Perception-Based Generalization in Model-Based Reinforcement Learning

BY Bethany R. Leffler

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Michael L. Littman

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2009

ABSTRACT OF THE DISSERTATION

Perception-Based Generalization in Model-Based Reinforcement Learning

by Bethany R. Leffler

Dissertation Director: Michael L. Littman

In recent years, the advances in robotics have allowed for robots to venture into places too dangerous for humans. Unfortunately, the terrain in which these robots are being deployed may not be known by humans in advance, making it difficult to create motion programs robust enough to handle all scenarios that the robot may encounter. For this reason, research is being done to add learning capabilities to improve the robot's ability to adapt to its environment. Reinforcement learning is well suited for these robot domains because often the desired outcome is known, but the best way to achieve this outcome is unknown.

In a real world domain, a reinforcement-learning agent has to learn a great deal from experience. Therefore, it must be sample-size efficient. To do so, it must balance the amount of exploration that is needed to properly model the environment with the need to use the information that it has already obtained to complete its original task. In robot domains, the exploration process is especially costly in both time and energy. Therefore, it is important to make the best possible use of the robot's limited opportunities for exploration without degrading

the robot's performance.

This dissertation discusses a specialization of the standard Markov Decision Process (MDP) framework that allows for easier transfer of experience between similar states and introduces an algorithm that uses this new framework to perform more efficient exploration in robot-navigation problems. It then develops methods for an agent to determine how to accurately group similar states. One proposed technique clusters states by their observed outcomes. To make it possible to extrapolate observed outcomes to as-yet unvisited states, a second approach uses perceptual information such as the output of an image-processing system to group perceptually similar states with the hope that they will also be related in terms of outcomes. However, there are many different percepts from which a robot could obtain state groupings. To address this issue, a third algorithm is presented that determines how to group states when the agent has multiple, possibly conflicting, inputs from which to choose. Robot experiments of all algorithms proposed are included to demonstrate the improvements that can be obtained by using the approaches presented.

Preface

Portions of this dissertation are based on work previously published by the author [Leffler et al., 2005, 2007, 2008].

Acknowledgements

This dissertation could not have been accomplished without the support and assistance of the following people. Thank you all so much.

My advisor and friend, Michael Littman for providing me with guidance for the last four and a half years. His ability to lead me forward while allowing me to find my own direction, led me to develop as a researcher.

My thesis committee whose efforts improved the quality of my dissertation.

My undergraduate advisor turned friend Jennifer Kay, for being the first to see the computer scientist in me. Her continued support and friendship has been invaluable.

The RL^3 lab for the many things that I learned and laughs that I had spending time with them.

Chris Mansley, Ari Weinstein, and Warren Code for providing needed and much appreciated proof-reading of this dissertation. I look forward to doing the same for them.

My friends and family, especially Paula Checchi, Mary Ann Smith, and the Lefflers for supporting my decision to go to graduate school and not allowing me to doubt that I belonged there.

Lastly, my husband, Timothy, for always believing in me. His ever flowing words of encouragement and endless patience helped me through every step of this journey. Words cannot describe how grateful I am for your friendship.

Dedication

This dissertation is dedicated to my parents, Karen and Randy. Their love, understanding, and encouragement has been there through all the challenges in my life. Thank you.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1
Introduction

Recent advances in robotics have allowed for robots to venture into places too dangerous for humans. In some severe environments, it is difficult for humans to control the robots; autonomy is needed. One such environment is Mars [Volpe et al., 2000]. One message takes several minutes to be transmitted from a controller on Earth to a rover on Mars. Instead of waiting for each instruction to be transmitted from humans, a Mars rover, shown in Figure 1.1, performs many different routines based on its sensor data.

To perform navigation tasks, the robot needs to know how its location in the world will change as the result of taking an action. This information is called the robot's *dynamics* [Murray et al., 1994]. By modeling its dynamics, a robot can calculate the series of actions required to achieve a navigation task. Many robots, including the Mars rover, are given equations to calculate the expected outcome from taking an action [Balaram, 2000]. These equations are usually a function of many variables such as a vehicle velocity and contact point location that are carefully studied and calibrated by NASA engineers before the robot is deployed. These parameters can then be adjusted by the robot after it arrives on Mars.

Unfortunately, it is difficult to allow for all scenarios that the robot may encounter with a set of equations. Over time, the terrain that the robot travels on or even the robot itself may change causing the robot to have an improper model of the world. Without an accurate understanding of its dynamics, the robot cannot reliably perform its task. One way make a more reliable robot is to

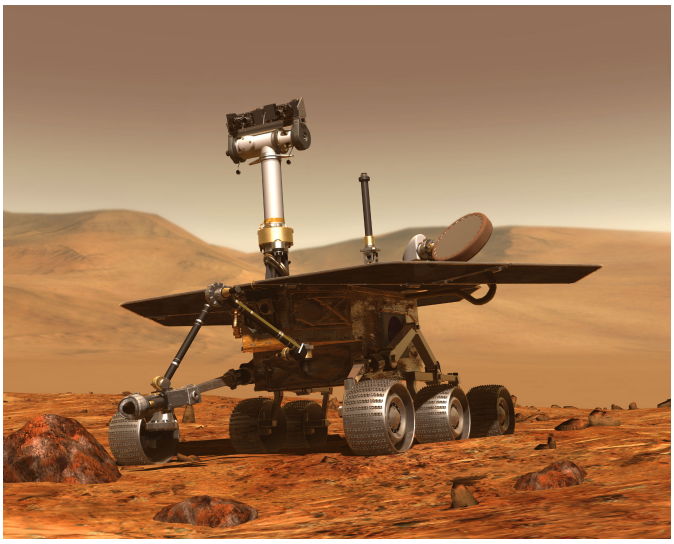


Figure 1.1: Artists rendering of a Mars rover. Courtesy of NASA/JPL-Caltech.

give it the ability to *learn*.

Machine learning is the process in which a machine reliably improves its performance of a task on the basis of experience [Mitchell, 2006]. In a learning system, an *agent*, or dedicated processing unit, is fed experience and updates its model of the environment.[1] In a robot-navigation task, performance is often evaluated on the *path*, or sequence of actions, that the robot takes to arrive at its goal.

The field of machine learning can be divided into sub-problems with many different inputs, outputs, and objectives. Robot problems can be formulated as several different machine-learning problems (see Chapter 2). This dissertation will formulate the robot-navigation task as a *reinforcement-learning* problem.

[1]It is important to note that the agent is not the robot. Often, it is designed so that an agent is a learner and planner, and the robot is the actuator. However, many agents can be combined to control a single robot—each with their individual tasks such as path planning, learning the dynamics model, language processing, etc. [Thrun et al., 2000].

1.1 Problem Setting

Reinforcement learning is a form of machine learning where the agent modifies its behavior based on *rewards* provided by the environment [Sutton and Barto, 1998]. This approach is especially well suited for tasks where a desired outcome is known, but the best way to achieve this outcome is unknown. For instance, in the robot-navigation task, the desired outcome would be for the robot to arrive in a desired location. So, a large reward would be given to the agent any time that it entered this desired location. Learning from this reward, the agent would calculate what actions it should perform at any given time.

The current status of the environment is detailed as a *state*. The state can include the status of all objects in an environment, but is often simplified to monitor only objects that can affect the agent's ability to achieve its given task. For example, in a simple robot-navigation task, the state consists of the robot's location and orientation. However, if a task were to have a robot navigate through a room full of people, the state would consist not only of the location and orientation of the robot, but the location of each of the people in the room as well.

Figure 1.2 shows the interaction between a reinforcement-learning agent and its environment. Initially, an agent receives the current state. An agent then chooses an *action* to take. A set of actions can be continuous like turning a wheel to a desired position, or discrete like choosing between preset actions such as "turn left". Each individual action can be continuous or discrete in time. For continuous actions, an agent determines how long to perform each action. An discrete alternative is to have each action performed for a predetermined length of time. Environments detailed in this dissertation will consist of a discrete set of discrete actions (for continuous domains, see the work of Doya [2000]).

After an action is completed, the environment sends the updated current state and reward to the agent. This cycle repeats until the agent reaches a *terminal*

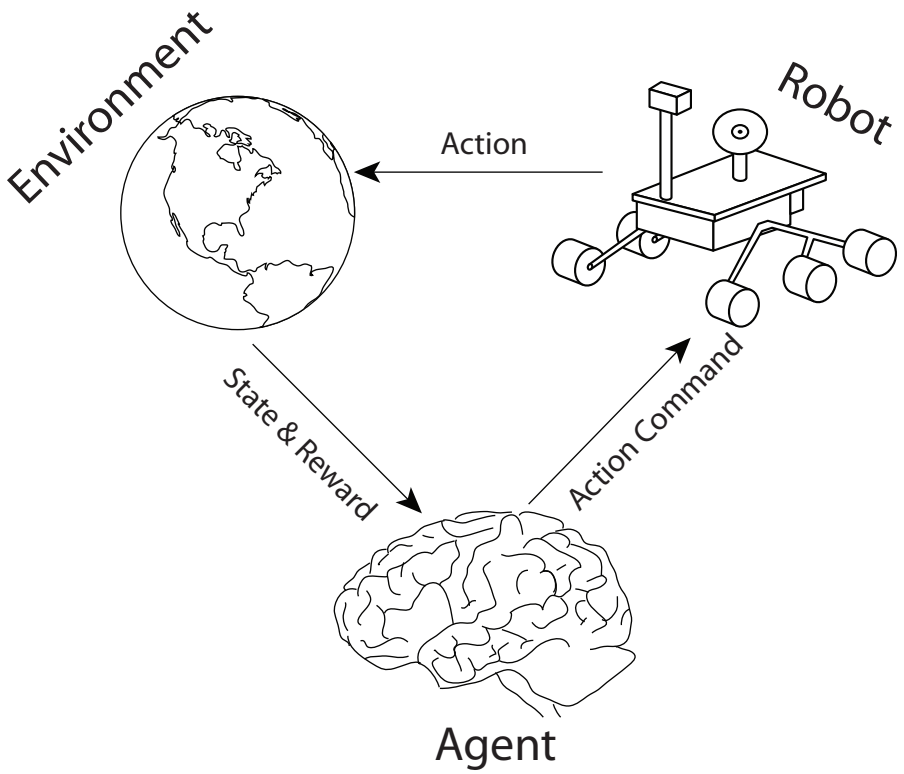


Figure 1.2: A diagram showing the exchange of information in a reinforcement-learning problem.

state. A state is considered terminal if the episode ends once the agent arrives in it.

In robot-navigation tasks, the objective of the reinforcement-learning agent is to determine the best path to reach the *goal state* which is usually a terminal state with high reward.

1.1.1 Path Planning

As mentioned earlier, the performance in a robot-navigation task is determined by the path that the agent takes to arrive at its goal. A *policy* is a set of actions, one for each state, where the agent has determined each action to be the best action for its corresponding state [Sutton and Barto, 1998]. These actions are chosen based on the long-term expected reward that the agent hopes to receive.

The *optimal policy* is the policy that leads to the highest possible reward from all states. One way to learn the optimal policy would be for the agent to calculate the *value*, or expected reward, of each of the state-action pairs using data collected about the environment from previous experience.

A *model-free*, or direct, reinforcement-learning agent estimates the value of a state-action pair based solely on the rewards received from previous experiences of that state-action pair [Sutton and Barto, 1998]. Rewards received from performing each same state-action pair are averaged and mean values for all state-action pairs are stored. When the agent chooses an action, it looks at each of the state-action pairs for the current state and chooses the action corresponding to the highest value.

Another way to obtain the value of a state-action pair is through *model-based* reinforcement learning [Sutton and Barto, 1998]. With this approach, the agent calculates the environment's reward structure and dynamics model from the state and reward information provided by the environment in previous experiences. With this information, the agent builds a model of the world and uses it to calculate the values of each state-action pair (see Section 1.2 for more details).

Both model-free and model-based approaches have their benefits. Model-free methods require very little calculation because of their direct use of previous rewards. Model-based methods create a more complete model of the environment, requiring less experience but more calculation to determine a reasonable policy. Although there is no single right answer on the question of model-based versus model-free learning, in the robot domain, actions in the world are generally slow compared to the time needed for computation. Therefore, I have opted to focus primarily on model-based methods, although I have run model-free algorithms for comparison. This dissertation explores the use of model-based reinforcement learning to perform robot-navigation tasks.

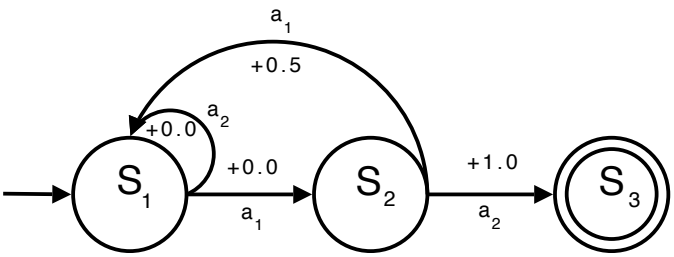


Figure 1.3: A simple Markov Decision Process (MDP) model.

1.2 Markov Decision Processes

The traditional model used to formulate the model-based reinforcement-learning tasks is the Markov Decision Process (MDP) model [Puterman, 1994]. In addition to states and actions, an MDP model includes a *transition function* and *reward function*. The transition function, $T(s, a, s')$, can be thought of as an agent's action model or, in a robot domain, the agent's model of a robot's dynamics. It is a probability distribution over possible next states, s', from any given state-action pair. This information is helpful for an agent to be able to plan a path, and subsequently reach its goal.

The reward function, $R(s, a)$, is the immediate reward for each state-action pair, $\langle s, a \rangle$. An agent uses this information to determine its policy.

Figure 1.3 shows a simple MDP for illustration. For this MDP, the set of states S consists of $\{s_1, s_2, s_3\}$, the set of actions A consists of $\{a_1, a_2\}$, and s_3 is a terminal state as marked by the double circle. A *transition* is the probability of reaching any given state given a state and action. In the figure, arrows indicate the transitions.

This environment is *deterministic* because all of the transitions have a probability of 0 or 1. For instance, if the agent takes action a_1 in state s_1, the probabilities of arriving in states s_3 and s_2 are 0 and 1, respectively.

The rewards are shown as real numbers near each of the arrows. For instance, the reward for taking action a_2 from state s_2 is 1.0.

1.2.1 Solving the Model

Using an MDP model of the environment, an agent can solve for the optimal policy, π^*, using dynamic programming [Puterman, 1994]. The Bellman equation, shown in Equation 1.1, determines an optimal value for a state, $V(s)^*$, by performing a max operation over all actions from the current state. The value of a state-action pair, $Q(s, a)$, is calculated by summing up the largest value of all possible next states, s', weighted by the probability that action a would take the agent to s'. That sum is then multiplied by a *discount factor*, γ, and added to the reward for that state-action pair:

$$V^*(s) = \max_a Q(s, a) = \max_a \{R(s, a) + \gamma \sum_{s'} T(s, a, s')V^*(s')\}. \qquad (1.1)$$

The discount factor determines how heavily the agent should weight future reward versus immediate reward. This parameter is set to a value between zero and one. When the discount factor is zero, future actions have no bearing on the agent's decision. In contrast, when the discount factor is set to one, every action in the future is as important as the current action. If the environment does not have a reachable terminal state, the discount factor must be less than one to guarantee that the value of a policy is well defined.

The Bellman equation in Equation 1.1 form a set of simultaneous equations through recursion. *Value iteration* is the process of repeatedly computing the value of each state from values stored in a table during previous iterations [LaValle, 2006]. This process will continue for a user-specified number of steps or until it converges to a user-specified tolerance.

These values, in combination with the transition function, can be used to determine the best action to take in each state. For example, an agent in the simple MDP environment of Figure 1.3 with a discount factor of 0.1 would calculate the value of state s_1 by calculating the value of action a_1 as

	State s_1	State s_2	State s_3
iteration 0	0.0	0.0	0.0
iteration 1	0.0	1.0	0.0
iteration 2	0.1	1.0	0.0
iteration 3	0.1	1.0	0.0

Table 1.1: Values for the states in the sample environment for the steps of value iteration with a discount factor of 0.1.

$$\begin{aligned} Q(s_1, a_1) &= R(s_1, a_1) + \gamma \sum_{s'} T(s_1, a_1, s') V^*(s') \\ &= 0.0 + 0.1((0 \cdot V^*(s_1)) + (1 \cdot V^*(s_2)) + (0 \cdot V^*(s_3))) \\ &= 0.1(V^*(s_2)). \end{aligned}$$

For this example, the values of all states were initialized to 0. These values are shown as iteration 0 in Table 1.1. Based on these initial values, the calculated value of taking action a_1 from state s_1 is 0. The same calculation can be performed for action a_2. Again, since there is no stored value for s_1, it is initialized to 0, making the value of taking action a_2 from state s_1 also 0. The max over these two actions is 0, and thus, this number is stored as the current value for s_1 for iteration 1 of value iteration.

Now, the agent would update its value for state s_2 using the Bellman equation. The calculations for action a_1 would be

$$\begin{aligned} Q(s_2, a_1) &= R(s_2, a_1) + \gamma \sum_{s'} T(s_2, a_1, s') V^*(s') \\ &= 0.5 + 0.1((1 \cdot V^*(s_1)) + (0 \cdot V^*(s_2)) + (0 \cdot V^*(s_3))) \\ &= 0.5 + 0.1(V^*(s_1)) \\ &= 0.5 + 0.1(0) \\ &= 0.5. \end{aligned}$$

The calculations for action a_2 would be

$$\begin{aligned} Q(s_2, a_2) &= R(s_2, a_2) + \gamma \sum_{s'} T(s_2, a_2, s') V^*(s') \\ &= 1.0 + 0.1((0 \cdot V^*(s_1) + (0 \cdot V^*(s_2)) + (1 \cdot V^*(s_3)) \\ &= 1.0 + 0.1(V^*(s_3)) \\ &= 1.0 + 0.1(0) \\ &= 1.0. \end{aligned}$$

With the state s_3 initialized to 0, the value of taking action a_2 is greater than that of taking action a_1 from state s_2. So, the value for state s_2 is 1.0. The agent now goes on to compute the value of state s_3. However, since it is a terminal state, no actions can be taken and its value will remain 0. Since the values of some of the states just changed significantly, the agent once again computes the value of each of the states. Using the techniques shown previously, the value of state s_1 changes to 0.1 in iteration 2 based on the value of action a_1. The value of state s_2 is then re-evaluated to account for the new value of state s_1. Since the value of action a_2 is still greater than the value of action a_1 and the agent is solving for the maximum value, the value of state s_2 does not change. One more round of these calculations is performed until the agent sees no change in the value of any of the states. At that point, the agent determines that the best policy, based on the current transition function, reward function, and discount factor is to take action a_1 from state s_1 and action a_2 from state s_2.

As mentioned earlier, the agent chooses its actions based on a temporal weighting of its expected reward in future states by the discount function. When acting in the environment modeled by the MDP in Figure 1.3, if the discount factor were very small, an agent in state s_2 would choose action a_2, but if it were larger, it would choose action a_1 because, over time, it would achieve a larger cumulative reward. For instance, if the discount factor was 0.99, the agent would choose to take action a_1 instead of action a_2 because the cycle of s_1, s_2, s_1, s_2, s_1, etc. would

	State s_1	State s_2	State s_3
iteration 0	0.0	0.0	0.0
iteration 1	0.0	1.0	0.0
iteration 2	0.99	1.0	0.0
iteration 3	0.1	1.4801	0.0
iteration 4	1.4652	1.4801	0.0
iteration 5	1.4652	1.950548	0.0
iteration 6	1.93113955	1.950548	0.0
iteration 7	1.93113955	2.41182815	0.0
iteration 8	2.38770987	2.41182815	0.0
iteration 9	2.38770987	2.86383277	0.0
iteration 10	2.83519444	2.86383277	0.0

Table 1.2: Values for the states in the sample environment for the steps of value iteration with a discount factor of 0.99.

yield rewards of 0, 0.5, 0, 0.5, 0, etc. This result is due to changes that start at iteration 3, as detailed in Table 1.2. The calculations for state s_1 are the same, except for the change in discount function resulting in a value of 0.99 instead of 0.1. This larger value for state s_1 leads to the following calculations for action a_1 from state s_2:

$$\begin{aligned} Q(s_2, a_1) &= R(s_2, a_1) + \gamma \sum_{s'} T(s_2, a_1, s') V^*(s') \\ &= 0.5 + 0.99((1 \cdot V^*(s_1)) + (0 \cdot V^*(s_2)) + (0 \cdot V^*(s_3))) \\ &= 0.5 + 0.99(V^*(s_1)) \\ &= 0.5 + 0.99(0.99) \\ &= 1.4801. \end{aligned}$$

The calculations for action a_2 would be

$$\begin{aligned} Q(s_2, a_2) &= R(s_2, a_2) + \gamma \sum_{s'} T(s_2, a_2, s') V^*(s') \\ &= 1.0 + 0.99((0 \cdot V^*(s_1) + (0 \cdot V^*(s_2)) + (1 \cdot V^*(s_3)) \\ &= 1.0 + 0.99(V^*(s_3)) \\ &= 1.0 + 0.99(0) \\ &= 1.0. \end{aligned}$$

Now, action a_1 has a higher value than action a_2 because of the increased value of state s_1 and the value of state s_2. The table entry for state s_2 in iteration 3 is updated to 1.4801. For the next iteration, the calculations for action a_1 from state s_1 would be

$$\begin{aligned} Q(s_1, a_1) &= R(s_2, a_1) + \gamma \sum_{s'} T(s_2, a_1, s') V^*(s') \\ &= 0.0 + 0.99((1 \cdot V^*(s_1)) + (1 \cdot V^*(s_2)) + (0 \cdot V^*(s_3))) \\ &= 0.0 + 0.99(V^*(s_2)) \\ &= 0.0 + 0.99(1.48) \\ &= 1.4652. \end{aligned}$$

The calculations for action a_2 would be

$$\begin{aligned} Q(s_1, a_2) &= R(s_2, a_2) + \gamma \sum_{s'} T(s_2, a_2, s') V^*(s') \\ &= 0.0 + 0.99((1 \cdot V^*(s_1) + (0 \cdot V^*(s_2)) + (0 \cdot V^*(s_3)) \\ &= 0.0 + 0.99(V^*(s_1)) \\ &= 0.0 + 0.99(0.99) \\ &= 0.9801. \end{aligned}$$

The value for state s_1 would be updated and the calculations would continue for a user-specified number of steps or until it converges to a user-specified tolerance. Several more iterations of these calculations are shown in Table 1.2.

1.2.2 Learning the Model

If the model were completely known at the beginning of a task, there would be no point in having an agent learn. Programmers could solve the equations themselves and hard code the optimal policies. A reinforcement-learning agent becomes beneficial when the model of the environment is not fully known. In the Mars rover example, programmers know the behavior that they want from the robot. So, they are able to determine the reward function and discount factor, but they might not be able to fully model the robot's dynamics. For this reason, they could have the agent try to learn its transition function. The sample MDP of a robot-navigation task is shown in Figure 1.4. In this sample domain, the state consists of an x and y location shown as a grid. In an ideal world, the actions would include moving forward, back, left, and right a fixed amount of space. To learn its transition function, the agent would try each of its actions once in each state. After performing an action, the agent would store the resulting state as the expected next state for the previous state-action pair.

However, most environments are not this predictable. Unlike the simple environment mentioned in the previous section, the majority of the world is not deterministic; it is *stochastic*. When a robot moves in the world, there is a probability distribution over possible next states. Figure 1.5 shows a stochastic transition function of state s_{13} in a robot-navigation MDP. The transition probabilities are shown in parenthesis. For example, choosing action a_3 results in the state s_{14} with a probability of 0.5 and state s_{145} with a probability of 0.5.

