Copyright

by

Jefferson Provost

2007

The Dissertation Committee for Jefferson Provost
certifies that this is the approved version of the following dissertation:

Reinforcement Learning In High-Diameter, Continuous Environments

Committee:

Benjamin J. Kuipers, Supervisor

Risto Miikkulainen, Supervisor

Raymond Mooney

Bruce Porter

Peter Stone

Brian Stankiewicz

Reinforcement Learning In High-Diameter, Continuous Environments

by

Jefferson Provost, B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2007

Acknowledgments

A couple of years ago someone forwarded me a link to an article interpreting Frodo's quest in The Lord of the Rings as an allegory for writing and filing a dissertation (Lee, 2005). At the time, I saw this as nerdy humor for PhD-seeking Tolkien geeks like myself. But when I read the article again last week, it made me embarrasingly emotional. I won't belabor this page with the details, to spare you the boredom, and to spare myself any more of the ridicule that Pat, Laura, Foster, Pic, and Todd surely all have waiting for me after reading that. (What else are friends for, after all?) I'll only say this: I think I had much more fun than Frodo ever did (except maybe just before the end, where we're about even), and my fellowship of advisors, friends, and family was far larger and richer than Frodo's.

My first thanks go to my advisors and committee. Ben Kuipers and Risto Miikkulainen were simultaneously the most unlikely and yet most complementary co-advisors I could imagine. Each provided his own form of guidance, rigor, inspiration, and prodding without which my education would be incomplete. Ray Mooney, Bruce Porter, Peter Stone, and Brian Stankiewicz all were there over the years with inspiration, advice, ideas, and sometimes even beer.

Pat Beeson, Joseph Modayil, Jim Bednar and Harold Chaput were my comrades in the trenches at the UT AI Lab—all just as quick with a laugh as a good idea. Along with Tal Tversky, Ken Stanley, Aniket Murarka, Ram Ramamoorthy, Misha Bilenko, Prem Melville, Nick Jong, Sugato Basu, Shimon Whiteson, Bobby Bryant, Tino Gomez, and everyone else... It's hard to imagine another collection of great minds who would also be so much fun to be around.

My brother Foster was my secret weapon through the years, providing advice from the perspective of a professor, but intended for a peer. And without him, who would Pat and I have to drink with in a red state on election night? ...all that, and he let me freeload in his hotel rooms at conferences for years. Thanks, man!

To Pic, Todd, Rob, Rachel, Sean, Meghan, Ezra, Christy, and all my other non-UT friends, it's great to know that there's life and laughs outside academia. Thanks for sticking with me.

Last, first and always are Laura and Maggie. Laura trooped along with me for all these years, only occasionally asking me what the heck I was doing and why the **** it was taking so long. Well here it is. I love you. Maggie, some day you'll read all this nonsense and say, "Are you kiddin' me?!" And you won't have any idea why your mom and dad think that's so hilarious. I love you, bucket!

JEFF PROVOST

The University of Texas at Austin
August 2007

Reinforcement Learning In High-Diameter, Continuous Environments

Publication No. ____________________

Jefferson Provost, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Benjamin J. Kuipers

Many important real-world robotic tasks have *high diameter*, that is, their solution requires a large number of primitive actions by the robot. For example, they may require navigating to distant locations using primitive motor control commands. In addition, modern robots are endowed with rich, high-dimensional sensory systems, providing measurements of a continuous environment. Reinforcement learning (RL) has shown promise as a method for automatic learning of robot behavior, but current methods work best on low-diameter, low-dimensional tasks. Because of this problem, the success of RL on real-world tasks still depends on human analysis of the robot, environment, and task to provide a useful set of perceptual features and an appropriate decomposition of the task into subtasks.

This thesis presents Self-Organizing Distinctive-state Abstraction (SODA) as a solution to this problem. Using SODA a robot with little prior knowledge of its sensorimotor system, environment, and task can automatically reduce the effective diameter of its tasks. First it uses a self-organizing feature map to learn higher level perceptual features while exploring using primitive, local actions. Then, using the learned features as input, it learns a set of high-level actions that carry the robot between *perceptually distinctive states* in the environment.

Experiments in two robot navigation environments demonstrate that SODA learns useful features and high-level actions, that using these new actions dramatically speeds up learning for high-diameter navigation tasks, and that the method scales to large (building-sized) robot environments. These experiments demonstrate SODAs effectiveness as a generic learning agent for mobile robot navigation, pointing the way toward *developmental robots* that learn to understand themselves and their environments through experience in the world, reducing the need for human engineering for each new robotic application.

Contents

Acknowledgments **iv**

Abstract **vi**

Contents **viii**

List of Tables **xi**

List of Figures **xiii**

Chapter 1 Introduction **1**

1.1 Motivation 1
1.2 Reinforcement Learning in Realistic Robots 3
1.3 Learning Without Prior Knowledge 5
1.4 Bootstrapping to Higher Levels of Representation 7
1.5 Approach 7
1.6 Overview of the Dissertation 10

Chapter 2 Background and Related Work **12**

2.1 The Building Blocks of SODA 13
  2.1.1 The Spatial Semantic Hierarchy (SSH) 13
  2.1.2 Reinforcement Learning 16

2.1.3 Self-Organizing Maps (SOMs) 26
2.2 Bootstrap Learning 32
2.2.1 Drescher's Schema System 33
2.2.2 Constructivist Learning Architecture 34
2.2.3 Bootstrap Learning for Place Recognition 35
2.2.4 Learning an Object Ontology with OPAL 36
2.2.5 Other methods of temporal abstraction 37
2.3 Automatic Feature Construction for Reinforcement Learning 38
2.3.1 U-Tree Algorithm 39
2.3.2 Backpropagation 40
2.4 SODA and Tabula Rasa Learning 41
2.5 Conclusion 42

Chapter 3 Learning Method and Representation **44**
3.1 Overview 44
3.2 Top^n State Representation 46
3.3 Trajectory Following 48
3.4 Hill-climbing 49
3.4.1 HC using Gradient Approximation 50
3.4.2 Learning to Hill-Climb with RL 53
3.5 Conclusion 56

Chapter 4 Learning Perceptual Features **58**
4.1 Experimental Setup 58
4.2 T-Maze Environment 61
4.3 ACES Fourth Floor Environment 64
4.4 Feature Discussion 66

Chapter 5 Learning High-Level Actions **70**
5.1 Trajectory Following 71
5.2 Hill Climbing 76
  5.2.1 Hand-coded Predictive Action Model 76
  5.2.2 Hill-Climbing Experiment 79
  5.2.3 HC Learning Discussion 83
5.3 Action Learning Conclusion 85

Chapter 6 Learning High-Diameter Navigation **86**
6.1 Learning in the T-Maze 87
6.2 The Role of Hill-Climbing 92
6.3 Scaling Up The Environment: ACES 4 97
6.4 Navigation Discussion 98
6.5 Navigation Conclusion 101

Chapter 7 Discussion and Future Directions **102**
7.1 SODA with Perceptual Aliasing 103
7.2 Dealing with Turns 107
  7.2.1 Alternative Distance Metrics for Feature Learning 108
  7.2.2 Learning Places, Paths, and Gateways 109
7.3 Scaling to More Complex Sensorimotor Systems 112
7.4 Prior Knowledge and Self-Calibration 114
7.5 Discussion Summary 117

Chapter 8 Summary and Conclusion **118**

Bibliography **121**

Vita **131**

List of Tables

3.1 Open-loop and Closed-loop Trajectory-following. *Top:* The open-loop trajectory-following macro repeats the same action until the current SOM winner changes. *Bottom:* The closed-loop Option policy chooses the action with the highest value from the Option's action set. The action set, defined in Equation (3.7) is constructed to force the agent to make progress in the direction of a_i while being able to make small orthogonal course adjustments. 47

3.2 Pseudo-code for hill-climbing policy π_i^{HC} using gradient estimation. The value of G_{ij} is the estimated change in feature f_i with respect to primitive action a_j^0. G_{ij} can be determined either by sampling the change induced by each action or by using an action model to predict the change. Sampling is simple and requires no knowledge of the robot's sensorimotor system or environment dynamics, but is expensive, requiring $2|\mathcal{A}^0|-2$ sampling steps for each movement 51

3.3 Summary of the components of SODA. 55

4.1 T-Maze Abstract Motor Interface and Primitive Actions 61

4.2 ACES Abstract Motor Interface and Primitive Actions 66

5.1 **The average trajectory length for open loop vs. learned trajectory-following.** Learned TF options produce longer trajectories. 75

List of Figures

1.1 **The Reinforcement Learning Problem Framework** In reinforcement learning problems an agent interacts with an environment by performing actions and receiving sensations and a reward signal. The agent must learn to act so as to maximize future reward. 4

1.2 **The SODA Architecture** This expanded version of the diagram in Figure 1.1 shows the internal structure of the SODA agent (For the purposes of this description, the physical robot is considered part of the environment, and not the agent). The agent receives from the environment a sensation in the form of a continuous *sense vector*. This vector is passed to a *self-organizing feature map* that learns a set of high-level perceptual features. These perceptual features and the scalar *reward* passed from the environment are fed as input to a *learned policy* that is updated using reinforcement learning. This policy makes an *action selection* from among a set of high-level actions consisting of combinations of learned *trajectory-following (TF)* and *hill-climbing (HC)* control laws. These high-level actions generate sequences of *primitive actions* that take the form of continuous *motor vectors* sent from the agent to the robot. 8

2.1 **Continuous-to-Discrete Abstraction of Action** In the SSH Control Level (top), continuous action takes the form of hill-climbing control laws, that move the robot to perceptually distinctive states, and trajectory-following control laws, that carry the robot from one distinctive state into the neighborhood of another. In the Causal Level, the trajectory-following/hill-climbing combination is abstracted to a discrete action, A, and the perceptual experience at each distinctive state into views $V1$ and $V2$, forming a causal schema $\langle V1|A|V2\rangle$. (Adapted from Kuipers (2000).) 14

2.2 **Example 5x5 Self-Organizing Map** A SOM learning a representation of a high-dimensional, continuous state space. In training, each sensor image is compared with the weight vector of each cell, and the weights are adapted so that over time each cell responds to a different portion of the input space. [Figure adapted from (Miikkulainen, 1990)] 27

2.3 **A Growing Neural Gas (GNG) Network** An Example GNG network adapting to model an input distribution with 1-dimensional, 2-dimensional, and 3-dimensional parts. The GNG network has several properties that make it a potential improvement over typical SOMs for our feature learning problem: It requires no prior assumptions about the dimensionality of the task, it can continue to grow and adapt indefinitely. The Homeostatic-GNG developed for SODA regulates its growth in order to maintain a fixed value of cumulative error between the input and the winning units. Figure adapted from Fritzke (1995) 30

4.1 **Simulated Robot Environment, T-Maze**. (a) A "sensor-centric" plot of a single scan from the robot's laser rangefinder in the T-maze environment, with the individual range sensor on the X axis and the range value on the Y axis. (b) The egocentric plot of the same scan in polar coordinates, with the robot at the origin, heading up the Y axis. This format provides a useful visualization of laser rangefinder scans for human consumption. (c) A screen-shot of the Stage robot simulator with the robot in the pose that generated the scan used in (a) and (b). The shaded region represents the area scanned by the laser rangefinder. The robot has a drive-and-turn base, and a laser rangefinder. The environment is approximately 10 meters by 6 meters. The formats in (a) and (b) will be used in later figures to display the learned perceptual prototypes. 60

4.2 **Example Learned Perceptual Prototypes, T-Maze.** The agent's self-organizing feature map learns a set of perceptual prototypes that are used to define perceptually distinctive states in the environment. This figure shows the set of features learned from one feature-learning run in the T-Maze. In each row the first figure represents a unit in the GNG, and the other figures represent the units it is connected to in the GNG topology. Some rows are omitted to save space, but every learned feature appears at least once. Each feature is a prototypical laser rangefinder image. On the left, the ranges are plotted in the human-readable format of Figure 4.1(b). On the right, the ranges are plotted in the sensor-centric format of Figure 4.1(a). The agent learns a rich feature set covering the perceptual situations in the environment 63

4.3 **Three Learned Prototypes, Enlarged** Enlarged views of features 8, 35, and 65 from Figure 4.2. These figures show prototypical sensory views of looking down the long corridor in the T-Maze, looking toward the side wall at the end of the corridor, and the view from the middle of the T intersection, respectively. 64

4.4 **Simulated Robot Environment, ACES**. The ACES4 environment. A simulated environment generated from an occupancy grid map of the fourth floor of the ACES building at UT Austin. The map was collected using a physical robot similar to the simulated robot used in these experiments. The environment is approximately 40m × 35m. The small circle represents the robot. The area swept by the laser rangefinder is shaded. This environment is much larger and perceptually richer than the T-maze. 65

4.5 **Example Learned Perceptual Prototypes, ACES.** The agent's self-organizing feature map learns a set of perceptual prototypes that are used to define perceptually distinctive states in the environment. The richer variety of perceptual situations in the ACES environment produces a larger set of features using the same parameter settings for the Homeostatic-GNG. 67

4.6 **Views of T-Maze Intersection** The use of Euclidean distance as the similarity metric for the GNG leads to the learning of many similar views that differ only by a small rotation. A small rotation in the robot's configuration space induces a large (Euclidean) movement in the robot's input space causing large distortion error in the GNG, which must add more features in that region of the space to reduce the error. The large number of features means that the agent must traverse many more distinctive states when turning than when traveling forward and backward. Nevertheless, the experiments in Chapter 6 show that SODA still greatly reduces task diameter and improves navigation learning. 68

5.1 **Trajectory Following Improvement with Learning** These figures show the results of 100 trajectory following runs from each of three locations (marked with large black disks). The end point of each run is marked with a '+'. The top figure shows the results of open-loop TF, and the bottom figure shows the results of TF learned using reinforcement learning. The TF learned using RL are much better clustered, indicating much more reliable travel. 72

5.2 **The average trajectory length and endpoint spread for open loop vs. learned trajectory-following for the last 100 episodes in each condition.** Learned TF options produce longer trajectories. Error bars indicate ± one standard error. All differences are significant at $p < 6 \times 10^{-5}$ 74

5.3 **Test Hill-Climbing Features.** The hill-climbing experiments tested SODA's ability to hill-climb on these five features from the GNG in Figure 4.2in the T-Maze environment. 79

5.4 **Hill-climbing Learning Curves.** These curves compare learned hill-climbing using the Top^3, Top^4, and Top^5 state representations against hill-climbing by approximating the gradient with user-defined linear action models. Each plot compares hill-climbers on one of the five different SOM features in Figure 5.3. The Y axis indicates the final feature activation achieved in each episode. The thick straight line indicates the mean performance of sampling-based HC, and the two thin straight lines indicate its standard deviation. This figure shows that Learned HC does about as well as an HC controller that manually approximates the feature gradient at each point, and sometimes better. 80

5.5 **Hill-climbing performance with and without learned options.** Using learned options makes hill-climbing achieve the same feature values faster. *Top:* The average lengths of hill-climbing episodes in the neighborhoods of the three different features shown in Figure 5.3. All differences are significant ($p < 2 \times 10^{-6}$). The bottom chart shows the average maximum feature value achieved for each prototype per episode. The plots compare the last 100 HC episodes for each feature with 100 hard-coded HC runs. Differences are significant between learned options and the other two methods ($p < 0.03$) for all features except 65, which has no significant differences. Across all features, the maximum values achieved are comparable, but the numbers of actions needed to achieve them are much smaller. 81

6.1 **T-Maze Navigation Targets.** The red circles indicate the locations used as starting and ending points for navigation in the T-Maze in this chapter. In the text, these points are referred to as *top left*, *top right*, and *bottom*. Navigation between these targets is a high-diameter task: they are all separated from each other by hundreds of primitive actions, and from each target the other two are outside the sensory horizon of the robot. 87

6.2 **T-Maze Learning, all routes.** These learning curves show the length per episode for learning to navigate between each pair of targets shown in Figure 6.1. The curves compare learning with $\mathcal{A}^1$ actions and learning with $\mathcal{A}^0$ actions. Each curve is the aggregate of three runs using each of three trained GNGs. Error bars indicate +/- one standard error. In all cases the $\mathcal{A}^1$ agents dramatically outperform the $\mathcal{A}^0$ agents. 89

6.3 **Navigation using Learned Abstraction**. An example episode after the agent has learned the task using the $\mathcal{A}^1$ actions. The triangles indicate the state of the robot at the start of each $\mathcal{A}^1$ action. The sequence of winning features corresponding to these states is [8, 39, 40, 14, 0, 4, 65, 7, 30, 62], shown in Figure 6.4. The narrow line indicates the sequence of $\mathcal{A}^0$ actions used by the $\mathcal{A}^1$ actions. In two cases the $\mathcal{A}^1$ action essentially abstracts the concept, 'drive down the hall to the next decision point.' Navigating to the goal requires only 10 $\mathcal{A}^1$ actions, instead of hundreds of $\mathcal{A}^0$ actions. In other words, task diameter is vastly reduced. 90

6.4 **Features for Distinctive States** These are the perceptual features for the distinctive states used in the navigation path shown in Figure 6.3, in the order they were traversed in the solution. (Read left-to-right, top-to-bottom.) The first feature [8] and the last two [30, 62] represent the distinctive states used to launch long actions down the hallways, while the intervening seven features [39, 40, 14, 0, 4, 65, 7] show the robot progressively turning to the right to follow the lower corridor. The large number of turn actions is caused by the large number of features in the GNG used to represent views separated by small turns, discussed in Section 4.4 and Figure 4.6. Despite the many turn actions, however, SODA still reduces the task diameter by an order of magnitude over primitive actions. 91

6.5 $\mathcal{A}^1$ **vs TF-only Learning Performance, T-Maze** These learning curves compare the length per an episode in the T-Maze top-left-to-bottom navigation task using primitive actions ($\mathcal{A}^0$), trajectory-following alone (TF), and trajectory following with hill-climbing ($\mathcal{A}^1$). Each curve is the average of 50 runs, 5 each using 10 different learned SOMs. Error bars indicate +/- one standard error. The agents using just TF actions, without the final HC step learn as fast and perform slightly better than the agents using $\mathcal{A}^1$ actions. (TF vs $\mathcal{A}^1$ performance is significantly different, $p < 0.0001$, in the last 50 episodes.) 93

6.6 **Hill-climbing Reduces Transition Entropy.** These plots compare the average transition entropy for all state/action pairs for each of the 10 different GNGs used in the T-Maze experiment. The x-axis indicates the transition entropy (in bits) using hill-climbing, and the y-axis indicates the entropy without hill-climbing. The solid line indicates equal entropy ($y = x$). Error bars indicate 95% confidence intervals. Hill-climbing reduces the entropy by about 0.4 bits, on average. This is approximately equivalent to a reduction in branching factor from 3 to 2. These results indicate that hill-climbing makes actions more deterministic, making them more useful for building planning-based abstractions on top of SODA. 95

6.7 **Hill-climbing Improves Task Diameter.** Bars show the average abstract length per episode of the successful episodes taken from the last 100 episodes for each agent in the T-Maze experiment. Abstract length is measured in the number of high-level actions needed to complete the task. Using trajectory-following with hill-climbing, the agents require an average of 23 actions to complete the task, while without hill-climbing they require an average of 67. (Error bars indicate +/- one standard error. Differences are statistically significant with infinitesimal p.) This result indicates agents using hill-climbing the hill-climbing component of $\mathcal{A}^1$ actions will be make them more useful for building future, planning-based abstractions on top of the SODA abstraction. 96

6.8 **ACES Navigation Task.** The circles indicate the locations used as starting and ending points for navigation in ACES in this chapter. The green circle on the right indicates the starting point and the red circle on the left indicates the ending point. The shaded area shows the robot's field of view. The longer task and added complexity of the environment make this task much more difficult than the tasks in the T-maze. 97

6.9 **Learning Performance in the ACES Environment.** These learning curves compare the length per episode for learning to navigate in ACES from the center-right intersection to the lower-left corner of the environment. The curves compare primitive actions and $\mathcal{A}^1$ actions. Each curve is the average of 50 runs, five each using 10 different learned SOMs. Error bars indicate +/- one standard error. The minimum length path is around 1200 actions. The agents using the high-level actions learn to solve the task while those using only primitive actions have a very difficult time learning to solve the task in the allotted time. 99

7.1 **Grouping SODA Actions into Gateways, Places and Paths.** Topological abstraction of SODA's actions could be accomplished identifying the states that initiate or terminate sequences of long similar actions as *gateways*, and then aggregating the sequences of states connected by long actions between two gateways into *paths*, and the collections of states connecting groups of gateways into *places*. In this T-Maze example, from Chapter 6, the states in which the robot enters and leaves the intersection terminate and begin sequences of long actions, and thus are gateways. The states and actions moving the robot down the corridor are grouped into paths, and the collection of states traversed in turning are grouped into a place. This second-level ($\mathcal{A}^2$) abstraction would reduce the diameter of this task from ten actions to three. 111

Chapter 1

Introduction

A generic learning agent boots up in a brand new mobile robot, with a high-resolution, high-dimensional range sensor, and continuous motor commands that send voltages to the wheel motors to drive and turn the robot. It also has a list of tasks, requiring navigating to distant locations in an office building. Being generic, however, the agent knows only that it has a large vector of continuous input values, a few continuous outputs, and some reward signals whose values it must maximize over time. This dissertation addresses the following question: Under these conditions, How can the agent learn to complete its tasks?

1.1 Motivation

The ultimate goal of machine learning in robotics is an agent that learns *autonomously*. In other words, it is a generic learning agent that can learn to perform its task without needing humans to point out relevant perceptual features or break down its task into more tractable subtasks. Constructing such an agent is difficult for two major reasons: (1) Many important real-world robotic tasks have a *high diameter*, that is, their solution requires a large number of primitive actions by the robots. (2) Modern robots are endowed with rich, high-dimensional sensorimotor systems, providing measurements of a continuous environment, and sending commands to a continuous motor system. These factors make the space of possible policies for action extremely large.

Methods of *reinforcement learning* — by which an agent learns to maximize a reward signal — have shown promise for learning robot behavior, but currently work best on low-diameter, low-dimensional tasks. Thus, current applications of reinforcement learning to real-world robotics require prior human analysis of the robot, environment, and task to provide a useful set of perceptual features and an appropriate task decomposition.

This dissertation presents *Self-Organizing Distinctive-state Abstraction* (SODA) a method by which a robot with little prior knowledge of its sensorimotor system and environment can automatically reduce the diameter of its tasks by bootstrapping up from its raw sensory motor system to high-level perceptual features and large-scale, discrete actions. Using unsupervised learning, the agent learns higher level perceptual features while exploring using primitive, local actions. Then, using the learned features, it builds a set of high-level actions that carry it between perceptually distinctive states in the environment. More specifically,

- SODA improves reinforcement learning in *realistic robots*, that is, robots with rich sensorimotor systems systems in high-diameter, continuous environments;
- it learns autonomously, with little or no prior knowledge of its sensorimotor system, environment, or task;
- it develops an accessible internal representation that is suitable for further bootstrapping to yet higher representations.

The remainder of this chapter discusses each of these goals in more detail. Section 1.2 briefly reviews the broad field of reinforcement learning, and discusses the difficulties of applying reinforcement learning in robots with rich sensorimotor systems in high-diameter, continuous environments. Section 1.3 discusses the motivations for learning without prior knowledge. Section 1.4 motivates the need for learning an internal representation suitable for further bootstrap learning, and the constraints it places upon the methods to be used. Section 1.5 gives an overview of the SODA method. Finally, section 1.6 lays out the organization of the rest of the material in the dissertation.

1.2 Reinforcement Learning in Realistic Robots

The learning problem that SODA addresses is a special case of *reinforcement learning*. Such problems all involve an agent interacting with an environment. The agent receives sensations from the environment, and performs actions that may change the state of the environment. It also receives a scalar reward signal that indicates how well it is performing its task. The learning problem for the agent, generally, is to discover a policy for action that maximizes the cumulative reward. Rewards may be sparsely distributed, and delayed, for example occurring only when the agent achieves some goal. As a result of this sparsity, the agent must determine how to assign credit (or blame) for the reward among all of the decisions leading up to the reward.

Reinforcement learning spans a vast set of learning problems that vary along many dimensions. Sensations and actions may be continuous or discrete. Rewards may be frequent or sparse, immediate or delayed. Tasks may be episodic, with a definite termination, or may continue indefinitely. Agents may begin learning *tabula rasa*, or may be given a full model of the environment *a priori*. A single agent may learn alone, or multiple agents may cooperate to maximize a single, global reward signal, or compete against one another.

SODA addresses the problem of a single agent learning in a *realistic robot*. Realistic robots have a number of features that make reinforcement learning with little prior knowledge difficult. There are two particular features of realistic robots that are of interest to this dissertation: (1) they have rich sensors measuring a complex environment, and (2) they have short-range, noisy, local actions that give their tasks a high-diameter. Efficient learning requires an abstraction of the continuous, high-resolution sensorimotor system to make the problem tractable.

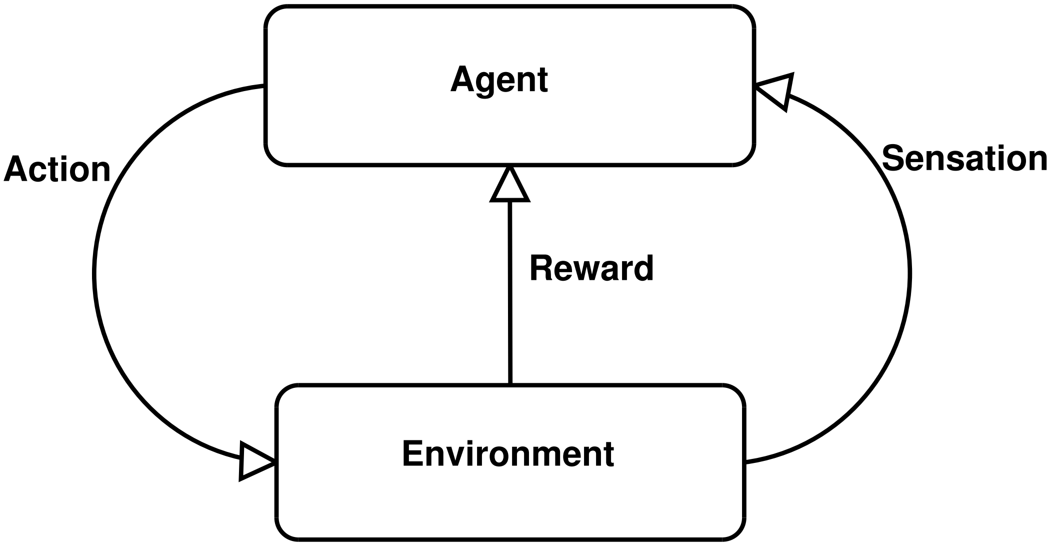


Figure 1.1: **The Reinforcement Learning Problem Framework** In reinforcement learning problems an agent interacts with an environment by performing actions and receiving sensations and a reward signal. The agent must learn to act so as to maximize future reward.

First, modern robots have rich sensory systems such as laser range-finders or cameras, that provide an array hundreds or thousands of measurements of the environment at once. In addition, real environments are rich and complex. In typical robotics applications, engineers use their knowledge of the nature of the sensors and environment to endow the agent with an appropriate set of feature detectors with which to interpret its sensory input. For example, an agent might be programmed with perceptual routines designed based on the knowledge that its sensors are range-finders and that it should attempt to extract features in the form of line segments because it operates in an indoor office environment. An agent learning to operate a robot without such prior knowledge is confronted simply with a large vector of numbers. The agent must learn on its own to extract the appropriate perceptual features from that input vector by acting in the world and observing the results.

Second, modern robots have motor systems that accept continuous-valued commands, such as voltages for controlling motors. Although robots controlled by digital computers still act by sending a discrete sequence of such commands, the size of the discretization is typically determined by the controller's duty cycle, which usually has a frequency of many cycles per second. As a result, the robot's primitive actions are typically fine-grained and local. Performing meaningful tasks, such as navigating to the end of a corridor, may require hundreds or thousands of these actions. Such long sequences of actions, combined with sparse reward, make the problem of assigning credit to the decisions in that sequence difficult, requiring a great deal of exploration of the space of possible sequences of actions. An engineer might shorten the task diameter by providing the agent with prior knowledge encoded in a set of high-level action procedures that abstract a long sequence of actions into a single step, such as *turn-around* or *follow-wall-to-end*. Without this prior knowledge, an agent must develop its own high-level abstraction of action if it is to reduce its task diameter.

In addition to rich sensors and local actions, real, physical robots have a number of other properties that make their use challenging, but that are not directly related to the main thesis of this dissertation, which is the autonomous development of a useful abstraction to aid reinforcement learning. These properties include the fact that they can operate for a limited time on a single battery charge, that they often must be physically repositioned at a starting point in order to run controlled experiments. For these reasons, the evaluation of SODA in Chapters 4, 5, and 6, is done in a *realistic simulation* in which the robot is modeled on an actual physical robot with rich sensors, fine-grained continuous actions, with realistic noise. This simulation allows many experiments to be run very efficiently while preserving the properties that are important for evaluating SODA.

1.3 Learning Without Prior Knowledge

The second goal of this work is to make progress toward the generic autonomous learning agent mentioned at the outset of this chapter. There are both engineering and scientific reasons for this goal. The engineering reason is that building an agent that requires little prior knowledge to learn a task saves human labor in adding the knowledge manually. Furthermore, such an agent should be robust and adaptable to new situations, since it must learn to act with few prior assumptions about itself or its world. The scientific reason is that by exploring the limits of how much prior knowledge is required to learn to act in the world we gain a greater understanding of the role of such knowledge. With such understanding, we can make progress toward a general theory of how much knowledge is necessary to perform a task, and how and when one might add prior knowledge to an agent to expedite the solution of a particular problem, without compromising the agent's robustness and adaptability to new situations.

In practice, it is probably impossible for an artificial agent to begin learning entirely *tabula rasa*. At a minimum, it is likely that the algorithms used will have free parameters that must be set properly for learning to succeed, and their values will encapsulate some prior knowledge. The representational framework will likely also embody some assumptions about the nature of the problem and the world. In addition, SODA is not intended to address every single problem encountered in applying reinforcement learning in robots. Some problems, such as an inability of robot to completely observe its state through available sensors, are orthogonal research questions, already under investigation elsewhere (see Section 2.2.5). When such problems, known as *perceptual aliasing* or *partial observability* are encountered in experimental demonstrations of SODA, it is reasonable to add sufficient additional information to SODA's sensory input(e.g. a compass or coarse location sensor) to allow the experiments to run without confounding the issues of high-diameter with those of partial observability.

However, by consciously attempting to identify and limit the prior knowledge given to the agent, we can try to set an upper bound on the prior knowledge needed for an important class of learning problems, and make steps toward a principled framework for understanding how and when such knowledge should be added. One means of constructing such a theory is to divide an agent's cognitive system into a set of modules with well-defined inputs, and then examine the assumptions that the modules make about their inputs to see how much prior knowledge must be engineered into each module. In this way, it may be possible to create a hierarchy of such modules, in which the lowest level modules assume only raw input and output from the robot, and the higher level modules assumptions are satisfied by the output from the lower modules, bootstrapping up to high-level behavior.

1.4 Bootstrapping to Higher Levels of Representation

SODA can be seen as one investigation in a larger research area known as *Bootstrap Learning* (Kuipers *et al.*, 2006), that seeks to understand how an agent can begin with the "pixel-level" ontology of continuous sensorimotor experience with the world and learn high-level concepts needed for common-sense knowledge. This learning is achieved through the hierarchical application of simple-but-general learning algorithms, such that the concepts learned by one level become the input to learning processes at the next level. One goal of SODA is to produce a method can serve as a substrate for bootstrapping of yet higher-level concepts and behaviors. Thus it is desirable to produce representations that can be used as input by a wide variety of learning methods.

This goal constrains the choices of methods available for constructing the learning architecture. Some methods, such as backpropagation networks and neuro-evolution, are attractive for their performance on these kinds of problems, but their internal representations are implicit, distributed and often inscrutable. These properties make it difficult to use them as part of a bootstrap learning process. The methods used in SODA were chosen in part because they abstract the agents experience into components, such as discrete symbols and kernel functions, that are usable by a wide variety of learning methods.

1.5 Approach

Given a robot with high-dimensional, continuous sensations, continuous actions, and one or more reinforcement signals for high-diameter tasks, the agent's learning process consists of the following steps (a detailed formal description of the method is provided in Chapter 3).

