
**School of Informatics, University of Edinburgh**

**Institute of Perception, Action and Behaviour**

# Adaptive Abstraction for Model-Based Reinforcement Learning

by

Mykel J. Kochenderfer

**Informatics Research Report EDI-INF-RR-0806**

**School of Informatics** **April 2006**
http://www.informatics.ed.ac.uk/

# Adaptive Abstraction for Model-Based Reinforcement Learning

Mykel J. Kochenderfer

Informatics Research Report EDI-INF-RR-0806

SCHOOL *of* INFORMATICS
Institute of Perception, Action and Behaviour

April 2006

**Abstract :** This paper presents a novel model-based reinforcement learning framework called the Adaptive Modelling and Planning System (AMPS). The challenge of a model-based reinforcement learning agent is using experience in the world to generate a model. In problems with large state and action spaces, the agent must generalise from limited experience by grouping together similar states and actions, effectively partitioning the state and action spaces into finite sets of regions. Several different abstraction approaches have been proposed in the literature, but the existing algorithms have many limitations. They generally only increase resolution, require a large amount of data before changing the abstraction, do not generalise over actions, and are computationally expensive. AMPS aims to solve these problems using a new kind of approach.

AMPS splits and merges existing regions in its abstraction according to a set of heuristics. The system introduces splits using a mechanism related to supervised learning and is defined generally, allowing AMPS to leverage a wide variety of representations. The system merges existing regions when an analysis of the current plan indicates that doing so could be useful. Because several different regions may require revision at any given time, AMPS prioritises revision to best utilise whatever computational resources are available. Changes in the abstraction lead to changes in the model, requiring changes to the plan. AMPS prioritises the planning process, and when the agent has time, it replans in high-priority regions. This paper demonstrates the flexibility and strength of this approach in learning intelligent behaviour.

**Keywords** : reinforcement learning, abstraction, semi-Markov decision process

# 1 Introduction

One of the most fundamental problems in artificial intelligence is the control of an agent situated in a world. It is common to frame the problem of intelligent control in terms of reinforcement learning because of its flexibility in modelling a broad class of problems. In reinforcement learning tasks, the agent receives reward while interacting with the environment. The objective of the agent is to behave in such a way so as to maximise its expected accumulation of reward.

For many reinforcement learning problems, the agent does not possess a complete understanding of the world dynamics or the distribution of reward. The agent must therefore learn from experience how to accomplish its task. The literature contains a wide variety of methods for learning how to solve reinforcement learning problems from experience. Some methods involve learning an explicit model of the dynamics and reward while others do not. This paper takes a model-based approach.

One of the primary challenges for both model-based and model-free approaches to reinforcement learning is generalising from limited experience. When the state and action spaces are large, it can be unlikely that the agent finds itself in the same state twice. When the agent encounters a state it has not encountered before, it should use information it knows about related states to decide how to proceed.

There are many ways to achieve generalisation in reinforcement learning, but the adoption of a model-based approach to reinforcement learning generally requires the use of abstraction. Abstraction involves partitioning the state and action spaces into regions and treating the regions collectively. The literature contains many suggestions on how to go about creating and revising abstractions as the agent acquires experience. Unfortunately, the existing approaches have some limitations. They generally only increase resolution, require a large amount of data before changing the abstraction, do not generalise over actions, and are computationally expensive.

This paper introduces the Adaptive Modelling and Planning System (AMPS), a flexible framework that addresses many of the limitations of existing abstraction approaches. AMPS splits and merges existing regions in its abstraction according to a set of heuristics. The system introduces splits using a mechanism related to supervised learning and is defined generally, allowing AMPS to leverage a wide variety of representations. The system merges existing regions when an analysis of the current plan indicates that doing so could be useful. Because several different regions may require revision at any given time, AMPS prioritises revision to best utilise whatever computational resources are available. Changes in the abstraction lead to changes in the model, requiring changes to the plan. AMPS prioritises the planning process, and when the agent has time, it replans in high-priority regions.

Although this work incorporates many ideas explored by other researchers, it makes several important contributions. An important contribution of AMPS is the view that both modelling and planning can be performed as interleaved prioritised processes, allowing the model and plan to be adapted when the agent has only short, periodic time slices available for computation. This paper also advances the view that because representation is critical to generalisation, the details of the representation should be abstracted away and the mechanism for interacting with the representation should be self-contained in an independent module. The way in which AMPS combines supervised and unsupervised learning techniques for the purpose of abstraction in reinforcement learning is novel.

This paper proceeds as follows. The next section reviews reinforcement learning. Section 3 introduces how AMPS approaches abstraction in reinforcement learning, and Section 4 explains how AMPS introduces perceptual and actional distinctions. Section 5 discusses how AMPS manages the complexity of the model it learns. Section 6 explains how the principles introduced in the earlier sections are integrated into an implemented system for learning intelligent behaviour. Section 7 discusses related work. Section 8 evaluates the approach presented in this paper. The final section concludes and presents ideas for further work.

# 2 Reinforcement Learning

The objective of reinforcement learning (surveyed by Sutton & Barto, 1998; Kaelbling, Littman, & Moore, 1996) is to learn to behave in such a way so as to maximise the accumulation of reward. Reinforcement learning approaches operate with the assumption that the system dynamics follow some class of model,

typically a Markov decision process (MDP). AMPS uses a continuous-time generalisation of an MDP known as a semi-Markov decision process (SMDP).[1]

After reviewing SMDPs, this section explains how one may use dynamic programming to compute optimal policies for SMDPs. This section then discusses issues with estimating the parameters of an SMDP from experience. This section concludes with a description of an adaptive prioritised value iteration algorithm that AMPS uses to update its reactive plan as it accumulates experience.

## 2.1 Semi-Markov Decision Processes

SMDPs specify how the state of the world evolves over time in response to the actions taken by the agent. In this paper, $\mathbb{S}$ represents the state space and $\mathbb{A}$ represents the action space. For many problems it is useful to restrict the actions available from particular states, so $\mathbb{A}(s) \subset \mathbb{A}$ represents the set of actions available from state $s$.

As the agent interacts with the world, it receives positive or negative reward. It is common to discount this reward at some continuous compound discount rate $\beta \in (0, \infty)$. Any reward received at time $t$ is discounted by a factor $e^{-\beta t}$, thereby encouraging the agent to pursue reward more aggressively.

### 2.1.1 Dynamics

In an SMDP, if an agent takes an action $a$ in some state $s$, it will transition to a new state $s'$ selected from a fixed probability distribution depending only on $s$ and $a$. The duration of time spent in this transition and the reward received is also selected from fixed probability distributions. In particular:

- $P(s' \mid s, a)$ is the probability that taking action $a$ in state $s$ will result in a transition to state $s'$.
- $P_t(t \mid s, a, s')$ is the probability that the transition from $s$ to $s'$ by action $a$ completes within time $t$.
- $r(s, a, s')$ is the expected reward when transitioning from $s$ to $s'$ by action $a$.

It is common for agents to receive reward at some rate while making a transition. Generalising the equations in this paper to allow reward rates in addition to lump-sum rewards is not difficult (see Kochenderfer, 2006).

The literature often assumes that the probability distributions for duration and reward do not depend on the state to which the agent transitions, i.e. $P_t(t \mid s, a, s') = P_t(t \mid s, a)$ and $r(s, a, s') = r(s, a)$. However, in some problems it is important that this probability distribution depends upon the new state $s'$.

The experience of the agent may be written down as a sequence,

$$(s_1, t_1, a_1, r_1, s_2, t_2, a_2, r_2, \ldots),$$

where $s_k$ is the state, $t_k$ is the time, and $a_k$ is the action of the $k$th decision. The resulting reward is given by $r_k$. The *discounted reward* during the $k$th transition is defined to be

$$R_k = e^{-\beta(t_{k+1} - t_k)} r_k.$$

### 2.1.2 Optimality

The objective of the agent is to find a policy, which is a mapping $\pi : \mathbb{S} \to \mathbb{A}$, that maximises the expected discounted return. Optimality is defined using value functions. A *value function* is a mapping $V : \mathbb{S} \to \mathbb{R}$. The value function $V^\pi$ at a state $s$ is the expected discounted return when starting at state $s$ and following the policy $\pi$:

$$V^\pi(s) \equiv E\left\{ \sum_{k=1}^{\infty} e^{-\beta t_k} R_k \mid s_1 = s, a_k = \pi(s_k) \right\}.$$

The *optimal value function*, written $V^*$, is defined to be

$$V^* \equiv \max_{\pi \in \Pi} V^\pi,$$

[1] SMDPs have been considered in various texts including those of Howard (1971) and Puterman (1994, Chapter 11).

where $\Pi$ is the space of policies. An *optimal policy*, written $\pi^*$, is a policy that satisfies $V^* = V^{\pi^*}$.

Although *action-value functions* are not necessary to define optimality, they are useful when discussing solution methods. The action-value function $Q^\pi$ evaluated at a state $s$ and action $a$ is the expected discounted return when taking action $a$ from $s$ and then following $\pi$. The *optimal action-value function* $Q^*$ evaluated at a state $s$ and action $a$ is the expected discounted return when starting at state $s$, taking action $a$, and continuing with an optimal policy.

## 2.2 Dynamic Programming

Dynamic programming (Bellman, 1957) is an efficient way to compute optimal policies. When discussing dynamic programming for SMDPs, the following definitions are useful:

$$\gamma(s, a, s') \equiv \int_0^\infty e^{-\beta t} dP_t(t \mid s, a, s') \tag{1}$$

$$R(s, a) \equiv \sum_{s' \in \mathbb{S}} P(s' \mid s, a)\gamma(s, a, s')r(s, a, s') \tag{2}$$

The functions $\gamma$ and $R$ appear throughout this paper. The function $\gamma(s, a, s')$ is the discounted value of unit reward received after transitioning from $s$ to $s'$ by action $a$. The function $R(s, a)$ is the total expected discounted reward when starting in $s$ and executing action $a$ until transitioning to some other state.

The *Bellman update operator* is the mapping

$$BV(s) \equiv \max_{a \in \mathbb{A}(s)} \left[ R(s, a) + \sum_{s' \in \mathbb{S}} P(s' \mid s, a)\gamma(s, a, s')V(s') \right].$$

Because $B$ is a contraction mapping, it is possible to compute $V^*$ to any desired precision by repeatedly applying $B$ to any value function $V$. In other words, $B^k V \to V^*$ as $k \to \infty$ (for convergence proofs, see Kochenderfer, 2006). Solving for $V^*$ in this way is known as *value iteration*. Once $V^*$ is known, an optimal policy $\pi^*$ can be constructed as follows:

$$\pi^*(s) = \arg\max_{a \in \mathbb{A}(s)} R(s, a) + \sum_{s' \in \mathbb{S}} P(s' \mid s, a)\gamma(s, a, s')V^*(s').$$

## 2.3 Estimation

If the model of the system dynamics is unknown to the agent, then the agent can estimate it from experience. A common scheme in model-based reinforcement learning (Barto, Bradtke, & Singh, 1995) is to update a single point estimate of the model as the agent acquires experience.[2] So long as the agent uses a suitable estimation strategy and conducts sufficient exploration where sub-optimal actions are taken with non-zero probability (Thrun, 1992), then the estimate $\hat{\theta}$ will converge strongly to $\theta$.

The parameters to be estimated from experience are $P(s' \mid s, a)$, $R(s, a)$, and $\gamma(s, a, s')$. One way of estimating a parameter from experience is with *maximum-likelihood estimation*. Maximum-likelihood estimation computes the most likely parameter setting given the data. The maximum-likelihood estimate for $P(s' \mid s, a)$ is

$$\hat{P}(s' \mid s, a) = n(s, a, s')/n(s, a),$$

where $n(s, a, s')$ is the number of times action $a$ was taken in $s$ resulting in a transition to $s'$ and $n(s, a)$ is the number of times action $a$ was taken in $s$. It can be shown by the strong law of large numbers that $\hat{P}$ converges almost surely to the true $P$.

Estimating $R(s, a)$ and $\gamma(s, a, s')$ is slightly more involved because they depend upon $r(s, a, s')$ and $P_t(t \mid s, a, s')$. There are two different ways to estimate $R$ and $\gamma$. The first method involves computing $\hat{P}_t$ by estimating the parameters of an assumed distribution model. The second method does not require prior knowledge of the distribution model for $P_t$; instead $\hat{R}$ and $\hat{\gamma}$ are estimated directly. Both methods have their advantages and disadvantages.

[2] Kumar (1985) discusses an alternative, Bayesian approach. For complex problems the non-Bayesian approach of using a single point estimate of the model parameters is more commonly used.

### 2.3.1 Parametric Model Estimation

If $P_t$ follows a known parameterised distribution such as an exponential distribution, then maximum likelihood can estimate its parameters. With an estimate of the distribution, it is possible to integrate Equation 1 to estimate $\gamma$. It is possible to estimate $R$ by substituting the estimates of $\gamma$ and the maximum likelihood estimate of $r$ into Equation 2. The estimates for $R$ and $\gamma$ will strongly converge to their true values.

### 2.3.2 Nonparametric Model Estimation

This section considers the problem where $P_t$ follows some arbitrary unknown distribution. Instead of estimating this distribution, it is easier to estimate $\gamma$ and $R$ directly. Calculating $\gamma$ and $R$ involves integrating over probability distribution functions. Monte Carlo integration (see Gentle, 2002, Section 2.2) is one way to evaluate the integrals that define $\gamma$ (Equation 1). Monte Carlo integration approximates the evaluation of the definite integral involving the function $g(x)$ and cumulative distribution function $F(x)$ as follows:

$$\int_{-\infty}^{\infty} g(x)dF(x) \approx \frac{1}{n}\sum_{k=1}^{n} g(x_k),$$

where the samples $\{x_1, \ldots, x_n\}$ are selected from $F(x)$. The estimate converges as the number of samples increases. The agent may incrementally update its estimate of $\gamma$ after observing the completion of the $k$th transition and updating $n(s_k, a_k, s_{k+1})$:

$$\hat{\gamma}(s_k, a_k, s_{k+1}) \leftarrow \hat{\gamma}(s_k, a_k, s_{k+1}) + \left(e^{-\beta t_k} - \hat{\gamma}(s_k, a_k, s_{k+1})\right)/n(s_k, a_k, s_{k+1}).$$

The agent may use $\hat{\gamma}$ to estimate $R$. After some algebraic simplification

$$\hat{R}(s, a) = \frac{1}{n(s, a)} \sum_{s' \in \mathbb{S}} \hat{\gamma}(s, a, s')\sigma_r(s, a, s'),$$

where $\sigma_r(s, a, s')$ is the sum of the reward received when transitioning from $s$ to $s'$ by action $a$.

## 2.4 Adaptive Prioritised Value Iteration

Full value iteration is not necessary every time the agent updates its estimate of the model. It is far more efficient to use *adaptive prioritised value iteration*, which focuses computational effort on updating the value function at high-priority states. Adaptive prioritised value iteration is how AMPS plans over the model it learns.

Moore and Atkeson (1993) suggest an adaptive prioritised value iteration algorithm called *prioritised sweeping* for MDPs. Their heuristic prioritisation scheme is based on how much the value function changed in the past. Wiering (1999) and Wingate and Seppi (2005) suggest alternative prioritisation heuristics.

Algorithm 1 is a general adaptive prioritised value iteration algorithm. It uses a priority queue that supports the following operations:

- INSERT$(H, s, p)$ inserts state $s$ into the queue $H$ with priority $p$.
- EXTRACT-MAX$(H)$ extracts the highest priority state from the queue $H$.
- CONTAINS$(H, s)$ indicates whether state $s$ is contained in the queue $H$.
- INCREASE-PRIORITY$(H, s, p)$ increases the priority of state $s$ to $p$ in the queue $H$.
- REMOVE$(H, s)$ removes state $s$ from the queue $H$.

The priority queue may be implemented efficiently with a relaxed heap (Driscoll, Gabow, Shrairman, & Tarjan, 1988) or a Fibonacci heap (Fredman & Tarjan, 1987). Each iteration of the adaptive prioritised value iteration algorithm involves extracting the highest priority state from the priority queue and calling UPDATE on that state.

The procedure UPDATE in Algorithm 1 may be implemented differently depending on the heuristic one wishes to adopt. Algorithm 2 is an implementation of the heuristic used by Moore and Atkeson (1993), generalised for SMDPs. If the state $s$ is chosen to be updated, the algorithm will update its value and also update the priorities of its predecessors (i.e. the states that lead immediately to that state). In order to perform the updates quickly, the algorithm maintains the predecessor set for each state. The predecessor set for a state $s$ is given by $\mathrm{pred}(s)$ and contains a set of tuples containing every state and action pair leading to $s$.

**Algorithm 1** Adaptive Prioritised Value Iteration

**loop**
  Execute an action according to some exploration strategy
  Update the model estimate according to the state transition and reward
  $s \leftarrow$ most recent state
  **if** CONTAINS$(s)$ **then**
    INCREASE-PRIORITY$(H, s, \infty)$
  **else**
    INSERT$(H, s, \infty)$
  **while** $H \neq \emptyset$ **do**
    $s \leftarrow$ EXTRACT-MAX$(H)$
    $\Delta(s) \leftarrow 0$
    UPDATE$(H, \Delta, s)$

**Algorithm 2** UPDATE$(H, \Delta, s)$

$v \leftarrow V(s)$
$V(s) \leftarrow \max_{a \in \mathbb{A}(s)} R(s,a) + \sum_{s' \in \mathbb{S}} P(s' \mid s,a)\gamma(s,a,s')V(s')$
**for all** $(s', a') \in \mathrm{pred}(s)$ **do**
  $p \leftarrow P(s \mid s', a')\gamma(s', a', s)\,|V(s) - v|$
  **if** $p > \Delta(s')$ **then**
    $\Delta(s') \leftarrow p$
    **if** $\Delta(s') > \epsilon$ **then**
      **if** CONTAINS$(H, s')$ **then**
        INCREASE-PRIORITY$(H, s', \Delta(s'))$
      **else**
        INSERT$(H, s', \Delta(s'))$

Provided that the agent employs an appropriate exploration strategy and its estimate of the world model strongly converges to the true model, adaptive prioritised value iteration will lead to strong convergence to the optimal value function (see Kochenderfer, 2006).

Unlike Moore and Atkeson (1993), AMPS plans over regions of the state and action space, not individual states and actions. The next section explains how AMPS dynamically partitions the state and action spaces into regions.

# 3 Abstraction

One of the primary challenges in reinforcement learning problems is generalising from limited experience. Several generalisation methods have been suggested in the literature, including parametric approximation, local approximation, and abstraction. Parametric approximation involves approximating the optimal value function using a representation dependant on a parameter that is learned using a gradient descent procedure (Lin, 1992; Bertsekas & Tsitsiklis, 1996). Local approximation schemes use distance metrics to approximate the value function (Smart, 2002; Smith, 2002). Abstraction involves using a map to partition the state and action spaces into regions and treating the regions collectively in a model.