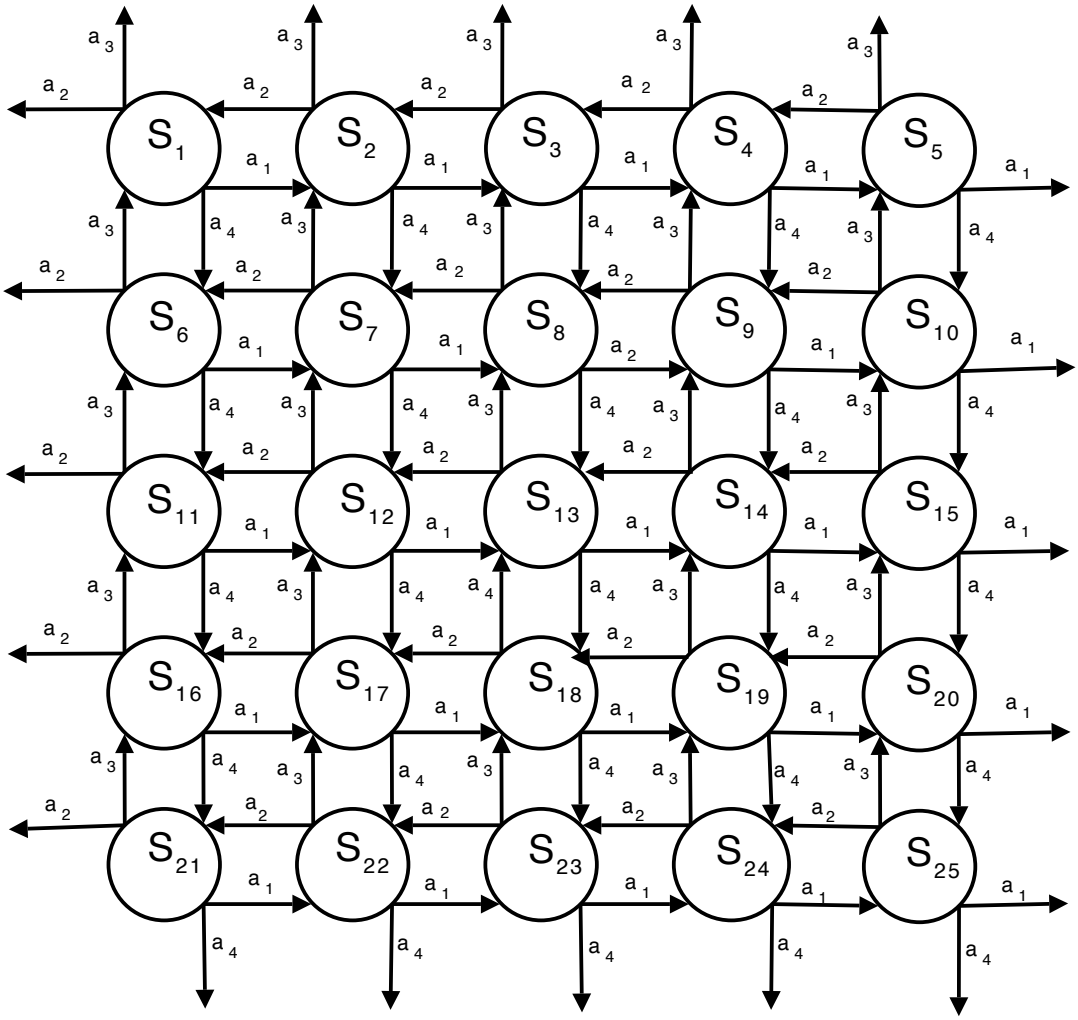


Figure 1.4: An MDP of a two-dimensional robot environment.

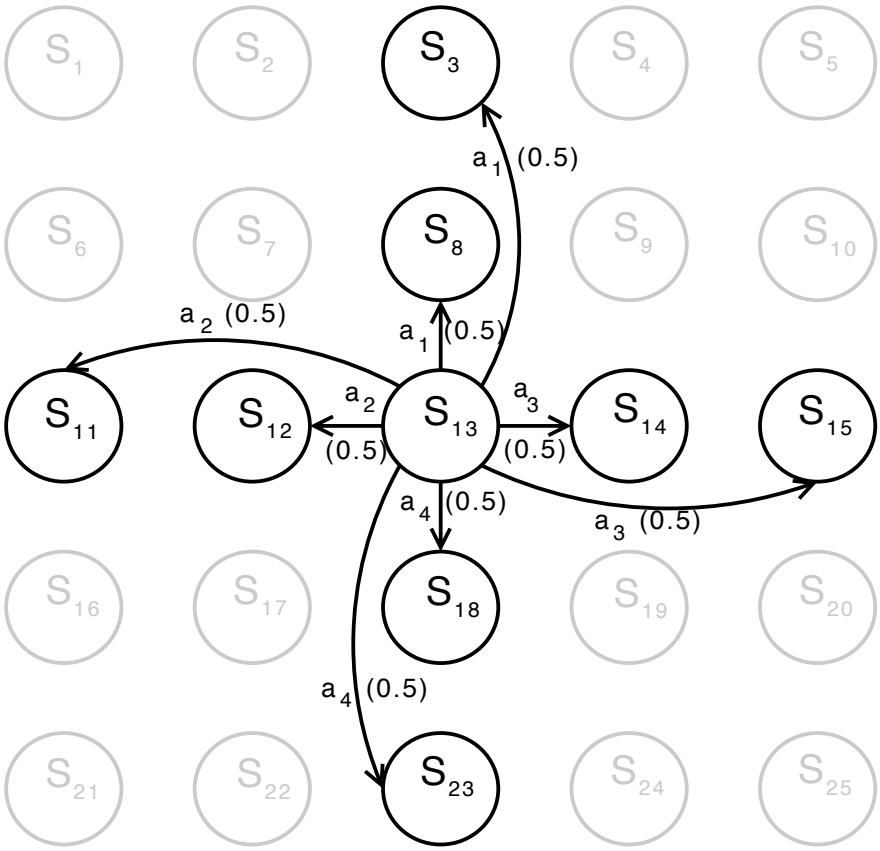


Figure 1.5: A stochastic MDP of a two-dimensional robot task. Only the transition function for state s_{13} is shown.

1.2.3 Exploration

To create an accurate model of an environment, an agent must acquire sufficient experience. Navigating stochastic environments requires more experience than deterministic environments because an agent needs to learn the probability distribution of all possible next states instead of learning a single next state.

The justification behind an agent taking an action can be in one of two forms: *exploration* or *exploitation*. An explorative action is an action that is taken for the purpose of gathering information. An exploitative action is an action taken with the intent to receive high reward. If an agent has a purely explorative policy, it will get low reward, as its actions will not necessarily take actions that result in high rewards. On the other hand, if an action has a purely exploitative policy, it may not fully explore its environment, leading to an improper or incomplete model. Without an accurate model, an agent is likely to derive a sub-optimal policy that achieves a low reward.

In order to receive near-optimal reward, a balance between these two types of actions must be achieved. This choice of which type of action an agent should perform is called the *exploration/exploitation dilemma* [Thrun, 1992]. By properly balancing these two types of actions, an agent is able to build a proper model and achieve optimal performance with a reasonable amount of experience.

For a reinforcement-learning agent to be beneficial in a real-life domain, the amount of exploration must be small. This restriction is especially true in robot domains where the exploration process is costly in both time and energy. Therefore, it is important to make the best possible use of the robot's limited opportunities for exploration without degrading the robot's performance.

1.2.4 Generalization

Even with a purely exploratory algorithm, some agents would take impractically long to fully model their environments. Assuming that every square foot of Mars was its own state, a rover would have to visit every square foot of Mars and perform each action once to have a simple deterministic model of the transition function. Obviously, this approach is not feasible. The time an agent spends exploring must be limited. One way to make the most of limited experience is to use *generalization* [Sutton, 1988].

In the context of reinforcement learning, generalization is the use of prior experience to infer information about unseen state-action pairs. For example, an agent might experience driving forward on a specific piece of pavement and assume from that experience that driving forward on any piece of pavement will have the same result. By making this assumption, the agent no longer has to drive forward on all of the locations covered in pavement, limiting the amount of exploration that is needed to learn the model.

This assumption, while speeding up the learning process, could lead to the agent choosing a poor policy. In the previous example, performing this generalization would cause problems if the environment included hills. Driving forward on pavement is different depending on whether the road segment is uphill or not. Without a proper model of the world, the agent would not know that it needed to use more power to travel uphill. Therefore, generalization needs to be accurate to be truly beneficial.

Value-Function Approximation

A common way of using generalization in a large state space is *value-function approximation*. This approach takes data from experience and uses it to approximate the entire value function [Sutton, 1988]. In one common form of this

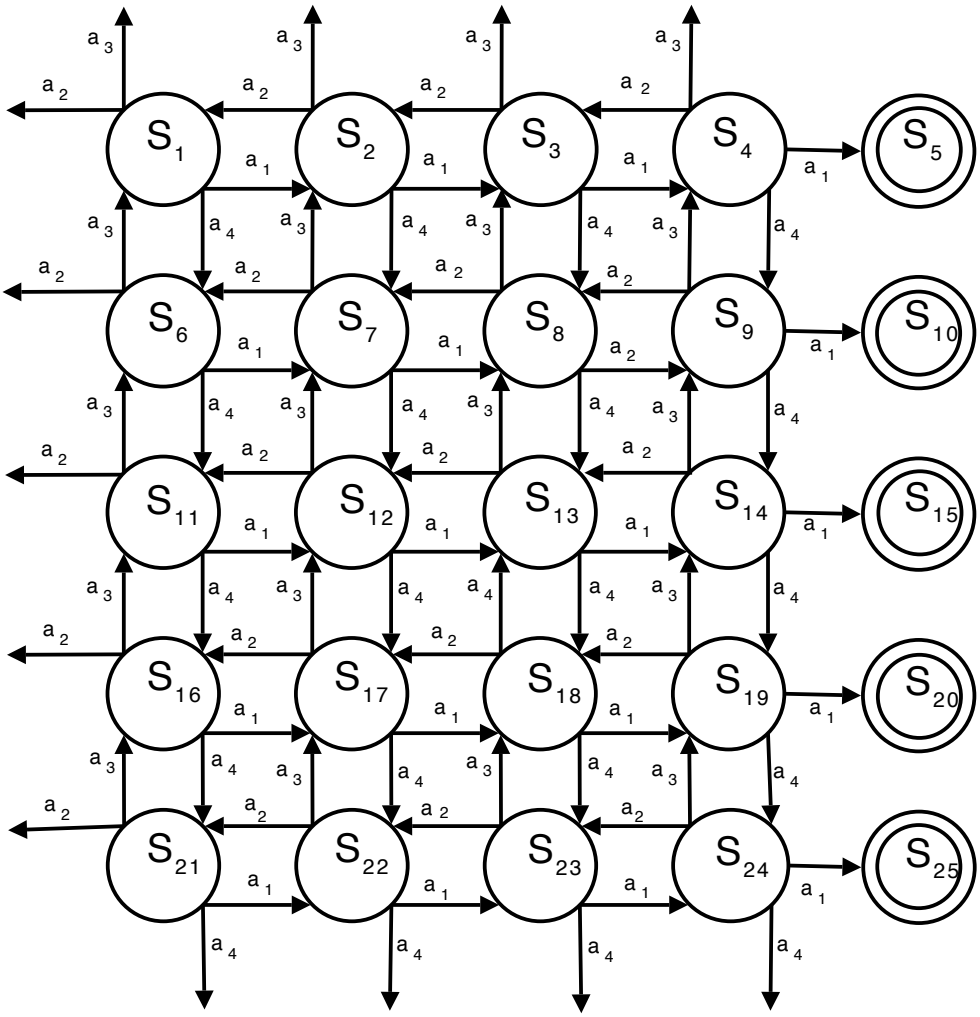


Figure 1.6: A Markov Decision Process (MDP) model for a robot navigation task where states s_5, s_{10}, s_{15}, s_{20}, and s_{25} are goal states.

technique, an agent infers a state-action pair's value by assuming that states in close proximity will have similar values [Gordon, 1995]. Referring back to the Mars rover example, if every square foot is its own state, then locations next to each other have similar values. So, if the agent went down a path, states next to that path would be assigned similar values. This approach, though, has been shown to be unstable often leading to the divergence of value iteration and/or poor performance [Moore, 1995].

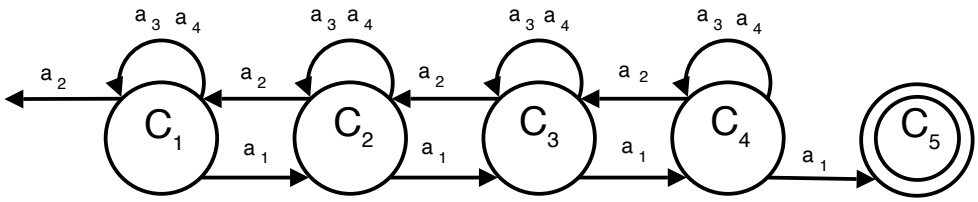


Figure 1.7: An aggregate model of the environment modeled in Figure 1.6. Each state represents a cluster of states with similar values in the original model.

State-Space Aggregation

Another way to perform generalization in model-based reinforcement-learning tasks is to reduce the number of states in the model using *state-space aggregation* [Dean and Givan, 1997]. With this approach, the agent creates a reduced model of the environment by *clustering* groups of states from the original representation and mapping them to a single state in the *aggregate* model. The aggregate model is then used for planning purposes. This mapping from original models to aggregate models is called *abstraction* and can be performed based on similar transition function, reward function, and optimal actions [Li et al., 2006; Boutilier and Dearden, 1994].

Figure 1.6 shows an MDP model of a robot navigation task where the goal is to enter one of the right most states. For this example, let us assume that any state-action pair that leads to a goal state has a reward of 1.0 and any other state-action pair has a reward of 0.01. An aggregate model of the MDP is shown in Figure 1.7 where abstraction was performed on states with similar state-action values. Since states s_5, s_{10}, s_{15}, s_{20}, and s_{25} have the same dynamics and rewards, they are mapped to the same state, c_5, in the aggregate model. Taking action a_1 in states s_4, s_9, s_{14}, s_{19}, and s_{24} would lead to a reward of 1.0, so these states are clustered and the state c_4 in the aggregate model. This process is continued until all states in the original model are mapped to a state in the aggregate model.

Once the aggregate model is determined, it is learned and solved as previously detailed in Section 1.2.1. If the aggregate model has the same number of states as

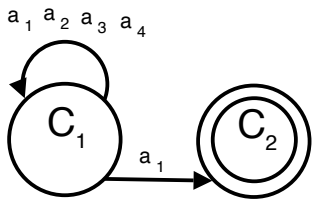


Figure 1.8: An aggregate model of the environment modeled in Figure 1.6. Each state represents a cluster of states with the same dynamics in the original model.

the original model, the resulting values for all the state-action pairs are identical.

The benefit of this approach is that the exploration necessary for the agent to learn the model is dependent on the number of states in the aggregate model instead of the number of states in the original. If the aggregate model is much smaller than the original model, the necessary exploration is greatly reduced.

There are difficulties with performing state-space aggregation. For instance, if the only goal state in the environment was state s_{25} as in Figure 1.9. In this environment, the determination of the aggregate model would have been more difficult. If the clustering of states was based on state-action values, no states would be clustered and the aggregate model would be the same size as the original model. On the other hand, if the clustering were based on which states had similar dynamics models, then state s_{25} would be one cluster and all states would be another, as shown in Figure 1.8. Another possible solution would be to cluster states based on the maximum possible value for each state. States would then be clustered based on their proximity to the goal as shown in Figure 1.10. The difficulty with this method, though, is that information about the state-action pairs are lost. For instance, the maximum value for states s_{20} and s_{24} are $+1.0$ and, therefore, are both represented as state c_7 in the aggregate model. However, once the aggregate is created, the information about how to achieve this reward is lost. In the original model, the agent had to take action a_4 from state s_{20} or action a_1 in state s_{24}. In contrast, if the agent is in state c_7 in the aggregate model, an agent has no way to know whether action a_1 or a_4 will lead to the goal.

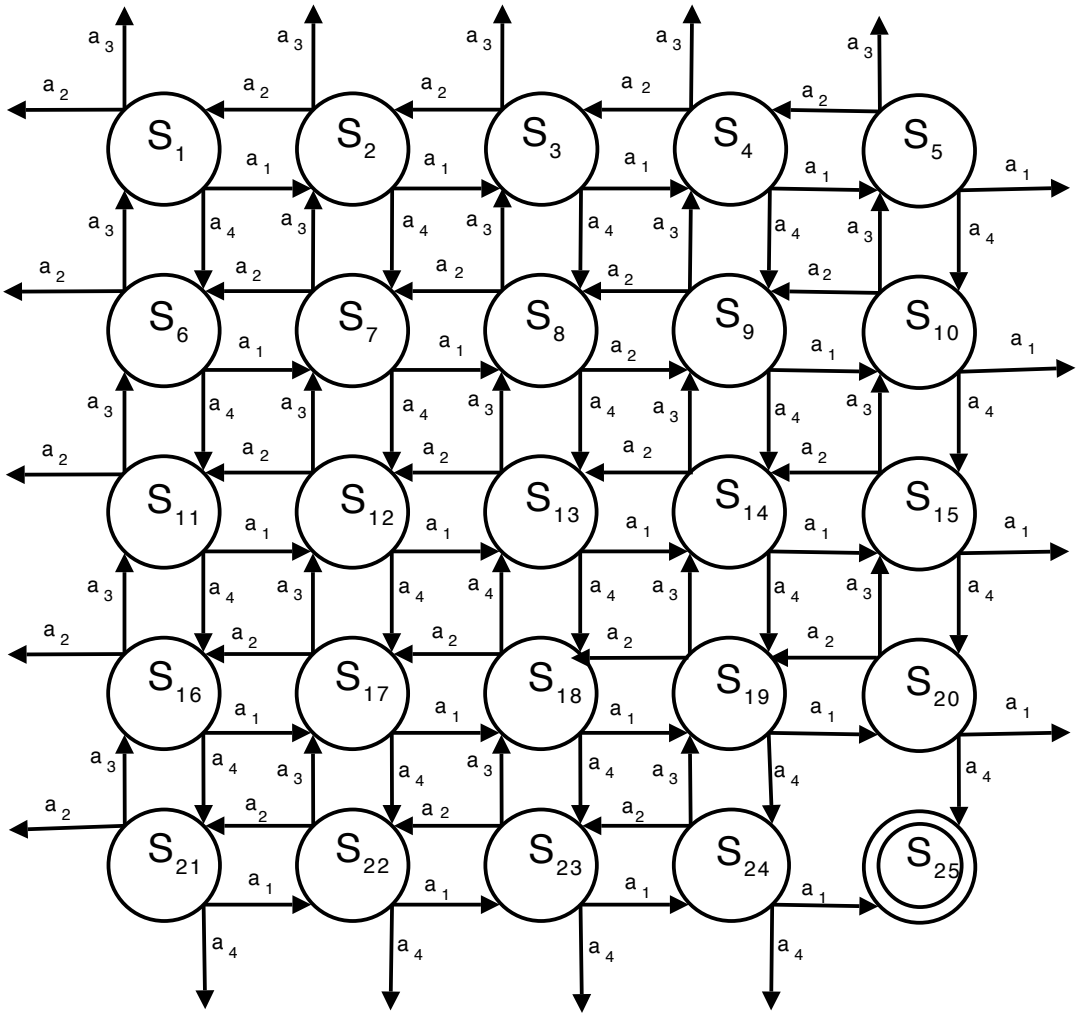


Figure 1.9: A Markov Decision Process (MDP) model for a robot navigation task where states s_{25} is a goal state.

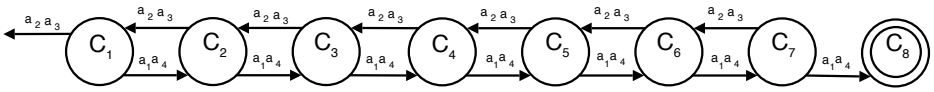


Figure 1.10: An aggregate model of the environment modeled in Figure 1.9. Each state represents a cluster of states equidistant to the goal in the original model. Each state represents a cluster of states with similar values in the original model.

1.3 Relocatable Action Models

To avoid this difficulty while performing generalization of the model, Sherstov and Stone [2005] created the relocatable action model (RAM) representation as a formalism for describing a decomposition of MDPs. This new formalism decomposes the transition function into three separate functions—a *type function*, a *relocatable action model*, and a *next-state function*. The type function, κ, groups states with similar dynamics into *clusters*. The number of clusters, C, is between one, where all states are classified as similar, and one per state where all states are classified as a different cluster. The transitions for each cluster are captured in the relocatable action model, t. This model maps each transition called an *outcome*, o.

For outcomes to be used for planning, the agent must be able to calculate possible next states from a state and an outcome. The function for performing this mapping is the next-state function, η. In the robot-navigation environment, where the state space consists of locations, the next-state function can be expressed using simple mathematical transformations such as subtraction and addition.

Using a RAM MDP is more beneficial than state-space aggregation because in addition to sharing experience among similar states, it also keeps all information about the original model. This information allows for the agent to properly model its dynamics. In fact, a RAM MDP can be used to perform traditional state-space aggregation. However, the inverse is not true (see Appendix A).

While using a RAM MDP allows for a better model, it does need more background knowledge for determining the clustering function, relocatable action model, and the next-state function. This dissertation will discuss how to learn these functions in the robot-navigation environment with little additional information.

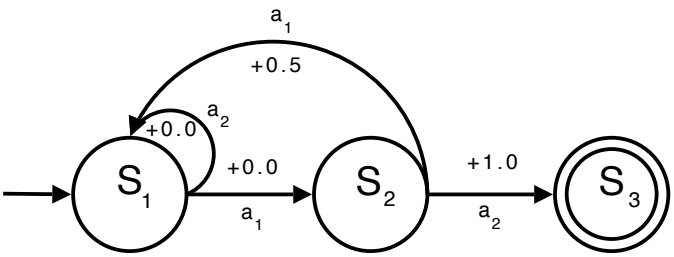


Figure 1.11: The simple Markov Decision Process (MDP) from Section 1.2.

1.3.1 Solving the RAM MDP

Like a traditional MDP, a RAM MDP can be solved using value iteration. However, the RAM formalism uses different functions and variables, which need to be converted to be used in the Bellman equation. Equation 1.2 shows the modified Bellman equation used for solving a RAM MDP. Changes in the formula include the reward function, which is dependent on the cluster-action pair instead of the state-action pair, and the transition function, which is a combination of κ, t, and η. It also uses outcomes instead of the next state:

$$V^*(s) = \max_a Q(s,a) = \max_a \{R(\kappa(s),a) + \gamma \sum_o (t(\kappa(s),a,o)V^*(\eta(s,o)))\}. \quad (1.2)$$

Figure 1.11 is the simple MDP introduced in Section 1.2. Assuming that $s_1 \in c_1$, $s_2 \in c_2$, and $s_3 \in c_3$, the state values are the same as the traditional formalism. For instance, an agent in the simple MDP environment with a discount factor of 0.1 would calculate the value of state s_1 by calculating the value of action a_1 as

$$\begin{aligned} Q(s_1,a_1) &= R(\kappa(s_1),a_1) + \gamma \sum_o t(\kappa(s_1),a_1,o)V^*(\eta(s_1,o)) \\ &= R(c_1,a_1) + \gamma \sum_o t(c_1,a_1,o)V^*(\eta(s_1,o)). \end{aligned}$$

In this one-dimensional model, an outcome would be the change in the x

dimension. Therefore, the outcome o_1 for taking action a_1 from state s_1 would be a change of $+1$ in the x dimension. State s_1 has a value of 1 in the x dimension. Therefore, $\eta(s_1, o_1) = s_2$ and the value of taking action a_1 from state s_1 is

$$\begin{aligned} Q(s_1, a_1) &= R(c_1, a_1) + \gamma t(c_1, a_1, o_1) V^*(\eta(s_1, o_1)) \\ &= R(c_1, a_1) + \gamma t(c_1, a_1, o_1) V^*(s_2) \\ &= 0.0 + 0.1 \cdot 1 \cdot V^*(s_2) \\ &= 0.1(V^*(s_2)). \end{aligned}$$

Similar to the example in the previous chapter, the values for the remaining state-action pairs would be calculated using value iteration and the Bellman equation having the same results as a traditional MDP.

The process of learning the RAM MDP is very similar to learning a traditional MDP. It involves a balance of explorative and exploitative actions to properly model the environment and perform the given task. The benefits of the RAM MDP are introduced when trying to reduce the amount of exploration needed. By learning the agent's transitions by cluster-action pairs, instead of state-action pairs, the amount of exploration necessary to learn an environment is dependent on trying every action in every cluster, instead of trying every action in every state. The greater the state to cluster ratio, the more beneficial the RAM MDP.

