Learning MDP Action Models Via Discrete Mixture Trees

Michael Wynkoop and Thomas Dietterich

Oregon State University, Corvallis, OR 97370, USA
{wynkoop,tgd}@eecs.oregonstate.edu

Abstract. This paper addresses the problem of learning dynamic Bayesian network (DBN) models to support reinforcement learning. It focuses on learning regression tree (context-specific dependence) models of the conditional probability distributions of the DBNs. Existing algorithms rely on standard regression tree learning methods (both propositional and relational). However, such methods presume that the stochasticity in the domain can be modeled as a deterministic function with additive noise. This is inappropriate for many RL domains, where the stochasticity takes the form of stochastic choice over deterministic functions. This paper introduces a regression tree algorithm in which each leaf node is modeled as a finite mixture of deterministic functions. This mixture is approximated via a greedy set cover. Experiments on three challenging RL domains show that this approach finds trees that are more accurate and that are more likely to correctly identify the conditional dependencies in the DBNs based on small samples.

Recent work in model-based reinforcement learning uses dynamic Bayesian network (DBN) models to compactly represent the transition dynamics of the actions and the structure of the reward function. DBN models require much less space than tabular models (Dean & Kanazawa, 1989), and they are able to generalize to novel parts of the state space. Additional compactness can be obtained by representing each conditional probability distribution by a regression tree (Boutilier et al., 1995), a structure we will refer to as a TDBN. Boutilier and colleagues have developed a family of approximate value iteration and policy iteration algorithms that manipulate tree-structured representations of the actions, the rewards, and the value functions (Boutilier et al., 2000).

An additional advantage of DBN representations is that they explicitly identify which state variables at time t influence the state variables at time $t + 1$. By analyzing the structure of such dependencies, it is possible to identify legal state abstractions in hierarchical reinforcement methods such as MAXQ (Dietterich, 2000). In recent work, Jonsson and Barto (2006) and Mehta, et al. (2007) have shown how to automatically discover subroutine hierarchies through structural analysis of the action and reward DBNs.

Algorithms for learning TDBNs generally employ the standard set of techniques for learning classification and regression trees (Breiman et al., 1984). Internal nodes split on one or more values of discrete variables or compare continuous values against a threshold. If the target variable is discrete, a classification tree is constructed (Quinlan, 1993), and each leaf node contains a multinomial distribution over the values of the target variable. One variation on this is to search for a decision graph (i.e., a DAG, (Chickering et al., 1997)). Search is typically top-down separate-and-conquer with some form of pruning to control overfitting, although Chickering et al. (Chickering et al., 1997) employ a more general search and control overfitting via a Bayesian scoring function. If the target variable is continuous, a regression tree is constructed. Each leaf node contains a Gaussian distribution with a mean and (implicitly) a variance (Breiman et al., 1984).

Many generalizations of the basic methods have been developed. One generalization is to allow the splits at the internal nodes of the tree to be relational (e.g., by evaluating a predicate that takes multiple variables as arguments or by evaluating a function of one or more variables and comparing it against a threshold (threshold Kramer, 1996; Blockeel, 1998)). Another is to allow the leaf nodes of regression trees to contain regression models (so-called *Model Trees*; Quinlan, 1992) or other functions (Torgo, 1997). Gama's (2004) Functional Trees combine functional splits and functional leaves. Vens et al. (2006) combine relational splits with model trees.

It is interesting to note that for discrete random variables, the multinomial distribution in each leaf represents stochasticity as random choice across a fixed set of alternatives. However, in all previous work with regression trees, each leaf represents stochasticity as Gaussian noise added to a deterministic function.

In many reinforcement learning and planning problems, this notion of stochasticity is not appropriate. Consider, for example, the $GOTO(agent, loc)$ action in the real-time strategy game Wargus (2007). If the internal navigation routine can find a path from the agent's current location to the target location loc, then the agent will move to the location. Otherwise, the agent will move to the reachable location closest to loc. If we treat the reachability condition as unobserved, then this is a stochastic choice between two deterministic outcomes, rather than a deterministic function with additive Gaussian noise. Another case that arises both in benchmark problems and in real applications is where there is some probability that when action a is executed, a different action a' is accidentally executed instead. A third, more mundane, example is the case where an action either succeeds (and has the desired effects) or fails (and has no effect).

The purpose of this paper is to present a new regression tree learning algorithm, DMT (for Discrete Mixture Trees), that is appropriate for learning TDBNs when the stochasticity is best modeled as stochastic choice among deterministic alternatives. Formally, each leaf node in the regression tree is modeled as a multinomial mixture over a finite set of alternative functions. The learning algorithm is given a (potentially large) set of candidate functions, and it must determine which functions to include in the mixture and what mixing probabilities to use. We describe an efficient algorithm for the top-down induction of such TDBNs. Rather than pursuing the standard (but expensive) EM-approach to learning finite mixture models (McLachlan & Krishnan, 1997), we instead apply the greedy set cover algorithm to choose the mixture components to cover the data points in each leaf. The splitting heuristic is a slight variation of the standard mutual information (information gain) heuristic employed in C4.5 (Quinlan, 1993).