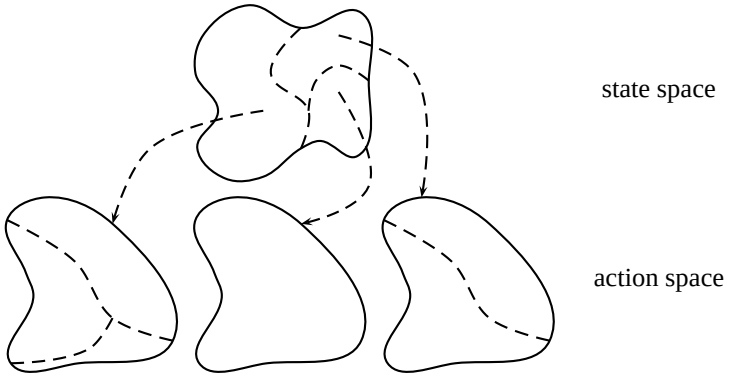



Figure 1: The relationship between state partitions and action partitions. Each region in the state space has its own partition of the action space. The region on the left side of the state space defines a partition of the action space with three regions, the region on the bottom right of the state space defines a partition with a single region, and the region on the top right of the state space defines a partition with two regions.

This paper focuses on abstraction because it allows for efficient model-based learning through dynamic programming. Several different abstraction methods that have been suggested in the literature are surveyed in Section 7. This section suggests an approach to abstraction that addresses many of the issues with existing algorithms.

## 3.1 Map Revision

An agent with limited computational resources must be able to efficiently and effectively use the experience it accumulates to revise the map that partitions the state and action spaces. It is important to note that there is no single "correct" way of partitioning the state and action spaces since any partitioning leads to a sensible SMDP. Therefore, it is necessary to rely upon heuristics to guide the adaptation of the map. AMPS adapts the partitions of the state and action spaces by splitting and merging regions according to a set of heuristics. AMPS also uses heuristics to prioritise revisions to the map. When time is available, the agent revises the highest priority regions. Sections 4 and 5 describe these heuristics in depth.

The map in AMPS defines a separate partition for each state region as Figure 1 illustrates. Other researchers (e.g., Smith, 2002) maintain only a single discretisation of the action space. It is useful for each state region to define its own partition of the action space because it allows greater generalisation. In some regions it is necessary to distinguish very finely between actions, but in other regions such high resolution distinctions are not useful.

This paper uses the following notation when discussing maps:

- $\mathcal{S}$ is the partition of the state space,
- $S(s)$ is the state region to which state $s$ belongs,
- $\mathcal{A}(S)$ is the action partition associated with the state region $S$, and
- $A(\mathcal{A}, a)$ is the action region in the partition $\mathcal{A}$ to which action $a$ belongs.

For convenience, $A(s, a)$ is shorthand for $A(\mathcal{A}(S(s)), a)$, which gives the action region associated with action $a$ from state $s$. Whatever data structure is chosen to implement the map, it is important that it be able to efficiently map states to their state regions and state-action pairs to their action regions. In addition, it is important that the data structure supports an efficient implementation of the following operations for revising the map:

- SPLIT$(S, S_1, \ldots, S_n)$, splits state region $S$ into some number of new regions such that each set of states $S_1, \ldots, S_n \subset S$ becomes part of separate state regions,

- SPLIT$(A, A_1, \ldots, A_n)$, splits the action region $A$ into some number of new regions such that each set of actions $A_1, \ldots, A_n$ becomes part of separate action regions,
- MERGE$(S_1, \ldots, S_n)$, merges the state regions $S_1, \ldots S_n$, and
- MERGE$(A_1, \ldots, A_n)$, merges the action regions $A_1, \ldots, A_n$.

The subscript $n$ in the description above may vary. For the SPLIT operations, the data structure is not obligated to split the sets of states perfectly. In fact, it might be desirable to not split the sets of states perfectly to prevent *overfitting*, which is a common problem in supervised learning. The implementation of the SPLIT operations actually performs a form of supervised learning. Splitting the sample states or actions involves learning a classifier that classifies the samples into different categories.

It is important that the data structures supporting the map be efficient and support arbitrary representations of states and actions. The current version of AMPS supports two kinds of implementations of the map, one based on decision graphs and the other based on nearest neighbour generalisation. These two kinds of data structures are particularly well-suited for AMPS because they can perform splits and merges efficiently and may be adapted to different kinds of state and action representations. In some domains, it is useful to combine different data structures to partition the different spaces. For example, nearest neighbour might partition the state space, but a decision graph might partition the action space.

The splitting operations require processing the underlying state and action representations. Because representation is key to successful generalisation, it is important to keep all representation-dependent implementation in a small, self-contained module that may be tailored to leverage the representation. The decision graph representation must have access to a module that produces tests, either binary or multi-valued, from examples. The nearest neighbour implementation must have access to a module that computes distances between two states or two state-action pairs. Details of how decision graphs and nearest neighbour support separating and merging states and actions are reserved for Sections 4 and 5.

### 3.2 Estimation Assumptions

With the state and action spaces partitioned into regions, the agent may estimate an SMDP over these regions using one of the strategies from Section 2.3. In an abstract SMDP, a transition from $S$ to $S'$ is a trajectory through states in $S$ to some state in $S'$. Hence, abstract SMDPs do not contain self-loops. In other words, $P(S \mid S, A) = 0$ for all state regions $S$ and action regions $A$.

Since the agent is able to sample the world only at some finite frequency, certain assumptions are necessary about what occurs between samples when estimating the model parameters. AMPS assumes that when transitioning from $S$ to $S'$, the first state sample in $S'$ is the first state in $S'$ the agent encountered. Hence, when estimating the duration required to transition from $S$ to $S'$, AMPS uses the duration of time from the first sample in $S$ until the first state sample in $S'$.

### 3.3 Termination

There are different ways to handle termination. One way is to add an *absorbing state region* $S_0$ where $P(S_0 \mid S_0, A) = 1$ for all action regions $A$. In other words, once the agent reaches $S_0$, it cannot exit. The agent receives zero reward indefinitely once reaching an absorbing state. The state region $S_0$ is *synthetic*; it does not correspond to a collection of actual sensor readings. A transition from state region $S$ to $S_0$ consists of the trajectory in $S$ until termination in $S$. It might be the case that the episode terminates immediately upon arrival to some region $S$. In this case, a duration of zero contributes to the estimate of time required to transition from $S$ to $S_0$. Only transitions to $S_0$ may be deterministically instantaneous, meaning $P_t(0 \mid S, A, S_0)$ may be 1 for some region $S$.

## 4 Distinction

In the absence of any prior knowledge about the dynamics of the world, the agent begins with the simplest possible model where all states belong to the same state region and all actions belong to the same action

region. As the agent tries to refine this model to accommodate its incoming stream of experience, it must introduce distinctions in the state and action spaces.

To illustrate the challenges of model revision, consider the robot football domain where the model might encode the hypothesis *if I am close to a ball and I kick then I will score a goal.* Further experience in the world might indicate that this hypothesis does not always hold and that the hypothesis should instead be *if I am close to a ball and I kick then I will score a goal half the time.* This hypothesis is valid and the agent is free to accept this nondeterminism as stochasticity inherent in the system. However, it is likely to be beneficial to the agent to make distinctions between states or actions. For example, the agent might revise its hypothesis to *if I am close to a ball and facing the goal and I kick then I will score a goal* or *if I am close to a ball and kick in the direction of the goal then I will score a goal.* Deciding when to accept non-determinism, make perceptual distinctions, or make actional distinctions is key to successful modelling and, therefore, successful behaviour.

The distinctions that are important to incorporate in a model are dependent on the task. If the objective is to score as many goals as possible, making perceptual distinctions with respect to the goal is essential for satisfactory behaviour. However, if the objective of the agent is to simply keep the ball away from another player, then there is no need to make perceptual distinctions with respect to the goal. It is important to avoid introducing more distinctions than necessary for the task because too many distinctions impair generalisation. The modelling process in AMPS uses heuristics to decide how and when to make distinctions.

Making distinctions in the state and action spaces requires computation on the underlying representation. AMPS limits all processing of the underlying representation to a self-contained module for two reasons. The first reason is that it enables AMPS to be applied to a wide variety of domains with only the representation-dependent module requiring special engineering. The second reason is that *representation is everything*. The successful application of any learning algorithm depends strongly on the representation. A domain expert can engineer the representation-processing models in AMPS to leverage the representation. Engineering the module is not necessarily difficult or time consuming. The current implementation of AMPS includes tools and algorithms for handling a variety of representations.

This section first introduces the heuristics used by AMPS to decide when and how to introduce distinctions in the state and action spaces. This section continues with a presentation of two very different approaches for introducing perceptual and actional distinctions. The first approach involves a decision graph where distinctions are made by tests that perform computation on the underlying representation of the states and actions. The second approach relies upon a distance metric for instance-based generalisation. Following this discussion, this section presents value-clipping, which is necessary for efficient planning when interleaved with incremental map revision.

## 4.1 Heuristics

There are two kinds of splitting heuristics in AMPS, *value revision* and *failure revision*. These heuristics attempt to separate groups of observed trajectories that exhibit different behaviour. The heuristics determine the grouping of trajectories into disjoint sets $T_1, \ldots, T_n$. This grouping is presented to the $\textsc{Split}(T_1, \ldots, T_n)$ function, which attempts to distinguish these trajectories by introducing a representation-dependent distinction in the state or action space as Section 4.2 later describes.

Both the value revision and failure revision heuristics look at states and actions involved in *greedy transitions*. Given a policy $\pi$ computed using the dynamic programming techniques of Section 2.4, a greedy transition from $S$ is one that transitions to another region $S'$ by actions in $\pi(S)$.

### 4.1.1 Value Revision

The objective of value revision is to ensure that all transitions resulting from a greedy action have approximately the same estimated value. Value revision attempts to separate state-action pairs in transitions with differing estimated value. The value of a transition from $S$ to $S'$ by $A$ is

$$Q(S, A, S') \equiv \gamma(S, A, S')r(S, A, S') + \gamma(S, A, S')V(S').$$

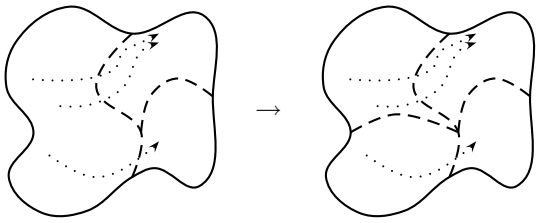


Figure 2: Value revision. On the left, two trajectories lead to a high-valued region, and one trajectory leads to a low-valued region. Because these trajectories are of different value, it is likely to be beneficial to split the source region as shown on the right.

At a region $S$ and for a greedy action $A$, the variation of $Q(S, A, S')$ over different successor regions $S'$ could be due to stochasticity or *aliasing*. Aliasing occurs when two states or actions that behave differently are grouped into the same region. If the variation of the transition values are significantly different, it might be beneficial to distinguish the states and actions involved in higher-valued transitions from those involved in lower-valued transitions. Figure 2 shows an example of how the state space might be split.

The agent must decide which trajectories should be kept together and which trajectories should be kept apart based only on their transition values. The problem reduces to *clustering*, a kind of unsupervised learning. There are many algorithms for grouping items into $k$ clusters as surveyed by Xu and Wunsch (2005). One of the difficulties of clustering is deciding how many clusters exist in the data. There have been many techniques proposed for determining the number of clusters (Milligan & Cooper, 1985). Many of the techniques involve incrementally increasing the number of clusters until there is a steep change in some criterion function. Everitt, Landau, and Leese (2001, Section 5.5) discuss such techniques further, including some more principled approaches for determining the number of clusters, but they are computationally expensive.

The current implementation of AMPS simply computes the mean value of the transitions from a state-action region, and it splits the trajectories involved in transitions according to whether their values are above or below the mean. Of course, this process produces only two clusters. Although the values plotted in the figure indicate the existence of only two clusters, this need not be the case. Splitting at the mean is generally a poor choice given the data from the point of view of optimising some criterion function. However, suboptimal clustering does not seem to impact the actual performance of the agent since it is given opportunities to later refine the clusters.

Because it is not practical for an agent to split every state that needs splitting while interacting with the world, the agent prioritises the splits. The priority of a split is related to the variability of the transition values. In AMPS, the priority of splitting at a state $S$ is given by the variance of $Q(S, \pi(S), S')$. When the agent decides to split $S$, it will compute the mean transition value and call SPLIT to separate the transitions whose values are no greater than the mean from those greater than the mean.

#### 4.1.2 Failure Revision

Failure revision uses a failure signal to detect whether the continual application of an action is likely to result in progress towards some goal. For example, in a wall-following task, the agent might have some bump sensor that indicates that it is against the wall and that continual forward motion is not possible. In Taxi World, the agent receives a failure signal when it tries to pick up a passenger who is not in the same cell as the taxi. In Blocks World, the agent encounters failure when attempting to put down a block it is not holding.

The Parti-Game algorithm (Moore & Atkeson, 1995) uses a similar signal to detect when the agent becomes "stuck" to revise its partition of the state space. TTree (Uther, 2002) detects deterministic self-transitions, an indication of failure, to determine when to split the state space. It is important that the agent has a failure signal, otherwise the agent does not know whether a transition is simply taking a long time or whether it is really stuck and should take another course of action.

The priority of failure revision at a particular state region $S$ is equal to the number of trajectories

resulting in failure divided by the total number of trajectories resulting in either failure or a successful transition to another region or termination. When the agent revises $S$, it will split transitions resulting in success from those resulting in failure.

## 4.2 Trajectory Separation

Having identified which trajectories need to be separated (using the heuristics in the previous section), the agent must decide how to separate the trajectories. The procedure $\textsc{Split}(T_1, \ldots, T_n)$ may introduce a distinction in the state space or in the action space to separate the states or actions involved in the trajectories belonging to different sets. If the agent decides to split state region $S$, it will call $\textsc{Split}(S, S_1, \ldots, S_n)$, where $S_k$ is a set consisting of the states in the trajectories of $T_k$. If the agent decides to split action region $A$, it will call $\textsc{Split}(A, A_1, \ldots, A_n)$, where $A_k$ is a set consisting of the actions in the trajectories of $T_k$.

The agent must rely on a heuristic to decide whether to split in the state space or action space. Information gain (Shannon, 1948) may be used as an indicator of which way of splitting is best. It might be the case that both ways of splitting give the same information gain, in which case the agent might use some measure of generalisation error such as cross-validation (Lachenbruch & Mickey, 1968) or bootstrapping (Efron & Tibshirani, 1993). Some kinds of bootstrapping, such as the 0.632 Bootstrap, provide poor estimates of generalisation error for nearest neighbour classification (Jain, Dubes, & Chen, 1987). Weiss (1991) empirically shows that stratified twofold cross-validation is the best generalisation estimator for nearest neighbour in comparison to other cross-validation and bootstrap techniques. Computing cross-validation error can be expensive, so it may be advantageous to simply split in the state space by default when splitting in the action space does not perfectly split the trajectories.

Once the agent decides whether to split in the state or the action space, it calls either $\textsc{Split}(S, S_1, \ldots, S_n)$ or $\textsc{Split}(A, A_1, \ldots, A_n)$ accordingly. The current implementation of AMPS includes two alternatives for supporting the SPLIT operations, one involving decision graphs and the other involving nearest neighbour metrics. Sections 4.3 and 4.4 discuss these approaches in depth. To keep the discussion generic, these sections refer to the splitting of general "objects" instead of specifically states or actions. So, instead of describing $\textsc{Split}(S, S_1, \ldots, S_n)$ and $\textsc{Split}(A, A_1, \ldots, A_n)$ separately, these sections describe the generic $\textsc{Split}(X, X_1, \ldots, X_n)$, which splits the sets of objects $X_1, \ldots, X_n$ belonging to region $X$.

The procedure $\textsc{Split}(X, X_1, \ldots, X_n)$ performs what may be viewed as supervised learning. Objects in the set $X_i$ may be thought of as training examples that belong to class $i$. The objective is to revise the map by dividing region $X$ into a set of new regions such that the training examples belonging to different classes get mapped to different regions. The split may not be perfect and it is important to avoid overfitting the training data.

Both the decision graph approach and the nearest neighbour approach are suitable for general representations. An object might be represented, for example, as a binary bit string, a vector of real values, a set of logical sentences, or XML. The decision graph and the nearest neighbour approaches require computation over the underlying representation of the objects. The decision graph computes the outcome of a test based on the underlying representation, and nearest neighbour computes distances based on the underlying representation.

## 4.3 Decision Graph Approach

One way to implement the map is with a decision graph. A decision graph can very quickly map a state or an action to a region through a series of tests. Figure 3 shows how a decision graph may partition both the state and action spaces. In general, as the previous section mentions, different data structures might partition the state and action spaces.

Introducing distinctions in a decision graph involves introducing a test at a leaf node. The agent wishes to introduce the highest quality test according to some measure. The success of the decision graph in splitting the objects depends on the available tests. For some problem representations, the space of possible tests is infinite. For other representations, the space of tests is finite. It may not be possible, or even desirable, to split the objects perfectly. For efficiency and to avoid overfitting, the agent should introduce simple tests that are fast to compute.

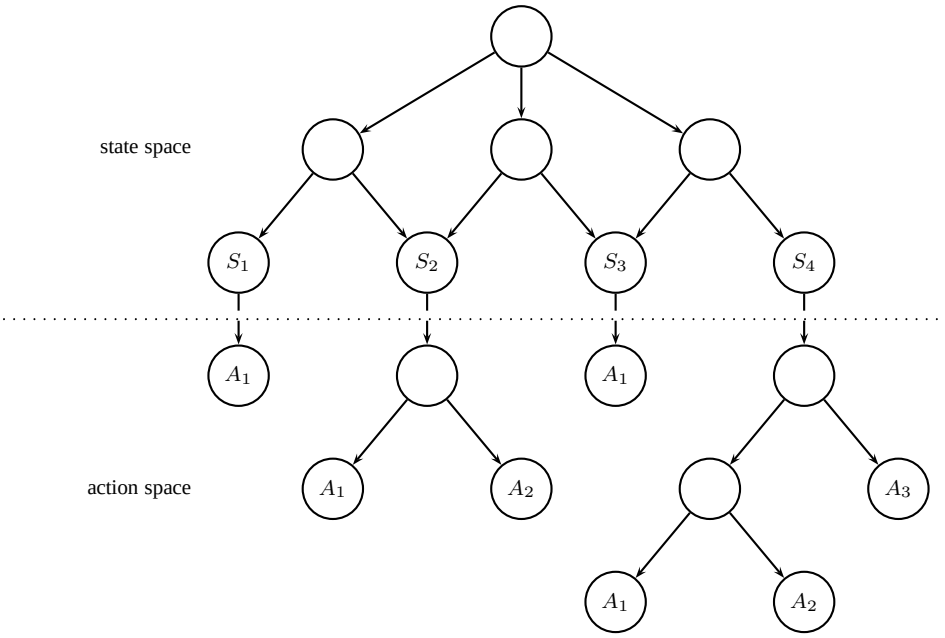



Figure 3: A decision graph partitioning the state and action spaces.

This section uses $\hat{X}$ to represent the union of the sets $X_1, \ldots, X_n$. The procedure SPLIT$(X, X_1, \ldots, X_n)$ results in the redistribution of the objects in $\hat{X}$ into the sets $\hat{X}_1, \ldots, \hat{X}_m$. The remainder of this section discusses distinction quality measures and efficient ways to handle attribute-value representations, vector representations, and more general representations.

