LEARNING THE STRUCTURE OF ACTIVITIES FOR A MOBILE ROBOT

A Dissertation Presented

by

MATTHEW D. SCHMILL

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2002

Computer Science

LEARNING THE STRUCTURE OF ACTIVITIES FOR A MOBILE ROBOT

A Dissertation Presented

by

MATTHEW D. SCHMILL

Approved as to style and content by:

for my mother

ACKNOWLEDGMENTS

Aristotle once said, “education is the best provision for the journey to old age.” In my years as a student at UMass, I have had ample opportunity to stock up on provisions. Though my journey does not end with the writing of this dissertation, I have many fine people to thank for making this possible this accomplishment.

Foremost among them is my advisor Paul Cohen. Had it not been for his audacity to suggest that I had a dissertation in me, there would be no dissertation today. I cannot imagine a graduate career under any other advisor. I thank him for the years of support and encouragement that have gone into this document.

I could not have asked for a better dissertation committe. My deepest appreciation goes to Paul Cohen, Neil Berthier, Rod Grupen, and Victor Lesser for their advice, patience and insight.

This dissertation was also helped immeasurably by my many friends and colleagues at UMass. I would like to thank David Hart for welcoming me into the Experimental Knowledge Systems Laboratory as an undergraduate. I thank David Westbrook for showing me the Way of Lisp, his tireless support, and his keen insights into softball theory over the years. To Tim Oates I owe a great debt of gratitute; much of his work would provide a foundation on which I would define my own thesis, not to mention the LaTeX jams Tim has bailed me out of. My work with David Jensen, a fine scholar and scientist, proved both rewarding and pivotal to this dissertation as well. Gary King, Brent Heeringa, and other members of EKSL past and current, too numerous to list, have all proven invaluable to my work in one way or another.

Finally, I would like to acknowledge the fine contributions of my friends and family. It is only with their patience and support I was able to accomplish what I have.

ABSTRACT

LEARNING THE STRUCTURE OF ACTIVITIES FOR A MOBILE ROBOT

MAY 2002

MATTHEW D. SCHMILL

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Paul R. Cohen

At birth, the human infant has only a very rudimentary perceptual system and similarly rudimentary control over its musculature. As time goes on, a child *develops*. Its ability to control, perceive, and predict its own behavior improves as it interacts with its environment. We are interested in the process of development, in particular with respect to activity. How might an intelligent agent of our own design learn to represent and organize procedural knowledge so that over time it becomes more competent at its achieving goals in its own environment? In this dissertation, we present a system that allows an agent to learn models of activity and its environment and then use those models to create units of behavior of increasing sophistication for the purpose of achieving its own internally-generated goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

We are interested in understanding the nature of activity in intelligent agents. How does an infant, starting with very modest sensory and motor apparati, develop activities that allow it to reliably achieve its goals? How might robots, or computer-based agents, do the same? This dissertation describes a system architecture designed to model the development of activity in robotic or simulated agents.

Our interest in the development of activity is twofold. A system that can develop activity has many practical applications; agents that can learn and adapt their behaviors are more robust and flexible than those that cannot in all but the most trivial applications. Real-world domains are too complex for an agent designer to anticipate every contingency that might come up. From industrial assembly tasks to scientific exploration, adaptation provides a distinct practical advantage over hard-wired behaviors.

A second motivation for this dissertation is that we believe that the development of activity is fundamental to the development of all types of conceptual knowledge. A system that allows an agent to develop activity is therefore central to artificial intelligence, and can shed light on the nature of human intelligence.

Our activity-centric perspective on human intelligence is inspired by the interactionist philosophy of Lakoff and Johnson [47] and Piaget's theories of early development [30, 25]. The central themes of interactionist philosophy, as well as Piaget's theory (certainly the first stages), are that little native structure is necessary to support learning and development, and that the bases of conceptual knowledge are bootstrapped from simple sensorimotor experience between an actor and its environment. In Piaget's words[1]:

> *The aquisition of knowledge is a gradual developmental process made possible through the interaction of the child with the environment.*

To this end, activities, and the ability of an intelligent agent to organize its behavior into units that allow it to achieve goals, are of paramount importance, and basic to the development of all conceptual knowledge. We can identify four basic conceptual structures that are hallmarks of human intelligence. *Activities* encode knowledge about how the primitive motor capabilities of an agent can be combined to produce complex behaviors and achieve the agent's goals. *Classes* are extensional structures that relate the features and objects of the environment to the types of activities they afford; empty soda cans and tennis balls belong to the class of *objects I have grasped*. *Concepts* are the intensional versions of classes. They encode the necessary conditions of class membership, for example, what is required for an object to be *graspable*; perhaps objects with a diameter less than 4 inches. Finally, activity, classes, and concepts serve as the foundations for creating associations between spoken or written words and knowledge grounded in the environment. These associations form the basis of *language*.

A simple understanding of how these four types of knowledge emerge is pictured in figure 1.1. This "waterfall" model of development may be interpreted as follows. In the beginning of the process, an agent bootstraps conceptual development by exercising its primitive motor skills and building models of the results. The agent essentially learns what its actions do and how the environment mediates the effects of its actions. As its models improve, the agent begins to organize its motor skills into activities of increasing complexity.

[1]G.Lefrancois. (1995) Theories of Human Learning (3rd ed.) Brooks; Cole

As activities grow in complexity, opportunities for objects in the environment to play significant roles in determining the outcome of these activities become more frequent. Suppose, for example, the agent has pieced together an exploratory activity to locate, grasp, and lift an object. If a heavy stone is chosen to fill the "object to be lifted" role of the lift portion of the behavior, the activity will unfold differently than if an empty soda can is chosen. In the former case, the agent will fail to lift the heavy object. Perhaps it will strain itself and feel pain. In the latter case, the agent will lift the empty can without difficulty. Experiences like these allow the agent to build the beginnings of classes. The heavy stone belongs to the class we might call "objects that hurt to lift", while the soda can falls into a class we might label "objects that have been lifted".

As classes grow in size, it becomes possible to abstract out the features of objects that determine class membership. This allows the agent to generate intensional *concepts*, which provide the agent the capability to reason about how its activities will unfold in novel situations. Our reaching and grasping agent might build up a concept that could be labeled "heavy" or "easy to lift".

Further experience with activities, classes and concepts in the presence of spoken word facilitates the acquisition of language. As our agent attempts to lift the stone, an observer might utter the word "heavy", allowing the agent to make the connections between the spoken word and active conceptual structures like its the concept of a heavy object. Word meanings form the basis for natural language understanding, and language, in turn, allows the developmental process to short-circuited. Later activities and concepts may be grounded in language instead of experience.

The waterfall model, and Piaget's words, underscore a fundamental difference between what we will call *development* and *learning*. Learning is the process of acquiring and refining a particular structure: how to grasp an object, or what it means to be graspable. Development is the incremental process of an agent posing,

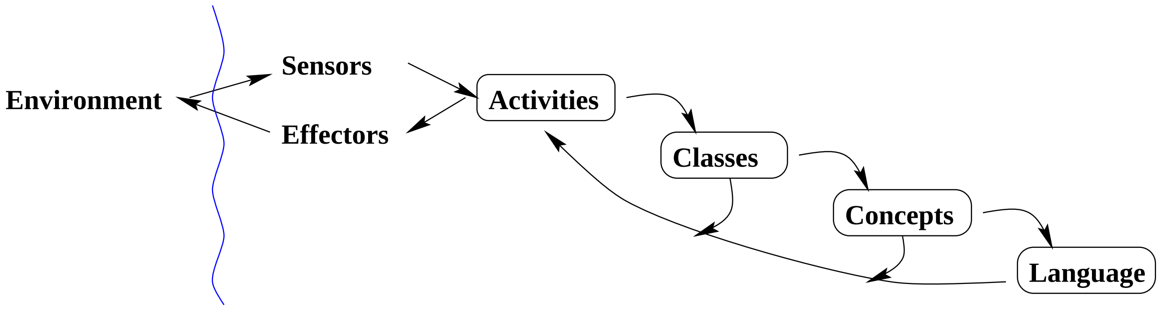

Figure 1.1. A waterfall model of the development of conceptual structure.

solving, and refining a multitude of cooperative learning tasks. It is a gradual process in which learning tasks interact; gains made in one learning problem can carry over to other learning processes. For instance, learning and optimizing the grasp activity enables an agent to produce and refine a *throw* activity. The template architecture for any developmental process, including activity learning, is pictured in figure 1.2.

This dissertation documents our interest in development, specifically, the development of activity, the pivotal first step of the waterfall model. The remainder of this dissertation will be devoted to filling in the boxes of figure 1.2 in such a way that an agent using this model will develop activities of increasing sophistication and utility. Furthermore,

We attempt to make as few nativist concessions (innate endowments) as is practically possible.

Filling out the boxes should support downstream conceptual learning of classes, concepts, and language.

The basic unit of currency that the developing has to work with is the *experience* – a sensorimotor representation of an interaction with its environment.

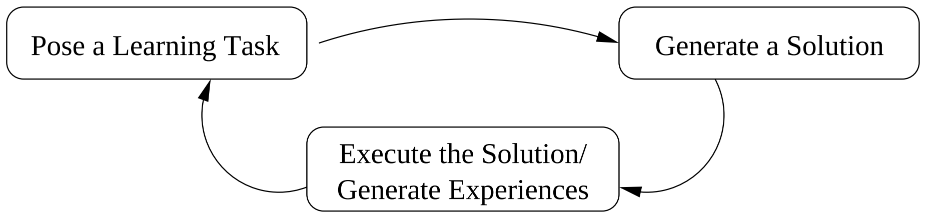

Figure 1.2. The basic architecture of a developmental process.

1.1 Sensorimotor Agents and Experiences

Before filling in the boxes of figure 1.2, it is important to establish what we mean by a *sensorimotor agent* as well as what an *experience* is. Simply, sensorimotor agents are agents with access only to raw sensory information and motor function. Sensorimotor agents have not built abstractions on their sensors, attached meaning to them, nor have they built composite activities from their raw motor function or even discovered what their effectors do. Sensorimotor agents are endowed with the ability to access sensor values and activate motor controls, and nothing more. We define a sensorimotor experience as time series of sensor readings, polled at some frequency, during a span of time where an agent is engaged in intentional action. Different types of sensorimotor agents experience their environment in different ways.

The work described in this dissertation is built on the premise that enivornments that are rich with structure offer a sensorimotor agent more opportunities to learn increasingly sophisticated activities. As the number of possible sensations, interactions, and activities goes up, though, the learning problem may become more difficult to manage. Less complex environments, while limiting the number and variety of activities to be learned, offer the advantage of simpler learning tasks. Platforms for learning exist on a spectrum that can be alternately viewed in terms of the difficulty of the developmental undertaking and the richness of the potential results.

At one end of the spectrum are simple domains. Domains such as the gridworld (used as an evaluation domain in many early reinforcemnt learning publications) racetrack domain (introduced by Barto *et al* in [2]) provide an agent with a small number of discrete-valued sensors, complete information about the sensor (state) space, and a small number of discrete effectors. These domains neither afford nor demand terribly complex activities. However, learning in these domains is generally tractable, and their value lies in proof of concept; they show that a particular approach is capable of learning on a limited scale, and whether scaling up is worth pursuing at all.

At the other end of the spectrum are complex domains such as robots interacting in cluttered environments. Their sensory input is high-dimensional and continuously-valued. Their effector space is multidimensional, continuous, and effectors have temporal extent. These environments are for all intents and purposes probabilistic and partially observable. While knowledge generated by learning in these real-world environments is clearly apt to be rich and useful, the sheer volume of sensory data that such a domain offers is often overwhelming for a learning algorithm. Many learning algorithms simply have no answer for the unique challenges that real-world domains can present.

The approach we have taken is to design our system to address the demands of a complex environment, implement it at the simple side of the spectrum, and scale it up to increasingly complex environments. In this section, we describe domains of varied sophistication that will be used as examples and testbeds throughout the dissertation: one simulated domain and two real-world, robotic platforms.

1.1.1 GridSim

The simplest domain we consider is based on the grid-world domain that is often used as a test bed for machine learning systems. The archetypical grid world domain consists of an agent, an $m \times n$ grid, and possibly objects placed on the grid. In its simplest incarnation, grid cells are either unoccupied, occupied by a wall, or occupied by a goal indicator. The agent can move from its current cell to any adjacent cell

vision	n-cam-shape	n-cam-color	n-cam-dist
	e-cam-shape	e-cam-color	e-cam-dist
	s-cam-shape	s-cam-color	s-cam-dist
	w-cam-shape	w-cam-color	w-cam-dist
	ccs-shape	ccs-color	
bay	bay-color	bay-shape	
drives	pain	reward	

Table 1.1. The primitive sensor suite for GridSim agents.

as long as the cell is not occupied by a wall. In typical implementations of the grid world, the effects of actions are discrete and instantaneous, and the agent can sense its location on the grid in cartesian coordinates. In many instances, the task given to grid-world agents is to find and move onto a goal cell.

The simplicity of this domain presents a tractable set of problems to learning systems by limiting the kinds of activities that are possible and limiting the complicating factors present in real-world domains like noise, temporal issues, and so on. Many learning systems designed for such domains are not guaranteed to scale up, and frequently are ill-equipped to cope with more complex environments. In order to add richness to this environment, and to introduce some real-world complexities (we would like our system to scale to real-world domains), we have extended the basic grid world simulation in the following ways:

- The agent has sensors rather than access to its location on the grid. Sensors return properties of the agent's surrounding environment, and are modeled after sensors that might be found on a real-world robot. A list of the sensor suite, which includes mostly visual sensors (camera sensors), we use is shown in table 1.1.

- Sensor information is dynamic and sensors may be discrete or real-valued. Sensor values are updated every 100μsec. Sensor values change continuously over the duration of an activity.

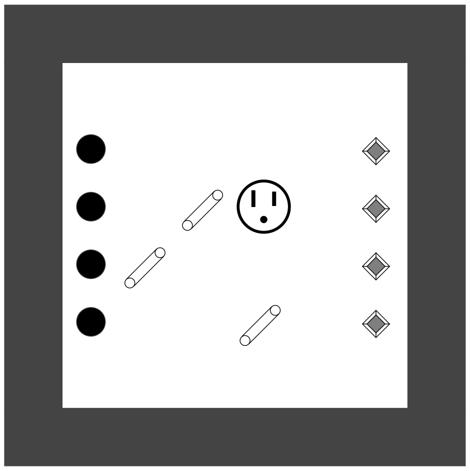

Figure 1.3. A rendering of the GridSim simulator.

- Grid cells can contain objects other than walls. In our simulation, there are debris objects, with which an agent can share a cell, as well as specially marked cells which have various roles in interactions between the robot and environment. Different types of objects and marked cells are sensed by the agent's visual sensors, which report their shape, color, and distance.
- The agent has two actions at its disposal in addition to movement. The agent can still move from cell to cell in the cardinal directions, as in the original grid world, but can also lift and drop debris.

A sample snapshot of a gridworld simulation is shown in figure 1.3. This 8×8 grid is surrounded by a wall and contains several pieces of debris (rendered as diagonally-oriented rods) as well as visible dropoff locations where debris may be deposited (rendered as circles). The agent is rendered as face slightly off the center of the grid. Walls, debris, and dropoff cells are all visible to the agent. Each of the agent's visual sensors will report a color and shape code for any visible object, and each of the directional sensors will report a distance to the nearest visible object. Along the east wall of the grid are cells with a unique shape and color (rendered as diamonds) whose function is user-definable.

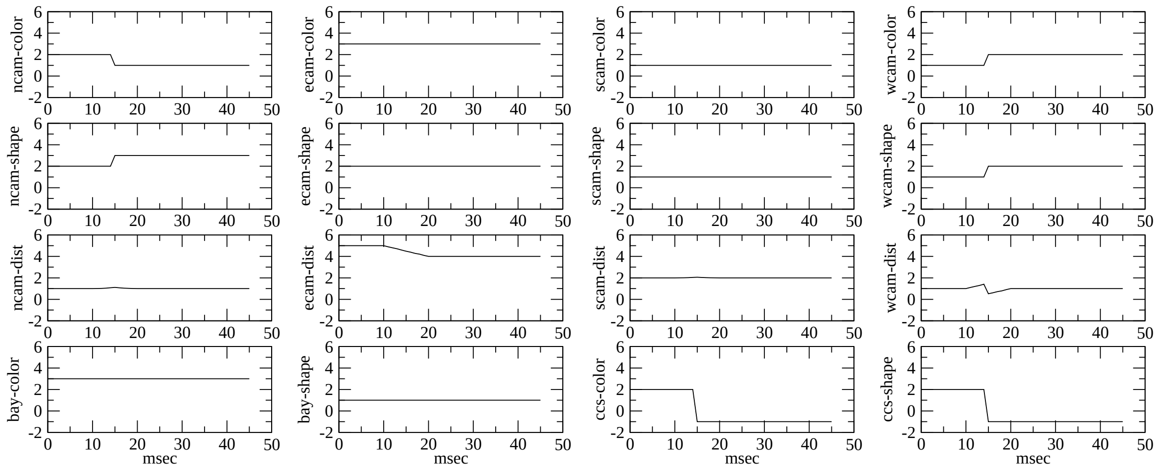

Figure 1.4. A GridSim experience: moving east.