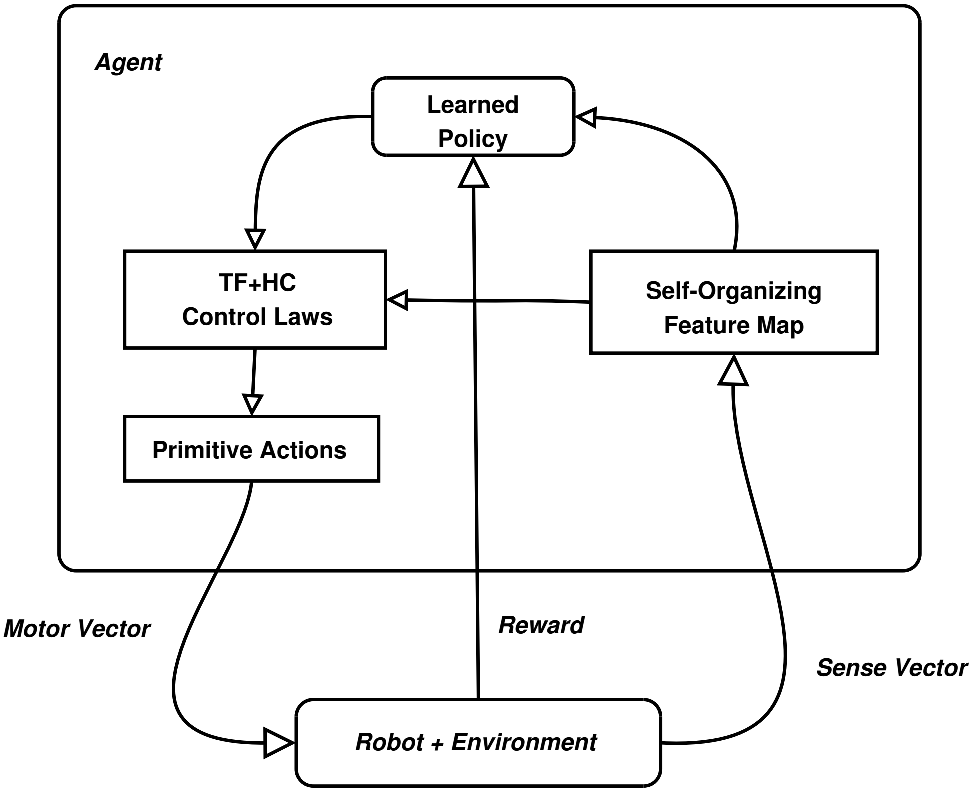


Figure 1.2: **The SODA Architecture** This expanded version of the diagram in Figure 1.1 shows the internal structure of the SODA agent (For the purposes of this description, the physical robot is considered part of the environment, and not the agent). The agent receives from the environment a sensation in the form of a continuous *sense vector*. This vector is passed to a *self-organizing feature map* that learns a set of high-level perceptual features. These perceptual features and the scalar *reward* passed from the environment are fed as input to a *learned policy* that is updated using reinforcement learning. This policy makes an *action selection* from among a set of high-level actions consisting of combinations of learned *trajectory-following (TF)* and *hill-climbing (HC)* control laws. These high-level actions generate sequences of *primitive actions* that take the form of continuous *motor vectors* sent from the agent to the robot.

1. **Define Primitive Actions.** The agent defines a set of discrete, short-range, local actions to act as the primitive motor operations. They are a fixed discretization of a learnable abstract motor interface consisting of a set of "principal motor components".

2. **Learn High-level Perceptual Features.** The agent explores using the primitive actions, and feeds the observed sensations to a self-organizing feature map that organizes its units into a set of high-level perceptual features. These features are prototypical sensory impressions used both as a discrete state abstraction suitable for tabular reinforcement learning, and a set of continuous perceptual features for continuous control.

3. **Learn High-level Actions.** Using these new features, the agent defines perceptually distinctive states as points in the robot's state space that are the local best match for some perceptual feature, and creates actions that carry it from one distinctive state to another. The actions are compositions of (1) trajectory-following control laws that carry the agent into the neighborhood of a new distinctive state, and (2) hill-climbing control laws that climb the gradient of a perceptual feature to a distinctive state.

4. **Learn Tasks.** The agent attempts to learn its task using a well-known and simple reinforcement learning method with a tabular state and action representation (such as Sarsa(λ)), using the perceptual categories in the self-organizing feature map as the state space, and the the new high-level actions as actions.

Figure 1.2 shows the architecture of the learning agent after the high-level features and actions are learned. This method of *Bootstrap Learning* in which each stage of the representation builds upon previously learned stage allows the robot to start with a high-diameter, high-dimensional, continuous task and build up a progressively more abstract representation. The learned features reduce the high-dimensional continuous state space to an atomic, discrete one. Likewise, the high-level actions reduce the effective diameter of the task by ensuring that the agent moves through the environment in relatively large steps. This reduction in task diameter allows the agent to learn to perform its tasks much more quickly than it would if it had to learn a policy over primitive actions.

1.6 Overview of the Dissertation

This dissertation demonstrates the effectiveness of SODA in learning navigation, using a series of experiments in two different environments on a realistic simulation of a mobile robot. The dissertation is structured as follows:

- Chapter 2 describes SODA's foundations in self-organizing feature maps, reinforcement learning, and the Spatial Semantic Hierarchy. The chapter also discusses related work such as other autonomous learning architectures, hierarchical reinforcement learning (including the Options framework), automatic temporal abstration for learning agents, and automatic feature construction for reinforcement learning.
- Chapter 3 presents a formal description of SODA's algorithm and assumptions, a formal definition of trajectory-following and hill-climbing actions, some different methods used for implementing TF and HC controllers, including learning them with RL, and the novel state-representation used by SODA for learning the controllers.
- Chapter 4 describes the simulated robot and the two learning environment used in the experiments in this dissertation, one a small test environment, the other large, realistic simulation of the floor of an office building. The chapter goes on to describe the results of the feature learning phase of SODA in the two environments, showing that the agent learns features representing the variety of possible perceptual situations in those environments, such as corridors and intersections at different relative orientations to the robot.
- Chapter 5 describes experiments and results comparing learned trajectory-following

and hill-climbing actions to hard-coded alternatives showing that learning the actions as nested reinforcement learning problems makes them more efficient and reliable while minimizing the amount of prior knowledge required.

- Chapter 6 presents experiments comparing SODA's ability to learn long-range navigation tasks to that of an agent using only low-level, primitive actions. SODA learns much faster. In addition, this chapter analyzes the role of hill-climbing in SODA's high-level actions. When SODA uses a hill-climbing step at the end of each high-level action, its actions are more reliable and the total task-diameter is lower than when just using trajectory-following alone.
- Chapter 7 discusses SODA's performance, strengths and weaknesses in more detail, and as the role of SODA in the general Bootstrap Learning framework. In addition, this chapter discusses a variety of possible future extensions to or extrapolations from SODA, including (1) replacing SODA's model-free reinforcement learning with some form of learned predictive model, (2) bootstrapping up to a higher, topological, representation, (3) improving SODA's feature learning by using better distance metrics for comparing sensory images. In addition, this section discusses the general role of prior knowledge in the context of creating a "self-calibrating" robot learning algorithm.
- Chapter 8 summarizes the SODA architecture and the experimental results showing the effectiveness of the SODA abstraction in reducing task diameter, and concludes with a view of the relationship between SODA, bootstrap learning, and the general enterprise of creating artificially intelligent agents.

Chapter 2

Background and Related Work

The first section of this chapter provides background information on the main pieces of prior work on which SODA is based: (1) the Spatial Semantic Hierarchy, (2) artificial reinforcement learning, including hierarchical reinforcement learning and (3) self-organizing maps. The next sections, 2.2 and 2.3 discuss related work:

- Section 2.2 describes Bootstrap Learning, a philosophy for building agents that learn world representations from primitive sensorimotor experience by first learning primitive concepts and using those learned concepts to bootstrap the learning of higher-level concepts.
- Section 2.3 discusses approaches for automatic feature construction or state abstraction in reinforcement learning, and relates these to SODA's use of self-organizing maps.

Finally, Section 2.4 looks at SODA vis-a-vis *tabula rasa* learning, and examines the assumptions implicit in SODA's architecture.

2.1 The Building Blocks of SODA

SODA is built upon three major components of prior work: (1) The Spatial Semantic Hierarchy is used as a framework for building agents that automatically learn navigate, as discussed in Section 2.1.1; (2) artificial reinforcement learning, discussed in Section 2.1.2, is a SODA agent's method for learning how to act; (3) Self-Organizing Maps, described in Section 2.1.3, are used by SODA to extract useful perceptual features from the agent's sensory input stream.

2.1.1 The Spatial Semantic Hierarchy (SSH)

The Spatial Semantic Hierarchy (Kuipers, 2000), a model of knowledge of large-scale space used for navigation, provides part of the representational framework for SODA's learning problem. The SSH abstracts an agent's continuous experience in the world into a discrete topology through a hierarchy of levels with well-defined interfaces: the *control level*, the *causal level*, the *topological level*, and the *metrical level*. SODA is founded on the control and causal levels, which are described below. An possible extension of SODA based on ideas from the topological level is described in Section 7.2.2.

The control and causal levels of SSH provide a framework for grounding the reinforcement learning interface in continuous sensorimotor experience. Specifically, the control level of the SSH abstracts continuous sensory experience and motor control into a set of discrete states and action defined by *trajectory-following* (TF) and *hill-climbing* (HC) control laws. It defines the *distinctive state* as the principal subgoal for navigation. A distinctive state is a stationary fixed point in the environment at the local maximum of a continuous, scalar perceptual feature, reachable (from within some local neighborhood) using a hill-climbing control law that moves the robot up the gradient of the feature. The control level also includes a set of trajectory-following control laws that carry the robot from one distinctive state into the neighborhood of another. The causal level of the SSH represents the sensorimotor interface in terms of discrete actions that carry the robot from one dis-

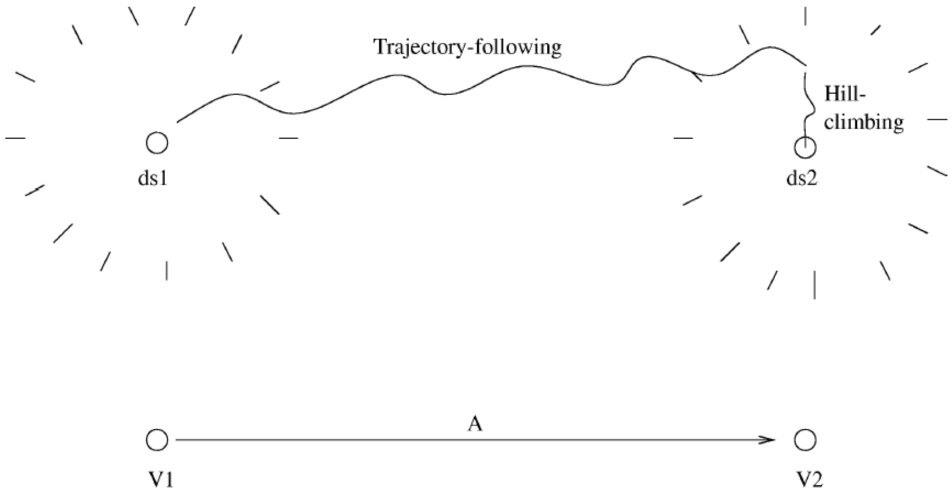


Figure 2.1: **Continuous-to-Discrete Abstraction of Action** In the SSH Control Level (top), continuous action takes the form of hill-climbing control laws, that move the robot to perceptually distinctive states, and trajectory-following control laws, that carry the robot from one distinctive state into the neighborhood of another. In the Causal Level, the trajectory-following/hill-climbing combination is abstracted to a discrete action, A, and the perceptual experience at each distinctive state into views $V1$ and $V2$, forming a causal schema $\langle V1|A|V2\rangle$. (Adapted from Kuipers (2000).)

tinctive state to another, and *views* that are a discrete abstraction of the robot's continuous perceptual state experienced at distinctive states.

The *control level* to *causal level* transition of the SSH provides a discrete state and action model, well grounded in continuous sensorimotor interaction. This abstraction is the means by which the learning agent will reduce its task diameter, by doing reinforcement learning in the space of these large-scale abstract actions, rather than small-scale local actions.

SODA builds on previous work by Pierce & Kuipers (1997) on learning the SSH control level in a robot with a continuous sensorimotor interface, but little prior knowledge of the nature of its sensors, effectors, and environment. In that work, the learning agent successfully learned the structure of the robot's sensorimotor apparatus and defined an abstract interface to the robot suitable for low-level control. By observing correlations in sensor values during action, the agent was able to group similarly behaving sensors together, separating out, for example, a ring of range sensors from other sensors. Then, using a variety of statistical and other data analysis techniques, the agent discovered the spatial arrangement of the sensor groups. By computing sensory flow fields on the major sensor groups and using principal component analysis (PCA), the agent defined an abstract motor interface, represented by a basis set of orthogonal motor vectors $\mathbf{u}^0, \ldots, \mathbf{u}^n$.

To define higher-level actions, the agent defined two kinds of control laws, *homing behaviors* for hill-climbing and *path-following behaviors* for trajectory following. Each kind of behavior was represented by a simple control template. Every control law is an instantiation of one of those two templates, with the parameters set to minimize an error criterion. The error criteria are defined in terms of a set of learned scalar perceptual features $\mathcal{Y}$, such as local maxima or minima over sensory images, defined through a set of feature generators.

Each homing behavior is defined by a *local state variable* $y \in \mathcal{Y}$, and is applicable only when y is known to be controllable by a *single* motor basis vector $\mathbf{u}^j$. The homing control template defines a one-dimensional proportional-integral controller that scales $\mathbf{u}^j$ as a function of $y^* - y$, where y^* is the target value for y. Each controller only operates along one axis of the motor space, requiring the sensory and motor spaces to decompose in closely corresponding ways.

Each path-following behavior is defined by a set of local state variables $\{y_1, \ldots, y_n\} \subseteq \mathcal{Y}$, and is applicable when all its local state variables can be held constant while applying some $\mathbf{u}^j$ (or its opposite) to move the robot. The path-following control template comprises a constant application of $\mathbf{u}^j$ plus a linear combination of the other motor basis vectors where the linear coefficients are determined by proportional-integral or proportional-derivative controllers designed to keep all y_i within a target range.

These methods – sensor grouping and learning an abstract, low-level motor interface – allow the robot to learn continuous control with little prior knowledge, but are not a suitable discrete sensorimotor abstraction for reinforcement learning. In particular the feature discovery mechanism constructs a tree of heterogenous features of different types and dimensionalities, and it is difficult to see how this variety of different kinds of percepts could be used as input to an RL algorithm. Furthermore, the learned behaviors are limited by their use of simple controller templates and their applicability only when a local state variable can be controlled using a single component of the abstract motor interface.

SODA assumes an that the agent has discovered the primary sensor grouping and learned the abstract motor interface using the methods above. SODA then augments this system with a new feature generator that provides features that can be used both as local state variables for continuous control, and as a discrete state abstraction for well understood reinforcement learning algorithms. It also defines a new set of trajectory-following and hill-climbing controllers based on these new features that are not subject to the limitations of Pierce & Kuipers' method.

2.1.2 Reinforcement Learning

Expanding the summary of reinforcement learning (RL) presented in the introduction, this section introduces the basic concepts in reinforcement learning used by SODA. The material below lays out the formal underpinnings of reinforcement learning in Markov Decision processes (MDPs), describes the methods that SODA's RL policies use to select exploratory actions for learning, and describes the Sarsa learning algorithm, used by SODA. The section goes on to discuss hierarchical extensions to reinforcement learning used by SODA.

Formal Framework of RL

The general formal framework for reinforcement learning is the *Markov Decision Process* (MDP): Given a set of states $\mathcal{S}$, a set of actions $\mathcal{A}_s$ applicable in each state $s \in \mathcal{S}$, and a reinforcement signal r, the task of a reinforcement learning system is to learn a *policy*, π :

$\mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. Given a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}_s$, $\pi(s, a)$ indicates the probability of taking action a in state s. The optimal policy maximizes the expected *discounted future reward*, as expressed in the *value function* of the state under that policy:

$$V^{\pi}(s) = E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t = s,\ \pi\right\}, \tag{2.1}$$

indicating the expected discounted sum of reward for future action subsequent to time t assuming the state at time t is s, and actions are selected according to policy π. $\gamma \in [0, 1]$ is the *discount rate* parameter. When a predictive model of the environment is available, in the form of the *transition function* $T(s_t, a_t, s_{t+1}) = P(s_{t+1}|s_t, a_t)$ it is possible to derive the policy for action from the value function by predicting what state will result from each of the available actions, and setting the probabilities $\pi(s_t, a_t)$ to maximize the expected value of $V(s_{t+1})$.

When no model of the environment is available to provide the T function, it is typical to collapse V and T into an intermediate function, the *state-action-value function*, $Q^{\pi}(s, a)$, giving the value of taking action a in state s under policy π:

$$Q^{\pi}(s, a) = E\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \mid s_t = s,\ a_t = a,\ \pi\right\}. \tag{2.2}$$

The policy can be derived from Q^{π} through a number of methods, for example by always selecting the action with the highest Q value. Therefore a large family of reinforcement learning methods concentrate on learning estimates of one of these two functions, V or Q.

Methods for learning Q rely on the fact that the optimal policy Q^* obeys the *Bellman Equation*:

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a' \in \mathcal{A}_s} Q^*(s', a'), \tag{2.3}$$

where $R(s, a)$ is the expected reward r resulting from taking action a in state s.

An iterative algorithm for approximating Q^* via dynamic programming (DP) uses an update operation based on the Bellman equation:

$$Q_{k+1} = R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a' \in \mathcal{A}_s} Q_k(s', a'). \tag{2.4}$$

Many methods exist that combine DP updating with *Monte Carlo policy iteration* to update a policy π on-line such that Q^{π} provably converges to Q^{*} under a handful of simplifying assumptions. *Sarsa*, the method used by SODA, is one such method and is described in more detail later in this section. The approximations necessary to apply these techniques in robots with rich sensorimotor systems generally violate these simplifying assumptions, and convergence proofs in these situations have thus far remained elusive. Nevertheless, these techniques have been quite successful in real-world domains such as robotic soccer (Stone & Veloso, 2000), albeit by manually incorporating a significant amount of prior knowledge of the task.

In order to learn, these on-line algorithms need to choose at each step whether to perform the action with the highest Q-value, or whether to choose some other, exploratory action, in order to gain more information for learning. The most popular of these methods are covered in the next section.

Exploration vs. Exploitation in RL

In reinforcement learning, agents attempt to learn their task while performing it, iteratively improving performance as they learn. An agent in such a situation must decide when choosing actions whether to *explore* in order to gain more information about its state-action space, or *exploit* its current knowledge to attempt to immediately maximize future reward. The typical way of dealing with this choice is for the agent's policy to explore more early, and gradually reduce exploration as time goes on. The two most popular main methods for implementing this tradeoff, both used by SODA, are *epsilon-greedy action selection*, and *optimistic initialization*.

In epsilon-greedy action selection, at each step the agent chooses a random exploratory action with probability ϵ, $0 < \epsilon < 1$, while with probability $1 - \epsilon$, the agent chooses the *greedy* action, that it estimates will lead to the highest discounted future reward, i.e. $\arg\max_{a \in \mathcal{A}_s} Q(s, a)$. Learning begins with a high value for ϵ that is annealed toward zero or some small value as learning continues.

Optimistic initialization refers to initializing the estimates of Q with higher-than-expected values. As the agent learns, the estimates will be adjusted downward toward the actual values. This way greedy action selection biases the agent to seek out regions of the state space or state-action space that have not been visited as frequently. It is possible to combine optimistic initialization with ϵ-greedy action selection in a single agent.

Episodic vs. Continual RL Tasks

Some reinforcement learning problems terminate after some period of time, and are characterized as *episodic*. Others continue indefinitely and are called *continual* (or *non-episodic*). Trajectory-following and hill-climbing in the SSH are examples of episodic tasks, since each continues for a finite period of time, while large scale-robot navigation using TF and HC actions may be either continual or episodic, depending on the task.

Episodic tasks have a *terminal state* in which action stops and the agent receives its "final reward" (for that episode). Non-episodic tasks, on the other hand, have no terminal state, and the agent continues to act indefinitely. The learning problem in episodic tasks is usually to maximize the total reward per episode, so it is not necessary to discount the reward and $\gamma = 1.0$. In continual tasks, reward will accumulate indefinitely, so it is necessary to discount future reward with $0 < \gamma < 1$. Generally speaking, episodic tasks can be solved with either on-line or off-line methods, while continual tasks must be solved on-line. Depending on the nature of the task, however, it may be possible for an agent in a continual task to stop acting and "sleep" periodically in order to learn off-line.

The experimental navigation tasks used to test SODA in Chapter 6 are all episodic. However, SODA could just as easily be applied to continual navigation, for example in a delivery robot that must continually make rounds of a large building.

The Sarsa(λ) Algorithm

The feature and action construction methods in SODA are intended to be independent of the specific reinforcement learning algorithm used. The experiments in this thesis uses Sarsa(λ), described by Sutton & Barto (1998). Sarsa was chosen because it is simple and well understood. Also, Sarsa has good performance when used with function approximation – a necessity when using RL in robots. Some other methods, like Q-learning (Watkins & Dayan, 1992), are known to diverge from the optimal policy when used with function approximation (Baird, 1995).

'Sarsa' is an acronym for State, Action, Reward, State, Action; each Sarsa update uses the states and actions from time t and the reward, state, and action from time $t + 1$, usually denoted by the tuple: $\langle s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}\rangle$. In the simplest form, Sarsa modifies the state-value estimate $Q(s, a)$ as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]. \tag{2.5}$$

This rule updates the current Q value by a fraction of the reward after action a_t plus the *temporal difference* of the discounted reward predicted from the next state-action pair, $Q(s_{t+1}, a_{t+1})$, and the current estimate of $Q(s_t, a_t)$. The parameter $0 < \alpha < 1$ is a learning rate that controls how much of this value is used. The parameter $0 < \gamma \leq 1$ is the discount factor. For episodic tasks that ultimately reach a terminating state, $\gamma = 1$ is used, allowing $Q(s, a)$ to approach an estimate of the remaining reward for the episode.

For faster learning, Sarsa(λ) performs multiple-step backups by keeping a scalar *eligibility trace*, $e(s, a)$, for each state action pair, that tracks how recently action a was taken in state s — i.e., how "eligible" it is to recieve a backup. Using this information the algorithm updates the estimates of the value function for many recent state-action pairs on each step, rather than just updating the estimate for the most recent pair. When a step is taken, each eligibility trace is decayed according to a parameter $0 \leq \lambda < 1$:

$$\forall s, a : e(s, a) \leftarrow \lambda e(s, a), \tag{2.6}$$

Then the eligibility trace for the most recent state and action is updated:

$$e(s_t, a_t) \leftarrow 1. \tag{2.7}$$

Finally the Q table is updated according to the eligibility trace:

$$\forall s, a : Q(s, a) \leftarrow Q(s, a) + e(s, a)\, \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]. \tag{2.8}$$

This method can speed up learning by a large margin, backing up the reward estimates to recently experienced state-action pairs, rather than just the most recent. (Note that when $\lambda = 0$ the eligibility trace is zero for all but the most recent state-action pair, and the update rule in Equation 2.8 reduces to the one step update in Equation 2.5.) Sarsa(λ)'s simplicity and known convergence properties make it a good choice for use in SODA.

Hierarchical Reinforcement Learning and Options

Reinforcement learning methods that maintain a single, flat, policy struggle on high-diameter problems. To address this failing, much recent work has gone into extending the paradigm to learn a task as a decomposition of subtasks, or by using other forms of temporal abstraction. Originally this work assumed a task that had been decomposed by a human, and concentrated on policy learning. More recently, however, work has gone into agents that automatically discover a useful task decomposition or temporal abstraction.

The most widely-used formalism for representing hierarchical task decomposition in reinforcement learning is the semi-Markov decision process (SMDP). SMDPs extend MDPs by allowing actions with variable temporal extent, that may themselves be implemented as SMDPs or MDPs, executed as “subroutines.” Such processes are “semi-Markov” because the choice of primitive actions (at the lowest level of the decomposition) depends not only on the environmental state, but also on the internal state of the agent, as manifest in the choice of higher-level actions.

There are three major learning systems and frameworks based on SMDPs: *Options* (Precup, 2000; Sutton, Precup, & Singh, 1999), *MAXQ value function decomposition* (Dietterich, 2000), and *Hierarchies of Abstract Machines* (HAMs; Parr & Russel, 1997; Andre & Russell, 2001) . These methods derive much of their power from the fact that short policies are easier to learn than long policies, because the search space is smaller. Using this knowledge they structure a long task as a short sequence of extended subtasks, that may themselves be compositions of lower-level subtasks. In other words, all these methods attempt to reduce the diameter of high-diameter tasks.

SODA's abstraction is built on Options, because its formalism provides a convenient framework for specifying trajectory-following and hill-climbing actions. In addition, hand-specified Options have been successfully used in real-world robotics tasks. For example, Stone & Sutton (2001) used the options framework in the complex task of robot soccer. Finally Options has been the most successful framework for automatic discovery of high-level actions. MAXQ and HAMs have primarily been used as a means by which a human can specify a useful set of subtasks for learning. The rest of this section presents the Options framework in more detail, and describes some existing methods for automatic Option discovery.

The *options* formalism extends the standard reinforcement learning paradigm with a set of temporally-extended actions $\mathcal{O}$, called options. Each option o_i in $\mathcal{O}$ is a tuple $\langle \mathcal{I}_i, \pi_i, \beta_i \rangle$, where the *input set*, $\mathcal{I}_i \subseteq \mathcal{S}$, is the set of states where the option may be executed, the policy, $\pi_i : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, determines the probability of selecting a particular action in a particular state while the option is executing, and the *termination condition*, $\beta_i : \mathcal{S} \rightarrow [0, 1]$, indicates the probability that the option will terminate in any particular state. For uniformity, primitive actions are formalized as options so each SMDP's policy chooses among options only. Each primitive action $a \in \mathcal{A}$ can be seen as an option whose input set $\mathcal{I}$ is the set of states where the action is applicable, whose policy π always chooses a, and whose termination condition β always returns 1. In cases where the set of actions is very large (or continuous) it is often desirable to allow each option to act only over a limited set of actions $\mathcal{A}_i$, such that $\pi_i : \mathcal{S} \times \mathcal{A}_i \rightarrow [0, 1]$. In addition, most work in options focuses on cases where the option policies are themselves learned using reinforcement learning — although this is not strictly necessary within the formalism. In this case, it is often useful to augment the option with a *pseudo-reward* function R_i, different from the MDP's reward function, to specify the subtask that the option is to accomplish. SODA's trajectory-following and hill-climbing actions are defined and learned as options. Sections 3.3 and 3.4.2 define $\mathcal{I}_i$, π_i, β_i, R_i, and $\mathcal{A}_i$ for SODA's TF and HC options.

To learn a policy for selecting options to execute, the Sarsa(λ) algorithm reviewed in Section 2.1.2 must be slightly modified to accommodate options (Precup, 2000). Assume an option o_t is executed at time t, and takes τ steps to complete. Define $\rho_{t+\tau}$ as the cumulative, discounted reward over the duration of the option:

$$\rho_{t+\tau} = \sum_{i=0}^{\tau-1} \gamma^i r_{t+i+1}. \tag{2.9}$$

Using this value, the one-step update rule (Equation 2.5) is modified as follows:

$$Q(s_t, o_t) \leftarrow Q(s_t, o_t) + \alpha \left[\rho_{t+\tau} + \gamma^\tau Q(s_{t+\tau}, o_{t+\tau}) - Q(s_t, o_t) \right]. \tag{2.10}$$

For Sarsa(λ), the multi-step update rule (Equation 2.8) is modified analogously. The eligibility trace now tracks state-option pairs, $e(s, o)$, and is updated upon option selection.

Note that for one-step options, these modifications reduce to the original Sarsa(λ) equations. Also, when $\gamma = 1$, as is often true in episodic tasks, the reward ρ is simply the total reward accumulated over the course of executing the option, and the update rule is again essentially the same as the original, if the reward r in the original rule is taken to mean the reward accumulated since the last action.

Work is ongoing to refine and extend the options framework, including sharing information for learning between options (Sutton, Precup, & Singh, 1998) and using the task decomposition to improve state abstraction (Jonsson & Barto, 2000). Although much early work on options has assumed an *a priori* task decomposition, more recent work has focused on discovering a useful set of options from interaction with the environment. This and other work on automatic discovery of hierarchical actions is reviewed in the next section.

Automatic Hierarchy Discovery

Manually decomposing a task into subtasks is a means of using prior knowledge to shrink the effective diameter of a high-diameter task. One of the goals of SODA is an agent that learns to reduce the diameter of its task without such prior knowledge.

Some work has been done on automatically learning a task hierarchy for this purpose. Nested Q-Learning (Digney, 1996, 1998) builds a hierarchy of behaviors implemented as learned sub-controllers similar to options (Section 2.1.2). It operates either by proposing every discrete feature value as a subgoal and learning a controller for each, or proposing as subgoals those states that are frequently visited or that have a steep reward gradient. The former method can only be tractable with a relatively small set of discrete features. Digney intended the latter version to be tractable with larger feature sets, although it was only tested in a very small, discrete grid-world.

The work of McGovern & Barto (2001) and McGovern (2002) is similar in many respects to the work of Digney. States are selected as subgoals using a statistic called "diverse density" to discover states that occur more frequently in successful trials of a behavior than in unsuccessful trials, and new options are created to achieve these states as subgoals. This method differs from Nested Q-Learning in that it is able to use negative evidence (that a state is not on the path to the goal, and thus not likely to be a subgoal). To do this, however, it requires that experiences be separated into "successful," and "failed" episodes, which seems to eliminate its use in non-episodic tasks. Furthermore, it requires the use of a hand-crafted, task-specific "static filter" to filter out spurious subgoals. In testing on 2-room and 4-room grid-worlds with narrow doorways connecting the rooms, this method correctly identified the doorways between the rooms as subgoals.

Building on the work of McGovern & Barto, Şimşek et al have developed two new ways of discovering useful options. The first method uses a new statistic called *relative novelty* (Şimşek & Barto, 2004) to discover states through which the agent must pass in order to reach other parts of the state space. The method then proposes these *access states* as subgoals and constructs options for reaching them. The second method uses local graph partitioning (Şimşek, Wolfe, & Barto, 2005) to identify states that lie between densely connected regions of the state space, and proposes those states as subgoals.

SODA is similar to these methods in that it also learns a task decomposition, but it does so by abstracting the agent's continuous sensorimotor experience into a set of discrete, perceptually distinctive states and actions to carry the robot between these states. In contrast, the option-discovery methods above assume a discrete state abstraction, and attempt to find a smaller set of states that act as useful subgoals and learn new subtask policies that carry the robot to these states. In each case, the agent's task then decomposes into a sequence of higher-level, temporally extended actions that decrease the effective diameter of the task. SODA's hill-climbing actions are similar to these sub-goal options in that they attempt to reach a distinctive state as a sub-goal. SODA differs in that hill-climbing options are quite local in their scope, and trajectory-following options are the means of carrying the robot from one distinctive state into the local neighborhood of another. Trajectory-following (which is formulated as an option in Section 3.3) is different from sub-goal options: Its purpose is to make progress on a given trajectory for as long as possible, rather than to achieve a termination condition as quickly as possible. In this way trajectory-following defines a new kind of option that is the dual of traditional sub-goal-based options. For very large diameter tasks, it may be necessary to add more layers of features and actions to further reduce the diameter. The methods above may be directly useful for bootstrapping to even higher levels of abstraction. One possible method of identifying and constructing higher-level options on top of SODA is proposed in Section 7.2.2.

To summarize, SODA uses reinforcement to learn policies for trajectory-following and hill-climbing actions, as well as for high-diameter navigation tasks using those actions. The agent's action set includes high-level actions as described above in Section 2.1.1, that are defined as Options using hierarchical reinforcement learning, and the state representation is learned using a self-organizing map, described in the next section.

2.1.3 Self-Organizing Maps (SOMs)

The unsupervised feature learning algorithm used in SODA is the Self-Organizing Map (SOM; Kohonen, 1995). The self-organizing map has been a popular method of learning perceptual features or state representations in general robotics, as well as world modeling for navigation (Martinetz, Ritter, & Schulten, 1990; Duckett & Nehmzow, 2000; Nehmzow & Smithers, 1991; Nehmzow, Smithers, & Hallam, 1991; Provost, Beeson, & Kuipers, 2001; Kroöse & Eecen, 1994; Zimmer, 1996; Toussaint, 2004). Many of these systems, however, provide some form of prior knowledge or external state information to the agent, and none attempt to build higher level actions to reduce the task diameter. The first part of this section presents the basic SOM learning method in the context of the standard "Kohonen Map" SOM algorithm, and details the reasons why SOMs are well suited for feature learning in SODA. The second part describes the Growing Neural Gas (GNG) SOM algorithm, and the *Homeostatic-GNG* variant developed for use with SODA.