### 4.3.1 Distinction Quality Measures

Splitting $X_1, \ldots, X_n$ involves choosing a test that maximises some measure of quality. There are many different ways to measure the quality of a split. AMPS, like other tree induction systems, defines the quality of a split in terms of the decrease in impurity, where impurity is a measure of the mixture of objects belonging to different categories. One such impurity measure borrowed from information theory (Shannon, 1948) is *entropy impurity*, which is given by

$$i(X') = -\sum_{i=1}^{n} P(X_i \mid X') \lg P(X_i \mid X'),$$

where $X'$ is some subset of $X$, the union of the sets $X_1, \ldots, X_n$. For any two sets $X'$ and $X''$,

$$P(X' \mid X'') \equiv |X' \cap X''| \,/\, |X''| \,.$$

The decrease in impurity is the weighted impurity after the split subtracted from the impurity before the split,

$$i(X) - \sum_{j=1}^{m} P(\hat{X}_j \mid X) i(\hat{X}_j).$$

This quantity is also called the *information gain* (Shannon, 1948). If the information gain of a split is small, then it is not advantageous to introduce the distinction. An alternative to the information gain as a quality

measure is the *gain ratio*, which is the information gain divided by

$$-\sum_{j=1}^{m} P(\hat{X}_j \mid X) \lg P(\hat{X}_j \mid X).$$

Quinlan (1986) suggests using the gain ratio so that the quality measure does not inherently favour splits with many outcomes. The literature contains many other impurity and quality measures (see Murthy, Kasif, & Salzberg, 1994). AMPS uses information gain by default.

#### 4.3.2 Attribute-Value Representation

Attribute-value representations associate values to attributes. McCallum (1995) assumes an attribute-value representation for his UTree algorithm. A single state is simply an assignment of values to all of the attributes. Assuming some ordering of the attributes, the state may be equivalently represented as a tuple of values. The domain of values for a particular component is finite. Calling $\textsc{Split}(X, X_1, \ldots, X_n)$ on sets of objects associated with the region $X$ involves choosing an attribute test that maximises the distinction quality. If the attribute has $m$ different possible values, then the split partitions $X$ into $X'_1, \ldots, X'_m$.

#### 4.3.3 Vector Representation

Attribute-value representations are often not appropriate for real-world applications. Vector representations are more general and allow for greater generalisation. Whereas there is no implied relationship between the nominal values assigned to an attribute in an attribute-value representation, a vector representation with an associated inner product defines relationships between objects including angles and distances. Supervised learning methods typically involve generalisation over objects embedded in a vector space.

Efficient algorithms exist for determining the basis-orthogonal hyperplane that maximises information gain (Fayyad & Irani, 1992). Restricting the decision boundaries to basis-orthogonal hyperplanes limits how well they can separate the objects. Although allowing oblique hyperplanes can increase the information gain, the algorithms for choosing arbitrary hyperplanes have greater complexity.

In general, finding the best oblique hyperplane according to some metric is not practical. As proven by Heath (1992, Appendix C), even minimising the sum-minority metric is NP-complete. Hence, a number of algorithms employ heuristic search methods. CART (Breiman, Friedman, Olsen, & Stone, 1984, Section 5.2) uses a deterministic heuristic search procedure for the vector $\theta$ and the scalar $c$ that approximately maximises the information gain. Heath, Kasif, and Salzberg (1993) use simulated annealing to find the hyperplane that approximately maximises the quality according to the sum-minority metric. The oblique decision tree induction system OC1 (Murthy et al., 1994) uses local heuristic search to find a locally optimal split and then perturbs the hyperplane.

The quality measures in Section 4.3.1 do not utilise the structure inherent in the vector space. It can be advantageous to use the inner product to define the measure of quality. For example, one might wish to minimise the sum of the distances of misclassified objects to the separating hyperplane. Interestingly, there are efficient methods that optimise over this measure of quality, even though optimising the sum-minority measure is NP-complete. Duda, Hart, and Stork (2000, Chapter 5) survey a variety of methods for choosing decision boundaries that maximise criteria involving inner products.

Many classification algorithms, such as *support vector machines* (see Cristianini & Shawe-Taylor, 2000), rely solely on the inner product, and their performance can be greatly enhanced with an appropriate choice of kernel function. The literature contains a number of kernel functions for a variety of representations including graphs, sets, unstructured text, and structured data (Schölkopf & Smola, 2000; Shawe-Taylor & Cristianini, 2004).

#### 4.3.4 Generating Tests

For some domains, especially relational domains, it is not natural or desirable to transform the object space into a vector space and define an inner product. AMPS provides an alternative way to generate tests for a decision graph. The system synthesises tests from a specification of typed predicates and functions that

perform operations on the underlying representation of the object space. AMPS assembles the test that maximises the information gain.

The current implementation of AMPS accepts an XML specification of a hierarchical type system. The first part of the file specifies the names of the types and "is-a" relationships between the types. The second part of the file specifies the functions. Each function has a name, a return type, a set of typed parameters, and a location of the function. The third part of the file specifies the predicates, including their name, typed parameters, and location.

Presently, the string in the XML file denoting the location of a function or predicate is a fully-qualified class name of the Java implementation. However, it is possible to extend the implementation of AMPS to support functions implemented in other languages in remote locations or calls to XML Web Services. The AMPS package includes a variety of general-purpose types, functions, and predicates, but in principle AMPS can utilise a distributed library of types, functions, and predicates.

The types do not simply denote how the underlying data is represented; the types convey semantics. Although two types might share the same representation encoded as an XML Schema, they may have different semantics. A sequence of integers, for example, might represent an unordered set of integers, an ordered set of integers, or a binary tree with integers associated with the nodes. When introducing distinctions between objects, it is important to be faithful to the semantics. In order to differentiate between types with different semantics, one might assign a Unique Resource Identifier (URI) to the type.

AMPS constructs all of the well-formed grounded predicates within some depth limit in memory. A maximum depth must be imposed because it is possible for functions to be infinitely nested. For example, the function $f$ might take a parameter of type $\tau$ and return a value of type $\tau$, allowing itself to be infinitely nested. When AMPS introduces a test into a decision tree, it simply points to the grounded-predicate already in memory.

Although a properly designed type system can greatly restrict the number of possible grounded predicates, it is likely that AMPS produces tautologous, inconsistent, or redundant grounded predicates with respect to a set of axioms $\Psi$. Such predicates should be removed from consideration. A grounded predicate $\psi$ is tautologous when $\Psi \models \psi$ and is inconsistent when $\Psi \models \neg\psi$. Such grounded predicates are not useful in distinguishing objects. If $\psi$ and $\psi'$ are grounded predicates, and $\Psi \models (\psi \leftrightarrow \psi') \vee (\psi \leftrightarrow \neg\psi')$ then $\psi$ is redundant with $\psi'$ and should be removed from consideration.

AMPS uses the JTP automated reasoning system (Fikes, Jenkins, & Frank, 2003) to prove whether a grounded predicate can be pruned from consideration. In order to prove whether a grounded predicate is useful, one must specify a set of axioms over the functions and predicates. For example, in the Taxi World domain, some useful axioms include $X = X$, $\neg\, north(X, X)$, and $east(X, Y) \leftrightarrow west(Y, X)$, with $X$ and $Y$ as universally quantified variables. The axioms are specified in the standardised Knowledge Interchange Format (Genesereth & Fikes, 1992).

Pruning unnecessary grounded predicates with a theorem prover is not trivial, requiring minutes to compute. Fortunately, this pruning only needs to occur once, before the agent commences interaction with the world. AMPS also includes the functionality to serialise to a file the set of pruned ground predicates in memory. The compiled file can later be restored to memory quickly, without having to use the theorem prover.

When the agent needs to decide upon a grounded predicate that best splits $X_1, \ldots, X_n$, it iterates through the possible grounded predicates and chooses the one that maximises the information gain or some alternative quality measure. Assuming that evaluating grounded predicates requires constant time, the computational complexity of choosing the best grounded predicate is $O(mn + \ell n)$, where $m$ is the number of objects in $X_1, \ldots, X_n$ and $\ell$ is the number of grounded predicates.

## 4.4 Nearest Neighbour Approach

The previous section explains various ways to add distinctions and achieve generalisation with decision graphs. This section discusses a completely different approach supported by AMPS that involves nearest-neighbour generalisation (Cover & Hart, 1967). Nearest neighbour is a kind of instance-based learning algorithm (Aha, Kibler, & Albert, 1991) where labelled instances are generalised to unlabelled objects by means of a distance metric. In AMPS the labels correspond to regions of the state or action space. When

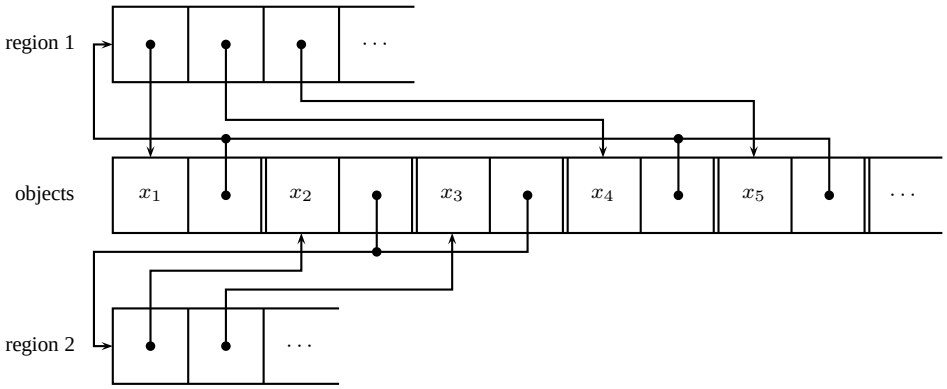



Figure 4: The nearest neighbour data structure. A pointer associates each object with its region. A region consists of a list of references to objects in the region. In the illustration, $x_1$, $x_4$, and $x_5$ belong to region 1, and $x_2$ and $x_3$ belong to region 2. This diagram is a simplification and does not show the additional information associated with objects and regions.

presented with a novel object, the nearest neighbour rule predicts its label to be the same as its closest neighbour. The objects may have any representation so long as one can define a suitable distance metric.

A *metric* is a nonnegative, symmetric function $d(x, x')$ that denotes the distance between the object $x$ and object $x'$. Metrics satisfy $d(x, x) = 0$ and the *triangle inequality*

$$d(x, x') + d(x', x'') \geq d(x, x'').$$

If $d(x, x') = 0$ does not necessarily imply $x = x'$, then $d(x, x')$ is called a *pseudometric*. The discussion in this section about metrics also applies to pseudometrics. A set of objects endowed with a metric is called a *metric space*.

As the agent interacts with the world, it records the information from its trajectory, including the states observed and the actions taken. When the agent records a state or action, it associates with the object a label designating its region. The object receives the label of its closest neighbour according to some specified metric. This label is actually a reference to a linked list in memory consisting of all the objects previously experienced in the same region (Figure 4). After assigning the label to the object, AMPS updates the list by adding the new object. When the agent records its first object, the object will not have neighbours, so the agent associates with the object a new linked list consisting only of the object.

The procedure SPLIT$(X, X_1, \ldots, X_n)$ revises the regions so that the objects in the sets $X_1, \ldots, X_n$ get mapped to different regions and the remaining objects in $X$ get mapped to the region of its closest neighbour (Algorithm 3). The process begins by iterating through each set $X_i$, first creating a new region and then moving all the objects in $X_i$ from the old region $X$ to the new region. After the iteration is complete, the process adds each remaining object in region $X$ to the region of its closest neighbour in the sets $X_1, \ldots, X_n$.

**Algorithm 3** SPLIT$(X, X_1, \ldots, X_n)$

**for** $i = 1$ to $n$ **do**
  create a new region $X'$
  move the objects in $X_i$ from $X$ to $X'$
**for all** remaining objects $x$ in $X$ **do**
  add $x$ to the region of its closest neighbour in the sets $X_1, \ldots, X_n$

Planning (Section 2.4) occurs over regions of the state space, not individual states. The learned policy maps state regions to action regions. When the agent encounters a new state, the agent must compute its region. The decision-graph approach computes the region through a series of tests, but the nearest-neighbour approach computes the region by searching for the closest labelled instance. Because the agent

must compute the nearest neighbour at least as frequently as it samples the environment, it is important that the nearest neighbour calculations are efficient.

A naive implementation of nearest neighbour involves a full search through the set of instances. More efficient implementations utilise some form of indexing data structure. Many different applications require computing the nearest neighbours of a query point, and so the literature is full of indexing methods for efficient nearest neighbour calculation. Chávez, Navarro, Baeza-Yates, and Marroquín (2001) and Hjaltason and Samet (2003) survey several approaches and compare their efficiency. It should be emphasised that these indexing systems work with any metric space, regardless of distance measure or object set.

Ramon (2002, Chapter 3) discusses a wide range of distance metrics in depth, including metrics over sets, trees, strings, and first-order logic objects. Wilson and Martinez (1997) describe other distance metrics. Although one might be able to define a distance metric for a space, there is no guarantee that the metric will provide the desired generalisation.

## 4.5 Value Clipping

This section introduces value clipping as used in AMPS. The necessity for value clipping arises due to the interleaving of incremental modelling and planning. Value clipping results in improved estimates of the value function by clipping the value function at state regions where the estimated value is easily recognised to be too high or too low. This process is especially useful in identifying situations where the agent should take exploratory moves, as this section explains.

The value clipping algorithm in this section assumes that the agent receives zero reward except when terminating, i.e. when it transitions to the synthetic terminal region $S_0$ (see Section 3.3). The value of a state region $S$ can be no greater than the maximum value of $r(S', A, S_0)$ over all action regions $A$ from all state regions $S'$ reachable from $S$. Similarly, the value of a state region $S$ can be no less than the minimum value of $r(S', A, S_0)$ over all action regions $A$ from all state regions $S'$ reachable from $S$. In other words,

$$V(S) \leq \max_{S' \in \text{REACH}(S), A \in \mathcal{A}(S')} r(S', A, S_0) \tag{3}$$

$$V(S) \geq \min_{S' \in \text{REACH}(S), A \in \mathcal{A}(S')} r(S', A, S_0), \tag{4}$$

where $\text{REACH}(S)$ is the set of all state regions reachable from $S$ in the model.

The value clipping algorithm identifies when either Equation 3 or Equation 4 is violated by the estimated value function and clips the values appropriately. The value clipping algorithm is composed of two separate algorithms. One algorithm clips from above and the other algorithm clips from below. Since the algorithm that clips from below is almost identical to the one that clips from above, this section only discusses clipping from above. The algorithm for clipping from above (Algorithm 4) involves calling the recursive helper procedure VALUE-CLIP-MAX-HELPER (Algorithm 5) until all of the state regions have been marked as being clipped. In Algorithm 5, $\text{pred}(S)$ is the set of all predecessors of $S$ in the model.

**Algorithm 4** VALUE-CLIP-MAX

```
create a list L consisting of the elements in S
sort L according to max_{A∈A(S)} r(S, A, S_0)
for all regions S ∈ L do
  if S is not marked then
    VALUE-CLIP-MAX-HELPER(S, max_{A∈A(S)} r(S, A, S_0))
```

The value clipping algorithm is particularly useful when the value at a region gets clipped to zero, indicating that the model does not contain information on a satisfactory way to proceed to a goal. In such a situation, it is beneficial for the agent to explore. In this discussion, goal regions refer to regions where there is a non-zero probability of transitioning to $S_0$ with positive expected reward for some action. Nonterminal regions are regions where the probability of transitioning to $S_0$ is zero. A region $S$ is alienated from another region $S'$ if there is zero probability of ever reaching $S'$ from $S$, regardless of the policy the agent follows.

When integrating incremental planning and modelling, it is common for an added distinction in the state space to alienate a nonterminal region from all goal regions. Assuming that reward is always nonnegative,

**Algorithm 5** VALUE-CLIP-MAX-HELPER$(S, v)$

```
mark S
if V(S) > v then
  V(S) ← v
for all S' ∈ pred(S) do
  if S' is not marked then
    VALUE-CLIP-MAX-HELPER(S', v)
```

the alienated region should have zero value. However, if the planning process assigned positive value to the region before it was alienated, it can take many iterations of prioritised value iteration for the value to go to zero. Kochenderfer (2006) provides a concrete example of a situation where value clipping is useful.

# 5 Simplification

AMPS is different from other abstraction approaches (Section 7) in that it attempts to both simplify and refine its model. The other standard abstraction approaches only introduce perceptual distinctions when experience indicates that the current model is not sufficient to explain the observations. Consequently, the models monotonically increase in complexity as the agent acquires experience. However, data collected later by the agent might indicate that the model is unnecessarily complex and should be simplified.

Complexification of the system model in AMPS involves splitting regions of the state and action space. Simplification involves merging regions. AMPS is more aggressive than the other abstraction algorithms, such as the G Algorithm (Chapman & Kaelbling, 1991) and UTree (McCallum, 1995), in that it does not wait for sufficient data to prove statistical significance before performing a split. AMPS may introduce many more irrelevant distinctions in the state and action spaces than the other methods, but it does so with the idea that the model will be simplified later if further data indicates that it is useful to do so.

Simplification involves either merging regions of the state space or merging regions of the action space. The first part of this discussion assumes that the action space is small and does not require generalisation and explains generalisation in the state space. The second part generalises the discussion to problems with large action spaces that require generalisation.

Suppose for the moment that the model is deterministic and that the agent possesses some optimal policy $\pi$. Assume also that the action space is small and does not require generalisation. It is desirable to simplify the model while maintaining an "adequate representation" for $\pi$. There are two situations where the agent may merge the regions $S'$ and $S''$:

- **Chain merge:** If $\mathrm{succ}(S', \pi(S')) = S''$, $\mathrm{succ}(S'', \pi(S'')) = S$, and $\pi(S') = \pi(S'')$ then merge $S'$ with $S''$.
- **Sibling merge:** If $\mathrm{succ}(S', \pi(S')) = S$, $\mathrm{succ}(S'', \pi(S'')) = S$, and $\pi(S') = \pi(S'')$ then merge $S'$ with $S''$.

For a deterministic model, $\mathrm{succ}(S, a)$ denotes the region to which the agent transitions after taking action $a$ from region $S$. These two kinds of merging, *chain merging* and *sibling merging*, are similar to the Squish algorithm[3] for simplifying goal-directed, reactive plans encoded as teleo-reactive trees (Nilsson, 1992). AMPS includes the first generalisation of Squish in a reinforcement learning context. AMPS extends the basic idea of Squish to nondeterministic transitions and partitioned action spaces. Figure 5 illustrates chain merging and sibling merging.

Chain and sibling merging may be generalised to nondeterministic environments. In nondeterministic environments, $\mathrm{succ}(S, A)$ is not necessarily a single element, but the function may be adapted such that

$$\mathrm{succ}(S, A) \equiv \begin{cases} S' & \text{if } P(S' \mid S, A) \approx 1 \\ \text{NIL} & \text{otherwise} \end{cases},$$

[3]The Squish algorithm was developed by George H. John at Stanford University in 1994 but has not been published. The most detailed description of the algorithm is available in a paper by Nilsson (2000). The Squish algorithm was designed as a behavioural cloning technique, but the basic ideas are transferable to model simplification.