We study three variants of DMT. The full DMT algorithm employs relational splits at the internal nodes and mixtures of deterministic functions at the leaves (DMT). DMT-S ("minus splits") is DMT but with standard propositional splits. DMT-F ("minus functions") is DMT but with constant values at the leaves. We compare these algorithms against standard regression trees (CART) and model trees (M5P). All five algorithms are evaluated in three challenging domains. In the evaluation, we compute three metrics: (a) root relative squared error (RRSE; which is most appropriate for Gaussian leaves), (b) Recall over relevant variables (the fraction of relevant variables included in the fitted model), and (c) Precision over relevant variables (the fraction of the included variables that are relevant). The results show that in two of the domains, DMT gives superior results for all three metrics. In the third domain, DMT still has better Recall but produces mixed results for RRSE and Precision.

1 Tree Representations of DBNs

Figure 1(a) shows a DBN model involving the action variable a, three state variables x_1, x_2, x_3, and the reward value r. In this model (and the models employed in this paper), there are no probabilistic dependencies within a single time step (no synchronic arcs). Consequently, each random variable at time $t+1$ is conditionally independent given the variables at time t. As always in Bayesian networks, each node x stores a representation of the conditional probability distribution $P(x|\mathbf{pa}(x))$, where $\mathbf{pa}(x)$ denotes the parents of x.

In this paper, we present a new algorithm for learning functional tree representations of these conditional probability distributions. Figure 1(b) shows an example of this representation. The internal nodes of the tree may contain relational splits (e.g., $x_2(t) < x_3(t)$) instead of simple propositional splits (e.g., $x_2(t) < 1$). The leaves of the tree may contain multinomial distributions over functions. Hence the left leaf in Figure 1(b) increments x_1 with probability 0.7 and decrements it with probablity 0.3.

There are many ways in which functional trees provide more compact representations than standard propositional regression trees (Figure 1(c)). First, relational splits are much more compact than propositional splits. To express the condition $x_2(t) < x_3(t)$, a propositional tree must check the conjunction of $x_2(t) < \theta$ and $x_3(t) \geq \theta$ for each value of θ. Second, functional leaves are more compact than constant leaves. To express the leaf condition $x_1(t+1) := x_1(t) + 1$, a standard regression tree must introduce additional splits on $x_1(t) < \theta$ for each value of θ. Finally, standard regression trees approximate the distribution of real values at a leaf by the mean. Hence, the left-most leaf of Figure 1(c) would be approximated by the constant 0.4 with a standard deviation (mean squared error) of 0.92.

This compactness should generally translate into faster learning, because in the functional trees, the data are not subdivided into many "small" leaves. However, if the learning algorithm must consider large numbers of possible splits and leaf functions, this will introduce additional variance into the learning process which could lead to overfitting and poor generalization. Hence, to obtain the

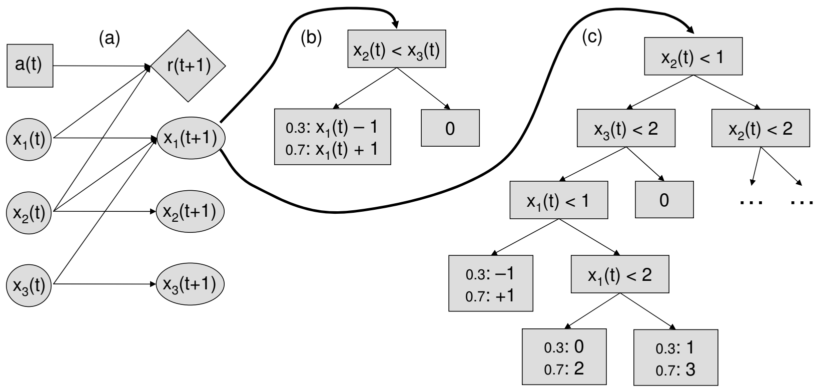

Fig. 1. (a) time slice representation of the DBN. The square action node $a(t)$ affects all nodes at time $t+1$, but for readability those arcs have been omitted. Circles represent state variables, and the diamond is the reward node. (b) a tree representation for $P(x_1(t+1)|x_1(t), x_2(t), x_3(t))$ with relational internal nodes and a probability distribution over functions ($x_1(t+1) := x_1(t)+1$ and $x_1(t+1) := x_1(t)-1$) in the left leaf. (c) a tree with propositional nodes and constant leaves must be much more copmlex to represent the same conditional probability distribution.

benefits of functional trees, the engineer must identify a constrained set of candidate relational splits and functional leaves. We adopt a proven approach from inductive logic programming (Lavrac & Dzeroski, 1994) and specify these candidate splits and functions for each domain via a context-free grammar.

Because different actions exhibit different probabilistic dependencies, all work in tree-based DBN learning—including our own—learns a separate set of regression trees for each action.

As mentioned above, other researchers have studied regression trees with relational splits and functional leaves. Our contribution is to extend these to handle multinomial mixtures of functions in the leaves.

2 Algorithm

To construct a regression tree for $x_i(t+1)$, we follow the standard recursive top-down divide-and-conquer approach using the values of $x_1(t), \ldots, x_n(t)$ as the input features and $x_i(t+1)$ as the response variable. However, we introduce two modifications. First, given a set of N values for $x_i(t+1)$ (i.e., at a leaf), we fit a mixture of functions by applying the well-known greedy set cover algorithm (Johnson, 1973). That is, we score our candidate leaf functions according to the number of training values that they fit and choose the function that fits the most points. Those points are then removed from consideration, and the process is repeated until all points are covered. The result of the set cover is a list of the form $((f_1, n_1), (f_2, n_2), \ldots, (f_k, n_k))$, where each f_j is a function and n_j is the number of data points covered by f_j that were not covered by functions $f_1, \ldots, f_{j-1}$. We then estimate the multinomial distribution as $P(f_j) = n_j/N$.