To further illustrate the advantages of using a RAM MDP, Figure 1.12 shows the same two dimensional robot-navigation environment as Figure 1.4, but with the addition of the type function. The state's type, or cluster, is indicated by color. Each cluster has its own set of outcomes per action. For the red cluster, action a_1 changes the x dimension by $+1$, action a_2 changes the x dimension by -1, action a_3 changes the y dimension by $+1$, and action a_4 changes the y dimension by -1. The green cluster's outcomes are reversed.

If the two clusters were known, an agent would take a minimum of eight steps

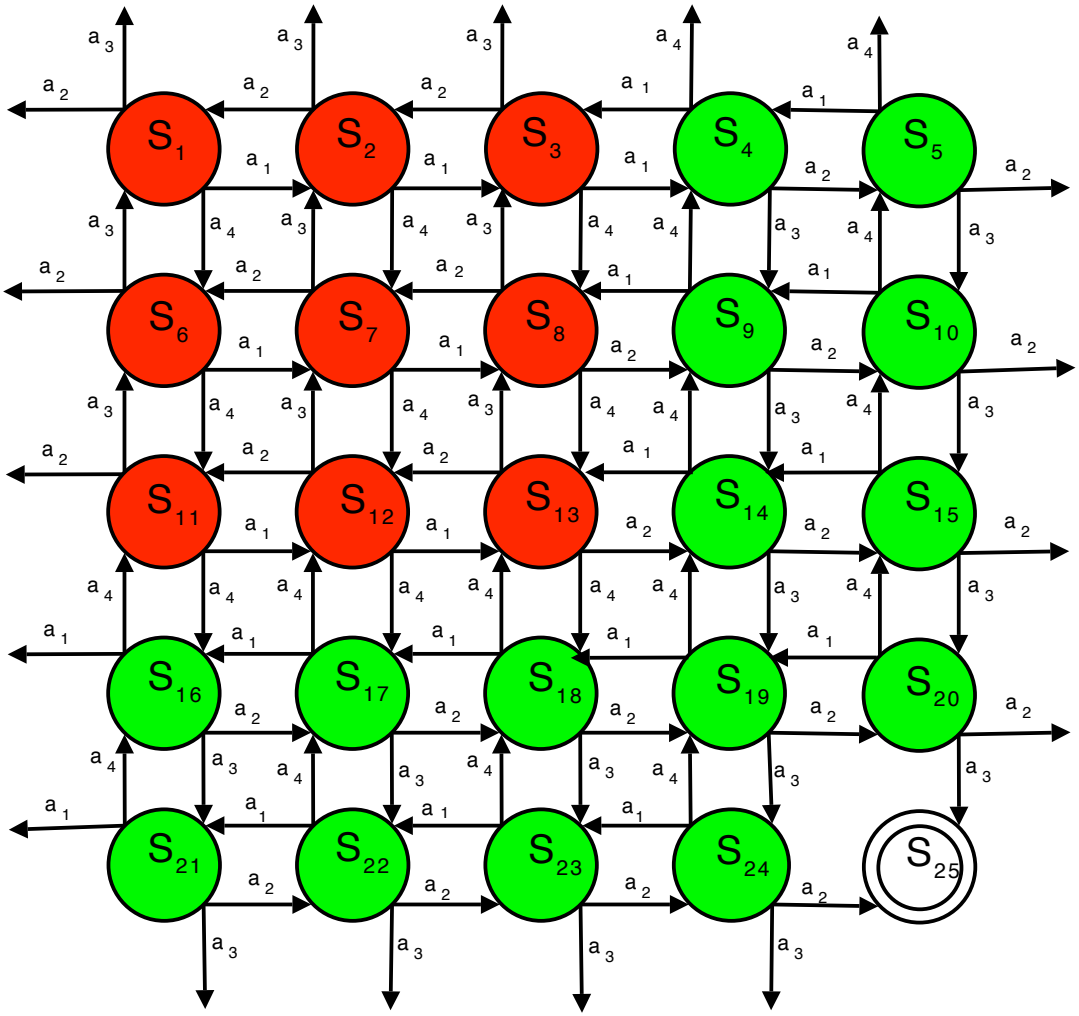


Figure 1.12: A RAM MDP model of a two-dimensional robot-navigation environment with two clusters.

(four actions for each of the two clusters) to learn all of the transitions in the environment. This result is in contrast to the 100 steps (four actions for each of the 25 states) it would take an agent to learn the transition function for the traditional MDP.

One detriment of the RAM MDP model is the increased number of calculations that are introduced when solving for the transition function. However, in robot-navigation domains where actions can take up to several seconds to perform, the additional cost is insignificant.

1.3.2 Determining Accurate Clusters

As mentioned earlier, to exploit the benefits of the RAM formalism, the state space needs to be clustered. Determining how to group states is a difficult problem. This dissertation addresses the issue of state-space clustering by making the assertion that in robot-navigation domains there are often perceivable clues available to the agent through sensors or other inputs that can be exploited to determine beneficial clustering.

This improvement in learning time does not only hold true when the clusters are apparent. Even in the absence of additional perceptual data, an agent can learn action-independent state clusterings using reward data and still improve learning time.

1.4 Thesis Statement

The central thesis of this dissertation is that state-space clustering determined by perceptual information can be exploited to provide more efficient exploration when performing robot-navigation tasks. By utilizing additional information about the world, such as visual information about surface types or sensor information about wall locations, the agent is able to generalize between states and

minimize exploration costs by focusing on learning at the level of clusters. By performing this more efficient exploration, the amount of experience needed to learn the dynamics model is reduced from being dependent on the number of states to the number of clusters. In addition, it is possible and still beneficial to learn the clusterings from possibly ambiguous percepts.

1.5 Outline

The remainder of this document demonstrates how using perceptual information for generalization allows for more efficient exploration. Chapter 2 further examines research that has been done in areas of machine learning in robotic tasks and generalization in reinforcement learning.

Chapter 3 introduces an algorithm for learning and solving RAM MDP models in robotic environments when the state-space clustering is given. One technique for learning the state-space clustering in a domain with limited inputs is shown in Chapter 4. Chapter 5 expands on the learning of state-space clustering by introducing an algorithm that learns accurate clusters of the state space when multiple contradictory inputs are given. Each of these chapters demonstrate positive empirical results from physical robot domains.

Finally, the dissertation concludes with a summary of the work detailed in previous chapters. Also discussed are the ways in which the techniques detailed in this dissertation could be applied to domains not explored here.

Chapter 2
Related Work

2.1 Machine Learning and Robotics

This section gives an overview of select machine-learning algorithms that have been implemented to learn dynamics or aid navigation in the robot domain. Each of these approaches, which can be classified into three categories: unsupervised, supervised, and reinforcement learning, and the robot tasks that they solved, are described in Sections 2.1.1, 2.1.2, and 2.1.3, respectively.

2.1.1 Unsupervised Learning

Unsupervised learning is a branch of machine learning in which an agent learns about an environment without any feedback about the correct output [Russell and Norvig, 2003]. An unsupervised learning problem called *clustering* is the grouping of input into classes, or *clusters* [Alpaydin, 2004]. A simple example application is when a large group of people is broken up by age or gender.

Clustering was used to aid in the autonomous navigation of the Carnegie Mellon NAVLAB vehicle [Crisman and Thorpe, 1991]. The goal of their UNSCARF algorithm was to identify road segments from red, green, and blue values of pixels in an image taken by a camera mounted on the robot. Class boundaries were then calculated based on the determined clusters and compared with "traditional" shapes of roads. The group of pixels with the shape best fitting that of a road was then determined to be a road. This process showed great benefit in situations

where shadows were cast on the road.

Stronger and Stone [2004] proposed an unsupervised learning algorithm called SCASM that calibrated the action and sensor models of a Sony Aibo. This calibration was performed by having the robot walk forward and backward at different velocities while facing a multicolor pole, or beacon. Using camera data and information about the beacon, SCASM was able to determine the action model in terms of the sensor model and vice versa.

2.1.2 Supervised Learning

Unsupervised learning can be applied to simple clustering tasks; however, it doesn't scale well to more difficult problems in which greater precision or reliability is needed [Alpaydin, 2004]. For problems such as classification or regression, a supervised-learning agent is provided with a set of inputs and outputs on which it is trained to learn the mapping from one to the other. After the training process is complete, the agent can then take in additional input examples and predict the corresponding outputs.

Artificial Neural Networks

One way to implement a supervised learner is by creating an artificial neural network. This data structure, modeled after the human brain, consists of many densely interconnected simple units. Each of these units takes a number of real-valued inputs and produces a real valued output [Mitchell, 1997]. To capture greater complexity, the outputs of these units can be fed as inputs puts of other units. This data structure is often robust to errors in training data and can handle cases in which many inputs work together to produce the correct output.

A multilayer neural network was used to train sensors on the NAVLAB vehicle to distinguish paved road segments from non-paved surfaces [Pomerleau, 1989].

The ALVINN algorithm consisted of forty rounds of training of the neural network on 1200 simulated road images and ground truth steering directions. Then, as the robot drove, the neural network was fed input from three sources: a video camera, a laser range finder, and a feedback unit based on previous classifications. ALVINN returned the amount and direction that the steering wheel should turn, which was then fed to NAVLAB. Using ALVINN, the NAVLAB robot was able to drive 400 meters at 0.5 meters per second along a wooded path.

More recently, Rasmussen [2002] drove an Experimental Unmanned Vehicle (XUV) along a rural road using a neural network to determine whether a pixel in an image was to be classified as road. The network was trained on several values of the visual feature space as input including height, smoothness, texture, and color, where the outputs were hand-marked classifications. No implementation on the robot was presented; however, the results of the classification task showed good performance.

Bentivegna and Atkeson [2001] showed another use for supervised learning by combining observation of humans with neural networks to teach a humanoid robot to play air hockey against humans. Since air hockey is a complex task for an agent to learn, the developers broke the task into "primitive" trajectories separated by collisions of the puck and the paddle. The primitives that the agent learned were *left hit, straight hit, right hit, block, prepare*, and *multi-shot*. Before the tasks were learned, the agent watched a human play multiple games. By observing human play, the agent was shown when to start each action and where the puck should end up. The inputs to the neural network were the velocity and position of the puck before and after it was hit. The outputs were the paddle's velocity and location at the time of collision with the puck. Once the primitive training was completed, the robot was able to play a game against a human by determining what primitives to perform based on the action it observed the human perform when it was in a similar state and then executing the primitives using the neural

network.

Self-supervised Learning

One difficulty in developing a supervised-learning agent is the hand labeling of training data. In order for an agent to be robust, it must see a lot of training data, which means that a developer has to label a lot of input. This process can be time consuming. One solution to this problem is self-supervised learning. With self-supervised learning, one sensor on the robot is able to determine the output corresponding to another sensor's input.

Dahlkamp et al. [2006] developed an agent for driving a Volkswagen Touareg across a desert using two different sensors for road detection. To train the agent, a laser range finder was used to identify which parts of the nearby area (22 meters) were considered drivable. This information was then sent to the agent as output to be matched with the video camera image of that area as input. Once the learner was trained, the agent was able to identify where else in the image was passable. Since the video range was 70 meters, the learned predictions were more useful than the range sensor alone. Since the agent's classifications weren't as reliable as the range finder, the authors used the learner's output as an obstacle pre-warning system. If the learner thought that part of the incoming image was not drivable, it would tell the robot to slow down for the laser range finder to make a determination about that area.

2.1.3 Reinforcement Learning

Self-supervised learning addresses the common situation of a developer being unable or unwilling to specify the proper output or classification of a given input. However, it assumes that a robot is able to use multiple sensors to provide both input and output. Frequently, ground truth cannot be determined by the agent alone. Reinforcement learning addresses this issue by using information from the

environment to provide the agent with feedback after every action it performs. Using this technique, an agent is able to evaluate its performance using only feedback from the environment.

Model Free

Most reinforcement-learning algorithms can be categorized as either *model-free* or *model-based* algorithms. In model-free, or direct, reinforcement learning, an agent learns the long-term value of taking each action in each state [Sutton and Barto, 1998]. By comparing the value of taking each action from the agent's current state, the agent can determine the best policy.

Watkins and Dayan [1992] revolutionized reinforcement learning with their Q-learning algorithm, which they proved converges to the optimal policy in the limit. This algorithm was based on the Bellman equation mentioned in Section 1.2. Watkins and Dayan [1992]'s addition to this work was recognizing the importance of a Q-value, the definition of which is

$$Q^{\pi^*}(s,a) = R(s,a) + \gamma \sum_a T(s,a,s')V^*(s'). \tag{2.1}$$

By keeping a lookup table of Q-values, an agent quickly determines which action to take. This lookup table is updated after every step using Equation 2.2 where α is the learning rate, which specifies how to weight new information. Q-learning allows for minimal per-step planning time and guarantees convergence. However, this guarantee only holds in the limit. The form of the update is

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t) \right]. \tag{2.2}$$

Mahadevan and Connell [1992] implemented two versions of Q-learning on an OBELIX robot that navigated through a room and pushed a cardboard box from its initial position to the other side of the room. To implement the Q-learning algorithm in a robot domain, the states were grouped by similarity to limit the

number of times that each individual state had to be visited. This requirement was met using weighted Hamming Distance and statistical clustering to relate the states to each other. To improve performance further, the authors broke the task into three smaller subtasks and hard coded when the agent should be performing each task. These subtasks were to find the box, push the box, and recover from a stall such as being stuck in a corner. Each "learning run" of this algorithm took two hours and included 2000 steps of the robot.

Model-Based Methods

Another approach to reinforcement learning is to use the feedback from the environment to learn a model of the world. This approach is also based on the Bellman equation. Referring back to Equation 1.1, a model of the environment can be expressed in the form of the reward function $R(s, a)$ and the transition function $T(s, a, s')$. By creating a full model of the environment, the agent can learn the structure underlying the value function instead of just estimating it from experience. This approach gives the agent a more accurate prediction of the value function with less data.

Modeling From Observation

In the robot domain, it is often difficult to explore thoroughly enough to get a full model of the critical parts of the world. Instead, like humans, some algorithms take an apprenticeship, or demonstration, approach. With this approach, the agent observes a human, usually an expert, acting in the world and is able to model the world before acting in it. This view is different from supervised learning because instead of a human labeling a bunch of input, the human acts in the environment and the agent learns from that experience.

Bagnell and Schneider [2001] used this apprenticeship technique to command the CMU Yamaha R50 helicopter to hover in place. For the agent to learn the

dynamics of the helicopter, data was collected from human pilot tele-operation. Once the model was constructed using locally weighted Bayesian regression, the policy space was searched using the Amoeba algorithm, which involves creating a geometric figure of points, where there was one point for each feature, and moving and reshaping that figure to search the policy space.

Ng et al. [2004] also explored controlling a Yamaha R50 helicopter by processing human control data to model the environment. Once the model was constructed using locally weighted linear regression, the agent took actions in the world and used feedback from the environment to improve its policy using the gradient ascent-like algorithm PEGASUS. With this algorithm, the agent learned to fly several patterns that were identified as difficult by organizers of a human remote-control helicopter competition.

Atkeson and Schaal [1997] used demonstration to teach a SARCOS robot arm to perform a pendulum swing up task. In this instance, the agent observed a human perform the task of swinging a pole from a downward position to keeping it balanced straight up vertically. The agent then tried to mimic the behavior of the human. However, due to differences in the poles used by the human and the robot, direct emulation of the human was not enough to perform the task properly. Instead, while performing the reenactment, the agent learned a model of the dynamics from the reward that it received. From this model, the agent was able to improve on its behavior and successfully perform the task after a small number of trials.

Modeling From Exploration

Learning a model through observation is an efficient way to model a subset of the environment. Unfortunately, a robot cannot always observe a human performing the desired task such as when a robot is trying to learn about its sensor or action models. In these situations, a robot can use feedback from the environment to

learn a model of the world from experience.

As mentioned in Section 1.1, the exploration-exploitation dilemma is frequently encountered when modeling an environment through experience. In Q-learning research, this tradeoff is often managed by the agent taking a random action a small percentage of the time. However, there are more explicit methods for addressing the problem.

Kearns and Singh [1998] introduced an algorithm with a much smaller learning-time upper bound than Q-learning. The E^3 algorithm, which stands for Explicit Explore or Exploit, divides all of the states into two sets: known states that the agent has visited sufficiently, and unknown states. Then, the agent chooses an action that will either lead to exploitation of the known set or exploration of the unknown set with high probability if possible. By choosing actions based on this criterion, the agent balances the exploration-exploitation tradeoff and converges to near optimal policies faster than other methods.

A closely related algorithm to the E^3 algorithm is R_{max} [Brafman and Tennenholtz, 2002], which provides an implicit way of addressing the tradeoff. Instead of having the agent calculate the value of learning more versus doing what it knows will pay off, R_{max} initializes the rewards of all states to the maximum reward. This reward does not get updated to its observed value until the agent visits the state a target number of times. This high expected value for unseen states drives the agent to explore unseen states if the value of doing so exceeds the value of known states and leads to more efficient exploration.

2.2 Generalization in Reinforcement Learning

The main bottleneck when attacking machine-learning tasks in robot-navigation domains is balancing the tradeoff between exploration and exploitation. Many researchers have worked on knowledge transfer between environments and other

ways to decrease sample-complexity demands [Thrun and O'Sullivan, 1996; Diuk et al., 2006]. Other avenues of research include generalization over states and value functions. This section will highlight some of the more common techniques being studied.

2.2.1 Value-Function Approximation

As shown in Section 1.2.4, a common form of generalization used in reinforcement learning is value-function approximation. In model-free reinforcement learning, this technique often involves using a parameterized function approximators to generalize the value function between similar states and actions [Sutton, 1996]. A well-known example of this approach working successfully is TD-Gammon [Tesauro, 1995]. This system used neural networks to learn the values of different parameters of a general non-linear function approximator to play backgammon.

Another algorithm called fitted Q iteration [Ernst et al., 2005] uses a regression algorithm to fit a value-function approximator to training data. Once enough experience is gathered, the learned function approximator is used to provide approximation of values in the entire state-action space. In tree-based implementations of this algorithm, regression trees are used to determine similarities in the state space from values of state-action pairs experienced. Values from unseen state-action pairs are then predicted using approximations from experience collected in the same area of the tree.

Guestrin et al. [2003] approximated value functions not over the each state, but over each state variable. They assumed that each state variable had a value subfunction and that the sum of the values of these subfunctions could accurately approximate the actual value function. In a robot-navigation domain, this approach would require approximating a value subfunction to a robot's x, y and θ values independently and then summing the subvalues to approximate the value of the state as a whole.

2.2.2 Model Reduction

Another approach to generalization in reinforcement learning is to try to condense the size of the model with which the agent is working. This approach is most commonly seen in the planning literature. Bean et al. [1987] minimize their model by performing state abstraction to create a higher level model of the environment. This smaller model is then used to find shortest paths between higher level nodes. Once the high level path is found, it is propagated to the original model and the learner finds the remaining shortest paths.

This idea of state abstraction can also be combined with value approximation to infer similar values for similar states. As mentioned in Section A.1, a problem with abstraction is that important information can be lost. To limit this problem, Moore [1991] introduced Variable Resolution Dynamic Programming (VRDP) which, as the name suggests, allows for multiple levels of abstraction during the planning process. This algorithm generalizes values across regions deemed to be of low importance.

Another way of limiting information loss in state clustering is soft-state aggregation [Singh et al., 1995]. In contrast to regular state-space aggregation, when using soft-state aggregation, a state can belong to several clusters with a certain probability. A state's approximated value, then, is a weighted sum of all of the values of the clusters to which it could possibly belong.

Even with soft state aggregation, though, the underlying state variables are lost, can lead to sub-optimal performance. One way to ensure that the generalization is only performed across uninformative features is to factor the state space. A factored-state Markov Decision Process (MDP) [Dean and Givan, 1997] is similar to a traditional MDP, but instead of having transitions be from state to state, they are modeled at the level of changes in the values of state variables. By learning the transition function between state variables, the model can be minimized by aggregating over state variables that do not affect a state's transitions.

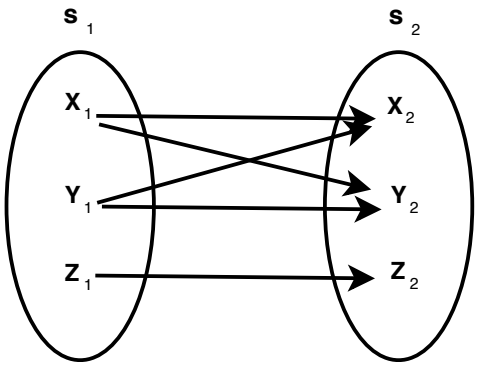


Figure 2.1: A factored-state Markov Decision Process (MDP).

An example in the robot-navigation domain would be if a state space consisted not only of the robot's location, but also the angle of its arms. This example is demonstrated in Figure 2.1 where X and Y are the robot's location and Z is the position of the robot's arm. The go forward action would not affect the angle of its arms, so these state variables would not need to be considered when calculating the result of a go forward action. Adaptations have been made to well-known algorithms such as R_{max} and E^3 to benefit from from the factored-state MDP representation [Guestrin et al., 2002; Kearns and Koller, 1999].

2.2.3 Generalization in the Transition Function

While factored-state MDPs look at transitions to decompose the state space, the transitions learned are still direct mappings from one state to another. In a robot-navigation domain, where the state variables like x, y, and θ have a large set of possible values, trying to explore all values of these variables is not feasible. For this reason, generalization in the transition function is necessary. One approach to solving this problem is to use topological theory in relational reinforcement learning [Lane and Wilson, 2005]. By building in relations about cardinal directions, the agent can determine its location with respect to the goal, such as north and west. This implementation, however, needs predictable actions and cannot handle obstacles.

Other research has addressed generalized transition functions by making the reasonable assumption that "nearby" state-actions can help in predicting state-action dynamics. This constraint limits exploration by assuming that states that have a small Euclidean distance have similar dynamics models. Algorithms like Metric E^3 and Fitted R_{max} use this assumption to speed up learning [Kakade et al., 2003; Jong and Stone, 2007]. While these techniques limit the amount of exploration that an agent needs to perform, the way in which they determine similarities in the state space can lead to errors in the model.

In the remainder of this dissertation, I will discuss techniques for modeling environments as relocatable action model (RAM) Markov decision processes (MDP) and performing state-space clustering on perceptual inputs to allow for more efficient exploration in the robot-navigation domain.

Chapter 3
Exploiting Known Perceptual Clusters

3.1 Introduction

As mentioned in Section 1.3, the relocatable action model (RAM) Markov decision process (MDP) enables more efficient exploration than a traditional MDP by generalizing experience across similar states. This chapter will introduce the RAM-R_{max} algorithm, which models robot-navigation environments as RAM MDPs and uses perceptual information to cluster the state space. In two separate domains, this algorithm is shown to perform more efficient exploration than several state-of-the-art algorithms.

3.2 The RAM-R_{max} Algorithm

Algorithms that model the environment using a traditional MDP, such as the R_{max} algorithm, require an agent to try every action in every state at least once to determine the transition function of that state-action pair (see Section 2.1.3 for further details). This requirement is due to the necessity of learning the transition function, which maps state-action pairs directly to next states. There is no way to infer the result of taking action a_1 in state s_0 without visiting s_0, in the general setting.

To allow for generalization across states, the RAM MDP decomposes the traditional transition function into three parts—the clustering function, κ, the relocatable action model, t, and the next-state function, η. The RAM-R_{max}

algorithm is a modification of the R_{max} algorithm that uses the RAM MDP to model its environment. Once κ and η are known, the amount of exploration that the RAM-R_{max} algorithm needs to build a complete model of the environment is reduced from being dependent on the number of state-action pairs to being dependent on the number of cluster-action pairs. This section will explore how the RAM-R_{max} algorithm performs efficient exploration to learn outcomes and policies when κ and η are given.

3.2.1 Determining κ, the Clustering Function

In this chapter, we make the assumption that as a robot navigates through its environment, it has access to additional information, such as perceptual cues, that can be used to classify states into proper clusters. One type of classifier, called *terrain classification*, uses image-segmentation algorithms on images of the environment's terrain to determine which states are similar. For example, results of a particular image-segmentation algorithm called Edge Detection and Image SegmentatiON (EDISON) are shown in Figure 3.1 [Christoudias et al., 2002]. When a camera is placed above the environment, facing the surface, image segmentation can be used to determine surfaces that have different visual properties, such as color or texture. These visual properties often correspond to differences in surface properties. Based on informal experimentation, the assumption that a robot's dynamics change across surfaces with visually detectable is reasonable.

