Decision Tree Methods for Finding Reusable MDP Homomorphisms

Alicia Peregrin Wolfe[*] and **Andrew G. Barto** [†]
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01035, USA
{pippin, barto}@cs.umass.edu

Abstract

State abstraction is a useful tool for agents interacting with complex environments. Good state abstractions are compact, reuseable, and easy to learn from sample data. This paper combines and extends two existing classes of state abstraction methods to achieve these criteria. The first class of methods search for MDP homomorphisms (Ravindran 2004), which produce models of reward and transition probabilities in an abstract state space. The second class of methods, like the UTree algorithm (McCallum 1995), learn compact models of the value function quickly from sample data. Models based on MDP homomorphisms can easily be extended such that they are usable across tasks with similar reward functions. However, value based methods like UTree cannot be extended in this fashion. We present results showing a new, combined algorithm that fulfills all three criteria: the resulting models are compact, can be learned quickly from sample data, and can be used across a class of reward functions.

Introduction

In this paper, we focus on state abstraction, specifically, Markov Decision Process (MDP) homomorphisms (Ravindran 2004). In general, a homomorphism is mapping from one mathematical structure to another, possibly many to one, which preserves some properties or operations on the original structure. In the case of MDP homomorphisms, the mapping is from the states and actions of one MDP to the states and actions of another abstract MDP, and the properties preserved are the transition and reward functions. Due to this preservation of reward and transition function, this representation also preserves the *value function* over states, which can be used to construct a policy for acting in the environment.

Methods like UTree (McCallum 1995), which can be used to find state abstractions from sample interactions with the environment, focus directly on modeling the value function. Although this can result in compact models it has certain drawbacks. The homomorphism framework, because it retains the transition and reward functions directly, can be easily extended to form models that can be used for more than one reward function and task. These models, which are formed by what we will term Controlled Markov Process (CMP) homomorphisms, model some specific output function over the state space rather than a specific reward function. A single output function supports a family of related reward functions—those which are some function of the output function. Consider a navigating robot: one task might be to get to a particular location, while an output function that could support this goal as well as many others would simply give the robot's current location. The model appropriate for a specific task may be more compact than the model for the general output function, but the output function model can be reused in multiple tasks.

The UTree algorithm is quite appealing, however, in that the whole structure of the underlying MDP, whether expressed as an MDP or as a dynamic Bayes network (DBN), need never be known. Most current homomorphism finding techniques start with a complete model in one of these two forms, then reduce the model to a simpler abstract MDP. When using Utree, however, only those features relevant to the task at hand are included, and the agent need not do more exploration than is necessary to learn the abstract model: there is no need to learn the complete model. This paper therefore modifies the algorithm to find CMP homomorphisms, by including the transition probabilities and output probabilities that support a specific output function, while retaining the advantages of the decision tree approach.

The resulting algorithm tackles both of the basic problems facing any agent that performs autonomous abstraction—how to learn compact, but reusable models of an environment from data as quickly and accurately as possible.

Background

A Markov Decision Process (MDP) consists of tuple (S, A, T, R) comprising a state set (S), action set (A), transition function ($T : S \times A \times S \to [0, 1]$), and expected reward function ($R : S \times A \to \mathbb{R}$). The transition function defines the probability of transitioning from state to state given the current state and chosen action, while the reward function represents the expected reward the agent receives for being in a particular state and executing an action.

An MDP homomorphism (Ravindran 2004) is a mapping, $h : S \times A \to S' \times A'$, from the states and actions of a

[*]This research was facilitated in part by a National Physical Science Consortium Fellowship and by stipend support from Sandia National Laboratories, CA.