The Standard Kohonen SOM

A standard SOM consists of a set of units or cells arranged in a lattice.[1] The SOM takes a continuous-valued vector $\mathbf{x}$ as input and returns one of its units as the output. Each unit has a weight vector $\mathbf{w}_i$ of the same dimension as the input. On the presentation of an input, each weight vector is compared with the input vector and a *winner* is selected as $\arg\min_i \|\mathbf{x} - \mathbf{w}_i\|$.

In training, the weight vectors in the SOM are initialized to random values. When an input vector $\mathbf{x}_t$ is presented, each unit's weights are adjusted to move it closer to the input vector by some fraction of the distance between the input and the weights according to

$$\mathbf{w}_i \leftarrow \mathbf{w}_i + \eta_t N_t(i)(\mathbf{x}_t - \mathbf{w}_i), \tag{2.11}$$

where $\mathbf{w}_w$ is the winning unit's weight vector and $0 < \eta_t < 1$ is the *learning rate* at time

---

[1] The lattice is often, but not necessarily, a 2D rectangular grid.

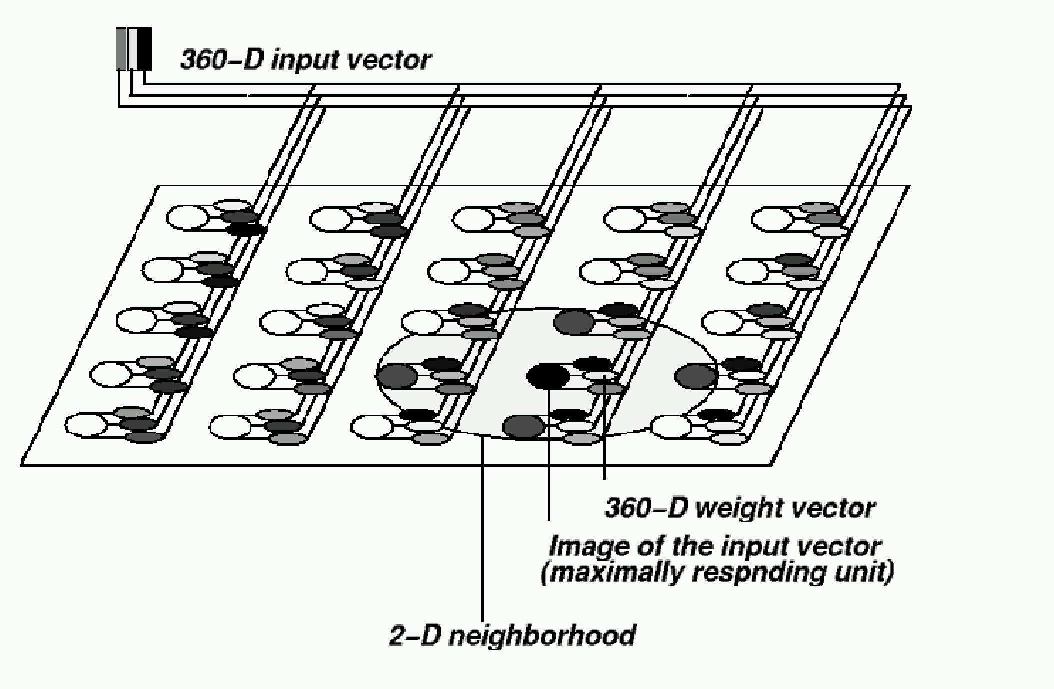


Figure 2.2: **Example 5x5 Self-Organizing Map** A SOM learning a representation of a high-dimensional, continuous state space. In training, each sensor image is compared with the weight vector of each cell, and the weights are adapted so that over time each cell responds to a different portion of the input space. [Figure adapted from (Miikkulainen, 1990)]

t, and $N_t : \mathbb{N} \rightarrow [0, 1]$ is the *neighborhood function* at time t. The neighborhood function returns 1 if i is the winning unit, and decreases with the distance of unit i from the winner in the SOM lattice, eventually decreasing to zero for units "outside the neighborhood" (Figure 2.2).

Training begins with initially large values for both the learning rate and the neighborhood size. As training proceeds, the learning rate and neighborhood are gradually annealed to very small values. As a result, early training orients the map to cover the gross topology of the input space, and as the parameters are annealed, finer grained structure of the input space emerges.

Self-organizing maps have several properties that lend themselves well to our feature learning task:

- *Data- and sensor-generality.* Because they operate on any input that can be expressed in the form of a vector, they are not specific to any particular kinds of sensor or environment, making them especially well suited to learning with an unknown sensorimotor system.

- *Clustering with topology preservation.* SOMs partition the input space into a set of clusters that tends to preserve the topology of the original input space in reduced dimensions. Thus features near one another in the SOM will be similar to one another. This property can be exploited to speed up reinforcement learning when a SOM is used to represent the state space (Smith, 2002).

- *Incremental training.* A SOM can be trained incrementally, with training vectors presented on-line as they are received during robot exploration. As training progresses the SOM first organizes into a rough, general approximation of the input space that is progressively refined. This property coincides well with the on-line nature of most reinforcement learning algorithms, and enables the agent to learn its policy concurrently with learning its state representation (Smith, 2002).

- *Adapting to the input distribution.* Unlike *a priori* fixed discretizations, SOMs concentrate their units in areas of the perceptual space where the input is distributed. Furthermore, the Growing Neural Gas (GNG) a modified SOM algorithms used in SODA (described below) is good at following non-stationary input distributions (Fritzke, 1997), making it easier for the robot to learn incrementally as it performs its explores its environment, without needing store training data and present it in batch. In some cases, this allows the agent to learn task policies with Sarsa while training the SOM concurrently.

In SODA, a SOM is used to learn a set of perceptual features from the sensory input. The features have three complementary roles. First, the units are used as discrete *perceptual* *categories* that form the state space for the reinforcement learning algorithm that chooses high-level actions. Second, the continuous activation values on the SOM are used by the agent to define closed-loop hill-climbing and trajectory-following control laws with which to construct high-level actions. Specifically, the agent hill-climbs to perceptually distinctive states, each defined by the local maxima of activation on a SOM unit, and trajectory-follows so as to make progress while maintaining the activation of the current SOM unit. Third, the sorted list of the n closest units to the current input forms the Top^n state representation, described in Section 3.2, which is used when learning hill-climbing and trajectory-following using reinforcement learning.

The Growing Neural Gas Algorithm

The SOM implementation used by SODA is based on a variant on the standard SOM algorithm called the Growing Neural Gas (GNG; Fritzke, 1995). The GNG begins with a small set of units and inserts new units incrementally to minimize distortion error (the error between the input, and the winning unit in the network). The GNG is able to continue learning indefinitely, adapting to changing input distributions. This property makes the GNG especially suitable for robot learning, since a robot experiences its world sequentially, and may experience entirely new regions of the input space after an indeterminate period of exploration. In addition, the GNG is not constrained by the pre-specified topology of the SOM lattice. It learns its own topology in response to experience with the domain. An abbreviated description of the GNG algorithm follows.

- Begin with two units, randomly placed in the input space.
- Upon presentation of an input vector $\mathbf{x}$:
    1. Select the two closest units to $\mathbf{x}$, denoted as w_1 and w_2 with weight vectors $\mathbf{q}_1$ and $\mathbf{q}_2$, respectively. If these units are not already connected in the topology, add a connection between them.

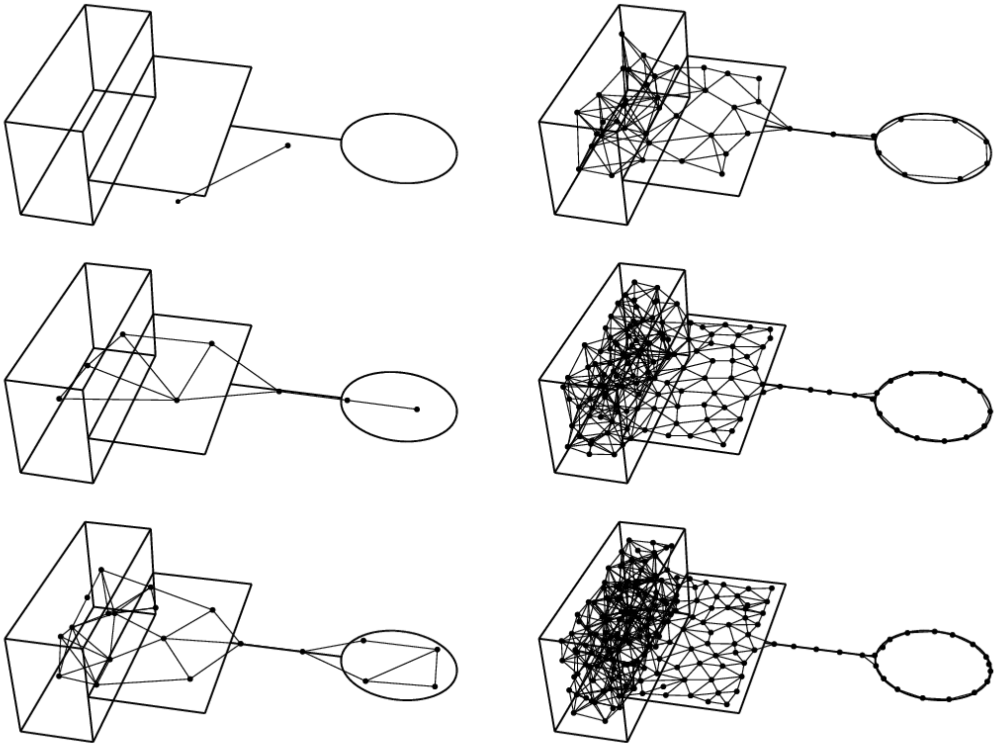


Figure 2.3: **A Growing Neural Gas (GNG) Network** An Example GNG network adapting to model an input distribution with 1-dimensional, 2-dimensional, and 3-dimensional parts. The GNG network has several properties that make it a potential improvement over typical SOMs for our feature learning problem: It requires no prior assumptions about the dimensionality of the task, it can continue to grow and adapt indefinitely. The Homeostatic-GNG developed for SODA regulates its growth in order to maintain a fixed value of cumulative error between the input and the winning units. Figure adapted from Fritzke (1995)

2. Move $\mathbf{q}_1$ toward $\mathbf{x}$ by a fraction of the distance between them. Move the weights of all the topological neighbors of w_1 toward $\mathbf{x}$ by a smaller fraction of the respective distances.
3. Increment the age $a_{w_1 j}$ of all edges j emanating from w_1.
4. Set the age $a_{w_1 w_2}$ of the edge between w_1 and w_2 to 0.
5. Remove any edges ij for which $a_{ij} > a_{\max}$. If this results in any units with no

edges emanating from them, remove those units as well.

6. Add the squared error $||\mathbf{x} - \mathbf{q}_1||^2$ to an accumulator e_{w_1} associated with w_1.
7. Decay the accumulated error of all nodes by a fraction of their values: $\forall i : e_i \leftarrow e_i + -\beta e_i$, where $0 < \beta < 1$.

- Periodically, every λ inputs, add a unit by selecting the existing unit with the greatest accumulated error and the unit among its topological neighbors with the most accumulated error; between these two units add a new unit whose weight vector is the average of the two selected units. Connect the new unit to the two selected units, and delete the original connection between the two.

The original GNG algorithm adds units until the network reaches some fixed criterion such as a maximum number of units. SODA uses a new, slightly modified algorithm, *Homeostatic-GNG*, that has no fixed stopping criterion, but rather only adds units if the average discounted cumulative error over the network is greater than a given threshold. Every λ inputs, the Homeostatic-GNG checks the condition $\bar{e} > e_\tau$, where e_τ is the error threshold. If the condition is true, and the error is over threshold, a unit is added, otherwise none is added. Homeostatic-GNG was first published as *Equilibrium-GNG* (Provost, Kuipers, & Miikkulainen, 2006).

Given a stationary input distribution, an Homeostatic-GNG will grow until reaching an equilibrium between the rate of accumulation of error (per unit) and the rate of decay. If the distribution changes to cover a new part of input space, the error accumulated in the network will increase above the threshold, and the network will grow again in the region nearest the new inputs.

To summarize, SODA uses a variant self-organizing map algorithm, Homeostatic-GNG network, to learn a set of prototypical sensory images that form the perceptual basis of the abstraction. Homeostatic-GNG is well suited for this task because it can be trained incrementally, covering the changing input distribution as the robot explores, and because it does not require a fixed specification of the number of features to learn, instead adjusting the number of features to maintain a prespecified level of discounted cumulative error.

This section described SODA's foundations in the Spatial Semantic Hierarchy, reinforcement learning, and self-organizing maps. The remainder of this chapter discusses a variety of related work in bootstrap learning, hierarhical reinforcement learning, and automatic feature construction for reinforcement learning.

2.2 Bootstrap Learning

Most robot learning architectures endow the agent with significant amounts of prior knowledge of the robot, environment, and task. As mentioned in Section 1.4, SODA is an instance of a "bootstrap learning" algorithm (Kuipers *et al.*, 2006). Agents using bootstrap learning algorithms build representations of their world progressively from the bottom up, by first learning simple concepts and then using those as building blocks for more complex concepts. These kinds of algorithms can be classified into two broad classes: homogeneous and heterogeneous.

Homogeneous bootstrap learning methods use the same learning method or set of methods at every level of learning, positing a single, uniform learning algorithm to account for agent learning from raw pixels and motor commands all the way up to high-level behavior. Drescher's Schema Mechanism, described in Section 2.2.1, and Chaput's Constructivist Learning Architecture (Section 2.2.2) are two such methods. Heterogeneous bootstrap learning methods, on the other hand, use different algorithms as needed for different levels of behavior. Human research determines hierarchy of methods and interfaces, within levels agents learn representations autonomously. Heterogeneous bootstrap learning methods are generally directed at learning specific kinds of agent knowledge rather than the whole scope of high-level behavior. In addition, they often assume the existence of some lower-level knowledge that has already been learned. As a result of this presumption, those methods can themselves be seen as building blocks in a larger bootstrap learning process that knits together the individual methods. Heterogeneous bootstrap learning methods include the work of Pierce & Kuipers, SODA, the place recognition system of Kuipers & Beeson (2002), and OPAL (Modayil & Kuipers, 2006, 2004). Pierce and Kuipers' work is described in Section 2.1.1; Kuipers and Beeson's place detection and Modayil's OPAL are described in Sections 2.2.3 and 2.2.4.

2.2.1 Drescher's Schema System

The *Schema System* (Drescher, 1991) uses a constructivist, Piagetian model of child development as a framework for an intelligent agent that learns to understand and act in its world with no prior knowledge of the meaning of its sensors and effectors. It was not applied to realistic robots, but was only tested in a very small discrete grid world. The Schema System does not use reinforcement learning, but instead explores its world attempting to find reliable *context-action-result* schemas. It assumes a primitive set of discrete, propositional features and discrete, short-range actions. High-level perceptual features are represented as propositional conjuncts, generated by exhaustively pairing existing features, or their negations, and testing the resulting conjuncts to see if they can be reliably achieved through action. High-level actions are created by chaining together sequences of primitive actions that reliably activate some high level feature.

One especially interesting feature of its representation is what Drescher calls the *synthetic item*. Each schema has a synthetic item representing the hidden state of the world that would make that schema reliable. For example, the schema for moving the hand to *position-X* and then feeling something touching the hand would have a synthetic item that could be interpreted as the proposition that there is an object at *position-X*. Drescher defined heuristics for when the agent would "turn on" synthetic items. For example, after a schema had successfully executed, its synthetic item would stay on for some period of time. He also proposed that artificial neural networks might be used to learn when to turn them on and off.

The Schema System was tested in a simulated "micro-world," i.e. a small, discrete two dimensional world with a hand that can touch and grasp "objects," a simple visual system with foveation, and a few other sensors and effectors. Even in this simple world, the number of combinations of propositions to search through is extremely large. Given that Drescher's implementation was unable to scale up to the full micro-world even using a Connection Machine, it is inconceivable that the system as implemented would scale to modern physical robots with rich, continuous sensorimotor systems, and there's no evidence that any attempt at such a system has been made (but see Section 2.2.2 for an alternative implementation in the same simulated world).

One of the main contributions of the Schema System is the idea that an agent with practically no prior knowledge can learn to understand itself and its world through a bottom-up, constructive search through the space of causal schemas, using ideas from developmental psychology both to provide a representational framework, and a set of heuristics for agent behavior to guide the search. Another contribution, embodied in the synthetic items, is the idea that, through a kind of informal abductive process, the agent can begin to form a representation of the latent concepts that explain its sensorimotor experience.

However, unlike the Schema System, SODA explicitly concentrates on learning in a continuous world, developing a useful continuous-to-discrete abstraction that improves learning. Using this abstraction, SODA uses reinforcement learning to learn to perform tasks, rather than just learning a model of the world.

2.2.2 Constructivist Learning Architecture

The Constructivist Learning Architecture (CLA; Chaput, 2004, 2001; Chaput & Cohen, 2001; Cohen, Chaput, & Cashon, 2002) is another computational model of child development that uses SOMs (Section 2.1.3) as its feature representation, and constructs high-level features by using higher level SOMs that learn the correlations of features on two or more lower-level SOMs. The architecture has been used successfully to model development of infants' perception of causation, and other child developmental processes. It also successfully replicates Drescher's results from the Schema System operating in the micro-world (Chaput, Kuipers, & Miikkulainen, 2003). In addition, it has been used to implement a learning robot controller for a simple foraging task in a simulated robot.

Part of the power of CLA is that it recognizes that an agent's actual sensorimotor experience is a very small subset of the set of experiences that can be represented in its sensorimotor system, and it uses data-driven, unsupervised, competitive learning in SOMs to focus the search for high-level features and actions on the regions of the state space in which the agent's sensorimotor experience resides. The rest of the power comes from its hierarchical structure, and the fact that it explicitly uses the *factored* and *compositional* structure of high-level percepts and actions. That is, high-level features are a small subset of the set of possible combinations of lower-level features, that are a small subset of combinations of still lower-level features, continuing on down to the primitive features. Assuming this kind of structure, CLA can further focus the search by first learning the lowest level of features, then conducting the search for higher level features in the reduced space formed by combining the existing lower level feature sets.

Like the Schema System, CLA does not explicitly address the continuous-to-discrete abstraction, but merely assumes that such an abstraction exists, whereas SODA explicitly deals with learning such an abstraction. SODA, however, learns features with a single, monolithic SOM on one large sensor group. Extending SODA to learn a factored continuous-to-discrete abstraction using techniques from CLA is an interesting direction for future work that is discussed in Section 7.3.

2.2.3 Bootstrap Learning for Place Recognition

Kuipers and Beeson's (2002) bootstrap learning system for place recognition uses unsupervised clustering on sensory images, and the topology abduction methods of the Spatial Semantic Hierarchy to bootstrap training data for a supervised learning algorithm that learns to recognize places directly from their sensory images.

The method assumes that a major sensor group has been provided or found through a grouping method like that of Pierce & Kuipers (1997). It also assumes an existing set of TF and HC control-laws such as those learned by SODA. Using these control laws the agent moves through the environment, from one distinctive state to another, collecting sensor images from distinctive states. The agent then clusters the set of sensor images into a set of views, choosing a number of clusters small enough to ensure that there is no image variability within a distinctive state (i.e., every distinctive state has the same view), but assuming that such a small set of clusters will create perceptual aliasing (i.e. more than one state has the same view). The algorithm then uses an expensive exploration procedure combined with the SSH's topology abduction to produce a set of sensory images associated with their correct place labels. These data are then used to train a supervised learning algorithm (k-nearest neighbor) to immediately recognize places from their sensory images, without any exploration or topology abduction.

2.2.4 Learning an Object Ontology with OPAL

The Object Perception and Action Learning (OPAL) (Modayil & Kuipers, 2004, 2006), assumes the existence of a method for constructing an occupancy-grid representation of the world from range data, and uses a hierarchy of clustering and action learning methods to progressively distinguish dynamic objects from the static background, track the objects as they move, register various views of the same object into a coherent object model, classify new instances of objects based on existing object models, and learn the actions that can be performed on different classes of objects. Each one of these steps builds on the representations learned in the one below it, forming a multi-layer bootstrap learning system for an object and action ontology.

2.2.5 Other methods of temporal abstraction

Ring (1994, 1997), developed two methods for temporal abstraction in reinforcement learning, neither one based on SMDPs. The focus of both methods, however, was on reinforcement learning in non-Markov problems. By using temporal abstraction, he was able to develop reinforcement learning methods to work on *k-Markov* problems, that is, problems in which the the next state is dependent only on the current state and some finite k immediately previous states.

Ring's first method, *Behavior Nets* (Ring, 1994) measured how often temporally successive actions co-occur. This data was used to chain together actions that frequently occur in sequence to form new, ballistic "macro actions"; these actions are added to the agent's repertoire of available actions, and also are available for further chaining. These new actions are similar in some respects to the compound actions in Drescher's Schema System (Section 2.2.1); however, Drescher's compound actions form trees of connected schemas that converge upon some goal state, while Behavior Nets form single chains and are not goal-directed. Although this work was addressed to the problem of non-Markov reinforcement learning, it is possible that such macros may be useful in high-diameter reinforcement learning problems as well. However, as shown in Chapter 5, open-loop behaviors such as these are less effective in physical robots with noisy motor systems, and closed loop methods are preferred for usable high-level actions.

Ring's second method, *Temporal Transition Hierarchies* (Ring, 1994, 1997), does not create new actions. Rather, it builds a hierarchy of units that encode into the current state representation information from progressively more distant time steps in the past, forming a kind of task-specific memory. The agents' policy function is implemented as a neural network with a single layer of weights mapping from the state representation to the primitive actions. The state representation initially consists only of units representing primitive perceptual features, such as walls bordering each side of a cell in a grid-world. As the agent learns, it monitors the changes in the weights of the network, and identifies weights that are not converging on fixed values. When it identifies such a weight w_{ij}, it adds a new unit that takes input from the previous time step. The output of this new unit is used to dynamically modify the value of *weight* w_{ij} based on the previous state. Each new unit's weights are likewise monitored, and the system progressively creates a cascading hierarchy of units looking further back in time. This method, combined with Q-Learning for learning state-action value functions forms the CHILD algorithm for continual learning (Ring, 1997). CHILD is a very effective means of dealing with partially observable environments, i.e., environments in which many places produce the same perceptual view. Possible extensions of SODA to such environments, including the use of CHILD, are discussed in Section 7.1.

2.3 Automatic Feature Construction for Reinforcement Learning

Tabular value-function reinforcement learning methods cannot learn in large discrete state spaces without some mapping of the continuous space into a discrete space. Furthermore, even large discrete spaces make learning difficult, because each state, or state-action pair, must be visited in order to learn the value function (Sutton & Barto, 1998, Ch. 8). To deal with this it is necessary to extract or construct features from the sensory vector that provide a usable *state abstraction* from which the agent can learn a policy. Many times the state abstraction is constructed by hand and given *a priori*, as with a popular method called *tile coding* (also called CMAC; Sutton & Barto, 1998, Sec. 8.3.2). Unlike these methods, SODA's GNG network learns a state abstraction from input. Two other methods that learn a state abstraction from input are the U-Tree algorithm, and value-function learning using backpropagation.

2.3.1 U-Tree Algorithm

The *U-Tree* algorithm (McCallum, 1995) progressively learns a state abstraction in the form of a decision tree that takes as input vectors from a large space of discrete, nominal features and splits the space on the values of specific features, forming a partitioning of the state space at the leaves of the tree. Splits are selected using a statistical test of utility, making distinctions only where necessary to improve the Q function. In addition, the algorithm considers not only the current state, but the recent history of states when making splits, allowing it to act as a compact state memory in k-Markov problems.

U-Tree's principal demonstration was in a simulated highway driving task with a state vector of 8 features having from 2 to 6 possible discrete, nominal values, for a total of 2592 total perceptual states and much of the state hidden. The individual features and actions were related to a set of *visual routines* coded into the agent that encapsulated a great deal of prior task knowledge. For example, one feature indicates whether object in the current gaze is a car, road, or road shoulder. The algorithm eventually discovered a state abstraction with fewer than 150 states within which it could perform the task.

It is unclear from this experiment how U-Tree would perform using as input, for example, a raw laser rangefinder vector with approximately 500^{180} perceptual states. Nevertheless, U-tree could potentially be useful once an initial discrete interface to the robot has been learned. The possibility of replacing Sarsa in SODA with U-Tree or other methods is discussed in Section 7.1.

In addition, supervised learning algorithms that learn decision trees for classification problems can discover and use thresholds for splitting continuous-valued attributes (Mitchell, 1997). It may be possible to implement similar continuous attribute splitting to apply U-Tree to continuous state spaces, but there are problems with any method that abstracts a continuous space simply by partitioning it. MDP-based reinforcement learning methods assume that the environment obeys the Markov property – that there is no perceptual aliasing – but partition-based state abstractions automatically alias all states within a partition. If the size of the action is not well matched to the granularity of the state partitioning, action can be highly uncertain, since it is impossible to know whether executing an action will keep the robot within the same perceptual state or carry it across the boundary into a new state. Furthermore, even if the scale of the actions is large enough to always move the robot into a new perceptual state, small amounts of positional variation, especially in orientation or other angular measurements, can often lead to large differences in the outcome of actions. Although SODA partitions the input space using a SOM, it defines high-level actions that operate within the perceptual partition (neighborhood) defined by a single SOM unit. SODA reduces the positional uncertainty induced by the SOM by hill-climbing to perceptually distinctive states within each perceptual neighborhood.

2.3.2 Backpropagation

One popular non-tabular method of value-function approximation in reinforcement learning is training a feed-forward neural network using backpropagation to approximate the Q or V functions. While not explicitly a method of feature construction, backpropagation networks implicitly learn an intermediate representation of their input in their hidden layer, and thus can be said to be performing a form of state abstraction.

Unfortunately, these features are typically not accessible in a form that allows easy reuse of the features for other purposes, such as for further bootstrap learning. Typically these features are encoded in distributed activation patterns across the hidden units in such a way that understanding the encoded features requires further analysis using methods like clustering and principal component analysis on the hidden unit activations. Given this difficulty, it is not clear what benefit the backprop hidden layer provides over doing the similar analyses on the original inputs.

Moreover, even these networks often require substantial manual engineering of the input features to work successfully. For example, TD-Gammon (Tesauro, 1995) is often cited as an example of a reinforcement learning method that used a backpropagation network to learn to play grandmaster-level backgammon by playing games against itself. However, TD-Gammon incorporated significant prior knowledge of backgammon into its input representation (Pollack & Blair, 1997; Sutton & Barto, 1998) before any backpropagation learning took place.

Finally, unlike state abstraction using linear function approximators like tile coding, some of which are proven to converge near the optimal policy (Gordon, 2000), backpropagation is non-linear, and no such convergence proofs exist for it.

2.4 SODA and Tabula Rasa Learning

Although one of SODA's goals is to minimize the need for human prior knowledge in the learning process, it must be acknowledged that there is no truly *tabula rasa* learning, and SODA is not entirely free of prior knowledge. This prior knowledge can be divided into three basic categories: (1) general learning methods, (2) domain-specific knowledge assumed to come from a lower-level bootstrap learning process, and (3) the parameter settings of SODA's constituent learning methods.

First, SODA contains prior knowledge of a variety of *general* learning methods and representations, specifically those described in Section 2.1: the SSH, Homeostatic-GNG, Sarsa(λ), and Options. These methods embed in them assumptions about the nature of the world:

- that the world is generally continuous and the agent travels through it on a connected path,
- that high-dimensional sensory input is distributed in a way that can be modeled usefully by a topological graph structure like the GNG,
- that the spatial world has regularities that allow hill-climbing and trajectory following,

- and as Hume (1777) pointed out, that the past is a reasonable guide to the future.

Second, SODA assumes that certain domain-specific knowledge has already been learned by existing methods, namely:

- a main sensor group or modality that has been separated out from other sensors on the robot,
- an abstract motor interface that has been learned through interaction with the environment.

These items of knowledge can be learned using the methods of Pierce & Kuipers (1997), which are assumed to form the bootstrap learning layer beneath SODA. These assumptions are described more formally in Section 3.1.

Third, the parameter settings of SODA's various learning methods embed some prior knowledge about the world, such as how long the agent must explore the world in order to learn a good feature set (Chapter 4), or the degree of stochasticity in the environment, which determines an appropriate setting for Sarsa's learning rate α. Section 7.4, discusses parameter setting in more detail, and proposes some methods by which the need for such prior knowledge can be further reduced.

2.5 Conclusion

To summarize, SODA rests on three foundational areas: (1) the causal and control levels of the Spatial Semantic Hierarchy provide a continuous-to-discrete abstraction of action that reduces the agent's task diameter and reduces state uncertainty between actions; (2) hierarchical reinforcement learning methods allow the automatic construction of high-level actions; and (3) self-organizing maps learn a state abstraction that is concentrated in the important regions of the state space, providing a discrete abstraction for reinforcement learning, and continuous features usable by the SSH control level. The next chapter describes how the SODA agent combines these three foundational methods for learning in high-diameter, continuous environments with little prior knowledge.

Chapter 3

Learning Method and Representation

The previous chapter reviewed the prior work on which SODA is based, as well as a variety of related work. This chapter presents a formal description SODA's learning method, followed by detailed descriptions of the various components of the method. The first section lays out the formal assumptions of the methods and the steps of the learning algorithm. The following sections describe the state representation used for learning trajectory-following (TF) and hill-climbing (HC) actions, the formal definition of TF and HC actions as Options, and alternative formulations of TF and HC actions that are used for comparison the experiments in subsequent chapters.

3.1 Overview

The SODA algorithm can be characterized formally as follows. Given

- a robot with a sensory system providing experience at regular intervals as a sequence of N-dimensional, continuous *sensory vectors* $\mathbf{y}_1, \mathbf{y}_2, \ldots$, where every $\mathbf{y}_t \in \mathbb{R}^N$;

- a continuous, M-dimensional motor system that accepts at regular intervals from the agent a sequence of *motor vectors* $\mathbf{u}_1, \mathbf{u}_2, \ldots$, where every $\mathbf{u}_t \in \mathbb{R}^M$;
- an almost-everywhere-continuous world, in which small actions usually induce small changes in sensor values, though isolated discontinuities may exist; and
- and a scalar *reward signal*, $r_1, r_2, \ldots$, that defines a high-diameter task, such that properly estimating the value of a state requires assigning credit for reward over a long sequence of motor vectors $\mathbf{u}_t, \ldots \mathbf{u}_{t+k}$),

the SODA algorithm consists of five steps:

1. *Define a set of discrete, local primitive actions* $\mathcal{A}^0$. First, using methods developed by Pierce & Kuipers (1997), learn an *abstract motor interface*, i.e., a basis set of orthogonal motor vectors $\mathcal{U} = \{\mathbf{u}^0, \mathbf{u}^1, \ldots \mathbf{u}^{n-1}\}$ spanning the set of motor vectors $\mathbf{u}_t$ possible for the robot. Then define $\mathcal{A}^0$ to be the set of $2n$ motor vectors formed by the members of $\mathcal{U}$ and their opposites: $\mathcal{A}^0 = \mathcal{U} \cup \{-\mathbf{u}^i | \mathbf{u}^i \in \mathcal{U}\}$.

2. *Learn a set $\mathcal{F}$ of high-level perceptual features*. Exploring the environment with a random sequence of $\mathcal{A}^0$ actions, train a Growing Neural Gas network (GNG) with the sensor signal $\mathbf{y}_t$ to converge to a set of high-level features of the environment. For each unit i in the GNG with weight vector $\mathbf{w}_i$ there is an *activation function* $f_i \in \mathcal{F}$ such that:

$$f_i(\mathbf{y}) = exp\left(-\frac{|\mathbf{w}_i - \mathbf{y}|^2}{\sigma^2}\right), \tag{3.1}$$

$$\sigma = \frac{\sum_i \sum_{j=1}^{i} n(i,j)|\mathbf{w}_i - \mathbf{w}_j|}{\sum_i \sum_{j=1}^{i} n(i,j)}, \tag{3.2}$$

$$n(i,j) = \begin{cases} 1, & \text{if } i,j \text{ adjacent} \\ 0, & \text{otherwise.} \end{cases} \tag{3.3}$$

These equations define each feature function $f_i(\mathbf{y})$ as a Gaussian kernel on $\mathbf{w}_i$ and with a standard deviation equal to the average distance between adjacent units in the GNG.