3.2.2 Learning Outcomes

In many environments, learning a relational transition, such as an outcome, can be difficult. However, in robot-navigation tasks, the state space is based on real numbers such as an agent's location (x, y) and orientation, θ. By assuming that the dimensions of the environment are isotropic and continuous, the distance

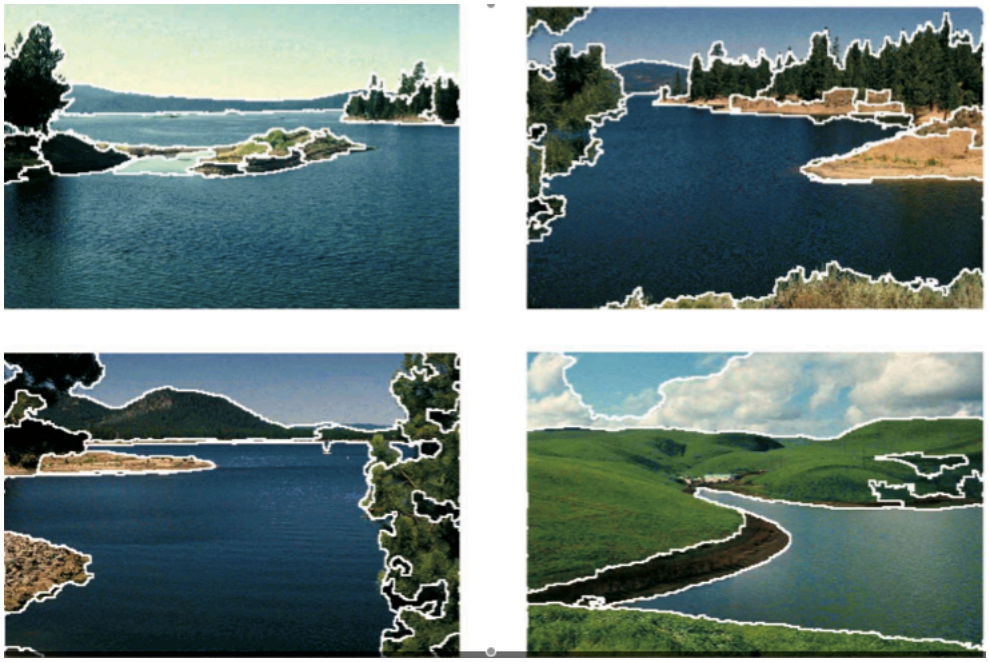


Figure 3.1: Results of the EDISON image segmentation algorithm on different landscape images [Christoudias et al., 2002]. The different "terrains" are outlined in white.

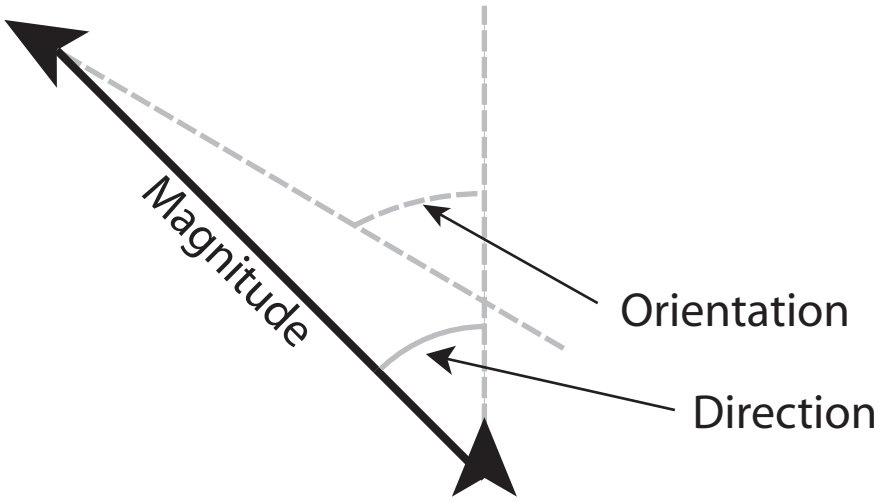


Figure 3.2: A sample outcome where the agent starts out at $x = 0$, $y = 0$, $\theta = 90$ and ends up at $x = -5.66$, $y = 5.66$, $\theta = 150$. The resulting outcome is *magnitude* $= 8$ points, *direction* $= 45$ degrees, *change in orientation* $= 60$ degrees.

between states can be reliably calculated.

For instance, Figure 3.2 shows a sample outcome where the starting position of the agent is $x = 0$, $y = 0$, $\theta = 90$. Since outcomes are calculated from the agent's starting coordinate frame, it is easiest to consider the agent's starting position to be the origin. The agent's next state is shown to be $x = -5.66$, $y = 5.66$, $\theta = 150$. To accurately capture the results of an action, outcomes are stored in terms of *magnitude*, *direction*, and *change in orientation*. Magnitude measures the absolute distance that the robot has traveled. The direction of the outcome is the direction of this displacement. In this example, the outcome has a magnitude of 8 points and a direction of 45 degrees. Finally, the change in orientation is the amount that the robot's orientation has rotated. In the example outcome shown, the agent went from an orientation of 90 degrees to an orientation of 150 degrees, making the change in orientation 60 degrees.

After each action, the outcome is calculated, stored, and tallied (as t_C) for the corresponding cluster-action pair. The maximum size of the outcome list is finite due to finite number of possible outcomes in a discrete environment. However, research has shown that outcomes can also be stored as a Gaussian distribution to keep the computation time from scaling with experience [Brunskill et al., 2008].

3.2.3 Determining η, the Next-State Function

Once the agent has learned its outcomes, it can use η to make predictions about possible next states. Since a RAM MDP maintains all of the state information, simple arithmetic such as matrix transformations can be used to calculate η using outcomes and state information (as shown Section 1.3.1).

Figure 3.3 is a graphical representation of the steps for calculating an agent's possible next states. Figure 3.3(a) shows a set of outcomes that the agent has experienced. Once again, the transformation has been made so that the agent's starting location is assumed to be $x = 0$, $y = 0$, $\theta = 90$. The diagonal dashed

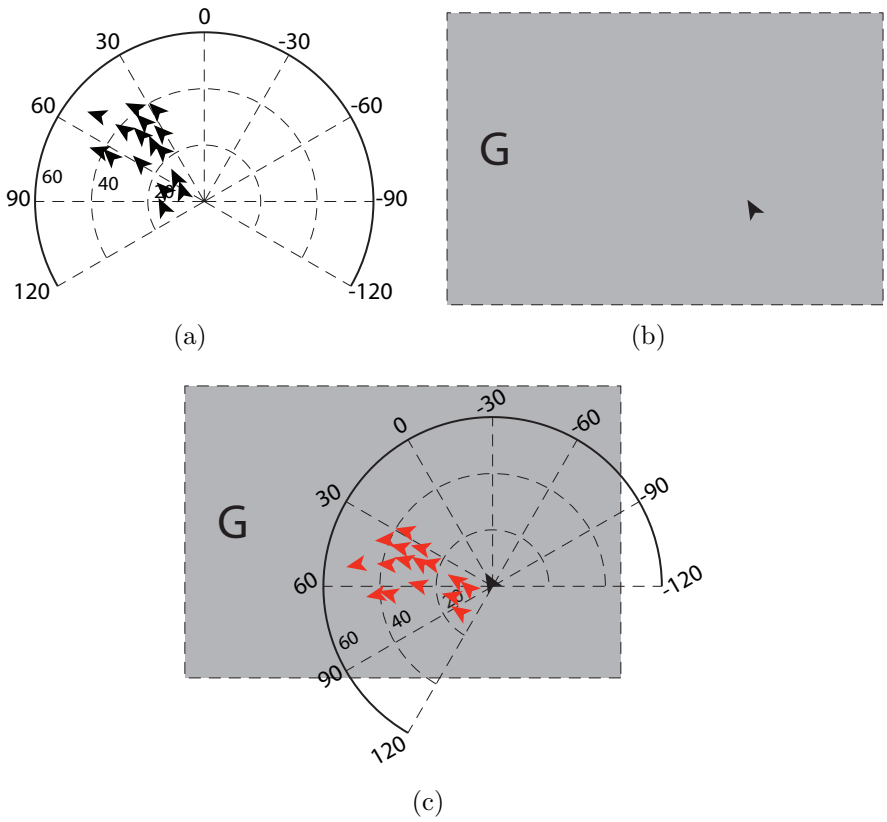


Figure 3.3: These three steps show how possible next states are calculated using outcomes and the agent's current state. (a) Sample outcomes for a turn left action. The agent's start location is shown as $x = 0$, $y = 0$, $\theta = 90$. The dashed lines indicate direction and the dashed arcs indicate magnitude. (b) Sample environment. "G" marks the goal state and the arrowhead indicates the agent's current location. (c) Align the outcomes with the agent's current state to calculate the agent's distribution over next states, shown in red.

lines indicate direction, and the dashed arcs measure magnitude. The arrowheads show resulting orientation.

The sample environment is shown in Figure 3.3(b) where the goal state is marked with a "G" and the agent's current state is shown with an arrowhead. To calculate the agent's possible next states, a transformation is performed to move the origin of the outcome coordinate frame to the agent's current state, as shown in Figure 3.3(c).

3.2.4 Planning

$$Q(s,a) = R(s,a) + \gamma \left(\sum_{o \in O} \left(\frac{t_C(\kappa(s), a, o)}{z} \times \max_{a' \in A} Q(\eta(s,o), a') \right) \right) \tag{3.1}$$

To decide which action to perform in the environment, the agent uses a parameterized and typed variation of the R_{max} algorithm [Brafman and Tennenholtz, 2002], called RAM-R_{max}. Shown in Algorithm 1, this version of R_{max} initializes the value of each state-action pair to the maximum reward, R_{max}. This value remains until the action has been experienced in that state's cluster enough to be "known". An action becomes "known" for a particular cluster when the robot has performed that cluster-action pair M times, where M is a constant free parameter. The value of the action can be derived by solving for $Q(s,a)$ as shown in Equation 3.1 which is updated as t_C changes. Here, z is a normalization constant calculated by

$$z = \sum_{o \in O} t_C(\kappa(s), a, o).$$

To calculate the value of a state-action pair, all outcomes for that state's cluster are mapped to next states using η. This calculation is a weighted sum over the possible next state of every outcome. The weighting is determined by the tally of cluster-action-outcome tuple, $t_C(\kappa(s), a, o)$.

Once the value for all state-action pairs are calculated, the agent chooses the action with the highest value from the current state and sends the corresponding command to the robot. Planning can take several milliseconds on standard hardware. Calculations are performed while the robot is performing its actions (each action takes approximately one second), so there is no computational delay.

3.2.5 System Architecture

Figure 3.4 shows the flow of data through our system for a robotic domain. Before the robot is placed in the environment, a picture is taken with an overhead camera and sent through an image-segmentation engine to determine terrain classification

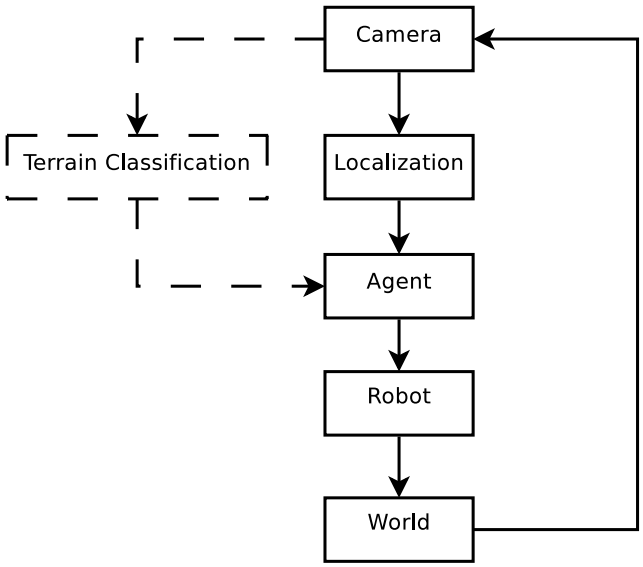


Figure 3.4: Flow chart of the system architecture. The dashed lines indicate information passing that occurs at startup.

(see Section 3.3.2). Classification information is then sent to the agent as a look-up table.

Next, the robot is placed in its starting configuration and the agent queries the localization system for the robot's position in the world. Using this information, the agent, with the guidance of the RAM-R_{max} algorithm (see Figure 1), chooses which action to take based on the outcomes it has observed through experience.

The selected action is then sent to the robot to execute. After the action is completed, the agent once again retrieves the robot's location information and calculates the latest outcome, which is added to a list of outcomes seen in the same cluster. The agent then updates its values for each of the states and chooses the next action to take. This process continues until the localization system tells the agent that the robot is in the goal region or out of bounds. These occurrences end the episode; the robot is placed back in the starting location to execute another episode. Experience are maintained through out the experiment.

Global data structures: Q, t_C
Constants: R_{max}, M, *goal*

`RAM_Rmax()`:
begin
  `INITIALIZE()`
  state $\Leftarrow$ `getLocation()`
  while *state* $\neq GOAL$ **do**
    action $\Leftarrow$ `getBestAction()`
    `takeAction(`*action*`)`
    nextState $\Leftarrow$ `getLocation()`
    `UPDATE(`*state,action,nextState*`)`
    state $\Leftarrow$ *nextState*
end

`INITIALIZE()`:
begin
  for $c \in C, a \in A, o \in O$ **do**
    $t_C(s, a, o) \Leftarrow 0$
  for $s \in S, a \in A$ **do**
    $Q(s, a) \Leftarrow R_{max}$
end

`UPDATE(`*s,a,s'*`)`:
begin
  $o \Leftarrow s' - s^{\dagger}$
  $t_C(\kappa(s), a, o) \Leftarrow t_C(\kappa(s), a, o) + 1$
  for $s \in S$ **do**
    for $a \in A$ **do**
      $Q(s, a) \Leftarrow R_{max}$
  repeat
    for $s \in S$ **do**
      for $a \in A$ **do**
        $z \Leftarrow \sum_{o \in O} t_C(\kappa(s), a, o)$
        if $z \geq M$ **then**
          $$Q(s, a) \Leftarrow r(s, a) + \gamma \sum_{o \in O} \frac{t_C(\kappa(s), a, o)}{z} \times \max_{a' \in A} Q(\eta(s, o), a')$$
  until $Q(s, a)$ *stops changing*
end
†This subtraction is a transformation expressing s' in the coordinate frame of s.

Algorithm 1: The RAM-R_{max} algorithm.

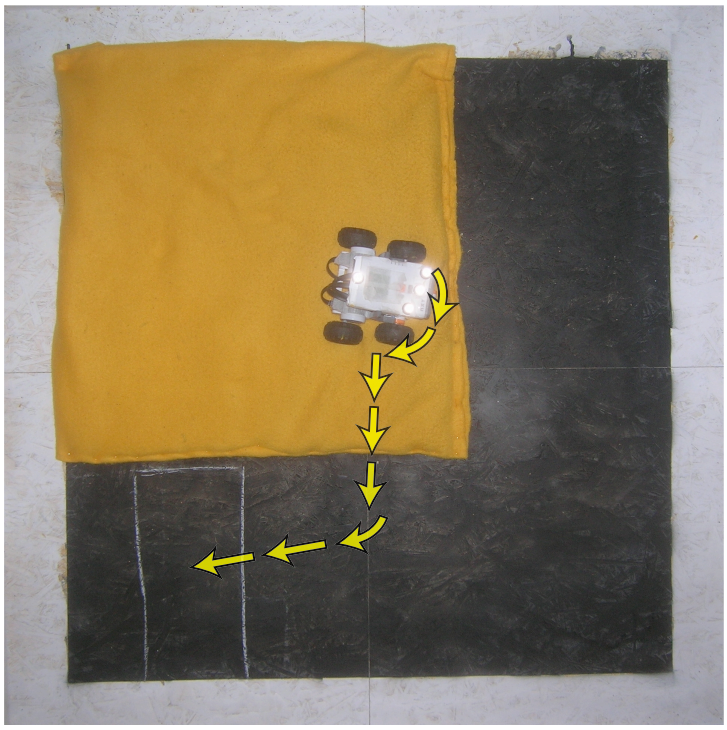


Figure 3.5: Photograph of the sandbag experimental environment. The robot is in the starting location and the arrows indicate the learned policy.

3.3 Evaluation Experiments

In this section, we describe two robotic experiments that illustrate the utility of the RAM-R_{max} algorithm in real-life tasks. Both experiments demonstrate more efficient exploration in comparison with current state-of-the-art techniques.

3.3.1 Experiment with Hand-Tuned Clusters

Sandbag Experimental Setup

Constructed from a LEGO® Mindstorms NXT kit, the robot used had four wheels and was powered by two independent motors. Control computations were performed on a laptop, which issued commands to the robot via Bluetooth® using the LeJOS framework. A VICON motion-capture system was used to determine the robot's state in terms of location and orientation. A hand-tuned color-based

classifier performed terrain classification.

The Sandbag experimental environment, shown in Figure 3.5, was a 4×4-foot "room" with two different surfaces textures (wood and sand-filled cloth bag) discretized into 625 states. We found that the robot traversed the cloth roughly 33% more slowly than it did the wood.

The agent was informed at the beginning of the experiment which states were the goal, having a reward of 1, and which states were out of bounds, having a reward of -1. The agent also knew there was a reward of -0.01 for each state-action pair that did not result in a terminal state.

At the beginning of each timestep of the experiment, the agent was fed several pieces of information. The localization system told the agent its state. The image parser informed the robot of which cluster was associated with the current state based on calibration performed before the experiment.

The actions available to the agent were: turn left, turn right, and go forward. Each action was performed for 500 ms. Once an action completed, there was a 250 ms delay to allow the robot to come to a complete stop. Displacement of the turn right and go forward actions are shown in Figure 3.5.

Results

To determine how RAM-R_{max} performed in relation to current state-of-the art algorithms, it and R_{max}were run in the Sandbag experimental environment. We had originally planned to also run Q-learning [Watkins and Dayan, 1992] in the environment, but other experiments shows that it would not be able to learn given the battery life of the robot [Leffler et al., 2007]. For both of these algorithms, we set $M = 4$ and $\gamma = 1$. Figure 3.6 shows the cumulative reward that each of these learners received over 50 episodes. Each episode began with the agent being placed in roughly the same place and ended when the robot entered a terminal state. The rapid rise of the RAM-R_{max} curve shows that the learner almost

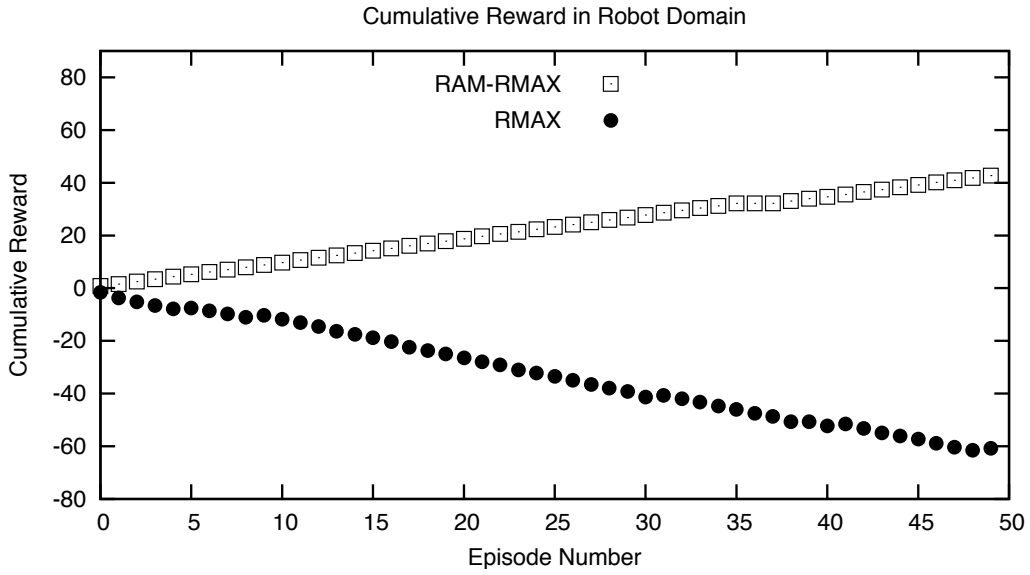


Figure 3.6: Cumulative reward per run of all algorithms compared in the sandbag experimental environment.

immediately started to follow an excellent policy.

After the first episode, the exploration phase of the learning was complete for the RAM-R_{max} agent; all actions had been explored M times in both state types. Over the next few runs, the probabilities in the model converged, and starting at Episode 5, the path taken by the RAM-R_{max} agent seldom varied from that shown in Figure 3.5. The R_{max} agent, on the other hand, kept the value of R_{max} for the majority of state-action pairs after 50 episodes, and, therefore, it was still actively exploring. Setting $M = 1$ did not visibly speed the agent's learning process. That is, the agent did not have a large reward for revisiting states and, without the ability to generalize, the R_{max} algorithm was unable to learn effectively in this domain.

A more finely discretized state space could have led to a more effective policy, but was not tried due to the negative impact that it would have had on R_{max}. In contrast, the amount of exploration performed by the RAM-R_{max} algorithm would have remained the same. This more efficient exploration policy is due to the its dependence on the number of clusters which would not have changed.

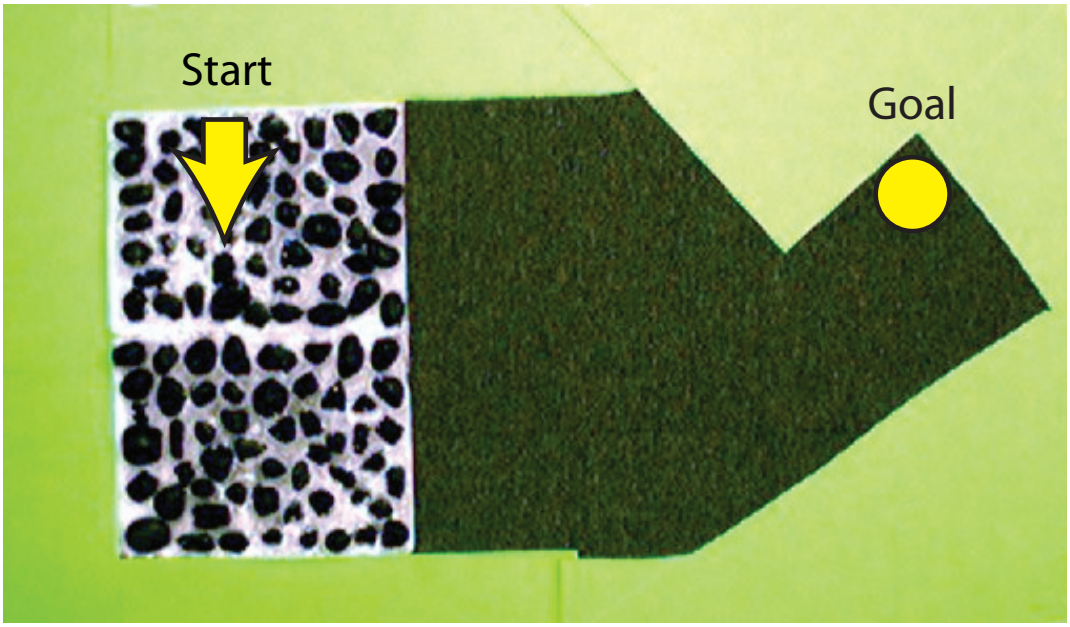


Figure 3.7: Image of the RockyRoad environment. The start location and orientation is marked with an arrow. The goal location is indicated by the circle. Green pieces of poster board are shown here marking the boundaries of the environment.

3.3.2 Experiment with Automated Terrain Classification

For a second experiment, the system architecture included a camera-based localization system and automated terrain classification.

Terrain Classification

An IEEE1394 video camera was used to take an image of the navigation environment without the robot. This image was then fed into the Edge Detection and Image SegmentatiON (EDISON) system [Christoudias et al., 2002] where similar patches of terrain were determined based on color, texture, and proximity of pixels. The image-segmentation system determined the number of clusters on its own, but needed a parameter to be set stating the minimum size of a cluster. This value was set to the number of pixels that the robot occupied in the image to ensure that all of the robot's wheels could occupy a single patch of terrain at the same time.

Figure 3.8: Image of the LEGO® Mindstorms NXT robot in the RockyRoad experimental environment.

