AN ABSTRACT OF THE THESIS OF

Michael S. Wynkoop for the degree of Masters of Science in Computer Science presented on June 9, 2008.

Title: Learning MDP Action Models via Discrete Mixture Trees

Abstract approved: _____

Thomas G. Dietterich

This thesis addresses the problem of learning dynamic Bayesian network (DBN) models to support reinforcement learning. It focuses on learning regression tree models of the conditional probability distributions of the DBNs. Existing algorithms 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 a random 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. To combat overfitting, pruning techniques incorporating log likelihood and KL-Divergence are employed. Experiments on three challenging RL domains, two with stochastic variants, 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.

Learning MDP Action Models via Discrete Mixture Trees

by

Michael S. Wynkoop

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Masters of Science

Presented June 9, 2008
Commencement June 2009

Masters of Science thesis of Michael S. Wynkoop presented on June 9, 2008.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electric Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Michael S. Wynkoop, Author

ACKNOWLEDGEMENTS

This work would not have been possible without the contributions and counsel from my major professor, Dr. Tom Dietterich. I would also like to thank Neville Mehta, Dr. Souyma Ray and Dr. Prasad Tadepalli. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-05-2-0249.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 – Introduction

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 Tree-structured Dynamic Bayesian Network (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 state abstractions in hierarchical reinforcement methods such as MAXQ (Dietterich, 2000). In recent work, Jonsson and Barto (2006) and Mehta, et al. (2007) (2008) 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. (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 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 thesis 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 a finite number of 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) (Friedman et al., 1998), 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).

Like most other tree learning procedures, there exists the possibility of overfitting. To combat this, we implemented two forms of pruning, the first a simple pre-pruning technique based on the expected change in entropy from inserting a split. The second is a post-pruning technique that employs two methods to reduce overfitting, one for pruning the mixtures in the leaves based on the log likelihood over a holdout set and then calculating the log likelihood of a sub-tree to determine if it should be pruned away or not. In order to discourage splits resulting in very similar distributions to the parent, a penalty term based on the KL-Divergence (Kullback & Leibler, 1951) is added to the log likelihood of a potential split.

We study three variants of DMT along side our post-pruning algorithm. 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). Unless indicated, all trials use the simple pre-pruning technique; however we also test the full version of our algorithm using the aforementioned post-pruning method (DMT-p).

All six algorithms are evaluated in three challenging domains, with two stochastic variations. 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 all 5 domains, DMT gives superior results in all metrics the exception of DMT-S, which has a slight advantage in two domains. They also show that the post-pruning algorithm, DMT-p, performs comparably with DMT in most cases, and improves its performance in several areas.

Chapter 2 – Tree Representations of DBNs

Figure 2.1(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 thesis), there are no probabilistic dependencies within a single time step (no synchronic arcs). Consequently, each random variable at time $t+1$ is conditionally independent of other variables at time $t+1$ 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 2.1(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 2.1(b) increments x_1 with probability 0.7 and decrements it with probability 0.3.

There are many ways in which functional trees provide more compact representations than standard propositional regression trees (Figure 2.1(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

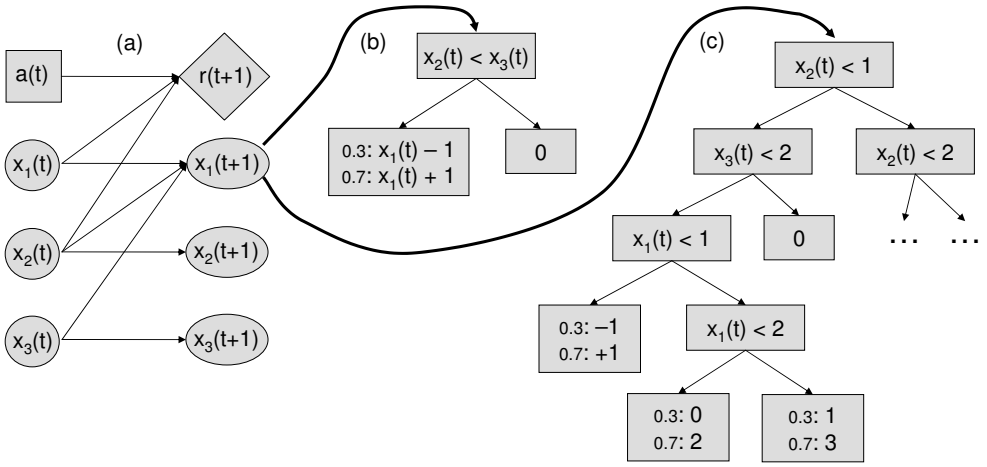

Figure 2.1: (a) time slice representation of the DBN. The square action node $a(t)$ affects all nodes at time $t+1$, but for readability those arcs have been omitted. Circles represent state variables, and the diamond is the reward node. (b) a tree representation for $P(x_1(t{+}1)|x_1(t), x_2(t), x_3(t))$ 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 complex to represent the same conditional probability distribution.

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 2.1(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 benefits of functional trees, the engineer must identify a constrained set of candidate relational splits and functional leaves. We adopt a method of defining these functions that can easily be generated with a context-free grammar, an approach from inductive logic programming (Lavrac & Dzeroski, 1994).

Because different actions exhibit different probabilistic dependencies, our work 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.

Chapter 3 – 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 two forms of pruning. One is a pre-pruning method based on the expected entropy of a potential split. The other is a post-pruning method using the log likelihood of the model over a holdout set. The details of both methods will be discussed later.

Algorithm 1 shows the DMT 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.

Algorithm 1 DMT: Grow a decision tree top-down

GrowTree(examples: E, treenode: T, setcover: C, real: ϵ)
E is the set of training examples
T is a tree node (initially a leaf)
C is the set cover (with associated probability distribution) of the node
let $h_{root} :=$ Entropy(C)
Initialize variables to hold information about the best split:
let $h^* := h_{root}$
let $E^*_{left} := E^*_{right} :=$ empty set
let $C^*_{left} := C^*_{right} :=$ empty set cover
let $s^* :=$ null
for all candidate splits s **do**
  let $E_{left} := \{e \in E | s(e)\}$ {Examples for which s is true}
  let $E_{right} := \{e \in E | \neg s(e)\}$ {Examples for which s is false}
  let $P_{left} := \frac{|E_{left}|}{|E|}; P_{right} := \frac{|E_{right}|}{|E|}$
  let $C_{left} :=$ GreedySetCover(E_{left})
  let $C_{right} :=$ GreedySetCover(E_{right})
  let $h_s = P_{left} \cdot$ Entropy(C_{left}) $+ P_{right} \cdot$ Entropy(C_{right})
  if $h_s < h^*$ **then**
    let $h^* := h_s; s^* := s$
    $E^*_{left} := E_{left}; E^*_{right} := E_{right}; C^*_{left} := C_{left}; C^*_{right} := C_{right}$
if $|h_{root} - h^*| > \epsilon$ **then**
  set $T.split := s^*$
  let $T_{left} :=$ new treenode($LEAF, C^*_{left}$)
  let $T_{right} :=$ new treenode($LEAF, C^*_{right}$)
  set $T.left :=$ GrowTree($E^*_{left}, T_{left}, C^*_{left}, \epsilon$)
  set $T.right :=$ GrowTree($E^*_{right}, T_{right}, C^*_{right}, \epsilon$)

3.1 Efficient Splitting Function Search

Algorithm 1 requires performing two greedy set cover computations to evaluate each potential 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. This method is shown in Algorithm 2.