This approach introduces two approximations. First, greedy set cover is not optimal set cover (although it does give very good approximations; Slavík, 1996). Second, there may be points that are consistent with more than one of the functions $f_1, \ldots, f_k$. Strictly speaking, the probability mass for such points should be shared equally among the functions, whereas we are assigning it to the first function in the greedy set cover. In our application problems, this second case occurs very rarely and typically only affects one or two data points.

Our second modification concerns the loss function to use for scoring candidate splits. Virtually all regression tree algorithms employ the expected squared error of the children and choose the split that minimizes this squared error. This is equivalent to assuming a Gaussian likelihood function and maximizing the expected log likelihood of the training data. If we followed the same approach here, we would score the expected log likelihood of the training data using the multinomial mixture models. However, this does not work well because we assume that the mixture components (i.e., the individual functions) are themselves deterministic, so if a leaf node contains a single function with assigned probability of 1, the log likelihood of a data point is either 0 (if the function matches a data point) or $-\infty$ (if it does not). This leads to a very non-smooth function that does not work well for scoring splits. Instead, we adopt the approach that has worked well for learning classification trees (Quinlan, 1993): we score each candidate split by the expected entropy of the probability distributions in the leaves and choose the split that minimizes this expected entropy.

To prevent overfitting, we employ a form of "pre-pruning". If no test reduces the expected entropy by more than a constant ϵ, we stop splitting. In future work, we plan to replace this by a more sophisticated technique such as pessimistic pruning or MDL pruning.

Algorithm 1 shows the algorithm. It follows the standard recursive divide-and-conquer schema for top-down induction of decision trees. Ties in split selection are broken in favor of splits that introduce fewer new variables into the tree.

2.1 Efficient Splitting Function Search

Algorithm 1 requires performing two greedy set cover computations to evaluate each split. Despite the fact that greedy set cover is very efficient, this is still extremely time-consuming, especially if the set of candidate leaf functions is large. We therefore developed a method based on Uniform Cost Search (UCS) for finding the best set cover without having to evaluate all candidate leaf functions on all candidate splits.

Suppose we define a partial set cover to have the form $((f_1, n_1), (f_2, n_2), \ldots, (f_{k-1}, n_{k-1}), (\text{else}, n_k)))$.This represents the fact that there are n_k data points that have not yet been covered by any leaf function. A node in the Uniform Cost Search consists of the following information:

- the candidate splitting condition s, P_{left}, and P_{right}
- partial set covers C_{left} and C_{right} for the branches

Algorithm 1. DMT: Grow a decision tree top-down

```
1: GrowTree(examples: $E$, treenode: $T$, setcover: $C$, real: $\epsilon$)
2: $E$ is the set of training examples
3: $T$ is a tree node (initially a leaf)
4: $C$ is the set cover (with associated probability distribution) of the node
5: let $h_{root}$ := Entropy($C$)
6: Initialize variables to hold information about the best split:
7: let $h^* := h_{root}$
8: let $E^*_{left} := E^*_{right} :=$ empty set
9: let $C^*_{left} := C^*_{right} :=$ empty set cover
10: let $s^* :=$ null
11: for all candidate splits $s$ do
12:    let $E_{left} := \{e \in E | s(e)\}$ {Examples for which $s$ is true}
13:    let $E_{right} := \{e \in E | \neg s(e)\}$ {Examples for which $s$ is false}
14:    let $P_{left} := \frac{|E_{left}|}{|E|}; P_{right} := \frac{|E_{right}|}{|E|}$
15:    let $C_{left}$ := GreedySetCover($E_{left}$)
16:    let $C_{right}$ := GreedySetCover($E_{right}$)
17:    let $h_s = P_{left} \cdot$ Entropy($C_{left}$) $+ P_{right} \cdot$ Entropy($C_{right}$)
18:    if $h_s < h^*$ then
19:       let $h^* := h_s; s^* := s$
20:       $E^*_{left} := E_{left}; E^*_{right} := E_{right}; C^*_{left} := C_{left}; C^*_{right} := C_{right}$
21: if $|h_{root} - h^*| > \epsilon$ then
22:    set $T.split := s^*$
23:    let $T_{left}$ := new treenode($LEAF, C^*_{left}$)
24:    let $T_{right}$ := new treenode($LEAF, C^*_{right}$)
25:    set $T.left$ := GrowTree($E^*_{left}, T_{left}, C^*_{left}, \epsilon$)
26:    set $T.right$ := GrowTree($E^*_{right}, T_{right}, C^*_{right}, \epsilon$)
```

– the entropy of the partial set covers h_{left} and h_{right}
– the sets of uncovered response values V_{left} and V_{right}
– the current expected entropy $h = P_{left} \cdot h_{left} + P_{right} \cdot h_{right}$

The key observation is that the current expected entropy is a lower bound on the final expected entropy, because any further refinement of either of the partial set covers C_{left} or C_{right} will cause the entropy to increase.

The split selection algorithm starts by creating one UCS node for each candidate split s with empty set covers C_{left} and C_{right} and pushing them on to a priority queue (ordered to mininize h). It then considers the best greedy addition of one leaf function to C_{left} and to C_{right} to expand the two set covers, recomputes V_{left}, V_{right}, and h, and pushes this new node onto the priority queue. Note that the size of the priority queue remains fixed, because each candidate split is expanded greedily rather than in all possible ways (which would produce an optimal set cover instead of a greedy set cover).