Many types of agents could be placed in the GridSim simulator, but we work with one primary agent, which we will call the NESW agent. This agent has 6 effectors: move-n, move-e, move-s, move-w, lift, and drop. The first four attempt to move the agent one cell to the north, south, east, and west, respectively. The latter two attempt to lift any debris in the current cell into the agent's cargo bay or drop those contents, respectively. This agent has the sensors listed in table 1.1: 4 visual sensors, each of which senses the color, shape, and distance to the nearest object in each of the four cardinal directions. It can also sense the color and shape of any object it shares a cell with (via a sensor called the current cell sensor, or ccs) as well as what it holds in its cargo bay (via the bay sensor). Color and shape sensors take integer-encoded values; the color red might be encoded as the digit "1", for example.

Our simulated agents also have two additional sensors: a reward sensor and a pain sensor. The pain and reward sensors can be used to simulate negative and positive reinforcement in these environments. Pain, by default, is incurred when the agent moves into the walls surrounding the grid. Both pain and reward can be administered at the discretion of the experiment designer. An example use of the reward sensor would be to administer reward for dropping debris into the dropoff cells.

An example experience of the NESW agent moving in the GridSim domain is shown in figure 1.3. Each plot is a time series of sensor readings, one sample taken per millisecond, for each of the primary sensors of the agent[2]. Note in this experience, the visual sensors change as the agent moves away from an object in its current cell: the current cell sensor goes low indicating that the experience started with a visible object in the current cell and ended without one. The east camera sensor shows a decrease in distance, indicating it is approaching an object. The north facing camera's shape and color sensors change over the course of the activity, indicating a new object comes into view as the agent moves east. The west camera shows changes in shape, color, and distance, which correspond to the object that was previously in the current cell sensor being picked up by the west facing camera as the agent moves away from it. Finally, the bay sensor and south facing sensors show no change over the course of the experience. Experiences like this one, represented as sensor time series, form the basis of development in this dissertation.

The 8×8 grid serves as a baseline in our analyses. Larger grids, with greater variation in the types of objects the agent interacts with, resulting in new and richer experiences, are possible and we consider these as we assess the scalability of our system.

1.1.2 Pioneer-1 mobile robot

The ultimate goal of this research is to implement our theory on a real robot operating in the real world. The Pioneer-1 is a small (1.5 square foot) mobile robot pictured in figure 1.5. The Pioneer-1 has two independently controllable drive wheels as well as a gripper module for picking up objects, affording the robot the opportunity to take 4 primitive activities: CLOSE-GRIPPER (close and raise the gripper), OPEN-

[2]The GridSim sampling rate is user-definable. We use a rate of 10Hz here because this is the sampling rate of our real-world platform, the Pioneer mobile robot.

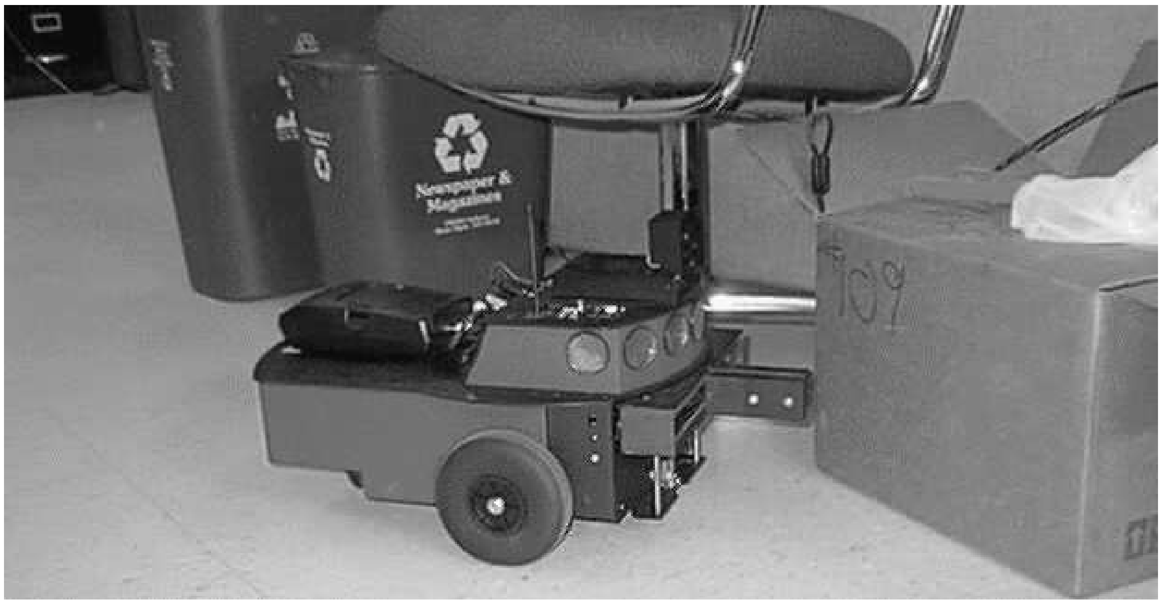
Figure 1.5. The Pioneer-1 mobile robot.

GRIPPER (lower and open the gripper), MOVE (forward or backward in a straight line), and TURN (in place, clockwise or counter-clockwise). These actions may be executed simultaneously (with the exception of CLOSE-GRIPPER and OPEN-GRIPPER). The MOVE and TURN actions, can be activated to achieve and maintain rates of $-500 \ldots 500\frac{mm}{sec}$ and $-120 \ldots 120\frac{deg}{sec}$ respectively.

The Pioneer-1 has 36 primitive sensors that can be thought of as falling into 4 different modalities. Its Cognachrome vision system is capable of tracking 3 objects of different colors at 60Hz. The 3 channels for color blob tracking are labeled A, B, and C, with each channel reporting the X and Y location of the dominant (largest area) blob in the visual field, as well as the blob's width, height, and area. We also approximate the blob's distance from the Pioneer-1, yielding a total of six attributes per blob being observed. Seven sonars provide a second modality of sensing, each with a range of five meters, with five of the seven sonars pointing roughly forward and one each pointing 90 and -90 degrees to the side. A third modality is derived from wheel encoders, which provide feedback about rotational, translational, and individual wheel velocities as well as binary sensors for detecting when the robot is

sonar	sonar-0	sonar-1	sonar-2	sonar-3	
	sonar-4	sonar-5	sonar-6		
motion	r-wheel-vel	l-wheel-vel	trans-vel	rot-vel	
	r-stall	l-stall	robot-status		
gripper	grip-state	grip-front-beam	grip-rear-beam	grip-bumper	
vision	vis-a-area	vis-a-x	vis-a-y	vis-a-h	vis-a-w
	vis-b-area	vis-b-x	vis-b-y	vis-b-h	vis-b-w
	vis-c-area	vis-c-x	vis-c-y	vis-c-h	vis-c-w
	vis-a-dist	vis-b-dist	vis-c-dist		

Table 1.2. The Pioneer-1's 36 sensors.

stalled or moving. Finally, the gripper module provides 5 discrete-valued sensors: one bump sensor, two break beams for detecting objects within the gripper's paddles, and one gripper state variable. A list of the Pioneer-1's sensors is shown in table 1.2.

1.1.3 Pioneer-2 mobile robot

The Pioneer-2 mobile robot, pictured in figure 1.6, is RWI's next-generation Pioneer robot. The Pioneer-2 improves on the the Pioneer-1, most notably in that its vision system is much more robust and more capably distinguishes objects of different color. The Pioneer-2 also has 7 rear-mounted sonars as well as a ring of bumpers mounted on the rear of the robot. These sensors provide additional sensory information for the rear of the robot, which the camera cannot cover. The additional sensor information, as well as the improved vision system make the Pioneer-2 a superior platform to the Pioneer-1. As such, the references to the "Pioneer" platform throughout this document will implicitly indicate the Pioneer-2, whereas the Pioneer-1 will indicate the older platform.

The full suite of sensors available to the Pioneer-2 are listed in table 1.3. They are essentially the same as the Pioneer-1, with the additional ring of rear-facing sensors, the panel of rear-facing bump sensors. The gripper bump sensor from the Pioneer-1 has been eliminated from the Pioneer-2 design.

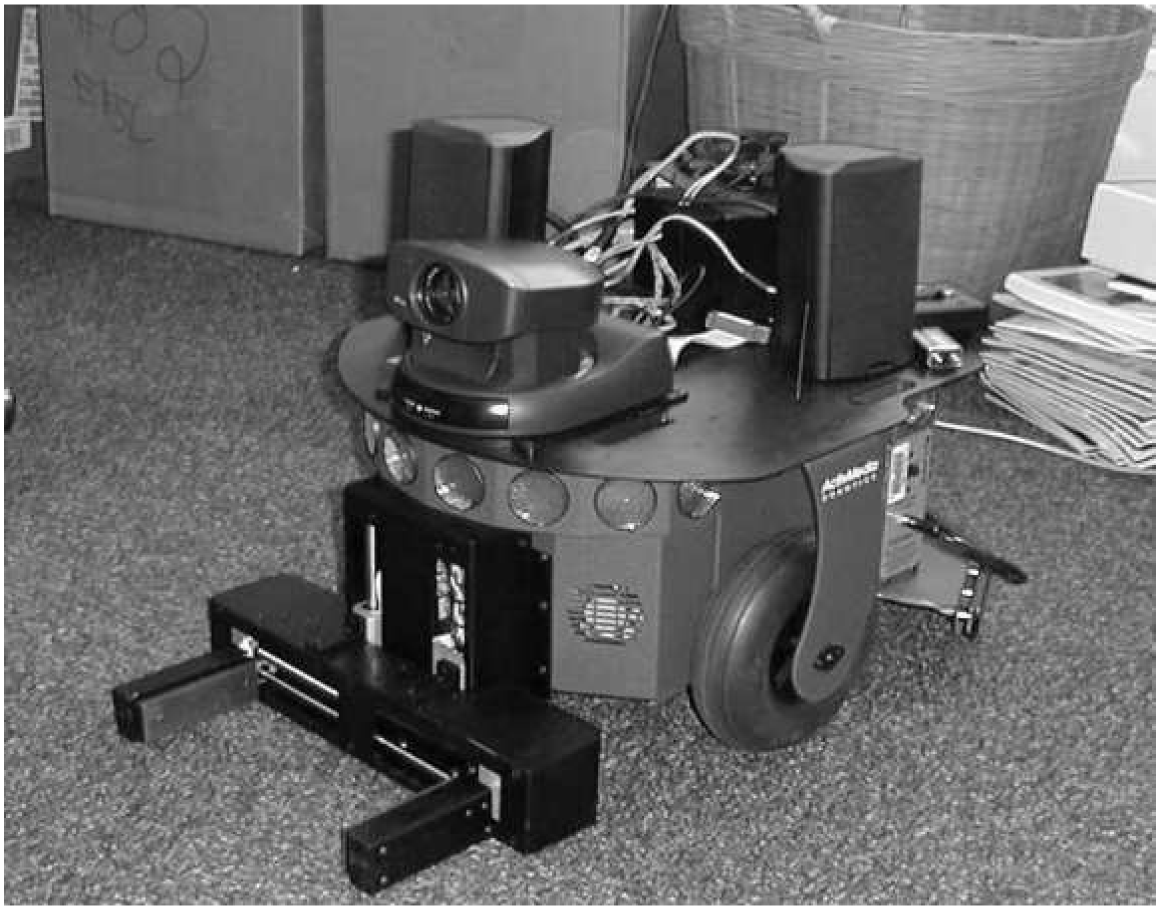

Figure 1.6. The Pioneer-2 mobile robot.

sonar	front-sonar-0	front-sonar-1	front-sonar-2	front-sonar-3	
	front-sonar-4	front-sonar-5	front-sonar-6		
	rear-sonar-0	rear-sonar-1	rear-sonar-2	rear-sonar-3	
	rear-sonar-4	rear-sonar-5	rear-sonar-6		
motion	r-wheel-vel	l-wheel-vel	trans-vel	rot-vel	
	r-stall	l-stall	robot-status		
	rear-bumper-0	rear-bumper-1	rear-bumper-2	rear-bumper-3	rear-bumper-4
gripper	grip-state	grip-front-beam	grip-rear-beam		
vision	vis-a-area	vis-a-x	vis-a-y	vis-a-h	vis-a-w
	vis-b-area	vis-b-x	vis-b-y	vis-b-h	vis-b-w
	vis-c-area	vis-c-x	vis-c-y	vis-c-h	vis-c-w
	vis-a-dist	vis-b-dist	vis-c-dist		

Table 1.3. The Pioneer-2's 47 sensors.

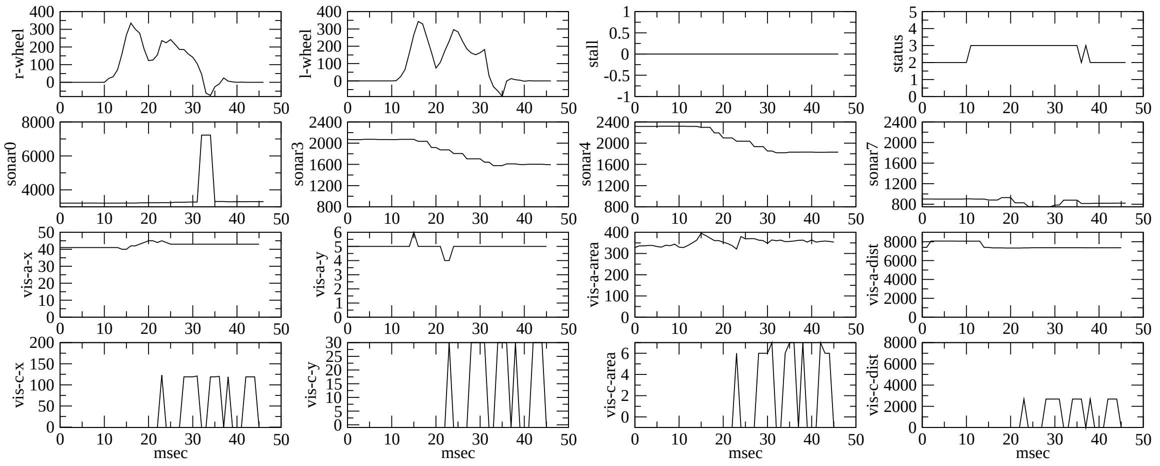

Figure 1.7. A Pioneer-2 experience: moving forward at 250mm/sec.

An example experience of the Pioneer-2 robot moving forward at 250 mm/sec is shown in figure 1.7. The difference in granularity, noise, and complexity from the GridSim experience should be immediately clear. The wheel velocity sensors show some oscillation during which the Pioneer's movement is in reality somewhat jerky. Transitions in the sonar sensors as the Pioneer approaches an object are not smooth and monotonic as in GridSim. Finally, channel C of the vision sensor is simply noise. While a naive agent might assume that there is structure in the noise, a developing agent must learn to differentiate true structure from noise, and account for it as necessary. Real-world platforms offer significant challenges to learning (and thus developmental) systems. Still, the data on which development is based are the same: experiences in which the agent is engaged in activity, represented as time series sensor readings. Throughout the dissertation, we will refer back to the Pioneer-2 as a testbed and ultimate platform of choice for this work.

1.2 An Example Activity

Internally, activities may be represented in a number of ways: they may be specified as simple action sequences, as rule-based systems, or scripts similar to computer programs. But, what defines an activity; what distinguishes one activitiy from another? What does an activity *look like*?