3. *Define trajectory-following control-laws.* For each distinctive state defined by $f_i \in \mathcal{F}$, define trajectory-following control laws that take the agent to a state where a different feature $f_j \in \mathcal{F}$ is dominant. Methods for definining trajectory-following control laws are described in Section 3.3.

4. *Define a hill-climbing (HC) control law for each $f_i \in \mathcal{F}$.* For each f_i, in the context where $\arg\max_{f \in \mathcal{F}} = f_i$, the controller HC_i climbs the gradient of f_i to a local maximum. Methods for defining hill-climbing controllers are described in Section 3.4.

5. *Define a set of higher-level actions $\mathcal{A}^1$.* Each $a^1_j \in \mathcal{A}^1$ consists of executing one TF control law, and then hill-climbing on the resulting dominant feature f_j. At this point the agent has abstracted its continuous state and action space into a discrete Markov Decision Process (MDP) with one state for each feature in $\mathcal{F}$, and the large-scale actions in $\mathcal{A}^1$. At the $\mathcal{A}^0$ level, this abstraction forms a Semi-Markov Decision process, as described in Section 2.1.2, in which the choice of $\mathcal{A}^0$ action depends on both the current state and the currently running $\mathcal{A}^1$ action. In the case where there is perceptual aliasing, the abstract space forms a Partially Observable Markov Decision Process (POMDP); extending SODA to the POMDP case is discussed further in Section 7.1.

Tasks such as robot navigation have considerably smaller diameter in this new $\mathcal{A}^1$ state-action space than in the original $\mathcal{A}^0$ space, allowing the agent to learn to perform them much more quickly

3.2 Top^n State Representation

Trajectory-following and hill-climbing Options operate within an individual perceptual neighborhood. In order to learn policies for them, the learner needs a state representation that provides more resolution than the winning GNG unit can provide by itself. Therefore,

```
$\pi_i^{\mathrm{TF}}$: Open-loop Trajectory-follow on $a_i^0$:
        $f_w \leftarrow \arg\max_{f \in \mathcal{F}} f(\mathbf{y})$
        while $f_w = \arg\max_{f \in \mathcal{F}} f(\mathbf{y})$:
                execute action $a_i^0$
$f_{ds} \leftarrow \arg\max_{f \in \mathcal{F}} f(\mathbf{y})$
```

```
$\pi_{ij}^{\mathrm{TF}}$: Closed-loop Trajectory-follow on $a_i^0$ in feature $f_j$:
        $f_w \leftarrow \arg\max_{f \in \mathcal{F}} f(\mathbf{y})$
        while $f_w = f_j$:
                $a \leftarrow argmax_{b \in \mathcal{A}_{ij}^{\mathrm{TF}}} Q_{ij}^{\mathrm{TF}}(\mathbf{y}, b)$
                execute action $a$
$f_{ds} \leftarrow \arg\max_{f \in \mathcal{F}} f(\mathbf{y})$
```

Table 3.1: Open-loop and Closed-loop Trajectory-following. *Top:* The open-loop trajectory-following macro repeats the same action until the current SOM winner changes. *Bottom:* The closed-loop Option policy chooses the action with the highest value from the Option's action set. The action set, defined in Equation (3.7) is constructed to force the agent to make progress in the direction of a_i while being able to make small orthogonal course adjustments.

the TF and HC Options described in Sections 3.3 and 3.4 use a simple new state abstraction derived from the GNG, called the Top^n representation.

If $i_1, i_2, ..., i_{|\mathcal{F}|}$ are the indices of the feature functions in $\mathcal{F}$, sorted in decreasing order of the value of $f_i(\mathbf{y})$, then $\mathrm{Top}^n(\mathbf{y}) = \langle i_1, ..., i_n \rangle$. This representation uses the GNG prototypes to create a hierarchical tessellation of the input space starting with the Voronoi tessellation induced by the GNG prototypes. Each Voronoi cell is then subdivided according to the next closest prototype, and those cells by the next, etc. This tuple of integers can be easily hashed into an index into a Q-table for use as a state representation. This state representation allows the agent to learn TF and HC Options policies using simple, tabular reinforcement learning methods like Sarsa(λ) using existing information from the GNG, instead of needing an entirely new learning method for Option policies.

3.3 Trajectory Following

The purpose of trajectory-following actions is to move the robot from one perceptual neighborhood to another by moving the robot through a qualitatively uniform region of the environment. The classic example is following a corridor: beginning in a pose aligned with the corridor, the robot moves down the corridor until it reaches a qualitatively different region of space, e.g. an intersection or a dead end.

The simplest form of trajectory-following is to repeat a single action until the SOM winner changes, as described in Table 3.1. Chapter 5 will show that this sort of *open-loop macro* is unreliable when a realistic amount of noise perturbs the robot's trajectory. Angular deviations from motor noise accumulate, causing large deviations in the trajectory, sometimes pushing the robot off of the "side" of the trajectory. As a result, the end state of the TF actions varies greatly, making their outcomes highly unreliable. The solution described below constructs closed-loop TF Options that can correct for the perturbations of noise and keep the robot moving along the trajectory that best matches the current perceptual prototype. For example, in corridor following, the closed-loop TF Option is expected to move down the hallway, correcting for deviations to maintain the view looking forward as much as possible. As described in Section 2.1.2, each Option is defined by an initiation set $\mathcal{I}$, a termination function β, a pseudo-reward function R, an action set $\mathcal{A}$, and a policy π. The remainder of this section defines these elements for trajectory-following Options.

To achieve reliable trajectories, SODA defines a closed-loop TF Option for each combination of prototype and primitive action: $\{TF_{ij} | \langle \mathbf{a}_i, f_j \rangle \in \mathcal{A}^0 \times \mathcal{F}\}$. The initiation set of each TF Option is the set of states[1] where its prototype is the winner:

$$\mathcal{I}_{ij}^{\mathrm{TF}} = \{\mathbf{y} | j = \arg\max_k f_k(\mathbf{y})\}. \tag{3.4}$$

---

[1] To simplify the terminology, these descriptions refer to the input vector $\mathbf{y}$ as if it were the state s. Since $\mathbf{y}$ is a function of s, this terminology is sufficient to specify the Options.

The Option terminates if it leaves its prototype's perceptual neighborhood:

$$\beta_{ij}^{\mathrm{TF}}(\mathbf{y}) = \begin{cases} 0 & \text{if } \mathbf{y} \in \mathcal{I}_{ij}^{\mathrm{TF}} \\ 1 & \text{otherwise.} \end{cases} \tag{3.5}$$

Each TF Option's pseudo-reward function is designed to reward the agent for keeping the current feature value as high as possible for as long as possible, thus:

$$R_{ij}^{\mathrm{TF}}(\mathbf{y}) = \begin{cases} f_j(\mathbf{y}) & \text{if not terminal,} \\ 0 & \text{if terminal.} \end{cases} \tag{3.6}$$

In order to force the TF Options to make progress in some direction (instead of just oscillating in some region of high reward), each Option is given a limited action set consisting of a *progress action* selected from $\mathcal{A}^0$, plus a set of corrective actions formed by adding a small component of each orthogonal action in $\mathcal{A}^0$:

$$\mathcal{A}_{ij}^{\mathrm{TF}} = \{\mathbf{a}_i\} \cup \{\mathbf{a}_i + c_{tf}\mathbf{a}_k | \mathbf{a}_k \in \mathcal{A}^0, \mathbf{a}_k^T \mathbf{a}_i = 0\}. \tag{3.7}$$

Lastly, the Option policy π_{ij}^{TF} is learned using tabular Sarsa(λ), using the $\mathrm{Top}^n(\mathbf{y})$ state representation described in Section 3.2, and the actions $\mathcal{A}_{ij}^{\mathrm{TF}}$.

This definition of trajectory-following actions as Options allows the agent to learn closed-loop trajectory-following control for each combination of perceptual feature and primitive action. Experiments in Chapter 5 show that the learned Options are far more reliable than open-loop TF in a robot with realistic motor noise.

3.4 Hill-climbing

Once a SODA agent has executed a trajectory-following action to carry the robot from one distinctive state into the neighborhood of another, it then performs hill-climbing to reach a new distinctive state. Hill-climbing actions remove positional uncertainty that may have accumulated during the execution of a trajectory-following action by moving the robot to a fixed point in the environment defined by the local maximum of the activation of the current winning SOM unit. One way of moving to the local maximum is to estimate the gradient of the winning feature function f and follow it upward. An alternative method, used by SODA, is to learn a hill-climbing Option for each perceptual neighborhood, that climbs to a local maximum.

Section 3.4.1 below describes two means of gradient-estimate hill-climbing: sampling the feature changes from each action and using action models to estimate the feature changes. These methods have drawbacks: the former is inefficient, and the latter requires substantial prior knowledge of the dynamics of the sensorimotor system. Section 3.4.2 describes how HC actions can instead be formulated as Options and learned using reinforcement learning, Chapter 5 will the show that learning to hill-climb in this fashion results in actions that are as efficient as those using a hand-built action model, but that do not need prior knowledge of the action dynamics.

3.4.1 HC using Gradient Approximation

Ideally, a hill-climbing action would move the agent exactly in the direction of the feature gradient (the greatest increase in feature value). Unfortunately, knowledge of the exact direction of the gradient is not available to the agent. In addition, it is likely that none the primitive actions $\mathcal{A}^0$ will move the agent exactly in the gradient direction. However, it is possible to approximate the gradient by selecting the primitive action that increases the value of the feature by the greatest amount at each step. The change in a feature value induced by a particular action, or the *feature-action delta* is denoted $G_{ij}(t)$, and defined as

$$G_{ij}(t) \triangleq f_i(y_{t+1}) - f_i(y_t) \text{ given } \mathbf{u}_t = a_j^0. \tag{3.8}$$

When the time t is obvious in context (e.g. the current time at which the agent is operating), the feature-action delta will be abbreviated simply as G_{ij}.

Hill-climbing using gradient approximation is accomplished using the simple greedy policy shown in Table 3.2. This policy chooses the action with the greatest estimated

```
$\pi_i^{\mathrm{HC}}$: Hill-climb on $f_i$:
    while not $\beta_i^{\mathrm{HC}}$:
        $w \leftarrow \arg\max_j G_{ij}$
        execute action $a_w^0$
```

Table 3.2: Pseudo-code for hill-climbing policy π_i^{HC} using gradient estimation. The value of G_{ij} is the estimated change in feature f_i with respect to primitive action a_j^0. G_{ij} can be determined either by sampling the change induced by each action or by using an action model to predict the change. Sampling is simple and requires no knowledge of the robot's sensorimotor system or environment dynamics, but is expensive, requiring $2|\mathcal{A}^0| - 2$ sampling steps for each movement

feature-action delta and executes it, terminating when all the estimated deltas are negative. The two methods described in this section differ in how they obtain the estimate of G_{ij}, and in how the termination condition β_i^{HC} is defined.

Sampling the Deltas

The simplest way to estimate the deltas is by applying each action a_i^0, recording the feature change, and reversing the action (by applying $-a_i^0$). The termination criterion for this policy is simply to stop when all the estimates are negative:

$$\beta_i^{\mathrm{HC}} = 1 \text{ iff } \max_j G_{ij} > 0. \tag{3.9}$$

This method easily produces an estimate for each action, but it requires $2|\mathcal{A}^0| - 2$ exploratory actions for each action "up the hill." (The two steps are saved by caching the feature change from the last up-hill action, so that it is not necessary to sample back in the direction from which the agent came.) These extra actions are very costly, even with small action spaces.

Predicting the Deltas with Action Models

Although the purpose of SODA is to construct an agent that learns with little *a priori* knowledge from human engineers, a human-engineered predictive action model for hill-climbing makes an useful comparison with the autonomously learned HC Options described in Section 3.4.2. A controller using a predictive model can dispense with costly gradient sampling and simply use its model to predict the gradient. In this case, the model would take the form of a function D such that

$$\hat{\mathbf{y}}_{i,t+1} = D(\mathbf{y}_t, a_i), \tag{3.10}$$

where $\hat{\mathbf{y}}_{t+1}$ is the estimated value of $\mathbf{y}$ after executing action a_i at time t. Using this model, the feature-action delta can be estimated as:

$$G_{ij}(t) \approx f_j(\hat{\mathbf{y}}_{i,t+1}) - f_j(\mathbf{y}_t). \tag{3.11}$$

When using approximate models, in some cases the gradient magnitude will be near the precision of the model, and approximation error may cause a sign error in one or more of the estimates. In this case the process may fail to terminate, or may terminate prematurely, under the termination condition in Equation 2.10. Without the termination condition, however, the greedy hill-climbing policy will cause the agent to "hover" near the true local maximum of the feature, with little net increase in $f_i(\mathbf{y})$ over time. Thus it is possible to define a new, k-Markov stopping condition in which the Option terminates if the average step-to-step change in the feature value over a finite moving window falls below a small fixed threshold:

$$\beta_i^{\mathrm{HC}}(\mathbf{y}_{t-c_w}, \ldots, \mathbf{y}_t) = \left\lceil c_{stop} - \frac{\sum_{i=0}^{k-1} |\Delta^{t-k} f_i(\mathbf{y})|}{c_w} \right\rceil, \tag{3.12}$$

where c_w is the window size, c_{stop} is a constant threshold and

$$\Delta^t f_i(\mathbf{y}) = f_i(\mathbf{y}_t) - f_i(\mathbf{y}_{t-1}) \tag{3.13}$$

is the change in feature value at time t.

Chapter 5 describes a predictive model of sensorimotor dynamics specifically engineered for the robot and environments used in this dissertation, and compares its performance with both delta sampling and learned HC Options, described in the next section.

3.4.2 Learning to Hill-Climb with RL

The greedy, gradient-based methods in Section 3.4.1 above have some drawbacks. Specifically, the sampling-based method wastes many actions gathering information, while the model-based method requires an action model D. Such a model could be provided *a priori*, but one of the objectives of SODA is to develop a system that can learn with no such prior knowledge. Although it may be possible to learn D from interaction with the environment using supervised learning techniques, that would require adding yet another learning method to the system. Given that the system already learns to follow trajectories by reinforcement lexarning, it is natural to use the same methods for hill-climbing as well.

With hard-coded policies (Table 3.2), SODA would need only a single hill-climbing policy that worked in all perceptual neighborhoods. In contrast, when hill-climbing is learned, each feature presents a different pseudo-reward function, and thus requires a separate HC Option, HC_i, for each f_i in $\mathcal{F}$. As with the TF Options, the initiation set of each HC Option is the perceptual neighborhood of that Option's corresponding GNG prototype:

$$\mathcal{I}_i^{\mathrm{HC}} = \{\mathbf{y} | \arg\max_j f_j(\mathbf{y}) = i\}. \tag{3.14}$$

Termination, however, is more complicated for hill-climbing. The perceptual input is unlikely to ever match any perceptual prototype exactly, so the maximum feature value attainable in any neighborhood will be some value less than 1. Because it is difficult or impossible to know this value in advance, the stopping criterion is not easily expressed as a function of the single-step input. Rather, the Options use the same k-Markov termination function described above in Equation 3.12 for use with gradient approximation.

The task of the HC Option is to climb the gradient of its feature as quickly as possible, and terminate at the local maximum of the feature value. On nonterminal steps the pseudo-reward for each HC Option is a shaping function (Ng, Harada, & Russell, 1999) consisting of a constant multiple of the one-step change in f_i, minus a small penalty for taking a step; on terminal steps, the reward is simply f_i itself:

$$R_i^{\mathrm{HC}} = \begin{cases} c_{R1}\Delta f_i(\mathbf{y}) - c_{R2} & \text{if not terminal,} \\ f_i(\mathbf{y}) & \text{if terminal} \end{cases} \tag{3.15}$$

Finally, the action set for HC Options is just the set of primitive actions:

$$\mathcal{A}_i^{\mathrm{HC}} = \mathcal{A}^0. \tag{3.16}$$

Hill-climbing policies learned in this way do not use explicit estimates of the feature-action delta G_{ij}. When they are learned using standard temporal-difference policy learning methods like Sarsa (Section 2.1.2), they do learn a similar function, i.e. the state-action value function $Q(\mathbf{y}, a)$. This function can be thought of as a kind of "internal gradient," on which the controller hill-climbs. One important difference between the learned value function and the true gradient, however, is that because the learning algorithm distributes credit for reward changes over the sequence past actions, it is possible for the controller to sometimes achieve higher feature activations than possible with greedy, gradient-based policies. This is possible because the learned policies can perform down-gradient actions that eventually lead to a higher ultimate reward. An example of this is shown in Chapter 5.

This section has given the formal definition for SODA's learned hill-climbing Options and described two alternatives using gradient approximation. Chapter 5 presents experiments showing that the learned Options perform as well as the other two HC methods, without requiring extensive prior knowledge, or expensive sampling. When executed after a trajectory-following Option (Section 3.3) HC Options form the second step in SODA's two-step $\mathcal{A}^1$ actions used for high-level navigation. The next section concludes this chapter with a summary of the formal definition of SODA and reviews the questions answered by the experiments in the next three chapters.

$\mathcal{A}^0$ (Primitive) Actions	
Action output	Application of motor a vector from basis $\mathbf{u}^0, ..., \mathbf{u}^m$
GNG Feature learning	
Input	Raw input vector $\mathbf{y}$
Outputs	Perceptual prototypes i Continuous features f_i $\text{Top}^n(\mathbf{y})$ State representation for TF and HC Options.
Trajectory-following Options	
Policy	Learned with RL
Initiation	$\mathcal{I}_{ij}^{\text{TF}} = \{\mathbf{y} \mid j = \arg\max_k f_k(\mathbf{y})\}$
Termination	$\beta_{ij}^{\text{TF}}(\mathbf{y}) = \begin{cases} 0 & \text{if } \mathbf{y} \in \mathcal{I}_{ij}^{\text{TF}} \\ 1 & \text{otherwise.} \end{cases}$
State Representation	$\text{Top}^n(\mathbf{y})$
Actions	$\mathcal{A}_{ij}^{\text{TF}} = \{\mathbf{a}_i\} \cup \{\mathbf{a}_i + c_{tf}\mathbf{a}_k \mid \mathbf{a}_k \in \mathcal{A}^0 \mathbf{a}_k^T \mathbf{a}_i = 0\}$
Reward	$R_{ij}^{\text{TF}}(\mathbf{y}) = \begin{cases} f_j(\mathbf{y}) & \text{if not terminal,} \\ 0 & \text{if terminal.} \end{cases}$
Hill-climbing Options	
Policy	Learned with RL
Initiation	$\mathcal{I}_{i}^{\text{HC}} = \{\mathbf{y} \mid i = \arg\max_j f_j(\mathbf{y})\}$
Termination	$\beta_i^{\text{HC}}(\mathbf{y}_{t-c_w}, ..., \mathbf{y}_t) = \left\lceil c_{stop} - \frac{\sum_{i=0}^{k-1} \lvert\Delta^{t-k} f_i(\mathbf{y})\rvert}{c_w} \right\rceil,$
State Representation	$\text{Top}^n(\mathbf{y})$
Actions	$\mathcal{A}^0$
Reward	$R_i^{\text{HC}} = \begin{cases} c_{R1}\Delta f_i(\mathbf{y}) - c_{R2} & \text{if not terminal,} \\ f_i(\mathbf{y}) & \text{if terminal.} \end{cases}$
$\mathcal{A}^1$ Actions	
Policy	Hard-coded: TF+HC
Initiation	Same as TF component
Termination	Same as HC component
State Representation	N/A
Actions	TF, HC
Reward	N/A

Table 3.3: Summary of the components of SODA.

3.5 Conclusion

Table 3.3 summarizes the formal specification of the components of SODA: the primitive actions $\mathcal{A}^0$, GNG for feature learning, TF and HC Options, and $\mathcal{A}^1$ high-level actions used for high-diameter navigation. The next three chapters present experimental results that answer several questions about the method:

- Can the GNG learn a feature set that covers a robot's environment from data gained through random exploration? Chapter 4 introduces a simulated mobile robot and two navigation environments and shows that SODA learns rich feature sets for both environments.

- Do the learned TF and HC Options perform as well or better than as obvious hand-coded alternatives? Chapter 5 presents experiments comparing open-loop and closed-loop TF, showing that learned closed-loop TF Options produce longer, more reliable trajectories. In addition, the chapter compares learned HC Options against HC by sampling feature-deltas, and HC using feature-delta estimates from a hand-coded predictive model of the robot's perceptual dynamics; these experiments show that while all three methods achieve comparable final activation levels, the learned Options are far more efficient than sampling, while not requiring the *a priori* knowledge needed for the hand-coded model.

- Do the $\mathcal{A}^1$ actions reduce task diameter? Chapter 6 shows that SODA reduces the diameter of robot navigation tasks by an order of magnitude.

- Do the $\mathcal{A}^1$ actions enable the agent to learn to navigate more quickly? Experiments in Chapter 6 show dramatic speedups in navigation using $\mathcal{A}^1$ actions over using $\mathcal{A}^0$ actions.

- What is the contribution of the HC step in the $\mathcal{A}^1$ actions? An ablation study in Chapter 6 shows that using hill-climbing makes state transitions more reliable, and reduces task diameter over navigating using TF Options alone.

To summarize, this chapter has presented the formal description of SODA. The following chapters present empirical evaluation of SODA's feature learning, trajectory-following and hill-climbing Options, and high-diameter navigation.

Chapter 4

Learning Perceptual Features

In the first phase of learning, a SODA agent explores the environment, collecting sensor observations. As the observations are received they are given as training examples to an Homeostatic-GNG (Section 2.1.3) network that learns a set of perceptual features in the form of prototypical sensor views. This chapter describes experimental results from running this phase of learning on a simulated robot in two environments: the first is a small hand-built environment, and the second a large, realistic environment, derived from an actual floor of a building on the University of Texas campus. Example feature sets learned in each environment show that the GNG learns a wide variety of features covering the sensory space of the robot in the environment.

4.1 Experimental Setup

All the experiments in this dissertation were performed using the Stage robot simulator (Gerkey, Vaughan, & Howard, 2003). Stage is a widely-used simulator for two-dimensional mobile robot simulations. It can simulate a wide variety of robot hardware, and allows the fixed "architecture" of the robot's environment to be described easily using bitmapped layouts.

The robot configuration used in the experiments simulates an RWI Magellan Pro robot. The robot was equipped with a laser range finder reading 180 readings at 1° intervals over the forward semicircle around the robot, with a maximum range of 8000 mm. The Stage laser rangefinder model returns ranges in mm, with no noise. Sensor noise was simulated through a two-stage process of alternately adding Gaussian error and rounding, applied individually to each range reading. This model provides a good characterization of the error on a SICK LMS laser rangefinder. However, the actual noise in a SICK LMS is very small, approximately $\pm 10\mathrm{mm}$, and it is not a significant cause of uncertainty.

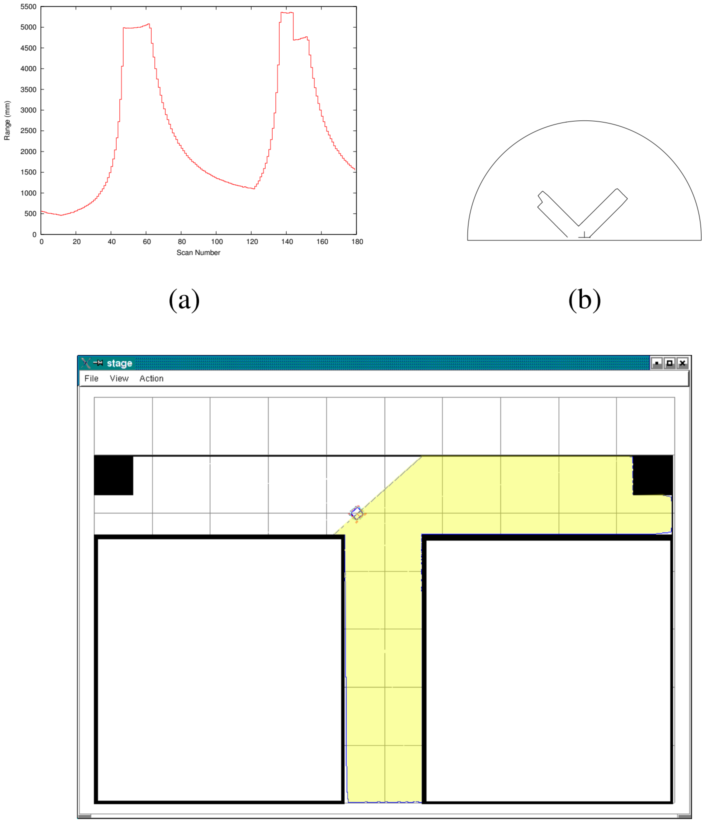
The robot was also equipped with a differential-drive base, taking two continuous control values, linear velocity v and angular velocity ω. The Stage simulator does not model positional error internally, so motor noise was simulated by perturbing the motor commands thus:

$$\hat{v} = \mathcal{N}(v, k_{vv}v + k_{v\omega}\omega) \tag{4.1}$$

$$\hat{\omega} = \mathcal{N}(\omega, k_{\omega v}v + k_{\omega\omega}\omega), \tag{4.2}$$

where $\hat{v}$ and $\hat{\omega}$ are the noisy motor command, and $\mathcal{N}(\mu, \sigma)$ is Gaussian noise with mean μ and standard deviation σ. The constants used were $k_{vv} = 0.1$, $k_{v\omega} = 0.1$, $k_{\omega v} = 0.2$, $k_{\omega\omega} = 0.1$. This is a simplified motor noise model, inspired by realistic models used in robot localization and mapping (Roy & Thrun, 1999; Beeson, Murarka, & Kuipers, 2006). The robot accepts motor commands from the agent 10 times per simulated second.

This simulated robot was used for all the experiments described in this dissertation, including those in Chapters 5 and 6. The two simulated environments used are described in the next two sections, along with any minor differences in simulation settings used in the different environments.



(a)

(b)

(c)

Figure 4.1: **Simulated Robot Environment, T-Maze**. (a) A "sensor-centric" plot of a single scan from the robot's laser rangefinder in the T-maze environment, with the individual range sensor on the X axis and the range value on the Y axis. (b) The egocentric plot of the same scan in polar coordinates, with the robot at the origin, heading up the Y axis. This format provides a useful visualization of laser rangefinder scans for human consumption. (c) A screen-shot of the Stage robot simulator with the robot in the pose that generated the scan used in (a) and (b). The shaded region represents the area scanned by the laser rangefinder. The robot has a drive-and-turn base, and a laser rangefinder. The environment is approximately 10 meters by 6 meters. The formats in (a) and (b) will be used in later figures to display the learned perceptual prototypes.

(a) Abstract Motor Interface		
	$\mathbf{u}^0$	$\mathbf{u}^1$
drive	250 mm/sec	0 mm/sec
turn	0°/sec	20°/sec

(b) Primitive Actions $\mathcal{A}^0$		
action	$\mathbf{u}$	step
a_0^0	$\mathbf{u}^0$	25mm
a_1^0	$-\mathbf{u}^0$	-25mm
a_2^0	$\mathbf{u}^1$	2°
a_3^0	$-\mathbf{u}^1$	$-2°$

Table 4.1: T-Maze Abstract Motor Interface and Primitive Actions

4.2 T-Maze Environment

The first experiment environment, called the T-Maze, is a 10 m $\times$ 6 m T-shaped room, shown in Figure 4.1. The T-Maze was designed to be large enough to provide an interesting test environment, yet small enough to allow experiments to run quickly. The environment has a single major decision point, the central intersection, the extremities of the space are separated from one another by several hundred primitive actions, allowing reasonably long-diameter navigation tasks.

In this environment the robot was given a basic drive speed of 250 mm/sec, and a basic turn speed of 20°/sec, resulting in the abstract motor interface and primitive action set shown in Table 4.1. For all experiments in both the T-Maze and the ACES environment (below), the discovery of the abstract motor interface and the primitive actions $\mathcal{A}^0$ (step 1 of the algorithm in Chapter 3) was assumed to have already been performed, using the methods of Pierce & Kuipers (1997).

To learn perceptual features in the T-Maze environment, the agent was allowed to wander by selecting randomly from its set of $\mathcal{A}^0$ actions for 500,000 steps (about 14 simulated hours), training its GNG with the input vector $\mathbf{y}$ received on each step. The GNG parameters (Fritzke, 1995) are $\lambda = 2000$, $\alpha = 0.5$, $\beta = 0.0005$, $\epsilon_b = 0.05$, $\epsilon_n = 0.0006$, $a_{max} = 100$. The GNG was configured to grow only if the average cumulative distortion error across all units was greater than 0.5%. These parameters were selected after hand experimentation with the algorithm, and can be understood as follows: λ allows the network to grow every 2000 input presentations, or just over three minutes of simulated time (if the error threshold is met). This allows rapid growth early on in learning (when error is usually high), but still allows the agent to experience a large amount of its local neighborhood before growing again. The error decay β was set to $1/2000$, the reciprocal of λ. This setting allows error to decay to about 40% of its original value during the course of one learning period, focusing learning on error accumulated in the most recent one or two growth periods. The parameters ϵ_b and ϵ_n are the learning rates for the winner and its neighbors, respectively, and are set to low values to counteract the similarity between successive inputs when they are presented from a random walk in the environment. If the learning rates are too high, the winning neighborhood will "track" the inputs, adapting so much on each presentation that the winning unit never changes. The parameter a_{max} sets how long the algorithm takes to delete edges in the connection graph when they connect nodes that have moved away from one another in the input space; $a_{max} = 100$ is a relatively high value, designed to keep good connectivity when the units are distributed in a high-dimensional input space. Finally the error threshold of 0.5% was set as the maximum value that produced reasonably full coverage of environmental features on each run.

Ten runs with these parameters were performed to train ten GNG networks for use in the experiments in Chapter 6, to allow those experiments to test navigation ability across multiple feature sets.

Figure 4.2 shows the features learned in one learning run in the T-Maze. The features are organized into rows according to the GNG topology. Since the GNG weight vectors have the same dimensionality as the sensory input, the learned weight vectors (features) can be thought of as perceptual prototypes. The first plot in each row shows the sensory prototype represented by the weight vector of GNG unit, and the remaining plots in the row show the prototypes represented by the neighboring units in the GNG topology. On the left the prototypes are plotted in the human-readable egocentric visualization of Figure 4.1(b), while on the right the same figures are plotted in the sensor-centric format of Figure 4.1(a).

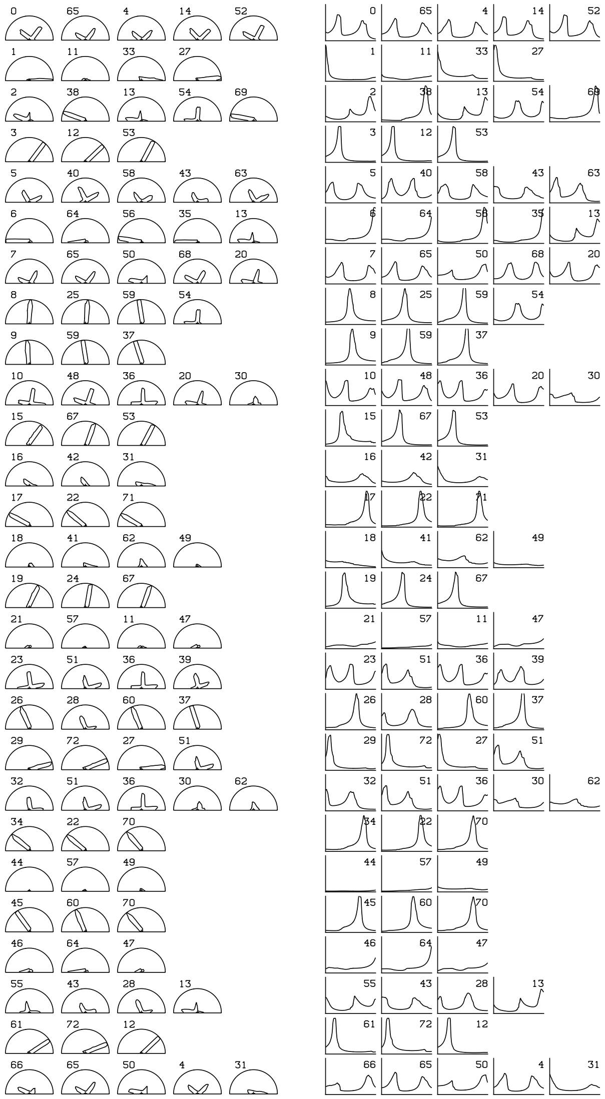


Figure 4.2: **Example Learned Perceptual Prototypes, T-Maze.** The agent's self-organizing feature map learns a set of perceptual prototypes that are used to define perceptually distinctive states in the environment. This figure shows the set of features learned from one feature-learning run in the T-Maze. In each row the first figure represents a unit in the GNG, and the other figures represent the units it is connected to in the GNG topology. Some rows are omitted to save space, but every learned feature appears at least once. Each feature is a prototypical laser rangefinder image. On the left, the ranges are plotted in the human-readable format of Figure 4.1(b). On the right, the ranges are plotted in the sensor-centric format of Figure 4.1(a). The agent learns a rich feature set covering the perceptual situations in the environment

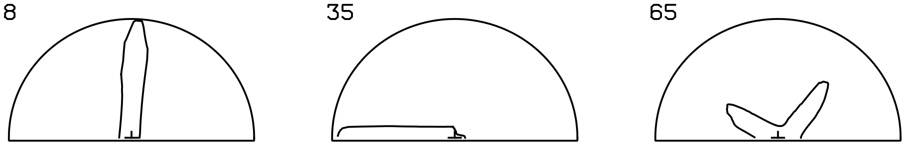


Figure 4.3: **Three Learned Prototypes, Enlarged** Enlarged views of features 8, 35, and 65 from Figure 4.2. These figures show prototypical sensory views of looking down the long corridor in the T-Maze, looking toward the side wall at the end of the corridor, and the view from the middle of the T intersection, respectively.