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 UCS 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
- 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 these nodes on to

Algorithm 2 Efficent Split Selection Algorithm for DMT

BestSplitSearch(examples: E, treenode: T, real: ϵ)
E is the set of training examples
T is a tree node (initially a leaf)
Initialize variables to hold information about the best split search:
let $h_{root} :=$ Entropy(C)
let Q_{ent} be a priority queue of search nodes sorted on the partial entropy
for all candidate splits s **do**
    let *node* be a new search node
    $node.s := s$
    $node.V_{left} := \{e \in E | s(e)\}$ {Examples for which s is true}
    $node.V_{right} := \{e \in E | \neg s(e)\}$ {Examples for which s is false}
    let $node.C_{left} := node.C_{right} :=$ the empty set
    let $node.h_{left} := node.h_{right} := 0$
    let $node.P_{left} := node.P_{right} := 0$
    Push($node, Q_{ent}$)
while Peek(Q_{ent}).V_{left}! = Peek(Q_{ent}).V_{right}! = the empty set **do**
    let $n :=$ Pop(Q_{ent})
    let f_{left} be the function that covers the most examples in $n.V_{left}$
    let f_{right} be the function that covers the most examples in $n.V_{right}$
    let p_{left} be the proportion of examples in left branch covered by f_{left}
    let p_{right} be the proportion of examples in left branch covered by f_{right}
    Remove covered examples from $n.V_{left}$ and $n.V_{right}$
    Update $n.C_{left}$ and $n.C_{right}$
    $n.h := n.h + n.P_{left} \cdot p_{left} \log p_{left} + n.P_{right} \cdot p_{right} \log p_{right}$
    Push(n, Q_{ent})
let $n :=$ Pop(Q_{ent})
if $|h_{root} - n.h| > \epsilon$ **then**
    Return split $n.s$
else
    Create leaf from examples

a priority queue (ordered to minimize h). It then repeatedly pops the node with smallest h, computes the best greedy addition of one leaf function to C_{left} and one 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.

3.2 Pruning Methods

Overfitting can be a problem with any tree growth algorithm. There are two common methods of pruning, early stopping of the tree growth upon some criterion is known as pre-pruning and growing a full tree and then removing sub-trees is known as post-pruning. Pre-pruning has the advantage of reducing tree growth time. However because it does not look ahead beyond the current node and its children, it can be caught in local minima. In contrast, with post-pruning, the tree is grown to full size before pruning. This allows useful sub-trees to be discovered that would not have been found with pre-pruning.

We employ two methods of pre-pruning in our tree growth algorithm. First,

no split can be made from a node that contains too few examples. Second, if the difference between the entropy of the best split and the entropy of the parent node is less than a threshold ϵ, then the node is not expanded.

Algorithm 3 Discrete mixture pruning via log likelihood

```
1: PruneLeafLL(examples: E, holdout: H, treenode: T) returns real: log likelihood
2: let LL* := −∞ be the Log Likelihood of the best mixture
3: let M* be the best mixture
4: for all candidate mixtures, M, of discrete mixture in T do
5:    let LL := 0
6:    let μ be the average value of examples in E not covered by M
7:    let p_μ be the probability that an example is not covered by M
8:    for all examples (X, y) ∈ H do
9:       LL := LL + p_μ ψ(μ, σ, y)
10:      for all function, probability pairs (f, p_f) ∈ M do
11:         if f(X) == y then LL := LL + p_f
12:   if LL > LL* then
13:      LL* := LL
14:      M* := M ∪ {(μ, p_μ)}
15: T.M := M*
16: return LL*
```

Our post-pruning algorithm focuses on first pruning the mixtures at the leaves and then replacing internal nodes with leaf nodes (see Algorithm 3). Here functions are removed from the mixture, and the examples that were covered by them are averaged into a pickup term. This term is not dependent on any variable and is simply the average of the uncovered examples. The addition of this term to the mixture allows the DMT algorithm to model a function where the stocasticity is in the form of Gaussian noise. To find the best mixture for the leaf, we remove functions starting with the least likely in the mixture and up from there. This is an approximation to testing an exponential number of possible mixtures and proves

Algorithm 4 DMT structure pruning

PRUNELL(examples: E, holdout: H, treenode: T) returns real: log likelihood
if T is a leaf **then**
  $LL = \text{PRUNELEAFLL}(E, H, T)$
  return LL
let $T.s$ be the split function in T
let $E_{left} := \{e \in E | T.s(e)\}$
let $E_{right} := \{e \in E | \neg T.s(e)\}$
let $H_{left} := \{h \in H | T.s(h)\}$
let $H_{right} := \{h \in H | \neg T.s(h)\}$
let $KL_{left} = \text{KLD}(T.M, T.left.M)$
let $KL_{right} = \text{KLD}(T.M, T.right.M)$
let $LL_{leaf} := \text{PRUNELEAFLL}(E, H, T)$
let $LL_{subtree} := \text{PRUNELL}(E_{left}, H_{left}, T.left) + \text{PRUNELL}(E_{right}, H_{right}, T.right) + \frac{C}{KL_{left}+KL_{right}}$
if $LL_{leaf} > LL_{subtree}$ **then**
  Delete children of T
  return LL_{leaf}
return $LL_{subtree}$

to be very accurate because functions that explain large numbers of examples tend to have very low error in the test set, while those that cover fewer examples tend to result from noise. Each possible mixture is scored via log likelihood, we model the pickup term as a Gaussian with mean as its value and a variance of σ^2 and the terms in the mixture as a discrete distribution. See equation 3.1, where $\psi(\mu, \sigma, y) = \frac{1}{\sigma\sqrt{2\pi}}\text{EXP}\left(\frac{-(\mu-y)^2}{2\sigma^2}\right)$.

$$LL(M, (\mu, p_\mu)) = \sum_{(X,y)\in H} \log\left[\sum_{(f,p_f)\in M} p_f \text{I}(f(X) == y) + p_\mu \psi(\mu, \sigma, y)\right] \tag{3.1}$$

In order to prune the tree, a post-order tree traversal tests the log likelihood of

the pruned discrete mixture versus the log likelihood of the sub-tree rooted at that node, Algorithm 4. Because this measure does not explicitly encourage smaller trees, we also add a penalty term to this score to punish a split for having children whose distributions are similar to the parent node's distribution. To measure the similarity between the parent and child distributions, we calculate the Kullback-Liebler divergence (3.2) between their mixtures over the unique (S, A, R, S) tuples in the holdout set.

$$\mathrm{KLD}(M_{parent}, M_{child}) = \sum_{(X,y)\in H} Pr(f(X) = y|M_{parent}) \log \frac{Pr(f(X) = y|M_{parent})}{Pr(f(X) = y|M_{child})} \tag{3.2}$$

Chapter 4 – 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. We also compared the standard DMT algorithm that uses pre-pruning with the same tree growth algorithm, but using post-pruning instead of pre-pruning (DMT-p).