To limit the number of spurious clusters, all clusters that appeared in fewer than one-tenth of the states had their pixels reassigned to adjacent segments. The remaining clusters were used to determine κ. This form of clustering was performed with the expectation of combining states with similar dynamics models.

Localization

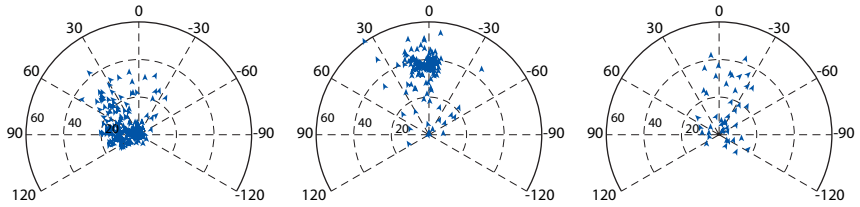
The localization system was a standard fiducial-based system, which for these experiments acted as an indoor global positioning system (GPS). Using the same overhead camera mentioned previously, the location of a marker affixed to the robot was obtained using commonly available color-segmentation software [Bruce et al., 2000]. The type of marker used was based on work done for a robotic soccer application [Bruce and Veloso, 2003]. The typical precision of this system is approximately 5mm.

RockyRoad Experimental Setup

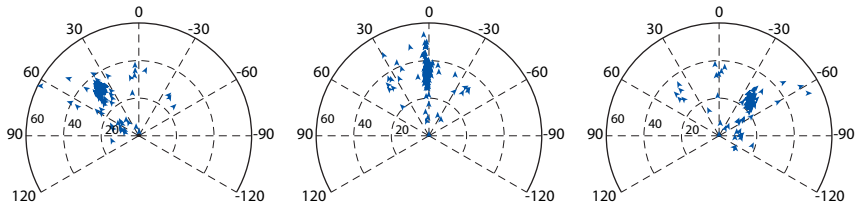
For this experiment, a LEGO® Mindstorms NXT robot(see Figure 3.8) was run on the RockyRoad environment. This domain, shown in Figure 3.7, consisted of a highly variable region comprised of rocks embedded in wax and a smoother carpeted area. The agent's task was to begin in the starting location (indicated in the figure by an arrow) and end in the goal (indicated in the figure by an ellipse) without going outside the environmental boundaries. The rewards were -1 for going out of bounds, 1 for reaching the goal, and -0.01 for taking an action that led to a non-terminal state. Reaching the goal or going out of bounds ended the episode.

One difficulty of the RockyRoad environment is the large difference in dynamics on the rock and carpet surfaces. Figure 3.9 shows the outcomes observed by the agent on these two surfaces. The center of the circle represents the starting location of the robot. The dashed lines indicate the direction (in degrees) and the arcs indicate the magnitude (in pixels) of displacement. From left to right, this figure shows the outcomes of the left turn, go forward, and right turn actions on the rock (top) and carpet (middle) surfaces. The bottom row shows the same outcomes as above, but combines the two terrains to demonstrate the amount of noise that is introduced when the terrains are assumed to be similar. Some actions, such as turning right on rocks, are more sparse than others in the figure due to the number of times that an action was taken during the exploitation phase.

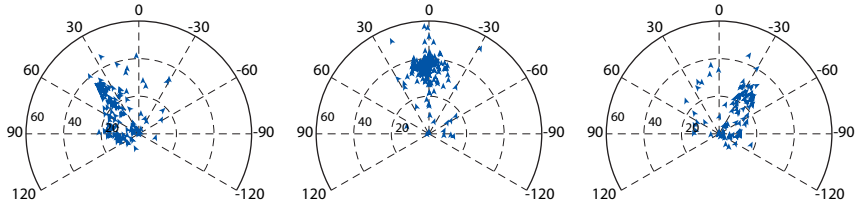
Due to the close proximity of the goal to boundary, the agent needs to accurately model the robot's dynamics to reliably reach the goal. To make this task even more difficult, the actions are limited to going forward, turning left, and turning right. Not allowing the agent to move backwards increases the need for the agent to accurately approach the goal. For example, if the robot enters the narrow portion of the environment facing away from the goal, it is not able to turn around without going out of bounds. Therefore, an agent with an inaccurate



(a) Left turn on rock (b) Forward on rock (c) Right turn on rock



(d) Left turn on carpet (e) Forward on carpet (f) Right turn on carpet



(g) Left turn on both (h) Forward on both (i) Right turn on both

Figure 3.9: Outcomes learned by the robot for different actions and surfaces.

dynamics model is likely to think this task is impossible and might choose to drive out of bounds quickly to minimize step cost.

For the experiments, we compared the RAM-R_{max} and tree-based fitted Q iteration [Ernst et al., 2005] algorithms (see Section 2.2.1), each with and without terrain classification. All algorithms were informed of the reward function. The agents without terrain classification assumed one cluster for the entire domain. Figure 3.10 shows the results of the EDISON image-segmentation engine when fed in the image of the world and the minimum region to segment. Recall that the minimum region to segment was specified to be the number of pixels that the robot occupies in the image, which for this experiment was 4000 pixels.

For all agents, the world was discretized to a forty by thirty by ten state space instead of the camera's full resolution of 640 by 480 by 360 degrees of orientation. This coarse discretization was used to limit the number of states that the robot could occupy at once. Lastly, each algorithm had the value of M set to ten, which was chosen after informal experimentation.

Results

Figure 3.11 shows the average performance and standard deviation of the RAM-R_{max} and fitted Q iteration agents with and without terrain classification over five runs of twenty episodes. When the RAM-R_{max} agent used image segmentation to determine the surface types in the environment, it reached the goal in 61% of the episodes as opposed to 22% of the episodes when no clustering data was given. This difference is more noticeable in the last 10 episodes after some learning had taken place. Narrowed to these instances, the success rates are 96% and 34%, respectively. The fitted Q iteration agents were not able to reach the goal in any of the runs with or without the terrain classification. Doubling the number of episodes to 40 in a run also did not result in any positive reward for either of the fitted Q iteration agents. Indeed, published results with this algorithm suggest

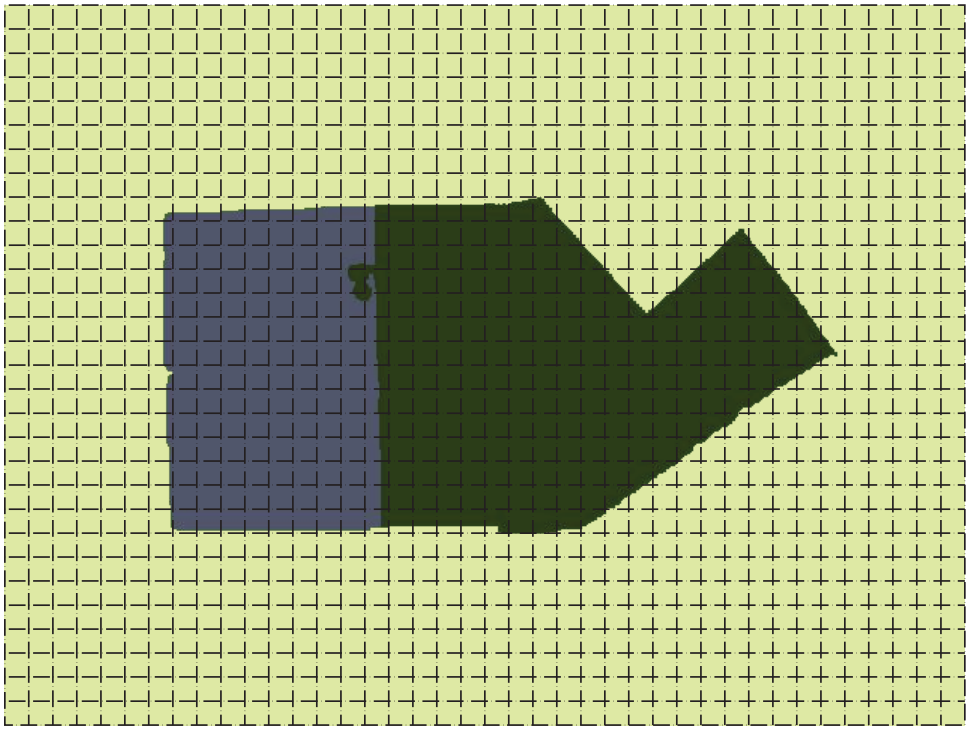


Figure 3.10: Resulting discretized segmented image from EDISON of the RockyRoad environment showing two different surface types. Several states were mislabeled due to the image processing algorithm, but these mislabelings did not harm the results.

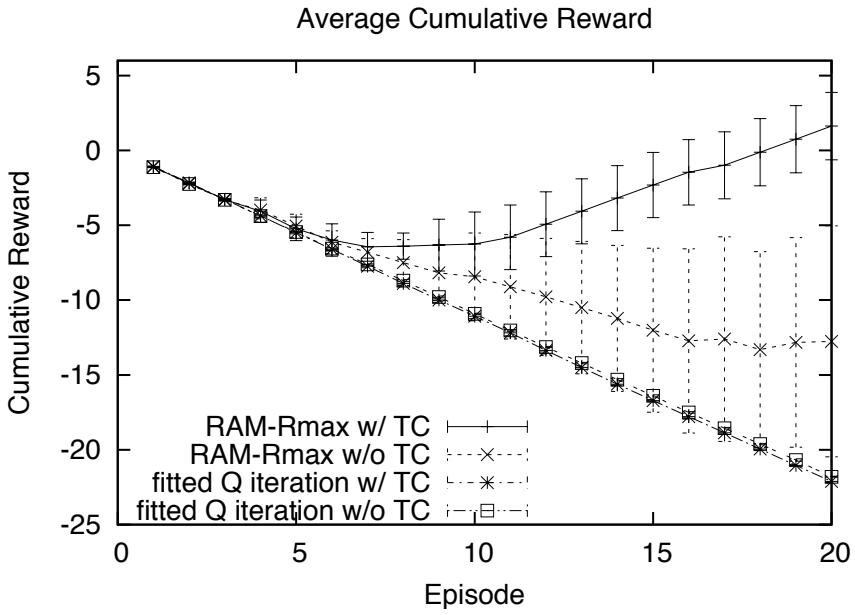


Figure 3.11: Graph of the RAM-R_{max} and fitted Q iteration algorithms' average cumulative reward with and without terrain classification. Agents not given terrain classification assumed one terrain.

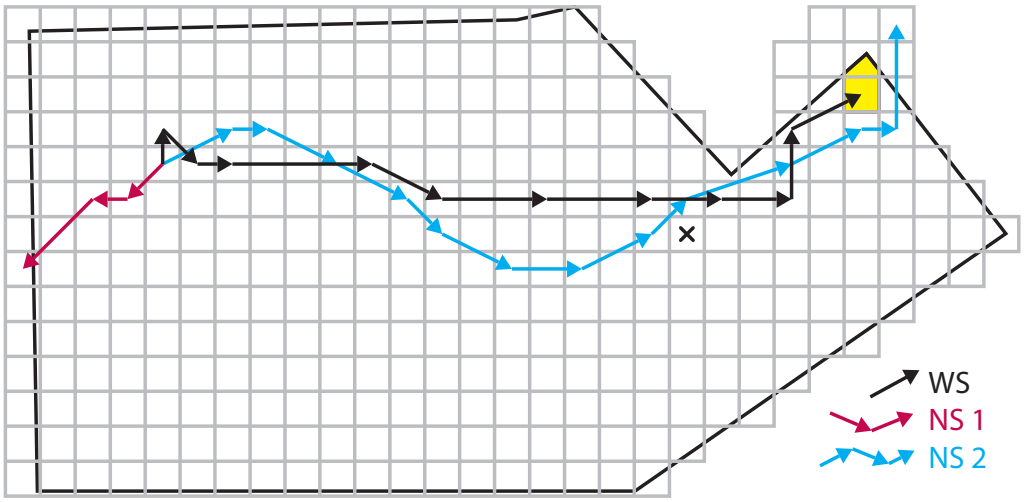


Figure 3.12: Diagram showing paths taken by the RAM-R_{max} algorithm with segmentation (WS) and with no segmentation (NS) *en route* to the goal after some learning has occurred (shown in yellow). NS1 demonstrates a sample path in which the agent judged the goal as unreachable and so minimized negative reward by exiting the environment quickly. NS2 shows a sample path in which the agent's inaccurate model caused it to miss the goal. "X" marks the location used for the example in Section 3.4.

hundreds or thousands of episodes are often needed [Ernst et al., 2005].

Figure 3.11 also shows a large variance in the performance of the RAM-R_{max} agent that did not use terrain classification. The reason for this difference is the variability in the dynamics model that the agent learned in each run. Because no single dynamics model is a good fit for the two terrains, the model changed over time, sometimes assigning the overall environment a very noisy model like the rocks and sometimes assigning a very reliable model like the carpet. These fluctuations in the model caused the chosen trajectory of the same agent to change drastically between runs as shown Figure 3.12. In two of the five runs, the RAM-R_{max} agent that had no terrain classification chose to navigate towards the goal as shown by the path marked NS 2. However, the dynamics model that it had learned was innaccurate, and the agent would accidentally drive out of bounds when approaching the goal. In the other half of the runs (marked in Figure 3.12 by NS 1), the agent did not think that it was possible to reach the goal based on

its learned dynamics model, and, therefore, had a negative maximum value. This RAM-R_{max} agent chose to drive out of the environment as quickly as possible to minimize negative reward.

In contrast, the RAM-R_{max} agent with image segmentation (WS) learned that the rocky surface was unpredictable, but that the carpet surface allowed for consistent actions. Once these two surfaces were learned, the agent was able to arrive at the goal reliably, as shown in Figure 3.12, seldom over-shooting the goal.

3.4 Discussion

The two RAM-R_{max} agents outperformed both of the fitted Q iteration agents due to the efficiency with which they used experience data. The RAM-R_{max} agents with and without terrain classification were able to generalize their observed outcomes to unseen states in the same cluster, which limited the amount of necessary exploration. Since fitted Q iteration does not model the environment, its generalization ability was limited to exploiting local consistency in the value function. Throughout the twenty episodes, neither of the fitted Q iteration agents were able to take advantage of the underlying structure and were still actively exploring the environment. Because the fitted Q iteration algorithm needs so much experience, they hadn't acquired a policy capable of reaching the goal without going out of bounds.

The performance discrepancy of the two RAM-R_{max} agents can be explained by examining the learned value function of the two agents. For example, when supplied with terrain classification, the average value of a state near the goal ($X = 25$, $Y = 15$, $\theta = 0$, marked with an "X" in Figure 3.12) was 0.450 with a standard deviation of 0.194. In comparison, the agents without terrain classification on average calculated the value of the same state to be 0.1782 with a standard deviation of 0.373. The difference in this state's expected values comes from

the variance in the learned dynamics models. Without terrain classification, the RAM-R_{max} agent learned a single dynamics model with high variance, which tells the agent that there is a relatively low probability of reliably reaching the goal. The agent with terrain classification, on the other hand, creates two different dynamics models—one with large variance and one with small variance. Using these two separate models allows the agent to create a more accurate model of the environment, which is necessary to determine that the probability of reaching the goal from the carpet is high.

However, there is a limit to how much improvement additional context can add. If the terrain classifier were to have found a third distinct type corresponding to states on the border between the two terrains, the agent might have been able to model the dynamics of when its front wheels were on one surface and its back wheels on another, possibly improving performance. To improve the policy further, the agent could also have declared each rock its own surface, allowing for the ability to model its likelihood of getting stuck on each particular rock. The downside of this approach is experience efficiency. The more surface types that the learner recognizes, the less it generalizes and the more exploration it needs; in the limit, as the number of clusters approaches the number of states, this algorithm becomes equivalent to (non-generalizing) R_{max}. The experiments reported in Section 3.3.1 suggest that such an approach would be completely ineffective in the RockyRoad environment.

3.5 Conclusion

This chapter introduced the RAM-R_{max} algorithm and explained how it learns the outcomes and relocatable action model of a RAM MDP in a robot-navigation environment. This algorithm performs efficient exploration through state-space clustering determined by perceptual information, specifically terrain classification.

The two experiments detailed in this chapter showed that using terrain classification for state-space clustering as an input to the RAM-R_{max} algorithm is a reliable approach to achieving accurate models with limited exploration. They also demonstrated the benefits of using the RAM MDP for generalization in contrast to current state-of-the-art techniques, namely R_{max} and fitted Q iteration. In addition, they demonstrated validity of the assumptions used in our algorithm.

Chapter 4
Reward-Based Clustering

4.1 Introduction

Now that the benefits of using the RAM-R_{max} algorithm to learn and perform in robot-navigation environments have been introduced, this chapter will address the issue of learning the state-space clusters that are integral to performing efficient exploration.

As explained previously, the reinforcement-learning paradigm includes an agent getting feedback from its environment. In the last chapter, the agent relied on additional sensor data to obtain clustering information. This chapter will show how an agent can determine which states are similar with no additional information from the environment. Through its experience, an agent can detect the latent structure of the environment and perform efficient exploration by exploiting similarities in the state space.

4.2 Problem Definition

When a reinforcement-learning agent performs an action, it receives feedback from the environment. This data includes information about the current state and the reward for the previous state-action pair. Previous research has used reward information to determine similarities in the state space [Givan et al., 2003; Chapman and Kaelbling, 1991]. However, most of these approaches then use these similarities omit much of the state information. In contrast, our approach, clusters

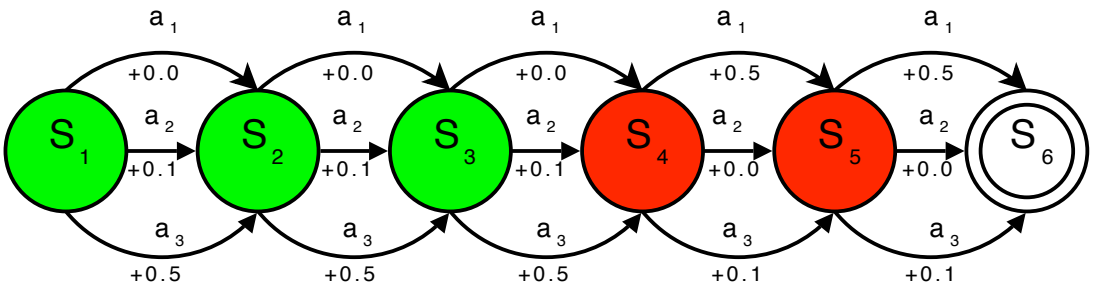


Figure 4.1: An RAM MDP with an action-independent transition function.

the state-space to infer the environment's underlying structure to allow for the transfer of experience between similar states.

Figure 4.1 shows a relocatable action model (RAM) Markov decision process (MDP) where all of the actions that the an agent can take from one state lead to the same next state. The transition function in this environment is, therefore, action-independent. Without any additional sensors, it seems difficult for the agent to accurately cluster the state space. However, as the agent gathers experience, it can use the rewards that it receives from the environment to determine similar states.

For example, if the agent navigated through the state space in Figure 4.1 and performed action a_3 in states s_1 through s_6, it would receive a reward of 0.5, 0.5, 0.5, 0.1, and 0.1, respectively. Using this information, the agent could use unsupervised learning (see Section 2.1.1) to determine that states s_1, s_2, and s_3 belong in one cluster, and states s_4 and s_5 belong in a second cluster. Since the agent cannot take an action from state s_6, it will not be included in either group. Once the state space similarities have been determined, the agent can generalize its experience across states in the same cluster, allowing for more efficient exploration.

One concern with having an agent assume that there are states with similar properties in the environment is the cost of finding the similarities in the state space—especially when there are no redundancies in the state space. However,

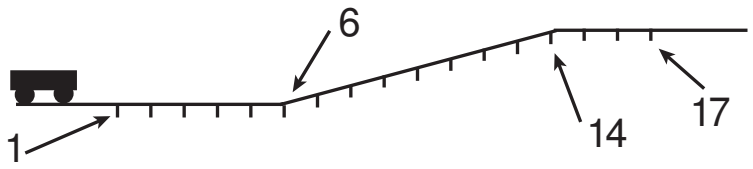


Figure 4.2: A diagram of a robot environment modeled as an MDP with an action-independent transition function.

Leffler et al. [2005] proved that in certain domains—even when every state is unique—adding methods to learn the underlying state clustering to an agent does not change the amount of experience needed to model the environment in the worst case. In addition, we showed that performing clustering has the potential to greatly improve the agent's experience efficiency by changing its sample-size from being dependent on the number of states to the number of clusters. We demonstrate precisely such as case in the next section.

4.3 Experiment

Experiments were performed to give an empirical demonstration of the benefits of state-space generalization with initially unknown clustering. The environment used was similar to the MDP in Section 4.2 and is shown in Figure 4.2. The state space consisted of 17 locations with 2 underlying clusters: uphill and flat. At each state the agent can take one of seven actions each corresponding to the robot's motor power. The reward function received was dependent on the time it took for the agent to travel between states.

4.3.1 Algorithms Tested

Several algorithms were compared in this environment to demonstrate the benefits of assuming and learning state clusters. The algorithms compared were a known-policy algorithm and three variations of Fong's "naïve" algorithm [Fong, 1995] that learns the optimal policy by trying every action in every state a fixed number

of times, like the R_{max} algorithm [Brafman and Tennenholtz, 2002].

No Clustering

In order to show the benefits of clustering the states, we ran an algorithm that had no clustering—known or learned—as a baseline. This algorithm was a direct implementation of the naïve algorithm where each of the seven actions was tried once in each state. After the seventh traversal, the agent determined the best action for each state and, thereafter, performed the expected best action.

Known Policy

For our known-policy algorithm, I collected data in advance by having the robot perform fourteen traversals of the environment "offline" to estimate model parameters. States were grouped by hand based on their perceived slope. These groupings were further verified by reward data. Actions chosen for the agent to perform in each state type were also determined by hand based on reward data received during training. This case was intended as an optimistic baseline.

Known Clustering

For verification of the upper limits of the benefits of clustering, one of the algorithms compared had the state clusters known in advance. Like the known-policy algorithm, these clusters were labeled by hand. With this information programmed in, the agent ran the naïve algorithm on the two clusters by trying every action once in each of the two clusters. Since a single traversal was long enough to sample each action in each cluster, it only took one traversal to learn reward values and obtain near-optimal behavior.

Figure 4.3: The robot used in the two-slope environment.

Learned Clustering

Finally, to see if state-clustering should be learned, we ran a RAM-naïve algorithm that used reward data to cluster the state space. Those clusters then were used to learn the remainder of the model more efficiently. To collect the reward data, the agent first traversed the entire course with a constant action three times. The resulting rewards were fed into a single-link hierarchical clustering algorithm [Hubert, 1974]. This algorithm constructed clusters based on the minimum L1 (Manhattan) distances between the rewards received.

With the the number of clusters set to two based on the slope perceived by the experimenter, the agent determined the state grouping. On the fourth traversal, the policy performed was identical to the policy of the known-clustering algorithm.

4.3.2 Experimental setup

To compare the algorithms in a physical setting, a robot, shown in Figure 4.3, was built to traverse a two-slope test course. The robot was constructed using parts from a Lego® Mindstorm kit, specifically a RCX 2.0 block, one rotation sensor, one motor with seven power levels, four wheels, and various connecting pieces. The course was made up of three two-by-two boards supported by plywood as

Figure 4.4: The robotic vehicle and environment used in the two-slope experiment. The agent started at the left-most location on the ramp and travelled 1700 clicks on the rotation sensor.

shown in Figure 4.4. The robot's software was built using the leJOS programming language.

Each algorithm was run for two epochs, consisting of 12 traversals each. The interval between states was defined as 100 rotation clicks as registered by the robot's rotation sensor. The reward for a traversal was calculated using the following formula:

$$-\sum_{l}(round(\frac{t_l - t_{l-1}}{100}) - t_g)^2 \tag{4.1}$$

where *round* and division by 100 are used to discretize the outcomes, t_g is the goal time elapsed between points as derived from the goal speed g, t_l is the time the robot reaches location l and the additive inverse is used to map cost to reward.

Since the RCX 2.0 does not possess adequate memory to perform the calculations needed for the aforementioned algorithms, the robot stored the observed speeds during each traversal. At the end of each traversal, the collected data was transmitted to a laptop, which processed the data and calculated the policy to use in the next traversal as per the algorithm being evaluated.