[†]This research was funded in part by NSF grant CCF 0432143.

base MDP, $M = (S, A, T, R)$, to an abstract model MDP $M' = (S', A', T', R')$. To be an MDP homomorphism, h must preserve both the reward function and some properties of the transition probabilities of M. Specifically, h consists of a set of mappings: $f : S \rightarrow S'$, and for each $s \in S$ a mapping $g_s : A \rightarrow A'$ that recodes actions in a possibly state-dependent way. The following properties must hold, for all state and action pairs:

$$R'(f(s), g_s(a)) = R(s, a) \quad (1)$$

$$T'(f(s_i), g_{s_i}(a), f(s_j)) = \sum_{s_k | f(s_j)=f(s_k)} T(s_i, a, s_k). \quad (2)$$

Subgoal *options* (Sutton, Precup, & Singh 1999) provide a formalism for specifying multiple episodic subtasks within an MDP. In addition to a reward function R over the state space, a subgoal option includes a termination condition $\beta : S \times A \rightarrow [0, 1]$ which specifies the probability that the option will terminate in any particular state. Homomorphisms for subgoal options include one additional constraint over the mapping h (Ravindran & Barto 2003):

$$\beta'(f(s), g_s(a)) = \beta(s, a). \quad (3)$$

When a mapping f can be found that is many-to-one, the abstract MDP M' has fewer states than M. The homomorphism conditions mean that M' accurately tracks the transitions and rewards of M but at the resolution of blocks of states, assuming some appropriate action recoding. Each block is a set of states that f maps to the same state of M'. These properties guarantee that policies optimal for M' can be *lifted* to produce optimal policies of the larger MDP M (Ravindran 2004).

Several algorithms exist for finding MDP homomorphisms given a model of an MDP. All proceed by partitioning the state set S in stages. At each stage, the states and actions are partitioned into two sets of blocks: a state (S) partition $\{B_1, ...B_m\}$ over states, and a state/action (SA) partition over (s, a) pairs, $\{P_1...P_n\}$. The S partition defines an f mapping: $s \in B_i \rightarrow f(s) = s'_i$. Similarly, the SA partition defines the set of g_s mappings: $(s, a) \in P_i \rightarrow g_s(a) = a'_i$.

The partitioning algorithm starts with an initial SA partition that follows from Equations 1 & 3: all state/action pairs in the same block P_i have the same expected reward and termination probability. Next, the model is repeatedly refined in a loop that alternately updates the S and SA partitions in order to satisfy Equation 2.

Each iteration first defines a new S partition, based on the current SA partition: any two states s_i and s_j, are placed the same state block B_i if and only if they are members of the same set of SA blocks (i.e., $\forall P_k, \exists a | (s_i, a) \in P_k \leftrightarrow \exists a | (s_j, a) \in P_k$). Using the new state labels defined by f, the blocks of the SA partition are next refined to satisfy Equation 2, and the algorithm repeats. When the blocks of the S partition do not change from one iteration to the next, the mapping is a homomorphism.

This version of the algorithm is taken from Ravindran (2004), though similar examples exist in Givan, Dean, & Greig (2003) and Boutilier, Reiter, & Price (2001).

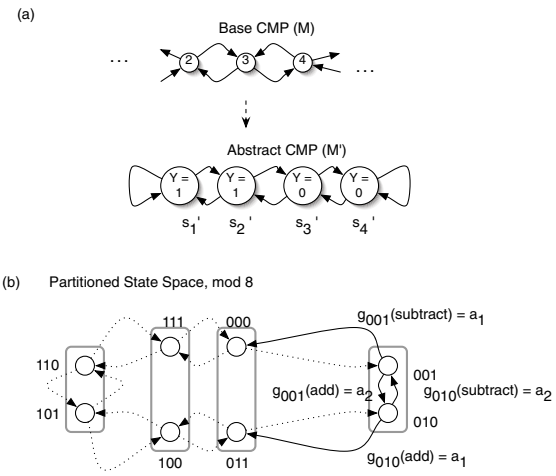

Figure 1: Abstract model produced by the homomorphic image of an integer counter (a), where the output variable is the 3rd least significant bit. Action set: {add 1, subtract 1}. The final reduction from 8 states to 4 is detailed in (b).

CMP Homomorphisms

A CMP is an MDP without the latter's reward function. A CMP with output is a CMP together with an output function that maps its state set to a set of outputs, or observations, Y. We think of the output function as singling out some aspect of the CMP as being of interest for some reason. This function might be as simple as the location or color of an object in the state. Thus, a CMP with output is a tuple (S, A, T, y), where S, A, and T are as in an MDP, and y is the output function $y : S \times A \rightarrow Y$. Given any function $r : Y \rightarrow \mathbb{R}$, $(S, A, T, r \circ y)$ is an MDP whose reward function is the composition of r and y. We say that this MDP is *supported by* y. This means that the reward depends only on the observations, not on the complete state. The termination conditions for the family of subgoal options supported by y have the form $\beta : Y \mapsto [0, 1]$.