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 (5th International 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 (The Wargus Team, 2007). In each domain, we generated 2048 independent trajectories. In TPP, TPP-R, Trucks and Trucks-R 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 101.4 actions in TPP (standard devi-

ation of 50.5), 108.9 actions in TPP-R (s.d. of 56.1), 591.1 actions in Truck (s.d. of 295.8), 643 actions in Truck-R (s.d. of 335.2) and 1907 actions in Wargus (s.d. of 1808).The 2048 trajectories for each domain were partitioned into 32 groups of 64 trajectories each. To create training sets of increasing size, disjoint sets of sizes 1, 2, 4, 8, 16, 32 and 64 are selected from each group. Each training set contains all trajectories used in the smaller sets. For each of these training sets, each of the six 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 (Jonsson & Barto, 2006; Mehta et al., 2007; Mehta et al., 2008), 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.

4.1 Domains

Here are the detailed specifications of the three domains.

4.1.1 Traveling Purchase Problem

The 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.

In a variation of this domain, TPP-R, stochasticity is added to the domain by introducing a 10% failure probability to each action. Upon failure all pick up actions receive a random amount of product less than the intended amount, the movement actions put the truck at a random location, and the drop off action fails to drop off the product.

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.

4.1.2 The Truck 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.

In a variation of this domain, TRUCK-R, a failure rate of 10% was added to every action. Failure of movement actions result in the truck to be moved ending up at a random location, and when all other actions fail the state remains unchanged.

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.

4.1.3 Wargus

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.

4.2 Results

For each combination of a domain, training set, output state variable (or reward), and action, we measured the three metrics. Overall, we see that the DMT algorithm as good as or better than the baselines in the majority of trials. When post pruning is applied, we see an improvement in both error and model size, and comparable scores in both precision and recall. Overall, the methods introduced by the DMT algorithm prove to be an improvement over existing methods of modeling trees in the tested domains.

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.

Table 4.1 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 algorithms tending to converge to the correct model given large training set sizes. Let us first compare DMT with DMT-F (constant leaves). In TTP, TPP-R, Truck and Truck-R, DMT performs much better than DMT-F. In Wargus, DMT out-performs DMT-F consistently in RRSE and precision; however in recall, the number of wins is overshadowed by the number of ties, so the improvement over DMT-F is marginal at best. Next, consider DMT and DMT-S (propositional

splits). In this case, DMT is dominant except for Wargus precision and recall, Truck and Truck-R precision. Numbers of wins and losses in TPP precision and recall, TPP-R precision and recall, Truck precision, and Wargus precision and recall are overshadowed by the number of ties, therefore the wins and losses in those categories are not significant.

Next, compare DMT with CART (i.e., DMT-SF). Here, DMT is superior on all metrics. Note in particular that for RRSE, it is superior in 96.5% of cases in TTP, 95.5% of cases in TTP-R, 99.6% of cases in Truck, 96.4% of cases in Truck-R, and 95.8% in Wargus. Finally, compare DMT with M5P, which is the Weka implementation of model trees. DMT is again dominant in the TPP, TPP-R, Truck-R, and Wargus domains. In the Truck domain, DMT is dominant for both precision and RRSE; however in recall M5P has 116 wins to DMT's 276 wins. This is a victory for DMT, however by a smaller margin than in other domains and measures.

Table 4.2 shows the statistical wins, losses and ties for DMT with pre-pruning versus DMT with post pruning. These tables are in the same format as Table 4.1 with wins describing those cases where DMT with pre-pruning out-performs DMT with post-pruning. For domains TPP and TPP-R, we see that the post-pruning algorithm does better in precision and RRSE, while it does so at a cost to recall. This suggests that the post-pruning algorithm was too aggressive in this domain and could be further tuned with a validation. In the Truck domain, the pre-pruning algorithm wins overall on RRSE but loses in terms of recall. In the Truck-R domain, post pruning wins in all 3 categories, but the margin is small in recall. In

Table 4.1: Statistical wins, losses and ties for DMT versus 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		
	W	L	T	W	L	T	W	L	T	W	L	T
Prec	734	0	386	26	2	1092	887	6	227	356	1	763
Recall	385	1	674	34	0	1026	421	0	639	384	1	675
RRSE	821	12	287	79	23	1018	1081	19	20	517	24	579
	TTP-R											
	DMT-F			DMT-S			CART			M5P		
	W	L	T	W	L	T	W	L	T	W	L	T
Prec	721	1	398	19	10	1091	863	27	230	370	12	738
Recall	393	1	666	35	0	1025	441	0	619	380	0	680
RRSE	797	25	298	53	25	1042	1070	34	16	522	37	561
	Truck											
	DMT-F			DMT-S			CART			M5P		
	W	L	T	W	L	T	W	L	T	W	L	T
Prec	692	8	1398	0	17	2081	1428	6	664	576	0	1522
Recall	299	120	1604	84	0	1939	370	93	1560	276	116	1631
RRSE	1178	84	838	179	21	1900	2083	10	7	837	35	1228
	Truck-R											
	DMT-F			DMT-S			CART			M5P		
	W	L	T	W	L	T	W	L	T	W	L	T
Prec	630	20	1449	34	77	1988	1301	105	693	511	27	1561
Recall	298	40	1686	142	0	1882	440	30	1554	369	54	1601
RRSE	1124	105	871	195	57	1848	2025	46	29	798	56	1246
	Wargus											
	DMT-F			DMT-S			CART			M5P		
	W	L	T	W	L	T	W	L	T	W	L	T
Prec	291	16	756	0	16	1047	497	14	552	247	12	804
Recall	66	16	982	1	7	1056	118	15	931	65	15	984
RRSE	745	49	270	415	24	625	1019	33	12	779	77	208

Table 4.2: Statistical wins, losses and ties for DMT versus the DMT with the post pruning algorithm 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			TTP-R		
	W	L	T	W	L	T
Prec	0	72	1048	0	113	1007
Recall	111	0	949	118	0	942
RRSE	58	103	959	42	88	990

	Truck			Truck-R		
	W	L	T	W	L	T
Prec	18	14	2066	59	100	1940
Recall	74	108	1841	41	62	1921
RRSE	201	106	1793	69	143	1888

	Wargus		
	W	L	T
Prec	68	37	958
Recall	16	28	1020
RRSE	520	76	468

the Wargus domain, the pre-pruning algorithm wins in both RRSE and precision.