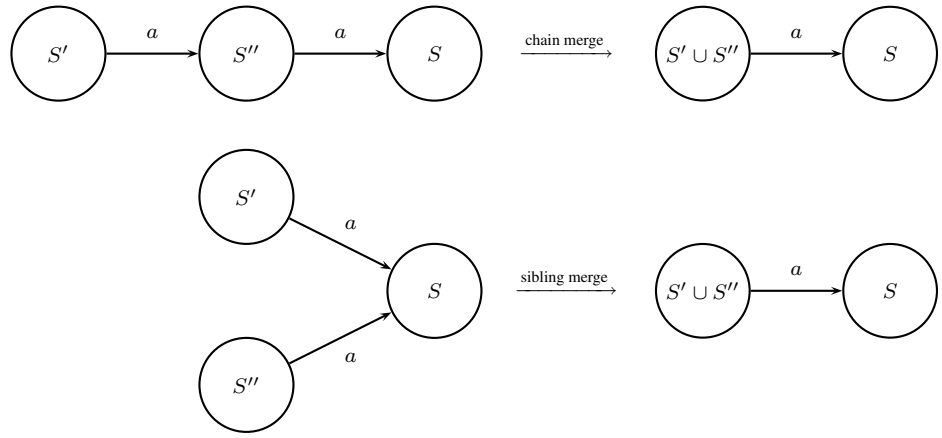



Figure 5: A demonstration of merging rules for state regions. The diagram shows the transformations of chain merging and sibling merging. The greedy actions are indicated along the transitions.

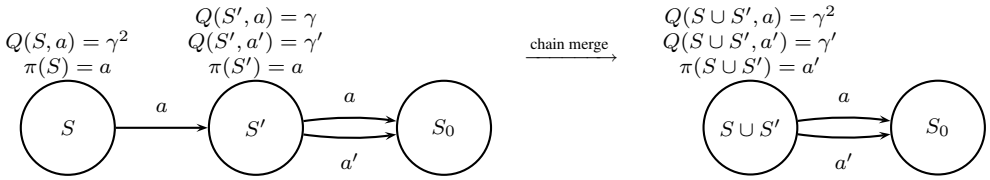



Figure 6: An example of how merging does not necessarily preserve the policy. The original model is estimated from two trajectories. The first trajectory involves $S \xrightarrow{a} S' \xrightarrow{a} S_0$, and the second trajectory involves $S' \xrightarrow{a'} S_0$. The policy changes after the merge if $\gamma^2 < \gamma' < \gamma$.

where $\approx 1$ means within some small parameter $\epsilon$.

Chain and sibling merging will not necessarily result in an equivalent plan. Since merging two regions changes the model, further planning on the new model can result in a different plan. Figure 6 provides a simple example of how the plan might change with chain merging. A similar example for sibling merging is not difficult to construct. Although merging can disrupt the policy, the disruption is not likely not hinder the performance of the agent in the long term so long as the agent is permitted to continue to revise its model.

In order to apply chain and sibling merging to problems with large action spaces, it is necessary to generalise the rules to:

- **Chain merge:** If $\text{succ}(S', \pi(S')) = S''$, $\text{succ}(S'', \pi(S'')) = S$, and $\pi(S') \sim \pi(S'')$, then merge $S'$ with $S''$.
- **Sibling merge:** If $\text{succ}(S', \pi(S')) = S$, $\text{succ}(S'', \pi(S'')) = S$, and $\pi(S') \sim \pi(S'')$, then merge $S'$ with $S''$.

In the earlier definition for small action spaces, the relation $A \sim A'$ denotes equality. However, it does not make sense to test for equality between two action regions associated with different state regions because different state regions partition the action space differently. The only way to compare $A$ and $A'$ is by consulting the experienced actions associated with these two action regions. AMPS tests to see if at least one of the examples associated with $A$ can be mapped to $A'$ and that one of the examples associated with $A'$ can be mapped to $A$. Testing in this manner makes the relation $A \sim A'$ reflexive and symmetric and indicates that there is some overlap between the two regions.

When merging two state regions $S$ and $S'$, AMPS also merges the action regions associated with those two state regions. Although this means that the action regions must be repartitioned, the splitting

process (Section 4) can quickly perform the necessary revision. An alternative is to simply use the partition associated with either $S$ or $S'$.

AMPS supports the merging of action regions. Since it is not beneficial to make actional distinctions between suboptimal actions, AMPS merges non-greedy action regions. Although merging suboptimal action regions does no harm if the agent is certain of its policy, in the presence of uncertainty such a strategy can lead to longer convergence to an optimal policy. If the agent mistakes a suboptimal action region as an optimal region, AMPS might merge the actual optimal region with other suboptimal regions. Discovering such a mistake may require substantial exploration.

Model simplification is important for two reasons. First, a simpler model reduces the amount of computation required for planning. Second, a simpler model is easier to estimate from limited experience. Surprisingly, the other model-based reinforcement learning approaches in the literature (Section 7) do not incorporate model simplification; they only refine the discretisation of the state space. AMPS is the first model-based reinforcement learning algorithm that constructs its model through a dynamic process of splitting and merging existing regions of the state and action spaces.

It should be noted that model simplification in AMPS differs from the *aggregation* and *model minimisation* algorithms in the stochastic control literature that attempt to solve known MDPs by grouping together base states (e.g., Schweitzer, Puterman, & Kindle, 1985; Bertsekas & Castañon, 1989; Givan, Dean, & Greig, 2003). These methods require a full specification of the MDP at the base level, and are therefore unsuitable for model simplification in AMPS.

# 6 Integration

Section 2.4 explained how AMPS performs incremental plan revision, and Sections 4 and 5 explained how to perform incremental model revision. This section discusses the integration of modelling and planning in AMPS and its implementation as part of the Java Reinforcement Learning Framework (JRLF). The complete source code of JRLF is publicly available from

`http://mykel.kochenderfer.com/jrlf`

and may be distributed according to the GNU General Public License. Although there are many publicly available reinforcement learning packages, most assume discrete time steps and are implemented in C or C++. In addition, most existing packages make assumptions about the representation of the states and actions.

The remainder of this section focuses on AMPS and its implementation as part of JRLF. AMPS is composed of four processes that operate on the map, experience, model, and plan data structures. The four processes include map revision, experience revision, action selection, and plan revision. These processes are independent of each other and can be interleaved in any order.

## 6.1 Map Revision

The map revision process is responsible for requesting the map data structure to split and merge regions of the state and action spaces. Associated with the map revision process is a collection of revisors, such as the value revisor (Section 4.1.1), failure revisor (Section 4.1.1), and simplification revisor (Section 5). Each revisor is capable of computing the priority of its revision at a particular state region as well as performing the revision.

The map revision process associates with each state region the highest priority revision and maintains a priority queue consisting of the regions whose priority is above a certain threshold. The revision process listens for changes in the data structures that impact the priorities it assigns to regions and updates the priorities accordingly. When the agent has a small slice of time available, it can activate the map revision process. The revision process quickly performs the highest priority revision at the highest priority state and returns control back to the agent.

## 6.2 Experience Revision

The experience revision process is responsible for notifying the experience data structure when the agent acquires new experience. If the agent samples the state at a high frequency, it is not practical to record every experience. The experience revision process can function as a filter, only recording experiences that it deems significant. The exact mechanism for determining significance is domain dependent, but one approach to filtering out states is to ignore all states until the agent has transitioned to a new state that is outwith some threshold distance.

In addition to filtering out insignificant experiences, the experience revision process is also responsible for removing old experiences when memory and processor constraints deem it necessary. The removal of old samples allows the agent to adapt to slowly changing environments. Sometimes the removal of old experience may leave one or more of the state regions without samples. AMPS may remove any state regions without samples and revise the map data structure appropriately.

## 6.3 Action Selection

The action-selection process is responsible for continuously selecting a single action to execute. Action selection needs to be extremely efficient because it is typically executed as frequently as the agent samples the state of the world. A good action-selection strategy adheres to the following four principles:

- **The strategy should encourage taking an action consistent with the estimated optimal policy.** The purpose of planning is to identify the actions that are likely to maximise the expected return. The action-selection strategy should exploit the beliefs it obtains through planning by executing the actions likely to be valuable.
- **The strategy should encourage occasional random exploration.** Although the agent should generally take the action recommended by the planning process, random exploration is also necessary in order for the agent to estimate an accurate model of the system dynamics.
- **The strategy should encourage taking the action taken in the previous step.** The continued application of an action is often necessary to transition from one abstract state region to another. If actions are frequently interrupted before completion, then the agent is not able to effectively estimate the parameters of the state-transition model.
- **The strategy should encourage executing actions that have not led previously to failure.** Failure indicates that the continued application of the same action is not likely to result in a useful transition. The action-selection strategy should avoid recommending actions that have led to failure.

The challenge is to balance these four competing principles. Most reinforcement learning algorithms attempt to balance the first two principles involving exploitation and exploration, however most algorithms in the literature are not designed to address the last two principles.

**Algorithm 6** CHOOSE-ACTION$(s)$

```
a ← last action taken
S ← S(s)
A ← A(a)
if RANDOM(1 − ε − (1 − 2ε)P_f(S, π(S))) then
    if A = π(S) then
        return a;
    return a random action in π(S)
if A ≠ NIL and RANDOM(1 − ε − (1 − 2ε)P_f(S, A)) then
    return a
return a random action
```

Algorithm 6 lists the basic action-selection strategy in AMPS. It follows the four principles of a good action-selection algorithm and is relatively simple. With probability inversely related to the estimated

| Algorithm | Representation | Planning | Splitting Criterion |
|---|---|---|---|
| G algorithm | binary strings | TD | $t$-test |
| Parti-game | Euclidean | minimax | losing |
| UTree | attribute-value | DP | Kolmogorov-Smirnov |
| Continuous UTree | Euclidean | DP | sum-squared error |
| TTree | attribute-value | DP | MDL |
| Variable Resolution | Euclidean | DP | influence-variance |
| TG algorithm | relational | TD | $F$-test |

Table 1: A summary of adaptive abstraction algorithms.

probability of failure, the agent will execute the greedy action. Otherwise, it will continue with the action it was previously executing, again with probability inversely related to the estimated probability of failure. Otherwise, it will execute a random action. Random exploration is controlled by means of the parameter $\epsilon$.

In Algorithm 6, the function $\textsc{Random}(p)$ is true with probability $p$ and is false otherwise. In the algorithm, the probability of taking a greedy action is given by

$$1 - \epsilon - (1 - 2\epsilon)P_f(S, \pi(S)).$$

This selection probability has two desirable properties. First, it decreases monotonically with respect to the estimated probability of failure and regardless of the failure rate. Second, regardless of failure rate, it is never deterministic, thereby allowing exploration. The probability of continuing the previous action is of a similar form.

The action-selection strategy in Algorithm 6 has the property that it will continue executing a greedy action within a state region for a duration selected from a geometric distribution, assuming a uniform sampling rate. As the sampling rate goes to infinity, the distribution over durations approaches an exponential distribution.

If value clipping (Section 4.5) is incorporated into the system, the action-selection algorithm should be adapted. If the value of a state $S$ is zero, assuming that reward is always positive, then the best action to take from $S$ is unclear and $\pi(S)$ in Algorithm 6 is not useful. In this case, the first *if* block in Algorithm 6 is ignored.

### 6.4 Plan Revision

The plan revision process is responsible for updating estimates of the value function and optimal policy. AMPS uses prioritised value iteration to choose which state regions to update. As Section 2.4 explains, there are different ways of implementing prioritised value iteration. By default, AMPS uses an implementation that follows that of Moore and Atkeson (1993), but other approaches are suitable for plan revision as well.

The plan revision process listens for changes in the model, including the splitting and merging of regions, and for changes in the estimated value function as a result of value clipping. Depending on the nature of the changes, the plan revision process updates the appropriate revision priorities. When the agent has time to perform planning, the planning process updates the value function and plan at the region at the front of the priority queue.

## 7 Related Work

This section surveys a selection of the most significant abstraction methods in chronological order of development. Abstraction involves partitioning the state and action spaces into regions and using a model-based or model-free approach to compute an optimal policy. None of the algorithms perform abstraction in the action space, although some of the authors speculate about ways of achieving generalisation over actions.[4]

The first algorithm in this section uses a human-engineered abstraction that does not change as the agent accumulates experience. The remainder of the algorithms in this section adapt their abstractions. Table 1

[4]See the discussions by Chapman and Kaelbling (1991, Section 2) and McCallum (1995, Section 8.2.3).

summarises some of the important properties of the adaptive abstraction algorithms. As the table reveals, these algorithms vary in their representation, planning algorithm, and splitting criterion. All of the algorithms use some form of tree structure to partition the state space. The tree structure grows incrementally when experience indicates that it should increase resolution at a particular region of the state space.

## 7.1 Boxes

Some problems can be solved effectively using a human-supplied discretisation of the state or action space. Michie and Chambers (1968a) discuss some early work on static state abstraction for the pole-and-cart task. They manually discretise a four-dimensional continuous state space into 255 regions, which they refer to as "boxes." Since their problem involves only two actions, there is no need to perform abstraction in the action space. They employ a Monte Carlo reinforcement learning algorithm to produce successful behaviour.

One of the main limitations of their approach is that the abstraction remains static. If the abstraction is too coarse, then the agent will not be able to find an optimal solution. If the abstraction is too fine, then the agent will take a long time to compute an optimal solution. In another paper on their Boxes algorithm, Michie and Chambers (1968b) speculate about the automatic "splitting and lumping" of regions:

> The weakness of our program in its present form is that it does not really live up to its ideal of independence of special knowledge of the physical apparatus. Some knowledge of this kind is in fact built into it when the user specifies the thresholds to be set on the state variables. It is easy to choose these in such a way as to make the control task impossible. In our plans for the next stage we aim to endow the program with the power to change the boundaries of boxes, by processes of 'splitting' and 'lumping.' (page 214)

They propose splitting regions where the best action has an expected value that is close to the expected value of other actions, and they propose merging neighbouring regions that agree which action is best. They did not implement this approach.

## 7.2 G Algorithm

Although variable resolution techniques for automatic control have been explored by others (e.g., Simons, Brussel, de Schutter, & Verhaert, 1982), the G algorithm by Chapman and Kaelbling (1991) is one of the most important early implementations, inspiring the other approaches in this section. Although there are a number of weaknesses in their approach, the idea of incrementally inducing a decision tree based on experience is useful when dealing with large state spaces.

The G algorithm assumes a fixed-length binary representation of the state space and incrementally builds a decision tree. The leaf nodes of the tree represent regions of the state space. Associated with the leaf nodes are state-action values obtained through a variation of Q-learning. The main contribution of the G algorithm is the use of Student's $t$-test to measure the statistical significance of individual bits in the state representation. If the $t$-test indicates that a bit is significant in predicting either immediate reward or discounted return, the region is split on that bit. Unfortunately, when a region is split, all the information associated with that region is lost, which makes for very slow learning.

## 7.3 Parti-Game

Moore and Atkeson (1995) introduce the Parti-Game algorithm as a way to produce goal-directed behaviour in continuous states spaces. Like the G Algorithm, Parti-Game splits regions where it deems important, but the approach and assumptions are significantly different. Parti-Game assumes that the state space is represented as a vector of real values and that the agent possesses a greedy controller that can move it towards any desired state. There is no guarantee that the greedy controller will succeed, but the agent is signalled when the greedy controller becomes "stuck" against some obstacle. Parti-Game assumes that the dynamics of the world are deterministic and that the goal state is known.

The objective is to produce behaviour that takes the agent to the goal without becoming stuck, which is in contrast to the objective of traditional reinforcement learning where the agent must maximise its expected discounted return. Parti-Game uses a game-theoretic approach to decide which neighbouring region to

aim towards with the greedy controller. It chooses the neighbouring region that minimises its estimated maximum possible cost to the goal, where the cost is the number of regions it must transition through to reach the goal. The agent maintains a database of previous experience to estimate this cost. Losing regions are regions whose minimax cost to the goal is infinite, indicating that the best policy from that region is expected to become stuck. The algorithm splits losing regions that have a non-losing neighbour and non-losing regions with losing neighbours.

Moore and Atkeson show that Parti-Game can learn competent behaviour in a variety of continuous domains with up to nine dimensions. In all of the domains they study, fewer than ten episodes are needed to learn satisfactory solutions. Unfortunately, the approach is currently limited to deterministic real-valued domains where the agent has a greedy controller. Al-Ansari and Williams (1999) and Likhachev and Koenig (2003) suggest a few improvements to the algorithm, but they do not overcome these basic limitations.

## 7.4 UTree

McCallum (1995) introduces the UTree algorithm, which extends the work by Chapman and Kaelbling (1991). The algorithm assumes an attribute-value representation of the state, where the values of the attributes are nominal. Like the G algorithm, UTree does not generalise over the action space. UTree not only makes distinctions over the current state, but it can also make distinctions based on previous observations, allowing it to handle partially observable domains.

The UTree algorithm records the experience of the agent and associates the observation, action, and reward with the appropriate leaf node in the tree. The algorithm uses these experiences to estimate the parameters of an MDP, which can be solved with dynamic programming. Periodically, the algorithm checks whether it needs to add additional distinctions to the leaves of the tree. The algorithm first computes the value $Q(o_k)$ of a particular observation $o_k$ stored at a leaf node according to

$$Q(o_k) = R_k + e^{-\beta t_k} V(o_{k+1}), \tag{5}$$

where $V(o_{k+1})$ is the value of the leaf to which $o_{k+1}$ belongs, $R_k$ is the discounted reward between observations, $t_k$ is the time between observations, and $\beta$ is the continuous compound discount rate.[5] The algorithm uses the Kolmogorov-Smirnov statistical test to determine whether adding a distinction will result in significantly different distributions of observation values.

## 7.5 Continuous UTree

Uther (2002) extends the UTree algorithm to use continuous attributes, allowing state spaces that are subsets of Euclidean space. His algorithm, however, assumes that the agent senses the underlying states of an MDP as opposed to observations emitted by a POMDP. The approach otherwise follows UTree quite closely.

To handle continuous attributes, continuous UTree considers splits between each consecutive pair of values along each dimension. Uther considers the sum-squared error splitting criterion in addition to the Kolmogorov-Smirnov test. The sum-squared error criterion involves computing the variance of the sampled state values on either side of a split. If the split significantly reduces the weighted variance, then the algorithm introduces the split into the tree.

## 7.6 TTree

Uther (2002) presents TTree as an extension to his continuous UTree algorithm. TTree learns a smaller SMDP from a generative model of a larger SMDP. A generative model of an SMDP is a randomised mapping that simulates transitions from a given state by a given action. Unlike continuous UTree and the other algorithms in this section, TTree is *not* intended to be used as an algorithm for generating intelligent behaviour from actual experience. TTree extends UTree by allowing actions with multi-step duration.