The algorithm terminates when a node popped off the priority queue has $V_{left} = V_{right} =$ the empty set. In which case, this is the best split s^*, because it has the lowest expected entropy and all other items on the priority queue have higher entropy.

3 Experiments

To evaluate the effectiveness of our DMT algorithm, we compared it experimentally to four other algorithms: CART (Breiman et al., 1984), Model Trees (Quinlan, 1992), DMT with propositional splits and functional leaves (DMT-S, "minus splits") and DMT with relational splits but constant leaves (DMT-F, "minus functions"). In effect, CART is DMT-SF, DMT without relational splits or functional leaves.

The experiment is structured as follows. We chose three domains: (a) a version of the Traveling Purchase Problem (TPP) adapted from the ICAPS probabilistic planning competition (2006), (b) the Trucks Problem, also adapted from ICAPS, and (c) a resource gathering task that arises in the Wargus real-time strategy game (2007). In each domain, we generated 200 independent trajectories. In TPP and Trucks, each trajectory was generated by choosing at random a legal starting state and applying a uniform random policy to select actions until a goal state was reached. In Wargus, all trajectories started in the same state because they were all generated from the same map, and there is only one legal starting state per map. On average, the trajectories contained 96.9 actions in TPP (standard deviation of 56.7), 600.0 actions in Truck (s.d. of 317.8), and 2065 actions in Wargus (s.d. of 1822). The 200 trajectories were randomly partitioned into a training set of 128 and a test set of 72 trajectories. To generate learning curves and to obtain independent training trials, the 128 training trajectories were further divided into 2 subsets of 64, 4 subsets of 32, 8 subsets of 16, 16 subsets of 8, 32 subsets of 4, 64 subsets of 2, and 128 subsets each containing only one trajectory. For each of these training sets, each of the five algorithms was run. The resulting DBN models were then evaluated according to three criteria:

– Root Relative Squared Error (RRSE). This is the root mean squared error in the predicted value of each state variable divided by the RMS error of simply predicting the mean. Some variables are actually 0-1 variables, in which case the squared error is the 0/1 loss and the RRSE is proportional to the square root of the total 0/1 loss.
– State variable Recall. Because we wish to use the learned trees to guide subroutine discovery algorithms (2006; 2007), we want algorithms that can correctly identify the set of parents $\mathbf{pa}(x)$ of each variable x. The recall is the fraction of the true parents $\mathbf{pa}(x)$ that are correctly identified in the DBN model.
– State variable Precision. We also measure precision, which is the fraction of parents in the learned DBN that are parents in the true DBN.

3.1 Domains

Here are the detailed specifications of the three domains.

Traveling Purchase Problem. (TPP) is a logistics domain where an agent controls a truck that must purchase a number of goods from different markets and then return them to a central depot. Each market has a supply of goods, as well as its own price, both of which are random for each problem instance and are provided as state variables. The state also contains variables that represent the remaining demand. These variables are initialized with the total demand for that product, and they are decremented as the agent buys goods from the markets. Actions in this domain consist of goto actions, actions that buy units of products from markets, and an action to deliver all purchased products to the central depot.

For these experiments, the domain was restricted to two markets, one central depot, and three products. This results in an MDP with 15 state variables and 10 actions. Initial values for product supply and demand can range from zero to 20, which produces an MDP with over 10^{12} states. The price variables do not count toward the size of the state space, because their values are constant throughout an instance of the problem.

The Truck Problem. is another logistics problem. However in this domain, the focus is on the logistics of picking up packages, placing them in the right order on the truck, dropping them off, and delivering them to their proper destination. Here the agent is in control of two trucks; each truck has two areas (front and rear) in which it can hold packages. As in a real delivery truck, these areas must be loaded and unloaded in the proper order. For example, if there is a package in the front area, an action that attempts to remove the package from the rear of the truck fails.

For our experiments, the two trucks are asked to deliver three packages from one of five locations to another of the five locations. Once a package is dropped off at the correct location, an action must be taken by the agent to deliver the package to the customer. Actions in this domain include loading and unloading a package on a truck, driving a truck to a location, and delivering a package to the customer once it is at its goal location. The domain has 25 actions and 12 state variables, with $4.3 \cdot 10^8$ states.

Wargus. is a resource gathering domain where an agent controls one or more peasants and directs them in a grid world that contains gold mines, stands of trees, and town halls. The agent can navigate to anywhere on the map with a set of goto actions. These are temporally extended actions that bring a peasant to specified region of the map. The regions are defined by the "sight radius" of the peasants. Within this radius, they can execute other actions such as mining gold, chopping wood, and depositing their payload in the town hall. Once the peasant has deposited one set of gold and one set of wood to the town hall, the episode ends and reward is received.

For this set of experiments, we use a single peasant on a map with a single gold mine and a single town hall. Trees are distributed randomly around the map. The state contains variables that list the position of the peasant on the map, what the peasant is holding, what objects of interest are within sight radius of the peasant (wood, gold, town hall), and the status of the gold and wood quotas. The domain has 19 actions and $9.8 \cdot 10^4$ states.

This domain differs from the other two domains because it is not fully observable. The map that the peasant is navigating is a hidden variable that determines not only the navigation dynamics but the presence of trees, gold mines, and town halls within the peasant's sight radius.