It is safe to say that experiences define an activity. We can be more specific with an example. Consider an agent operating in the GridSim domain, as configured in figure 1.3. Suppose the agent were to execute the sequence move-w, lift, move-w, move-w, drop. Each action, when executed, will produce an individual experience with characteristic sensory behavior. Based on the sensory behavior, one might label the individual experiences resulting from this action sequence, starting in the specified state, as follows:

1. move over debris
2. lift debris
3. approach receptacle
4. move over receptacle
5. drop debris into receptacle

Now suppose that the agent were to execute the same action sequence starting in the grid cell directly to the south of where it is in figure 1.3. We might label the individual experiences as follows:

1. approach debris
2. lift nothing
3. move over debris
4. move off of debris, on to receptacle
5. drop nothing

Because the context when each action in the sequence is different, the outcomes of each of action is also different. The lift action of step 2 fails to lift anything because it has not reached the debris yet. Clearly, the two sequences do not correspond to the same activity even though the sequence of actions is the same. The effects the actions have on the domain changes completely. Next, suppose the agent were to start one cell to the south of where it is in figure 1.3, and it were to execute the sequence move-w, move-w, lift, move-w, drop. We might label these experiences as follows:

1. apprach debris
2. move over debris
3. lift debris
4. move over receptacle
5. drop debris into receptacle

We would likely put this activity in the same class of activity as the first sequence, but again, not the second. This is because we perceive and classify activities by their changes on the domain, and in particular, how they end. This is in no small way due to our perception that in the simplest case, behavior is motivated, and that a particular unit of activity ends in the goal. The first and third sample activities end with a qualitatively similar experience; the perceived goal in each case was to drop debris into a receptacle. The second activity, though it consisted of the same action sequence as the first, ended differently, and thus our interpretation of the behavior, is wholly different. Let us define an activity as *a specification of behavior that ends in some desired experience.*

An activity we might expect a Pioneer robot to develop would be *acquire*. In the acquire activity, the Pioneer first *locates* an object by rotating, then *positions* itself

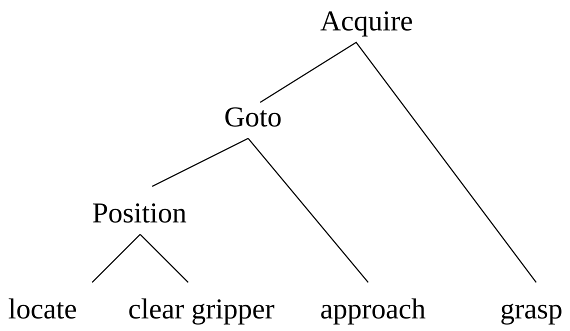

Figure 1.8. A sample hierarchy of activity. At the root is the `acquire` activity which finds and grasps an object.

such that the object is between its gripper paddles by moving up to the object, and then closes and raises the gripper, thus *grasping* the object.

Alternatively, the acquire activity could be described by a sequence of two subactivities: *goto* and *grasp*. The difference between this specification and the previous description is that here, the sequence of moves and turns leading up to the grasp experience is encapsulated into the more abstract operator called *goto*. Goto may in turn be broken down into a simple sequence of 2 subactivities (*position* and MOVE), and so on, as shown in figure 1.8. There are two major advantages of this hierarchical representation over a flat representation in which each activity is described at the level of primitive actions:

- It is computationally less expensive to search through a space of activity sequences if the search can be constrained to shorter sequences. To achieve more complex goals under such constraints, individual units of activities must be able to do more.

- The creation of intermediating units like *goto* increases economy where reuse is possible. Goto, for example, would be a useful subactivity for the *push* activity as well as *acquire*.

Figure 1.8 shows one possible hierarchy for describing the makeup of an acquire activity. It is important to note that the activity labels in this hierarchy are all post-hoc and included for readability, and our benefit only; the robot does not necessarily know that it has learned an activity called "acquire", or that the robot should be constrained to generate this particular hierarchy for an activity that results in the same outcome.

We contend that hierarchical activities like the one shown in figure 1.8 are created by building upward from the leaves (which are primitive actions), one level at a time, by simple composition during development. An activity called *foveate*, which keeps an object centered in the visual field, is composed using only TURN actions. The ability to foveate on objects then allows for more complicated activities to be generated, such as the position activity, which prepares the agent to acquire an object by opening its gripper and foveating on an object. The goto activity then builds on positioning, by moving forward and collecting the object between its gripper paddles. Finally, the simple combination of goto followed by CLOSE-GRIPPER completes the acquire activity.

This incremental building of skills allows the composition process at each node in the hierarchy to be a simple combination of a manageable number of subactivities in sequence (not necessarily always 2, as in this example). As the agent accumulates activities, it can do more in a single activity, and thus, simple composition can achieve more, never subjecting the agent to giant (or intractable) search spaces.

1.3 Overview

Having established some experimental platforms, the basic building blocks of development (experiences), and the goals of development (activities), we now turn to an overview of our system. Recall the blueprint for a developmental process as pictured in figure 1.2. Development is an ongoing, incremental process in which an agent poses a learning problem, generates a solution, and executes the solution. The solution provides feedback that the agent can use to refine the solution, should it decide to continue with the same learning task or pick it up at a later time. This blueprint works with any of the conceptual constructs from the overview of development in section 1: the development of language, classes, concepts, and activities can all fit this generate-and-test architecture. What and how the agent develops depends on how the individual boxes are implemented.

Figure 1.9 shows our system architecture for the development of activity. In this picture, we have replaced generic processes of figure 1.2 with more specific descriptions. What follows is a short overview of our approach.

In large part, how we fill in the box labeled *generate a solution* in the diagram of figure 1.2 determines what the rest of the system will look like. The inner workings of this box create activities that serve the goals of the developing agent, and each of the other components in our system will be tailored to work in tandem with the central component that generates activities for the agent to engage in. There are many available approaches to choose from: reinforcement learning, artificial neural networks, genetic algorithms and planning among them. Which system we choose will dictate how our system specifies learning tasks. A system based on reinforcement learning would solve tasks specified as reward functions, for example. A genetic programming approach would solve tasks defined by a fitness function. These systems would then determine a control policy and pass it on to the execution component, which would generate experiences to learn from, and the cycle repeats.

Our system is based on planning. Planning is an AI technology that has been well-studied for many years, dating back to 1963, when Newell and Simon introduced the General Problem Solver (GPS) [59]. The General Problem Solver explicitly reasons about actions, their effects, and how actions can be sequenced to change an agent's current state to a goal state. Over the years, GPS has spawned many lines of research. What these approaches generally share is that they reason about actions and their effects, and search for sequences of actions that achieve goals. These systems *plan.*

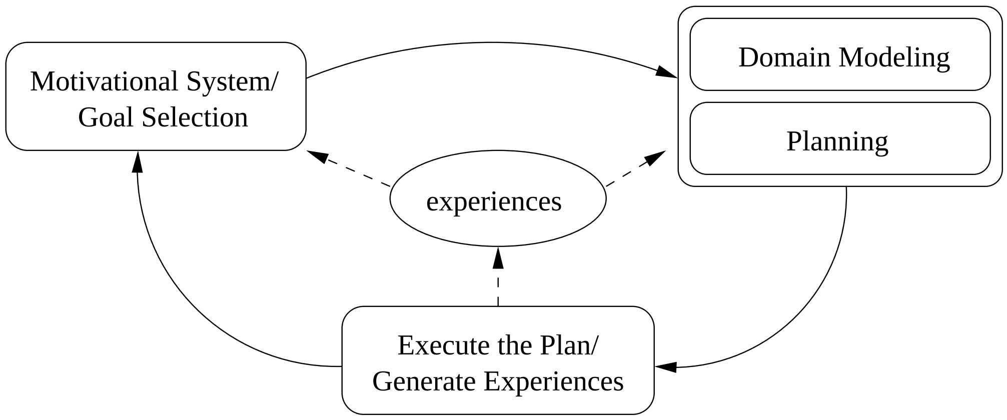

Figure 1.9. Our system architecture for the development of activity.

Our motivation for selecting planning as a basis for development is discussed in detail along with a literature review of approaches to generating behavior, in chapter 3. Not the least of our motives, though, is strong evidence supporting the use of means-ends analysis and planning in human problem solving [60, 1].

If an agent is going to engage in planning, it must have *operator models* that express the effects actions have on the environment. Operator models form the basis for reasoning about whether a candidate plan has a chance of achieving an agent's goals. Thus, as part of development, our system must include a component that learns operator models from experiences. The subsystem labeled *domain modeling* in figure 1.9 does just this. Operator models express a mapping between domain conditions and action outcomes. Our system learns these predictive maps between initial conditions and outcomes for use by the planner. Together, the domain modeling system and our planner produce activities for an agent to execute. The box labeled "execute the solution" handles this step.

The box labeled "pose a learning task" is critical in closing the loop of development. The handful of technologies that offer solutions to learning behavior – reinforcement learning, planning, et al. – do so with respect to a particular task specification. State of the art planning systems can navigate a cluttered hallway and deliver mail to recipients, but they require tasks to be specified up front, usually by the system's designers. Planning systems require goal specifications that correspond to the desired tasks and a perceptual space that supports achieving those goals. Each of these specifications require significant engineering on the part of the developer, and if some part of the required specification is botched, then the system may not succeed. Our interest in development, and closing the loop, requires us to internalize goal generation, and remove system designers from the loop. Our agent must be able to generate its own sensorimotor goals and select among them. Once it has done this, it can hand the goal off to the planner, which generates an activity for the execution module to implement. To this end, we introduce a *motivational system* as part of our system. The motivational system generates goals and decides which one to engage in when the agent has the opportunity to make that decision.

Classical planning defines goals in terms of the agent's state space. A goal is simply a specification of a desired world state. Classical planners generally work in highly abstracted propositional spaces; a goal might be to "have one's car keys", for example. Agents using propositional goals such as these can perceive whether or not a proposition is true easily. It can simply ask, "do I have my car keys?" This is far outside the scope of what a sensorimotor agent can do. Our sensorimotor agents can query sensors and test simple relations such as equality or less-than and greater-than. A Pioneer-2 goal state might be `(vis-a-dist = 0.0)`, or to drive the distance to the visible object in channel A down to 0. If a goal is a sensorimotor state (or a partial state specification), though, goal generation can be problematic in two ways that classical planners do not have to worry about. First, the space of sensorimotor states is multidimensional, and real-valued. The goal space is unbounded. How does an agent choose from an infinite space of goal states? Why choose a goal value of 0.1 for the robot's sonar instead of 0.11? Second, unconstrained sensorimotor goal selection opens the door to unachievable or practically unachievable goals, and thus pathological planner behavior. A sonar reading of 0 may be physically impossible for an agent, for example.

The task of goal generation can be constrained in one very important way by redefining the space of goals an agent might propose. We suggest that at the highest level of control, agents plan to *act* rather than to achieve states. This idea is motivated by Piaget's theories of development, which are centered around the importance of *schemas* (reproducable chunks of activity). Much of a developing child's time, Piaget observed, is spent discovering, and subsequently repeating schemas, which roughly correspond to experiences. A child learns what it learns about itself and its environment in the context of repeating experiences over and over [30]. The theory implies that it is the activity, not the state, that is rewarding, an idea that ties in with our observation of the relationship between activities and goals in section 1.2. It is the act of eating, not the state of having eaten, that is satisfying to an agent. The child who attempts to acquire a ball does so not to simply *have* the ball, but so that it may do something with it. If we are to agree with this idea, then it follows that planning serves to accommodate a goal *activity* rather than to place the agent in some goal *state*. This view is in contrast with most, if not all, contemporary planning systems, but it is one we adopt in our developmental system. We call this treatment of goals *planning to act*.

Once goals can be enumerated, then the box labeled "pose a learning task" boils down to rating each possible goal and selecting the most highly rated. Our motivational system is an extensible framework for doing just this. Any agent is said to be under the influence of various motivational factors: they might include hunger, aversion to pain, and curiosity. The importance of each motivational factor rises and falls according to the state of the agent, and using mathematical models of these factors, we can rate and rank goals, and then simply select the top ranked goal.

1.4 Claims

The goal of this dissertation is to produce a system capable of incrementally developing activities of increasing complexity from simple, sensorimotor interactions with its environment. We have developed a system architecture that allows an agent to simultaneously learn predictive models of its environment and compose activities that allow it to systematically and reliably produce desirable changes in its environment. We introduce a system based on this architecture that employs planning, intelligent data analysis, and inductive learning as bases for the development of activity. The central claim of this dissertation can be restated as follows:

Our system provides an agent with the means to learn plans that it can use to achieve its own goals.

Furthermore, we claim that our architecture, and the algorithms and components we have selected to fill out the architecture, are the reason the central claim is satisfied.

Let us consider in greater detail how we intend to demonstrate that the central claim of the dissertation is satisfied by our system. We start by breaking down the process of development into two subproblems: modeling and activity composition. We suggest the following metrics for judging the productivity of modeling and activity composition, respectively.

- The models learned by the modeling system must be *accurate* in the sense that they allow the agent to predict the outcomes of its actions.
- The system's planning compotent, in conjunction with the modeling system, produces executable plans that *facilitate* the goals of the agent. To facilitate a goal is to take the actions necessary to put the agent into a situation where the predicted outcome of one of its actions is its current goal.

These two concepts work in tandem. The role of the modeling system is to produce a mapping from action-state pairs to action outcomes. An efficient modeling system will, in the process of generating the mapping, produce a working space that is an abstraction of the overall sensory space of the agent. In the working space, large regions of the overall state space will map to a single action outcome. A useful modeling system will produce a mapping that achieves a high level of accuracy. The goals of the system being action outcomes, the planner has no chance of success if it cannot make the simple prediction of whether an action, in the current state, will achieve the goal outcome. In other words, we need not even consider facilitation if accuracy cannot be established. Let us first elaborate on the accuracy claim.

1.4.1 The Accuracy Claim

In any state s, an action a will produce an outcome that can be differentiated from other outcomes by the behavior of the agent's sensors as the outcome unfolds. For a given environment, some outcome o will unfold when action a is taken in state s with probability $p(o|s, a)$. It is useful for a system to be able to predict what the outcome of its actions will be. Furthermore, it is useful to be able to make predictions from any state, whether it has been visited previously or not, and so an agent should incorporate generalization into its predictors; a simple lookup table is inadequate in all but the simplest domains.

We denote a predicted outcome by the symbol $\hat{o}$ and an estimated (or observed) probability of an outcome $\hat{p}(o)$. An *uninformed* agent, when faced with a need to predict the outcome of an action in its current state, will make a prediction $\hat{o}_u$ without utilizing any knowledge of its environment. An uninformed agent might simply predict the most frequently occuring outcome of an action, maximizing the expected value $\hat{p}(o|a)$. An *informed* agent incorporates models of state and the role of the environment in determining outcomes, maximizing the expected value $\hat{p}(o|s, a)$.

Let the true outcome of action, that which is actually experienced by executing a in state s, be denoted by $o(s,a)$.[3] The prediction of an uninformed agent will be denoted $\hat{o}_u(s,a)$ and the prediction of the informed agent $\hat{o}_i(s,a)$. The *winner-take-all disparity* $d(o,\hat{o})$ between an estimate and a true outcome will be 0 if the estimate and the true outcome are an exact match, and 1 if they do not match exactly. A model provides a predictive advantage to an informed agent if the ratio $\frac{d(o(s,a),\hat{o}_i,(s,a))}{d(o(s,a),\hat{o}_u,(s,a))}$ is less than one over some range of state and action pairs. The model is totally *accurate* if, for any state action pair, the following holds:

$$\bar{d}(o(s,a),\hat{o}_i,(s,a)) = 1 - \max_o p(o|s,a) \qquad (1.1)$$

where $\bar{d}$ is the mean disparity over a number of trials, and the equality implies that the informed estimate will be correct as often as the most frequent outcome truly occurs in the environment, when action a is taken in state s. In deterministic domains, where there exists an operator o for all state action pairs such that $p(o|s,a) = 1$, the *accuracy* of a model is the complement of disparity, or $1 - \bar{d}(o(s,a),\hat{o}_i,(s,a))$ over all state action pairs.

Our accuracy claim states that the modeling system informs an agent such that it can make a prediction for any sensory state s, and that the accuracy of our model improves with experience. If we can establish accuracy, then we can turn our attention to facilitation with the certainty that if an agent can facilitate a goal outcome, it can reliably achieve it.

1.4.2 The Facilitation Claim

In all but the most trivial domains, a goal outcome o_g will have nonzero $p(o_g|s,a)$ for only a small range of s and a. That is to say, an arbitrary outcome can only be

[3] In nondeterministic environments, or those in which an agent's sensors are insufficient for exact predictions, $o(s,a)$ may be nonstationary from instance to instance.

achieved trivially (with a single action) from a limited region of the state space. This is why an agent requires the ability to plan. Plans allow the agent to reach those portions of the state space where the probability of a goal outcome are maximized; plans *facilitate* goal outcomes. The facility claim can now be specified in detail using model accuracy as a basis.

When an agent chooses to pursue a goal outcome o_g starting in state s_s, and the probability $p(o_g|s_s)$ is zero (or very low), the agent must construct a sequence of actions (a plan) that will move it into a state specification s_g, where s_g represents a region in sensory space in which $p(o_g|s_g)$ is non-zero (and preferably close to 1). A plan can be thought of as having two parts. The first part is an action sequence which moves the agent into the state region s_g, which we call the *prefix*, and denote $\wp(s_s, s_g)$. The second part we call the *goal step*, and is an action a_g chosen as $\arg\max_a p(o_g|s_g, a)$. An agent attempts to achieve goals by executing a plan prefix followed by a goal step.