Evaluating algorithms in the real world invites a host of noise factors that one would not consider in a pure simulation. One such factor in our implementation was the substantial effect battery power had on the robot's performance. As the batteries drained, the effect of the power commands on the movement of the

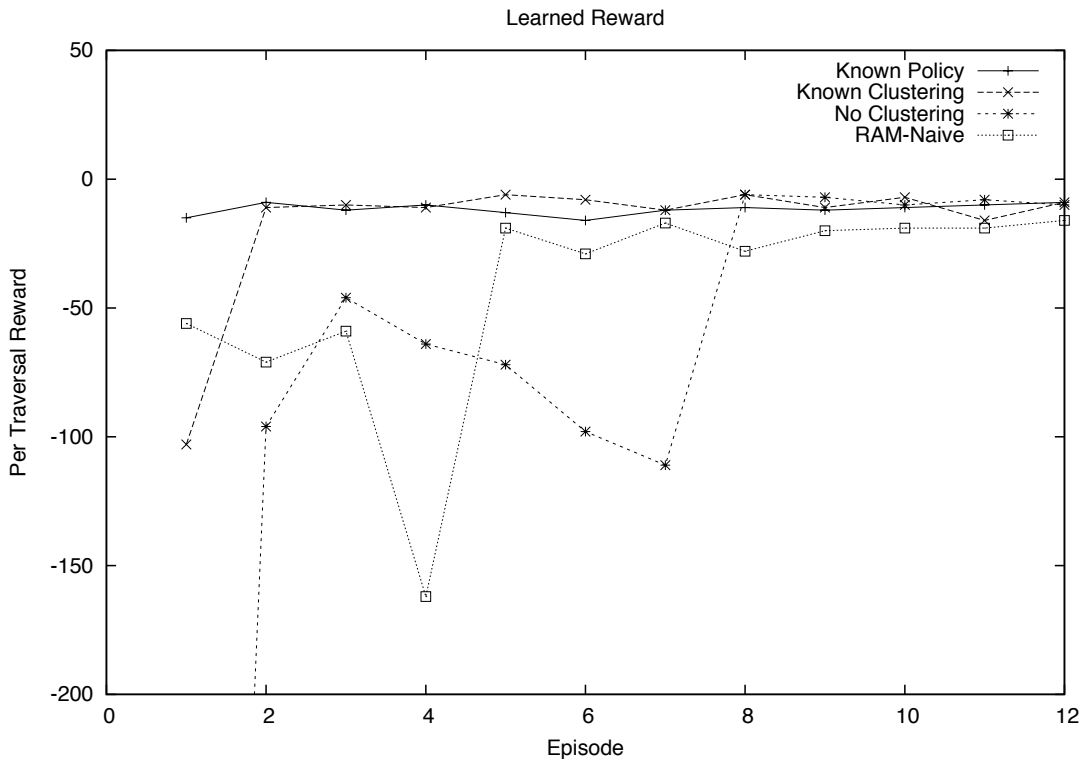


Figure 4.5: Per traversal reward for each of the tested algorithms.

robot changed. Although the learning algorithms were able to adapt to different battery powers, the large variability of the action effects threatened to make the results incomparable. To compensate, brand new disposable batteries were used for each algorithm studied. Thereafter, any decline in battery power appeared consistent across all algorithms.

4.3.3 Results

Figure 4.5 shows the per traversal reward of the algorithms demonstrating the performance of all resulting behaviors. The graphs for the second epoch were omitted because they did not differ significantly from Epoch 1. As expected, the known-policy performed well in every traversal since no learning was required. Following the same reasoning, the known-cluster algorithm learned a good policy faster than the no-clustering and RAM-naïve algorithms.

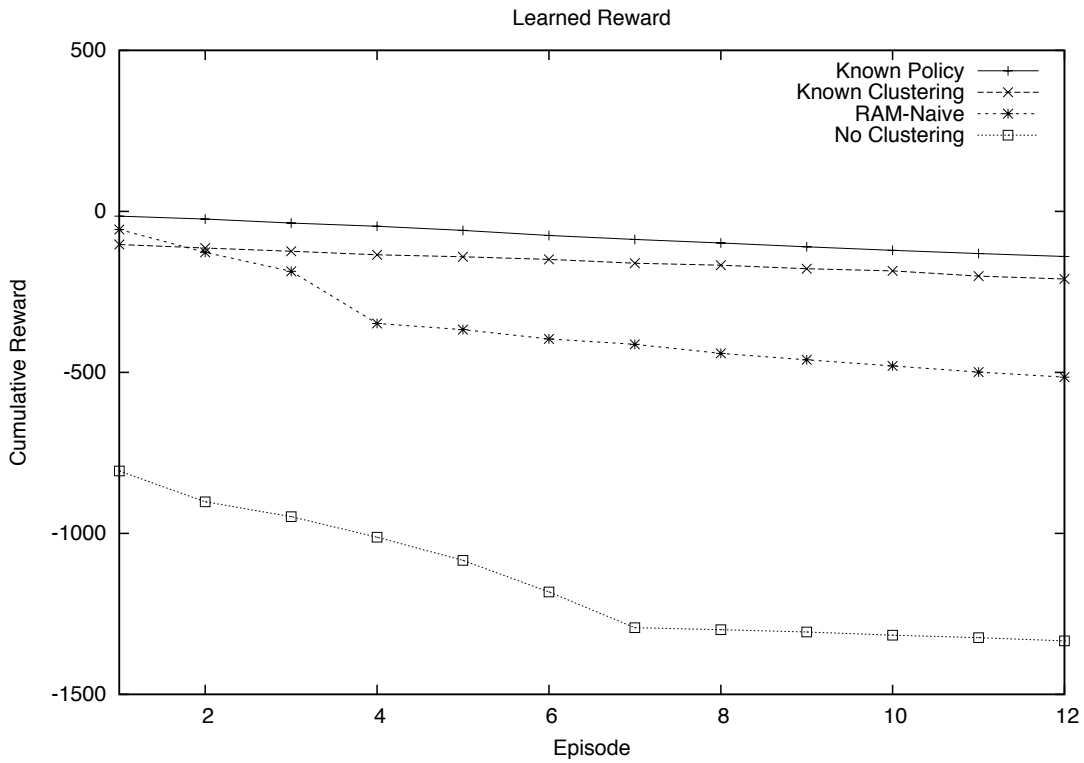


Figure 4.6: Cumulative reward for each traversal for all algorithms.

Surprisingly, the known-policy algorithm did not always receive the highest per traversal reward. This result is because the assumption that only two classes exist in the world (uphill and flat) was not entirely valid. In actuality, each position was slightly different. Certain locations, in particular, were really transitions between the flat and sloped surfaces and could have been their own cluster. So, even though the algorithms that exploited the latent structure quickly discover which positions belong to which clusters, they fail to achieve maximum reward using the small number of clusters. The non-clustering approach was able to represent and exploit the differences between the locations, resulting in slower learning, but apparently optimal reward.

The reward data collected by the RAM-naïve agent and the clusters that resulted are shown in Table 4.1 and Figure 4.7, respectively. As mentioned earlier, the level of clustering chosen was such that the states were divided into two

Table 4.1: Reward Data Collected by the RAM-naïve algorithm in each state during the first three traversals.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Trav. 1	-4	-9	-4	-9	-4	-4	-4	0	0	0	0	0	0	0	-4	-9	-4
Trav. 2	-9	-9	-9	-4	-9	-4	-4	0	0	0	0	0	0	-1	-9	-4	-9
Trav. 3	-9	-9	-4	-4	-9	-4	-1	0	0	0	0	-1	0	-1	-4	-4	-9

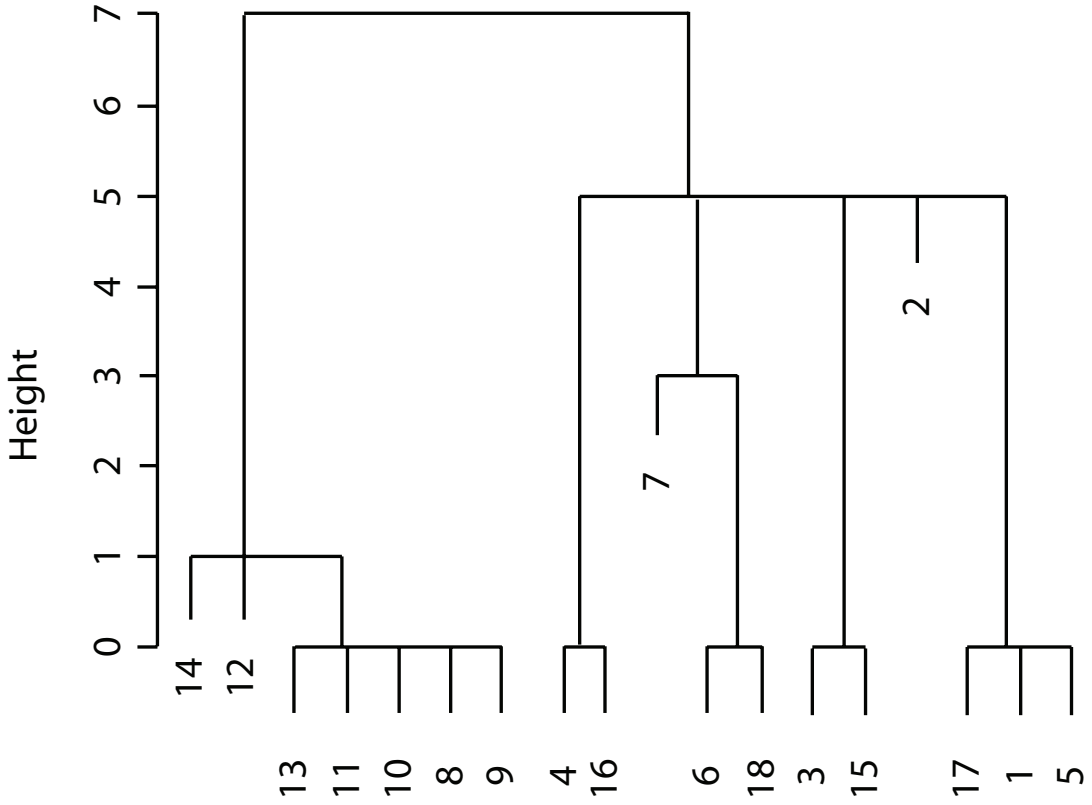


Figure 4.7: State clusters learned and given as input to the RAM-naïve algorithm. The clusters become more inclusive as the height gets larger. At a height of 5, all states are divided into two clusters.

clusters. While this algorithm was not able to achieve the same per-traversal reward as the other algorithms, it had learned the model and began receiving higher rewards several traversals earlier than the no-clustering algorithm. This more efficient exploration policy led to a higher cumulative reward, as shown in Figure 4.6. Even though the algorithm that performed no clustering ended up with a better policy, the difference in reward received does not make up for the large initial cost of exploration in this small number of traversals.

This experiment was also run using Expectation Maximization (EM) [Dempster et al., 1977] to perform the clustering task of the RAM-naïve algorithm. The clusters determined were the same as the results of the hierarchical clustering algorithm leading to similar behavior and cumulative reward.

4.3.4 Discussion

This experiment showed the inherent benefits of generalizing experience. Even with the exploration necessary to determine clusters, and learning a sub-optimal policy, the RAM-naïve algorithm was able to receive a higher cumulative reward than an algorithm that used no generalization.

However, this algorithm would not be beneficial in all domains. For instance, if the task were to continue for an additional one-thousand traversals, the cost of initial exploration would be outweighed by the per traversal reward. In addition, this environment was very limited in the transition function. Environments that do not contain action-independent transition functions would require a different planning policy to ensure that all state-action pairs were navigated to and explored.

Still, the advantages to assuming similarities in the state space and using a RAM MDP to model them often allows for more efficient exploration, even when those similarities have to be learned. Sample bounds have been found for these techniques [Leffler et al., 2005]: the non-clustering approach has a lower bound of

$\hat{O}(lk \ln l)$ while the RAM-naïve approach has a lower bound of $\hat{O}(l \ln l + nk \ln n)$, Where l is the number of states, n is the number of clusters, and k is the number of actions. As the number of clusters grows larger than the number of states, the RAM-naïve approach becomes more sample efficient than assuming that no clusters exist.

4.4 Conclusion

This chapter has introduced how an agent can learn state-space clusterings of an environment using only reward information. In addition, it showed that the benefits of the generalization can outweigh the costs of initial exploration in certain domains.

The experiment demonstrated these benefits in a real-world domain. The results of implementing an algorithm that models the environment as a RAM MDP in a robot-navigation tasks showed that even though the learned policy might not be optimal, an agent might receive a better cumulative reward by limiting exploration and exploiting a near-optimal policy sooner.

Chapter 5
Determining Accurate Classifiers

5.1 Introduction

Up until this point, the assumption has been made that there is one single classifier, such as terrain classification or reward function, from which state-space clustering can be determined. Unfortunately, this assumption does not always hold. In robot-navigation environments, the number of classifiers could be as large as the number of sensors on the robot. One option for handling this abundance of information is to incorporate all of the classifiers and create a large number of clusters. However, a large number of clusters hinders the benefits of generalization. Another option is to make an assumption about which classifier indicates the proper grouping. If this assumption is incorrect, though, the performance of the agent would be poor, possibly leading to the agent's inability to perform the given task. The solution presented in this chapter is to have the agent learn from experience which classifiers are indicators of states with similar dynamics. That is, it will learn the accurate classifiers.

5.2 The Meteorologist Algorithm

In order for an agent to be able to determine the features on which it should cluster, it needs to be able to detect whether a proposed clustering is accurate. For this reason, we use the *Knows What It Knows*, or KWIK, framework [Li et al., 2008]. This framework specifies that an agent can calculate whether it has

Table 5.1: Sample meteorologist predictions.

	Channel 2		Channel 3		Channel 4		Channel 5		actual
	Pred.	Error	Pred.	Error	Pred.	Error	Pred.	Error	
Day 1	0.1	0.01	0.25	0.0625	0.5	0.25	1.0	1.0	SUN
Day 2	0.75	0.0625	0.6	0.36	0.25	0.5625	——	——	RAIN
Day 3	0.9	0.01	0.75	0.0625	——	——	——	——	RAIN
Day 4	0.5	0.25	0.9	0.81	——	——	——	——	SUN
Day 5	0.25	0.0625	——	——	——	——	——	——	SUN

enough information to make accurate predictions about its environment. If this agent is given a particular hypothesis, it can also return whether that hypothesis is correct with a high probability. By combining a set of these hypotheses and testing them against the same input, an agent can determine which hypothesis is error-free.

One example of where this set of hypothesis classes can be applied is the task of deciding a television meteorologist. Let us assume that someone has just moved into a new city and wants to get good weather reports. Further, let's assume that there is one meteorologist who accurately predicts the weather—when he says that there is a 30 percent chance of rain tomorrow, then 30 percent of the time, it will rain. The other people might be almost as accurate, or possibly inaccurate, but at least one is right on the money.

Table 5.1 shows some sample predictions of four different television channels and how the most accurate channel is chosen using a mean squared error of greater than 50 percent to eliminate inaccurate meteorologists. On the first day, Channel 5 made a poor prediction and is disregarded. On Day 2, Channel 4 had an error of 56.25 percent, and is also excluded. This continues for several days until only Channel 2 is left as a reliable meteorologist. If more than one of the channels were deemed to be accurate, each of the channels' predictions would be weighted equally.

A *meteorologist-selection* algorithm can be used in conjunction with the RAM-R_{max} algorithm to detect which perceptual input gives the agent the best predictions about the dynamics of the environment.

5.3 The SCRAM-R_{max} Algorithm

In a robot-navigation domain, we can assign a "meteorologist" to each of robot's perceptual inputs. These meteorologists each have the hypothesis that the dynamics model of the robot is correlated to a given input. The meteorologist-selection algorithm then determines which hypothesis is most accurate. The chosen hypothesis is used as the clustering function of the RAM-R_{max} algorithm; the resulting algorithm is called the *Sensor Choosing for Relocatable Action Models-R_{max}*(SCRAM-R_{max}) algorithm.

This algorithm, shown in Algorithm 2, uses the same exploration policy as R_{max} and the same action-selection policy as RAM-R_{max}. The contribution of the SCRAM-R_{max} algorithm is the acquisition of the accurate clustering function and the use of that function to generalize experience. When an agent starts out in the world, it has the assumption that all the features give information about the robot's dynamics and creates a hypothesis class for each feature. This assumption allows for a more exploration-driven policy until the agent has enough data to determine whether this assumption is correct.

As the agent starts a task, it travels toward the goal, gathering experience. This data is input into a meteorologist-selection algorithm, which consults the set of hypothesis classes to see if any of them are giving predictions with a high error. If not, the agent will continue towards the goal as planned. However, once a hypothesis has a large enough error based on the agent's experience, it is eliminated. Dynamics models for each of these clusters are then recalculated and the values of each of the states are updated to reflect this change. At this point,

the agent can decide to perform more exploration based on the newly reduced clusters, or if the agent has enough experience, it can continue to the goal.

```
Global data structures: M
Constants: goal

SCRAM_Rmax():
begin
    INITIALIZE()
    state ⇐ getLocation()
    while state ≠ goal do
        action ⇐ getBestAction()
        takeAction(action)
        nextState ⇐ getLocation()
        CONSULT_METEOROLOGISTS(state,action,nextState)
        UPDATE(state,action,nextState)
        state ⇐ nextState
end

CONSULT_METEOROLOGISTS(state,action,nextState):
begin
    updateMeteorologists(state,action,nextState)
    enoughData ⇐ true
    forall m ∈ M do
        prediction ⇐ getPrediction(m)
        if prediction = IDK then
            enoughData ⇐ false
            break
        if enoughData then
            bestClassfier ⇐ getBestClassifier()
            regroupOutcomes(bestClassifier)
end
```

Algorithm 2: The SCRAM-R_{max} Algorithm where INITIALIZE and UPDATE are the same as in Algorithm 1.

5.4 Experiments

Two experiments using LEGO Mindstorm NXT® robots were performed to demonstrate the benefits of learning accurate classifiers. The first experiment shows the importance of knowing the correct perceptual input for state-space clustering. The second experiment evaluates the speed at which SCRAM-R_{max} can learn this information in a real-life task.

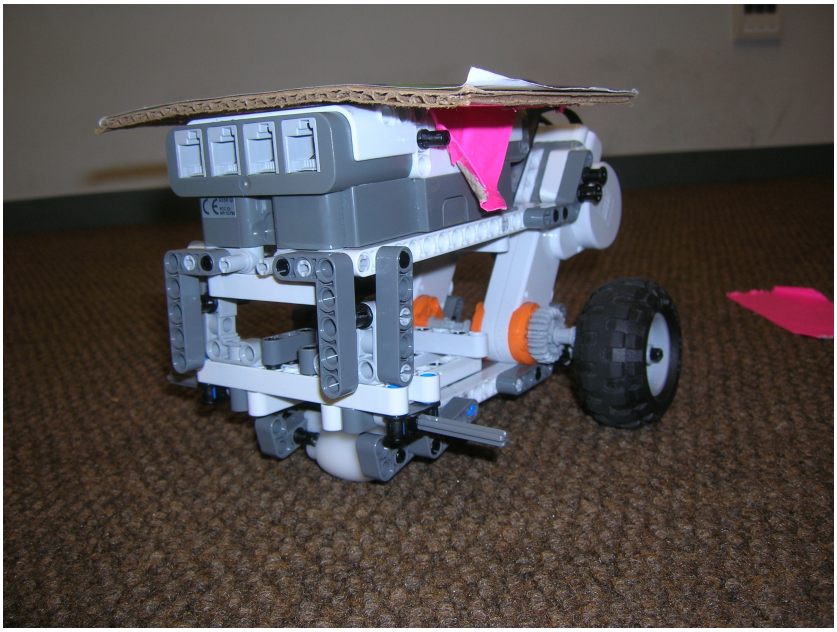
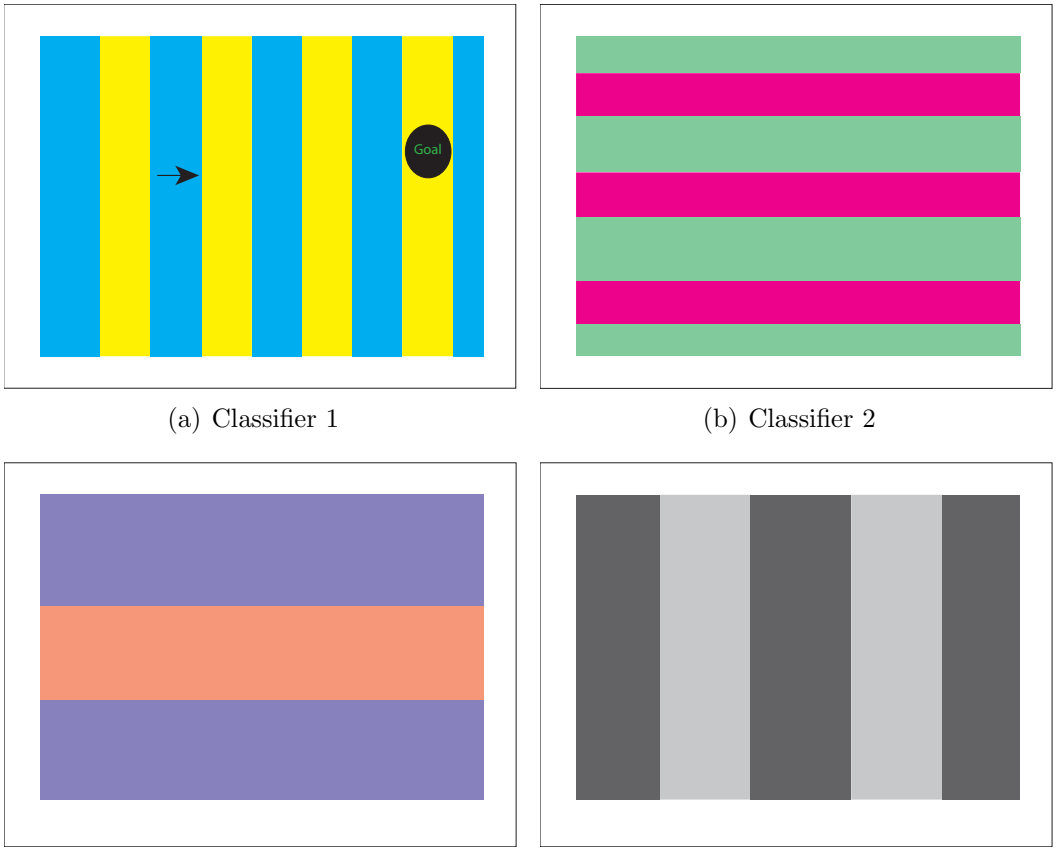


Figure 5.1: Image of the castered robot used in the artificial-dynamics environment.

5.4.1 Artificial-Dynamics Experiment

To demonstrate the importance of proper classification, we constructed an environment where the robot's dynamics greatly differed depending on one particular clustering of the state space. The agent was given four different classifiers and needed to identify the classifier that properly modeled the world. The first experiment included artificial variations in the robot's dynamics. All of the physical terrain that the robot traversed was identical, however, the mapping from action to outcome was different based on input from Classifier 1, as shown in Figure 5.2(a). When the agent is in a state labeled as blue, actions a_0, a_1, a_2, and a_3 result in the robot going left, right, forward and backward, respectively. In a state labeled as yellow, the resulting behaviors are right, left, backward and forward, respectively. With improper clustering of the environment, the agent is not able to navigate from the start state to the goal location which are labeled in Figure 5.2(a).

The robot, shown in Figure 5.1, was made up of two wheels and a caster. This



(a) Classifier 1

(b) Classifier 2

(c) Classifier 3

(d) Classifier 4.

Figure 5.2: Several different classifiers of the artificial-dynamics environment given to the SCRAM-R_{max} and RAM-R_{max} algorithms. (a)The actual classifier used to determine the robot's dynamics. The arrowhead indicates the start state of the agent and the ellipse indicates the goal location.(b),(c), and (d) show incorrect classifiers that were given as input to the different algorithms.