TTree samples trajectories using the generative model starting from random states within the leaf nodes. The algorithm stops sampling a trajectory when it either reaches another leaf, detects a deterministic self-transition, or exceeds some timeout. As the algorithm samples the trajectory, it tracks its accumulated

[5] Equation 5 is a straightforward generalisation of what is contained in the thesis by McCallum (1995) to SMDPs.

discounted reward. The expected discounted return of a sampled trajectory with accumulated discounted reward $R$ is given by $R + e^{-\beta t}V$, where $t$ is the duration of the sampled transition and $V$ is the value of the leaf in which the trajectory terminates. The values of the leaves are calculated from experience using dynamic programming. The algorithm samples multiple trajectories starting from the same state to produce an estimate of the expected discounted return and optimal action from that state. TTree splits a leaf node if it observes variation within a leaf of the expected discounted return for the same action. It uses minimum description length tests to decide how and when to grow the tree.

### 7.7 Variable Resolution Discretisation

Munos and Moore (2002) explore variable resolution techniques for solving reinforcement learning problems in a continuous-event dynamic system. Their techniques assume that the state space is a compact subset of Euclidean space and that the action set is finite. The world behaves deterministically, as in Parti-Game, according to a differential equation. Their algorithm begins with a coarse, grid-based discretisation of the state space. The algorithm estimates an MDP from these grid points and solves the MDP using standard dynamic programming techniques.

In contrast with the other abstract algorithms in this section, the value function and policy vary linearly within each region. Munos and Moore use Kuhn triangulation as an efficient way to interpolate the value function within regions. The algorithm refines its approximation by splitting cells according to a splitting criterion. Munos and Moore explore several local heuristic measures of the importance of splitting a cell including the average of corner-value differences, the variance of corner-value differences, and policy disagreement. They also explore global heuristic measures involving the influence and variance of the approximated system. The influence is a measure of non-local dependencies in the value function, and variance is an estimate of the error in the value function due to the grid approximation.

### 7.8 TG Algorithm

Driessens (2004) expands upon previous work in *relational reinforcement learning* (Džeroski, de Raedt, & Driessens, 2001). He presents the TG algorithm as a way of inducing logical decision trees for problems whose state spaces are described by relations between objects. A logical decision tree is a decision tree where the tests may have unbound variables. Logical decision trees attempt to capture the structure inherent in the problem.

The TG algorithm is essentially the G algorithm for logical decision trees. As the agent accumulates experience, it updates the statistics at the leaf nodes for all possible new tests. The tests are logical sentences that may include variables. The sentences may introduce new variables or they may refer to variables introduced higher in the tree. The $F$-test indicates when there is a significant difference between the state-action values before and after a split. If the significance is above a certain threshold, the algorithm will introduce the split. As with the G algorithm, the TG algorithm is model-free and does not memorise its experience. When the algorithm introduces a split, all of the statistics in that region of the state space must be cleared.

## 8 Evaluation

The purpose of this section is to evaluate AMPS, to see how it performs on problems, how its components contribute to the behaviour of the agent, and how it compares to other approaches. There are many ways to evaluate performance. Since the objective of AMPS is to learn *competent* behaviour—not necessarily *optimal* behaviour—it is necessary to have some definition of competence. Other researchers (e.g., Goodrich, Stirling, & Boer, 2000; Crook, 2006) have investigated learning *satisficing* behaviour instead of optimal behaviour. Simon (1956) coined the word *satisficing* in the context of agents with bounded rationality, using the word to mean behaviour that satisfies some minimum level of competency in achieving a goal. This chapter measures performance by counting the number of problems solved within a fixed time limit per episode. In the experiments, the agent is limited to 500 steps per episode in the Taxi World prob-

lem and 100 s of simulated time per episode in the Corner World problem. These limits were selected to accommodate a small amount of random exploration in each episode, regardless of initial state.

Many of the experiments compare estimates of mean performance, run time, and model complexity. When reporting estimates of the mean of a random variable, this chapter always provides 99% confidence intervals in both the text and in diagrams with error bars. Estimates in this chapter are always based on 100 samples to allow for reasonably tight confidence intervals. Some of the experiments involve comparing the performance of different algorithms. Claims that one algorithm performs better than another on a particular problem are based on 100 runs with typically 100 episodes in each run to allow the behaviour of the agent to stabilise. The degree of significance of these claims are measured by the Wilcoxon signed-rank test (Wilcoxon, 1945) and $p$-values are provided in the text. This chapter uses the nonparametric Wilcoxon test because the data points are paired as a result of using the same series of initial states across runs.

## 8.1 Problem Domains

The experiments in this section involve three different domains. These domains were chosen to evaluate how well AMPS can cope with different kinds of problem structures and representations. Considered are problems with both discrete and continuous state and action spaces.

### 8.1.1 Goal World

The simplest domain, considered primarily for illustrative purposes, is Goal World. In this problem, the agent is allowed to rotate and translate at some fixed velocity with the objective of reaching some goal. The agent perceives the state of the world as a vector

$$(x, y, \theta, \mathit{goal}\text{-}x, \mathit{goal}\text{-}y)\,,$$

where $x$ and $y$ are the coordinates of the centre of the agent, $\theta$ is the orientation of the agent, and $\mathit{goal}\text{-}x$ and $\mathit{goal}\text{-}y$ are the coordinates of the centre of the goal. The actions available to the agent include ROTATE-LEFT, ROTATE-RIGHT, and MOVE-FORWARD. The agent may translate only in the direction of its current orientation. When the centre of the agent comes within a small threshold of the centre of the goal, the agent receives unit reward.

### 8.1.2 Taxi World

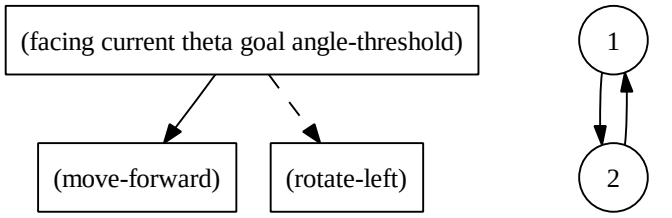
In the Taxi World domain, the agent drives a taxi, picking up passengers and delivering them to their destination. The state of the world is represented by some tuple

$$(\mathit{in}\text{-}\mathit{taxi}, \mathit{taxi}\text{-}x, \mathit{taxi}\text{-}y, \mathit{passenger}\text{-}x, \mathit{passenger}\text{-}y, \mathit{destination}\text{-}x, \mathit{destination}\text{-}y)\,,$$

where the first element indicates whether the passenger is in the taxi and the others represent the $x$ and $y$ coordinates of the taxi, passenger, and destination. The actions available to the agent include MOVE-NORTH, MOVE-SOUTH, MOVE-EAST, MOVE-WEST, PICK-UP, and DROP-OFF. These actions have their intended effect most of the time, but with some small probability they behave differently. The agent receives unit reward when the passenger reaches its destination.

### 8.1.3 Corner World

The Corner World domain has a continuous state and action space. The agent starts at a random location on the starting line and maneuvers along an L-shaped track to the finish line. The state space is represented by the tuple $(x, y)$ corresponding to the location of the agent. The agent may translate in any direction at some velocity. The agent samples the environment and makes control decisions at some frequency. As the agent interacts with the world, the position of the agent is perturbed by an amount selected from a normal distribution with some standard deviation.




(a) The map. (b) The model.

Figure 7: The map and the model learned in the Goal World domain. The map partitions the state space with a decision graph. This particular decision graph distinguishes states according to whether the robot is within some threshold of facing the goal. The model is represented by an SMDP estimated from experience. Shown is the connectivity between the two regions.

## 8.2 Demonstration

This section demonstrates AMPS in the Goal World, Taxi World, and Corner World domains. This section will begin with a discussion of the Goal World domain because it is the simplest to understand and for AMPS to solve.

### 8.2.1 Goal World

In order to build an abstract SMDP, AMPS needs some way to partition the state space. As Section 4 discusses, there are many ways to represent the partitioning of the state space. For example, one might use a nearest neighbour approach to cluster together similar states according to a distance metric, or one might use a decision graph that partitions the state space through a series of tests.

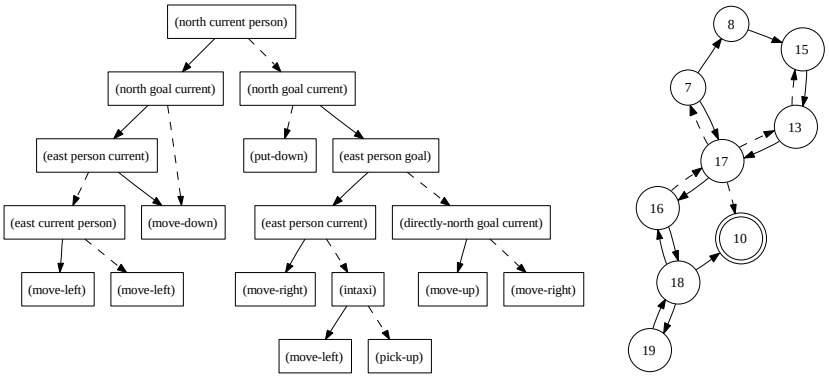
If AMPS is to use a decision graph to partition the state space, it must be provided with a description of the kinds of tests it is to use. Since the state space can be represented as a real-valued vector, one might wish to use hyperplanes to carve the state space into a finite set of regions. Alternatively, one could use grounded predicates as tests. The current implementation of AMPS supports the specification of typed relations, functions, and constants in XML. When AMPS discovers that it might be advantageous to introduce a distinction in the state space, it constructs a grounded predicate from the provided set of relations, functions, and constants that maximises some quality measure such as information gain (Section 4.3.1).

The set of types for the Goal World domain include states, points, angles, and scalar thresholds. There are several functions that take parameters and return values of these types. The relations include $facing$, $left$, $right$, and $close$. Since there are many ways of combining the relations, functions, and constants to form a grounded predicate, it is desirable to eliminate any unnecessary predicates from consideration, which is done using a set of theorems and a theorem prover as Section 4.3.4 explains.

Without a prior model of the dynamics and without intermediate reward to encourage progress towards the goal, the Goal World task is quite difficult. Random exploration by itself is unlikely to be successful within a reasonable amount of time. Therefore, the first episode involves an approximately optimal teacher steering the robot to its goal. The initial state is selected randomly from a uniform distribution over all possible states. The duration of this training episode is random, but for the run described here it lasted 6.5 s with decisions being made at 10 Hz.

After this first episode, AMPS is left on its own to control the robot. At first, AMPS treats the entire state space as a single region. Because the action MOVE-FORWARD was the action that the teacher used immediately before reaching the goal, the planning process identifies MOVE-FORWARD as the greedy action. AMPS begins executing the action MOVE-FORWARD until it crashes into the wall. Since crashing into the wall is identified as a failure, the failure revision process (Section 4.1.2) splits the state space based on samples that led to failure and samples that led to success. Figure 7 illustrates the state of the decision graph and the model after the split.

After the state space is split, AMPS performs prioritised plan revision and arrives at a competent plan.




(a) The map. (b) The model.

Figure 8: The map and the model learned in the Taxi World domain after one episode. Greedy transitions in the model are indicated by solid directed edges and non-greedy transitions are indicated by dashed directed edges. Region 10 (indicated with a double border) contains the goal.

The plan is to rotate left until the robot faces the goal and then move forward. If moving forward results in the robot not facing the goal, then the robot rotates left until it faces it again. This plan can be suboptimal because in some situations it may be better to rotate right instead of left.

### 8.2.2 Taxi World

The Taxi World domain is much more complex than the Goal World domain. This task involves picking up passengers and depositing them at their destinations. The taxis, passengers, and destinations occupy single cells within a $20 \times 20$ grid. The dynamics are nondeterministic. With probability 0.1, the taxi moves in a direction orthogonal to the intended direction.

Again, it is necessary to define the representation for perceptual distinctions. As with the Goal World task, AMPS might use a decision graph with predicates synthesised from an XML file. Some of the binary relations provided include $north$, $south$, $east$, $west$, $directly\text{-}north$, $directly\text{-}south$, $directly\text{-}east$, and $directly\text{-}west$. A set of theorems specify when it is possible to prune grounded predicates from consideration (Section 4.3.4).

As with the goal-world domain, it is extremely unlikely that random exploration will result in the agent accomplishing its task in a reasonable amount of time. Therefore, a noisy teacher directs the agent through the task. The noisy teacher chooses a random action 10% of the time and an optimal action the remainder of the time. In this chapter, the Taxi World experiments involve three training episodes by this noisy teacher. Three training episodes appears to be just enough training to enable the agent to solve the problem without having to rely too heavily on random exploration. The number of control examples provided by the noisy teacher in three episodes depends on the random state initialisation, the stochasticity in the system, and control noise. In this initial run, the teacher makes 126 control decisions over the three training episodes.

Following the three training episodes, AMPS is left on its own. Its initial simplistic model of the system dynamics is not sufficient for solving the task. The agent begins by trying to put down a passenger that is not in the taxi, resulting in failure. The map revision process adds a perceptual distinction and the plan revision process updates the plan. The agent then moves left until it crashes into the wall, resulting in failure and the addition of another perceptual distinction. It makes a few more mistakes until it heads towards the passenger. By the time the taxi reaches the passenger, the state space is divided into seven regions. The taxi then successfully picks up the passenger and delivers the passenger to its destination. Figure 8 shows the map and the structure of the SMDP after the successful completion of the task.

Following two successful episodes on its own, AMPS struggles in the sixth episode and does not manage to complete the task successfully within 500 steps. The failure of AMPS in this episode is due to "thrashing" between regions 55 and 52 (Figure 9). From region 55, the agent tries to move right so that

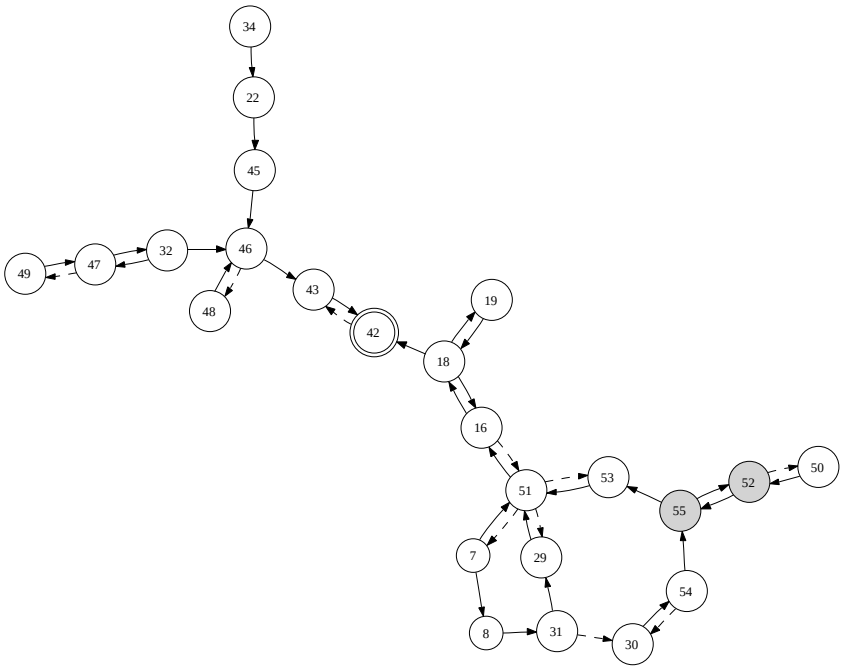



Figure 9: The Taxi World model in the sixth episode where thrashing occurs between regions 55 and 52 (shaded). Region 42 (indicated with a double border) contains the goal.

it can transition to region 53 on its way to region 42, the region containing the goal. However, moving right can also bring the agent to region 52. Because of the actual position the taxi is in at this point in the episode, moving right will usually bring the agent to region 52 instead of region 53. Once in region 52, the agent tries to move left, bringing it back to region 55, and so the cycle continues until the agent takes an exploratory action or revises its model. Thrashing in this situation is due to aliasing in region 55. Eventually, the transition revision process makes a perceptual distinction to resolve the issue with perceptual aliasing when the priority of adding that distinction rises above a specified threshold.

The agent is able to solve the task within the allotted time for 88 of the 100 episodes. Figure 10 shows the structure of the decision graph at the beginning of the tenth episode.

### 8.2.3 Corner World

In contrast with the Taxi World domain, there are uncountably many states and actions in the Corner World domain. Hence, the agent must be capable of generalisation in both the perceptual and actional spaces. This section describes two ways of achieving generalisation in this problem. The first involves learning a decision graph, and the second involves instance-based generalisation with a distance metric.

When using a decision graph for generalisation in the Corner World domain, it is useful to make distinctions using hyperplanes. To avoid overfitting and to reduce computation, AMPS should only consider distinctions involving basis-orthogonal hyperplanes. The tests that partition the state space involve comparing either the $x$ or $y$ coordinate of the position of the agent with some constant value. The tests that partition the action space involve checking whether the direction of the movement of the agent falls within some open interval.

As with the other domains in this section, random exploration is not likely to bring the agent to the goal. In this chapter, the agent is provided with two training episodes by a noisy teacher. A quarter of the time the teacher performs random actions and the remainder of the time the teacher follows a competent, but suboptimal, hand-crafted policy. Figure 11(a) shows the trajectories of the two training episodes.

On the third episode, AMPS continues on its own, revising its model and plan as it acquires experience.

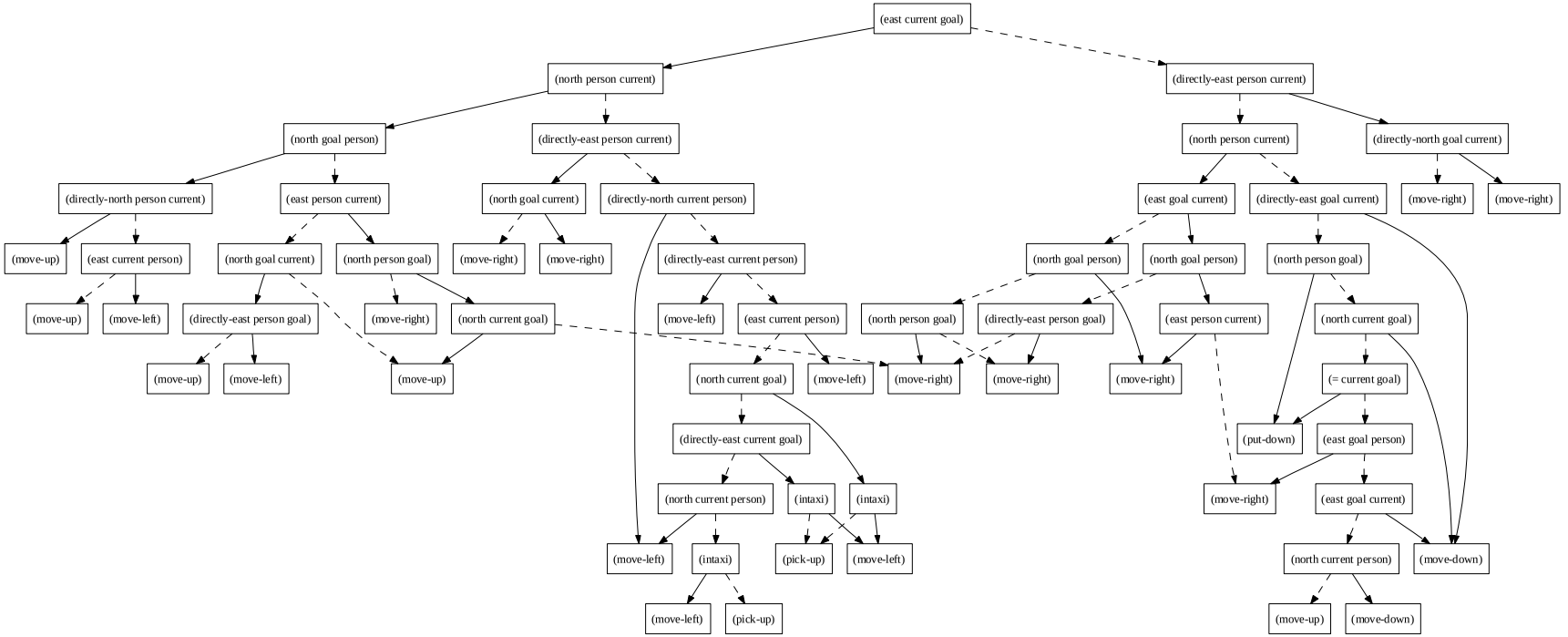
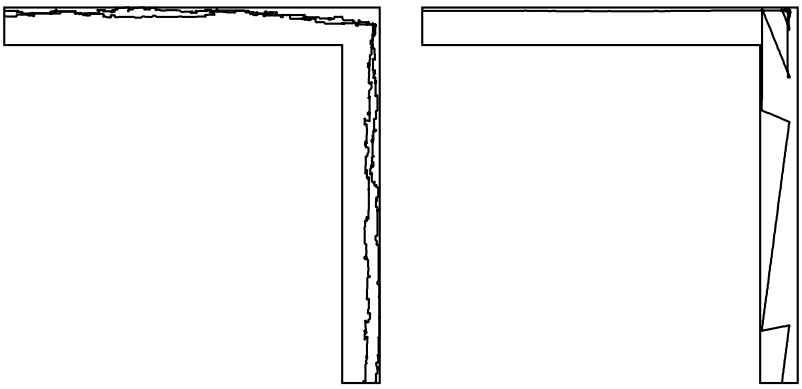



Figure 10: A decision graph learned while interacting with the Taxi World domain.