3.2 Results

For each combination of a domain, training set, output state variable (or reward), and action, we measured the three metrics. We performed an analysis of variance on each metric, holding the domain, action, variable and training set size constant and treating the multiple training sets as replications. We treated DMT as the baseline configuration and tested the hypothesis that the metric obtained by each of the other algorithms was significantly worse (a "win"), better (a "loss"), or indistinguishable (a "tie") at the $p < 0.05$ level of significance. The test is a paired-differences t test. Note that the power of this test decreases as the number of training trajectories increases. When the training set contains only 1 trajectory, there are 128 replications, but when the training set contains 64 trajectories, there are only 2 replications. Hence, we generally expect to see the percentage of ties increase with the size of the training set.

Table 1 aggregates the results of these statistical tests over all state variables and all actions in each domain. The large number of ties in each cell is largely an artifact of the loss of power for large training set sizes. Let us first compare DMT with DMT-F (constant leaves). In TTP and Truck, DMT performs much better than DMT-F. In Wargus, the situation is less clear. On Recall, DMT is always at least as good as DMT-F, but on RRSE the algorithms each have a large number of wins, and on precision, DMT-F tends to be better. Next, consider DMT and DMT-S (propositional splits). In this case, DMT is dominant except for Wargus precision, where DMT sometimes includes unnecessary variables that DMT-S avoids. Next, compare DMT with CART (i.e., DMT-SF). Here, DMT is almost always superior on all metrics. Note in particular that for RRSE, it is superior in 1057 out of 1120 cases (94.4%) in TTP, 2085 out of 2100 cases (99.3%) in Truck, and 519 out of 662 cases (78.4%) in Wargus. So the number of ties is actually quite small, despite the low number of replicates at large sample sizes. The only case of less than overwhelming superiority is again Precision in Wargus. Finally, compare DMT with M5P, which is the Weka implementation of model trees. DMT is again dominant in the TPP and Truck domains. In Wargus, DMT is always at least as good as M5P for Recall, but on Precision DMT is more often inferior to M5P than the reverse, and on RRSE, DMT has 247 wins whereas M5P has 207, so there is no overall winner.

Table 1 hides the effect of increasing sample size. To visualize this, Figure 2 shows the win/loss/tie percentages as a function of training set size for Recall comparing DMT versus CART. Each vertical bar is divided into three parts indicating wins, losses, and ties (reading from bottom to top). In virtually all cases, there are no losses, which means that DMT's Recall is almost always better than or equal to CART's Recall. Note also that as the size of the training set gets large (and hence, the number of training sets gets small), we observe

Table 1. Statistical wins, losses and ties for DMT all other tested algorithms on each domain. These results are over all non-reward variable models. A win (or loss) is a statistically significant difference between DMT and the indicated algorithm ($p < 0.05$; paired t test).

	TTP											
	DMT-F			DMT-S			CART			M5P		
	Win	Loss	Tie	Win	Loss	Tie	Win	Loss	Tie	Win	Loss	Tie
Precision	671	3	446	7	3	1110	787	9	324	358	4	758
Recall	404	0	656	20	0	1040	411	0	649	390	0	670
RRSE	716	40	364	44	21	1055	1057	38	25	475	37	608
	Truck											
	DMT-F			DMT-S			CART			M5P		
	Win	Loss	Tie	Win	Loss	Tie	Win	Loss	Tie	Win	Loss	Tie
Precision	594	0	1504	55	0	2043	1240	7	851	679	1	1418
Recall	356	22	1645	117	1	1905	566	14	1443	467	16	1540
RRSE	1046	83	971	211	56	1833	2085	5	10	838	9	1253
	Wargus											
	DMT-F			DMT-S			CART			M5P		
	Win	Loss	Tie	Win	Loss	Tie	Win	Loss	Tie	Win	Loss	Tie
Precision	14	87	930	15	21	995	291	135	605	120	206	705
Recall	118	0	946	15	1	1048	212	0	852	176	0	888
RRSE	172	182	308	55	16	591	519	87	56	247	207	208

more ties. This is a consequence of the loss of statistical power of the t test. Space limits prevent us from showing these curves for the other metrics or for the reward TDBNs.

Figure 3 presents learning curves for RRSE. We do not have space to show the learning curve for every combination of action and variable, so we chose one variable-action pair from each domain. For the TPP Supply variable (Purchase action), we see that for training sets of size 8 and above, DMT has the lowest RRSE, DMT-S and M5P come next, the DMT-F and CART are the worst. For the Truck variable Truck Area (Load action), DMT always has the lowest RRSE. At 8 trajectories, it is joined by DMT-F, while the other three algorithms have much higher error levels. This suggests that using relational splits is critical in this domain, and we observed this for several other variable-action-domain combinations. Finally, for the Wargus Reward variable (Navigate action), the three DMT variants have the lowest RRSE (and are indistinguishable). M5P comes next, and CART gives the worst performance. The explanation for this is less clear. Evidentally, good performance requires either relational splits or functional leaves but not both!