In our discussion of accuracy, we established expectations about the goal step. With a perfect model, the goal step of a plan will unfold as o_g with probability $p(o_g|s_g, a_g)$. The matter of evaluating facility, thus, is a matter of evaluating whether or not the prefix $\wp(s_s, s_g)$ succeeds in getting the agent from s_s into state s_g.

Let a *myopic actor* be an agent that cannot generate prefixes. The best that this type of actor can do in starting state s_s, with some goal outcome o_g, is execute the action with the highest estimated probability of resulting in the goal outcome, $\arg\max_a \hat{p}(o_g|s_s, a)$. In contrast, a *planning actor* is capable of producing a prefix $\wp_i(s_s, s_g)$ designed to move the agent from its start state s_s, into a region of the state space s_g where $\hat{p}(o_g|s_g, a) > \hat{p}(o_g|s_s, a)$. The facilitation claim, generally speaking, states that generating and executing a prefix improves the probability of achieving goals. Furthermore, we claim that facilitation improves as an agent collects experiences.

We substantiate the facilitation claims in a manner similar to the accuracy claims, using the same disparity metric. A myopic actor cannot generate or execute prefixes. It selects a single action a_g that maximizes the probability of the goal outcome o_g given the state s_s that it is in. When it executes a_g in state s_s, it will produce an actual outcome $o(s_s, a_g)$, and we can compute the disparity for the myopic actor's behavior as $d(o_g, o(s_s, a_g)$. An agent that can plan, however, will generate a prefix $\wp(s_s, o_g)$ that when executed will improve the probability of a_g resulting in the goal outcome. Let us denote the outcome of executing the prefix $\wp(s_s, o_g)$, followed by a_g, starting in state s_s, as $o(s_s, \wp(s_s, o_g), a_g)$. The disparity for the planning actor's behavior will be $d(o_g, o(s_s, \wp(s_s, o_g), a_g))$.

The ratio $\frac{d(o_g, o(s_s, \wp(s_s, o_g), a_g))}{d(o_g, o(s_s, a_g))}$ is useful in measuring the advantage that executing a prefix can provide. If the disparity between the actual outcome and the goal improves as a result of executing the prefix, then this ratio will be less than one. Let us define the facilitation statistic as the complement of the disparity ratio: $1 - \frac{o(s_s, \wp(s_s, o_g), a_g)}{d(o_g, o(s_s, a_g))}$.

As this formulation implies, facilitation can be measured for arbitrary combinations of states and goal outcomes. Like the accuracy measure, we can aggregate over the entire space of states and actions, or sample from that space, and arrive at more general, average performance measures. However, there are a great many (and in some cases, infinite) states and goals in all but the simplest domains. Averaging over such a large space is practically prohibitive, and likely to obscure informative structure of how facilitation changes over time. Questions such as, "are some outcomes easier to facilitate than others?" and, "do some states more advantageous to start in, in terms of facilitation?" are best answered by conducting a structured analysis of facilitation. Thus, we introduce two different perspectives on facilitation:

Coverage is the percentage of the state space for which a planning actor can facilitate a fixed income o_g.

Capability is the percentage of possible goal outcomes that can be facilitated from a fixed point or region of the state space.

As we did with the general facilitation statistic, we claim that both coverage and capability illuminate the advantages of planning, and that these statistics improve as our system sees more experiences and builds better models. These two new measures will allow for better experimental control and a more in-depth understanding of facilitation and planning when we revisit them in our evaluation.

1.5 Contributions

The contributions of this dissertation fall into two categories: general and technical. General contributions pertain to what we have learned and demonstrated in the process of constructing a system that extends the paradigm of machine learning to development, a closed-loop process in which an agent pursues goals of its own design and learns behaviors in an unsupervised manner. Technical contributions pertain to the algorithms that comprise our developmental model and how they advance the state of the art of autonomous agents.

The primary contribution of this dissertation is perhaps one of the simplest ideas. It is the idea of *planning to act*; that the goals of a self-determining agent are to reproduce experiences. It is this idea that allows us to close the developmental loop by reducing an intractable goal space to a manageable set of practicable goals. This effectively lifts the constraint that previously seperated learning systems from developmental systems: that individual learning tasks had to be specified by hand, either as goals for a planning system, a reward functions for reinforcement learning, fitness functions for genetic algorithms, and so on.

Closing the loop on development, in addition to the practical benefits of a system that develops knowledge necessary to explore and exploit the affordances in an environment, provides a starting point for a system that can develop many types of conceptual structure. For the many schools of research that contend all knowledge is grounded in information gathered from simple, sensorimotor interaction with the environment, a system capable of autonomously growing a corpus of activities is a platform on which later developmental programs can be built. In later sections, we show that our system does indeed build useable representations of activities, and builds representations that appear as perhaps the beginnings of classes and concepts. Related work, such as that described in [62], shows promise that systems for language development could be tied in with a system such as our own that would allow an agent to associate word meanings with representations of activity.

Our modeling system represents a technical contribution as a technique for learning planning operators under adverse conditions. Various approaches to learning planning operators and action models exist, including Benson's TRAIL agent [5, 4], Gil's extensions to the PRODIGY planner [28, 29] and Wang's subsequent extensions to that same planner [86]. Systems that perform planning at the sensorimotor level are rare, though, and currently those that learn planning operators are rarer still. Our modeling system demonstrates that it is possible to learn planning operators in domains where actions have temporal extent, have complex dynamics, and where a single operator will have context-dependent effects that are unknown a priori. Our system simultaneously learns to distinguish between the categorically different outcomes of primitive actions and the conditions under which the different outcomes are likely to occur. In addition to the technical contribution of generating useful planning operators from sensorimotor experience, our modeling system suggests a data point in the debate over innateness; we demonstrate that, contrary the history of classical planning, the full set of operators need not be innately known for an agent to develop coherent behavior, and that general purpose time series analysis and feature detection algorithms may suffice to handle this task over the course of development.

1.6 Outline

The remainder of this document is organized as follows. In the next four chapters, we look in detail at each of the individual components of our developmental architecture as pictured in figure 1.9. Each chapter contains a literature review of relevant work ending in a motivated argument for choice of algorithms and representations, followed by a detailed description of our implementation, and any intermediate experimental results generated over the course of developing our system.

We begin with domain modeling in chapter 2. Here, we describe how the experiences of an agent are transformed into operator models. This chapter includes a breakdown of how we approach modeling an agent's domain. We draw on considerable experience with the Pioneer-1 and Pioneer-2 mobile robots in this section to arrive at a process that involves automated data analysis and inductive learning.

Once we have an operational description of operator models, we can consider algorithms that generate activities that achieve goals. In chapter 3, we review approaches to the problem of learning activities. We look at each as a candidate for our architecture and how the requirements of a developmental system influence our eventual choice of planning as a basis for the development of activities. In chapter 4, we describe how plans are executed to achieve goals, and how plan failure can be detected and execution aborted. Plan execution is non-trivial in our system, and can have considerable influence on the success or failure of an accpetable plan actually achieving its goal.

In chapter 5, we complete the developmental cycle by describing goal selection in detail. We build up a *motivational system* that guides an agent through development. This system allows an agent to generate its own goals and then rank them according to a variety of motivations, ultimately resulting in a preference for what the agent would like to do next. The motivational system balances the wants and needs of an agent, effectively managing the tradeoff between exploration for the sake of learning and exploitation for the sake of satisfying its own needs (if it has any beyond exploration).

We evaluate our developmental system in chapter 6 by testing the claims introduced in section 1.4. Though examples and preliminary results with the Pioneer robot are added to the discussion throughout this dissertation, our evaluation is based on the GridSim domain, an environment for which we have tighter experimental control. We present trials of our system driving a GridSim agent under a variety of conditions to show how the measures of accuracy and facilitation behave as our agent accrues experience. We follow up our analysis of the system performance with conclusions and alook at the future directions this work might continue in chapter 7.

CHAPTER 2
MODELING

The modeling component of our system acts as an oracle. When the motivational system or planner require knowledge of how the domain works, they query the modeling system. The modeling system turns sensorimotor experiences into operational knowledge and shares that knowledge to any other component that needs it.

The primary consumer of domain knowledge is the planner. Recall from the system overview that the planner strings actions together to make activities. In order to do this, it needs *operator models* that express how an agent's actions affect its environment. Since we are dealing with agents at the sensorimotor level, the classical notion of a planning operator, which deals at the more abstract level of propositional logic, is not a perfect match for our system.

In classical planning, an agent has direct access to a set of *operators*. An operator is defined by its *preconditions* and *postconditions* that describe under what circumstances an operator can be executed and what happens when the operator is executed, respectively. An operator's preconditions are propositions that have truth values. Suppose an agent has an operator called *deliver-mail*. Its preconditions might be the proposition

$$\exists m | (mail(m) \wedge carrying(m))$$

This proposition states that there is exists something called m that is a parcel of mail, and that the agent is carrying m. The proposition is true if the agent has a parcel to deliver, and false otherwise. If this precondition is true, then the operator deliver-mail can be executed. It is said to *apply* in the current state. If either of the propositions in the precondition is not satisfied, the operator does not apply, and simply cannot be executed by the agent.

Postconditions in classical planning are specified similarly, as propositional assertions making up *add* and *delete lists*. The add list specifies new propositions that become true as a result of executing an operator and the delete list specifies propositions that are no longer true once the operator completes. The postconditions of deliver-mail would delete the proposition $carrying(m)$, in addition to any other effects of delivering mail that might be expressed in the mail delivery domain.

There are two major differences between the world that the classical planners operate in and the world of a sensorimotor agent. First, sensorimotor agents have direct access to *actions*, and not the more abstract notion of an operator. Actions always apply; there are never preconditions to attempting to turn an effector on or off. However, a single action may have a variety of *outcomes*. Consider the Pioneer robot. It has an action for moving forward, which can be executed at any time, without conditions on when it may or may not be used. When it does execute the MOVE action, any one of a number of experiences may unfold. It may bump into something that it can push, or it may bump into something immovable. It may not bump into anything at all. An observer may label these outcomes "push", "crash", and "free move", respectively. These outcomes may be identified by the sensory patterns they produce. A sensorimotor agent does not know the outcomes of its actions a priori.

The second difference between classical and sensorimotor operators is that preconditions and postconditions must be expressed in sensorimotor terms. There is no innate set of perceptual *propositions* that the agent can use to express preconditions and postconditions. Our system must express preconditions and postconditions at the sensorimotor level. At this level, preconditions amount to relational tests on sensor values like *is sonar-0 less than 1000?* Postconditions must express how sensors change as an action executes.

The job of generating operator models at the sensorimotor level, then, consists of two tasks. First, the agent must identify the qualitatively different outcomes of its actions. Each outcome forms the basis for an operator model. We call this task *outcome classification*. Next, the modeling system must learn the sensory conditions under which a given action will produce a given outcome. This roughly corresponds to learning preconditions, with one small distinction to be made. Preconditions in the classical planning sense express a causal relationship. An operator applies because the preconditions are true. In our system, we are learning conditions associated with outcomes. There may or may not be a causal relationship between the condition and the outcome. Therefore, we call the conditions our learning system detect *initial conditions*, and we refer to this second task of the domain modeler as *initial condition induction*. In the sections that follow, we describe algorithms we have studied and implemented to perform these two tasks.

2.1 Outcome Classification

Outcome classification can be viewed as an instance of a more general AI problem called *classifier induction*. Classifier induction denotes a class of problems in which an algorithm learns to classify instances given their features. Outcomes classification fits this description, with one caveat: classifier induction is traditionally a supervised process. That is, the set of classes that a problem instance might belong to are known in advance and the learner is generally given access to a set of training instances that have been correctly labeled. Inductive classification systems like decision tree induction [8, 38, 66] and version space algorithms [56], attempt to identify features of problem instances associated with class membership to produce rule-like structures that can be used for the classification of new, unlabeled instances. Statistical approaches like Bayesian classifiers [20, 11, 41] take a probabilistic approach to the same problem, generating conditional probability distributions over class labels given feature values, and labeling unknown instances based on the most probable result in training data. The problem is one of diagnosis, and indeed much of the literature of machine classification focuses on tasks such as medical diagnosis.

In our system, we would like to classify the qualitatively different outcomes of an agent's actions, without knowing them in advance. The first hurdle for our system to overcome if we are to adapt existing machine classification algorithms is the requirement that outcome classes be known a priori. We maintain that an agent does not begin life knowing that moving may result in crashing or pushing or any other type of experience. An agent can make only basic sensory distinctions, such as the distinction between *moving* and *stationary*, or *increasing* and *decreasing*. These simple distinctions can then be combined in various ways to identify the qualitative differences between different experiences.

A direct approach would be to specify the native sensor distinctions an agent can make and use these primitive distinctions as a priori class labels. Then, we build feature detectors for them. The task of classifying outcomes can then be posed as a supervised classification problem. In previous related work, we devised an algorithm for the induction of *context operator difference tables* [74] (CODTs). The CODT structure is similar in spirit to the *operator difference table*[59] of classical problem solving, a structure that relates actions to their effects on the environment. The CODT introduces an obvious difference: the CODT can encode conditional operator effects. The algorithm asks the question: "Given the sensory configuration at the outset of an activity, does sensor s increase, decrease, or stay stationary?" The CODT induction algorithm applies a simple feature detector (one that can detect increasing and decreasing trends) to sensor data in order to generate primitive class labels for sensor behavior (*increasing*, *decreasing*, or *neither*). Once the algorithm has labeled sensor trends, a supervised algorithm can be used to generate the contexts (which correspond roughly to initial conditions).

The learning procedure, in greater detail, is as follows: First, the agent selects an action to explore. Next, it records a snapshot of readings from its sensors, which we call the *context*, $\mathcal{C}$, in which the action is taken. Then, the agent initiates the action, and continues to record its sensor readings for the duration of activity, which we call the *effects*, $\mathcal{E}$, of the activity. The result is a context/effect pair ($\mathcal{C}$,e_s) for each sensor s in $\mathcal{E}$. The agent repeats this process, and when a handful or experiences have been recorded, it generates a decision tree for each action/sensor pair using the collected ($\mathcal{C}$,e_s) tuples. Each tree becomes a cell of the CODT, which, like the operator difference table, describes the expected change in each sensor given an action and a context. A typical CODT tree might assert that the MOVE activity will produce an increase in translational velocity unless the bump sensor is active, in which case, translational velocity will remain stationary.

Our experience with CODT induction was that reasonably predictive models could be generated, and those models illuminated basic structure of the environment: that sonars were predictive of collisions, that visual cues could predict objects entering the gripper of the Pioneer, and so on. The CODT work also underlines an important trade off associated with supervised classification algorithms. One must be careful not to choose class labels that are too specific (enumerating the entire space of different outcomes of an action, for example), thereby attempting to express *too much.* When the set of class labels becomes too specific, the engineering task of pre-specification becomes more difficult, and the amount of data required to fill out representations can become prohibitive. At the same time, one must not choose labels too general (like those used in CODT induction) and express *too little* for the representations to be useful to downstream components. While the CODT induction algorithm was successful at classifying slope changes in sensors associated with the onset of activity, we felt that it might not provide a requisite level of predictiveness for planning. More importantly, dependencies between the sensors could not be expressed, nor could they typically be inferred from the CODT. One could not predict how sets of sensors would change in response to action. In typical environments, dependencies among the various sensors are prevalent. Planners must be able to account for dependencies like these. The apparent lack of expressiveness in the CODT approach led us to the next body of work we will describe, in which no attempt is made to coerce the classification problem into a supervised task; instead, we approach outcome classifcation as an *unsupervised* task.

Unsupervised classification refers to a class of learning algorithms in which neither the set of class labels nor the criteria for class membership are known in advance. *Clustering* algorithms partition data into groups that are judged homogeneous by some *distance metric* [21]. Each of the groups produced by the clustering algorithm is considered to correspond to a class.

Typical clustering algorithms produce hierarchical groupings by computing inter-datum distances and repeatedly dividing or merging partitions of data based on the distances between them. Those algorithms that work by dividing larger partitions into smaller ones are called *divisive* while those that are based on merging partitions are called *agglomerative*. For a simple dataset, such as the two-dimensional dataset shown in figure 2.1a, Euclidean distance can be used as a basis for building clusters. These algorithms split or merge clusters recursively until some threshold is met, which may be that a desired number of clusters has been generated, or that a maximal intra-cluster variance has been reached. The partitions at the leaves of the hierarchy built during clustering may then be assigned cluster labels. Figure 2.1 shows 6 two-dimensional data plotted on X-Y axes (a), a hierarchical clustering carried out to a threshold of 3 clusters (b), and a graphical representation of those clusters (c).