A CMP homomorphism is a mapping from a CMP with output (S, A, T, y) to an abstract CMP with output (S', A', T', y'). The mapping functions h, f, and g_s are defined as for MDP homomorphisms. The conditions that the resulting model must satisfy in order to form a CMP homomorphism are similar to those for an MDP, with the constraints over the expected value of the reward for a state (Equation 1) and termination function β (Equation 3) replaced by a single constraint over the output function:

$$y'(f(s), g_s(a)) = y(s, a). \quad (4)$$

The transition function constraints (Equation 2) remain the same. The model formed by a CMP homomorphism can be used to learn the value function for any reward function $r \circ y$ and termination function $\beta \circ y$.

Figure 1 illustrates the abstract model formed by a homomorphic mapping on a binary integer counter with actions "add 1" and "subtract 1", where the output function is the 3rd least significant bit. The CMP reduces to a 4-state ab-

Algorithm 1 Decision Tree Algorithm

```
for t ← 1 to T do
    output[t] ← y(s_t)
    // all states initially have the same label
    nextStateLabels[t] ← 0
end for
tree ← rootNode()

stateLabelsChanged ← true
while stateLabelsChanged do
    repeat
        // classification with the current target
        improvement ← doBestSplit(tree, output, nextState-
        Labels)
    until (improvement < threshold)
    // update the target concept
    stateLabelsChanged ← updateNextStateLabels(tree,
    nextStateLabels)
end while
```

stract CMP. It is easy to see that all higher order bits are irrelevant to the task of predicting the 3rd bit, but the final reduction from 8 states to 4 is less clear. Consider an integer counter with only three bits, shown in Figure 1 (b). In order to reduce to 4 states, use the following action mapping: if bit 2 = 0, "add" $\mapsto a_2$ and "subtract" $\mapsto a_1$. If bit 2 = 1, "subtract" $\mapsto a_2$, and "add" $\mapsto a_1$ to produce the state space partition shown.

Existing methods for finding MDP homomorphisms apply with trivial modifications to the case of a CMP with output. The initial split on average reward must simply be replaced by a split that clusters the states according to the distribution of the output function.

Some supported reward functions may be representable using a smaller abstract model. This method therefore makes a compromise between finding a model which is compact and reusable, which depends on the output functions chosen.

Decision Tree Based Algorithm

Since the goal is to build an abstract model from sample data in time proportional to the size of the abstract model—not the entire MDP—existing state space partitioning algorithms are insufficient. Consider the integer counter from Figure 1. If the output feature is the third bit of the counter, with an optimal algorithm the agent can accurately learn the model in the same amount of sample interactions it would take to learn the model of a three bit counter. However, any metho requiring a model of the entire CMP to form the mapping would need to explore the entire (infinite) state space.

Algorithm 1 is a decision tree based algorithm similar to the Utree algorithm. It splits the state space using feature splits in a decision tree, considering features from both the action features set X_A and state feature set X_S. The difference is that rather than choosing splits to improve the prediction of the *value* of the next state, it chooses splits to improve the prediction of the abstract state label of the next state.

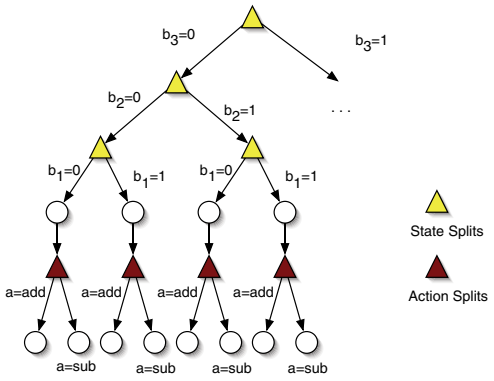

Figure 2: Portions of the decision tree for the counter from Figure 1. Action feature splits are strictly below state feature splits.