Three of the units, features 8, 35, and 65, are shown enlarged in Figure 4.3. Feature 8 shows a prototypical view of the agent's sensation of looking straight down the long corridor, feature 35 shows a view that represents facing the wall, at the end of the long corridor, and feature 65 shows a prototypical view from the middle of the T intersection.

4.3 ACES Fourth Floor Environment

In order to test the ability of SODA to scale up to larger environments, the robot was also run in a simulation of the fourth floor corridors of the ACES building on the UT Austin campus. This environment, shown in Figure 4.4 was constructed from an occupancy-grid map of the floor, collected using a physical RWI Magellan Pro robot similar to the simulated robot described in Section 4.1. To construct the simulated environment from the original occupancy-grid map, the walls were thickened slightly and some imperfections removed to ensure that the walls would be entirely opaque to the simulated laser scanner. In addition to being much larger than the T-Maze at approximately 40 m $\times$ 35 m, the ACES environment is perceptually richer, with a rounded atrium, T- and L-intersections, a dead end, and an alcove.

Because of the large size of the ACES environment, a very long random walk over primitive $\mathcal{A}^0$ actions would be required for the agent to experience enough of the environ-

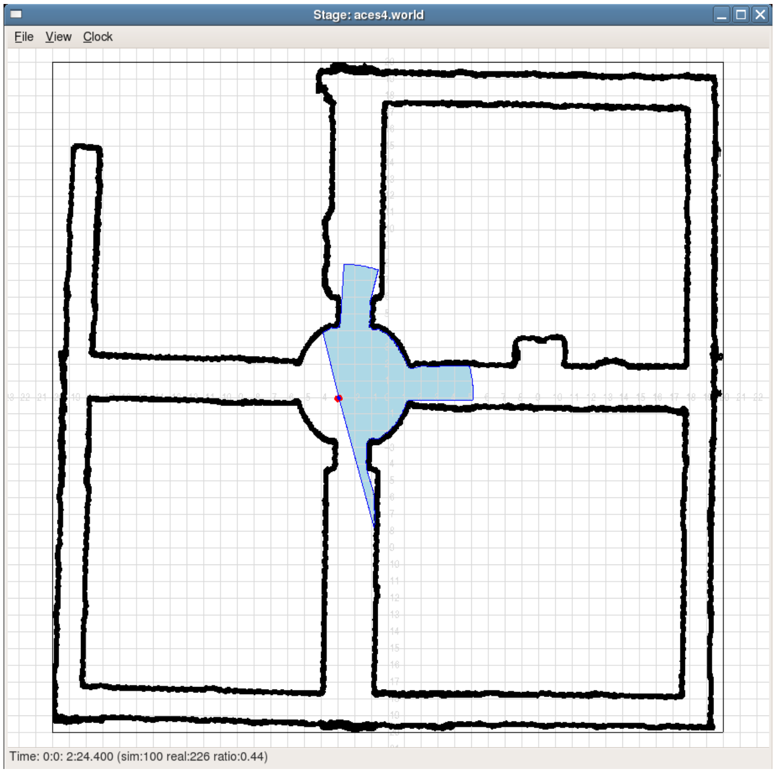


Figure 4.4: **Simulated Robot Environment, ACES**. The ACES4 environment. A simulated environment generated from an occupancy grid map of the fourth floor of the ACES building at UT Austin. The map was collected using a physical robot similar to the simulated robot used in these experiments. The environment is approximately 40m × 35m. The small circle represents the robot. The area swept by the laser rangefinder is shaded. This environment is much larger and perceptually richer than the T-maze.

ment to learn a useful feature set. Instead, SODA was configured to explore the environment using a random walk over trajectory-following options (Section 3.3). In this case the GNG was trained concurrently with the training of the TF option policies. However, for experimental clarity, the learned TF policies were discarded after the feature sets were learned, then the TF options were trained anew during the experiments in Chapter 6. The GNG was trained on each primitive step while the TF macros were running. In addition, to reduce the running time of the experiments in this larger environment, the robot's forward speed was doubled to 500 mm/sec, giving the abstract motor interface described in Table 4.2.

(a) Abstract Motor Interface		
	$\mathbf{u}^0$	$\mathbf{u}^1$
drive	500 mm/sec	0 mm/sec
turn	0°/sec	20°/sec

(b) Primitive Actions $\mathcal{A}^0$		
action	$\mathbf{u}$	step
a^0_0	$\mathbf{u}^0$	50mm
a^0_1	$-\mathbf{u}^0$	-50mm
a^0_2	$\mathbf{u}^1$	2°
a^0_3	$-\mathbf{u}^1$	$-2°$

Table 4.2: ACES Abstract Motor Interface and Primitive Actions

As with the T-Maze environment ten runs were performed to train ten GNGs for later use in the experiments in Chapter 6. In each run, the agent explored the environment for 5,000,000 time steps. An example learned GNG from the ACES environment is shown in Figure 4.5.

4.4 Feature Discussion

As shown in Figures 4.2 and 4.5, the features learned by SODA in these environments cover a broad range of the environment, showing prototypical views of corridors and intersections at a wide variety of relative angles, as well as, in ACES, a variety of views of the central atrium. In fact, a large part of every feature set is dedicated to representing similar views that differ mainly by small changes in relative angle (Figure 4.6). This preponderance of similar features results from the fact that a small rotation of the robot in its configuration space induces a “shift” of the values in the input vector generated by the laser rangefinder. When some elements of the input vector are very large and some are very small, as is often the case in these environments, such a shift moves the input a large Euclidean distance in the input space, essentially moving the vector from one “corner” of the space to another. Given that the stated purpose of SODA is to decrease the diameter of the task in the environment, one might argue that a more coarse-grained feature representation would be preferable. Indeed Chapter 6 will show that SODA agents tend to use many more $\mathcal{A}^1$ actions to turn than they use to travel forward (and backward). Section 7.2.2 describes possible future research

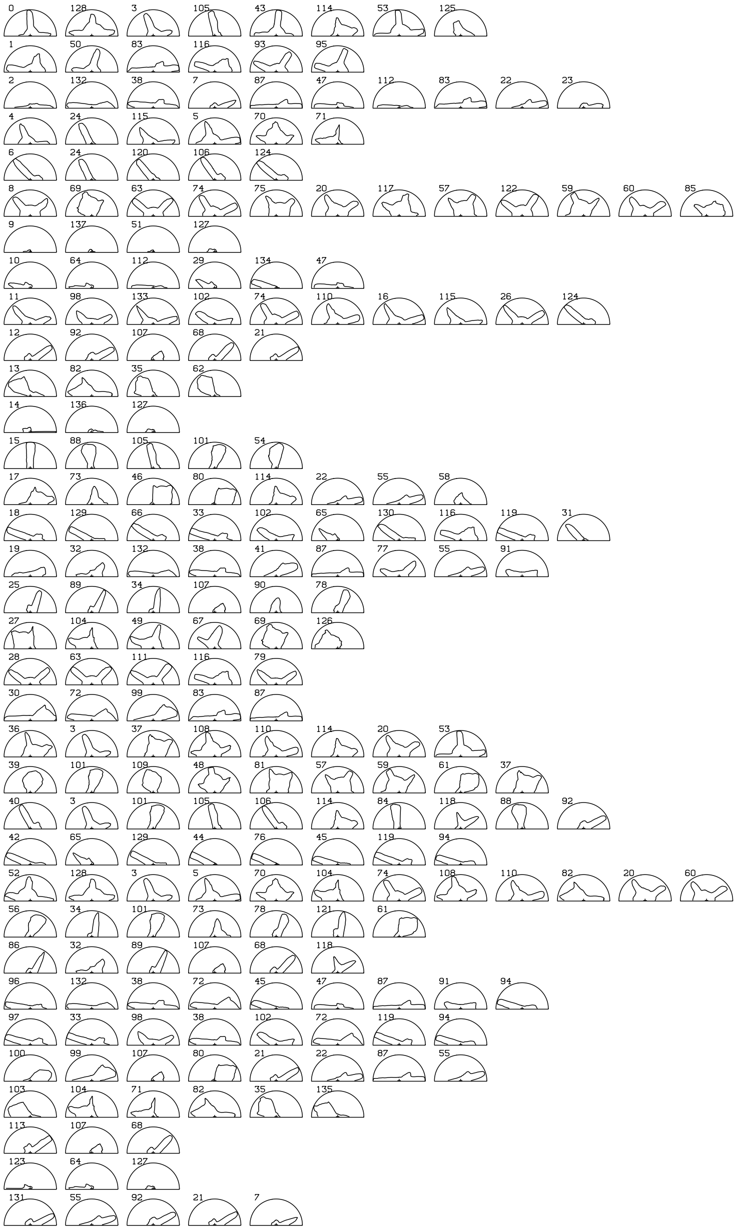


Figure 4.5: **Example Learned Perceptual Prototypes, ACES.** The agent's self-organizing feature map learns a set of perceptual prototypes that are used to define perceptually distinctive states in the environment. The richer variety of perceptual situations in the ACES environment produces a larger set of features using the same parameter settings for the Homeostatic-GNG.

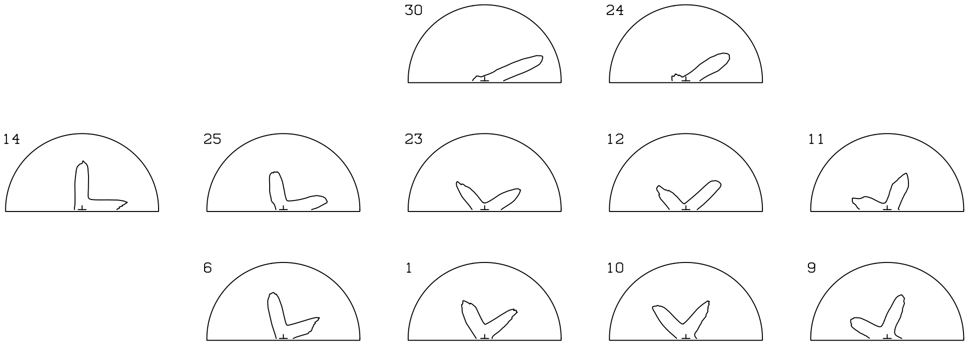


Figure 4.6: **Views of T-Maze Intersection** The use of Euclidean distance as the similarity metric for the GNG leads to the learning of many similar views that differ only by a small rotation. A small rotation in the robot's configuration space induces a large (Euclidean) movement in the robot's input space causing large distortion error in the GNG, which must add more features in that region of the space to reduce the error. The large number of features means that the agent must traverse many more distinctive states when turning than when traveling forward and backward. Nevertheless, the experiments in Chapter 6 show that SODA still greatly reduces task diameter and improves navigation learning.

directions for finding ways to cut down or prune this proliferation of features from turning. Chapter 6 will show, however, that despite these extra features, SODA still does a very good job of cutting down the task diameter in the T-Maze and ACES environments. The reinforcement learning algorithm learns from experience which features are useful in getting to the goal quickly, essentially doing a form of rudimentary feature selection. Furthermore, Chapter 5 describes how SODA agents learn trajectory-following and hill-climbing controllers that operate within the neighborhood of a single feature. These controllers use the proximity of the input to nearby features (other than the winner) to do this kind of "intra-neighborhood" navigation. The success of SODA at doing both high-level and low-level navigation suggests that detailed feature sets such as those shown in this chapter are not only sufficient for SODA's purposes, they might be necessary as well.

Finally, the large number of algorithmic parameters for the GNG network deserve some mention with reference to SODA's stated goal of reducing the need for human prior knowledge in agent's learning process (Section 1.3). As with most learning algorithms, the GNG has a set of parameter "knobs" that control learning that must be set properly for the agent to learn. To some extent, these settings come from human experimentation with the algorithm in the target domain. Reducing the need for such experimental knob twisting is a major direction for future research, discussed in more detail in Section 7.4. With respect to the use of the GNG for feature learning, however, these results do provide some reason for optimism. First, although human experimentation was required, a painstaking, exhaustive search was not needed to discover suitable learning parameters. Rather, the parameters used in this chapter were discovered by means of a relatively short "educated walk" through the parameter space. Second, the GNG's learning performance seems to be robust against small changes in learning parameters. Once a suitable region of the parameter space has been found, small tweaks in parameter values do not seem to induce major qualitative changes in the algorithm's behavior. Lastly, it was not necessary to find a new set of GNG parameters when the agent was moved from the T-Maze to ACES, despite ACES' greater perceptual richness – although this transfer says nothing about what would be necessary for a robot with a different sensorimotor configuration or a vastly different environment. These observations suggest that it might be possible to find a useful set of default parameter values, and a simple set of rules for searching for the correct parameter set, either automatically or by hand.

In conclusion, this chapter has presented the two experimental environments used throughout this dissertation, and demonstrated SODA's ability to learn a set of perceptual features through unsupervised interaction in each environment. The next two chapters will show that the learned features are useful both for learning local control in SODA's trajectory-following and hill-climbing actions, and for reducing the task-diameter of large-scale navigation tasks, thus improving the agent's ability to learn to navigate from place to place.

Chapter 5

Learning High-Level Actions

The previous chapter described how the agent learns a set of perceptual features that it can use for navigation. Once it has done so, its next task is to learn a set of trajectory-following and hill-climbing actions that can be combined to form high-level $\mathcal{A}^1$ actions. The formal structure of these actions is described in Chapter 3. This chapter examines how such actions are learned and shows that learning the actions as nested reinforcement learning problems (options) produces more reliable and efficient actions than alternative hard-coded means of constructing them, without requiring prior knowledge of the agent's dynamics.

To separate the problem of learning the high-level actions from that of learning large-scale navigation behavior (covered in Chapter 6), this chapter presents experiments testing SODA's trajectory-following (TF) and hill-climbing (HC) actions in isolation from any larger navigation problem. The trajectory-following experiment in Section 5.1 compares the two TF methods described in Section 3.3: learned TF options and ballistic, open-loop TF macros. This experiment shows that the learned options are more reliable and provide longer trajectories than the open-loop macros. The experiment in Section 5.2 compares the three HC methods discussed in Section 3.4: hill-climbing by manually sampling to approximate the feature gradient, hill-climbing by approximating the gradient using a hand-coded action model, and learning hill-climbing options using reinforcement learning.

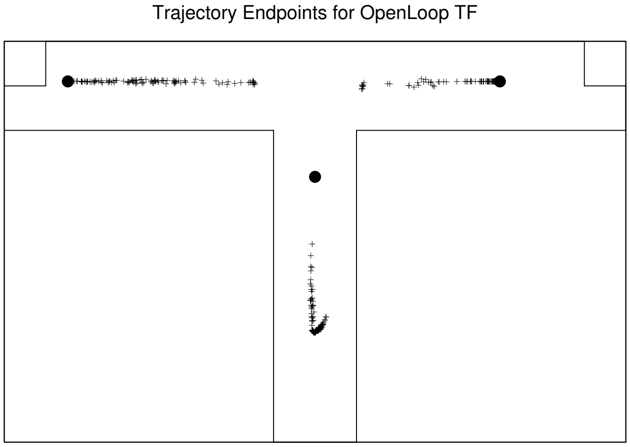
The experiment shows that the learned HC options are significantly more efficient than sampling, and perform comparably to the hand-coded action model, while not requiring prior knowledge of the dynamics of the robot and its environment.

5.1 Trajectory Following

The first step of a high-level action is trajectory following. In this task the SODA agent moves the robot out of its current perceptual neighborhood and into another. It does so by making progress along some axis of its abstract motor interface, while simultaneously trying to maintain the activation of the current SOM winner at as high a level as possible. One example of trajectory following is following a corridor to its end. Assuming the robot is facing nearly straight down a corridor, its current winning SOM feature should represent a view down a corridor, similar to Feature 8 in Figure 4.3. To trajectory-follow down the corridor the agent would move the robot forward while making adjustments left and right to keep the activation of the "facing down the hall" feature as high as possible for as long as possible, stopping when some other perceptual feature became the winner.

The purpose of TF actions is to give SODA's $\mathcal{A}^1$ actions spatial extent, reducing the effective diameter of the high-level navigation task. It is desirable for TF actions to follow a given trajectory for as long as possible, thus encapsulating many primitive actions in a single abstract action, and requiring fewer abstract actions to reach distant goals. Given this purpose, a natural question to ask is whether the simpler open-loop TF macros (described in Section 3.3) would be just as effective in decreasing task diameter. Open-loop TF macros merely repeat the progress (e.g. forward) action until the perceptual neighborhood changes, without making any corrective actions to keep the current feature maximized. The experiment below shows that the learned, closed-loop TF options not only increase the average trajectory length compared with open-loop macros, but also more reliably terminate near the same state in the environment.

The trajectory-following experiment in this section compared the reliability of open-


Trajectory Endpoints for OpenLoop TF

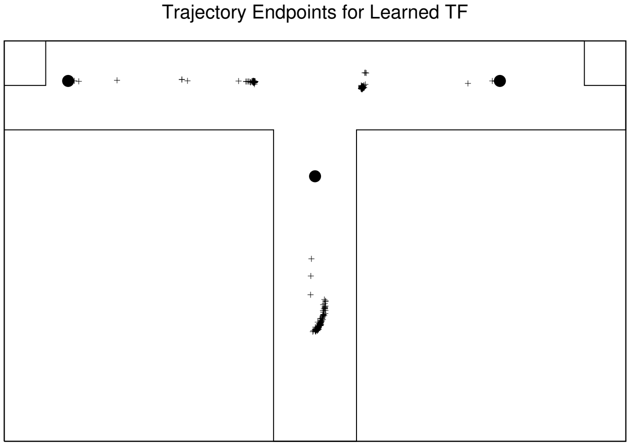
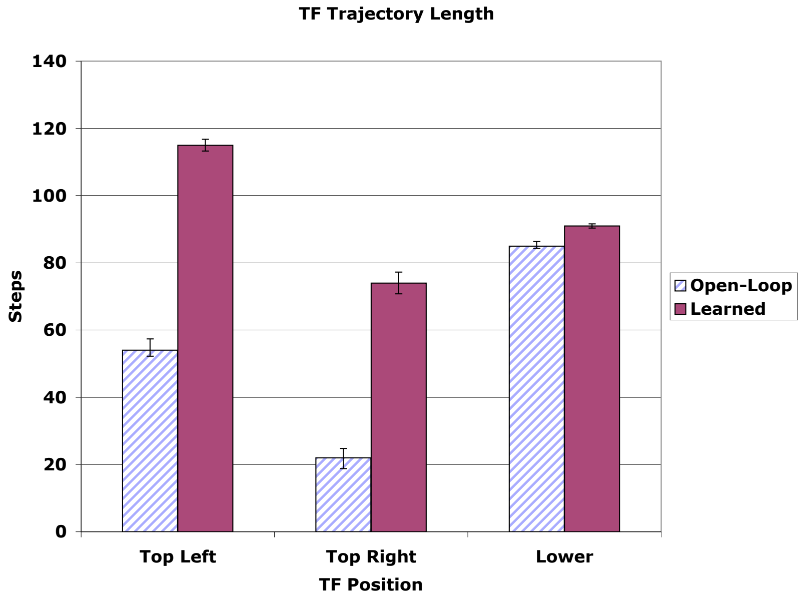

Trajectory Endpoints for Learned TF

Figure 5.1: **Trajectory Following Improvement with Learning** These figures show the results of 100 trajectory following runs from each of three locations (marked with large black disks). The end point of each run is marked with a '+'. The top figure shows the results of open-loop TF, and the bottom figure shows the results of TF learned using reinforcement learning. The TF learned using RL are much better clustered, indicating much more reliable travel.

loop TF macros and learned TF options. The experiment tested the agent's ability to learn to follow a trajectory forward down each of the three corridors of the environment. The option's action set consisted of the progress action (moving straight forward), $[1, 0]^T$, and two corrective actions (moving forward while turning left or right) $[1, 0.1]^T$ and $[1, -0.1]^T$. The option was trained for 2000 episodes from each starting point, although in each case the behavior converged within 100-400 episodes. The Sarsa(λ) parameters were: $\lambda = 0.9, \alpha = 0.1, \gamma = 1.0$, the option used ϵ-greedy action selection with $\epsilon_0 = 1.0$ and annealing down to $\epsilon_\infty = 0.001$ with a half-life of 400 steps. All Q values were initialized to 0, and the agent used the $\mathrm{Top}^3(\mathbf{y})$ state representation (Section 3.2). Runs using the Top^4 and Top^5 were also performed. Adding more winners to the representation caused the behavior to converge more slowly, but made no significant difference in the converged behavior. Figure 5.1 shows the ending points of the last 100 runs from each starting point, compared with the ending points for 100 runs using the open-loop TF macro. The bottom panel of Figure 5.2 shows the average inter-point distance between endpoints for those 100 runs in each condition. For each of the three starting points, the endpoints are more tightly clustered when using the learned option. This is because there are many fewer episodes where the trajectory terminates part of the way down the hall due to motor noise pushing the robot off of its trajectory and into a new perceptual neighborhood. As a result, the learned option produces longer trajectories more reliably, and the endpoints tend to be near one another. Figure 5.2 shows the average lengths (in steps) of the last 100 runs of the learned TF option, compared with 100 runs from the open-loop macro. Table 5.1 shows the precise values of the averages and standard deviations. In all three cases the average trajectory length from the learned options is significantly longer ($p < 6 \times 10^{-5}$), and from two of the three starting points it is dramatically longer.

These experiments show that the learned TF options are more reliable and produce longer trajectories than the naïve alternative of open-loop macros. However, even with this improvement, there is still a fairly large variation in the results, especially for the case in



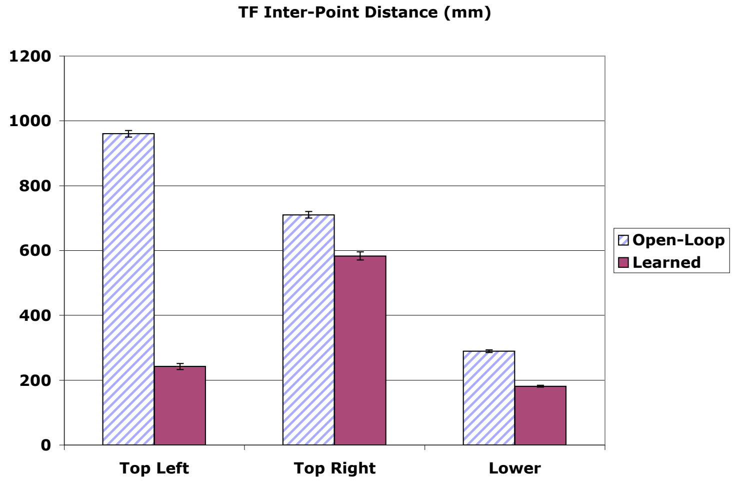


Figure 5.2: **The average trajectory length and endpoint spread for open loop vs. learned trajectory-following for the last 100 episodes in each condition.** Learned TF options produce longer trajectories. Error bars indicate ± one standard error. All differences are significant at $p < 6 \times 10^{-5}$

Starting point	Open-loop mean (σ)	Learned mean (σ)
Top left	56 (34.3)	114 (20.1)
Top right	26 (32.3)	77 (29.4)
Lower	86 (12.8)	90 (9.6)

Table 5.1: **The average trajectory length for open loop vs. learned trajectory-following.** Learned TF options produce longer trajectories.

the upper-right corridor of the environment. Although the average inter-endpoint distance shown in Figure 5.2 for the learned TF option in the upper right corner is very large, the distribution of endpoints in Figure 5.1 shows that nearly all of the endpoints are clustered into three clusters, one of which is far from the other two and very near the starting point, resulting in very short trajectories. Thus, despite the large standard deviation in trajectory length, the outcome of the learned action is still considerably more reliable relative to the open-loop case, in which the endpoints are distributed widely along the mid-line of the upper right hallway. In addition, the cluster of endpoints so near the starting point in the upper-right hallway illustrates just how narrow the trajectories are that the agent must learn to follow. The TF option is defined to terminate as soon as the SOM winner changes (Equation (3.5)). Because the learned feature set contains many similar representations of corridors at different relative angles (as described in Section 4.4), only a small deviation in heading from the mid-line of the corridor is required for the TF option to terminate. It is likely that some of this variation could be eliminated by allowing the TF option to stray temporarily from its designated perceptual neighborhood, as long as it returns within a short window of time. Such a non-Markov termination function would be similar in some respects to the termination function for the hill-climbing options in Equation (3.12) that terminate if no progress is made over a short window of time. Such a change would likely improve the average trajectory length somewhat as well as reducing both the variance in trajectory length and the uncertainty in the outcome state. Such a change would come at the cost of adding another free parameter to the algorithm (the size of the time window for termination). It is unclear, though, how much impact such a change would have on the overall navigation performance, described in Chapter 6.

5.2 Hill Climbing

Once the SODA agent has completed trajectory following, the second part of an $\mathcal{A}^1$ action is hill-climbing, in which the agent attempts to reduce its positional uncertainty by moving the robot to a local maximum of the activation of the current winning SOM unit. Section 3.4 described three HC methods. The first, hill-climbing by sampling, was shown to be somewhat effective in preliminary experiments (Provost, Kuipers, & Miikkulainen, 2006). It is inefficient, however, using $2|\mathcal{A}^0| - 2$ sampling steps for each step "up the hill." The second method eliminates these sampling steps by using a hand-engineered model to predict the perceptual outcome of actions, at the cost of considerable prior knowledge of the robot and environment required of a human engineer. The third method eliminates the need for manual sampling and human prior knowledge of the robot's dynamics by using reinforcement learning methods to learn a hill-climbing policy for each perceptual feature in the agent's feature set. The experiment in this section compares these three methods, showing that all three are able to achieve similar feature activations, but the action models and the learned options are considerably more efficient at doing so, with the learned options in particular performing comparably with the other two methods while requiring neither sampling nor prior knowledge. Below is a detailed description of the hand-coded action model used in the experiment, followed by the experiment description, results, and discussion.

5.2.1 Hand-coded Predictive Action Model

With detailed knowledge of the robot's sensorimotor configuration, actions, and environment dynamics, it may be possible for a human engineer to create a predictive action model that will allow the SODA agent to hill-climb by predicting the feature changes, as described in Section 3.4.1. Such a model can be constructed for the robot and T-Maze and environments described in Section 4.1. This model consists of four affine functions, one for each primitive action in $\mathcal{A}^0$, that give the difference between the current sensor input $\mathbf{y}_t$ and the estimated input on the next time-step $\hat{\mathbf{y}}_{t+1}$:

$$\hat{\mathbf{y}}_{t+1} - \mathbf{y}_t = \mathbf{A}_i \mathbf{y}_t + \mathbf{b}_i. \tag{5.1}$$

The four primitive actions for the robot in the T-Maze correspond to steps forward, backward, left, and right (Table 4.1). The linear components of the four predictive functions will be referred to as $\mathbf{A}_{\text{forward}}$, $\mathbf{A}_{\text{backward}}$, $\mathbf{A}_{\text{left}}$, and $\mathbf{A}_{\text{right}}$, and the translation components will be called $\mathbf{b}_{\text{forward}}$, $\mathbf{b}_{\text{backward}}$, $\mathbf{b}_{\text{left}}$, and $\mathbf{b}_{\text{right}}$.

The functions for the turning actions are relatively simple. Since they turn the robot approximately 2°, and the laser range-finder samples radially at 1° increments, the turn action function should return the difference between the current input and the same input shifted by two places, i.e this 180×180 matrix:

$$\mathbf{A}_{\text{left}} = \begin{bmatrix} -1 & 0 & 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \end{bmatrix}. \tag{5.2}$$

Because it is not possible to know what will be "shifted in" on the left side of the range-finder, the function predicts no change in the last two positions. (The laser range-finder scans are numbered counterclockwise, i.e. right-to-left, around the robot.) The right turn matrix $\mathbf{A}_{\text{right}}$ is constructed analogously. The translation component of the turn models is zero:

$$\mathbf{b}_{\text{left}} = \mathbf{b}_{\text{right}} = \mathbf{0}. \tag{5.3}$$

The functions for moving forward and backward are more complicated, since translating the robot produces a complicated change in the radially organized range-finder image. The intuition behind them comes from approximating how the input behaves at three key scans – far left, straight ahead, and far right – and interpolating the effects in between. When the robot executes a *forward* action, the forward facing scan (scan 90) is reduced by a constant amount, while the leftmost scan (scan 179) behaves roughly as if the robot has rotated to the right, and the rightmost scan (scan 0) behaves as if the robot has rotated to the left. The scans in between can be approximated with a combination of a constant change and a shift, with the proportion of each depending on the scan's relative angle — scans facing more forward get more of a constant change, scans facing more sideways get more of a shift.

The functions for moving forward and backward can be constructed using three blending matrices, to interpolate between the right, left, and center parts of the function. The $\sin^2$ and $\cos^2$ functions provide an ideal means for constructing these matrices since $\sin^2 x + \cos^2 x = 1$ and both functions have periods of 180°, the same as the extent of the arc of the laser range-finder. The three blending matrices are

$$\mathbf{C}_1 = Diag_{180}\left[\cos^2(0^\circ), \cos^2(1^\circ), ..., \cos^2(89^\circ), 0, ..., 0\right], \tag{5.4}$$

$$\mathbf{C}_2 = Diag_{180}\left[0, ..., 0, \cos^2(90^\circ), \cos^2(91^\circ), ..., \cos^2(179^\circ)\right], \tag{5.5}$$

and,

$$\mathbf{S} = \left[\sin^2(0^\circ), \sin^2(1^\circ), ..., \sin^2(179^\circ)\right]^T. \tag{5.6}$$

The matrix $\mathbf{C}_1$ is used for weighting the rotation component on the right side of the robot: it gives a weight of 1.0 to the rightmost part of the laser range-finder, falls to zero at the center, and remains zero weight left half of the scan. Likewise, $\mathbf{C}_2$ weights the rotation on the left side: it gives a weight of 1.0 to the leftmost part of the laser range-finder and falls to zero at the center. $\mathbf{S}$ weights the translation component: it gives a weight of 1.0 to the center and falls to zero at both sides.

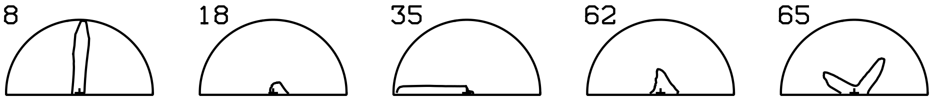


Figure 5.3: **Test Hill-Climbing Features.** The hill-climbing experiments tested SODA's ability to hill-climb on these five features from the GNG in Figure 4.2in the T-Maze environment.