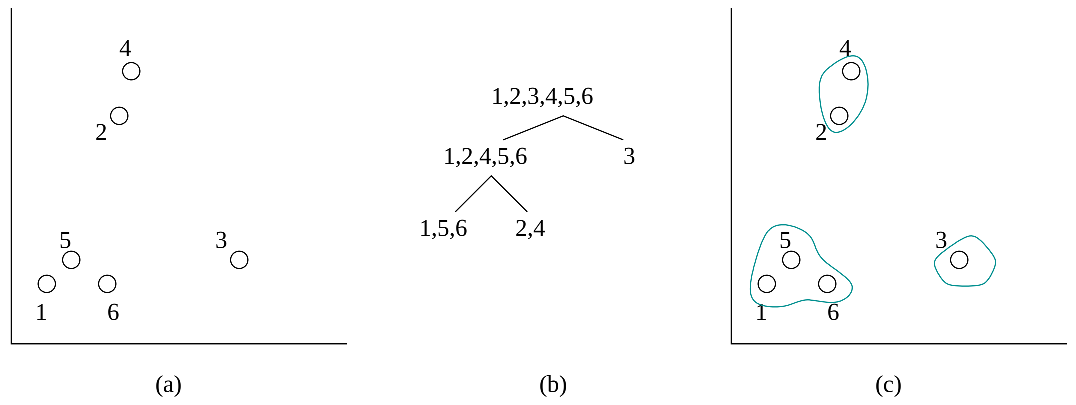

Figure 2.1. **(a)** Two data and their Euclidean distance. **(b)** A group of data and one possible clustering based on the Euclidean distance metric. **(c)** The clustering, interpreted graphically.

A variety of clustering algorithms exist for the continuous valued, multivariate clustering problem, including Cobweb [24], Autoclass [10], and Snob [85]. These systems have a track record of finding class structure in data when possible class labels are not initially known. Still others have extended clustering technology to produce clusters of time series [40, 90, 69]. Distance-based clustering algorithms can solve many classification problems so long as an appropriate distance metric can be found. A suitable distance metric for comparing the sensory time series like those found in our agents' experiences would allow us to use a simple clustering scheme to collect experiences into groups with qualitatively similar outcomes.

We now turn to a detailed description of a clustering system we developed for identifying qualitatively different action outcomes for the Pioneer mobile robot.

2.1.1 Identifying Outcome Classes by Clustering

Clustering algorithms comprise three components. The first is the mechanism by which data are either merged into clusters or clusters are divided into smaller ones. The second is the distance metric by which data are judged as similar or dissimilar. The final component is a set of *stopping criteria* which signal to the algorithm that it should stop dividing or merging clusters.

The most significant undertaking in our application is to define a measure of similarity for the experiences of a mobile robot. Finding such a measure of similarity is difficult because experiences that are qualitatively the same may be quantitatively different in at least two ways. First, they may be of different duration, making it difficult or impossible to embed the time series in a metric space and use something like Euclidean distance to determine similarity. Second, within a single time series, the rate at which sensors change can vary non-linearly. For example, the robot may move slowly or quickly toward a wall, leading to either a slow or rapid decrease in the distance returned by its forward-pointing sonar. In each case, though, the end result is the same, the robot approaches the wall and then bumps into it. Such differences in rate make similarity measures such as cross-correlation unusable.

The measure of similarity that we use is Dynamic Time Warping (DTW) [72]. DTW is a generalization of classical algorithms for comparing discrete sequences (e.g. minimum string edit distance [13]) to sequences of continuous values. It has been used extensively in speech recognition, a domain in which the time series are notoriously complex and noisy, until the advent of Hidden Markov Models which offered a unified probabilistic framework for the entire recognition process [37].

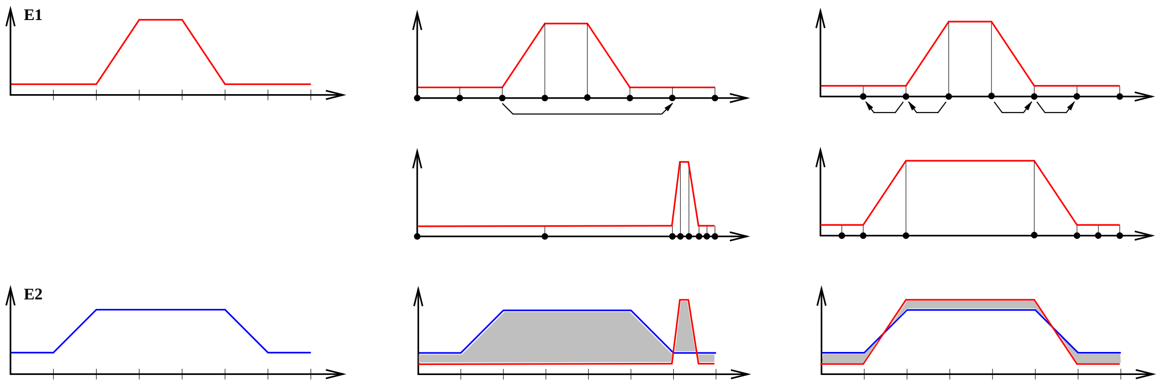

Figure 2.2. Two time series, E_1 and E_2, (the leftmost column) and two possible warpings of E_1 into E_2 (the middle and rightmost columns).

Let E denote an experience, a multivariate time series containing n measurements from a set of sensors such that $E = \{e_t | 1 \leq t \leq n\}$. The e_i are vectors of values containing one element for each sensor. Given two experiences, E_1 and E_2 (more generally, two continuous multivariate time series), DTW finds the warping of the time dimension in E_1 that minimizes the difference between the two experiences. Consider the two univariate time series shown in figure 2.2. Imagine that the time axis of E_1 is an elastic string, and that you can grab that string at any point corresponding to a time at which a value was recorded for the time series. Warping the time dimension consists of grabbing one of those points and moving it to a new location. As the point moves, the elastic string (the time dimension) compresses in the direction of motion and expands in the other direction. Consider the middle column in figure 2.2. Moving the point at the third time step from its original location to the seventh time step causes all of the points to its right to compress into the remaining available space, and all of the points to its left to fill the newly created space. Of course, much more complicated warpings of the time dimension are possible, as with the third column in figure 2.2 in which four points are moved.

Given a warping of the time dimension in E_1, yielding a time series that we will denote E_1', one can compare the similarity of E_1' and E_2 by determining the area between the two curves. That area is shown in gray in the bottom row of figure 2.2. Note that the first warping of E_1 in which a single point was moved results in a poor match, one with a large area between the curves. However, the fit given by the second, more complex warping is quite good. In general, there are exponentially many ways to warp the time dimension of E_1. DTW uses dynamic programming to find the warping that minimizes the area between the curve in time that is a low order polynomial of the lengths of E_1 and E_2, i.e. $O(|E_1||E_2|)$. DTW returns the optimal warping of E_1, the one that minimizes the area between E_1' and E_2, and the area associated with that warping.

The area is used as a measure of similarity between the two time series. Note that this measure of similarity handles nonlinearities in the rates at which experiences progress and is not affected by differences in the lengths of experiences. In general, the area between E'_1 and E_2 may not be the same as the area between E'_2 and E_1. We use a symmetrized version of DTW that essentially computes the average of those two areas based on a single warping. [44] Although a straightforward implementation of DTW is more expensive than computing Euclidean distance or cross-correlation, there are numerous speedups that both improve the properties of DTW as a distance metric and make its computation nearly linear in the length of the time series with a small constant.

The identification of an appropriate, effective distance metric facilitates the use of a variety of clustering mechanisms. We have implemented an agglomerative, group-average distance clustering algorithm. The algorithm works as follows. First, begin by computing a $m \times m$ distance matrix D, where the contents of cell $D_{i,j}$ contains the distance between experience E_i and E_j, computed using DTW. Next, the algorithm creates a cluster $\mathcal{C}_i$ for each individual experience, and greedily chooses the pair of clusters with the lowest *group-average distance*, calculated as the average of the distances between experiences in $\mathcal{C}_i$ and those in $\mathcal{C}_j$.

$$GAD(\mathcal{C}_i, \mathcal{C}_j) = \frac{\sum_{c_x \in \mathcal{C}_i} \sum_{c_y \in \mathcal{C}_j} D_{x,y}}{|\mathcal{C}_i||\mathcal{C}_j|} \tag{2.1}$$

This simple clustering algorithm continues until the stopping criterion is met. In many cases, the stopping criterion is simply that the algorithm has merged down to a prespecified number of clusters. For our purposes, this is unreasonable; are we to assume that the robot has innate knowledge of the number of distinctly different outcomes of an action? Instead, we have devised an automatic stopping criterion based on the t-test. Merging continues until the best candidate clusters for merging have a mean inter-cluster distance that is significantly different than the mean intra-cluster distance as measured by a t-test, with a user-adjustable p value.

To evaluate the clustering algorithm, we collected sensor data from 2 sets of experiences: 102 experiences with the robot moving forward or backward in a straight line, and 50 experiences with the robot turning in place. Clustering for both the turn and move sets was performed on two sensor subsets: one which includes the velocity encoders, break beams, and gripper bumper (which we will call the *tactile* sensors), and one which includes data from the Pioneer's vision system, including the X and Y location, area, and distance to a single visible object being tracked (which we will call the *visual* sensors). Visible objects and objects that impeded or obstructed the robot's path were present in many of the trials.

As we would like clusters to correspond directly to relevant outcomes of activity, our primary means of evaluating cluster quality is to compare them against clusters generated manually by the experimenter who designed the 152 robot experiences. Post-hoc labels assigned to the hand-built clusters are summarized in table 2.1. In the visual tracking problems, the clusters correspond to visible objects' relations to the agent during activity; the object may move across the visual field while turning or it may loom while being approached. In the tactile problems, clusters correspond to the Pioneer's velocity and the types of contact made with objects in the environment during the activity; heavy objects halt the Pioneer's progress, and are labeled "crash", while light, small objects merely trigger the break beams and are labeled "push".

We evaluate the clusters generated by DTW and agglomerative clustering with a 2×2 contingency table called an *accordance table*. Consider the following table:

	t_e	$\neg t_e$
t_t	n_1	n_2
$\neg t_t$	n_3	n_4

We calculate the cells of this table by considering all pairs of experiences e_j and e_k,

move/tactile	**turn/tactile**	**move/visual**	**turn/visual**
+250 unobstructed	+100 unobstructed	no object	no object
+100 unobstructed	+100 never stops	heavy noise	can't move
-100 unobstructed	+100 bump	approach on right	pass left to right
-250 unobstructed	+100 blocked	approach disappear	pass right to left
+250 temporary bump	+100 temporary bump	discover left reverse	discover right
+100 temporary bump	+100 blocked bump	vanish on right	discover left
+250 push delayed bump	-100 unobstructed	vanish on left	left to right
+250 delayed bump	-100 temporary bump	retreat left	vanish off right
+100 delayed bump	-100 impeded turn	discover right	vanish off left
+250 crash beam1	-100 blocked	approach ahead	
+250 squash		approach, gets big	
+250 push blocked		approach on left	
+250 push		approach, stays small	
+100 push			
+100 push shallow			
+100 blocked			
-100 blocked			

Table 2.1. Outcome labels given to the hand built clusters for each of the 4 experience sets.

and their relationships in the target (hand-built) and evaluation (DTW) clusterings. If e_j and e_k reside in the same cluster in the target clustering (denoted by t_t), and e_j and e_k also reside in the same cluster in the evaluation clustering (denoted by t_e), then cell n_1 is incremented. The other cells of the table are incremented when either the target or evaluation clusterings places the experiences in different clusters ($\neg t_t$ and $\neg t_e$, respectively).

Cells n_1 and n_4 of this table represent the number of experience pairs in which the clustering algorithms are in accordance. We call n_1+n_4 the number of *agreements* and n_2+n_3 the number of *disagreements*. The *accordance ratios* that we are interested in are $\frac{n_1}{n_1+n_2}$, accordance with respect to t_t, and $\frac{n_4}{n_3+n_4}$, accordance with respect to $\neg t_t$.

Table 2.2 shows the breakdown of accordance for the combination of dynamic time warping and agglomerative clustering versus the ideal clustering built by hand. The column labeled "$|\mathcal{C}^t| - |\mathcal{C}^a|$" indicates the difference between the number of hand-built and automated clusters (a negative value indicates fewer clusters in the automated case. In each problem, the automated algorithm clustered more aggressively, resulting in fewer clusters. The columns that follow present the accordance ratios for experiences grouped together, apart, and the total number of agreements and disagreements.

The table shows very high levels of accordance. Ratios ranged from a minimum of 82.2% for experiences clustered together (t_t) in the move/visual set to 100% for experiences clustered together in the turn problems. For the turn problems, the aggressive clustering may account for the high t_t accuracy, causing slightly lower accuracy in the $\neg t_t$ case. Various techniques for increasing accordance further have been explored, including post-hoc cluster optimization [63], and results are encouraging that this clustering scheme can serve as the basis of an intelligent agent that can distinguish experiences by their outcome.

	$\lvert\mathcal{C}^t\rvert - \lvert\mathcal{C}^a\rvert$	t_t	$t_t \wedge t_e$	%	$\neg t_t$	$\neg t_t \wedge \neg t_e$	%	Agree	Disagree	%
Move visual	-5	876	720	**82.2**	4275	4125	**96.4**	4845	306	**94.0**
Move tactile	-7	443	378	**85.3**	4708	4468	**95.0**	4846	305	**94.0**
Turn visual	-5	262	262	**100.0**	599	571	**95.3**	833	28	**96.7**
Turn tactile	-6	163	163	**100.0**	698	593	**85.0**	756	105	**87.8**

Table 2.2. Accordance statistics for automated clustering against the hand built clustering.

2.1.2 An Alternative Clustering Scheme

Distance-based agglomerative clustering has two fundamental shortcomings which begin to surface as an agent accumulates experiences. First, it is sensitive to the sample size it works with. The decision to merge two clusters is made based on a t-test and a parameter α, a probability of incorrectly rejecting the hypothesis that two experiences are generated by a different process. As the number of experiences grows, the number of merging decisions grows exponentially, and consequently, the probability of making a poor decision to merge also grows. When a poor decision to merge is made, operator clusters become impure and any inferences made of cluster membership are polluted accordingly. This problem of systematic overfitting has been studied by Cohen and Jensen [12]; adjustments can be made to reduce the probability of error when the number of comparisons can be counted or estimated. Still, this sensitivity to the number of experiences appears to manifest in our trials, resulting in unnecessarily impure clusters.

The second fundamental shortcoming of clustering applied to our problem is that distance-based clustering is only as good at building pure clusters as the distance metric accurately measures similarity in the desired way. Dynamic time warping is allowed to stretch and compress the temporal dimension of two experiences to optimize the distance between them. In some cases, DTW could compress what we as observers would consider interesting regions of experiences (regions where there is substantial, discernible behavior in the time series data) down to a tiny temporal extent in order to reduce its distance to an experience in which there was very little interesting behavior. Figure 2.3 shows a situation typical of the type of time series data we are working with. The figure shows sensor traces for two distinctly different experiences: one in which the sensor starts high and goes low, one in which the sensor starts low and goes high. This might be a sonar sensor as an agent approaches a wall in one case and retreats from the wall in another. The DTW algorithm is able to warp these two sequences to the point where they seem very similar.

In our trials, the result of these shortcomings was that clusters began to become unacceptably noisy as the agent accumulated more and more experiences. They began to fall out of accordance with human judgment as the faulty decisions to merge previously distinct clusters into one larger supercluster are made. Impurity complicates downstream portions of the system; initial condition induction becomes problematic. The planner, which relies heavily on initial condition induction, suffers accordingly.

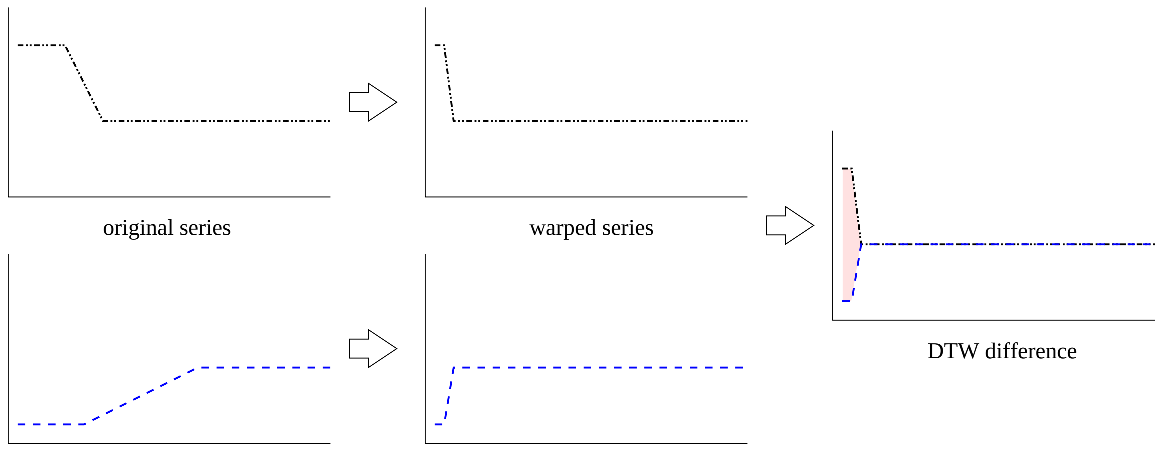

Figure 2.3. Two time series that are qualitatively different but can be manipulated by DTW to appear similar by compressing the region over which there are dissimilarities.