The algorithm learns a decision tree of the form shown in Figure 2. Note that all state feature splits are strictly above action feature splits. All states classified by the state leaf l_j are members of block B_j in the S partition. Similarly, all state/action pairs classified by state/action leaf l_i are members of SA block P_i. The parameters for the abstract CMP for the tree can be estimated by counting observed transitions between tree leaves.

The sample set consists of transitions sampled from some large MDP. Each sample is a tuple (s_t, a, s_{t+1}) composed of an initial state s_t, action and next state s_{t+1} for time step t. The state and action feature sets may include features over history, if the agent is in fact in a POMDP. The algorithm is also given the output function y over the state set.

The core of the algorithm is a decision tree algorithm with two target concepts: $y(s_t)$ for the initial state of each sample, and $f(s_{t+1})$ for the next state in each sample. There are two loops in the algorithm: the inner loop builds the classification tree, the outer loop uses the resulting f function to create new next state labels. The task of predicting the next state becomes more difficult with each iteration of the outer loop, as the tree refines the state space. The procedure stops when the state space can predict both $f(s_t)$, and $y(s_{t+1})$. At this point, f does not change from one iteration of the outer loop to the next.

Splitting Leaves ($doBestSplit$)

The choice of a decision tree algorithm implies certain limitations that cannot be ameliorated by the choice of leaf-splitting criteria. If two or more features $\{x_1...x_k\} \in X_S \cup X_A$ produce no change in output and transition probabilities when chosen individually, but do improve prediction when selected as a group, no decision tree algorithm will select them. The abstract model formed in this case will not satisfy Equation 4 and will not be Markov. Methods which use an exact model of the entire CMP do not have this problem, as they compare states and actions directly rather than splitting on features individually.

In order to form a compromise between these two meth-

ods, we will consider splits up to depth of k in the tree, proceeding to splits of depth j only if there are no splits of depth $j-1$ which are relevant.

Within these limitations the range of algorithms able to find accurate models is wide. Any algorithm which continues splitting on features until none improve prediction accuracy will find a model satisfying Equations 2 & 4 within the limitations of the choice of k. For these experiments, the Chi Squared significance test was used to determine whether a split made a significant difference in accuracy.

The order in which feature splits are chosen determines the size of the decision tree. A greedy approach based on Information Gain is commonly used in decision trees, and is the metric we use here. For action leaves, the calculation of Information Gain for an action feature $x_j \in X_A$ at action leaf a_i is simple: $I_{a_i}(x_j) = \sum_l H(o_l) - H(o_l|x_j)$, where $H(o_l)$ is the entropy of the target concept $o_l \in \{y(s_t), f(s_{t+1})\}$, and $H(o_l|x_j)$ is the conditional entropy of the target concept after the split.

State splits split the leaves of the state classification subtree, and may split several action leaves at a time. The gain for the split must be calculated over this set of action leaves. Ideally the gain at each action leaf would be weighted by the probability of that leaf, however, this probability may change as the policy changes for different tasks. Therefore, we assume that the specific action with the greatest gain is the most important action to improve and use its gain measure for the state leaf. The gain for a split on the state feature $x_j \in X_S$ at leaf s_i is therefore: $I_{s_i}(x_j) = max_{a_i \in subtree(s_i)} I_{a_i}(x_j)$, where $subtree(s_i)$ is the set of all action leaves with s_i as an ancestor.

Time Complexity

The algorithm performs at most l splits, where l is the number of leaves. For each split $O(w^k)$ possible feature combinations are tested on each leaf, where w is the number of feature values. In practice, it is only practical to maintain a set of exemplar samples of a constant size c at each leaf, where c depends on the amount of data needed at a leaf for this MDP. Thus, while the amount of exploration required to gather sufficient samples for each leaf depends on the MDP, the size of the sample set considered at each iteration of the algorithm is at most $c \cdot l$. The Information gain and Chi squared calculations are linear in the number of samples at a leaf, therefore, the time to process a leaf is $O(w^k \cdot c)$ and the overall time complexity is $O(l^2 \cdot c \cdot w^k)$.

Overall then, the worst case time complexity of a single execution of the algorithm does not depend on the size of the original MDP. The number of times the algorithm is executed depends on the number of interactions necessary to get a sufficiently large sample size.