Table 4.1 hides the effect of increasing sample size. To visualize this, Figure 4.1 shows the win/loss/tie percentages as a function of training set size for Recall comparing DMT versus CART, and Figure 4.2 shows the same for precision. Each vertical bar is divided into three parts indicating wins, losses, and ties (reading from bottom to top). In all cases, there are very few if any losses, which means that DMT's precision and recall are almost always better than or equal to CART's precision and recall. Note also that as the size of the training set gets large, we observe more ties in the recall plots for all domains but Wargus. This is

because as CART receives more training data, it produces larger trees that include more variables. We see in Figure 4.2 that, for small input sizes, CART produces models with comparable precision but as the size of the input set increases, CART's precision falls as it attempts to model the more complex interactions between variables.

Figure 4.3 presents learning curves for RRSE. In the interest of conciseness, we show one variable-action pair from each domain rather than listing plots for every possible variable and action. For the TPP-R Supply variable (Purchase action), we see that for training sets of size 8 and above, DMT has the lowest RRSE; followed by DMT-S, DMT-p (full DMT with post pruning) and M5P. DMT-F and CART perform the worst. For the TPP domain, a similar ordering occurs for training sets of size 8 and above. For the Truck variable Product Location (Unload action), DMT has the lowest RRSE until 8 trajectories, when it is overtaken by DMT-p and DMT-f. This suggests not only that using relational splits is critical in this domain, but that the use of functional leaves can be detrimental unless a post pruning method is used. This graph shape is observed across all Product Location models in Unload actions. In the Truck-R domain, we see that DMT-p has the lowest RRSE, joined by DMT at a training set size of 4, and DMT-F at a training set size of about 16. All other methods have higher RRSE scores. As in the Truck domain, we see that both relational splits and post pruning are useful. The added stochasticity of the Truck-R domain forces the pre-pruning methods to grow the tree large enough to create a more accurate model. Finally, for the Wargus Agent Resource variable (Harvest Wood action), the DMT variants have

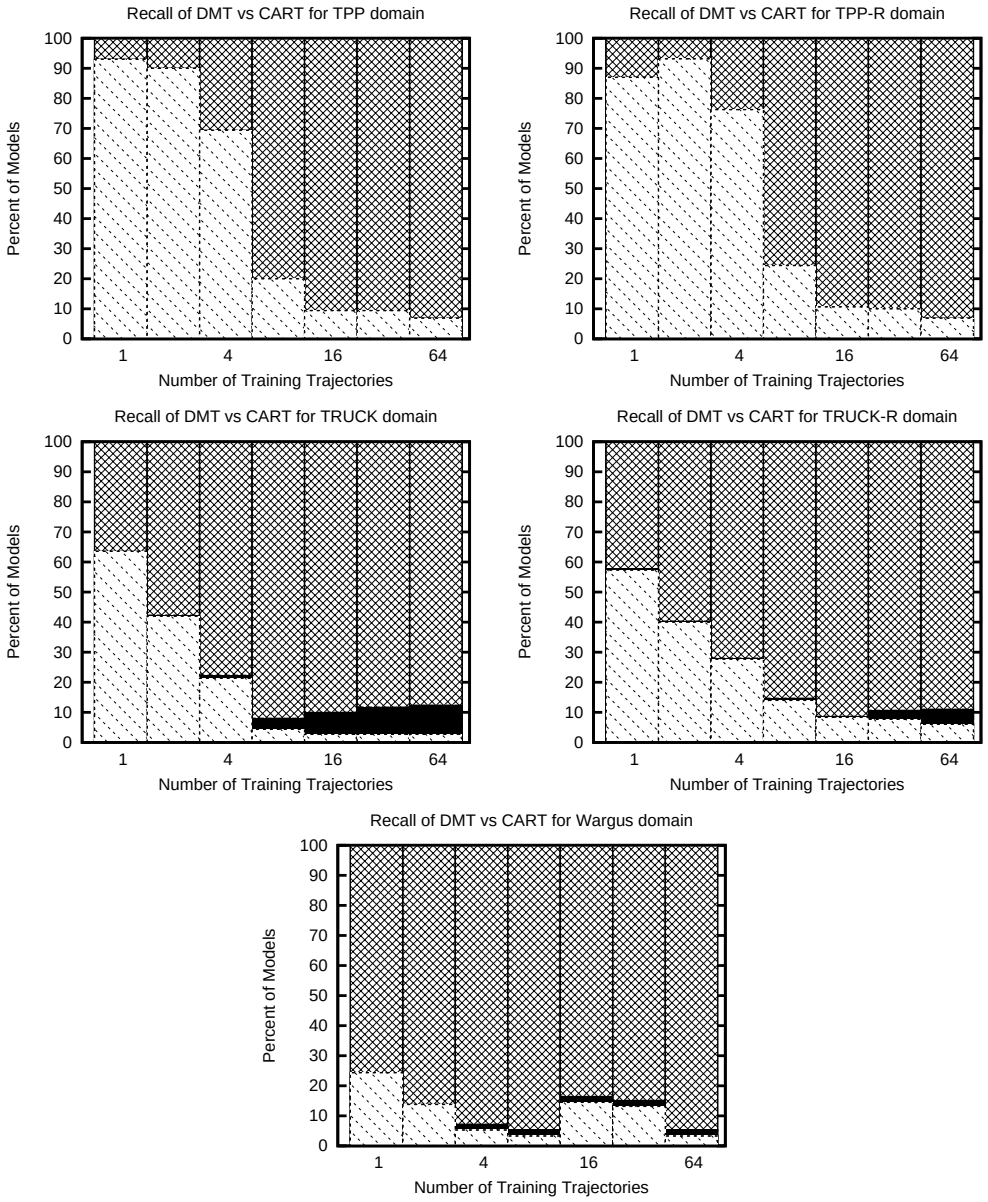

Figure 4.1: Recall of DMT vs. CART for state variables for TPP, TPP-R, Truck, Truck-R, and Wargus. Each bar is divided into three sections (wins, losses, and ties). Wins are the bottom bar with the lightest texture, ties are the top bar with the gray texture, and losses are in the middle with a heavy black texture.