The forward and backward motion models are created using the above blending functions, . The forward model,

$$\mathbf{A}_{\text{forward}} = \mathbf{C}_1\mathbf{A}_{\text{right}} + \mathbf{C}_2\mathbf{A}_{\text{left}}, \mathbf{b}_{\text{forward}} = -25\mathbf{S}, \tag{5.7}$$

combines a left turn on the right, a right turn on the left, and a 25 mm reduction of the centermost range value. The backward motion model,

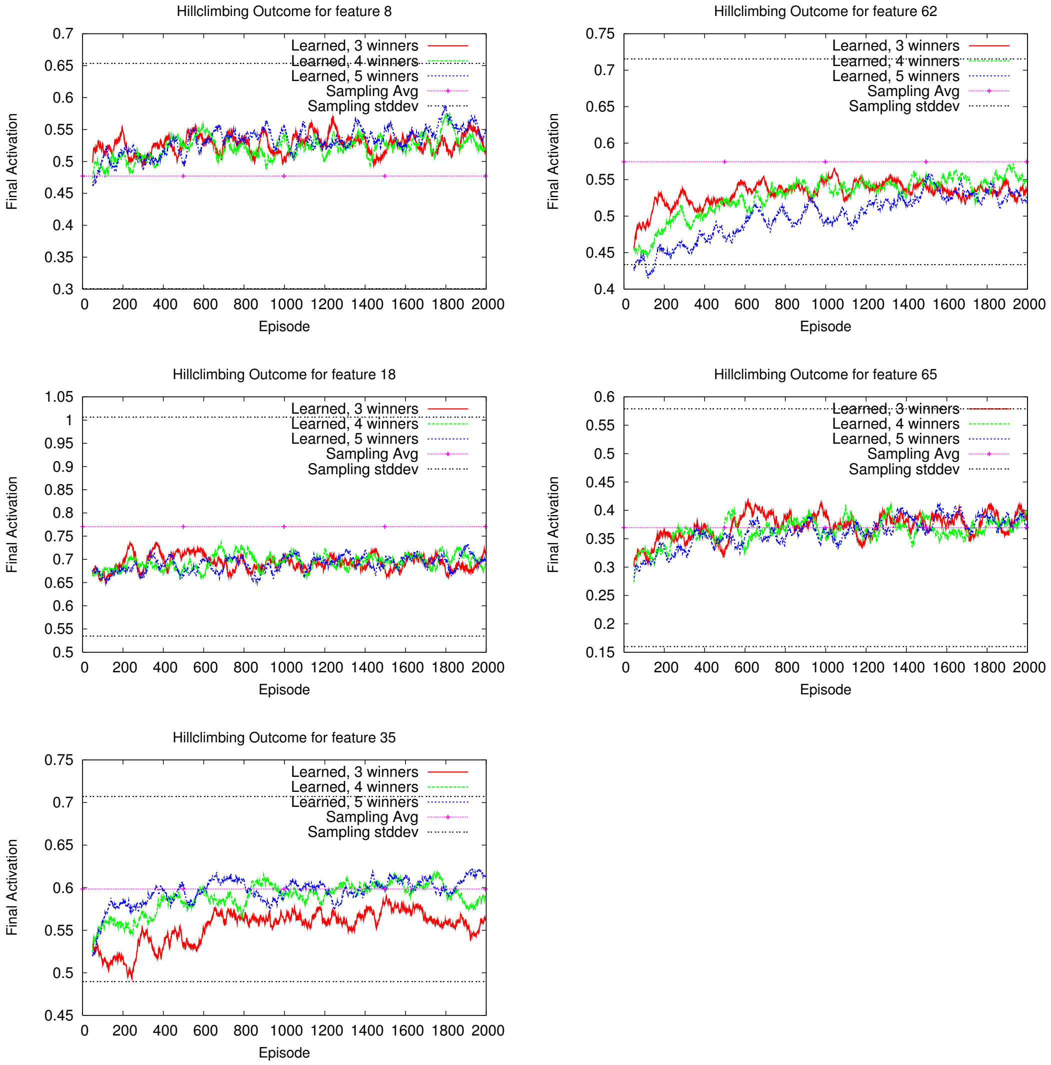
$$\mathbf{A}_{\text{backward}} = \mathbf{C}_1\mathbf{A}_{\text{left}} + \mathbf{C}_2\mathbf{A}_{\text{right}}, \mathbf{b}_{\text{backward}} = 25\mathbf{S}, \tag{5.8}$$

combines a left turn on the left, a right turn on the right and a 25 mm increase of the centermost range value.

As shown in the next section, a hill-climbing controller using these definitions to predict the feature changes does as well as the method that samples manually, while using many fewer actions. This model, however, requires considerable prior knowledge of the robot, including the size and direction of its primitive actions, and the type and physical configuration of the sensors. The next section will show that learned HC options can achieve similar performance without requiring that prior knowledge.

5.2.2 Hill-Climbing Experiment

The hill-climbing experiment compared the speed and effectiveness of the learned HC options against HC by gradient approximation using sampling, and using the models described



Hillclimbing Outcome for feature 8

Hillclimbing Outcome for feature 62

Hillclimbing Outcome for feature 18

Hillclimbing Outcome for feature 65

Hillclimbing Outcome for feature 35

Figure 5.4: **Hill-climbing Learning Curves.** These curves compare learned hill-climbing using the Top^3, Top^4, and Top^5 state representations against hill-climbing by approximating the gradient with user-defined linear action models. Each plot compares hill-climbers on one of the five different SOM features in Figure 5.3. The Y axis indicates the final feature activation achieved in each episode. The thick straight line indicates the mean performance of sampling-based HC, and the two thin straight lines indicate its standard deviation. This figure shows that Learned HC does about as well as an HC controller that manually approximates the feature gradient at each point, and sometimes better.

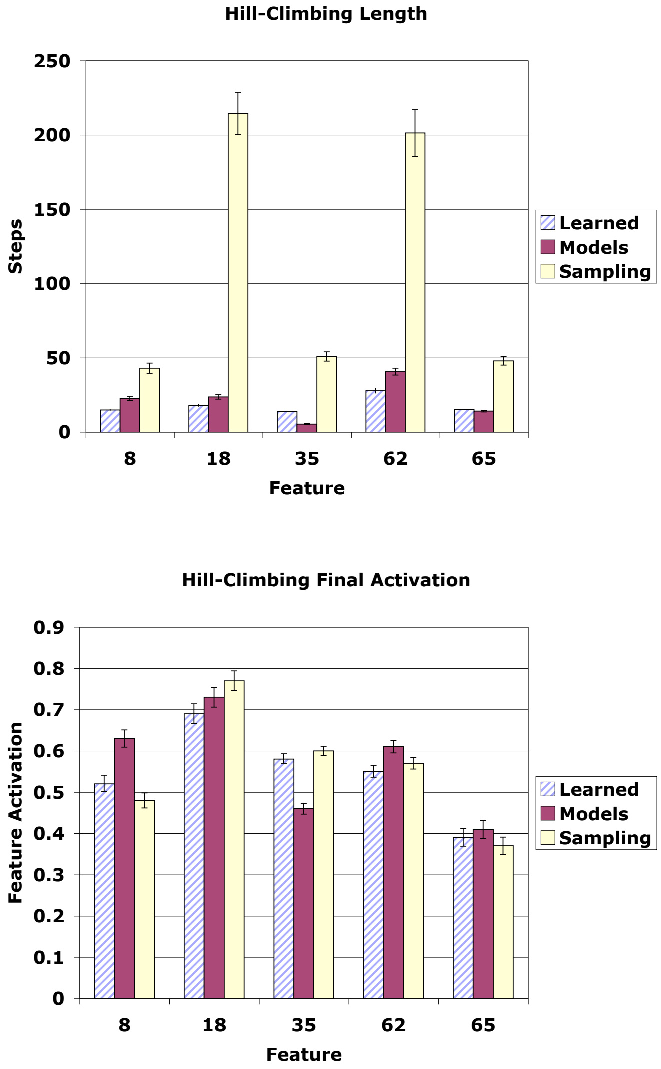


Figure 5.5: **Hill-climbing performance with and without learned options.** Using learned options makes hill-climbing achieve the same feature values faster. *Top:* The average lengths of hill-climbing episodes in the neighborhoods of the three different features shown in Figure 5.3. All differences are significant ($p < 2 \times 10^{-6}$). The bottom chart shows the average maximum feature value achieved for each prototype per episode. The plots compare the last 100 HC episodes for each feature with 100 hard-coded HC runs. Differences are significant between learned options and the other two methods ($p < 0.03$) for all features except 65, which has no significant differences. Across all features, the maximum values achieved are comparable, but the numbers of actions needed to achieve them are much smaller.

above. The experiment tested the agent's ability to hill-climb in the neighborhood of each of five specific features, shown in Figure 5.3, taken from the GNG in Figure 4.2. The features were chosen to sample a wide variety of different perceptual situations for the robot in the T-Maze, including views of corridors, walls, intersections, and dead ends. For each feature, the robot was repeatedly placed at 2000 randomly selected poses in the perceptual neighborhood of the given feature, and the hill-climbing option (or macro) was initiated from that point. The option's Sarsa(λ) parameters were $\lambda = 0.9$, $\alpha = 0.1$, $\gamma = 0.997$. The agent did not use ϵ-greedy action selection, but rather all Q-values were initialized optimistically to 1.0 to encourage exploration. HC options do not use ϵ-greedy exploration while TF options do, because hill-climbing is essentially a type of "shortest path" problem. It has a fixed upper bound on the reward achievable from any state, making it suitable for exploration by optimistic initialization. Trajectory-following, on the other hand, is a kind of "longest path" problem in which the agent must try to continue acting (within constraints) for as long as possible, with no obvious maximum value for an action, thus requiring another form of exploration policy. The ϵ-greedy method is standard in the reinforcement learning literature. The HC option parameters (Section 3.4) were $c_{stop} = 0.005$, $c_w = 10$, $c_{R1} = 10$, $c_{R2} = 0.001$. Runs were performed using the Top^3, Top^4, and Top^5, state representations (Section 3.2).

Figure 5.4 presents learning curves comparing the final feature activations achieved by the learned HC options with those achieved by the method of manually sampling to approximate the feature gradient. It shows that for all the features, the learned HC can achieve feature activations near (or better than) the manual sampling method within a few hundred episodes of training.

Figure 5.5 shows bar plots of the average episode lengths and final activations of the last 100 episodes for the learned HC options (using the Top^4 state representation), and the other two methods. It shows that all three methods achieve comparable final activations, while the learned options and the hand-coded models use many fewer steps to climb the hill, because the sampling method performs many extra steps for each step "up the hill." From this result one can conclude that the learned options make hill-climbing as efficient as a controller using hand-coded motion model, without the need for the detailed prior knowledge of the robot's configuration and dynamics that is embedded in the model.

5.2.3 HC Learning Discussion

The HC experiment shows that, after training, the learned HC options generally hill-climb as well as methods that climb the gradient directly, but the learned options do not require expensive sampling actions, or extensive prior knowledge of the robot's sensorimotor system in order to approximate the direction of the gradient. However, there are some aspects of the results that deserve further discussion.

First, the results have large variances in the final activation for all five features and all three HC methods. This variance is largely a result of the experimental methodology: The trials for a particular feature are started at random locations in the environment in the perceptual neighborhood of that feature. The features may apply in multiple, distinct locations in the environment, and the features are approximations of the actual perceptual image at the particular location. Thus, the local maximum near one starting point may be substantially different from the local maximum near another starting point for the same feature. The artificial variation in starting locations in this experiment is likely to be much greater than the actual variation in starting locations when running SODA in practice, since SODA does not begin hill-climbing at uniformly distributed random locations in the environment, but at the endpoints of trajectory-following control laws. The TF experiments in Section 5.1 show that once the TF control laws have been learned, the endpoints of trajectories (and the starting points for hill-climbing) are fairly tightly clustered.

A second point to note is that in Figure 5.4 the learned HC options for two features, 18 and 62, do not achieve quite as high a final activation as the sampling method. Interestingly, these two features both represent views that might be seen at the end of a corridor (Figure 5.3). Figure 5.5 also shows that an HC episode for these two features using sampling is much longer than for any other feature or method. In fact, for these two features the sampling HC episodes are much longer than could be accounted for by the extra $2|\mathcal{A}^0| - 2 = 6$ sampling actions needed for each "up-hill" step. The extra-long HC episodes and slightly higher activations for sampling seem result from the combination of the nature of these two particular features and the particular stopping conditions associated with the different HC methods. Features 18 and 62 are distinguished from the other features by the fact that all of their individual range values fall relatively near one another. (That fact that is more visible in the right hand column of Figure 4.2: compare variance in range values the row containing features 18 and 62 to the rows above and below it.) Because all the ranges in these features are relatively near one another, turn actions in these perceptual neighborhoods cause much less of a change in feature activation than it would for say, Feature 8. In addition, the sampling HC method only terminates when *all* sampled actions indicate a negative feature change, while the learned HC options and the model-based HC both terminate if the average change in feature activation over a short time window is near zero. As a result, when the robot gets into an area where the activation change for all actions is small enough to be lost in the motor noise, the learned HC options will stop relatively quickly, while the sampling HC method will "bounce around," continuing to collect noisy samples until the samples for all actions show a negative feature change. Until it terminates, the sampling method chooses to execute the action that showed the highest positive feature change, however slight, thus moving the robot stochastically up even a tiny feature gradient, should one exist. As a result, for features like 18 and 62, the sampling method is able to achieve slightly higher activations, at the cost of many more actions. This is not a reasonable trade-off given the need to navigate efficiently from one location to another.

Although the learned HC options do not achieve the highest activations for any of the test features, they are also not dominated by either of the other methods. The learned options achieve comparable final activations to the other two methods very efficiently, without needing a hand-engineered action model.

5.3 Action Learning Conclusion

In conclusion, this chapter described experiments examining trajectory-following and hill-climbing actions in isolation from any high-level navigation task. The TF experiments showed that learning trajectory-following actions as options using reinforcement learning produces longer, more reliable trajectories than the obvious naïve alternative of simple repeat-action macros. The HC experiments showed that learning HC actions as options produces actions that perform as well as either HC by sampling or HC using predictive models; yet learned HC options use many fewer primitive actions than the sampling method, and do not require the extensive prior knowledge needed for the predictive models. The next chapter shows how the learned TF and HC actions, combined together, improve learning in large-scale navigation tasks, over performing the same tasks using primitive actions.

Chapter 6

Learning High-Diameter Navigation

Once a SODA agent has learned a set of perceptual features and a set of high-level trajectory-following and hill-climbing actions based on those features (as described in Chapters 4 and 5), it can begin to use these new features and actions to navigate between distant locations in the environment. Navigating in the abstracted space defined by the learned $\mathcal{A}^1$ actions, the agent reduces its task diameter dramatically. This chapter describes several experiments in which agents learn navigation tasks using reinforcement learning over SODA actions in the environments described in Chapter 4. The first set of experiments, run in the T-Maze, show that agents can learn to navigate much more quickly using SODA actions than using primitive actions, using as few as 10 $\mathcal{A}^1$ actions to complete tasks requiring hundreds of $\mathcal{A}^0$. In addition, these experiments investigate the benefit provided by the hill-climbing step in the SODA actions, showing that hill-climbing produces more reliable actions, and produces solutions requiring fewer abstract actions. The second set of experiments, run in the ACES environment, show that SODA can scale up from the T-Maze to a realistic, building-sized environment, and that the benefit provided by the SODA actions is even greater in the larger environment.

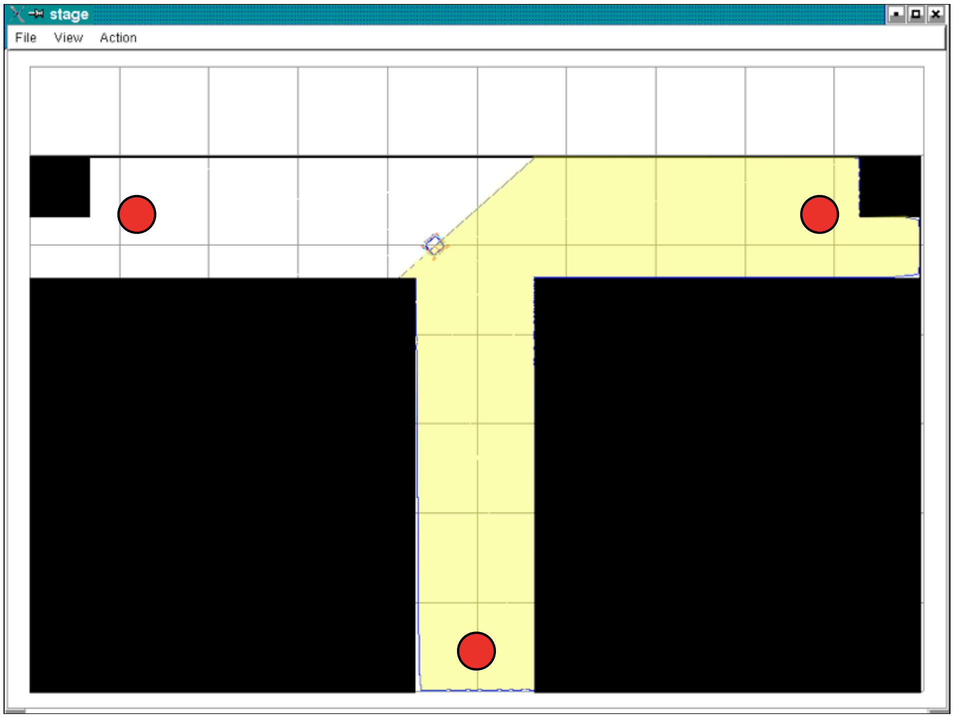


Figure 6.1: **T-Maze Navigation Targets.** The red circles indicate the locations used as starting and ending points for navigation in the T-Maze in this chapter. In the text, these points are referred to as *top left*, *top right*, and *bottom*. Navigation between these targets is a high-diameter task: they are all separated from each other by hundreds of primitive actions, and from each target the other two are outside the sensory horizon of the robot.

6.1 Learning in the T-Maze

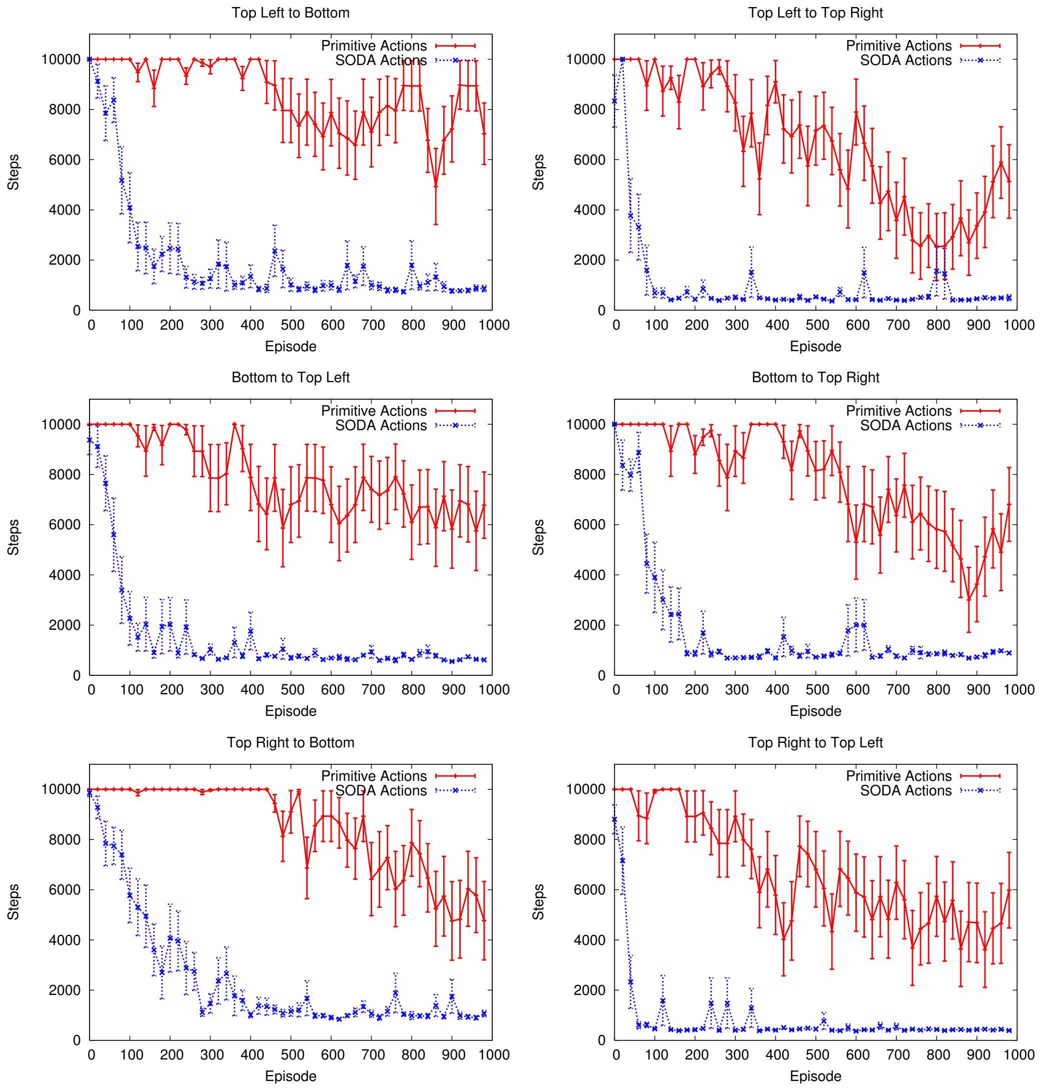
The first set of navigation experiments was conducted in the T-Maze, described in Section 4.2. For these experiments, three navigation targets were defined in the upper-left, upper-right, and bottom extremities of the environment, as shown in Figure 6.1. Each target was defined by a point in the environment. The robot was judged to have reached the target if its centroid passed within 500 mm of the target point. The agent was tested on its ability to learn to navigate between each pair of targets using Sarsa(λ) over $\mathcal{A}^0$ actions and over

$\mathcal{A}^1$ actions, to test whether the agents learn to navigate more effectively using SODA.

As explained in Section 1.3, SODA is not intended to address problems of partial observability (also known as perceptual aliasing), in which multiple distinct states in the environment have the same perceptual representation. However, in some cases the SODA abstraction induces perceptual aliasing in the environment. For example, in the T-Maze, the same GNG unit is active when the robot is in the center of the intersection facing southeast or facing southwest (unit 4 in the first row of Figure 4.2). Resolving such ambiguity is outside the scope of this dissertation. Therefore, to avoid resorting to complicated representations and algorithms for partially observable environments, the agents were given more sensory information to help reduce aliasing. The two new sensors were a stall warning to indicate collisions, and an eight-point compass. Using these sensors, the learning agent's state representation was the tuple $\langle$stall,compass,$(\mathrm{argmax}_j f_j \in \mathcal{F})\rangle$.

As with the Top-N representation (Section 3.2), the state tuple was hashed into an index into the Q-table. The reward function for the task gave a reward of 0 for reaching the goal, -1 for taking a step, -6 for stalling. Each episode timed out after 10,000 steps. When the robot reached the goal or timed out, the robot was automatically returned to the starting point. For each of the six pairs of starting and ending points, there were 18 total trials of 1000 episodes each: three trials using each of three different trained GNG networks using $\mathcal{A}^0$ actions and $\mathcal{A}^1$ actions. The Sarsa(λ) parameters were $\lambda = 0.9$, $\alpha = 0.1$, $\gamma = 1.0$. All Q values were initialized optimistically to 0. The agent also used ϵ-greedy action selection with $\epsilon_0 = 0.1$ annealing to $\epsilon_\infty = 0.01$ with a half-life of 100,000 (primitive) steps. The agents using $\mathcal{A}^1$ actions all learned their TF and HC options using the parameters described in Chapter 5. For each start-end pair, the experiment began with all options untrained, and option policy learning proceeded concurrently with high-level policy learning.

Figure 6.2 shows the learning curves comparing the performance of the agents using $\mathcal{A}^0$ actions to the agents using $\mathcal{A}^1$ actions. For every pair of start and end points, the agents using $\mathcal{A}^1$ actions learn dramatically faster than the agents using $\mathcal{A}^0$ actions. In each case,



Top Left to Bottom

Top Left to Top Right

Bottom to Top Left

Bottom to Top Right

Top Right to Bottom

Top Right to Top Left

Figure 6.2: **T-Maze Learning, all routes.** These learning curves show the length per episode for learning to navigate between each pair of targets shown in Figure 6.1. The curves compare learning with $\mathcal{A}^1$ actions and learning with $\mathcal{A}^0$ actions. Each curve is the aggregate of three runs using each of three trained GNGs. Error bars indicate +/- one standard error. In all cases the $\mathcal{A}^1$ agents dramatically outperform the $\mathcal{A}^0$ agents.

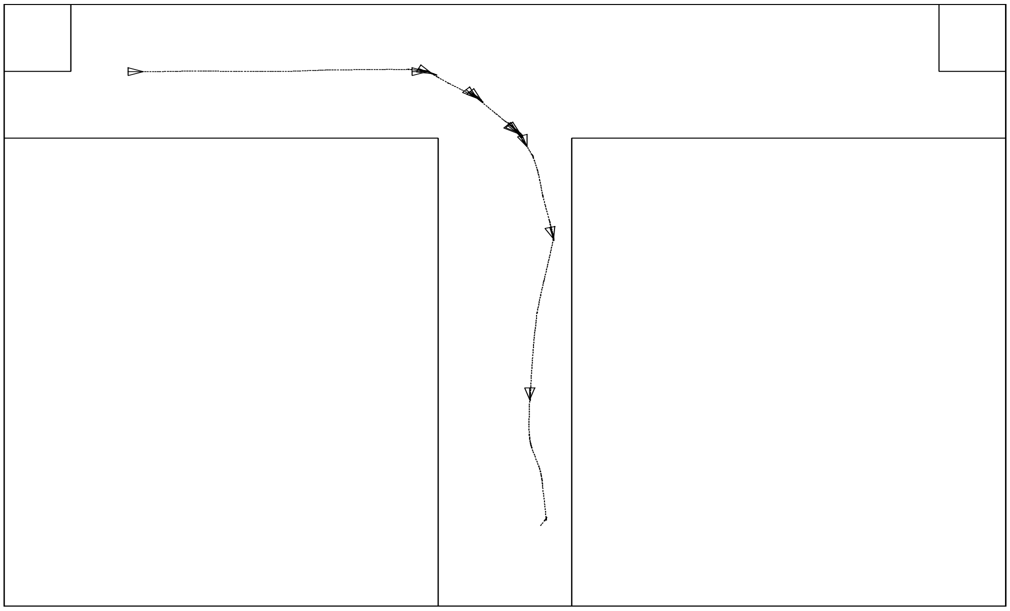


Figure 6.3: **Navigation using Learned Abstraction**. An example episode after the agent has learned the task using the $\mathcal{A}^1$ actions. The triangles indicate the state of the robot at the start of each $\mathcal{A}^1$ action. The sequence of winning features corresponding to these states is [8, 39, 40, 14, 0, 4, 65, 7, 30, 62], shown in Figure 6.4. The narrow line indicates the sequence of $\mathcal{A}^0$ actions used by the $\mathcal{A}^1$ actions. In two cases the $\mathcal{A}^1$ action essentially abstracts the concept, 'drive down the hall to the next decision point.' Navigating to the goal requires only 10 $\mathcal{A}^1$ actions, instead of hundreds of $\mathcal{A}^0$ actions. In other words, task diameter is vastly reduced.

the behavior of the agents using the high-level actions has converged easily within 1000 episodes (and often much sooner) while in no case did the behavior of the agents using primitive actions converge within that period. These results show that SODA significantly speeds up navigation learning.

Figure 6.3 shows a representative trace of learned behavior from one trained agent, traveling from the top-left location to the bottom location. This figure shows the starting points of the $\mathcal{A}^1$ actions as triangles, and the path of the agent in underlying $\mathcal{A}^0$ actions. The agent has abstracted a task requiring a minimum of around 300 primitive actions to a sequence of ten high-level actions. Figure 6.4 shows the perceptual features that define

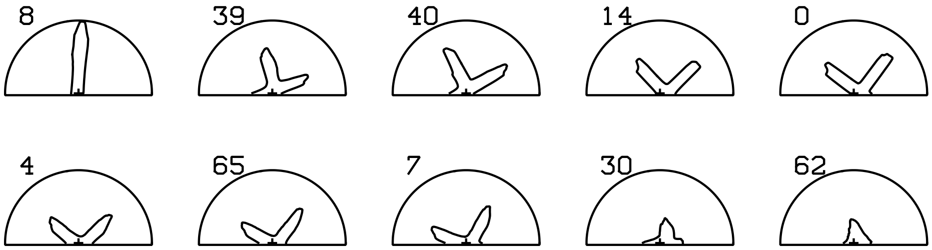


Figure 6.4: **Features for Distinctive States** These are the perceptual features for the distinctive states used in the navigation path shown in Figure 6.3, in the order they were traversed in the solution. (Read left-to-right, top-to-bottom.) The first feature [8] and the last two [30, 62] represent the distinctive states used to launch long actions down the hallways, while the intervening seven features [39, 40, 14, 0, 4, 65, 7] show the robot progressively turning to the right to follow the lower corridor. The large number of turn actions is caused by the large number of features in the GNG used to represent views separated by small turns, discussed in Section 4.4 and Figure 4.6. Despite the many turn actions, however, SODA still reduces the task diameter by an order of magnitude over primitive actions.

the distinctive states for the starting points of the actions in Figure 6.3. The trace and feature plots show that the agent began by traveling forward in a single $\mathcal{A}^1$ action from the starting point to the intersection, trajectory-following on a feature representing a view straight ahead down a long corridor. After reaching the intersection, the agent then used seven shorter actions to progress through the intersection while turning to face the lower corridor. Each of these seven distinctive states is defined by a feature resembling a view of an intersection turned progressively more at each step. Finally, the agent progresses down the lower corridor in two long actions. The lower corridor is about half the length of the upper corridor, and as the robot progresses down the corridor, the laser rangefinder view of the corridor grows progressively shorter. As a result, the view of the corridor is represented by two features (30 and 62) with shorter forward ranges than the feature used to represent the upper corridor (feature 8). Although, to the human eye, these final two features bear a less obvious resemblance to a corridor than the other features, they are still sufficient to allow the agent to navigate to the goal.

6.2 The Role of Hill-Climbing

The previous section showed that SODA agents learn to navigate faster than agents using primitive actions, and that SODA reduces the diameter of navigation tasks in the T-maze by an order of magnitude. The question arises, however: How much of the benefit of SODA comes from the trajectory-following and how much from hill-climbing? This section describes an ablation study comparing navigation using SODA with and without hill-climbing. The study shows that while all the speed-up in learning can be attributed to the TF component of the actions, the HC step makes the actions more deterministic and cuts the task diameter roughly in half compared to navigating using TF actions alone.

One of the stated goals of SODA, described in Section 1.4, is that it function as a building block in a bootstrap-learning process for robotics. One likely next stage of bootstrap learning is to move from the model-free navigation policies learned by Sarsa(λ) to navigation based on predictive models and explicit planning. Such systems benefit greatly from more deterministic actions, which have fewer possible outcomes, and thus reduce the branching factor of search for planning. They also benefit from shorter task diameters, which reduce the depth of search. As shown in the study in this section, the HC in $\mathcal{A}^1$ actions provides both of these benefits.

In the ablation study, the task of the agent was to navigate from the top-left location to the bottom location in the T-Maze (Figure 6.1). The parameters of the experiment were identical to those in the experiments in Section 6.1, with three exceptions: (1) In this case there were three experimental conditions: $\mathcal{A}^1$ actions, TF actions only, and $\mathcal{A}^0$ actions. (2) Each condition consisted of five independent learning agents for each of the ten different trained GNGs, to provide a larger sample for computing statistical significance, giving a total of $3 \times 5 \times 10 = 150$ total agents. (3) Each agent ran 5000 episodes, instead of 1000, to ensure that the TF and $\mathcal{A}^1$ agents ran until policy learning converged.