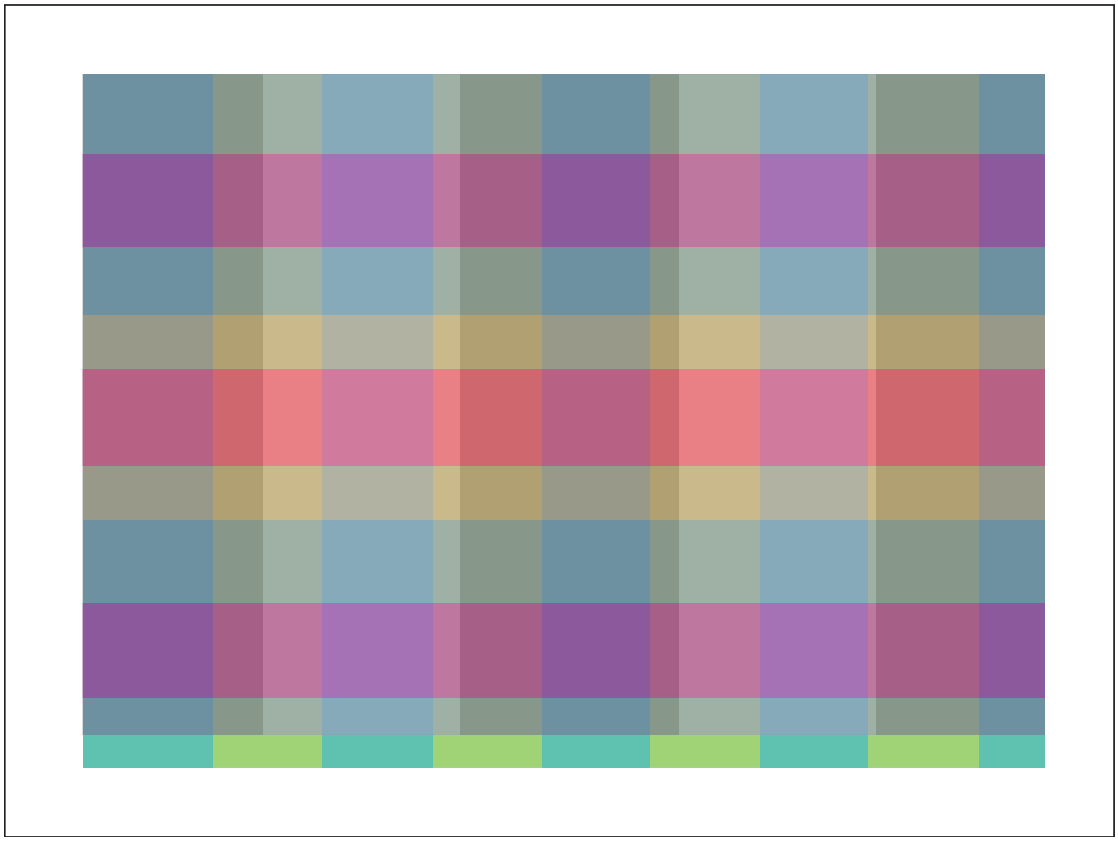


Figure 5.3: Combination of all features in Figure 5.2. Each individual color represents a separate cluster.

design was used to limit the friction during the turn actions.

Algorithms

For this experiment, we compared an agent running the SCRAM-R_{max} algorithm against three RAM-R_{max} agents. Each of the RAM-R_{max} agents had a different terrain classifiers as input. These classifiers consisted of Classifier 1, a four-stripe vertical classifier (Figure 5.2(a)); Classifier 2, a three-stripe horizontal classifier (Figure 5.2(b)); and a classifier that combined the all four of the given classifiers (Figure 5.3). All algorithms were given the goal location and had the parameters M and γ set to 10 and 1, respectively. The rewards were 1 for reaching the goal, -1 for going out of bounds, and -0.025 for every other state-action pair. Since a RAM-R_{max} agent with the incorrect model could run indefinitely, the episode

would "time-out" if the agent had seen more than M experiences in each type and took more than 50 steps. If a time-out occurred, the episode would terminate and the agent would receive a reward of -1.

Results

Figure 5.4 shows the various agents' performances. As expected, the RAM-R_{max} agent with the correct classifier (Classifier 1) performed the best, learning a seemingly optimal policy—receiving 0.75 for each episode. The SCRAM-R_{max} agent received the second best cumulative reward by converging to the same policy after learning that Classifier 1 was the most accurate clustering function. Next in performance was the RAM-R_{max} agent with the combined classifier. While its cumulative reward was at one point below -28, the agent eventually learned the optimal policy and began receiving positive rewards. The agent that performed the worst was the RAM-R_{max} with the incorrect classifier. This agent learned a very noisy dynamics model and was not able to reach the goal. In fact, without a proper dynamics model, the agent was often not able to go out of bounds to end the run, and frequently timed-out.

5.4.2 Real-Dynamics Environment

Experiment 1 detailed the pitfalls of improper classification. A second experiment was performed in a more realistic environment to show the ability of the SCRAM-R_{max} algorithm to learn the proper classification of real dynamics models. The environment for this experiment consisted of a combination of rock-embedded wax and carpet surfaces. As shown in Figure 5.5, half of each of the surfaces was colored blue and the other half was colored red. Only the surface texture, not the surface color, affected the robot's dynamics.

The start state is marked with an arrow and the goal area is marked with an ellipse. The agent's possible actions were: turn left, turn right, or go forward.

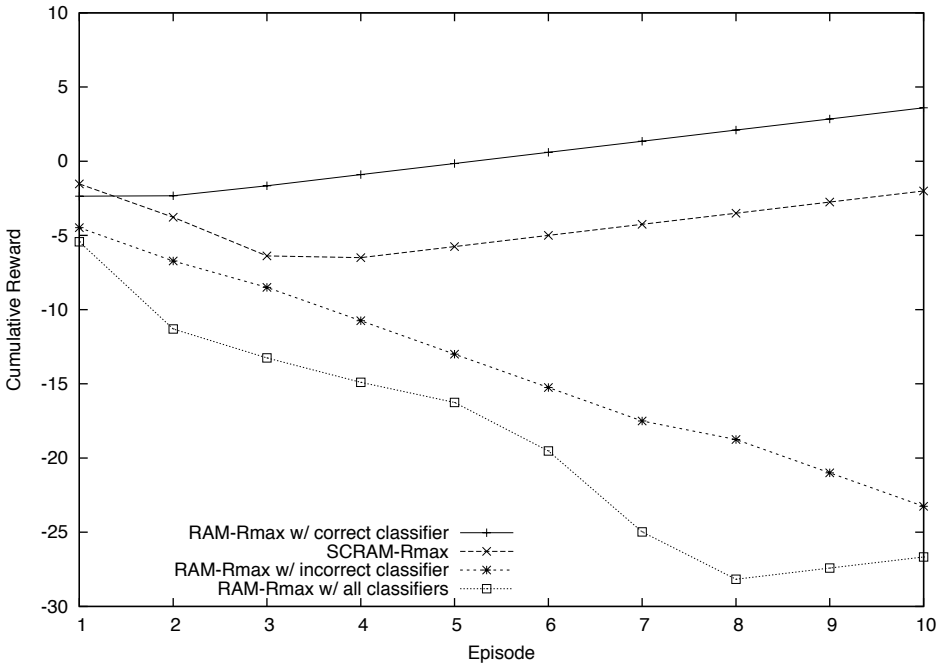


Figure 5.4: Cumulative Reward in the artificial-dynamics environment.

Figure 5.5: The real-dynamics environment consisting of rocky and carpet surfaces colored red and blue. The arrowhead indicates the starting position of the robot, and the ellipse marks the goal area.

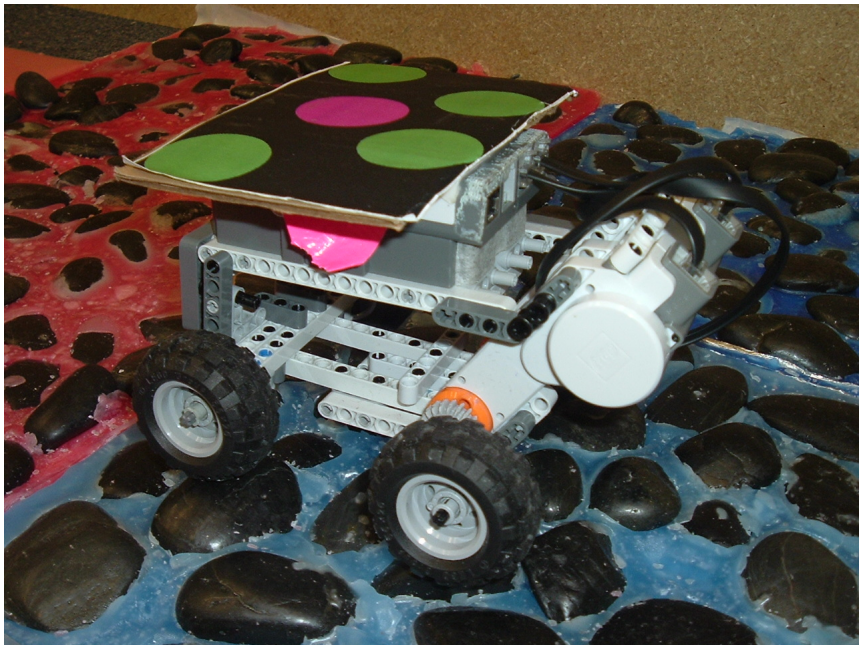


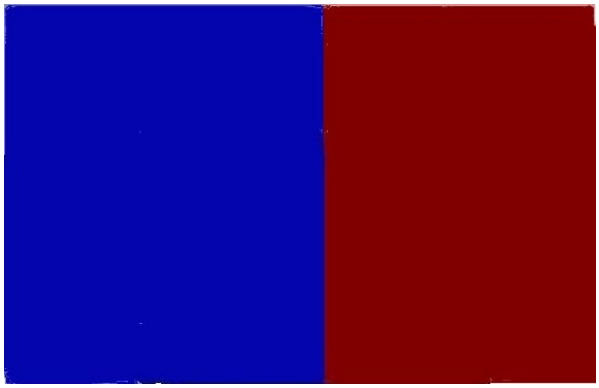
Figure 5.6: Image of the four-wheeled robot used in the real-dynamics environment.

Any area without color is considered to be out of bounds. If the agent were to enter such an area, the episode would end and the agent would be given a reward of -1. Reaching the goal also ended a run with a reward of 1. The reward for a state-action pair with any other resulting next state was 0.025.

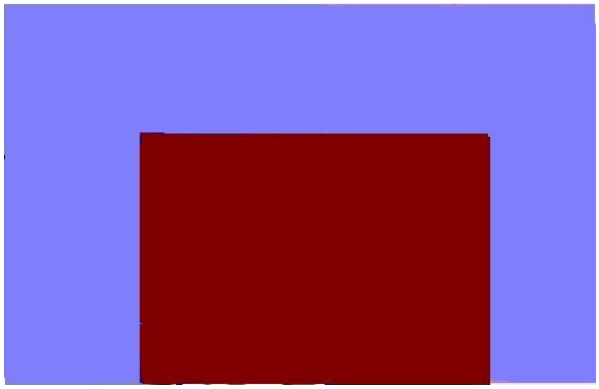
Figure 5.6 shows the robot used in the real-dynamics experiment. This robot has four wheels, where the back two wheels were powered individually.

Algorithms

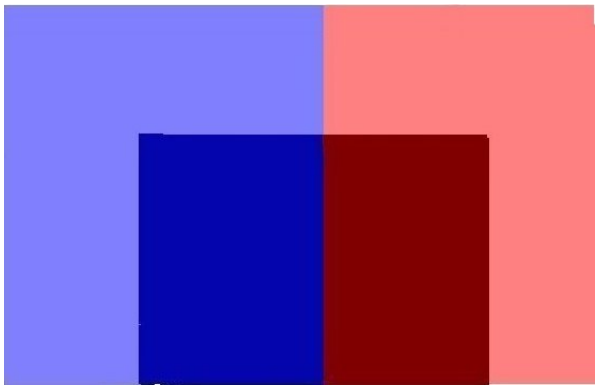
Since there are multiple features on which the terrain could be classified, we once again compared the SCRAM-R_{max} algorithm to several agents running the RAM-R_{max} algorithm with different inputs. Figure 5.7 shows the three different classifiers fed into the RAM-R_{max} algorithm—color, texture, and a combination of color and texture. SCRAM-R_{max} was given the color and texture classifiers to determine possible correlations with the robot's dynamics. The actual classifiers used were hand-tuned after being run through an image-segmentation program.



(a) Color segmentation of the environment



(b) Texture segmentation of the environment



(c) Color and texture segmentation of the environment

Figure 5.7: Classifiers fed into the RAM-R_{max} and SCRAM-R_{max} algorithms for the real-dynamics environment. From top to bottom: color-based, texture-based, and a combination of color-based and texture-based.

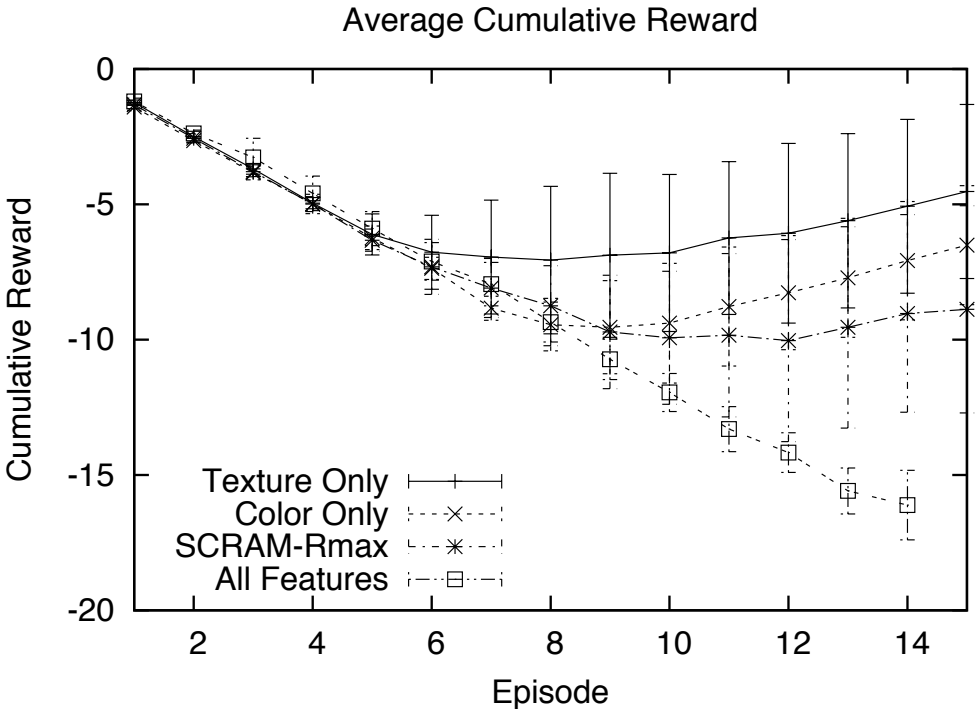


Figure 5.8: Cumulative Reward of the multiple agents in the real-dynamics environment

However, research in the vision community on color and texture segregation has shown that these classifiers could be obtained using automated techniques [Shi and Malik, 2000].

Results

Figure 5.8 shows the average cumulative reward received by each of the agents with error bars marking one standard deviation. The agents running the RAM-R_{max} algorithm with color and texture classifiers performed best due to the speed at which the agent learned the model. An accurate dynamics model was not necessary to perform the task reliably, so even the classifier based on color performed well. As a result, the difference in performance between the agents that used the color and texture classifiers was not statistically significant. The agent running the SCRAM-R_{max} algorithm learned that the texture classifier was the

most accurate classifier and reached the goal in fewer episodes than it took the RAM-R_{max} agent that assumed that all classifiers mattered.

The results of this experiment demonstrate that even if the clustering function needs to be learned, it is more efficient to learn that function than to assume that all information in the environment matters.

5.5 Discussion

The artificial-dynamics experiment demonstrated the benefits of correct classification when generalizing experience across state-space clusters. Improper clustering can lead to an agent being unable to perform its task. The performance of the SCRAM-R_{max} algorithm showed that with slightly more exploration, an agent can learn which classifier is associated with the robot's dynamics. This approach is shown to be more efficient than assuming that every classifier matters.

The real-dynamics experiment showed that the SCRAM-R_{max} algorithm can distinguish the more subtle differences in dynamics that are more likely to be present in the real world. However, an accurate dynamics model is not always necessary to perform a given task. SCRAM-R_{max} balances the cost of obtaining accurate models with the reward of efficient exploration.

5.6 Conclusion

This chapter has introduced a meteorologist-selection algorithm to determine accurate predictors. When combined with the RAM-R_{max} algorithm, the result is the Sensor Choosing for Relocatable Action Model-R_{max} (SCRAM-R_{max}) algorithm. This algorithm can take in multiple classifiers and learn which one most accurately clusters the states with similar dynamics.

Experiments showed the importance of learning proper classification when using SCRAM-R_{max} in comparison with several RAM-R_{max} agents with different

classifier inputs. SCRAM-R_{max} was also shown to perform more efficient exploration by learning important perceptual features instead of assuming that all features matter.

The contribution of this algorithm is the ability for an agent to learn the relocatable action model (RAM) Markov decision process (MDP) of an environment with multiple perceptual cues. The only domain knowledge that such an agent has is the assumption of continuous and isotropic states, allowing for the calculation of η.

Chapter 6
Conclusion

This dissertation has introduced the idea of using perception-based state-space clustering for learning the relocatable action model (RAM) Markov decision process (MDP) of an environment. Using this model, an agent can transfer experience between similar states to perform efficient exploration.

In Chapter 3, the RAM-R_{max} algorithm was introduced to demonstrate the benefits of generalization in the transition function. This algorithm was run in both the Sandbag and RockyRoad experimental environments and showed the level of generalization provided to the agent is important and can be determined using terrain classification.

Chapter 4 examined an agent's ability to learn proper clusterings from the reward function using the RAM-naïve algorithm. Experiments showed that in a limited number of runs, an agent can learn the latent structure of an environment with action-independent transitions and develop a near-optimal policy resulting in a large cumulative reward. These experiments also demonstrated the balance between extra exploration to learn the optimal policy exactly, and the quicker return of reward by settling for a near-optimal policy.

The last chapter outlined the issue of improper classification and proposed the SCRAM-R_{max} algorithm to solve this problem. This algorithm was compared to the RAM-R_{max} algorithm with several different inputs in multiple experiments to show both the need for proper classification and the speed with which it can be learned.

6.1 Future Extensions

The experimental results discussed have shown the large benefits of using a RAM MDP to limit the exploration process. Many of these environments were manufactured to demonstrate a specific point, but the motivation behind the examples are scalable to larger robotic domains. For example, a full-size car's dynamics may not vary greatly between a rocky surface and a paved one, but would vary if the vehicle were on an on incline or facing a wall. This section will present several future areas of research that can generalize the RAM family of algorithms to more domains.

In the majority of the experiments in this document, an overhead camera was used to obtain perceptual information. In many domains, such a camera is not available. For this reason, research into other perceptual channels is necessary to evaluate the use of relocatable action models for efficient exploration. Several sensors that can be used are distance sensors to detect objects in the path of the robot, an accelerometer to detect the pitch of the robot, a stereo-vision camera to detect surface texture, or surface mapping to detect whether the robot is on a sloped surface. For these sensors to be used for generalizing experience, some issues need to be addressed.

6.1.1 Anisotropic Worlds

Throughout this dissertation, the assumption was made that an agent's dynamics are the same in a particular x, y location independent of the robot's orientation. This assumption does not always hold (e.g. traveling uphill versus traveling downhill) and can lead to a poor model of the world. When a world is anisotropic—having different properties in different directions—the way in which states are clustered must be altered.

In order for the RAM family of algorithms to work in an anisotropic world,

states would have to be clustered not only based on location, but also orientation. For instance, an agent could use terrain classification to determine that there are two different types of locations. These clusters would have to be broken down further to include different orientations. Instead of having two clusters, there might be eight clusters—facing north, south, east, and west in each cluster.

The addition of more clusters would require more exploration, but a more accurate world model would be obtained. The approach detailed here is an intermediate way of handling the tradeoff between a detailed model of the environment and limited exploration.

6.1.2 Transitions Between Classes

For clustering based on perceptual information, this work assumed that a state belonged completely to one cluster. As discussed in Section 4.3, sometimes information is lost when performing this type of state-space clustering. An example of this loss of information is when a robot's back wheels are on a rocky surface and its front wheels are on a smooth surface. Assuming that this state is the same as either an all smooth or all rocky surface will lead to error in the learned dynamics model and, in the case of SCRAM-R_{max}, can lead to error in learning the classifier.

One way to improve upon our method is to consider additional classes of states that are in transitional locations; this prevent the loss of important information. However, adding this level of precision would affect the amount of necessary exploration which, depending on the task, might not be worth the cost.

6.1.3 Perceptual Information Obtained During Trial

In each of our experiments, the agent had all the perceptual information at the start of the experiment. This information allowed for more efficient exploration,

but is not entirely practical. An agent may not be able to obtain a complete map of the world before beginning exploration, but it can still generalize experience between states as perceptual information becomes available.

For instance, an agent that does not know in advance the location of walls in the environment can still learn about their effects and perceptual cues, thereby allowing for the transfer of experience to future states where walls are perceived.

To utilize perceptual information not available prior to the start of a run during planning, an agent can initialize its sensor to either an optimistic or "I don't know" value. If the initialization is optimistic, the agent's planner can perform value iteration and planning as normal assuming that there are no walls, and update its model when it encounters a wall. This initialization will allow the agent to focus more on traveling directly to the goal instead of calculating probabilities and planning around where walls *might* be.

If the agent initializes the sensor values of all states to "I don't know", on the other hand, the agent can calculate the expected value for each of the possible dynamics models for the state in question and choose the highest possible value. This process allows for the agent to consider different configurations of the world and plan for the best possible reward.

Both of these ways of initializing the sensors allows for optimism in the agent's planning and will result in more efficient exploration than if the agent explored the environment to learn the actual sensor values. The difference in the two approaches is the amount of planning and calculation that the agent performs. The cost of calculation, and, therefore the better approach, is dependent on the domain.

6.1.4 Knowledge Transfer

As expressed throughout this document, exploration is a large expense in robot domains, and, therefore, to learn a robot's dynamics model every time a new task

is presented is burdensome and impractical. One solution to this problem is to transfer a robot's dynamics model between tasks. Assuming that the perceptual data is consistent between tasks, an agent would be able to store the perceptual data from the initial task, and perform state-space clustering across tasks. This approach would allow for an agent to perform entirely new tasks using previously experienced outcomes. Thus, "transferring" acquired knowledge between tasks.

For instance, if an agent uses color-based terrain classification, it can learn about black surfaces in one task and store the information. Then, if in a second task the agent encounters pavement, it can use experience from the first task to calculate probable next states. The difficulty introduced by doing this would be the potential need to re-cluster the perceptual data. This situation could occur, for example, if the second task were actually a finer tuned clustering than the first task, differentiating between gravel and pavement. Without storing all of the data from the first task, the agent would not be able separate the previous experiences based on the new clusters.

6.1.5 Continuous Domains

All of the experiments presented in this dissertation have been in discrete environments. Brunskill et al. [2008] created the CORL algorithm, which adapted the RAM-R_{max} algorithm to continuous domains. To do so, it modeled outcomes as Gaussian distributions. This solution not only allowed for efficient learning in continuous domains, it had the added bonus that the agent's calculation time remained constant with the amount of experience.

6.2 Summary

This dissertation has shown that relocatable action models with state-space clustering determined by perceptual information enables efficient exploration in robot-navigation tasks. The RAM-R_{max} algorithm was introduced to demonstrate the benefits of using a relocatable action model (RAM) Markov decision process (MDP) to model robot-navigation environments using terrain classification to cluster states with similar dynamics models. Then, experiments showed that by feeding the reward function into a hierarchical clustering algorithm, an agent can learn the clustering function for the RAM-naïve algorithm and still perform more efficient exploration than algorithms that assumed no similarities in the state space. Finally, the SCRAM-R_{max} algorithm showed that an agent can choose from a set of classifiers to accurately cluster the states with similar dynamics with limited exploration. Experiments showed that each of these algorithms performed better in the given domains than current state-of-the-art algorithms. Extensions of this work were also proposed to adapt these algorithms to be applicable in more robot-navigation domains.

Appendix A
State-space Aggregation versus RAM MDP

As mentioned in Chapter 1, relocatable action model (RAM) Markov decision processes (MDP) allow for generalization while keeping the a complete model of the environment. For this reason, a RAM MDP can be used to perform state-space aggregation as shown in Section A.1. However, not all generalization performed by a RAM MDP can be done by state-space aggregation. An example of this is shown in Section A.2.