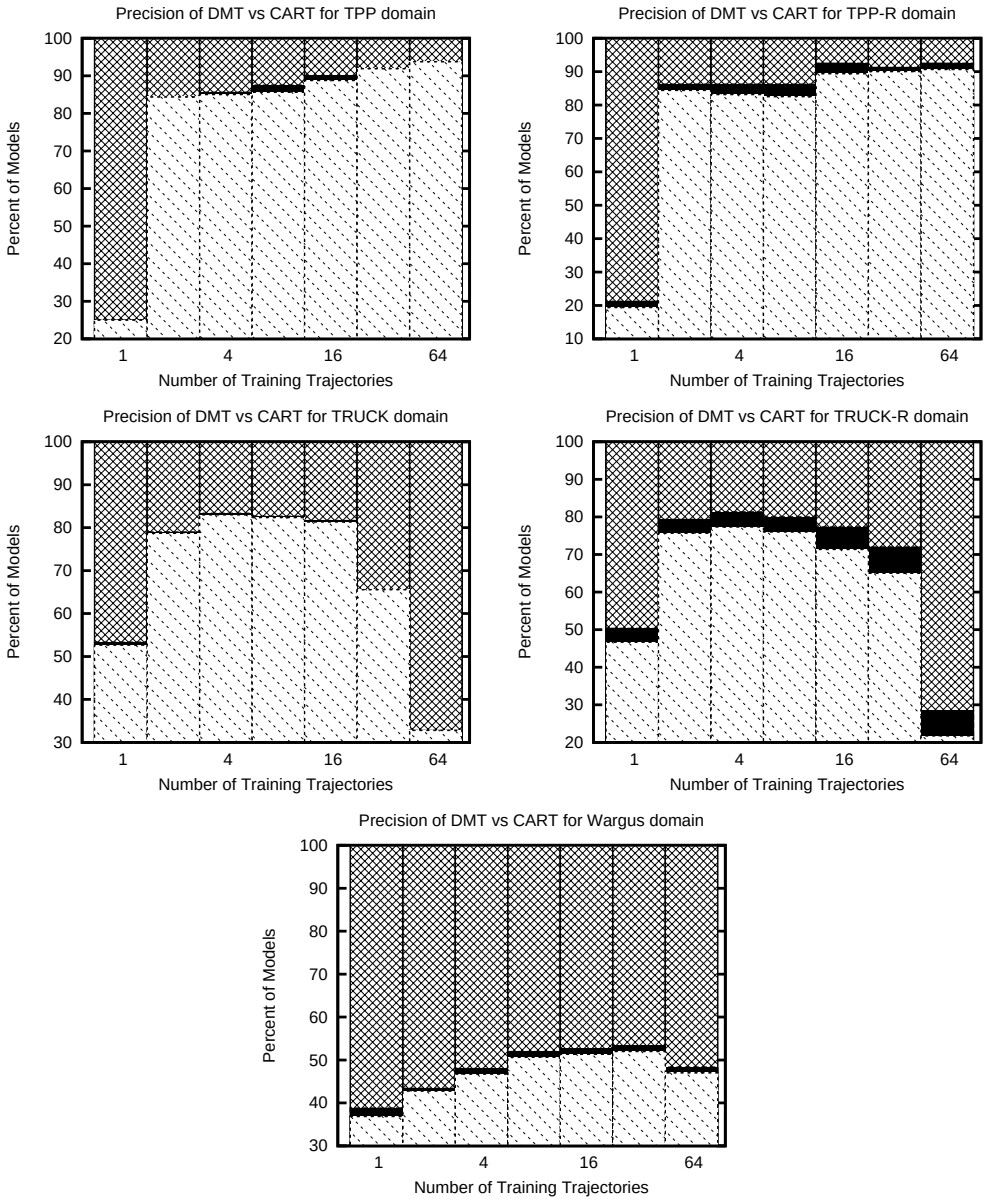

Figure 4.2: Precision of DMT vs. CART for state variables for TPP, TPP-R, Truck, Truck-R, and Wargus. Each bar is divided into three sections (wins, losses, and ties). Wins are the bottom bar with the lightest texture, ties are the top bar with the gray texture, and losses are in the middle with a heavy black texture.

