = 25$ to $C = 10000$. In general these costs do not have to be all equal and can be tuned to produce a minimal error rate by leaving aside part of the training data for that purpose. In other problems (e.g., medicine), these costs can be chosen to approximate the real misclassification costs.

[4] The figures for ID3 and C4.5 were taken from (Friedman, Kohavi, & Yun 1996). The column named 'Test' in the table indicates how the generalization performance of the algorithms was measured. The Monk-n datasets come with separate training and test data; on the other two problems the test data was generated by 5-fold cross validation: the data were partitioned into five segments, and fives runs were performed by leaving one different segment as test data. The results are the averages over these fives runs.

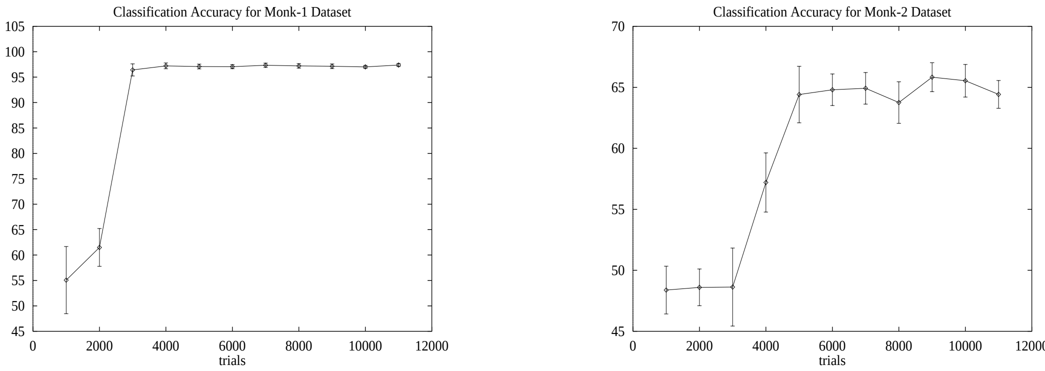

Figure 3: Classification Accuracy vs. Trials for Monk-1 and Monk-2

Table 1: Accuracy after 10000 trials compared with ID3 and C4.5

							RTDP	
Dataset	Feat.	Miss	Train	Test	ID3	C4.5	$C = 25$	$C = 10000$
monk-1	6	no	124	432	81.25 ± 1.89	75.70 ± 2.07	97.39 ± 0.29	97.39 ± 0.35
monk-2	6	no	169	432	69.91 ± 2.21	65.00 ± 2.30	64.42 ± 1.13	64.40 ± 0.81
monk-3	6	no	122	432	90.28 ± 1.43	97.20 ± 0.80	95.16 ± 0.49	94.33 ± 0.78
hayes-roth	4	no	160	CV-5	68.75 ± 8.33	74.38 ± 4.24	77.70 ± 4.65	72.04 ± 5.44
votes	16	yes	435	CV-5	93.10 ± 2.73	95.63 ± 0.43	94.42 ± 1.88	83.12 ± 6.75

5.3.3 Approximations

In another set of experiments we introduced an approximation in the evaluation of the probability $b_a(o)$, which in this case stands for the probability of observing a value v_j after testing an attribute t_j in a given context. The exact value of $b_a(o)$ is given by the number of instances in b whose attribute t_j has value v_j over the total number of instances in b. Following a similar approximation used in sorting, we approximated $b_a(o)$ uniformly as $1/n$, where n is the number of values that attribute t_j takes in the training set. As before the intuition was that the best action would be the most informative and would tend to split the data in that way. The results confirmed this intuition and matched up almost exactly the ones reported in Table 1. The CPU times were reduced three times on average. Yet even with this approximation, larger datasets could not be handled as memory tends to explode. The main problem is the lack of an informative heuristic that can guide the search, while leaving a large fraction of the (belief) state space unvisited. Heuristics such as 'information gain' (Quinlan 1990) are informative but are not calibrated with the expected costs.[5] As a result, they produce a focused search for the goal in the first few trials, but then become useless as some of the heuristic values are replaced (updated) by cost estimates. It seems that it should be possible to speed up the convergence of RTDP algorithms by the use of uncalibrated heuristics, but how to do that appears to be an open question.