After studying the failings of this clustering scheme, we arrived at a conclusion that too many clusters, as long as they retain purity, is preferable to too few clusters. This is because pure clusters are still useful for prediction. It is better to have two useable clusters when there should be one rather than one polluted cluster when there should be two. The combination of DTW and agglomerative clustering was prone to overly-aggressive merging, and thus too few clusters. We found the connection between the complicated distance metric and the eventual results was too opaque to produce acceptable results in the general case; we did not have fine enough control over the algorithm to arrive at a solution we felt would produce pure, useful operator models that would scale to large numbers of experiences. Eventually, we returned to the idea underlying the CODT work and redevelop the approach of using simple feature detectors to produce grounded class labels and blend in some of the aspects of agglomerative clustering that we felt provided an advantage over the CODT system.

We developed a simple clustering algorithm that utilizes feature detectors similar to the one used in the CODT work as a basis for building clusters. The goal of this algorithm was not to be sensitive to sample size and to most probably build too many clusters than too few. We call this algorithm the *simple symbolic clusterer*, and it is based on the following idea: for the purposes of planning and acting, “interesting” regions of time series are those in which there is a non-zero slope and the behavior of the time series is relatively predictable. By identifying portions of time series in which there is an identifiable, consistent nonzero slope, we can reduce a large amount of data to a small sequence of segments (or features).

Each individual interesting segment can then be classified and labeled. The algorithm essentially processes time series sensor traces into a sequence of labeled segments, or simpler still, symbol strings. For the simple symbolic clusterer, those experiences whose symbol strings match, exactly, belong to the same cluster. The actual mechanisms for segmenting and classifying segments are purposely left vague at this point, since many candidates produce more or less detailed results. We discuss several we have implemented below. The key to this new suite of algorithms is that the decision to merge clusters is deterministic. If feature detection is also deterministic (this depends on the individual detectors), then the entire classification system becomes deterministic and scalable. It will produce the same clusters, every time, no matter how much data.

A number of approaches to feature detection are possible. We call the various feature detection schemes *filters*. They vary in their level of detail and features that they attend to. We now present three variants of the simple symbolic clusterer that retain different levels of detail in the representations they produce. We revisit the question of what is the right level of detail for our purposes once the remainder of the modeling system is described in section 2.2.3.

The SSC1 Clusterer

The first outcome classification filter we describe is called the SSC1 clusterer. It retains the most detail from the data of the three variants. The SSC1 algorithm uses piecewise linear regression to break time series down into regions where the rate of change remains fairly uniform.

We employ a top-down piecewise linear fitting algorithm that works as follows. Given a time series of length n, find the cut point j that maximizes the difference in slope between the regression line computed over the range of data in $[1 \dots j]$ and the regression line for $[j+1 \dots n]$. Next, test the hypothesis that the two lines have the same slope. In essence we are testing whether the trends of the two ranges of data were produced by the same process. The mathematics behind this test are the same as the ones that were used in the context operator difference table induction scheme described in section 2.1. They are described fully in [74] and summarized in Appendix A. If the hypothesis cannot be rejected, the algorithm does no splitting, fits a single regression line to the data, and is finished. If the hypothesis is rejected, then j is marked as a split point, and the piecewise fitting algorithm is recursively called on the two data sets created by splitting the data about j.

The piecewise fitting algorithm breaks a time series down into line segments, each of which has a distinct slope and intercept, and a goodness of fit for the regression line to the data it fits. We use the slope and goodness of fit to then label each segment. Segments can be classified as `increasing`, `decreasing`, `stable`, or `unstable`. The first two classifications refer to positive and negative slopes, while the second two refer to slopes near zero where the fit is good (`stable`) or there is great variance in the data, and the fit is poor (`unstable`). These class labels can be arrived at in a variety of ways; they are currently assigned by a set of fixed rules, specified in advance, and tailored to the characteristics of each sensor. The ability to detect these features in sensor data could be considered an innate endowment of our agent. We also add two additional segment labels, `disc-` and `disc+`, for identifying discontinuities in the data. Discontinuities are simply places in the data where a very large change is encountered in a short temporal window. Since discontinuities are not handled well by piecewise linear

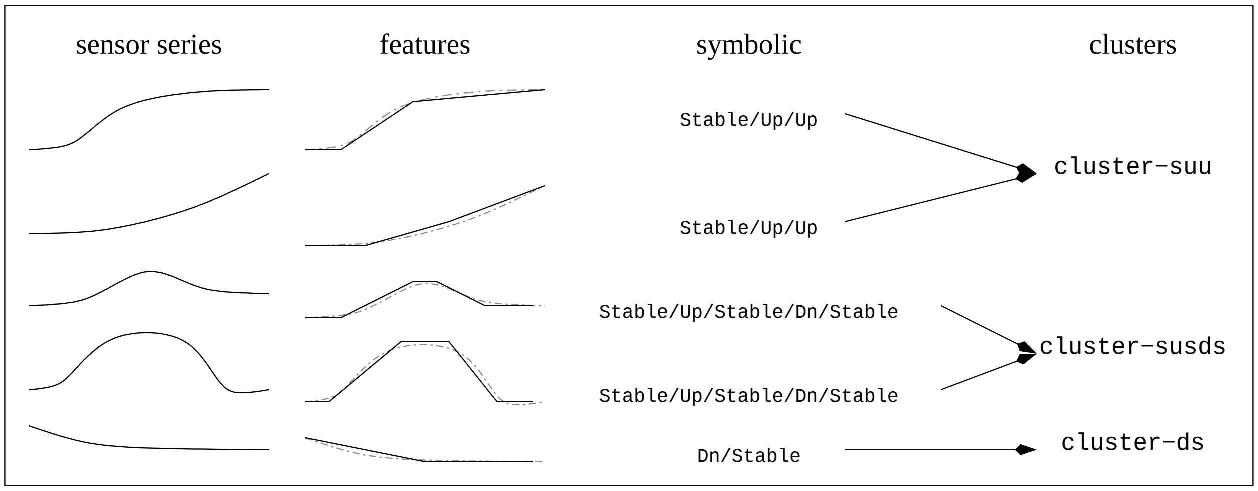

Figure 2.4. An illustration of the SSC1 clusterer.

fitting algorithms, they are detected by a hand-built filter and are marked as cut points prior to applying the fitting algorithm.

Each sensor series can now be characterized by a string of symbols. Since we are most interested in when and how the sensor change, we trim segments of the type (`stable` or `unstable`) from the beginning and end of a time series. Now each series is described by a symbol string that encodes the “interesting part” of the activity that generated it. Series described by the same symbol string can simply be clustered together, and by extension, experiences whose symbolic descriptions match for every sensor, are grouped together.

An illustration of the clustering algorithm is shown in figure 2.4. Along the left hand column are several time series. The second column shows typical piecewise linear fits to each of the original time series. The third column shows symbolic representations of the piecewise fits, and the fourth column shows the final step of clustering series with the same symbolic representations together. This simple illustration shows clustering for the univariate case. It extends simply to the multivariate case.

The Delta-Simple Clusterer

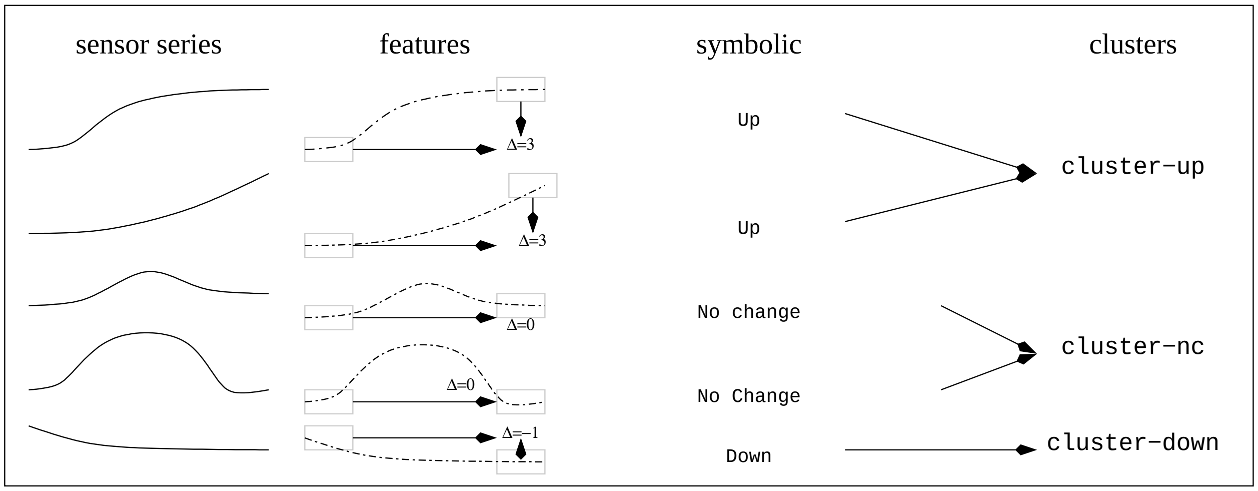

Figure 2.5. An illustration of the Delta-Simple clusterer.

The Delta-Simple clusterer takes the simplicity of the ssc principle to the furthest degree. The idea with Delta-Simple is that an action will either have some effect on a sensor, to increase or decrease its value, or it will not. The "shape" of the sensor time series, or precisely how it behaves during the action, is unimportant compared to the net result of the action in this scheme.

The symbolic description of a sensor time series under Delta-Simple is either `increases`, `decreases`, or `nc` (no change). The symbolic description is computed simply by comparing the state of a sensor before an action is taken with the state when the action is completed. Like ssc1, the thresholds for the classes are hand-coded specific to the various sensors, but could be learned inductively. Experiences whose symbolic sensor descriptions match exactly are grouped together.

An example of the Delta-Simple clustering algorithm is shown in figure 2.5. It differs from the ssc1 algorithm in the second and third columns where the symbolic representation is derived. Instead of a piecewise linear fit, a simple comparison is made to arrive at a value for Δ. If Δ is significantly different that 0, the series is labeled `up` or `down`, otherwise it is labeled `no change`.

The Delta Clusterer

The Delta filter is a straightforward extension of the Delta-Simple filter. Like Delta-Simple, this filter computes net sensor changes over the course of an activity and classifies it by whether the value goes up, down, or does not change. The Delta classifier, however, retains information of scale. If a sensor goes up by 3 units, it will classify the change as `up3`, for example. This allows Delta to make distinctions between outcomes in which the rate of change is significantly different. In our example, the number 3 is an integer, but the granularity of the Delta filter is not limited to integer values, and could be dependent on the characteristics of the sensor being monitored. The Pioneer-2's sonar sensors might work at a granularity of $1000mm$; an activity that moves the Pioneer $1900mm$ closer to a wall might be classified as `down2`. Again, the granularity of Delta on individual sensors is hand-coded in our system, but these values could be configured adaptively. An illustration of the Delta clusterer is shown in figure 2.6.

The Delta clusterer's operation is illustrated in figure 2.5. It differs from the Delta algorithm in the third column where the symbolic representation is derived from the Δ computation. The additional information of scale is incorporated into the labels `up3` and `down1`.

2.2 Initial Condition Induction

Given a classification scheme that groups experiences into clusters with qualitatively similar outcomes, we can go on to ask the question "under what circumstances can outcome o be expected to unfold?" For example, if we are dealing with the Pioneer robot, one might want to determine why moving in one instance results in unimpeded progress while in another the result is crashing into a wall. The answer to this question serves two purposes: First, the answer provides predictiveness. Reactive

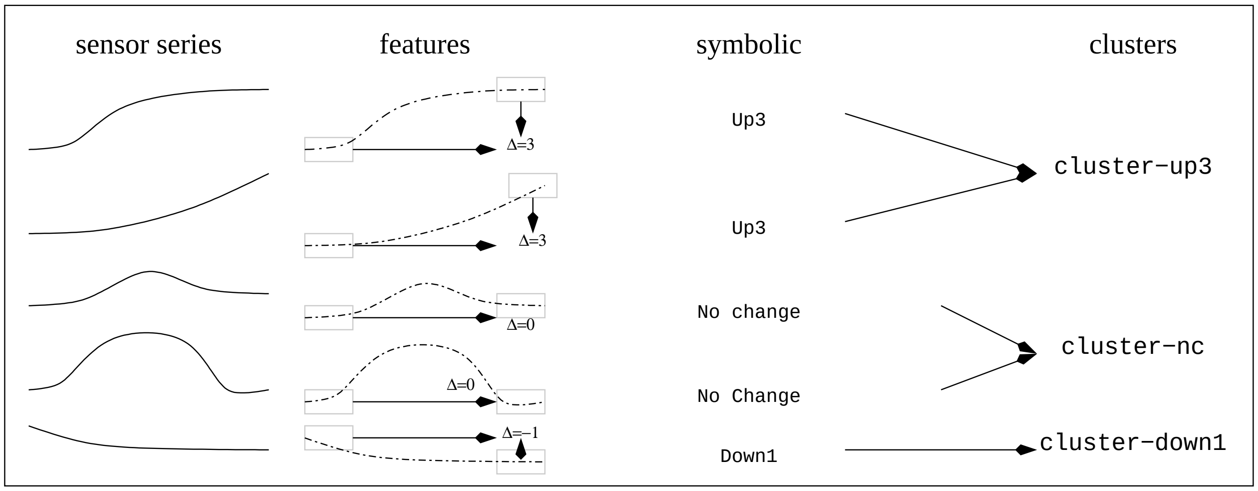

Figure 2.6. An illustration of the Delta clusterer.

control requires that an agent can predict the outcomes of its actions. Second, the answer provides information that can be used as a basis for means-ends analysis.

Before we address the problem of generating predictive models, and show how our models can be used to approximate preconditions, it is worth expounding on the subtle distinction between preconditions used by classical planners and initial conditions, which is what our system produces. When an agent like the Pioneer is considering which outcome an action will produce, it has only its sensors to guide it. When it considers what will happen if it executes the *move* action, it can access its vision system, its sonars, and its internal state sensors. It does not have the benefit of rich representations of the world to tell it that it is facing a wall as opposed to a trash can. It can only associate sensory states with outcomes: a small red object in the visual field is most closely associated with cluster c_1 unfolding, while a large blue object is associated with cluster c_5 unfolding. Whereas with classical planning operators, the link between preconditions and postconditions is causal, we cannot assert a causal relationship. Our system can only claim an associative relationship.

Another way of looking at the distinction between preconditions and initial conditions is as follows. Preconditions, in the classical planning sense, tell the planner two things. They tell the planner what the probability of an outcome is given that some conditions are satisfied. They also tell the planner the probability of that outcome if the conditions are not met. Preconditions are necessary and sufficient conditions for an operator to succeed. Initial conditions, on the other hand, make a weaker statement about the agent and its environment. They do express the probability of an outcome given a set of conditions, but they do not express the probability of failure when the conditions are not met. They can express sufficiency, but not necessity. For our practical purposes, sufficiency is adequate.

Let us call the sensory information collected during the γ seconds leading up to the initiation of an activity the *precursor phase* of that activity. We call the vector of mean sensor values across all available sensors in the precursor phase of an experience the *precursor* to that experience. Initial conditions, then, should be specified as propositions over precursor sensor values. A Pioneer robot agent might posit initial conditions for an activity called LIFT-RED-OBJECT to be

$$((\text{vis-a-area} \in [100 \ldots 1000]) \wedge (\text{grip-beam} \in [0.5 \ldots 1.0]))$$

indicating that the area of the blob in visual channel A must be nonzero (i.e. that there actually is a red object in view), and the gripper beam must be broken (the object must be within the grasp of the gripper).

The problem of initial condition induction, then, is one of learning a mapping between the precursor of an experience and its class label. Since we now have a class label for each experience of the robot (generated via the outcome classification step), we can go about the task of attempting to learn these maps. Again, there is a large body of literature in the field of machine classification that is all relevant to this problem. In the two sections that follow, we examine two major schools of machine classification and their applicability to our particular system: artificial neural networks and inductive methods.