Experiments

The first experiments shown used the integer counter CMP with 10 bits and two actions, add 1 and subtract 1. Actions were noisy: with probability 0.2 an add action instead did a subtraction, and similarly for subtract.

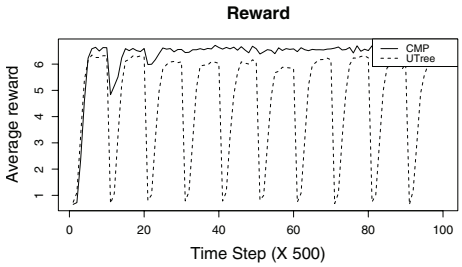

Figure 3: Average reward per time step for UTree and CMP homomorphism abstractions, for each 500 time step interval.

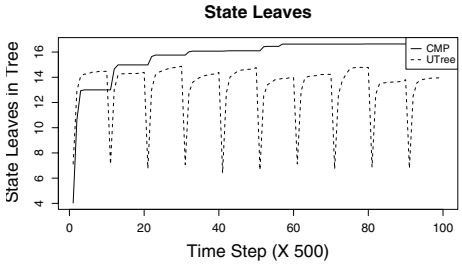

Figure 4: The size of the state space used by UTree and CMP homomorphism abstractions in the Integer Counter example.

The output function for these experiments singled out the 3rd and 4th least significant bits. The test scenario generated tasks where the agent's goal was a specific setting of these bits. This setting generated a reward of 40 and terminated the current episode. All other transitions generated a reward of -1. The agents were presented with 10 randomly generated tasks in sequence, with 5000 time steps in which to learn each task. At the start of each new task, the UTree algorithm created a new abstraction tree.

Learning within each task proceeded in stages: every 500 steps, the agent ran its respective decision tree algorithm. The resulting tree was then used to create a policy for the next 500 time-step interval. Random choices were used over action features that were not selected in the tree.

Each leaf of the CMP tree kept $c = 250$ exemplar transition samples. A setting of maximum feature depth $k = 1$ was sufficient for the CMP Tree, but a setting of $k = 2$ was necessary for the UTree algorithm. We hypothesize that this is due to the fact that UTree initially makes distinctions only between states with distinct rewards. States in which the agent achieves the goal are rare and difficult to predict.

The results, averaged over 100 runs, are shown in Figures 3 & 4. Peak performance is similar for the two algorithms, although UTree typically finds slightly smaller models. UTree failed to find the correct model in a few cases, perhaps indicating that its parameters need to be adjusted. CMP performance dips with the start of each of the first 3 or 4 tasks, as the state set is more thoroughly explored and the model grows. With the addition of 4 or 5 states over the size

of the UTree the CMP algorithm gains a model that can be reused for the remainder of the tasks.

The UTree variant used here was as close as possible to Algorithm 1, in order to clearly compare using the next state label as the classification target to using the value of the next state as the classification target. The Information Gain ordering function for splits of Algorithm 1 was replaced with the difference in expected value, the Chi-Squared test was replaced with the Kolmogorv-Smirnoff test with a cutoff of 0.95 and the state classification function f was replaced with a procedure that calculates the value of the next state for each sample.

To avoid overfitting in the CMP homomorphism finding algorithm, the Chi Squared measure was set quite high, to 0.99, and a cutoff of 0.1 was used on the Information Gain measure. Selecting extra features in the CMP algorithm is very expensive: consider what happens when the algorithm selects bit 10 in the integer counter example. Because the bit is now part of the state description, the algorithm tries to predict it, using bit 9, bit 8, etc. In one case out of 100 trials, even with stringent overfitting controls, the state space grew to 80 states. In no case did the algorithm fail to find the necessary state splits. The optimal state set size was 16: all of the other runs found this optimal state size.

Utree and related algorithms do not have as much of a problem with overfitting because they predict the value of the next state, rather than the next state label. This has the effect of merging state leaves as the estimate of the value function improves. If the split on bit 10 was due to noise in the samples, as more data is gathered and the value function estimates improve, corresponding states with the same true value merge, in effect, since these leaves have the same the state value labels.