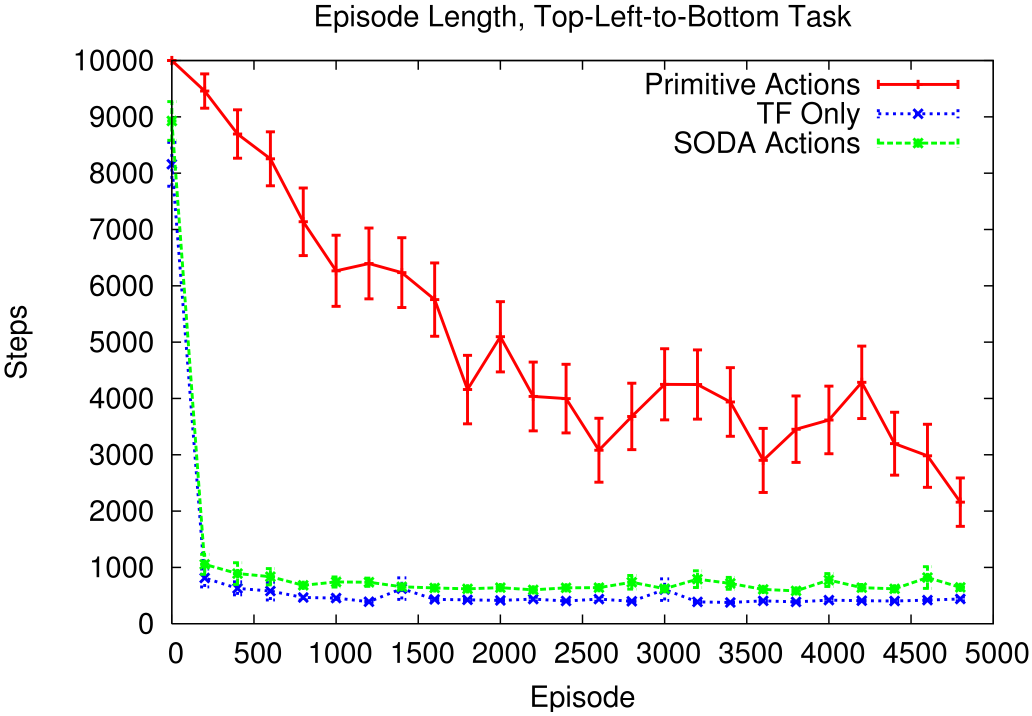


Figure 6.5: $\mathcal{A}^1$ **vs TF-only Learning Performance, T-Maze** These learning curves compare the length per an episode in the T-Maze top-left-to-bottom navigation task using primitive actions ($\mathcal{A}^0$), trajectory-following alone (TF), and trajectory following with hill-climbing ($\mathcal{A}^1$). Each curve is the average of 50 runs, 5 each using 10 different learned SOMs. Error bars indicate +/- one standard error. The agents using just TF actions, without the final HC step learn as fast and perform slightly better than the agents using $\mathcal{A}^1$ actions. (TF vs $\mathcal{A}^1$ performance is significantly different, $p < 0.0001$, in the last 50 episodes.)

The learning curves from the experiment are shown in Figure 6.5. In the first 1000 episodes, the $\mathcal{A}^1$ and $\mathcal{A}^0$ curves show the same basic shape as the learning curve for the same task above (Figure 6.2, top left plot), with lower variation in the average as a result of the larger number of agents run. Importantly, the TF-only curve is nearly identical to the $\mathcal{A}^1$ curve, showing that, in terms of learning speed, all the benefit of SODA actions comes from the TF component of the actions. In addition, the actions taken to perform hill-climbing exact a small cost. In the last 50 episodes, the average TF-only episode length is around 400 steps, while the average for TF+HC agents is around 600 steps over the same period. These differences are significant ($p < 0.0001$).

Although the hill-climbing component of $\mathcal{A}^1$ actions does not make learning faster, HC does improve navigation by making the actions more deterministic, as measured by lower *state transition entropy*. Given an environment's state-transition function $T(s, a, s')$, indicating the probability of ending in state s' after taking action a in state s, the state transition entropy of a state-action pair (s, a) is the entropy of the probability distribution over possible states s'. Because the entropy of a distribution is the number of bits needed to encode a selection from the distribution, the transition entropy can be interpreted informally as the $\log_2$ of the number of possible outcomes weighted by the likelihood of their occurrence. Thus an entropy of zero indicates exactly one possible outcome, an entropy of 1 indicates roughly two possible outcomes, etc.

To estimate transition entropy in the T-maze, every $\langle s, a, s' \rangle$ sequence the agent experienced over 5000 episodes was counted and the frequencies were used to estimate the distribution $P(s'|s, a)$. These estimates were then used to compute an entropy value for each (s, a) pair, and the entropies for all pairs were averaged to compute an average transition entropy for the run. Figure 6.6 compares the average transition entropy of agents using hill-climbing with those using only trajectory-following, for each GNG. This figure shows that transition entropies are an average of 0.4 bits lower for the agents using HC actions. This result corresponds to a decrease of about 30% in the number of possible outcomes for each state-action pair.

In addition to making actions more reliable, using hill-climbing also reduces the task diameter, as measured by the total number of abstract actions needed to reach the goal. Figure 6.7 shows the average length, in high-level actions, of successful episodes for agents using TF+HC actions versus agents using TF without HC. The bars show the average number of actions taken in the successful episodes taken from last 100 episodes for all TF+HC agents and all TF-only agents. (In 15 of the $100 \times 5 \times 2 = 10000$ episodes, the agents did not reach the goal before the episode timed out. These outliers were discarded

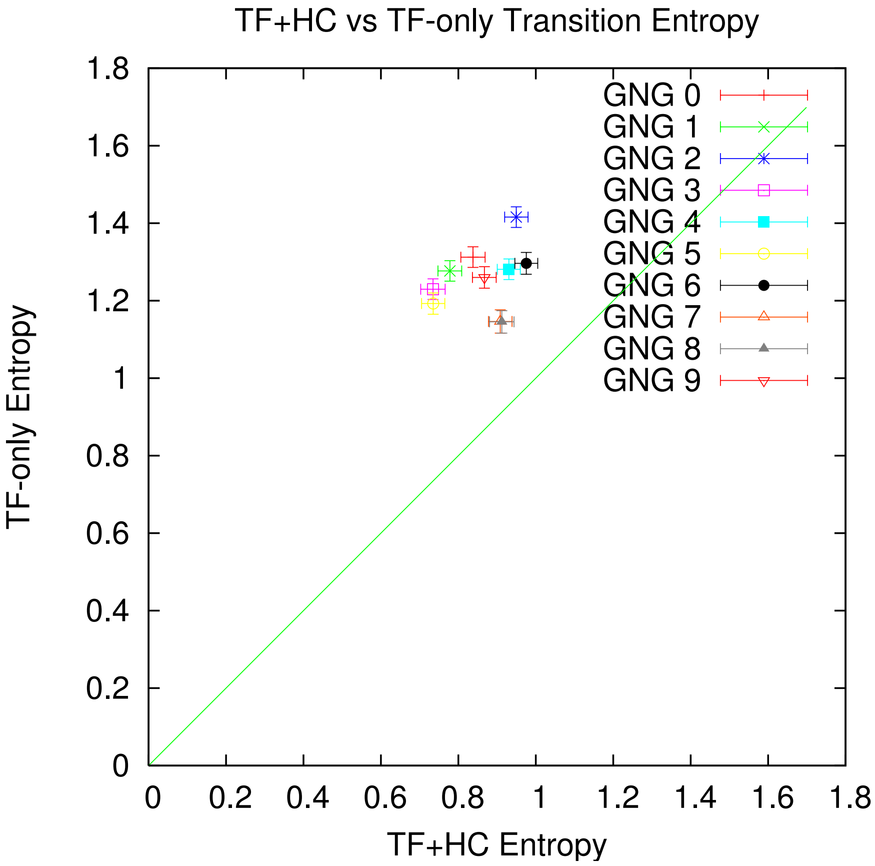


Figure 6.6: **Hill-climbing Reduces Transition Entropy.** These plots compare the average transition entropy for all state/action pairs for each of the 10 different GNGs used in the T-Maze experiment. The x-axis indicates the transition entropy (in bits) using hill-climbing, and the y-axis indicates the entropy without hill-climbing. The solid line indicates equal entropy ($y = x$). Error bars indicate 95% confidence intervals. Hill-climbing reduces the entropy by about 0.4 bits, on average. This is approximately equivalent to a reduction in branching factor from 3 to 2. These results indicate that hill-climbing makes actions more deterministic, making them more useful for building planning-based abstractions on top of SODA.

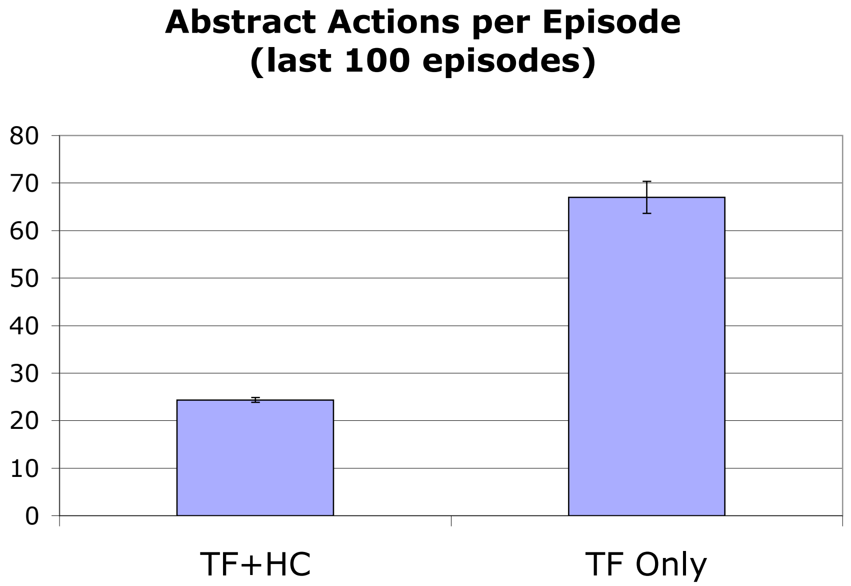


Figure 6.7: **Hill-climbing Improves Task Diameter.** Bars show the average abstract length per episode of the successful episodes taken from the last 100 episodes for each agent in the T-Maze experiment. Abstract length is measured in the number of high-level actions needed to complete the task. Using trajectory-following with hill-climbing, the agents require an average of 23 actions to complete the task, while without hill-climbing they require an average of 67. (Error bars indicate +/- one standard error. Differences are statistically significant with infinitesimal p.) This result indicates agents using hill-climbing the hill-climbing component of $\mathcal{A}^1$ actions will be make them more useful for building future, planning-based abstractions on top of the SODA abstraction.

because they are not representative of the learned task diameter.) The agents using HC actions are able to complete the task using an average of 23 actions while the agents using TF-only require an average of 67, a 62% decrease in task diameter.

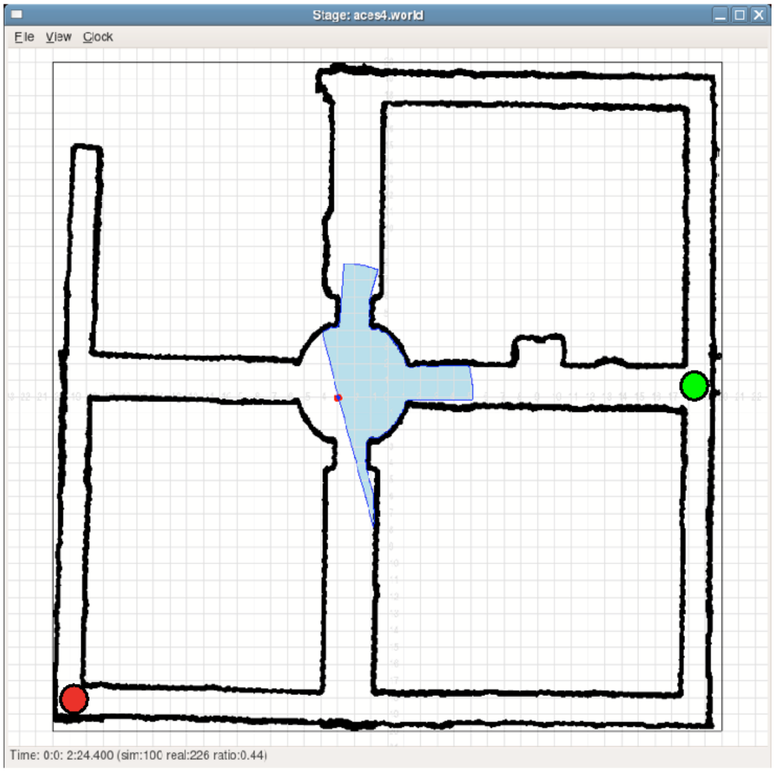


Figure 6.8: **ACES Navigation Task.** The circles indicate the locations used as starting and ending points for navigation in ACES in this chapter. The green circle on the right indicates the starting point and the red circle on the left indicates the ending point. The shaded area shows the robot's field of view. The longer task and added complexity of the environment make this task much more difficult than the tasks in the T-maze.

6.3 Scaling Up The Environment: ACES 4

The third experiment in this section investigates how SODA scales to the larger, richer ACES environment. As mentioned in Section 4.3, the ACES environment not only provides much longer task diameters at 40 m × 40 m, but it is far richer with a wide variety of different intersection types, different corridor widths, irregular wall features, and a circular atrium. In addition, the outer corridors of the environment are long and narrow, making it very unlikely for a robot with realistic motor noise robot to traverse them with open-loop trajectory-following macros, and thus requiring closed-loop TF.

The task in the ACES environment was to navigate from the center-right intersection to the lower-left corner of the environment, as shown in Figure 6.8. The experimental set-up and agent parameters were identical to those in the T-Maze experiments, with these exceptions: First, the general set-up of the environment and the robot was the same as described in Section 4.3. Second, the time-out for each episode was increased to 30,000 time-steps. Third, because the robot's rangefinder has a much shorter range than than the length of a typical corridor, many corridors and intersections are perceptually indistinguishable. To deal with this problem the agent was given extra state variables consisting of the eight-point compass and stall sensor used in the T-Maze experiment, plus a coarse tiling of the robot's (x, y) position in the environment, in which the x and y positions were tiled into ten-meter bins. This binning was chosen because it is sufficient to give each intersection a unique sensory signature, and can be interpreted as similar to an environment in which different areas of the building are painted a unique color. The agent was given the state tuple $\langle$stall,compass,x-bin,y-bin,$(\arg\max_j f_j \in \mathcal{F})\rangle$, which was hashed into the agent's Q-table.

Figure 6.9 shows the learning curves from the ACES environment. Each curve averages the performance of 50 agents — 5 runs using each of 10 trained GNGs — for agents using SODA's TF+HC options and agents using primitive actions only. The curves show that the SODA agents are able to learn to solve the navigation task while the agents using primitive actions are not able learn it within the alloted time-out period.

6.4 Navigation Discussion

The results in this chapter show that SODA allows an agent to learn to navigate dramatically faster than it can using primitive actions. However, there are two major points from these results worth further discussion: (1) SODA does not learn clean, high-level "turn" actions, and (2) the learning times in the larger environment are relatively long, compared to those in the T-maze.

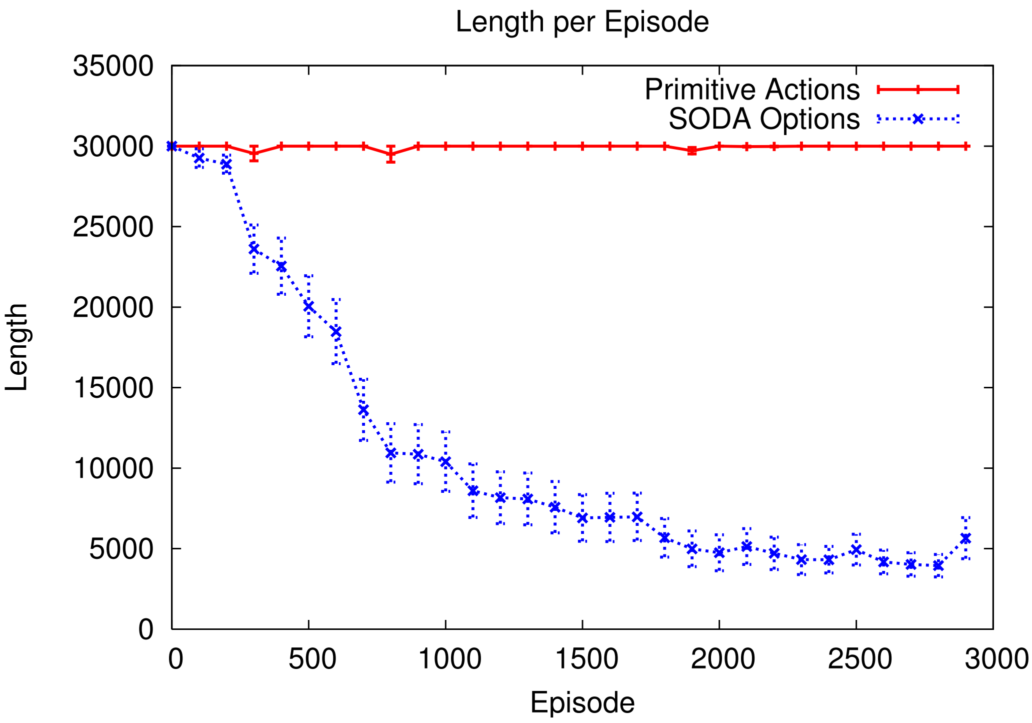


Figure 6.9: **Learning Performance in the ACES Environment.** These learning curves compare the length per episode for learning to navigate in ACES from the center-right intersection to the lower-left corner of the environment. The curves compare primitive actions and $\mathcal{A}^1$ actions. Each curve is the average of 50 runs, five each using 10 different learned SOMs. Error bars indicate +/- one standard error. The minimum length path is around 1200 actions. The agents using the high-level actions learn to solve the task while those using only primitive actions have a very difficult time learning to solve the task in the allotted time.

First, Figure 6.3 shows that although SODA learns to abstract “hallway following” into a few large actions, it does not cleanly learn to turn to a new corridor as a single large action. As discussed in Section 4.4, the Euclidean distance metric causes SODA to learn many perceptual features when turning, resulting in many closely spaced distinctive states separated by small turn actions. Using the $\mathcal{A}^1$ actions, the agent must traverse sequences of these states in order to turn 90°. Although SODA performs well despite this problem, an effective means of learning large turn actions could potentially reduce the task diameter by another order of magnitude. The ideal abstract path to the goal in the T-maze consists of three abstract actions: (1) drive to the intersection, (2) turn right, (3) drive to the goal. Achieving such an ideal on every run is unlikely, given that actions may terminate early because of motor noise, and that feature learning may not learn perfect features. However, given that the majority of the actions in the trace in Figures 6.3 and 6.4 are used to turn, a better method of learning turn actions could reduce the mean of 23 actions (Figure 6.7) to fewer than ten actions. Section 7.2.2 in the next chapter discusses methods by which SODA could be extended to more cleanly learn large turn actions, either by using alternative distance metrics that better handle turning, or by aggregation of states into "places" and "paths."

The second point for discussion is SODA's performance in the ACES environment. Although SODA's advantage over primitive actions in the larger environment is even greater than in the T-Maze, the absolute learning performance is notably worse than that in the T-Maze. In the T-Maze, performance converged in a few hundred episodes. In ACES, it took nearly the whole 3000 episodes. Much of this difference can be explained by the greater complexity of the environment. The T-Maze has only a single intersection, while ACES has eight. In addition, the state space in ACES is much larger: while the total number of states in the Q table of an agent in the T-Maze is around 500-600, the number of states for an agent in ACES is around 4000-5000. Note that because of the hashing used in the Q-table, the number of states in the table represents the number of states *visited* by the agent, not the total number of possible states in the environment. In other words, the agents in ACES have to explore much more environment (visit more states) in order to learn a solution. This substantial increase in complexity indicates that rapid learning in very large environments may require bootstrapping to a higher level of abstraction, such as the aggregation of states into places and paths as mentioned above and discussed in detail in Section 7.2.2.

6.5 Navigation Conclusion

To summarize, this chapter described experiments showing that SODA allows agents to learn to navigate much more quickly than agents using primitive actions. In the T-Maze, SODA agents out-performed agents using primitive actions in navigation between several locations in the environment. In addition, an ablation study showed that the hill-climbing phase of SODA's actions makes the actions more reliable than trajectory-following alone, and dramatically reduces the number of abstract actions needed to solve the task, at a small cost in the total number of primitive actions needed to solve the task. These features will be attractive for bootstrapping from model-free reinforcement learning methods to navigation using predictive models and planning. Finally, in the navigation experiment in the larger, more complicated ACES environment, SODA's advantage over primitive actions was even more pronounced, although the long learning times in this environment suggest that bootstrapping to a higher level of abstraction may be needed for learning to navigate in practice.

Chapter 7

Discussion and Future Directions

The last chapter showed that robotic agents learn to navigate much faster using SODA than using primitive actions, that the hill-climbing step of SODA's $\mathcal{A}^1$ actions makes actions more reliable and reduces the task diameter over using trajectory-following alone, and that SODA scales up to larger, realistic environments. This chapter discusses some potential problems raised in the previous chapters and how they might be addressed in future research. First, Section 7.1 reviews the SODA agents' need for extra state information to disambiguate perceptually aliased states, and proposes several replacements to Sarsa(λ) for learning navigation in partially observable environments. Next, Section 7.2 discusses the large number of features, and hence distinctive states, created to represent perceptual changes caused by turning the robot, and proposes how this problem may be dealt with either by replacing Euclidean distance in the GNG with a different metric, or by bootstrapping up to a higher $\mathcal{A}^2$ representation that aggregates distinctive states together into "places" and "paths." Section 7.3 considers how SODA may be scaled to even richer and more complicated sensorimotor systems. Finally, Section 7.4 characterizes how well SODA's goal of reducing the need for prior knowledge in constructing an autonomous robot has been met, and the extent to which the knowledge embedded in SODA's learning parameter settings might be further reduced.

7.1 SODA with Perceptual Aliasing

Under SODA's abstraction the environments used in the previous chapters suffer from the problem of perceptual aliasing, also called "partial observability", in which more than one distinctive state is represented by one GNG unit. Perceptual aliasing makes it difficult or impossible to navigate based only on current perceptual input, and considerable research exists on methods for learning to act in partially observable environments (Kaelbling, Littman, & Moore, 1996; Shani, 2004). SODA's design deliberately factors the problem of constructing a perceptual-motor abstraction away from the problem of perceptual aliasing, with the intent that the abstract representation constructed by SODA could be used as input to existing (or new) methods, thus bootstrapping up to a higher-level of representation, as described in Section 1.4. As a place-holder for one of these methods, the agents in Chapter 6 were given additional sensory information to disambiguate aliased states. In the T-Maze the agents were given an eight-point compass, to differentiate aliased poses in the intersection (e.g. facing southeast vs. facing southwest). In ACES, where the corridors are much longer than the maximum range of the rangefinder, the agent was also given a coarse tiling of the robot's x/y position in the environment, approximately equivalent to using different wall or floor colors in different parts of the building.

SODA's abstraction provides a ready interface for adding methods designed for reasoning and acting in aliased environments. Formally, a SODA agent's interaction with a perceptually aliased environment forms a *Partially Observable Markov Decision Process* (POMDP). A POMDP extends a standard Markov decision process (Section 2.1.2) consisting of a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, and a transition function $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ giving the probability that taking an action in a given state will lead to some subsequent state. In a POMDP, the agent is unable to observe the current state $s \in \mathcal{S}$ directly. Rather, it observes an observation o from a set of observations $\mathcal{O}$ where the probability of observing a particular observation o in state s is governed by an *observation function* $\Omega : \mathcal{S} \times \mathcal{O} \to [0, 1]$. SODA's distinctive states map on to the POMDP's states $\mathcal{S}$, the high-level actions $\mathcal{A}^1$ provide the actions, GNG units are the observations $\mathcal{O}$ and the GNG itself provides the observation function Ω. The remainder of this section discusses three major classes of methods for learning to act in partially observable environments: model-free methods that extend MDP-based reinforcement learning methods such as Sarsa(λ) and Q-learning to partially-observable environments without explicitly constructing a POMDP model, model-based methods that induce a POMDP from observations and then use it for planning and navigation, and newer model-based methods that construct other forms of predictive models than explicit POMDP models, based on experience.

The simplest method for bootstrapping SODA up to handling perceptual aliasing is simply to replace Sarsa(λ) for high-level navigation learning with another model-free reinforcement learning method designed to deal with partial-observability. Two methods discussed in Chapter 2, McCallum's U-Tree algorithm (Section 2.3.1) and Ring's CHILD (Section 2.2.5) have been successful in learning tasks in highly-aliased environments. Both of these methods track the uncertainty in the value of each (o, a) pair, i.e the uncertainty in $Q(o, a)$, and progressively build a memory m of recent observations and actions, such that the uncertainty in $Q(o, a, m)$ is minimized.

Another class of model-free learning methods, neuroevolution (NE), has had success on partially observable continuous control tasks (Gomez & Miikkulainen, 2002; Gomez, 2003). Neuro-evolution methods use an evolutionary search in the space of policy functions expressed as neural networks, without explicitly learning or storing a value function. Gomez, Schmidhuber, & Miikkulainen (2006) compared several neuroevolution methods against a variety of value-function-based and policy-search-based RL methods on a series of pole-balancing tasks. The NE methods dramatically outperformed the other methods, although neither U-Tree nor CHILD was included in the comparison. The pole-balancing tasks are low-dimensional, unlike the high dimensional input used in the SODA experiments. Nevertheless, it is worth considering whether these methods could be used to learn to navigate over SODA's $\mathcal{A}^1$ actions. One potential problem with using neuroevolution is the construction of the fitness function. For episodes in which the robot reaches the goal, the fitness should be inversely related to the number of steps the agent took to reach the goal. However, for episodes that time-out before reaching the goal, it is not clear how to fashion a fitness function without some external knowledge of the robot's actual position in the environment. This would nonetheless be an interesting direction for future research.

U-Tree, CHILD, and neuroevolution have all had success in learning to perform tasks in partially-observable environments. However, like Sarsa(λ) these methods create an entirely new learning problem for each navigation target in the environment, conflating general knowledge of the dynamics of the environment and specific knowledge for a single task. For example, in the set of six experiments described in Section 6.1, the agent had to learn the Q function, and hence the environment dynamics, afresh for each experiment. This is very inefficient, given that all the experiments are performed in the same environment, and the learned distinctive states, $\mathcal{A}^1$ actions, and state-transition function are shared among all the tasks. An alternative is to learn a model of the environment that allows the agent to predict the outcome of actions. This model can then be used to solve each navigation task in the environment, either by learning a state value function for each task (which is simpler than the state-action value functions learned by model-free methods), or by explicit look-ahead planning.

The most obvious representation for a predictive environment model for SODA is the POMDP representation itself. As mentioned above, the action set $\mathcal{A}$ and observation set $\mathcal{O}$ are already known, so learning the POMDP would entail estimating the hidden state set $\mathcal{S}$, transition function T, and observation function Ω. For a given $\mathcal{S}$, such a model can be learned from an observation/action trace using a generalized expectation-maximization algorithm for learning hidden Markov models known as the Baum-Welch algorithm (Duda, Hart, & Stork, 2001). Baum-Welch can also be used to learn POMDPs (Chrisman, 1992). When the number of hidden states is unknown, Baum-Welch can be applied repeatedly as the number of states is increased, to discover the number of states that maximizes the likelihood of a second, independently collected, validation trace. In addition to this iterative method, other search methods exist for learning graphical topologies in other domains, including Bayesian networks (Segal *et al.*, 2005; Teyssier & Koller, 2005), and neural networks (Stanley, 2003). It is likely that one or more of these methods could be extended to POMDPs, especially in cases where the transition function, and hence the graph topology, is very sparse, as it is with SODA—as shown in Figure 6.6, the average transition entropy in the T-Maze using $\mathcal{A}^1$ actions is less than 1 bit.

In addition to the POMDP model, new state representations for partially observable dynamical systems have been developed recently. Chief among these are *Predictive State Representations* (PSRs; Littman, Sutton, & Singh, 2002; Singh, James, & Rudary, 2004), and *Temporal-Difference Networks* (TD-Nets; Sutton & Tanner, 2005). Rather than explicitly representing each hidden state and the transitions between them, PSRs represent the agent's state as predictions over a set of *tests*, where a test is an action-conditioned sequence of future observations. Representationally, PSRs have advantages over history-based method like U-Tree and CHILD, in that they can accurately model some systems that cannot be modeled by any finite history method. In addition, PSRs are often more compact than the equivalent POMDP representation for the same underlying dynamical system. Given an appropriate set of "core tests," the parameters for updating a model can be learned (Singh *et al.*, 2003), and some progress has been made on discovering the core tests from data (Wolfe, James, & Singh, 2005), though the methods do not work well for all environments. However, navigation environments are considerably more constrained in their dynamics than what can be represented by a POMDP generally. For example, the agent cannot jump to arbitrary states, so the transition function is very sparse. It is worth investigating whether or not the properties that make PSR discovery fail in some cases apply in navigation.

TD-Nets are a different representation, also based on predictions, in which nodes in a graph represent scalar predictions about the world and links between them represent action-conditioned temporal dependencies between the predicted outcomes. The parameters of the network are learned from data using methods of temporal differences (Section 2.1.2), but no methods exist for discovering the predictions or network structure from experience, making it difficult to see how they can be applied to SODA.

Although SODA itself does not deal with perceptual aliasing, its sparse, discrete abstraction is well suited for use as input to a variety of methods for learning to act in partially observable environments. Investigating these methods is an attractive area for future research.

7.2 Dealing with Turns

Although the learning results in Chapters 5 and 6 show that SODA's feature learning method suffices to construct features that greatly reduce task diameter, the method still has the problem, discussed in Section 4.4, that turning the robot moves the perceptual vector farther in the feature space than translating forward or backward. Because the GNG network adds features to minimize the distance between the input and the winning feature, it creates many features that represent views of the environment that differ from one another by only a small turn. Since there is at least one distinctive state for each feature, this process creates many distinctive states that only differ by a small rotation. As a result, as shown in Figures 6.3 and 6.4, turning the robot requires several smaller $\mathcal{A}^1$ actions while traveling forward or backward only requires a few larger actions. Ideally, the abstraction would treat traveling and turning equivalently: traveling to an intersection, turning to face a new corridor, traveling again, etc. More generally, an ideal abstraction would cut the environment at its natural decision points, regardless of whether some primitive actions move the robot farther in perceptual space than others. SODA might be modified to create such an abstraction either by replacing Euclidean distance in the GNG network with another metric that better captures the distances between small actions, or by constructing a higher $\mathcal{A}^2$ layer of abstraction on top of the $\mathcal{A}^1$ layer, which aggregates the small turn actions together into larger actions.

These ideas are discussed in more detail below.

7.2.1 Alternative Distance Metrics for Feature Learning

One possible method of dealing with the large number of distinctive states created by turning is to replace Euclidean distance for comparing sensory images with a metric that can account for the relationships between the elements in the input vector. One such metric is *earth movers' distance*. Alternatively, it may be possible for SODA to learn a good distance metric for each new sensorimotor system.

Earth movers' distance (EMD; Rubner, Tomasi, & Guibas, 2000) is a distance metric used in computer vision that has properties well-suited for comparing sensor inputs. EMD computes the cost to "move" the contents of one distribution to best cover another distribution, given known distances between the bins of the distribution. By treating each input vector or GNG unit as a distribution histogram, it is possible to apply EMD to comparing inputs in SODA. Applying EMD this way requires a distance matrix $\mathbf{D} = [d_{ij}]$ indicating the distance between each sensor represented in the input vector. Although no such distance matrix is specified in SODA, the formalization in Chapter 3 assumes that the sensor group that produces the input vector was discovered using the sensor-grouping methods of Pierce & Kuipers (1997). These methods include automatically learning a distance matrix over the robot's sensor array that assigns small distances to sensors that are correlated (i.e. that sample nearby regions of space), and larger distances to sensors that are less correlated.

Using this distance matrix with the robot used in this dissertation, EMD should assign much smaller distances between input vectors separated by small turns than between input vectors separated by large turns, because with small turns the similar range values are much closer to one another (in terms of $\mathbf{D}$). One potential drawback to EMD in this context is that computing a single distance directly requires solving a linear program. This may be too computationally intensive for a program that must compute many distances many times per second. However, approximate EMD-like methods have been developed that provide significant speedups (Grauman & Darrell, 2007, 2005). Alternatively, it may be possible to use the computationally intensive EMD to generate training data for a faster function approximator that computes an approximate EMD between arbitrary input pairs.

An alternative to choosing a specific new distance function such as EMD is to have the SODA agent learn the distance function from experience. Vector-space similarity function learning has already been studied for use in clustering (Xing *et al.*, 2003) and text mining (Bilenko & Mooney, 2003). These methods require information that indicates which pairs of vectors are similar (or dissimilar). This information is used to train a function approximator that learns a distance function between pairs of vectors. In the case of SODA, this information is available in history of the training phase for the GNG. Since one of SODA's assumptions is that small actions produce small changes in the input (Section 3.1), input vectors separated by a single action can be defined to have a small distance between them. This information can be used to learn a distance metric in which turning actions and translating actions are separated by similar distances, reducing the number of GNG units (and hence distinctive states) devoted to representing variants of the same view that only differ by small turn actions.

To summaraize, if Euclidean distance in GNG training is replaced with a new distance metric that better represents the topological distance between inputs in the action space, SODA may be able to reduce task diameters in environments like the T-Maze and ACES by another order of magnitude.