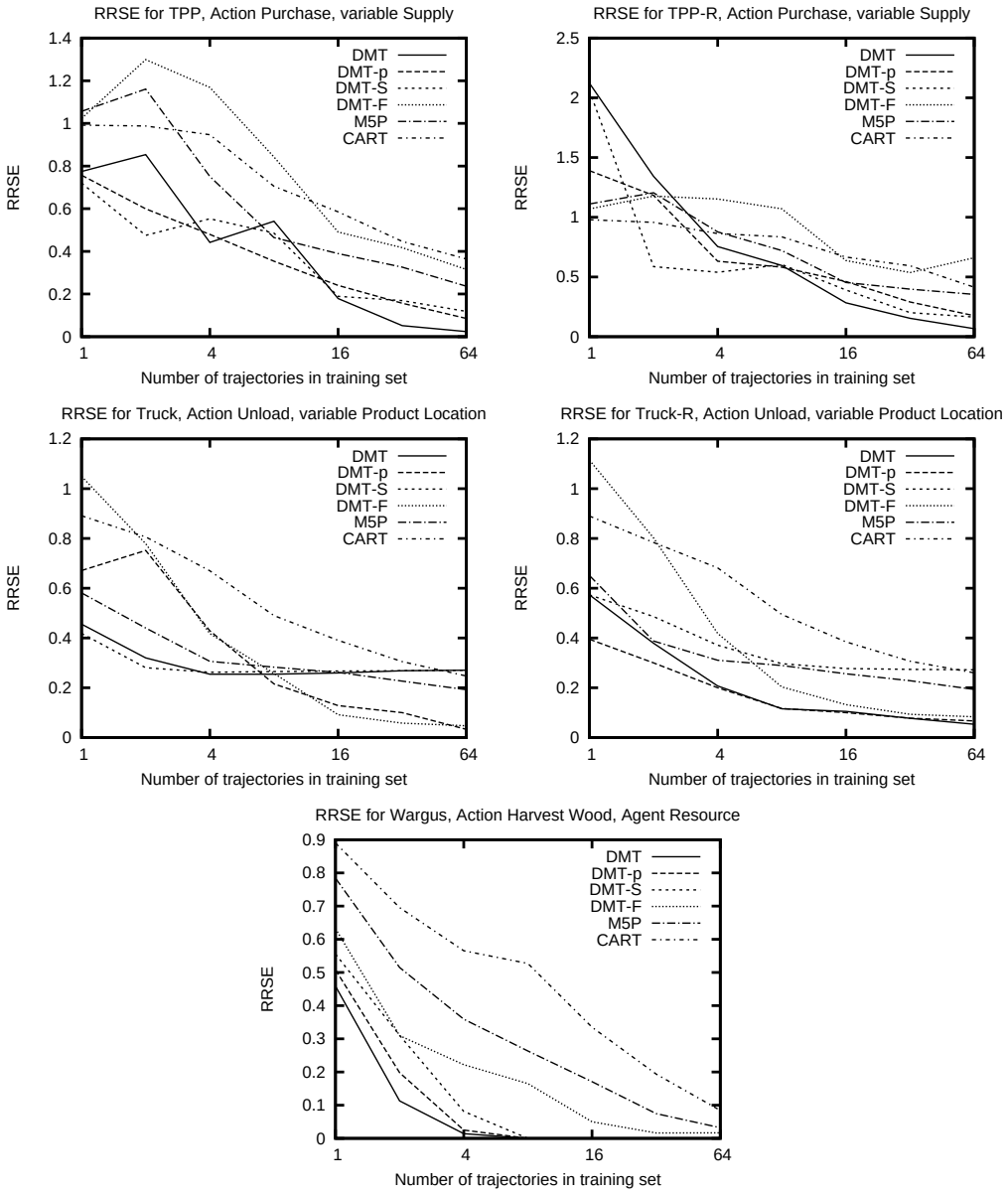

Figure 4.3: RRSE as a function of the number of trajectories in the training set for one chosen action and variable in each domain. Top: RRSE for Market Supply for the Purchase action in TTP and TPP-R; Middle: RRSE for Product Location for the Unload action in Truck and Truck-R; Bottom: RRSE of the Agent Resource for the Harvest Wood action in Wargus

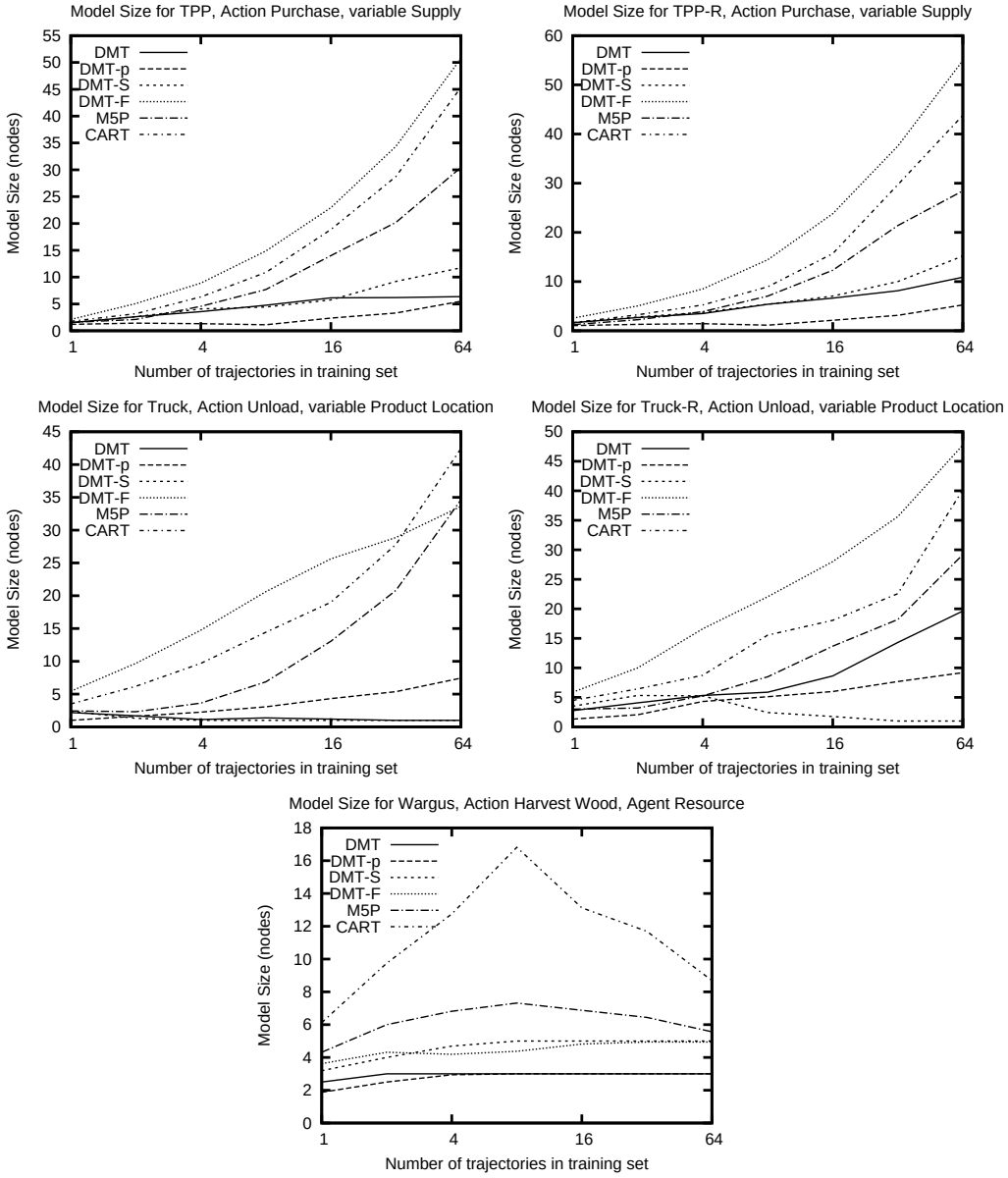

Figure 4.4: Model size in nodes as a function of the number of trajectories in the training set for one chosen action and variable in each domain. Top: Model size for Market Supply for the Purchase action in TTP and TPP-R; Middle: Model sizae for Product Location for the Unload action in Truck and Truck-R; Bottom: Model size of the Agent Resource for the Harvest Wood action in Wargus

the lowest RRSE, starting with DMT, then DMT-p and DMT-S, all three of which converge to zero RRSE by training set size of 8. DMT-F comes next, followed by M5P and CART. The ability to use functional splits and mixtures of functions in the leaves gives the best results in this domain.

Shown in Figure 4.4 are the model sizes for the variable-action pairs depicted in the corresponding RRSE plots shown in Figure 4.3. For the Supply variable (Purchase action) in the TPP and TPP-R domains, DMT-p performs the best followed closely by both DMT and DMT-S. M5P, CART and DMT-F produce the largest models. In the Truck domain's Unload action (Truck Area variable), DMT and DMT-S produce the smallest models, converging to a single node, while DMT-p makes slightly larger models and M5P, CART and DMT-F bring up the rear. Viewing the difference in size here between DMT with pre-pruning and DMT-p, which implements post-pruning, we see that the ability to examine the entire tree before pruning allows for more accurate models in many cases. In the Truck-R domain for the same actions, we see that DMT-S creates the smallest models, followed by DMT-p, DMT, M5P, CART and DMT-F. It is worth noting here that while DMT-S creates the smallest model, its RRSE score for this action is the worst of the DMT variants, and while DMT-F converges to join DMT and DMT-p in terms of RRSE, it creates the largest model. Finally, in the Agent Resource node for the Harvest Wood action in Wargus, we see that DMT-p and DMT create the smallest model, quickly leveling out to three nodes, followed shortly by the other DMT variants, with M5P and CART rounding out the set. CART has a dramatic spike in model size, this is most likely the result of a tree pruning within CART.