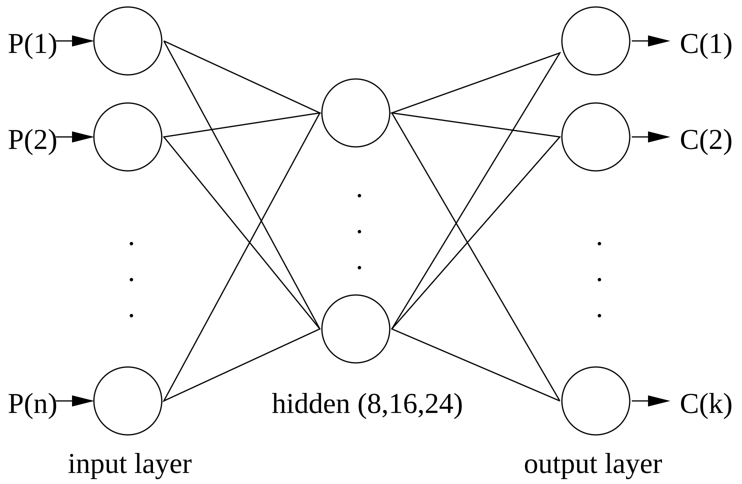

Figure 2.7. A neural network designed to learn how to classify experience outcomes based on their initial sensory states.

2.2.1 Relevant Work: Artificial Neural Networks

Artificial neural networks first emerged in the AI literature as an attempt to model the behavior of neurons in the brain. McCulloch and Pitts pioneered the idea by studying the behavior of a single neuron [53], and as computers became increasingly capable, the technology expanded to modeling simple networks of neurons [61]. Since that time, the scope and simulation of the neural units and networks has undergone much development to reach the point where they have been shown to accomplish a number of significant tasks, from reproducing arbitrary functions from data [16, 26, 33] to complex control tasks like driving a car [65] to machine classification [23]. An artificial neural network capable of learning a mapping from sensor readings to outcome classes would certainly be satisfactory for predicting outcomes, and combined with the ability to interpret the internal structure of a neural network in declarative terms [15, 14], could also be a suitable technology for inducing initial conditions.

We ran set of pilot experiments to evaluate artificial neural networks as the underpinnings of our initial condition induction system. Our initial experiments with artificial neural networks as outcome predictors were run using data collected from the Pioneer robot. Class labels were generated using our first-generation outcome classification scheme: the combination of DTW and agglomerative clustering as described in section 2.1. Recall that our DTW-based clustering system classifies outcomes with respect to a group of sensors' behavior. Thus, each cluster corresponds to an outcome characteristic of the form "sensors $s_i \dots s_j$ conformed to pattern $y_i \dots y_j$". For example, the clustering scheme might generate clusters corresponding to the following labels: "the front sonars decrease then leveled out" or "translational and rotational velocity remained constant". These examples are quite simple, but the combination of DTW and clustering allow for arbitrarily complex patterns to be compared and matched. Each sensor group has a corresponding set of clusters that represent the qualitatively different outcomes observed from those sensors in the experience data. The task was to train artificial neural networks to predict sensor outcome classes for unclassified experiences using precursor data.

We implemented a multilayer artificial neural network architecture as shown in figure 2.7. To train up the network, each experience in the training set was converted to a training instance by pairing its vector of precursor sensor readings with an outcome class label as prescribed by the clustering algorithm. We will refer to the precursor vector of an experience as P, and the precursor value of sensor i as $P(i)$. The class label for an experience will be referred to as C, where C is a bit-vector. The bit string C corresponding to experience e contains all zeros except $C(j)$, where if j is the jth bit of C, and $C(j) = 1$, then e belongs to cluster number j as labeled by the DTW-based clusterer. Training experiences are then presented to the network as pairs $< P, C >$ and the neural network must learn a map from P to C. Each pass through the training set is called a *training epoch*, and the network is updated using the backpropogation algorithm [71] as many epochs as is necessary for the network to stabilize (stop learning).

We tested the efficacy of this neural network for configurations containing 8 to 24 hidden units, using the representative set of 152 experiences as described in sec-

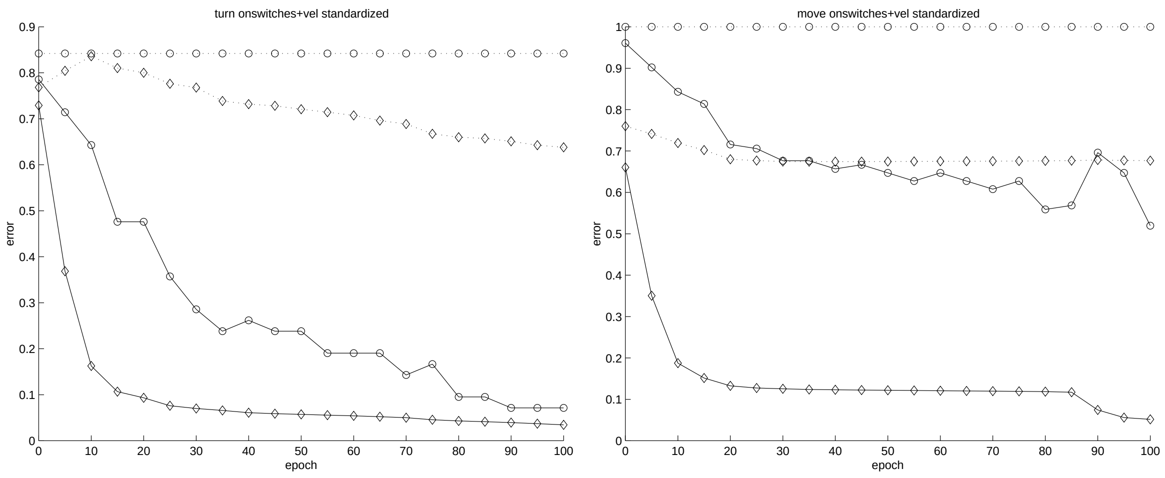

Figure 2.8. Learning curves for a neural network trained to predict outcome classes on Pioneer data for the *tactile* sensor group. Solid lines represent performance on training data, dashed lines represent generalization to test data. Circles plot winner-take-all performance, diamonds plot mean-squared error.

tion 2.1.1. We trained networks to predict the outcome classification for each of the visual and tactile sensor groups (as described in section 1.1, plus clusters built using the Pioneer's seven sonar sensors. We allowed training to proceed until the mean square error during training failed to change for 10 straight epochs.

After the training phase, the networks were tested both on the training data and a set of test data consisting of 25 randomly generated moves and 25 randomly generated turns classified by the clustering algorithm. Two representative learning curves from the 24 unit network are shown in figure 2.8. Solid lines in the plots represent performance of the network on the training data, while dashed lines represent generalization to testing data. Plots marked with circles show winner-take-all (wta) performance. In the winner-take-all task, the network is presented a feature set, computes its outputs, and the output with the highest activation level is the predicted class. The winner-take-all score is 1 if the correct class is predicted, 0 otherwise. Those plots marked with diamonds represent mean-squared error (mse). The mean-square error measure is a raw measure of the mean difference between the outputs of the network and the actual class vector of the test instance.

The results shown in figure 2.8 are typical across the sensor groups and different network configurations. The networks almost invariably converge on a configuration that drives the mean-squared error to near zero. Winner-take-all performance is considerably worse, even on training data. Generalization is poor in almost all cases, but better in the TURN condition. Classification (wta) error was in general very high, ranging from 40% to 70% on the training set, while test set generalization was far worse, averaging an error rate of more than 70%. A more detailed summary of our results can be found in [73].

If neural networks can represent arbitrary functions, why did our experiments fail to produce high levels of classification? There are several reasons that might have contributed to the demise of the neural network:

1. The Pioneer domain is noisy, and partially-observable
2. The neural network was evaluated by a winner-take-all metric
3. Overfitting is a significant problem for the neural network formulation
4. Nondeterminism in the clustering algorithm propogates down to the prediction task and hurts performance
5. Our choice of input or output representation, or our choice of network structure, was not well suited for the task.

The poor performance of the neural network is most likely caused by a confluence of some or all of the above problems. Consider the Pioneer mulling over the possibilities of engaging in the TURN action. Will a red object come into view? Will it become stuck? If the Pioneer has nothing in its visual field, and is not already stuck, nothing in the Pioneer's sensor array can shed any light on this question. This is what is meant by *partial observability* – the information relevant to predicting the outcome of a TURN action is simply not available to the Pioneer. When operating in a partially observeable domain, an agent has three choices. First, it can get better sensors and attempt to swing its balance towards *full observability*. Second, it can maintain some kind of *memory*, and add aspects of its memory into the modeling equation. Third, it can make its predictions probabilistic and do the best it can. The first solution is out of the question. We could engineer the entire problem away by building in better sensors or an advanced perceptual system and controllers, if that was our goal. Or goal is an account of development, though, and not stellar performance.

The second solution, a more sophisticated model of the environment in which an agent can remember the locations and features of objects would surely make a difference to the performance of a neural network, but we are not prepared to concede these models as native, either. Neural networks able to revise their predictions as time passes, such as *time delay neural networks* [48] and BPTT (back-propogation through time) [89] would also likely get better results, but would require the agent to engage in the activity for a period of time before a prediction could be made. This is incompatible with most notions of planning, in which the agent generates several steps of behavior in advance of starting execution. We want our agent to be able to plot out an activity in advance, then execute it and store it for later reuse.

The most acceptable choice might be to take the probabilistic approach: given the initial state, use the empirical distribution of observed outcomes to suggest what *might* happen. There are examples that show artificial neural networks can be trained to produce outputs that correspond to probabilistic estimates [6]. Revising the network in this way would allow us to address the first two potential problems in our list. Likewise, the fourth problem could be addressed if we were to rerun these pilot experiences with data from the SSC clusterers. There is significant literature dealing with the choice of input and output representation as well as network structure to draw from to incrementally refine a network that would perform better than our initial experiment. Overfitting, however, appears to remain a problem for neural networks.

Ultimately, the reason we would move away from neural networks would not be overfitting or the frequent requirement that network structure be tuned and retuned to succeed on a particular problem. The primary goal of this predictive component of our system is to use the structure of the model to identify initial conditions. There has been some work on interpreting the structure of a learned neural network [34, 55], but for the vast majority of neural network learning algorithms, the primary goal is prediction accuracy. The internal structure and whether it corresponds to knowledge of how the mapping between input and output works in the real world is traditionally not a priority to the developers of networks as long as predictive accuracy is high. By contrast, we are most interested in arriving at models from which declarative intitial conditions can be extracted, and it is important that they reflect the true contingencies in the environment as closely as possible. For this purpose, dissecting and interpreting neural networks is still a technology in its early stages, and for this reason, we turn our attention towards inductive methods.

2.2.2 Relevant Work: Inductive Methods

Decision tree induction is one instance of an inductive learning mechanism that produces predictive, declarative, and easily decomposable structures. While predictiveness is normally the name of the game in machine classification, the latter two characteristics are also highly desirable in our system. Decision tree structures have naturally probabilistic interpretations as well, another key feature for our initial condition induction mechanism. Among the many variants of decision tree induction to choose from are: the trailblazer of decision tree induction, ID3 [67], the current gold standard, C4.5 [66], an incremental tree building algorithm called ITI [83], and systems that increase the expressiveness of the basic decision tree formulation such as OC1 [57] and non-linear decision trees [35].

The general decision tree induction algorithm works with data described by feature vectors and class labels, as in our description of the neural network experiement of section 2.2.1. Decision trees are built via a two-phase process. In the first phase, a fully-elaborated tree is built. The tree is built by splitting the full set of data into subsets according to a *decision*, which is a test on some feature in the training data. Decisions are generally selected from the full set of possible tests greedily according to which decision results in subsets with the greatest degree of *purity*, where purity is a measure of homogeneity in the class labels in a set of data. The process is repeated recursively on the subsets until all of the subsets are completely pure (containing only data of a single class). The result of this first phase is a tree composed of *decision nodes* that produce splits of the data and pure *leaf nodes* where branches in the tree end. In the second phase, the tree is "pruned". During pruning, leaf nodes are merged bottom-up when the loss in purity is considered acceptable. The pruning phase is intended to fight overfitting – superfluous structure in the decision tree – which can hurt the prediction performance of a tree on unseen test instances.

We use an algorithm called TBA to generate decision trees in our system. TBA has a unique treatment for the problem of overfitting in that it addresses overfitting as a symptom of a systematic problem in any induction system that makes multiple, repeated comparisons. This problem of systematic error in algorithms making multiple comparisons is explained fully in [12] and the treatment that TBA uses is described in [38]. Since we intend to interpret tree structure as initial conditions to planning operators, we have additional incentive to prevent superfluous structure from finding its way into our decision trees. The results we describe in this section, and those that follow, are all based on decision trees built using the TBA algorithm.

An example decision tree built on TURN experiences clustered with respect to the velocity sensors and bump encoders of the Pioneer-1 is shown in figure 2.9. The experiences of the Pioneer-1 in this case have been hand labeled to correspond roughly to what happened through the eyes of a human observer as the Pioneer-1 took action. These labels are used to classify the experiences into operators and as the class labels for TBA to work with. The feature sets we use are precursor sensor readings, as described in the setup for the neural network experiment in section 2.2.1. The tree in figure 2.9 may be interpreted as follows: If the direction of rotation is clockwise, the most likely outcome is an unobstructed turn, which has happened in 46% of all clockwise turns. Other outcomes are possible, though less frequent, and can not yet be differentiated by precursor data. If the direction of rotation is counter-clockwise, then the outcome depends on whether or not the gripper beams are broken (whether or not something is in between the gripper paddles). If there is, 100% of all prior experiences indicate that the robot will be stuck. This represents the situation where the Pioneer-1 has moved upon a chair or table leg that prevents the robot from turning. If there is nothing in the gripper, then one of four outcomes is likely, the most likely being an unobstructed turn, having been observed 67% of the Pioneer-1's experiences.

This decision tree reflects some of the observable structure of the Pioneer-1's environment. In particular, the agent has represented that if it happens to get something in its gripper beams (such as a table leg), it is likely to not be able to turn. It cannot yet make finer distinctions. It cannot tell a table leg from a soda can, for instance. These objects can probably be disambiguated by the sensors, but with its current data, the disambiguiating features have not yet been learned. Likewise, if it isn't already stuck (something immovable whithin its gripper), the Pioneer-1 can only express that there is a chance it might become stuck. Since the Pioneer-1 cannot sense potential obstacles if they are behind the robot, it cannot represent with certainty the conditions under which it is most likely to become stuck.

The Pioneer is faced with the task of learning initial conditions in a partially-observable, probabilistic, dynamic environment. The best this agent can do in many

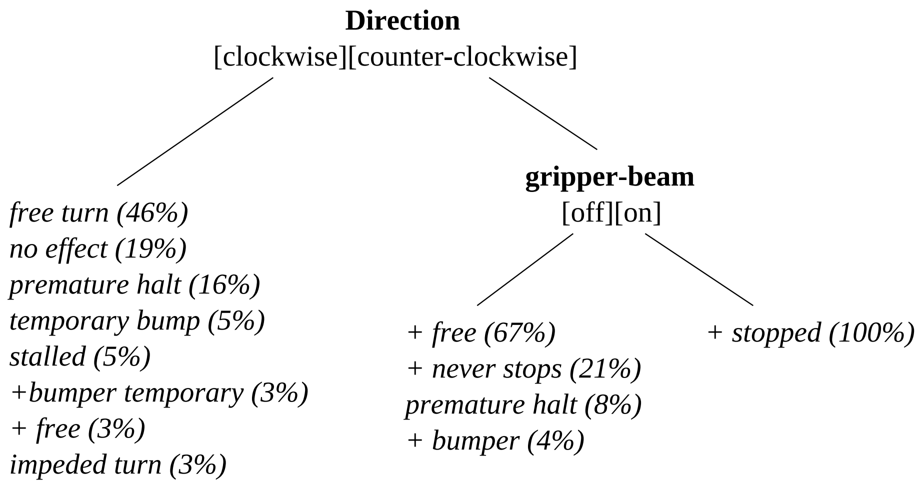

Figure 2.9. A decision tree generated using Pioneer-1 precursor phase data from clusters built on tactile data during turn experiences. Cluster labels, along with their relative frequency, are italicized at the leaves.

difficult situations is express what it has experienced probabilistically; that in 16% of prior clockwise turns, a pattern corresponding to an obstruction was encountered, or more simply, that this pattern is a possibility when turning. In other cases, the sensors of the Pioneer can make important distinctions with greater certainty. The gripper beam can distinguish a situation where free movement is most likely from one in which being stuck is most likely. A tree generated from MOVE experiences, clustered with respect to visual channel A (which tracks objects that are red in color), is shown in figure 2.13. The tree is significantly more complex than the TURN tree because finer distinctions are possible from the data, and as a result the tree is able to make stronger statements about actions and their probable outcomes. Greater differentiation is possible in this tree by using visual cues available in the precursor phase, and by making distinctions between what is possible when moving forward (objects disappearing from view) as opposed to backwards (objects appearing in view).

Whether fine distinctions can be made or only coarse ones, we have found that decision trees are better suited to our particular task than their neural network counterpart. This is because their structure is completely transparent and can be easily transformed into probabilistic propositions over the state space. It is unlikely that the performance of decision trees will be better in any or all cases, but for our purposes, it is more important that the structure of the models be easily made available to other components of our developmental system.