Also shown in Figure 3 is the model sizes for the variable-action pairs depicted in the corresponding RRSE plots. For the Supply variable (Purchase action) in the TPP domain, both DMT and DMT-S perform the best, followed by M5P, CART and DMT-F. DMT-S returns a single mixture of functions in this case because it does not have access to the more complex splits of full DMT. In the Truck domain's Load action (Truck Area variable), DMT always produces

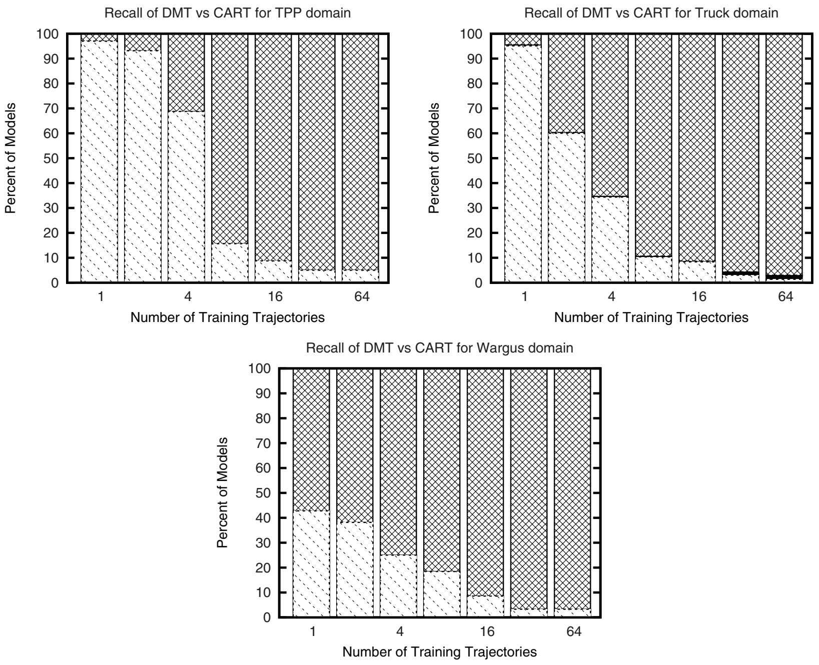

Fig. 2. Recall of DMT vs. CART for state variables for TPP, Truck, and Wargus. Each bar is divided into three sections (wins, losses, and ties). The losses are too infrequent to be visible in this plot.

the smallest models, followed by M5P, DMT-F, DMT-S and CART. Finally, in the reward node for a GOTO action in Wargus, we see that DMT and its variants produce the largest models, with the smallest models being produced by M5P and CART. This is consistent with our original hypothesis that the DMT algorithm performs best when the stochasticity is best represented as a mixture over discrete functions. GOTO is a temporally extended action that follows a navigation policy set by Wargus itself, its reward function is represents the distance between the current point and the destination point with noise added in from detours caused by obstacles in the agent's path.

To understand the Precision and Recall behavior of the algorithms, it is not sufficient to plot learning curves of the average Precision and Recall. This is because the distribution of measured Precision and Recall scores is highly skewed, with many perfect scores. Instead, we developed the Precision and Recall profiles shown in Figure 4 as a way of visualizing the distribution of Precision and Recall scores. For each domain, action, result variable, and training set, we computed the Precision and Recall of the fitted TDBN with respect to the set of variables included in the model compared to the variables included in the true DBN. For each domain, we sorted all of the observed scores (either Precision or Recall, depending on the graph) into ascending order and then for each value θ of the