A.1 State-space aggregation using a RAM MDP

To examine how a RAM MDP can be used to perform state-space aggregation, let us re-examine the MDP of the robot-navigation task from Chapter 1, shown in Figure A.1 where states s_5, s_{10}, s_{15}, s_{20}, and s_{25} are goal states. The aggregate model is shown in Figure A.2.

For clarity, the mapping from states in the original model, to states in the aggregate model can be called the clustering function, ξ. This clustering function can be used as the type function, κ, in the RAM MDP formalism. To represent this function graphically in the RAM MDP, each state in the aggregate model has been given a color. These color are shown in the RAM MDP in Figure A.3.

An aggregate model also contains information about transitions, T, in the form of

$$T(c, a) = c'.$$

where c' is the probability distribution over possible next clusters and a is

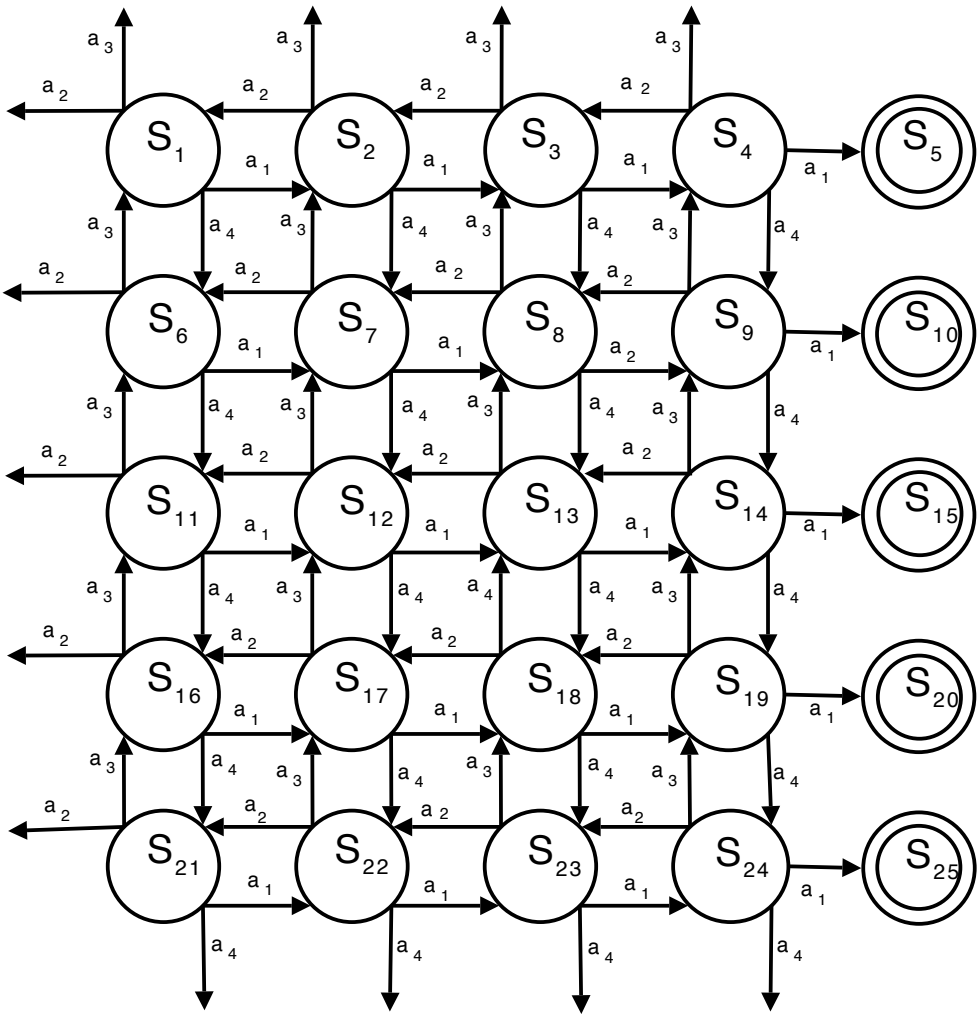


Figure A.1: A Markov Decision Process (MDP) model for a robot navigation task where states s_5, s_{10}, s_{15}, s_{20}, and s_{25} are goal states.

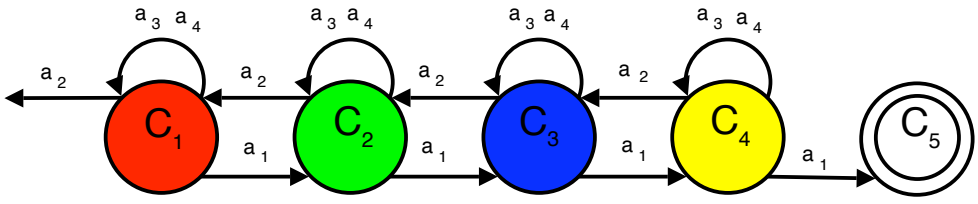


Figure A.2: An aggregate model of the environment modeled in Figure A.1. Each state represents a cluster of states with similar values in the original model.

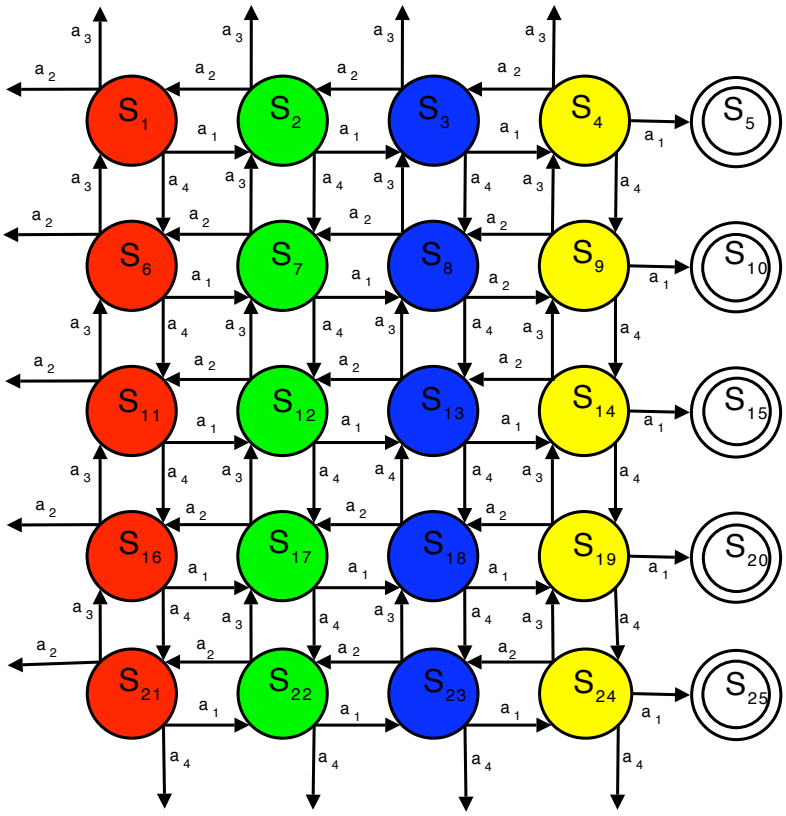


Figure A.3: A RAM MDP representing the state-space aggregation shown in Figure A.2.

the action taken. However, the since the agent running state-space aggregation is given information from the environment in terms of the original model, the transitions learned are actually in the form of

$$T(\xi(s), a) = T(c, a) = c'.$$

Of the three functions in the RAM formalism, the type function, κ $(S \rightarrow C)$; the relocatable action model, t $(C \times A \rightarrow O)$; and the next-state function, η $(S \times O \rightarrow S')$, there is not a function that directly maps cluster-action pairs to next clusters, this function would have to be calculated by

$$T(c, a) = \kappa(\eta(s, t(c, a))).$$

?? shows the steps to solve for c' for taking action a_1 from state s_7 where o_1 is an outcome that represents a change in the x dimension by $+1$. While solving the RAM MDP takes more calculations, the amount of exploration necessary to learn the model is the same as using state-space aggregation which, in the robot-navigation domain, is more costly than calculations.

$$\begin{aligned} T(\xi(s_7), a_1) &= \kappa(\eta(s_7, t(\kappa(s_7), a_1))) \\ &= \kappa(\eta(s_7, t(c_2, a_1))) \\ &= \kappa(\eta(s_7, o_1)) \\ &= \kappa(s_8) \\ &= c_3 \end{aligned}$$

A.2 Representing a RAM MDP with aggregation

To examine how an aggregate model cannot fully incorporate all of the information of a RAM MDP, let us re-examine the RAM MDP of the robot-navigation task from Chapter 1, shown in Figure A.4 where state s_{25} is a goal state. An aggregate

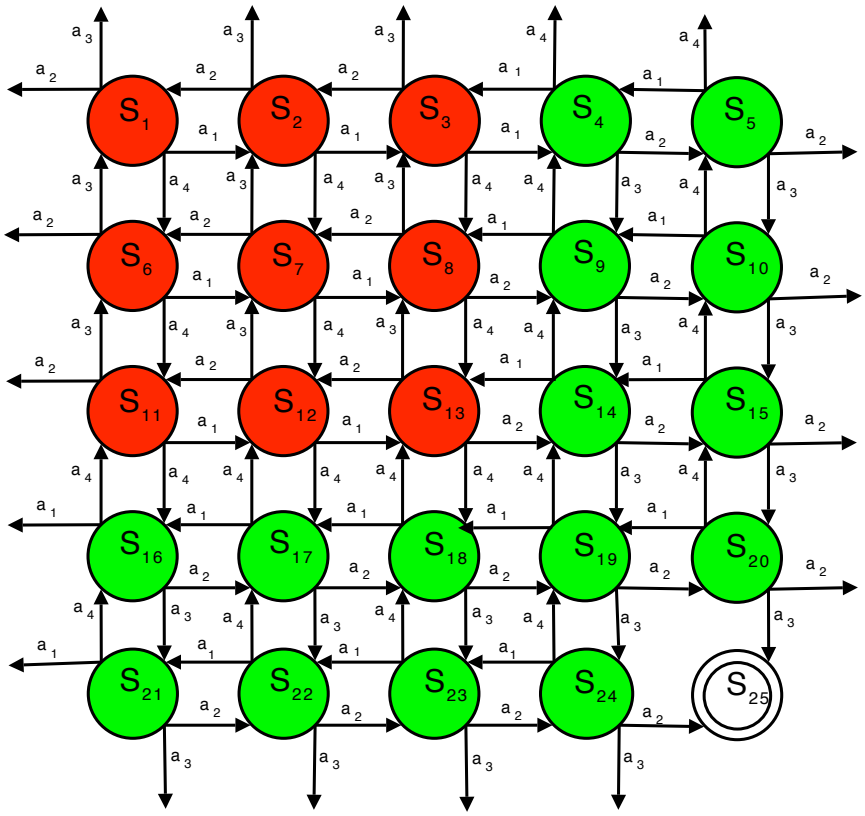


Figure A.4: A RAM MDP model of a two-dimensional robot-navigation environment with two clusters. State s_{25} is a goal state

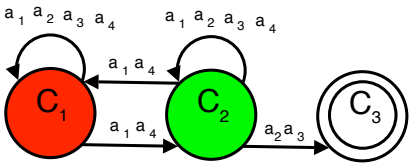


Figure A.5: An aggregate model of the environment modeled in Figure A.4. The type function of the RAM MDP was used to perform clustering.

model using the type function of the RAM MDP as the cluster function is shown in Figure A.5.

To solve for the next state, s', in a RAM MDP an agent would use the equation

$$\eta(s, t(\kappa(s), a)) = s'.$$

So, for a deterministic environment like Figure A.4, solving includes calculating the next state from taking action a_2 from state s_{13}.

$$\begin{aligned} s' &= \eta(s_{13}, t(\kappa(s_{13}), a_2)) \\ &= \eta(s_{13}, t(c_1, a_2)) \\ &= \eta(s_{13}, o_1)) \\ &= s_{14} \end{aligned}$$

For this two dimensional environment, outcome o_1 could be a change in the x dimension of $+1$. To solve for s' in the aggregate model, the agent would have to perform

$$\begin{aligned} c' &= T(\xi(s_{13}, a_2)) \\ &= T(c_1, a_2) \\ &= \{c_1, c_2\} \end{aligned}$$

which actually tells the agent very little about its resulting state. This problem with state-space aggregation is due to its over-generalization of the environment. By creating a second, paired down model of the environment, vital information is lost. The benefit of the RAM MDP is that the agent maintains all information about the original environment and is still able to generalize experience across states.

Appendix B
Tested Surfaces

The materials used for surfaces in the experiments in this dissertation were chosen after much experimentation. Below is a list of surface materials tested to alter the robot's dynamics.

Crushed walnut shells

To mimic the performance of sand without getting sand in the robot's gears, I tried a type of reptile litter made out of crushed walnut shells. This material shifted too easily. The robot sunk below the surface and was not able to move in the environment.

To keep the robot from sinking, I tried putting these walnut shells in a large plastic sealable bag. However, the shiny surface led to difficulties with the reflection-based tracking system. So, a felt bag was used instead (as mentioned in Chapter 3). This bag performed well, allowing the sand to shift without letting the robot sink below the surface. One problem with this surface, though, was introduced after several runs were performed. The more the robot travelled along a path, the more the sand would sink in that area. Every few runs, the sand would have to be shifted by hand to counteract this problem.

Shredded paper

To limit the robot's traction, I tried another type of animal litter made out of shredded paper. This material also shifted too easily. The robot sunk below the

surface and was not able to move in the environment.

Rocks

Another surface that I tried was small polished rocks in the hope of having a bumpy surface without the robot sinking. Unfortunately, the individual rocks shifted too much and sometimes led to the robot getting stuck. Since we wanted to have a uniform dynamics model for all parts of the environment made of the same material, having a situation where the robot could not move was not desirable.

Bubble wrap

To mimic the bumps of rocks without the slippage, I tried running the robot on bubble wrap. This material, though, did not significantly differ the robot's dynamics in comparison to a smooth surface.

Grass

Since idea of a multi-surface environment came from the real world, I tried running the robot on the grass outside of the CoRE building at Rutgers University. The robot, though, was too small for this environment and the undercarriage of the robot dragged on the grass.

Rocks embedded in wax

After trying the robot outside, I found one surface that seemed to have the desired effect on the robot's dynamics was rocks that had been embedded in dirt. The variation of the rocks slowed down the robot and since the rocks were stationary, the robot never actually got stuck. To mimic this surface, I created a surface of rocks embedded in candle wax that was used in Chapters 3 and 5. This surface allowed for the variability of rocks without the material slipping under the robot's

wheels. One difficulty of this surface was its fragility. Often transporting these surfaces led to breakage.

Wax

Once wax was used for the bumpy surface, I tried using wax for a smooth, possibly even slippery surface. However, the robot's dynamics on the wax were the same as its dynamics on carpet. Since the wax was so fragile and the carpet was in abundance, I opted not to use plain wax surfaces.

Carpet

One problem with using the carpet floor of our laboratory in the same environment as the rocks embedded in wax was the border between the two surfaces. The rocky surface was about two centimeters above the ground surface. Therefore the transitions between that surface and the ground were usually quite abrupt. To minimize these transitions, I used additional carpet squares on top of the laboratory floor to even the heights of the two surfaces. This was the configuration of both the RockyRoad environment from Chapter 3 and the real-dynamics environment in Chapter 5. In the real-dynamics environment, though, the carpet squares were covered with a laminated construction paper to allow for the same surface to have different visual properties.

Bibliography

Ethem Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004. 27, 28

Christopher G. Atkeson and Stefan Schaal. Robot learning from demonstration. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 12–20. 1997. 33

James Bagnell and Jeff Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the IEEE International Conference on Robotics and Automation*. May 2001. 32

J. Balaram. Kinematic state estimation for a mars rover. *Robotica*, 18(3):251–262, 2000. 1

James C. Bean, John R. Birge, and Robert L. Smith. Aggregation in dynamic programming. *Operations Research*, 35(2):215–220, 1987. 36

Darrin C. Bentivegna and Christopher G. Atkeson. Learning from observation using primitives. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1988–1993. 2001. 29

Craig Boutilier and Richard Dearden. Using abstractions for decision-theoretic planning with time constraints. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1016–1022. 1994. 18

Ronen I. Brafman and Moshe Tennenholtz. R-MAX—A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2002. 34, 44, 63

James Bruce, Tucker Balch, and Manuela Veloso. Fast and inexpensive color image segmentation for interactive robots. In *Proceedings of International Conference on Intelligent Robots and Systems*. Japan, October 2000. 51

James Bruce and Manuela Veloso. Fast and accurate vision-based pattern detection and identification. In *Proceedings of the IEEE International Conference on Robotics and Automation*. Taiwan, May 2003. 51

Emma Brunskill, Bethany R. Leffler, Lihong Li, Michael L. Littman, and Nicholas Roy. CORL: A continuous-state offset-dynamics reinforcement learner. In *Proceedings of the Twenty Fourth Conference on Uncertainty in Artificial Intelligence*. 2008. 42, 89

David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. pages 726–731. 1991. 60

Christopher M. Christoudias, Bogdan Georgescu, and Peter Meer. Synergism in low level vision. In *Proceedings of the Sixteenth International Conference on Pattern Recognition*, volume 4. Washington, DC, USA, 2002. 40, 41, 50

J. Crisman and Chuck Thorpe. UNSCARF, a color vision system for the detection of unstructured roads. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 3, pages 2496–2501. April 1991. 27

H. Dahlkamp, A. Kaehler, D. Stavens, S. Thrun, and G. Bradski. Self-supervised monocular road detection in desert terrain. In *Proceedings of Robotics: Science and Systems*. Philadelphia, USA, August 2006. 30

Thomas Dean and Robert Givan. Model minimization in Markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 106–111. 1997. 18, 36

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, 39(1):1–38, 1977. 69

Carlos Diuk, Alexander L. Strehl, and Michael L. Littman. A hierarchical approach to efficient reinforcement learning in deterministic domains. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 313–319. 2006. 35

Kenji Doya. *Neural Computation*, 12(1):219–245, 2000. 3

Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005. 35, 54, 56

Philip W. L. Fong. A quantitative study of hypothesis selection. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 226–234. 1995. 62

Robert Givan, Thomas Dean, and Matthew Greig. Equivalence notions and model minimization in markov decision processes. *Artificial Intelligence*, 147(1-2):163–223, 2003. 60

Geoffrey Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*. 1995. 17

C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*. 2003. 35

C. Guestrin, R. Patrascu, and D. Schuurmans. Algorithm-directed exploration for model-based reinforcement learning in factored MDPs. In *Proceedings of the International Conference on Machine Learning*. 2002. 37

Lawrence Hubert. Approximate evaluation techniques for the single-link and complete-link hierarchical clustering procedures. *Journal of the American Statistical Association*, 69(347):698–704, 1974. 64

Nicholas K. Jong and Peter Stone. Model-based exploration in continuous state spaces. In *Proceedings of the Seventh Symposium on Abstraction, Reformulation, and Approximation*. July 2007. 38

S. Kakade, M. Kearns, and J. Langford. Exploration in metric state spaces. In *Proceedings of the Twentieth International Conference on Machine Learning*. 2003. 38

Michael Kearns and Satinder Singh. Near-optimal reinforcement learning in polynomial time. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 260–268. 1998. 34

Michael J. Kearns and Daphne Koller. Efficient reinforcement learning in factored MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 740–747. 1999. 37

Terran Lane and Andrew Wilson. Toward a topological theory of relational reinforcement learning for navigation tasks. In *Proceedings of the Florida Artificial Intelligence Research Society Conference*, pages 461–467. 2005. 37

S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. 7

Bethany R. Leffler, Michael L. Littman, and Timothy Edmunds. Efficient reinforcement learning with relocatable action models. In *Proceedings of the Twenty Second Conference on Artificial Intelligence*, pages 572–577. The AAAI Press, Menlo Park, CA, USA, 2007. iv, 48

Bethany R. Leffler, Michael L. Littman, Alexander L. Strehl, and Thomas J. Walsh. Efficient exploration with latent structure. In *Proceedings of Robotics: Science and Systems*. Cambridge, USA, June 2005. iv, 62, 69

Bethany R. Leffler, Christopher R. Mansley, and Michael L. Littman. Efficient learning of dynamics models using terrain classification. In *Proceedings of the International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems*. 2008. iv

Lihong Li, Michael L. Littman, and Thomas J. Walsh. Knows what it knows: A framework for self-aware learning. In *Proceedings of the Twenty Fifth International Conference on Machine Learning*. 2008. 71

Lihong Li, Thomas J. Walsh, and Michael L. Littman. Towards a unified theory of state abstractions for mdps. In *In Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, pages 531–539. 2006. 18

Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2):311–365, 1992. 31

T. M. Mitchell. The discipline of machine learning. Technical Report CMU-ML-06-108, Machine Learning Department, Carnegie Mellon University, July 2006. 2

Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. 28

Andrew W. Moore. Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces. In *Proceedings of the Eighth International Conference on Machine Learning*. 1991. 36

Justin Boyan Andrew Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Neural Information Processing Systems 7*, pages 369–376. Cambridge, MA, 1995. 17

Richard M. Murray, S. Shankar Sastry, and Li Zexiang. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Inc., Boca Raton, FL, 1994. 1

Andrew Y. Ng, H. Jin Kim, Michael I. Jordan, and Shankar Sastry. Autonomous helicopter flight via reinforcement learning. In *Advances in Neural Information Processing Systems 16*. Cambridge, MA, 2004. 33

Dean A. Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems 1*, pages 305–313. San Francisco, CA, USA, 1989. 28

Martin L. Puterman. *Markov Decision Processes—Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, 1994. 6, 7

Christopher Rasmussen. Combining laser range, color, and texture cues for autonomous road following. In *Proceedings of the IEEE International Conference on Robotics and Automation*. 2002. 29

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003. 27

Alexander A. Sherstov and Peter Stone. Improving action selection in MDP's via knowledge transfer. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*. Pittsburgh, USA, 2005. 21

Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 2000. 82

Satinder P. Singh, Tommi Jaakkola, and Michael I. Jordan. Reinforcement learning with soft state aggregation. In *Advances in Neural Information Processing Systems*, volume 7, pages 361–368. 1995. 36

Daniel Stronger and Peter Stone. Simultaneous calibration of action and sensor models on a mobile robot. In *Proceedings of the IEEE International Symposium on Robotics and Automation*. August 2004. 28

Richard S. Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3(1):9–44, 1988. 16

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. 1996. 35

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. 3, 4, 5, 31

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995. 35

S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Probabilistic algorithms and the interactive museum tour-guide robot MINERVA. *International Journal of Robotics Research*, 19(11):972–999, 2000. 2

Sebastian Thrun. The role of exploration in learning control. In *Handbook for Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Florence, Kentucky, 1992. 15

Sebastian Thrun and Joseph O'Sullivan. Discovering structure in multiple learning tasks: The TC algorithm. In *Proceedings of the International Conference on Machine Learning*, pages 489–497. 1996. 35

Richard Volpe, Tara Estlin, and Sharon Laubach. Enhanced mars rover navigation techniques. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 926–931. 2000. 1

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992. 31, 48

Vita

Bethany R. Leffler

1998–2002 B. Sc. from Rowan University

2006–08 NSF Integrated Graduate Education and Research Trainee (IGERT), Department of Computer Science, Rutgers University

2004–06 Teaching assistant, Department of Computer Science, Rutgers University

2002–04 Developmental Computer Scientist, Titan Systems Corporation

Publications

Emma Brunskill, Bethany R. Leffler, Lihong Li, Michael L. Littman, and Nicholas Roy. CORL: A Continuous-State Offset-Dynamics Reinforcement Learner. In *Proceedings of the Twenty Fourth Conference on Uncertainty in Artificial Intelligence*. 2008.

Bethany R. Leffler, Christopher R. Mansley, and Michael L. Littman. Efficient Learning of Dynamics Models using Terrain Classification. In *Proceedings of the International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems*. 2008.

Bethany R. Leffler, Michael L. Littman, and Timothy Edmunds. Efficient Reinforcement Learning with Relocatable Action Models. In *Proceedings of the Twenty Second Conference on Artificial Intelligence*, pages 572–577. The AAAI Press, Menlo Park, CA, USA, 2007.

Bethany R. Leffler, Michael L. Littman, Alexander L. Strehl, and Thomas J. Walsh. Efficient Exploration With Latent Structure. In *Proceedings of Robotics: Science and Systems*. Cambridge, USA, June 2005.