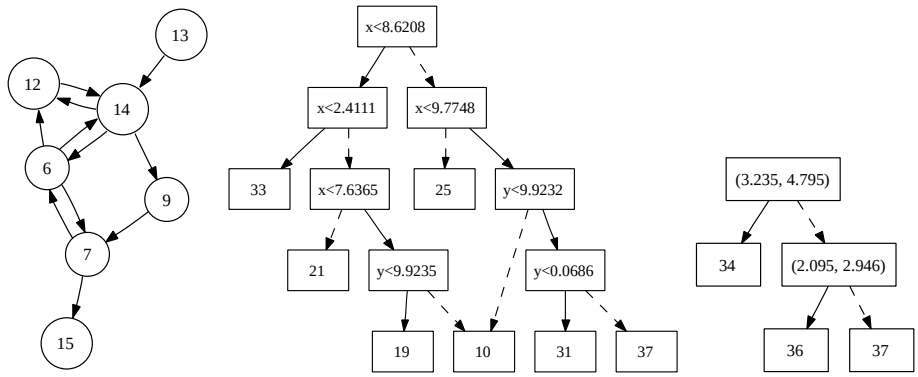
(a) Episodes 1 and 2 (training).

(b) Episode 3.

(c) Episode 4.

(d) Episode 100.

Figure 11: Trajectories through Corner World at various stages when using AMPS with decision graphs that partition the state and action spaces.




(a) The model. (b) The state map. (c) The action map.

Figure 12: The model and the state and action maps learned by the end of the third episode in the Corner World. Each node in the model is labelled by the ID of its associated state region. Each leaf node in the state map is labelled by the ID of the greedy action region associated with that region of the state space. Each leaf node in the action map is labelled by the ID of the action region. The action map shown is associated with the state region with ID 14.

Since its model of the system dynamics is initially too simplistic, the agent at first has difficulty navigating towards the goal as shown in Figure 11(b). However, AMPS is able to revise its model and plan quickly enough to complete its task within a reasonable amount of time.

Figure 12 shows the model and the decision graph that partitions the state space that AMPS learns by the end of the third episode. The figure also shows the partitioning of the action space associated with one of the state regions. The model AMPS learns by the end of the third episode provides a smooth trajectory to the goal in the fourth episode as Figure 11(c) shows.

As Section 6.2 discusses, the experience revision process purges old experiences. In this sample run, AMPS only retains experience from the ten most recent episodes in memory. When the agent starts the eleventh episode, AMPS purges the experience from the first episode. Keeping only ten episodes worth of memory for this task does not impair performance. More complicated tasks might require retaining more experience in memory.

Figure 11(d) shows the trajectory of the agent in its 100th episode. The trajectory is close to optimal, but the agent could have started moving left sooner around the corner. Even with more experience, the trajectory is still unlikely to be exactly optimal because of the need for explorative actions and the tradeoff between simplicity and accuracy in the model.

During the 100 episodes, AMPS revises the model by splitting and merging regions. As Figure 13 shows, the first few episodes involve increasing the complexity of the model. AMPS adds perceptual distinctions because of observed differences in expected value and failure along trajectories. The complexity of the model stabilises and with additional experience it becomes apparent to AMPS that it can simplify the model by merging some of the regions.

This initial simple model learned by AMPS produces reasonable behaviour until the 29th episode. The agent crashes against the walls several times in this episode. Although the learned model is good at predicting the dynamics of the system when the agent glides through the middle area of the track, the model does not accurately represent the dynamics when the agent is close to the wall, leading to relatively poor behaviour in episode 29 as shown in the plot in Figure 14. By the end of the 30th episode, the number of state regions doubles to accommodate the complexity of modelling wall interactions. With additional data, AMPS eventually simplifies the model. By the end of the 100th episode there are only ten state regions (Figure 15).

Figure 14 plots the duration of time to solve the task. The agent has some difficulty in a few episodes, but otherwise the behaviour is usually close to optimal and better than that of the teacher.

Using instance-based generalisation with a distance metric also provides good results. As Figure 16

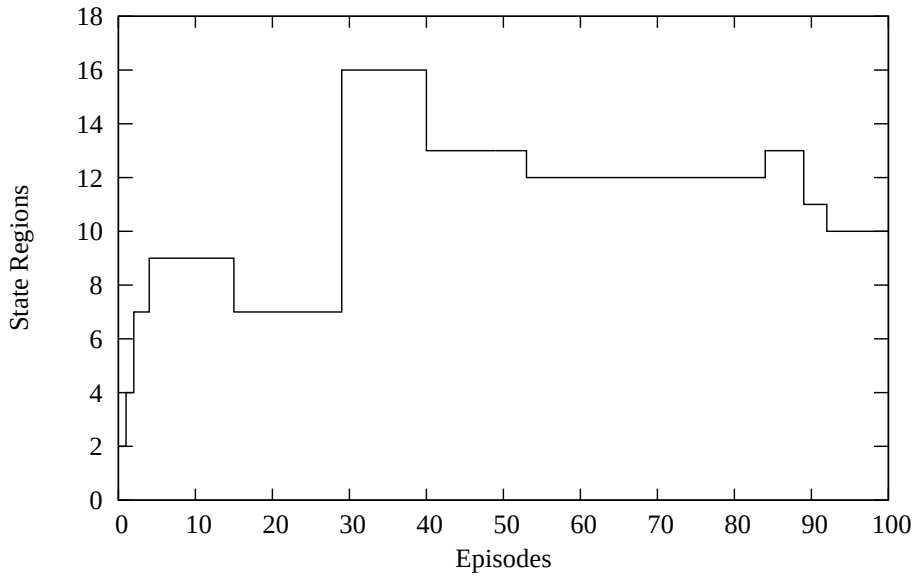



Figure 13: A plot showing how the number of state regions in the model changes as the agent acquires experience in the Corner World domain. As the agent begins to acquire experience, there is a steep increase in the number of state regions. Eventual reductions in the number of regions are due to the simplification process.

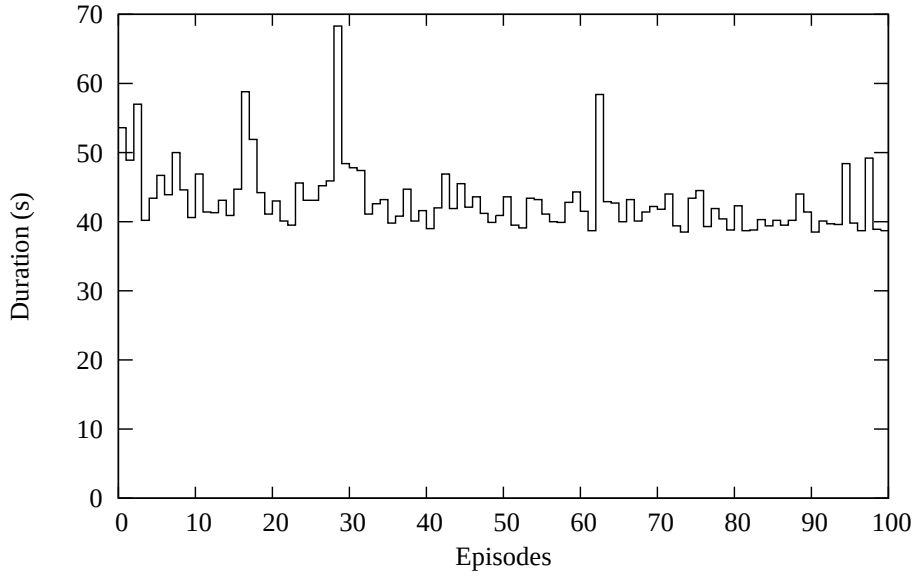



Figure 14: A plot of the duration of time required to solve the Corner World task over a series of episodes. Variation is due to randomness in the initial position on the starting line, environmental stochasticity, exploratory actions, and the evolution of the model and plan.

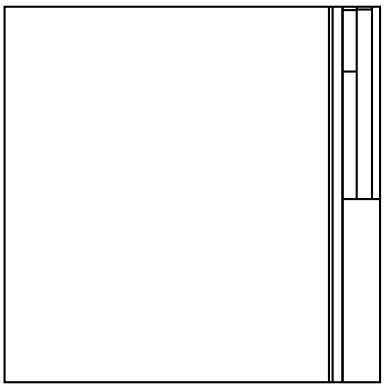


Figure 15: A physical partition of the state space at the end of the 100th episode of Corner World. There are ten regions. One region is extremely narrow and difficult to see at this scale. The two rectangular slivers near the top right of the space correspond to the same region.

reveals, AMPS solves the task almost always 20% faster than the noisy teacher. Only twice in the 100 episodes of the sample run does AMPS solve the task slower than the teacher. As Figure 17 shows, nearest-neighbour generalisation partitions the state space differently from decision-graph generalisation. Figure 17 illustrates the variation of the complexity of the model over time.

## 8.3 Examination

This section examines the mechanisms involved in AMPS in greater detail. In particular, this section closely examines separation heuristics, model simplification, and value clipping.

### 8.3.1 Separation Heuristics

In many problems, it is important that AMPS uses both the value revision heuristic and the failure revision heuristic. The importance of these two heuristics interacting together is especially pronounced in the Taxi World domain. If AMPS is left on its own for ten episodes after three episodes of training, the number of problems it solves when episodes are limited to 500 steps is

- $6.20 \pm 0.66$, when using both value and failure revision,
- $0.05 \pm 0.06$, when using only value revision, and
- $1.46 \pm 0.50$, when using only failure revision.

Value revision is useless without failure revision because it never results in a split. Value revision can only produce splits when there is a significant difference in value along different transitions leaving a state region in the model. When the agent commences its interaction with the world, there is only one state region other than the synthetic terminal region (Section 3.3), making it impossible for value revision to introduce any perceptual or actional distinctions.

Failure revision is always the first to introduce a split. Once the state or action space is split, the value revision process is able to refine the model. For the Taxi World problem, value revision is essential for good performance. Failure revision by itself does not produce competent behaviour. However, in some domains such as Corner World, failure revision by itself may be enough to produce satisfactory behaviour.

### 8.3.2 Model Simplification

In addition to introducing perceptual and actional distinctions into the model, it is important for the agent to simplify the model when additional experience indicates that doing so would be useful. Without simplification, the number of state and action regions can grow very quickly.

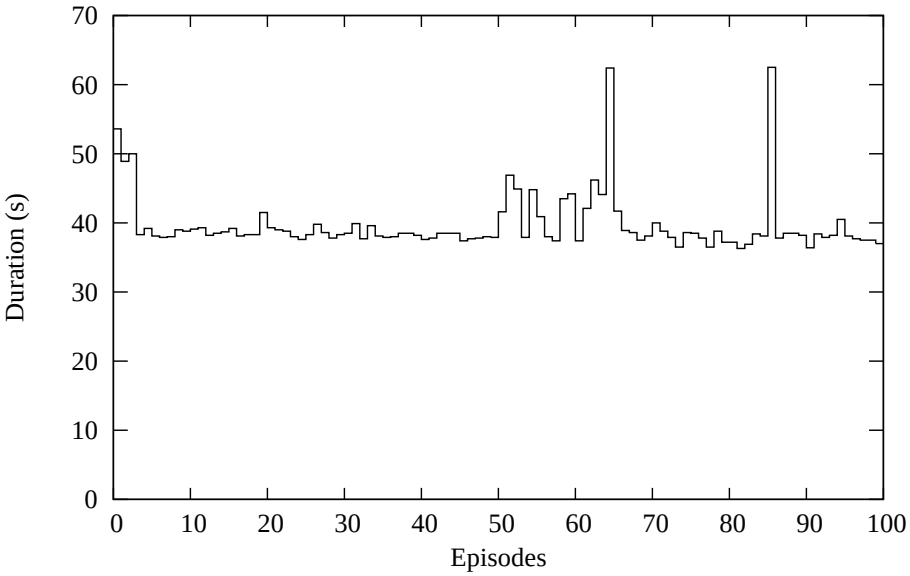
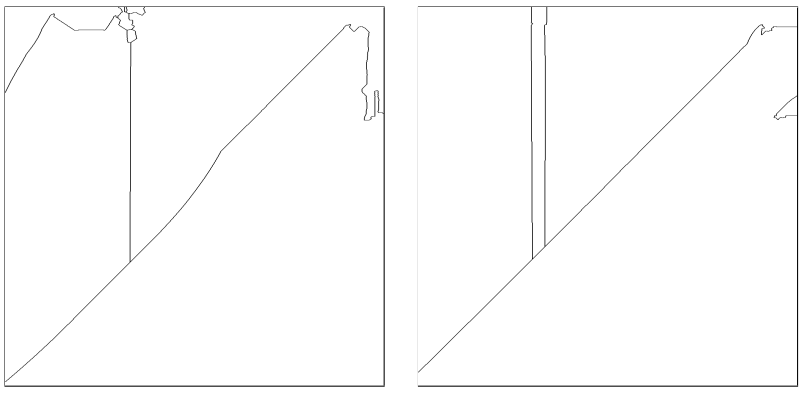



Figure 16: A plot of the duration of time required to solve the Corner World task over a series of episodes using nearest-neighbour generalisation. AMPS almost always performs significantly better than the teacher after only two training episodes.



(a) Episode 5. (b) Episode 50.

Figure 17: A physical partition of the Corner World state space at the end of episodes 5 and 50 learned with nearest-neighbour generalisation.

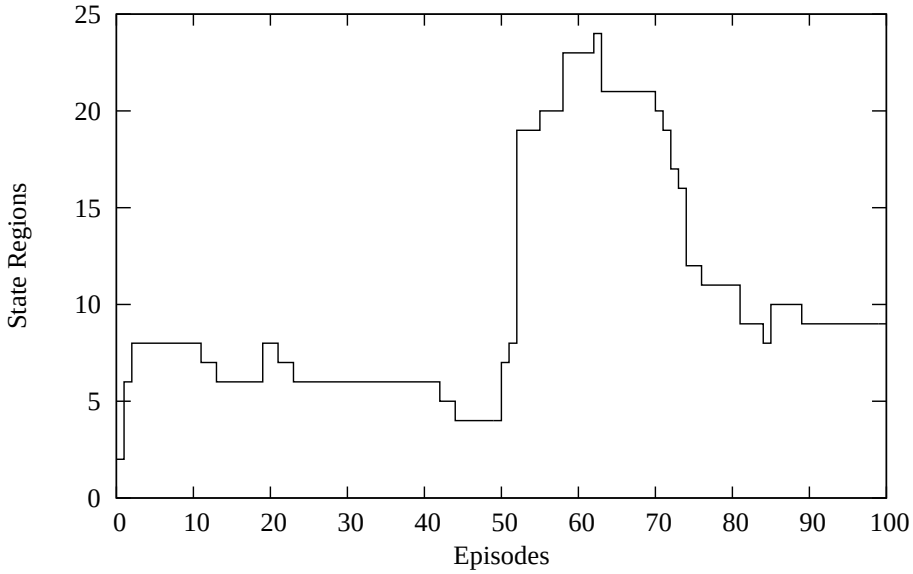



Figure 18: A plot showing how the number of state regions in the model changes over time using a nearest-neighbour approach in Corner World.

Figure 19 shows how the number of state regions changes over time with and without simplification in the Corner World domain. With simplification, the number of state regions in the model stays relatively small and constant. Without simplification, the number of state regions increases very quickly over time.[6] By the 100th episode, AMPS with simplification learns a model with only 10 regions and AMPS without simplification learns a model with 185 regions. Without model simplification, the agent solves only 58 problems instead of 100 and the simulation requires over three times the computation. In the Taxi World domain, the difference between AMPS with simplification and AMPS without simplification is less pronounced, but still significant as Figure 20 shows. For the experiments in this chapter, the agent retains 25 episodes of experience for the Taxi World domain and 10 episodes of experience for the Corner World domain.

The benefits of having a simpler model extend beyond savings in computation and memory. Fewer state and action regions can provide better generalisation from limited data. With fewer regions, more experience is available to estimate the transition probabilities, reward, and duration distribution. With a better model, the agent is able to construct a better plan. With model simplification, the agent is able to solve all 100 episodes of Corner World within a limit of 100 s of simulated time per episode. Without model simplification, the agent is only able to solve 58 episodes within the time limit. In the Taxi World domain, the agent solves 89 episodes with simplification and 82 without simplification.

### 8.3.3 Value Clipping

When interleaving incremental modelling with planning, updating the values of state regions solely through prioritised value iteration is not sufficient for satisfactory performance. There are situations where prioritised value iteration takes unnecessarily long to converge. Section 4.5 introduces value clipping as a way to speed the process. Value clipping is an efficient process that clips value estimates that are noticeably too high or too low given a quick inspection of the estimated reward function and model topology.

A sample run of 100 episodes of the Taxi World problem using the same parameters as usual illustrates the frequency of value clipping (Figure 21). Although there are some episodes that do not involve value clipping, some episodes require over 35 clips. Value clipping is most important during the early episodes

[6] Any decrease in the number of state regions without model simplification is due to the automatic removal of regions that are left without samples after the experience revision process purges old experience (Section 6.2).