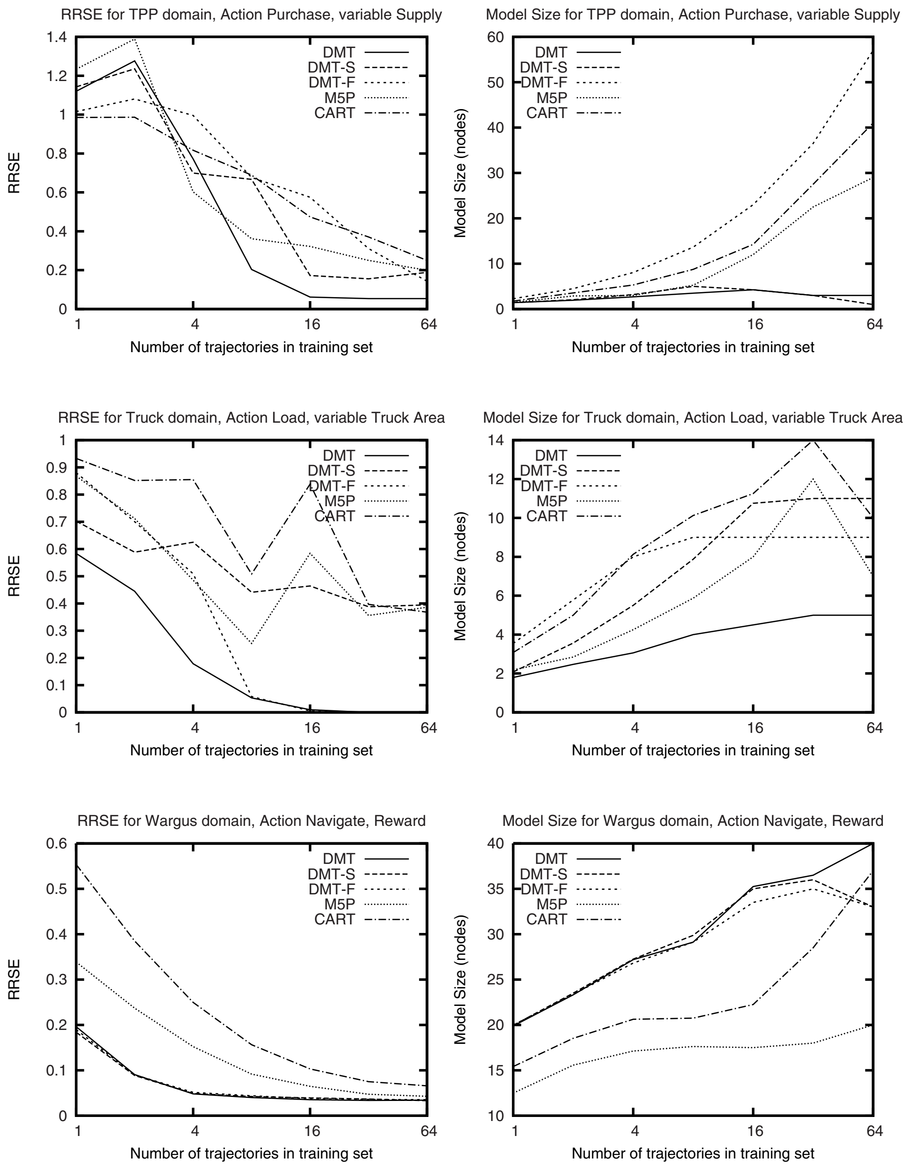

Fig. 3. RRSE and Model Size in nodes as a function of the number of trajectories in the training set for one chosen action and variable in each domain. Top: RRSE and Model Size for Market Supply for the Purchase action in TTP; Middle: RRSE and Model Size for Truck Area for the Load action in Truck; Bottom: RRSE and Model Size of the Reward node for a Goto action in Wargus.

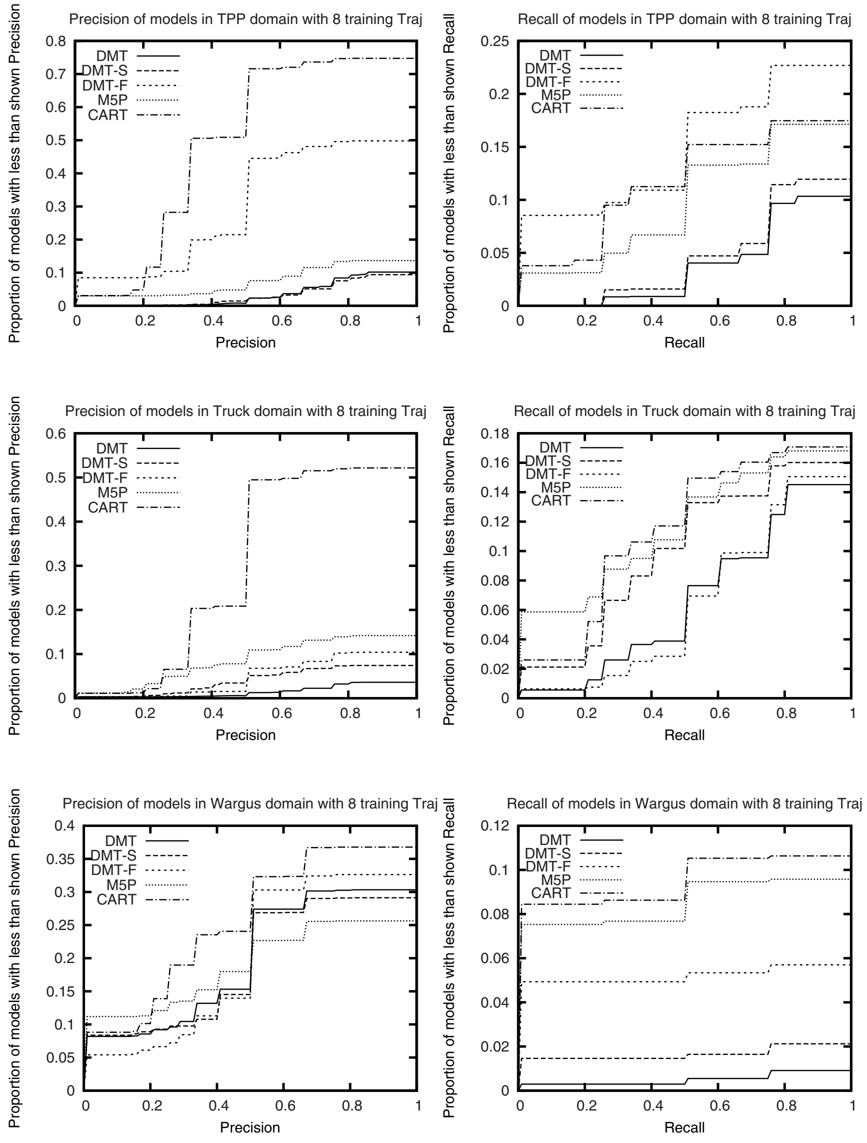

Fig. 4. Precision and Recall profiles for each domain when trained on 8 trajectories and compared to the true DBN models. These curves aggregate over all variables, actions, and training sets in each domain. Each plotted point specifies the fraction of learned models with Precision (or Recall, respectively) less than the value specified on the horizontal axis. Hence, the ideal curve would be a flat line at 0, corresponding to the case where all learned models had Precision (or Recall) of 1.0.

score we plotted the fraction of TDBNs where the score was $< \theta$. The ideal profile would be a flat line corresponding to the case where *all* learned TDBNs had a perfect score of $\theta = 1.0$, so none of them were less than θ. The higher and more rapidly the profile rises, the worse the performance. In short, these are cumulative distribution functions for Precision and Recall. Figure 4 shows these profiles for cases where the training set contains 8 trajectories. We chose this as the middle point on the learning curves (with respect to log sample size). The TPP profile is based on a total of 2560 models, the Truck profile on 4800 models, and the Wargus profile on 2736 models.

For the TPP domain, DMT and DMT-S track each other very closely and are consistently superior to all of the other algorithms. Only 10% of the trials had Precision or Recall less than 1.0. M5P comes next with excellent Precision. CART had the worst Precision and DMT-F was also quite bad. For Recall, DMT-F is the worst, while CART matches M5P's poor performance.