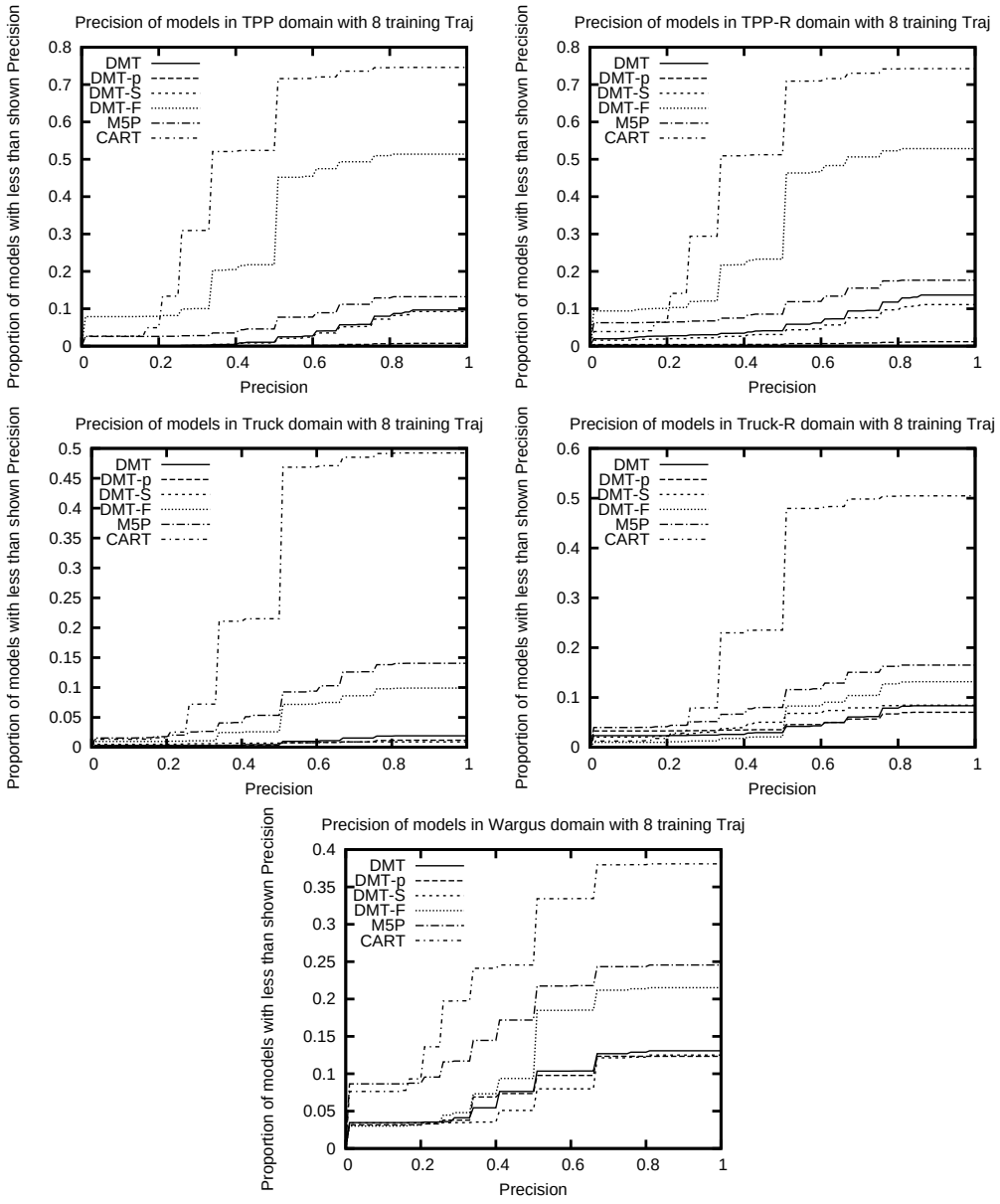

Figure 4.5: Precision profiles for each domain when trained on 8 trajectories and compared to the true DBN models. These curves aggregate over all variables, actions, and training sets in each domain. Each plotted point specifies the fraction of learned models with Precision less than the value specified 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 of 1.0.