One solution to this problem in the CMP algorithm might therefore be to run the original homomorphism finding algorithms on the model specified by the tree as more data is added, labeling multiple leaves with the same abstract state or state/action block label or pruning the tree.

Blocks World

The second environment we tested was a Blocks World environment. Each Blocks World task consisted of 3 blocks, each of which could have one of 4 colors and be in one of 4 piles. There were 16 actions: move the top block in each of the 4 piles to another pile.

Each episode started with a new Blocks World, consisting of three blocks of random color stacked in random piles. The output variable was the position (height and pile) of a specific block—the focus block. The decision tree built a general description of a Blocks World to predict focus block position.

The reward functions for the tasks ranged over the 16 possible target block positions, with a positive reward of 100 for achieving the desired focus block position and reward of -1 on all other transitions.

State features were the height, pile index, and color of the blocks. The features for the focus block were distinct from the features for the other blocks. Features for non-focus blocks were relational, in that splitting on a feature like "height = 2" merely indicates the existence of an object matching that feature. Once a feature of an object has been selected, the object description can be refined with additional features. If, for example, the feature "height = 2" has been selected, the tree can be further split with the linked feature "pile Index = 1" to indicate specifically that there is a block at height 2 in pile 1.

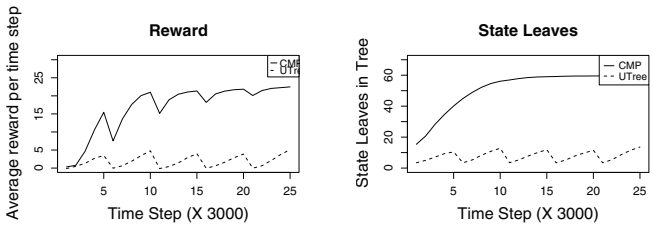

Figure 5: Average reward per time step and state space size for UTree and CMP homomorphism abstractions in the Blocks World example.

Action features were the index of the pile from which a block is taken, and the index of the pile onto which the block was deposited. If there was no block in the "from" pile, or if the action failed (prob = 0.2) the state remained unchanged.

Learning curves are shown in Figure 5. The interval between tree learning sessions was lengthened here to 3000 time steps, with 15000 steps total on each of 5 tasks.

The optimal state description for the CMP contained 60 states. In these experiments, both algorithms used $k = 1$, which plainly did not allow the UTree algorithm to find all the splits necessary to perform well. Preliminary experiments with $k = 2$ showed no improvement for UTree. The problem seems to be with difficult tasks, in which the target block position involved stacking the focus block on top of the two other blocks. The rewarding states in this scenario are hard to reach, making samples with reward rare. The policies and state sequences which lead to these states are also complex. It seems that one or both of these aspects of the task make it difficult for UTree to find an appropriate abstraction using these state and action features.

The noise features in this case were the block colors, as well as some action features. For the most part the algorithm correctly ignored these features.

Related Work

This paper merges two branches of well-studied research. Finding hidden states in POMDPs is a well-covered problem, going as far back as Whitehead & Ballard's (1991) work on the perceptual aliasing problem. The UTree algorithm is one of the more advanced of these methods. It has been extensively studied, both for POMDP state identification and MDP state abstraction for a fixed task, with many variants (McCallum 1995; Jodogne & Piater 2005; Jonsson & Barto 2001).

Partitioning the state space while preserving expected reward and transition functions via various related algorithms, all of which produce some type of homomorphism, has also

been extensively studied. The agent typically begins with some type of model of the complete system, either an MDP (Givan, Dean, & Greig 2003; Ravindran 2004), relational regression model (Boutilier, Reiter, & Price 2001) or DBN (Jonsson & Barto 2005; Ravindran 2004). Again, these methods focus on a specific task, though the models they build are in fact more general and could be used across tasks.

This work also shares some elements with the work of Hengst (2002) and Jonsson & Barto (2005).. Both of these algorithms focus on building hierarchies of tasks using output variables, from which options are constructed that reach each possible value of each variable. For output variables with many values, it may not be appropriate to cache policies for reaching every value. The cached policies also do not address more complex reward functions that do more than simply seek to reach a single value of the variable.