7.2.2 Learning Places, Paths, and Gateways

An alternative, potentially more promising method for dealing with turns may be to aggregate the large groups of distinctive states (dstates) linked by small turn actions into *places*, connected together by *paths* consisting of a few dstates linked by long "travel" actions. This is the approach used by the Spatial Semantic Hierarchy (Section 2.1.1) in moving from the *causal* abstraction level to the *topological* level. The SSH causal level is characterized by distinctive states linked by actions, while the topological level consists of *places* linked into sequences along paths. The resulting abstraction forms a bipartite graph of places and paths, in which the agent can plan. Building such an abstraction upon SODA's dstates and actions would both reduce task diameter further and form an abstraction suitable for navigation by planning.

The classic SSH formalization requires prior labeling of particular actions as either *travel* actions or *turn* actions. This labeling is antithetical to SODA's purpose of reducing the need for human prior knowledge for the learning process. It may be possible to get around the need for action labeling by adapting the concept of *gateways* from the *Hybrid SSH* (HSSH Kuipers *et al.*, 2004; Beeson *et al.*, 2003). The HSSH combines the strengths of topological mapping for representation of large-scale space, and metrical, probabilistic methods (Thrun, Burgard, & Fox, 2005) for representation and control in small-scale space. The HSSH labels each place (a decision point or intersection of paths) with a local perceptual model (e.g. an occupancy grid). Each place has a discrete set of *gateways* through which paths enter and leave. These gateways define the interface between the large-scale, topological representation, and the small-scale metrical representation. An analogous abstraction could be constructed from SODA's actions and distinctive states by identifying the dstates that begin or end sequences of one or more long actions, all of which begin by trajectory-following along the same progress vector (Section 3.3). These starting states would be labeled as "gateways"; the sequences of long actions (and associated dstates) that connect them would be labeled as "path segments"; and the groups of dstates and small actions that connect the path segments together would become "places." An example of such an abstraction in the T-Maze is shown in Figure 7.1. This abstraction requires a definition of what constitutes a "long" action. One method is to cluster all the $\mathcal{A}^1$ action instances in the training data into two sets using k-means, and treat the longer set as the "long" actions.

Control in this new topological $\mathcal{A}^2$ abstraction could be accomplished by a new set

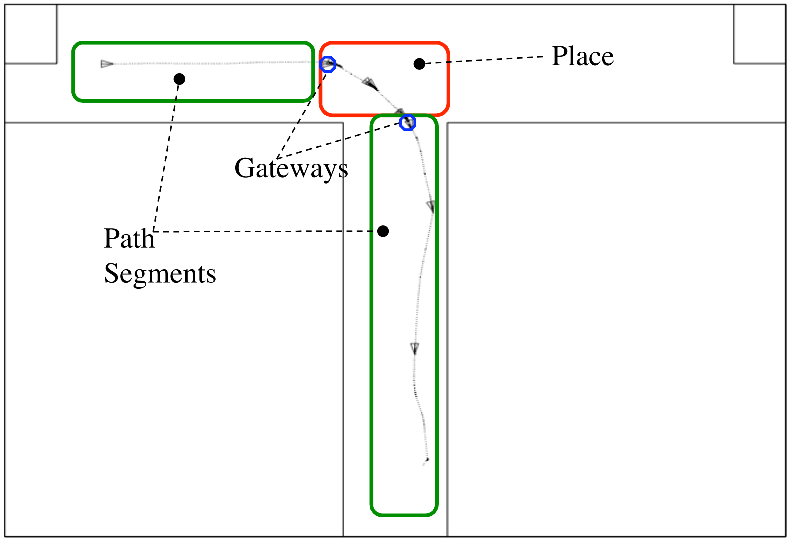


Figure 7.1: **Grouping SODA Actions into Gateways, Places and Paths.** Topological abstraction of SODA's actions could be accomplished identifying the states that initiate or terminate sequences of long similar actions as *gateways*, and then aggregating the sequences of states connected by long actions between two gateways into *paths*, and the collections of states connecting groups of gateways into *places*. In this T-Maze example, from Chapter 6, the states in which the robot enters and leaves the intersection terminate and begin sequences of long actions, and thus are gateways. The states and actions moving the robot down the corridor are grouped into paths, and the collection of states traversed in turning are grouped into a place. This second-level ($\mathcal{A}^2$) abstraction would reduce the diameter of this task from ten actions to three.

of options defined over $\mathcal{A}^1$ actions, allowing the robot to navigate robustly between gateways even when motor noise causes a TF action to terminate prematurely. Each gateway leaving a place would have an associated option for reaching that gateway from any dstate within the place (including the incoming gateways); likewise, each path segment would have an option for reaching the end gateway from anywhere within the path segment. Limiting these $\mathcal{A}^2$ actions to operating over $\mathcal{A}^1$ actions is likely to make their learned polices somewhat less efficient than similar options over $\mathcal{A}^0$. Once policies for these options have been learned using $\mathcal{A}^1$ actions, however, it may be possible to replace each option with a new optimized version that operates over $\mathcal{A}^0$ actions. Each optimized option's policy would be bootstrapped with action traces generated by the old $\mathcal{A}^1$-based policy, as was done by Smart & Kaelbling (2002), resulting (after further learning) in $\mathcal{A}^2$ actions that drive the robot directly to the gateways without intermediate trajectory-following and hill-climbing. Such an abstraction would reduce the task diameter considerably, reducing, for example, the diameter of the task in Figure 7.1 from ten actions to three.

7.3 Scaling to More Complex Sensorimotor Systems

The results in Section 6.3 show that SODA scales up to navigation environments of realistic size. Increasing the environment size, however, is only one kind of scaling. An important future direction for research in SODA is investigating how the abstraction scales along other dimensions, such as input dimensionality and output dimensionality.

The SODA experiments in previous chapters show that SODA operates well with 180 input dimensions, which is many more than the typical reinforcement learning system in which hand-coded feature extractors reduce the dimensionality fewer than 10 dimensions. Nevertheless, there are obvious ways in which the input dimensionality may be scaled up even further, either by using sensors of enormous dimensionality, such as cameras (a 640x480 RGB camera provides nearly a million individual input elements), or by adding multiple heterogeneous sensors. While in some limited environments it may be possible to pass full camera images or other very large input vectors directly to a single, huge SOM, such an approach is likely to be highly inefficient, as SOM lookups for such large, dense vectors are computationally very expensive. Instead, handling these very high-D inputs will require an extension to SODA's feature learning. Rather than learn a single flat feature set to characterize the whole input, a possible extension is to divide the large sensor group into many smaller (possibly overlapping) groups of scalar elements. Such groups are computed as an intermediate stage in the sensor grouping method of Pierce & Kuipers (1997). These groups can then each be characterized by its own SOM. Once these SOMs have been trained, a similar grouping can be performed on the SOMs, and another layer of SOMs learned to characterize the joint behavior of the first layer, and so on.

This scheme forms a *multistage hierarchical vector quantizer* (Luttrell, 1988; Luttrell, 1989). These kinds of networks are known to trade a small loss of encoding accuracy for a large increase in computational efficiency. This hierarchical approach is also reminiscent of CLA, the Constructivist Learning Architecture (Chaput, 2004) (Section 2.2.2). In CLA the choice of which lower level maps to combine in a higher-level map was made by the human engineer, but it may be possible to automate the process using distance-based grouping. In particular, since each SOM's output is effectively a discrete random variable, it may be possible to group them based on information-theoretic distance measures that maximize mutual information between grouped variables. Such methods were used by Olsson, Nehaniv, & Polani (2006). By building and combining such groupings, SODA should be able to scale its feature learning to very large input domains.

Another possible direction for scaling up is in the dimensionality of the robot's motor system. Many newer robot systems, such as manipulators, robot dogs, and humanoids, have many degrees of freedom, yet trajectory-following and hill-climbing still appear to be reasonable high-level actions for these more complicated robots. For example, it is easy to imagine a robot arm retrieving an object by following a trajectory to place the gripper near the object, then hill-climbing to a suitable position to grasp the object. The problem in these robots is that value-function-based reinforcement learning methods scale poorly as the size of the action space increases. One possible way of dealing with this problem would be to replace Sarsa-based learning in the TF and HC options with methods such as neuroevolution or policy-gradient methods, that learn their policies directly, rather than learning a value function. By encapsulating the interface to the high-dimensional action space in this way, much of the problem of scaling the robot's motor space should be alleviated.

7.4 Prior Knowledge and Self-Calibration

One of the stated long-term goals of SODA is to minimize or eliminate the need for human engineers to embed in each robot specific knowledge of its own sensorimotor system and environment. In the experiments in the previous chapters, SODA has largely succeeded in this, since parameters of its perceptual function and control policies are learned from experience (e.g. the GNG size and weights, and the control option Q-tables). In addition, although the *learning* parameters for those functions were still set by hand, the specification of a handful of learning parameters is often considerably more compact than explicit specification of the functions to be learned. For example, the hand-specified predictive motion model for hill-climbing in Section 5.2.1 is much more complicated to specify than the hill-climbing options learned in the following section. Furthermore, as with the perceptual learning described in Section 4.4, the parameters used for the options and other learning methods were chosen by relatively short "educated walks" in the parameter space — exhaustive searching was not required. Also, most of the learning parameters used in the T-Maze were used unmodified in ACES.

Nevertheless this kind of "knob-tweaking," common to all machine learning practice, embeds human prior knowledge and should be reduced, with the ultimate goal of achieving self-calibrating or "calibration-free" robots (Graefe, 1999). Assuming that there is no single set of learning parameters that will work for SODA in all robots and environments, two methods of automatically setting the parameter values present themselves for further study: homeostatic control, and evolutionary search.

"Homeostasis" is the term used in biology to describe a property of an organism (e.g. body temperature) that is kept constant by the organism's physiological processes. Homeostasis is distinguished from "equilibrium" by *active control*. While a closed system will fall into equilibrium in the absence of outside perturbation, an organism maintains its systems in homeostasis by actively changing the value of some parameters of the system to achieve constant target values on some measure or measures of system behavior.

Homeostasis in learning has been studied in computational neuroscience, modeling neurons whose activation threshold is modified to maintain a target firing rate distribution over time (Turrigiano, 1999; Triesch, 2005; Kurniawan, 2006).

The Homeostatic-GNG used in SODA for feature learning employs a homeostatic mechanism to set the size of the GNG, adding units only when the error rate rises above a set threshold, and deleting units that poorly represent the input distribution. The success of Homeostatic-GNG and the work in computational neuroscience suggest that homeostatic processes might be able to reduce manual search for parameter settings in SODA. It must be noted, however, that homeostatic processes do not eliminate prior parameter specification, but rather replace one or many parameters with a single new parameter, the homeostatic target value for the process. Homeostasis is effective when the same target value can be used in situations where the control parameters would have to change. For example in the T-Maze and ACES experiments, the Homeostatic-GNG error threshold used was the same between environments, while the number of features needed to maintain that error rate was much higher in ACES because of its greater perceptual variety.

It is possible that some parameters may not be controllable homeostatically. One important parameter that seems to fall into this category is the length of the initial exploration period for feature learning. To learn a complete feature set, the robot must explore the environment long enough to experience the full range of possible perceptual situations. Beyond that, extra exploration helps refine the features, but there is a point of diminishing returns. Experiments with learning high-level navigation concurrently with the feature set (by allowing the Q-table to grow as new features were added) were unsuccessful for two reasons. First, the navigation learner was chasing a moving target – by the time the Q-values were backed-up, the states they belonged to had changed. Second, they had poor exploration: the “optimistic initialization” exploration policy requires having a good representation of the state space before learning so that the Q-table can be initialized properly. For these reasons feature set needed to be learned and fixed before successful navigation.

This kind of multi-stage learning fits well with constructivist approaches to agent learning such as Drescher's Schema Mechanism and Chaput's CLA (Section 2.2.2), and it also fits with Bayesian-network-based hierarchical learning models of the cerebral cortex that train and fix each hierarchical layer before training the layers above (Dean, 2005). If, in fact, a period of dedicated exploration is truly necessary for SODA to learn a good set of features for navigation, then the length of exploration is a free parameter that cannot be set by a homeostatic process. However, it may be possible to set the length of the exploration phase by resetting the timeout clock each time a new feature is added. I.e., if the time elapsed since the last feature was added exceeds some preset value, then feature learning ends and the agent moves to the policy learning phase with a fixed set of features.

Alternatively, however, an evolutionary search for parameters may be more appropriate, where "evolutionary search" refers to any search method that generates parameter sets, tests their fitness by running each one in an entire developmental cycle, and uses the fittest to generate new parameters. Such an evolutionary search would not be particularly efficient in a single-robot learning scenario. It could work well, however, in an environment where many robots must perform similar tasks, since different parameter sets could be evaluated in parallel, and the fittest agents could be easily replicated into all the robots.

To summarize, SODA succeeds in its goal of learning without needing prior knowledge in that it is constructed entirely out of generic learning methods, and does not embed any implicit knowledge of the robot or environment, such as the detailed sensor and action models typically used in probilistic robotics (Thrun, Burgard, & Fox, 2005). However, some implicit prior knowledge is still embedded in the settings of SODA's learning parameters. Homeostatic control and evolutionary parameter search may allow SODA to reduce the amount of such implicit prior knowledge necessary to learn, moving the system closer to the goal of creating a self-calibrating robot.

7.5 Discussion Summary

This chapter has examined several of the most important directions for future research on SODA: dealing with perceptual aliasing, eliminating the large number of distinctive states created by turn actions either through a new distance metric in the GNG, or by aggregating them into “places,” scaling the method to more complex sensorimotor systems, and further reducing or eliminating the amount of human-provided prior knowledge built into the system. The next chapter summarizes this dissertation and concludes by placing this work in the greater context of bootstrap learning, and, more generally, the construction of intelligent agents.

Chapter 8

Summary and Conclusion

To summarize, this dissertation has presented Self-Organizing Distinctive-state Abstraction (SODA), a method by which a learning agent controlling a robot can learn an abstract set of percepts and actions that reduces the effective diameter of large-scale navigation tasks. The agent constructs the abstraction by first learning a set of prototypical sensory images, and then using these to define a set of high-level actions that move the robot between perceptually distinctive states in the environment.

The agent learns the set of perceptual features using a new, modified Growing Neural Gas vector quantizing network called *Homeostatic-GNG*. Homeostatic-GNG incrementally adds features in the regions of the input space where the representation has the most error, stopping when the accumulated training error over the input falls below a given threshold, and adding new units again if the error rises above the threshold. This algorithm allows the agent to train the network incrementally with the observations it experiences while exploring the environment, growing the network as needed, for example when it enters a new room it has never seen before. This automatic tuning of the feature set size makes it possible for the agent to learn features for environments of different size and perceptual complexity without changing the learning parameters.

After learning perceptual features, SODA uses these features to learn two types of abstract actions *trajectory-following* (TF) actions and *hill-climbing* (HC) actions. A TF action begins at a distinctive state, and carries the robot into the neighborhood of a new perceptual feature. An HC action, begins in the neighborhood of a perceptual feature, and takes the robot to a new distinctive state at a local maximum of this new feature. By pairing TF and HC actions, the agent navigates from one distinctive state to another. Both TF and HC actions are defined using the Options formalism for hierarchical reinforcement learning, and their policies are learned from experience in the environment using a novel prototype-based state abstraction called the Top^n representation. Top^n represents the agent's state as the sorted tuple of the top n closest perceptual prototypes, allowing TF and HC options to use the GNG features for navigation, even when navigating entirely within the neighborhood of one feature. In addition, the Top^n representation can be hashed into a table index, so that options can be learned with off-the-shelf tabular reinforcement learning algorithms like Sarsa and Q-learning.

Operating in this new, abstracted state space, moving from one distinctive state to the next, SODA agents then use reinforcement learning to learn to navigate between widely separated locations in their environment, using a relatively small number of abstract actions, each comprising many primitive actions.

Experiments using a simulated robot with a high-dimensional range sensor and drive-and-turn effectors showed that the SODA was able to learn good sets of prototypes, and reliable, efficient TF and HC options, in each environment. TF and HC options performed as well as hard-coded TF and HC controllers constructed with extensive knowledge of the robot's sensorimotor system and environment dynamics. In addition, they outperformed hard-coded controllers requiring no such prior knowledge. Using these learned perceptual features and actions, SODA agents navigating between distant locations in these environments learned to navigate to their destinations dramatically faster than agents using primitive, local actions. An ablation study showed that although HC options do not significantly improve overall navigation learning times, they make the high-level actions more reliable and reduce the average number of abstract actions needed to navigate to a target. These features make the abstraction more suitable for future extensions to model-based navigation by planning.

SODA learns its state-action abstraction autonomously, and the abstraction reflects only the environment and the agent's sensorimotor capabilities, without external direction. This abstraction is learned through a process of *bootstrap learning*, in which simple or low-level representations are learned first and then used as building blocks to construct more complex or higher-level representations. SODA bootstraps internally by first learning its perceptual representation using an unsupervised, self-organizing algorithm, and then using that representation to construct its TF and HC options, which are then used to learn high-level navigation. In addition, SODA itself is designed to be used as a building block in a larger bootstrap learning process: The identity of the main sensor group that SODA uses for learning perceptual prototypes can be determined automatically by existing lower-level bootstrap learning methods like those of Pierce & Kuipers (1997). In addition, SODA's representation of discrete states and actions is suitable for bootstrapping up to even higher representations, such as topological maps or learning agents that can function in the presence of hidden state.

Bootstrap learning is one approach to a larger problem: how can we create intelligent systems that autonomously acquire the knowledge they need to perform their tasks. This problem is predicated on the notion that an essential objective of AI is "the turning over of responsibility for the decision-making and organization of the AI system to the AI system itself." (Sutton, 2001) Achieving this objective is an enormous task that must be approached incrementally. This dissertation is one such step. This step enables a robot to learn an abstraction that allows it to navigate in large scale space—a foundational domain of commonsense knowledge—and can itself be used as the foundation for learning higher-level concepts and behaviors.

Bibliography

Andre, D., and Russell, S. J. 2001. Programmable reinforcement learning agents. In *Advances in Neural Information Processing Systems 12*, 1019–1025.

Baird, L. 1995. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, 30–37. Morgan Kaufmann.

Bednar, J. A.; Choe, Y.; De Paula, J.; Miikkulainen, R.; Provost, J.; and Tversky, T. 2004. Modeling cortical maps with Topographica. *Neurocomputing* 58-60:1129–1135.

Beeson, P.; MacMahon, M.; Modayil, J.; Provost, J.; Savelli, F.; and Kuipers, B. 2003. Exploiting local perceptual models for topological map-building. In *IJCAI-2003 Workshop on Reasoning with Uncertainty in Robotics (RUR-03)*.

Beeson, P.; Murarka, A.; and Kuipers, B. 2006. Adapting proposal distributions for accurate, efficient mobile robot localization. In *IEEE International Conference on Robotics and Automation*.

Bilenko, M., and Mooney, R. J. 2003. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2003)*, 39–48.

Chaput, H. H., and Cohen, L. B. 2001. A model of infant causal perception and its development. In *Proceedings of the 23rd Annual Conference of the Cognitive Science Society*, 182–187. Hillsdale, NJ: Erlbaum.

Chaput, H. H.; Kuipers, B.; and Miikkulainen, R. 2003. Constructivist learning: A neural implementation of the schema mechanism. In *Proceedings of the Workshop on Self-Organizing Maps (WSOM03)*.

Chaput, H. H. 2001. Post-Piagetian constructivism for grounded knowledge acquisition. In *Proceedings of the AAAI Spring Symposium on Grounded Knowledge*.

Chaput, H. H. 2004. *The Constructivist Learning Architecture: a model of cognitive development for robust autonomous robots.* Ph.D. Dissertation, The University of Texas at Austin. Technical Report AI-TR-04-34.

Chrisman, L. 1992. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In Swartout, W., ed., *Proceedings of the Tenth National Conference on Artificial Intelligence*, 183–188. Cambridge, MA: MIT Press.

Cohen, J. D.; MacWhinney, B.; Flatt, M.; and Provost, J. 1993. PsyScope: An interactive graphic system for designing and controlling experiments in the psychology laboratory using Macintosh computers. *Behavioral Research Methods, Instruments and Computers* 25(2):257–271.

Cohen, L. B.; Chaput, H. H.; and Cashon, C. H. 2002. A constructivist model of infant cognition. *Cognitive Development*.

Dean, T. 2005. A computational model of the cerebral cortex. In *The Proceedings of the Twentieth National Conference on Artificial Intelligence*. MIT Press.

Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research* 13:227–303.

Digney, B. 1996. Emergent hierarchical control structures: Learning reactive / hierarchical relationships in reinforcement environments. In *Proceedings of the Fourth Conference on the Simulation of Adaptive Behavior: SAB 98*.

Digney, B. 1998. Learning hierarchical control structure for multiple tasks and changing environments. In *Proceedings of the Fifth Conference on the Simulation of Adaptive Behavior: SAB 98*.

Drescher, G. L. 1991. *Made-up minds: A constructivist approach to artificial intelligence*. Cambridge, MA 02142: The MIT Press.

Duckett, T., and Nehmzow, U. 2000. Performance comparison of landmark recognition systems for navigating mobile robots. In *Proc. 17th National Conf. on Artificial Intelligence (AAAI-2000)*. AAAI Press/The MIT Press.

Duda, R. O.; Hart, P. E.; and Stork, D. G. 2001. *Pattern Classification*. New York: John Wiley & Sons, Inc., second edition.

Fritzke, B. 1995. A growing neural gas network learns topologies. In *Advances in Neural Information Processing Systems 7*.

Fritzke, B. 1997. A self-organizing network that can follow non-stationary distributions. In Gerstner, W.; Germond, A.; Hasler, M.; and Nicoud, J.-D., eds., *Proceedings of the Seventh International Conference on Artificial Neural Networks: ICANN-97*, volume 1327 of *Lecture Notes in Computer Science*, 613–618. Berlin: Springer.

Gerkey, B.; Vaughan, R. T.; and Howard, A. 2003. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, 317–323.

Gomez, F., and Miikkulainen, R. 2002. Robust nonlinear control through neuroevolution. Technical Report AI02-292, Department of Computer Sciences, The University of Texas at Austin.

Gomez, F.; Schmidhuber, J.; and Miikkulainen, R. 2006. Efficient non-linear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning*.

Gomez, F. 2003. *Robust Non-linear Control through Neuroevolution*. Ph.D. Dissertation, The University of Texas at Austin, Austin, TX 78712.

Gordon, G. J. 2000. Reinforcement learning with function approximation converges to a region. In *Advances in Neural Information Processing Systems*, 1040–1046.

Graefe, V. 1999. Calibration-free robots. In *Proceedings 9th Intelligent System Symposium*. Japan Society of Mechanical Engineers.

Grauman, K., and Darrell, T. 2005. The pyramid match kernel: Discriminative classification with sets of images. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

Grauman, K., and Darrell, T. 2007. Approximate correspondences in high dimensions. In *Advances in Neural Information Processing Systems (NIPS)*, volume 19.

Hume, D. 1777. *An Equiry Concerning Human Understanding*. Clarendon Press, Oxford.

Jonsson, A., and Barto, A. G. 2000. Automated state abstraction for options using the U-Tree algorithm. In *Advances in Neural Information Processing Systems 12*, 1054–1060.

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

Kohonen, T. 1995. *Self-Organizing Maps*. Berlin: Springer.

Kroöse, B., and Eecen, M. 1994. A self-organizing representation of sensor space for mobile robotnavigation. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems.*

Kuipers, B., and Beeson, P. 2002. Bootstrap learning for place recognition. In *Proc. 18th National Conf. on Artificial Intelligence (AAAI-2002).* AAAI/MIT Press.

Kuipers, B.; Modayil, J.; Beeson, P.; MacMahon, M.; and Savelli, F. 2004. Local metrical and global topological maps in the Hybrid Spatial Semantic Hierarchy. In *IEEE International Conference on Robotics & Automation (ICRA-04).*

Kuipers, B.; Beeson, P.; Modayil, J.; and Provost, J. 2006. Bootstrap learning of foundational representations. *Connection Science* 18(2).

Kuipers, B. 2000. The Spatial Semantic Hierarchy. *Artificial Intelligence* 119:191–233.

Kurniawan, V. 2006. Self-organizing visual cortex model using a homeostatic plasticity mechanism. Master's thesis, The University of Edinburgh, Scotland, UK.

Lee, S. J. 2005. Frodo Baggins, A.B.D. *The Chronicle of Higher Education*.

Littman, M.; Sutton, R. S.; and Singh, S. 2002. Predictive representations of state. In *Advances in Neural Information Processing Systems*, volume 14, 1555–1561. MIT Press.

Luttrell, S. P. 1988. Self-organizing multilayer topographic mappings. In *Proceedings of the IEEE International Conference on Neural Networks* (San Diego, CA). Piscataway, NJ: IEEE.

Luttrell, S. P. 1989. Hierarchical vector quantisation. *IEE Proceedings: Communications Speech and Vision* 136:405–413.

MacWhinney, B.; Cohen, J. D.; and Provost, J. 1997. The PsyScope experiment-building system. *Spatial Vision* 11(1):99–101.

Martinetz, T. M.; Ritter, H.; and Schulten, K. J. 1990. Three-dimensional neural net for learning visuomotor coordination of a robot arm. *IEEE Transactions on Neural Networks* 1:131–136.

McCallum, A. K. 1995. *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. Dissertation, University of Rochester, Rochester, New York.

McGovern, A., and Barto, A. G. 2001. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Machine Learning: Proceedings of the 18th Annual Conference*, 361–368.

McGovern, A. 2002. *Autonomous Discovery of Temporal Abstractions from Interaction with an Environment*. Ph.D. Dissertation, The University of Massachusetts at Amherst.

Miikkulainen, R. 1990. Script recognition with hierarchical feature maps. *Connection Science* 2:83–101.

Mitchell, T. M. 1997. *Machine Learning*. WCB/McGraw Hill.

Modayil, J., and Kuipers, B. 2004. Bootstrap learning for object discovery. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Modayil, J., and Kuipers, B. 2006. Autonomous shape model learning for object localization and recognition. In *IEEE International Conference on Robotics and Automation*.

Nehmzow, U., and Smithers, T. 1991. Mapbuilding using self-organizing networks in really useful robots. In *Proceedings SAB '91*.

Nehmzow, U.; Smithers, T.; and Hallam, J. 1991. Location recognition in a mobile robot using self-organizing feature maps. Research Paper 520, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK.

Ng, A. Y.; Harada, D.; and Russell, S. 1999. Policy invariance under reward transformations: theory and application to reward shaping. In *Proc. 16th International Conf. on Machine Learning*, 278–287. Morgan Kaufmann, San Francisco, CA.

Olsson, L. A.; Nehaniv, C. L.; and Polani, D. 2006. From unknown sensors and actuators to actions grounded in sensorimotor perceptions. *Connection Science* 18(2):121–144.

Parr, R., and Russel, S. 1997. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 9*.

Pierce, D. M., and Kuipers, B. J. 1997. Map learning with uninterpreted sensors and effectors. *Artificial Intelligence* 92:169–227.

Pollack, J. B., and Blair, A. D. 1997. Why did TD-gammon work? In Mozer, M. C.; Jordan, M. I.; and Petsche, T., eds., *Advances in Neural Information Processing Systems*, volume 9, 10. The MIT Press.

Precup, D. 2000. *Temporal abstraction in reinforcement learning*. Ph.D. Dissertation, The University of Massachusetts at Amherst.

Provost, J.; Beeson, P.; and Kuipers, B. J. 2001. Toward learning the causal layer of the spatial semantic hierarchy using SOMs. AAAI Spring Symposium Workshop on Learning Grounded Representations.

Provost, J.; Kuipers, B. J.; and Miikkulainen, R. 2006. Developing navigation behavior through self-organizing distinctive-state abstraction. *Connection Science* 18.2.

Ring, M. B. 1994. *Continual Learning in Reinforcement Environments*. Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712.

Ring, M. B. 1997. Child: A first step towards continual learning. *Machine Learning* 28(1):77–104.

Roy, N., and Thrun, S. 1999. Online self calibration for mobile robots. In *IEEE International Conference on Robotics and Automation*.

Rubner, Y.; Tomasi, C.; and Guibas, L. J. 2000. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision* 40(2):99–121.

Segal, E.; Pe'er, D.; Regev, A.; Koller, D.; and Friedman, N. 2005. Learning module networks. *Journal of Machine Learning Research* 6:557–588.

Shani, G. 2004. A survey of model-based and model-free methods for resolving perceptual aliasing. Technical Report 05-02, Department of Computer Science at the Ben-Gurion University in the Negev.

Şimşek, Ö., and Barto, A. G. 2004. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 751–758. ACM Press.

Şimşek, Ö.; Wolfe, A. P.; and Barto, A. G. 2005. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the Twenty-Second International Conference on Machine Learning*.

Singh, S.; Littman, M. L.; Jong, N. K.; Pardoe, D.; and Stone, P. 2003. Learning predictive state representations. In *The Twentieth International Conference on Machine Learning (ICML-2003)*.

Singh, S.; James, M. R.; and Rudary, M. R. 2004. Predictive state representations: an new theory for modeling dynamical systems. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*.

Smart, W. D., and Kaelbling, L. P. 2002. Effective reinforcement learning for mobile robots. In *Proceedings of the International Conference on Robotics and Automation*.

Smith, A. J. 2002. Applications of the self-organizing map to reinforcement learning. *Neural Networks* 15:1107–1124.

Stanley, K. O. 2003. *Efficient Evolution of Neural Networks Through Complexification*. Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, Austin, TX.

Stone, P., and Sutton, R. S. 2001. Scaling reinforcement learning toward roboCup soccer. In *Proceedings of the Eighteenth International Conference on Machine Learning*.

Stone, P., and Veloso, M. 2000. Layered learning. In *Eleventh European Conference on Machine Learning (ECML-2000)*.

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

Sutton, R. S., and Tanner, B. 2005. Temporal-difference networks. In *Advances in Neural Information Processing Systems*, volume 17, 1377–1384.

Sutton, R. S.; Precup, D.; and Singh, S. 1998. Intra-option learning about temporally abstract actions. In *Proceedings of the Fifteenth International Conference on Machine Learning(ICML'98)*, 556–564. Morgan Kaufmann.

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

Sutton, R. 2001. What's wrong with artificial intelligence. http://www-anw.cs.umass.edu/r̃ich/IncIdeas/WrongWithAI.html.

Tesauro, G. J. 1995. Temporal difference learning and TD-gammon. *Communications of the ACM* 38:58–68.

Teyssier, M., and Koller, D. 2005. Ordering-based search: A simple and effective algorithm for learning bayesian networks. In *Proceedings of the Twenty-first Conference on Uncertainty in AI (UAI)*, 584–590.

Thrun, S.; Burgard, W.; and Fox, D. 2005. *Probabilistic Robotics*. Cambridge, MA: MIT Press.

Toussaint, M. 2004. Learning a world model and planning with a self-organizing, dynamic neural system. In *Advances in Neural Information Processing Systems 16*.

Triesch, J. 2005. A gradient rule for the plasticity of a neuron's intrinsic excitability. *Artificial Neural Networks: Biological Inspirations - ICANN 2005* 65–70.

Turrigiano, G. G. 1999. Homeostatic plasticity in neuronal networks: The more things change, the more they stay the same. *Trends in Neurosciences* 22(5):221–227.

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

Wolfe, B.; James, M. R.; and Singh, S. 2005. Learning predictive state representations in dynamical systems without reset. In *Proceedings of the 22nd International Conference on Machine Learning*.

Xing, E. P.; Ng, A. Y.; Jordan, M. I.; and Russel, S. 2003. Distance metric learning with applications to clustering with side information. In *Advances in Neural Information Processing Systems (NIPS)*.

Zimmer, U. R. 1996. Robust world-modelling and navigation in a real world. *Neurocomputing* 13(2-4):247–260.

Vita

Jefferson Provost was born in Mt. Lebanon, Pennsylvania in 1968. He graduated from Mt. Lebanon High School in 1986, and received his B.S. in Computer Science from the University of Pittsburgh in 1990. From then until entering graduate school in 1998, he worked writing software to help psychologists design and run experiments, First, at the Carnegie Mellon University Psychology Department, he helped design and implement the PsyScope experiment design and control system(MacWhinney, Cohen, & Provost, 1997; Cohen *et al.*, 1993). Later, at Psychology Software Tools, Inc., he was part of the team that designed and implemented the E-Prime experiment system. More recently, he was a key member of the team that designed and developed the Topographica cortical simulator(Bednar *et al.*, 2004).

Permanent Address: 3257 Eastmont Ave
Pittsburgh, PA 15216
`jp@cs.utexas.edu`
`http://www.cs.utexas.edu/users/jp/`

This dissertation was typeset with LaTeX $2_\varepsilon$by the author.