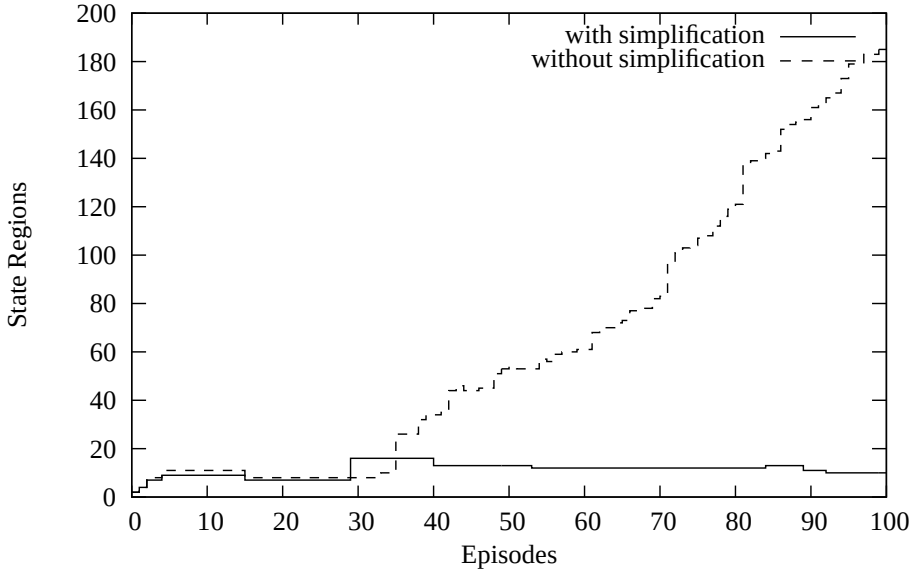



Figure 19: A plot showing how the number of state regions in the model changes over time using a decision-graph approach with and without model simplification in the Corner World domain.

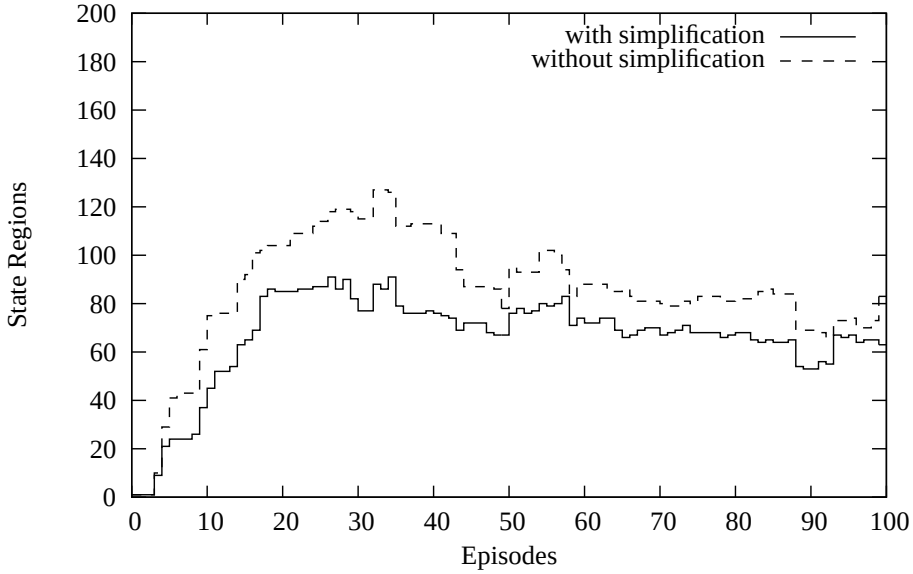



Figure 20: A plot showing how the number of state regions in the model changes over time using a decision-graph approach with and without model simplification in the Taxi World domain.

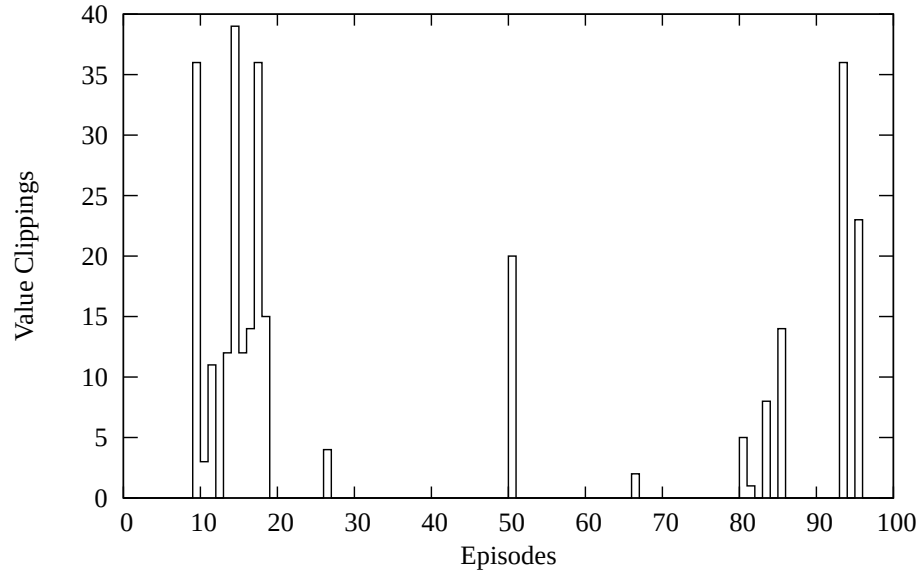



Figure 21: A plot of the number of clips performed by the value clipping process over 100 episodes in the Taxi World domain.

when the experience of the agent in the environment is sparse. During the first 25 episodes without a teacher, AMPS with value clipping is expected to solve $18.48 \pm 1.23$ problems whereas AMPS without value-clipping is expected to solve only $13.48 \pm 1.17$ problems. Not all types of problems stand to gain such a significant benefit from value clipping. Model revision in the Corner World domain, for example, rarely results in a need for value clipping.

## 8.4 Comparison

This section compares AMPS with several other approaches, including behavioural cloning, temporal difference learning, prioritised sweeping, UTree, and TTree. Side-by-side empirical evaluation of performance can be misleading. Different approaches make different assumptions about what is known. For example, prioritised sweeping assumes a discrete state and action space whereas AMPS learns discretisations while it interacts with the world. Behavioural cloning assumes the availability of a competent teacher, but temporal difference learning can eventually learn satisfactory behaviour without any guidance from an external source. The purpose of this section is not to identify which approach is "best." After all, the best approach to use strongly depends on the prior knowledge of the agent, the structure of the learning process, and the constraints of the task. The purpose of this section is to identify the strengths and weaknesses of various approaches and how they relate to AMPS.

### 8.4.1 Behavioural Cloning

An agent may learn to solve a task purely through supervised learning. The transfer of expertise from a skilled teacher to a learning agent is known as behavioural cloning. There have been many successful applications of behavioural cloning (e.g., Sammut, Hurst, Kedzier, & Michie, 1992; Pomerleau, 1993), but there are some major limitations to this approach in general. Perhaps the most fundamental problem with a behavioural clone is that it does not understand the effects of its actions, rendering it incapable of independent problem solving. Behavioural clones depend entirely on a skilled teacher. The agent is unable to use the experience it accumulates on its own to its advantage.

A supervised learning algorithm may use the same mechanism that AMPS uses in Section 8.2.2 to introduce distinctions in the state space. The learning algorithm may split the state space incrementally

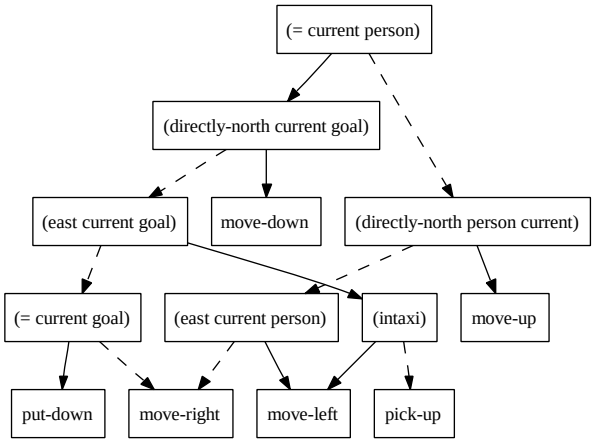



Figure 22: A decision graph learned by a behavioural clone for the Taxi World domain.

| | AMPS | clone |
|---|---|---|
| decision graph | 70.04±8.37 | 55.81±6.26 |
| nearest neighbour | 95.42±3.46 | 54.33±8.57 |

Table 2: A comparison of AMPS with behavioural clones using different generalisation methods in Corner World. Shown are the expected number of problems solved in 100 episodes without a teacher.

in a top-down fashion like other tree induction systems (e.g., Quinlan, 1993). If there are multiple leaf nodes belonging to the same category, the algorithm may merge these leaves together. After three training episodes in Taxi World, a decision graph induction system learns the decision graph shown in Figure 22. Learning static decision graphs in this way does not lead to good policies. Decision graph behavioural clones only solve $36.62 \pm 5.55$ on average of the first 100 episodes following three training episodes, whereas AMPS with decision graph partitioning solves $82.45 \pm 2.65$ on average.

In the Corner World domain, a behavioural clone may learn using decision-graph or nearest-neighbour generalisation. If the agent induces a decision graph, the decision graph can be extremely large and unnecessarily complex. In the experiments, the decision graph after two training episodes carves the state space into approximately 250 regions. The performance of decision graph clones is very poor in comparison to AMPS. If the behavioural clone uses nearest-neighbour generalisation with a Euclidean distance metric, the performance is still quite poor. Table 2 compares the performance of AMPS and behavioural clones with decision-graph and nearest-neighbour generalisation.

### 8.4.2 Temporal Difference Learning

When the state or action space is large, tabular temporal difference learning is not likely to perform well. The success of tabular temporal difference learning depends on the agent trying every action from every state multiple times. In the Taxi World and Corner World domains, tabular temporal difference learning results in behaviour not much better than random.

One may combine temporal difference learning with a Monte Carlo approach using eligibility traces. Several algorithms such as Q($\lambda$) due to Watkins (1989) and Sarsa($\lambda$) due to Rummery (1995) use eligibility traces to improve performance. Even with eligibility traces, performance can be quite poor without generalisation.

A domain expert may manually discretise large or continuous state and action spaces and then perform tabular temporal difference learning over the abstract states and actions. Table 3 summarises some results in the Taxi World domain using Q($\lambda$) and Sarsa($\lambda$) with a static abstraction.[7] The table reveals that using the

[7] The discretisation in these experiments is the finest partition allowed by the separation mechanism used by AMPS in this domain.

| Agent | Perf. |
|---|---|
| Random | 0.30±0.12 |
| Q($\lambda$) | 0.27±0.12 |
| Sarsa($\lambda$) | 0.27±0.12 |
| Abstract Q($\lambda$) | 45.93±3.50 |
| Abstract Sarsa($\lambda$) | 43.83±3.67 |
| AMPS | 82.45±2.65 |

Table 3: A comparison of the performance of several temporal difference learning algorithms against AMPS in the Taxi World domain. Shown are the expected number of problems the various kinds of agents can solve in 100 episodes. In the experiments $\lambda$ is set to 0.8.

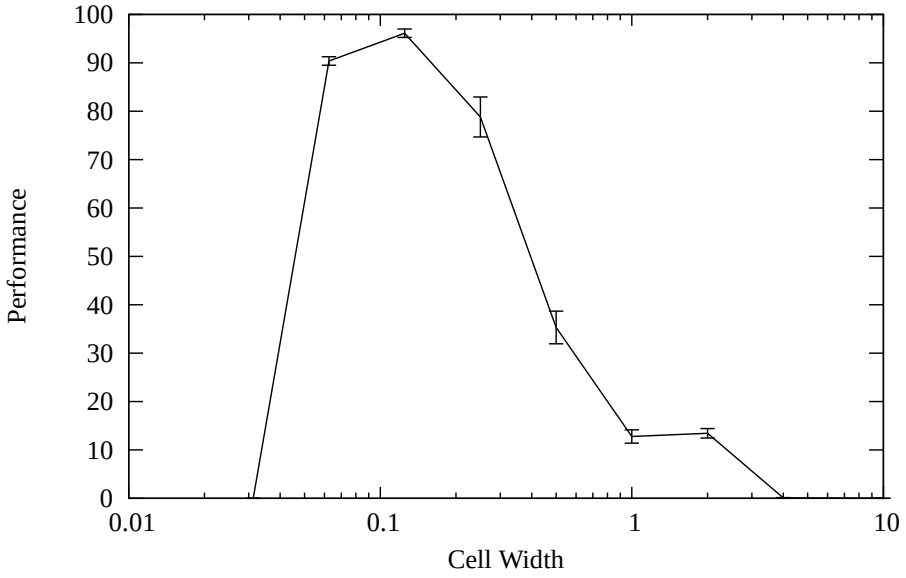



Figure 23: A plot indicating how the coarseness of the discretisation in the Corner World problem affects the performance of Q($\lambda$). In these experiments, the $10 \times 10$ m track was discretised at different resolutions. The horizontal axis represents different cell widths in metres. Performance is measured according to the average number of successes in 100 episodes. Again, $\lambda$ is set to 0.8.

abstraction greatly improves performance. However, neither Q($\lambda$) nor Sarsa($\lambda$) with abstraction performs nearly as well as AMPS. AMPS has the advantage because it uses a dynamic abstraction and explicitly learns a model of the system dynamics over which it may plan.

Another set of experiments investigate the performance of temporal difference learning with eligibility traces in the Corner World domain. Since the state space is continuous and the sample rate of the agent is finite, the agent will almost never find itself in a previously experienced state. Hence, temporal difference learning without generalisation will perform no better than an agent that moves randomly throughout the state space.

With an abstraction at the right level of detail, temporal difference learning can learn competent behaviour remarkably quickly. The experiments apply Q($\lambda$) to a grid-based discretisation of Corner World with variable cell widths. Unlike the earlier experiments with AMPS in Corner World, the agent is provided with only four actions corresponding to moving in each of the cardinal directions. Restricting the available actions greatly simplifies the problem. Figure 23 plots the cell width in the discretisation against performance. Clearly, the level of abstraction greatly affects performance.

The success of Q($\lambda$) at the right level of abstraction in Corner World is due to the use of an eligibility trace and teacher as well as the structure of the problem. The first two training episodes result in value updates in the cells along the trajectories of the teacher. These value updates usually result in reasonable

policy actions at the visited cells. When the agent is left to use Q($\lambda$) on its own, it will flail around until it reaches a cell visited by the teacher. If the agent chooses an on-policy action, it will generally head in the right direction. If the agent loses track of the trail left by the teacher, random exploration will bring it back on track, so long as the discretisation is sufficiently coarse. If the discretisation is too coarse the agent cannot make the necessary perceptual distinctions to navigate around the corner.

One of the main challenges of applying table temporal difference learning is determining the right level of abstraction. The advantage of AMPS is that it learns the appropriate level of abstraction while interacting with the environment.

Temporal difference learning can be used with generalisation approaches other than abstraction. Goebel (2005) investigates the use of self-organising maps and neural networks as generalisation methods for temporal difference learning. His thesis shows that these approaches perform rather poorly on both the Corner World and Taxi World tasks.

### 8.4.3 Prioritised Sweeping

Unlike temporal difference learning, prioritised sweeping (Moore & Atkeson, 1993) is a model-based reinforcement learning algorithm. Model-based approaches generally learn better policies with less experience than model-free approaches (Peng & Williams, 1993; Atkeson & Santamaría, 1997). Model-free approaches, such as those considered in Section 8.4.2, only update the value function along experienced trajectories, whereas model-based approaches can perform updates whenever the estimated model indicates it is necessary.

Implementing prioritised sweeping based on an existing implementation of AMPS is relatively easy. Prioritised sweeping is essentially AMPS with a static map that partitions the state and action spaces. Since the partitions do not change, there is no reason to retain old experiences in memory after updating the model estimate. AMPS, in contrast, retains as much experience in memory as possible so that it can use the potentially costly experience to update the model following map revision. If AMPS did not retain its experience in memory, it would have to relearn the model in areas altered by the map revision process.

Although prioritised sweeping typically consumes much less memory than AMPS, its primary limitation is that it requires a domain expert to provide a discretisation of the state and action spaces at a suitable resolution. AMPS, in contrast, attempts to learn the correct discretisation from experience. For some problems, it is easy to determine a suitable discretisation, but for other problems, it is not so straightforward. It is important to use a suitable discretisation.

AMPS, because of its use of dynamic abstraction, is able to outperform prioritised sweeping on the Taxi World task. Without an abstraction, just as with the temporal difference algorithms, prioritised sweeping performs randomly. With the same abstraction Section 8.4.2 uses with the Taxi World experiments, prioritised sweeping is expected to solve $67.96 \pm 3.30$ problems, which is about 50% more than $Q(\lambda)$ and Sarsa($\lambda$) with abstraction. AMPS, which learns its own abstractions, can solve on average $82.45 \pm 2.65$ problems.

Prioritised sweeping can learn competent behaviour very quickly for the Corner World task with an appropriate level of abstraction. As with the experiments in Section 8.4.2 involving Q($\lambda$), the experiments with prioritised sweeping use a grid-based discretisation of the state-space and an alternative version of the problem with only four actions. As can be expected, prioritised sweeping performs better than temporal difference learning with eligibility traces over a wider range of resolutions.

### 8.4.4 UTree

JRLF includes an implementation of the basic ideas underlying UTree. The version of UTree in JRLF is not intended to exactly replicate the work of McCallum (1995), and there are a few significant differences:

- **Representation:** McCallum assumes an attribute-value representation. The JRLF version of UTree borrows much of its implementation from AMPS, allowing it to use more general representations such as real-valued vectors as in the Continuous U-Tree algorithm (Uther, 2002).
- **Modelling and planning:** McCallum uses MDPs to model the system dynamics, but the JRLF version of UTree uses SMDPs. McCallum uses a single sweep of value iteration to update the value

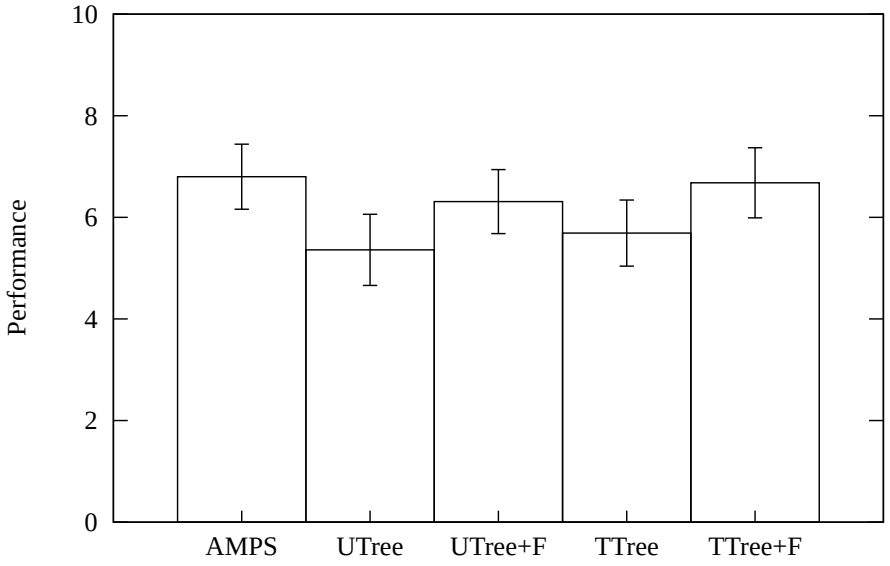



Figure 24: A comparison of the performance of various algorithms on the Taxi World problem. Performance is measured by the expected number of problems solved in ten episodes following three episodes of training.

function and policy, but the JRLF version of UTree uses the same prioritised value iteration algorithm as AMPS.