Discussion and Future Work

The underlying framework of this algorithm, of iteratively refining a model learned from samples to form a homomorphic mapping to a reusable abstract model, is general and powerful enough to be extended beyond the implementation outlined here. The decision tree implementation presented here provides a simple and straightforward example of this framework, appropriate for cases where the designer has knowledge of the feature set of the domain. In practice, real "agents" create new features as they create their models of the world. Ideally, in future the algorithm will construct its own feature set using samples and the output target as a labeled feature set in a classification task.

Approximate models can be produced by terminating the decision tree algorithm early. The approximate homomorphism model is the Bounded Parameter MDP model (Givan, Leach, & Dean 2000) . The range of possible values for each parameter under all policies are represented by upper and lower bounds. The bounds for the model in the nth iteration of the outer loop of the decision tree algorithm can be found by examining the $n + 1$th iteration.

The structure of the tree presents some clear possibilities for exploration policies: leaves with too few samples must be visited before splits can be evaluated. There are some deeper issues as well, however. Any accuracy or error measurements in the tree are based upon the sampling policy used to collect data, which determines the visitation rates for states and actions in the underlying MDP. Different sampling policies may result in smaller or larger decision trees by changing the order of features chosen. One of the remaining questions is whether an optimal sampling procedure for determining the most compact model can be approximated using the model at intermediate stages of the algorithm.

The clearest question this work raises, however, is: what are the optimal output functions for a particular space of tasks? What is the optimal trade-off between compression and generality? It would be more satisfying if the agent could see N tasks, and then generate $N' << N$ output functions which support the reward functions for most future tasks. Until then, if the designer can identify a small set of output functions which support a wide range of rewards, it will be beneficial to the agent.

References

Boutilier, C.; Reiter, R.; and Price, B. 2001. Symbolic dynamic programming for first-order mdps. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 690–697.

Chapman, D., and Kaelbling, L. P. 1991. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, volume 2, 726–731.

Ernst, D.; Geurts, P.; and Wehenkel, L. 2005. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research* 6:503–556.

Givan, R.; Dean, T.; and Greig, M. 2003. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence* 147(1-2):163–223.

Givan, R.; Leach, S. M.; and Dean, T. 2000. Bounded-parameter markov decision processes. *Artificial Intelligence* 122(1-2):71–109.

Hengst, B. 2002. Discovering hierarchy in reinforcement learning with hexq. In *Proceedings of the 19th International Conference on Machine Learning*, 243–250.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, 527–534.

Jodogne, S., and Piater, J. H. 2005. Interactive learning of mappings from visual percepts to actions. In *Proceedings of the 22nd International Conference on Machine Learning*.

Jonsson, A., and Barto, A. 2001. Automated state abstraction for options using the u-tree algorithm. In *Advances in Neural Information Processing Systems 13*, 1054–1060.

Jonsson, A., and Barto, A. 2005. A causal approach to hierarchical decomposition of factored mdps. In *Proceedings of the 22nd International Conference on Machine Learning*, volume 22.

Kersting, K., and Raedt, L. D. 2003. Logical markov decision programs. In *Proceedings of the 20th International Conference on Machine Learning (ICML-2003)*.

McCallum, A. K. 1995. *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. Dissertation, Rutgers University.

Poupart, P., and Boutilier, C. 2002. Value-directed compression of pomdps. In *Advances in Neural Information Processing Systems 15*, 1547 –1554.

Ravindran, B., and Barto, A. G. 2003. Smdp homomorphisms: An algebraic approach to abstraction in semi markov decision processes. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 1011–1016. AAAI Press.

Ravindran, B. 2004. *An Algebraic Approach to Abstraction in Reinforcement Learning*. Ph.D. Dissertation, University of Massachusetts.

Sutton, R. S.; Precup, D.; and Singh, S. P. 1999. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112:181–211.

Whitehead, S., and Ballard, D. 1991. Learning to perceive and act by trial and error. *Machine Learning* 7:45–83.

Wu, J.-H., and Givan, R. 2005. Feature-discovering approximate value iteration methods. In Jean-Daniel Zucker, L. S., ed., *Abstraction, Reformulation and Approximation: 6th International Symposium*.