For the Truck domain, CART gives extremely bad Precision—more than half of the runs had Precision of around 0.5 or less. All of the other methods do much better with DMT being best with more than 95% of the runs achieving Precision of 1.0. On Recall, all of the algorithms do fairly well with DMT and DMT-F doing the best and the others somewhat worse.

Finally, for Wargus DMT-F has the best Precision for low values but the second-worst Precision at high values. M5P is best at the high end. DMT and DMT-S are in the middle of the pack, and CART is the worst. For Recall, DMT is excellent with DMT-S very good and DMT-F respectable. M5P and CART are quite a bit worse. The very high Recall and poor Precision of DMT suggests that it is overfitting and creating large models that contain extra variables. This suggests that there is room for improvement in the overfitting avoidance mechanisms of DMT.

4 Concluding Remarks

This paper has presented a new algorithm, DMT, for learning regression tree models of conditional probability distributions for DBNs. The algorithm is designed to handle domains in which stochasticity is best modeled as stochastic choice over a small number of deterministic functions. This stochasticity is represented as a finite mixture model over deterministic functions in each leaf of the regression tree. These mixture models are learned via greedy set cover. Experiments on three challenging domains provide evidence that this approach gives excellent performance, both in terms of prediction accuracy but also, perhaps more importantly, in terms of the ability to correctly identify the relevant parents of each random variable. In two of the domains, DMT is clearly superior to CART and M5P. In the third domain (Wargus), there are many cases where DMT performs well, but there are also many cases where it gives worse prediction accuracy and precision than CART or M5P. This suggests that DMT may be overfitting in this domain.

It is interesting to note that this problem could also be viewed as a multi-label classification problem with overlapping classes. If each training point were labeled with the set of functions that apply to it, then the optimal mixture of classes for a set of examples represents the optimal discrete mixture of functions for those examples. This leads to some complications when classes partially or fully overlap, such as assigning probability mass of an example that is a member of several classes. These issues can be addressed by the same approximations used to increase efficiency that are described in this paper.

In future work, we plan to incorporate stronger methods for regularizing DMT by controlling both the tree size and the size of the set covers in each leaf. We would also like to extend this approach to allow stochasticity both in the mixture of functions and in the individual functions themselves. In addition, we plan to use the TDBNs learned by DMT as input to the MAXQ discovery algorithm that we have developed (2007).

Acknowledgement

Thank you Souyma Ray, Prasad Tadepalli and Neville Mehta for all your support. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-05-2-0249.

References

5th International Planning Competition, International conference on automated planning and scheduling (2006)

Blockeel, H.: Top-down induction of first order logical decision trees. Doctoral dissertation, Katholieke Universiteit Leuven (1998)

Boutilier, C., Dearden, R.: Exploiting structure in policy construction. In: IJCAI 1995, pp. 1104–1111 (1995)

Boutilier, C., Dearden, R., Goldszmidt, M.: Stochastic dynamic programming with factored representations. Artificial Intelligence 121, 49–107 (2000)

Breiman, L., Friedman, J., Olshen, R., Stone, C.: Classification and regression trees. Wadsworth Inc. (1984)

Chickering, D.M., Heckerman, D., Meek, C.: A Bayesian approach to learning Bayesian networks with local structure. In: UAI 1997, pp. 80–89 (1997)

Dean, T., Kanazawa, K.: A model for reasoning about persistence and causation. Computational Intelligence 5, 33–58 (1989)

Dietterich, T.G.: Hierarchical reinforcement learning with the MAXQ value function decomposition. JAIR 13, 227–303 (2000)

Gama, J.: Functional trees. Machine Learning 55, 219–250 (2004)

Johnson, D.S.: Approximation algorithms for combinatorial problems. In: STOC 1973, pp. 38–49. ACM, New York (1973)

Jonsson, A., Barto, A.G.: Causal graph based decomposition of factored MDPs. Journal of Machine Learning Research 7, 2259–2301 (2006)

Kramer, S.: Structural regression trees. In: AAAI 1996, pp. 812–819. AAAI Press, Menlo Park (1996)

Lavrac, N., Dzeroski, S.: Inductive logic programming, techniques and applications. Ellis Horwood (1994)

McLachlan, G., Krishnan, T.: The EM algorithm and extensions. Wiley, New York (1997)

Mehta, N., Wynkoop, M., Ray, S., Tadepalli, P., Dietterich, T.: Automatic induction of maxq hierarchies. In: NIPS Workshop: Hierarchical Organization of Behavior (2007)

Quinlan, J.R.: Learning with Continuous Classes. In: 5th Australian Joint Conference on Artificial Intelligence, pp. 343–348 (1992)

Quinlan, J.R.: C4.5: Programs for machine learning. Morgan Kaufmann, San Francisco (1993)

Slavík, P.: A tight analysis of the greedy algorithm for set cover. In: STOC 1996, pp. 435–441. ACM, New York (1996)

The Wargus Team, Wargus sourceforge project (Technical Report) (2007), `wargus.sourceforge.org`

Torgo, L.: Functional models for regression tree leaves. In: Proc. 14th International Conference on Machine Learning, pp. 385–393. Morgan Kaufman, San Francisco (1997)

Vens, C., Ramon, J., Blockeel, H.: Remauve: A relational model tree learner. In: ILP 2006, pp. 424–438 (2006)