- **Historical distinctions:** McCallum allows for the introduction of historical distinctions. Instead of only splitting on attributes of the current state, his version of UTree can split on attributes of previously observed states. The current version of UTree as part of JRLF does not include this functionality, but it is an area of further research.

- **Significance tests:** McCallum only uses the Kolmogorov-Smirnov statistical test when deciding when and how to split a region. The version of UTree packaged with JRLF can use a variety of different statistical tests, but uses sum-squared error by default like the Continuous U-Tree algorithm (Uther, 2002).

- **Prioritisation:** McCallum introduces splits to any region where the split is deemed significant above a certain threshold by the Kolmogorov-Smirnov test statistic. The JRLF version of UTree prioritises its introduction of perceptual distinctions.

Figure 24 compares the performance of UTree against AMPS on ten episodes of the Taxi World problem following three training episodes. To control for the use of failure sensing by AMPS, the performance of a version of UTree with failure revision is also shown in the figure as UTree+F. Both versions of UTree do not perform as well as AMPS[8] and require much more time to run as Figure 25 shows.

There are several reasons why AMPS performs better than UTree with respect to both behavioural competency and real-time computation.

- **Model simplification:** Unlike UTree, AMPS attempts to simplify its model when experience indicates it might be useful. As the experiments in Section 8.3.2 indicate, simplification can greatly improve performance. Not only do simpler models require less memory to represent and less time to compute their optimal policies, but they are able to better generalise from limited experience. UTree,

[8]The $p$-values for the Wilcoxon signed-rank test (Wilcoxon, 1945) when comparing AMPS with UTree and Utree+F are $3.8 \times 10^{-6}$ and $0.038$ respectively.

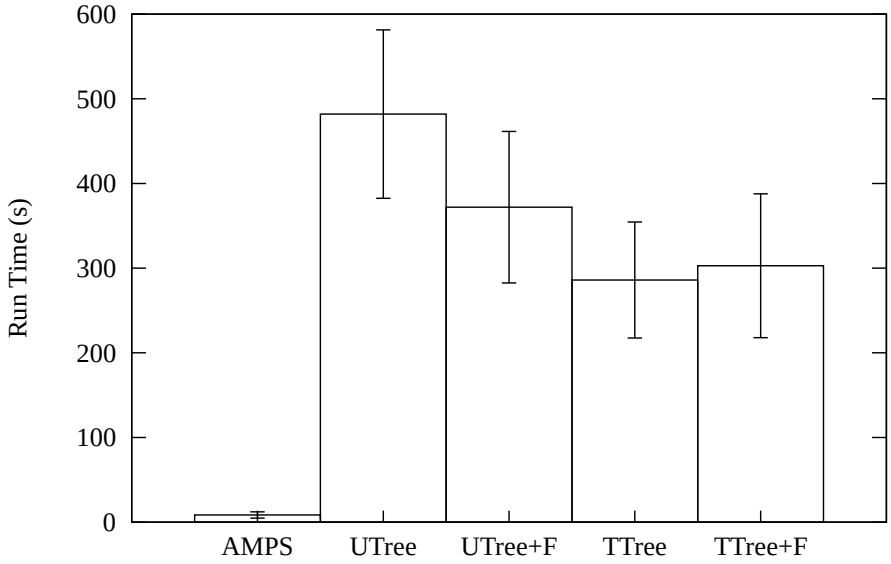



Figure 25: A comparison of expected run times of various algorithms on the Taxi World problem.

like other abstraction methods for reinforcement learning, does not attempt to simplify its model in the same way as AMPS.

- **Model-based heuristics:** AMPS uses model-based splitting heuristics, but UTree uses a samples-based splitting heuristic. To decide when to split, AMPS only needs to inspect the estimated model, which can be done very quickly. If AMPS decides to introduce a perceptual distinction, it uses an approach based on supervised learning, which is typically very fast. In contrast with the model-based heuristics of AMPS, the samples-based heuristic that UTree uses can be computationally expensive. Before UTree can determine whether to split a particular region, it must estimate the value of each individual sample within the region. UTree estimates the value of each sample according to Equation 5. Unfortunately, it is not possible to cache these estimated values with the individual observations because they depend on the global structure of the model and the current plan. Once UTree computes the values of all of the samples in the region, it iterates through every possible way of splitting the samples. For each way of splitting the samples into two or more new regions, UTree computes the difference between the resulting distributions of sample values. If these distributions are significantly different, UTree will introduce the split.

- **Trajectory heuristics:** UTree estimates the value of a sample based on the immediate reward associated with the sample and the value of the region to which the next sample in the trajectory belongs. If the agent only receives non-zero reward at the goal, as is the case in the Taxi World and Corner World domains, then the only difference in sample value will be at the edges of regions. Hence, as Uther (2002, Section 3.5) observes, the splits that UTree introduces only occur at the edges of regions, resulting in the creation of many more regions than necessary. AMPS avoids this problem in a manner similar to TTree by introducing distinctions based on trajectories instead of individual samples.

### 8.4.5 TTree

JRLF contains an implementation of TTree for the purpose of comparison with AMPS. As with the JRLF version of UTree, the JRLF version of TTree shares much of its implementation with AMPS. Besides the lack of model simplification in TTree, the main difference between TTree and AMPS is the way they

introduce splits. The JRLF version of TTree introduces splits based on the estimated value of samples, just like UTree. However, the estimation of sample values is different between TTree and UTree.

TTree computes the value of a sample state $s$ in region $S$ in the following way. Let $R$ be the accumulated discounted reward the agent receives while transitioning from $s$ in $S$ to some other region, and let $V$ be the estimated value of the resulting region. If $\beta$ is the continuous-time discount rate and $t$ is the amount of time required to transition from $s$ in $S$ to another region, then the value of the sampled state $s$ is $R + e^{-\beta t}V$.

As with UTree, the TTree algorithm may use any heuristic measure of significance to decide when and how to introduce a perceptual distinction. By default, JRLF uses sum-squared error as it does with UTree. The original version of TTree, however, uses a minimum message length heuristic.

As Figure 24 shows, TTree performs marginally better than UTree ($p = 0.08$), but not quite as well as AMPS ($p = 2.47 \times 10^{-5}$). When the agent uses TTree with failure revision, it can perform almost as well as AMPS on this problem, ignoring issues with computation. As Figure 25 reveals, the JRLF implementation of TTree requires around 35 times the computation of AMPS. The inefficiency of TTree is due to the same reasons as UTree (Section 8.4.4).

# 9 Conclusion and Further Work

AMPS is a general and flexible system for integrating modelling and planning for learning adaptive behaviour. The system efficiently revises its model, efficiently revises its plan, and efficiently generalises from limited experience. One may apply AMPS to a wide variety of problems without having to reimplement the core system when changing representations.

In summary, this paper makes the following contributions:

- **A novel approach that connects reinforcement learning to supervised and unsupervised learning techniques for the purpose of generalisation.** Although other reinforcement learning algorithms have used unsupervised learning techniques (e.g., self-organising maps) and data structures frequently used in supervised learning (e.g., neural networks), the way in which this approach combines supervised and unsupervised learning with reinforcement learning is original.

- **A flexible framework for leveraging different kinds of representations.** Generalisation from limited experience depends upon being able to extract structure from the underlying representation. Section 4 discusses ways of tailoring AMPS to the representation using decision graph and nearest neighbour approaches, and Section 8 shows how these approaches can be used in practice.

- **An efficient model-based implementation that learns competent behaviour from little experience.** Section 8 shows that AMPS can learn extremely quickly on different kinds of problems. AMPS appears to be the first system that prioritises both planning and modelling. Existing model-based algorithms such as UTree and TTree are not nearly as efficient or flexible as AMPS as discussed at the end of Section 8.

- **An abstraction system that dynamically increases and decreases resolution in both the state space and action space.** The abstraction methods surveyed in Section 7 only introduce new perceptual distinctions as the agent accumulates experience; they never decrease resolution when further experience indicates doing so might be useful. AMPS appears to be the first abstraction approach to perform generalisation in the action space, although some local and parametric approximation methods do attempt action space generalisation.

Since AMPS is a new approach and has been tested on relatively few problems, further research is necessary to determine how well the approach scales to more complex problems. Besides applying the existing implementation of AMPS to new problems, there are several other promising lines of further research.

- **Logical decision trees:** AMPS currently only uses standard decision graphs. It might be interesting to see how the generalisation of logical decision trees to logical decision graphs will fare on problems with relational structure such as Blocks World.

- **Historical Distinctions:** Currently, AMPS only makes distinctions based on the current observation. Depending on the problem, it might be useful to make distinctions based on past observations or actions as done in UTree. Further thought is require for generalising the ideas in UTree to AMPS.
- **Experience Consolidation:** The experience revision process removes old experiences from memory to keep from exceeding some specified memory limit and to reduce computational demands. It would be interesting to investigate ways of consolidating old experiences instead of disposing of them completely.
- **Parallelisation:** AMPS is designed specifically for a single processor that executes a stream of sequential instructions that have access to a single bank of memory. However, the modelling and planning processes in AMPS may be executed in parallel, across multiple processors. Further research would address various parallelisation issues such as how to best distribute state regions across processors and how to minimise communication between processors to enhance scalability.


## Acknowledgements

The author would like to thank Gillian Hayes for helpful suggestions on an earlier draft of this paper.


# References


Aha, D. W., Kibler, D., & Albert, M. K. (1991). Instance-based learning algorithms. *Machine Learning*, *6*(1), 37–66.

Al-Ansari, M. A., & Williams, R. J. (1999). Robust, efficient, globally-optimized reinforcement learning with the Parti-game algorithm. In Kearns, M. J., Solla, S. A., & Cohn, D. A. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 11, pp. 961–967. MIT Press, Cambridge, Mass.

Atkeson, C. G., & Santamaría, J. C. (1997). A comparison of direct and model-based reinforcement learning. In *Proceedings of the International Conference on Robotics and Automation*, Vol. 4, pp. 3557–3564. IEEE.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*(1–2), 81–138.

Bellman, R. (1957). *Dynamic Programming*. Rand Corporation research study. Princeton University Press, Princeton.

Bertsekas, D. P., & Castañon, D. A. (1989). Adaptive aggregation methods for infinite horizon dynamic programming. *IEEE Transactions on Automatic Control*, *34*(6), 589–598.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Optimization and neural computation series. Athena Scientific, Belmont, Mass.

Breiman, L., Friedman, J. H., Olsen, R. A., & Stone, C. J. (1984). *Classification and Regression Trees*. The Wadsworth statistics/probability series. Wadsworth International Group, Belmont, Calif.

Chapman, D., & Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In Mylopoulos, J., & Reiter, R. (Eds.), *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pp. 726–731, San Mateo, Calif. Morgan Kaufmann.

Chávez, E., Navarro, G., Baeza-Yates, R., & Marroquín, J. L. (2001). Searching in metric spaces. *ACM Computing Surveys*, *33*(3), 273–321.

Cover, T. M., & Hart, P. E. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, *13*(1), 21–27.

Cristianini, N., & Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines: And Other Kernel-Based Learning Methods*. Cambridge University Press, Cambridge.

Crook, P. A. (2006). *Learning in a State of Confusion: Employing Active Perception and Reinforcement Learning in Partially Observable Worlds*. Ph.D. thesis, School of Informatics, University of Edinburgh.

Driessens, K. (2004). *Relational Reinforcement Learning*. Ph.D. thesis, Departement Computerwetenschappen, Katholieke Universiteit, Leuven.

Driscoll, J. R., Gabow, H. N., Shrairman, R., & Tarjan, R. E. (1988). Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, *31*(11), 1343–1354.

Duda, R. O., Hart, P. E., & Stork, D. G. (2000). *Pattern Classification* (2nd edition). Wiley, New York.

Džeroski, S., de Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*(1–2), 7–52.

Efron, B., & Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. Monographs on Satistics and Applied Probability. Chapman and Hall, New York.

Everitt, B. S., Landau, S., & Leese, M. (2001). *Cluster Analysis* (4th edition). Oxford University Press, New York.

Fayyad, U. M., & Irani, K. B. (1992). On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, *8*(1), 87–102.

Fikes, R., Jenkins, J., & Frank, G. (2003). JTP: A system architecture and component library for hybrid reasoning. Tech. rep. KSL-03-01, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, *34*(3), 596–615.

Genesereth, M. R., & Fikes, R. E. (1992). Knowledge interchange format, version 3.0 reference manual. Tech. rep. KSL-92-86, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Gentle, J. E. (2002). *Elements of Computational Statistics*. Statistics and computing. Springer, New York.

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

Goebel, M. C. (2005). An empirical investigation into function approximation with reinforcement learning. Master's thesis, School of Informatics, University of Edinburgh.

Goodrich, M. A., Stirling, W. C., & Boer, E. R. (2000). Satisficing revisited. *Minds and Machines*, *10*(1), 79–109.

Heath, D. (1992). *A Geometric Framework for Machine Learning*. Ph.D. thesis, Department of Computer Science, Johns Hopkins University.

Heath, D., Kasif, S., & Salzberg, S. (1993). Induction of oblique decision trees. In Bajcsy, R. (Ed.), *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Vol. 2, pp. 1002–1007, San Mateo, Calif. Morgan Kaufmann.

Hjaltason, G. R., & Samet, H. (2003). Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, *28*(4), 517–580.

Howard, R. A. (1971). *Dynamic Probabilistic Systems*, Vol. 2 of *Series in decision and control*. Wiley, New York.

Jain, A. K., Dubes, R. C., & Chen, C.-C. (1987). Bootstrap techniques for error estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *9*(5), 628–633.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kochenderfer, M. J. (2006). *Adaptive Modelling and Planning for Learning Intelligent Behaviour*. Ph.D. thesis, School of Informatics, Unversity of Edinburgh.

Kumar, P. R. (1985). A survey of some results in stochastic adaptive control. *SIAM Journal on Control and Optimization*, *23*(3), 329–380.

Lachenbruch, P. A., & Mickey, M. R. (1968). Estimation of error rates in discriminant analysis. *Technometrics*, *10*(1), 1–11.

Likhachev, M., & Koenig, S. (2003). Speeding up the Parti-game algorithm. In Becker, S., Thrun, S., & Obermayer, K. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 15, pp. 1563–1570. MIT Press, Cambridge, Mass.

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, *8*(3–4), 293–321.

McCallum, A. K. (1995). *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. thesis, Department of Computer Science, University of Rochester.

Michie, D., & Chambers, R. A. (1968a). BOXES: An experiment in adaptive control. In Dale, E., & Michie, D. (Eds.), *Machine Intelligence*, Vol. 2, pp. 137–152. Oliver and Boyd, Edinburgh, Scotland.

Michie, D., & Chambers, R. A. (1968b). 'Boxes' as a model of pattern-formation. In Waddington, C. H. (Ed.), *Towards a Theoretical Biology*, Vol. 1, pp. 206–215. Edinburgh University Press, Edinburgh.

Milligan, G. W., & Cooper, M. C. (1985). An examination of procedures for detecting the number of clusters in a data set. *Psychometrika*, *50*, 159–179.

Moore, A. W., & Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, *13*(1), 103–130.

Moore, A. W., & Atkeson, C. G. (1995). The Parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, *21*(3), 199–233.

Munos, R., & Moore, A. (2002). Variable resolution discretization in optimal control. *Machine Learning*, *49*(2–3), 291–323.

Murthy, S. K., Kasif, S., & Salzberg, S. (1994). A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, *2*, 1–32.

Nilsson, N. (1992). Toward agent programs with circuit semantics. Tech. rep. STAN-CS-92-1412, Department of Computer Science, Stanford University.

Nilsson, N. J. (2000). Learning strategies for mid-level robot control: Some preliminary considerations and results. www.robotics.stanford.edu/users/nilsson/trweb. Robotics Laboratory, Department of Computer Science, Stanford University.

Peng, J., & Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, *1*(4), 437–454.

Pomerleau, D. A. (1993). *Neural Network Perception for Mobile Robot Guidance*. The Kluwer international series in engineering and computer science. Kluwer Academic Publishers, Boston.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley series in probability and mathematical statistics. Wiley, New York.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. The Morgan Kaufmann series in machine learning. Morgan Kaufmann Publishers, San Mateo, Calif.

Ramon, J. (2002). *Clustering and Instance Based Learning in First Order Logic*. Ph.D. thesis, Departement Computerwetenschappen, Katholieke Universiteit, Leuven.

Rummery, G. A. (1995). *Problem Solving with Reinforcement Learning*. Ph.D. thesis, Department of Engineering, University of Cambridge.

Sammut, C., Hurst, S., Kedzier, D., & Michie, D. (1992). Learning to fly. In Sleeman, D. H., & Edwards, P. (Eds.), *Proceedings of the Ninth International Workshop on Machine Learning*, pp. 385–393, San Mateo, Calif. Morgan Kaufmann.

Schölkopf, B., & Smola, A. J. (2000). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass.

Schweitzer, P. J., Puterman, M. L., & Kindle, K. W. (1985). Iterative aggregation-disaggregation procedures for discounted semi-Markov reward processes. *Operations Research*, *33*(3), 589–605.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, *27*, 379–423 and 623–656.

Shawe-Taylor, J., & Cristianini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge.

Simon, H. A. (1956). Rational choice and the structure of the environment. *Psychological Review*, *63*, 129–138.

Simons, J., Brussel, H., de Schutter, J., & Verhaert, J. (1982). A self-learning automaton with variable resolution for high precision assembly by industrial robots. *IEEE Transactions on Automatic Control*, *27*(5), 1109–1113.

Smart, W. D. (2002). *Making Reinforcement Learning Work on Real Robots*. Ph.D. thesis, Department of Computer Science, Brown University.

Smith, A. J. (2002). Applications of the self-organising map to reinforcement learning. *Neural Networks*, *15*(8–9), 1107–1124.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass.

Thrun, S. B. (1992). The role of exploration in learning control. In White, D. A., & Sofge, D. A. (Eds.), *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pp. 527–559. Van Nostrand Reinhold, Florence, Kentucky.

Uther, W. T. B. (2002). *Tree Based Hierarchical Reinforcement Learning*. Ph.D. thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College, University of Cambridge.

Weiss, S. M. (1991). Small sample error rate estimation for k-NN classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *13*(3), 285–289.

Wiering, M. (1999). *Explorations in Efficient Reinforcement Learning*. Ph.D. thesis, Faculteit der Wiskunde, Informatica, Natuurkunde en Sterrenkunde, Universiteit van Amsterdam.

Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics Bulletin*, *1*(6), 80–83.

Wilson, D. R., & Martinez, T. R. (1997). Improved heterogeneous distance functions. *Journal of Artificial Intelligence Research*, *6*, 1–34.

Wingate, D., & Seppi, K. D. (2005). Prioritization methods for accelerating MDP solvers. *Journal of Machine Learning Research*, *6*, 851–881.

Xu, R., & Wunsch, D. (2005). Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, *16*(3), 645–678.