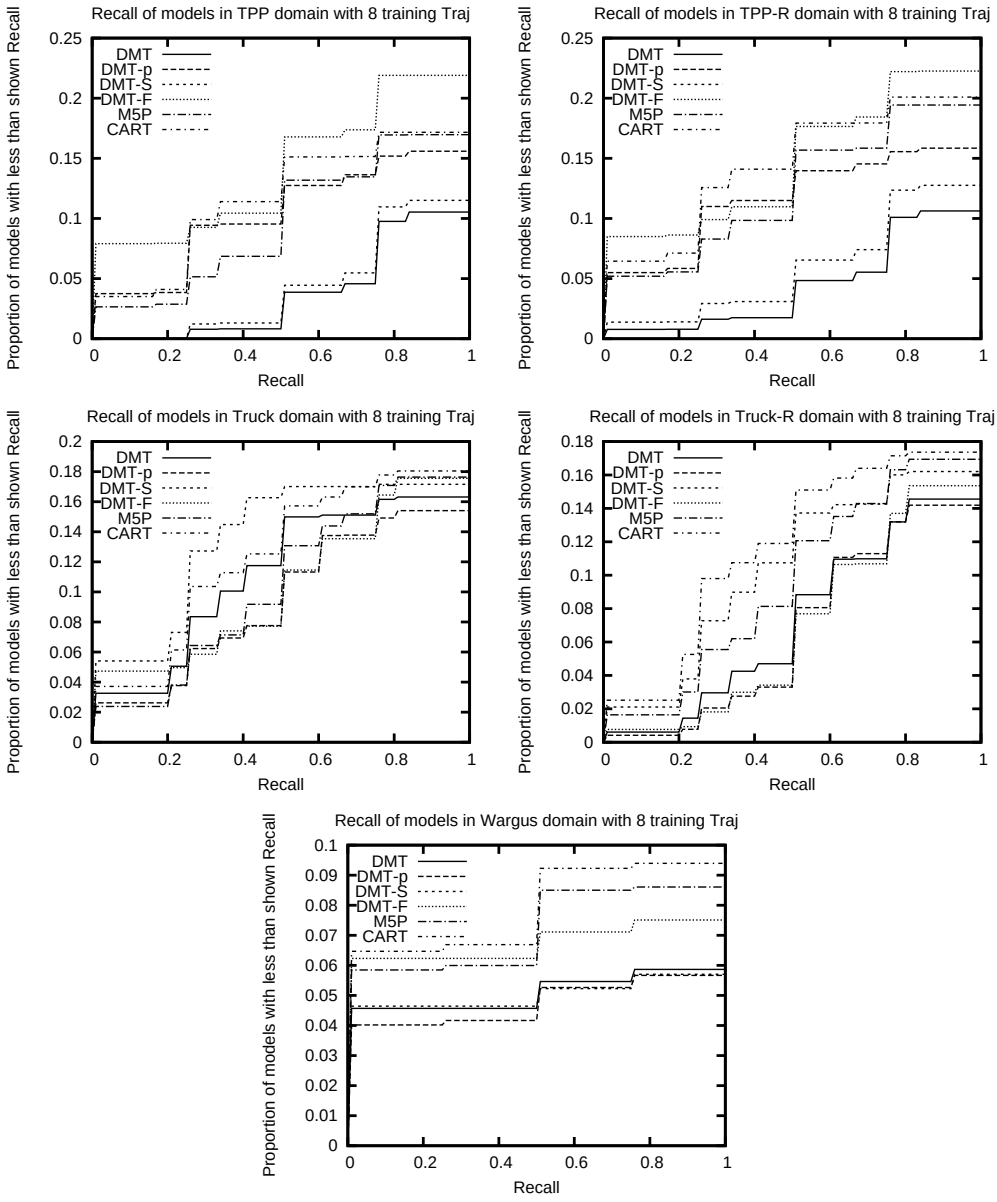

Figure 4.6: Recall profiles for each domain when trained on 8 trajectories and compared to the true DBN models. These curves are generated in the same manner as Fig. (4.5)

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 profiles shown in Figures 4.5 and 4.6 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 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 4.5 shows these profiles for precision over cases where the training set contains 8 trajectories, and Figure 4.6 shows the same for recall. We chose this as the middle point on the learning curves (with respect to log sample size).

For the TPP and TPP-R domains shown in Figure 4.5, DMT and DMT-S track each other very closely and are consistently superior to all of the other algorithms other than DMT-p, which follows the bottom axis very closely with less than 1% of trials having a precision of less than 1.0. Only 10% of DMT and DMT-S trials had a precision of less than 1.0. M5P comes next with excellent Precision. CART

had the worst precision, and DMT-F was also quite bad. For Recall, shown in Figure 4.6, we see that DMT and DMT-p have better recall overall than the other methods. However, DMT is consistently better than DMT-p. This indicates that the post-pruning methods were more aggressive than the pre-pruning methods in this domain.

As in the TPP domains, CART gives extremely bad precision in the truck domains. About half of the runs had Precision of around 0.5 or less. All of the other methods do much better with DMT, DMT-S and DMT-p reporting a precision score of 1.0 in almost all cases. In Truck-R, the results are similar, but with DMT, DMT-S and DMT-p converging to about 10% of cases below a precision score of 1.0. On recall, all of the algorithms do fairly well with, DMT-p doing the best in the end, followed closely by DMT and DMT-F. The other algorithms performed slightly worse.

Finally, for Wargus DMT-F has the among best Precision for low values but the third-worst Precision at high values, where it performs worse than the other DMT variants which dominate the precision scores. For Recall, DMT-p starts out with the best score, but soon converges with DMT and DMT-S at about 6% of cases less than a recall score of 1.0. This is followed by DMT-F, M5P and CART whom all perform worse, but still respectable with less than 10% of cases reporting less than 1.0 recall.

Chapter 5 – Conclusions

This thesis has presented a new algorithm, Discrete Mixture Trees, 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. To combat overfitting, two methods of pruning are introduced, one that focuses on stopping the tree growth early, and another that allows the full model to be built before pruning back to a more succinct and correct tree.

Experiments on three challenging domains, two with stochastic variants, 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 four of the domains, DMT is clearly superior to CART and M5P. In the fifth domain (Wargus), there are many cases where DMT performs well, but not by as large a margin as in the other domains.

Experiments also included a more advanced form of pruning than was implemented on DMT and its variants. This post-pruning technique allows for informative branches to be present in the tree that would have been removed by the

pre-pruning method. In three domains (TPP, TPP-R and Truck-R), the post-pruning algorithm performed better in terms of precision. In Truck and Wargus, DMT performed better, although in Truck the wins and losses are very close. Similarly, in recall DMT-p wins in 3 domains but loses to DMT in both TTP and TPP-R. This suggests that the post pruning algorithm itself is not well tuned for the TPP domain. However it does perform well in general. Much like other pruning methods, our post pruning algorithm has a tunable parameter that can be determined via the use of a validation set.

In future work, we plan to use the TDBNs learned by DMT as input to the MAXQ discovery algorithm developed by Mehta et. al. (2007). We would also like to study methods analogous to the Pessimistic Pruning algorithm employed by Ross Quinlan in C4.5. Finally, we are interested in extending model trees to handle mixtures of fitted linear models in the leaves.

Bibliography

5th International Planning Competition (2006). International Conference on Automated Planning and Scheduling.

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

Boutilier, C., Dearden, R., & Goldszmidt, M. (1995). Exploiting structure in policy construction. *IJCAI-95* (pp. 1104–1111).

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

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

Chickering, D. M., Heckerman, D., & Meek, C. (1997). A bayesian approach to learning bayesian networks with local structure. *UAI-97* (pp. 80–89).

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

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

Friedman, N., Murphy, K., & Russell, S. (1998). Learning the structure of dynamic probabilistic networks.

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

Johnson, D. S. (1973). Approximation algorithms for combinatorial problems. *STOC-73* (pp. 38–49). New York, NY: ACM.

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

Kramer, S. (1996). Structural regression trees. *AAAI-96* (pp. 812–819). Menlo Park: AAAI Press.

Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, *22*, 79–86.

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

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

Mehta, N., Ray, S., Tadepalli, P., & Dietterich, T. (2008). Automatic discovery and transfer of MAXQ hierarchies. *Proceedings of the 25th Annual International Conference on Machine Learning (ICML 2008)* (pp. 648–655). Helsinki, Finland: Omnipress.

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

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

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

Slavík, P. (1996). A tight analysis of the greedy algorithm for set cover. *STOC-96* (pp. 435–441). New York, NY: ACM.

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

Torgo, L. (1997). Functional models for regression tree leaves. *Proc. 14th International Conference on Machine Learning* (pp. 385–393). Morgan Kaufmann.

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