5.4 SUMMARY

We have shown that decision tree induction can be modeled and solved as a POMDP problem and that solutions, while more expensive to compute, may compete in quality with the standard approaches. POMDPs may provide a fresh perspective on the problem of inferring decision trees from data as aspects such as noisy tests and data, tests and misclassification costs, and missing values, fit into the POMDP approach in a natural way. The POMDP algorithm used, however, does not scale up yet to large datasets involving many attributes, nor does it apply to datasets involving continuous attributes.

6 CONCLUSIONS

We aimed to show two things. One is that POMDPs can be used to solve complex problems of sequential decision by the use of suitable heuristics, representa-

[5] That is, information gain is not a good estimate of the expected costs.

tions, and decompositions. The second is that POMDPs provide a novel perspective on the problem of inferring decision trees from data that may be worth exploring in further depth. We have been able to solve very large POMDPs with million of states and obtain solutions that compete in quality with those produced by some of the best algorithms (Quicksort, C4.5). We expect that some of the lessons learned will be applicable to other problems such as the problem of handling continuous attributes in decision tree learning that appears to have many aspects in common with sorting. We also think that the POMDP methods used in this paper can be refined so that larger datasets could be handled. A number of interesting questions that may be relevant for the application of POMDP methods to other problems remain open; e.g., how can sorting policies be generalized to arbitrary array sizes, whether misclassification costs can be used effectively to deal with the problem of overfitting, how uncalibrated heuristics can be used to speed up converge of RTDP algorithms, etc.

Acknowledgments

This work was supported in part by a grant from Conicit, S1-96001365.

References

Aho, A.; Hopcroft, J.; and Ullman, J. 1983. *Data Structures and Algorithms.* Addison-Wesley.

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

Bellman, R. 1957. *Dynamic Programming.* Princeton University Press.

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

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

Breiman, L.; Friedman, J.; Olshen, R.; and Stone, C. 1984. *Classification and Regression Trees.* Wadsworth International Group.

Cassandra, A.; Kaebling, L.; and Kurien, J. 1996. Acting under uncertainty: Discrete bayesian model for mobile robot navigation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robot and Systems.*

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

Cassandra, A.; Kaebling, L.; and Littman, M. 1995. Learning policies for partially observable environments: Scaling up. In *Proc. of the 12th Int. Conf. on Machine Learning.*

Diettriech, T. 1997. Machine learning research. *Artificial Intelligence Magazine* 18(4):97–136.

Friedman, J.; Kohavi, R.; and Yun, Y. 1996. Lazy decision trees. In *Proceedings AAAI-96*, 717–724. MIT Press.

Geffner, H., and Bonet, B. 1998a. High-level plannig and control with incomplete information using POMDP's. Available at `http://www.ldc.usb.ve/~hector`.

Geffner, H., and Bonet, B. 1998b. Solving large POMDPs using real time dynamic programming. Available at `http://www.ldc.usb.ve/~hector`.

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

Mitchell, T. 1997. *Machine Learning.* McGraw-Hill.

Murphy, P. M., and Aha, D. W. 1998. UCI repository of machine learning databases. `http://www.ics.uci.edu/learn`.

Murthy, S. 1998. Automatic construction of decision trees from data: A multidisciplinary survey. Technical report, Siemens Corporate Research.

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

Quinlan, J. R. 1990. Induction of decision trees. *Machine Learning* 1(1).

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning.* Morgan Kauffman.

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

Sondik, E. 1971. *The Optimal Control of Partially Observable Markov Processes.* Ph.D. Dissertation, Stanford University.