2.2.3 Closing the Modeling Loop

After our review of work relevant to our modeling needs, as well as some development of our own, we settled on the combination of the simple symbolic clusterer to handle outcome classification and a decision tree algorithm (TBA) to handle initial condition induction. In this section, we will describe in detail how this tandem works and how the modeling system handles its job as an oracle to the other developmental components. For clarity, we will shift gears in this section, and use the GridSim simulator as described in section 1.1.1 in a running example.

We have not described in detail either the motivational system or the planner, but assume for the sake of this discussion that the motivational system is somehow selecting goals for the planner. As a part of the *planning to act* scheme, the motivational system at times will query the modeling system to evaluate its options. The motivational system will ask the following questions:

1. What outcomes am I allowed to choose from?

2. What is the most likely outcome of action a in the state that I am in?

The reasons why the motivational system will pose these questions to the modeling oracle will become clear when we consider the motivational system in detail. In short, the motivational system simply needs to enumerate potential goals and use the answers to these questions to decide among them.

The planner, as part of its job, will also need information from the modeling system. In particular, it will ask the oracle of modeling the following question:

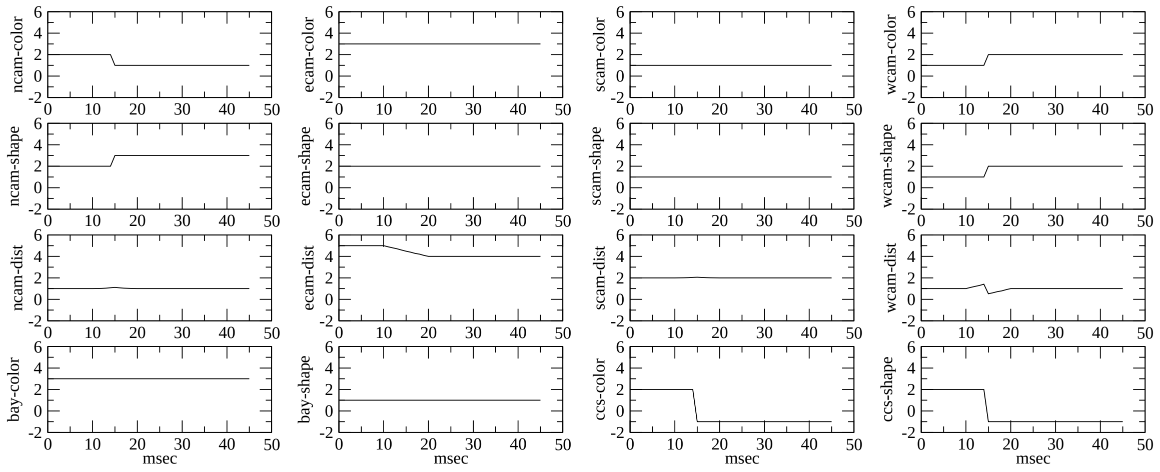

Figure 2.10. A GridSim experience: moving west.

1. What are the initial conditions for outcome *o*?

Let us consider how these three questions are answered by the modeling system in the context of a running example of an agent acting in the GridSim environment.

At the outset, the agent has no experiences, and so neither the planner nor the motivational system have any reason to influence the behavior of the agent. This will end very soon once the agent has one or two experiences, but the first few experiences of the agent are for practical purposes selected randomly. The agent picks an action, executes it, and collects the experience. Let us assume that the agent moves west. The sensor behavior for all 16 GridSim sensors for this experience are shown in figure 2.10.

Once the experience is established, the outcome classification scheme churns out a symbolic description. Suppose our agent is using the Delta-Simple filter. Looking at figure 2.10, we note that `ncam-color` goes down, `ncam-shape` goes up, and so on. The complete listing of the sensors and their Delta-Simple description, are listed in table 2.11.

A convenient shorthand for outcomes described with the Delta-Simple filter is to use a single letter abbreviation for each sensor trend, separated by dashes, in the order presented in figure 2.11. The following string is shorthand for our agent's first

```
(bay-shape nc)         (bay-color nc)
(n-camera-color down)  (n-camera-shape up)  (n-camera-dist nc)
(e-camera-color nc)    (e-camera-shape nc)  (e-camera-dist down)
(s-camera-color nc)    (s-camera-shape nc)  (s-camera-dist nc)
(w-camera-color up)    (w-camera-shape up)  (w-camera-dist nc)
(ccs-shape down)       (ccs-color down)
```

Figure 2.11. A Delta-Simple outcome description for the experience of figure 2.10.

experience. The first two characters correspond to the two bay sensors, then the north camera sensors, followed by the east, south, and west cameras, and finally, the ccs sensors:

`N-N-D-U-N-N-N-D-N-N-N-U-U-N-D-D`

At this point, the modeling system can answer the motivational system's first two questions: *what are the outcomes I have to chose from?* and *what is the most likely outcome of action a?*, so long as *a* is `move-w`. The answer to the first question will always be the set of all unique Delta-Simple labels. There is only one at this point, so it is not much for the motivational system to work with yet. The most likely outcome of `move-w` will simply be the only thing that it can predict, which is the only outcome it has observed. Answering this question, along with the others, becomes more interesting when there is more than one experience for the modeling system to work with.

Decision trees generally need 20-30 training instances to become useful in discriminating outcomes. From the motivational system's perspective, up until this point, most activities appear to be equally appealing to engage in. From the planner's perspective, operator models are hopelessly inadequate until this point, and so planning is an exercise in futility. So, for all practical purposes, the system will more or less wander, much like it did to generate its first experience, until discrimination starts becoming possible. Let us assume the agent behaves in this fashion through 100 experiences, more or less uniformly distributed among the agent's 6 actions.

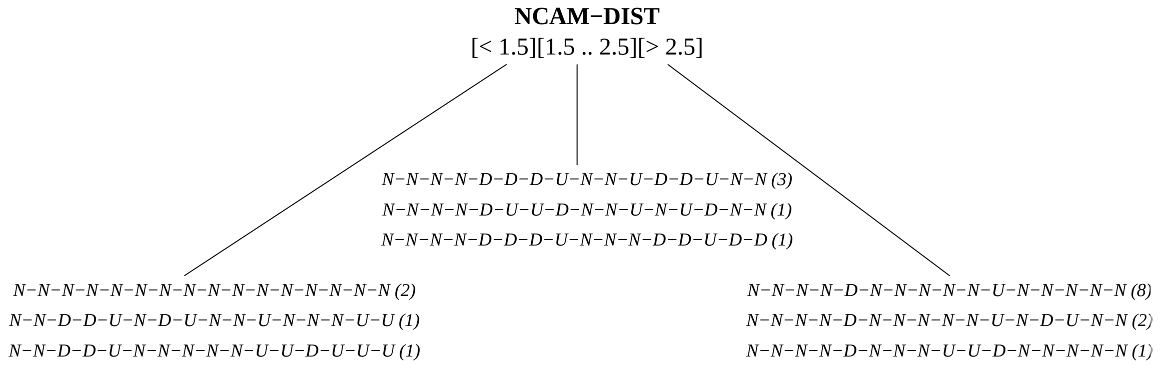
NCAM−DIST
[< 1.5][1.5 .. 2.5][> 2.5]
N−N−N−N−D−D−D−U−N−N−U−D−D−U−N−N (3)
N−N−N−N−D−U−U−D−N−N−U−N−U−D−N−N (1)
N−N−N−N−D−D−D−U−N−N−N−D−D−U−D−D (1)
N−N−N−N−N−N−N−N−N−N−N−N−N−N−N−N (2)
N−N−D−D−U−N−D−U−N−N−U−N−N−N−U−U (1)
N−N−D−D−U−N−N−N−N−N−U−U−D−U−U−U (1)
N−N−N−N−D−N−N−N−N−N−U−N−N−N−N−N (8)
N−N−N−N−D−N−N−N−N−N−U−N−D−U−N−N (2)
N−N−N−N−D−N−N−N−U−U−D−N−N−N−N−N (1)

Figure 2.12. A classification tree built on 20 GridSim experiences moving north.

A sampling of 100 experiences in the GridSim simulator might come up with 10-20 instances of each of the six actions. Suppose the motivational system asks about the most likely outcome of an action, for example, `move-n`, now. The modeling system would build a decision tree on all `move-n` experiences as described in section 2.2.2. A sample tree built on 20 such experiences is shown in figure 2.12. Note that the model already makes an important distinction: whether or not something is directly to the north of the agent before it moves in that direction. This case corresponds to the leftmost branch in the tree of figure 2.12, where the north camera spots an object less than 1.5 units away before moving. The outcomes along that branch represent the outcomes possible when moving toward an occupied cell: the first outcome represents impeded movement, and the second two represent instances of moving into a cell occupied by debris, as evidenced by the ccs sensor going up (the last two symbols in their description are `U`). However, in total there are only 4 experiences of this type, and so the modeling system cannot differentiate moving into a wall from moving over debris at this time. Once it has more instances down this path in the tree, it will split these experiences, most likely using the `ncam-shape` sensor to differentiate hitting a wall from moving over debris.

Using this tree, the modeling component can predict the most likely outcome. It does this by following the path through the tree matches the current state. If the sensor `ncam-dist` was reporting a value of 4, for example, the current state would correspond to the rightmost branch in the tree. Therefore, the modeling system would be able to select the most prevalent outcome in that leaf as the most likely outcome. In this case, the result is the outcome in which `ncam-dist` goes down, `scam-dist` goes up, and no other sensors change:

```
N-N-N-N-D-N-N-N-N-N-U-N-N-N-N-N
```

When trees are built on relatively little data, as in our tree of 20 experiences, and often in domains where outcomes are probabilistic or difficult to disambiguate with the available sensors, the correct move for the modeling system is to predict the outcome which has been observed the most in the current context. But these trees also allow the modeling system to make a different kind of prediction. Rather than asking for the most likely single outcome, the modeling system can make sensor-by-sensor predictions. Take, for example, the rightmost branch of figure 2.12. Suppose that a subsystem like the planner is interested only in what will happen to the current cell sensor, represented by the last two symbols in each experience description. In every instance in this leaf, the ccs shape and color remain unchanged. The modeling system is relatively certain that in this context, the ccs will remain unchanged. More certain, in fact, than it is that the outcome `N-N-N-N-D-N-N-D-N-N-N-N-N-U-N-N` will occur. In fact, the bay sensors never change, the north camera shape, color, and distance sensors report `N`, `N`, and `D` 100% of the time as well. Using this information, the modeling system can produce a *generalized outcome* as its prediction. The generalized outcome that corresponds to the rightmost leaf, if we only include sensors that have always been observed to change the same way, is as follows:

```
N-N-N-N-D-N-N-N-?-?-?-N-?-?-N-N
```

This generalized operator asserts that the bay and ccs sensors do not change. The north facing camera shape and color, similarly, are not expected to change but the distance to the north is projected to decrease. The east camera will show no changes, and the color to the south will not change. The west camera and south camera shape and distance vary from experience to experience in the leaf, and so in the generalized operator, these are marked with `?`, indicating that these sensors are for practical purposes unpredictable.

So, these trees can answer the question "what is the most likely outcome of action a, now, in two ways. It can predict a fully instantiated outcome, along with an estimated frequency of the prediction given the current state. Alternatively, the modeling system can make sensor-by-sensor predictions that meet a minimum probability threshold in the form of generalized operators.

These same trees can be used to generate initial conditions for arbitrary outcomes. Extracting the initial conditions for some outcome o from a decision tree works as follows. First, find each leaf that contains one or more matches for o. For each such leaf, build a proposition by tracing a path to the root of the decision tree and taking the conjunction of all the decisions along that path. The initial conditions for o are taken to be the disjunction of each path through the tree, each of which corresponds to a conjunctive clause.

A simple example can be taken from the tree of figure 2.12. Suppose we want the initial conditions for `N-N-N-N-D-N-N-D-N-N-N-N-N-U-N-N`. That outcome can be found in the rightmost leaf of the tree (and in no other leaves). The path back to the root contains a single decision, and the initial conditions based on this tree, at this stage of development, would be:

$$(\texttt{ncam-dist} > 2.5)[72\%]$$

This simply states that if `ncam-dist` is greater than 2.5, there is an estimated probability of 72% that the desired outcome will occur. A considerably more complex set of initial conditions can be extracted from a tree built on Pioneer-2 MOVE data. Figure 2.13 shows an initial condition tree for sample data taken from the Pioneer-2,

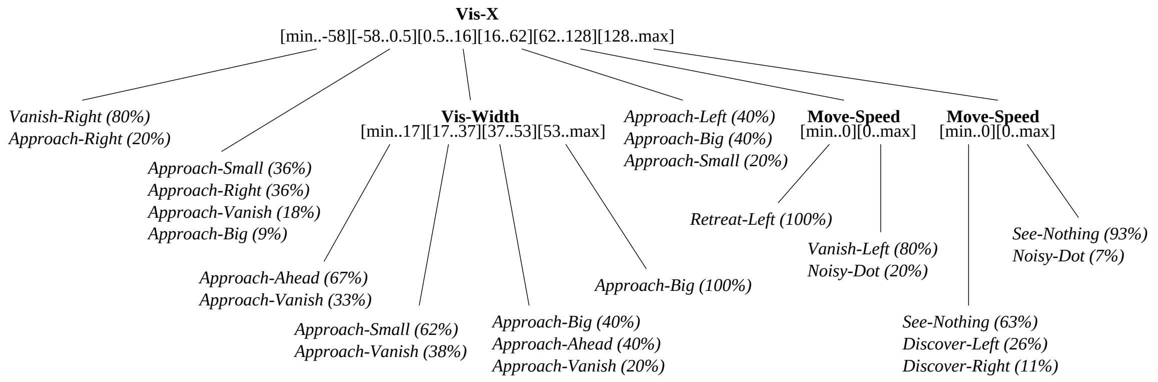

Figure 2.13. A classification tree generated using precursor phase data from the Pioneer-2.

and labeled by hand (not by an SSC filter). Suppose the planner asks for the initial conditions for the outcome *approach-big*, an experience which an object is approached until it consumes almost the entire visual field. This outcome appears in 4 leaves of figure 2.13. Its initial conditions are thus the disjunction of each path leading from the leaves to the root:

$$(\text{vis-x} \in [-58\ldots0.5])[9\%]\vee$$
$$((\text{vis-x} \in [0.5\ldots16]) \wedge (\text{vis-w} \in [37\ldots53]))[40\%]\vee$$
$$((\text{vis-x} \in [0.5\ldots16]) \wedge (\text{vis-w} \in [53\ldots max]))[100\%]\vee$$
$$(\text{vis-x} \in [16\ldots62])[40\%]$$

It is worth noting that these initial conditions offer the robot a potential tradeoff for the purposes of generating behavior. The third disjunct offers the best observed probability of success, at 100%, but its conditions may be difficult to achieve. Conversely, the second and fourth disjuncts offer a lower probability of the outcome in return for conditions that may be easier to satisfy.

Interpreting classification trees in this way gives the modeling system a way to answer the last remaining question of the three we identified at the beginning of this section: *what are the initial conditions for o?* What if *o* is a generalized outcome, though? That is, what if the planner asks for the initial conditions for the GRIDSIM outcome

`N-N-N-N-D-?-?-?-N-N-U-?-?-?-N-N`

This outcome specification corresponds to a situation in which the agent approaches an object to the north. Since the east and west cameras do not factor into this outcome, they are left out of the specification. Notice that in the tree of figure 2.12, experiences that match this generalized outcome profile sit in each of the three leaves. The initial conditions for this profile, built on that tree, will be particularly unwieldy:

$$(\texttt{ncam-dist} < 1.5)[50\%]\vee$$
$$(\texttt{ncam-dist} \in [1.5\ldots2.5])[80\%]\vee$$
$$(\texttt{ncam-dist} > 2.5)[91\%]$$

We call the scheme that produced the decision tree of figure 2.12 a *global* induction scheme. We call it a global scheme because it tries to represent all the possible outcomes of an action in a single tree. It has no impetus to try to keep experiences that are partial matches, like those in which `ncam-dist` goes down, together. The decision tree induction algorithm has no impetus to keep any two arbitrary experiences together if it can improve the overall purity of the tree by separating them, even if they are an exact match.

The global scheme makes sense for prediction tasks, then, but is perhaps be less than perfect for initial condition induction, a situation where local information could be used to keep experiences of interest together. Our *local* induction scheme labels experiences with the symbol `match` if their SSC description matches a target profile, and `non-match` if it does not. Furthermore, the local scheme is not tied to any particular action. Since the question is *what are the initial conditions associated with o*, it is not necessary to fix the action. The local scheme can respond with conditions that include, among the sensory conditions, the action that must be taken. Thus the local scheme builds its trees using all available experiences, and includes an additional feature called `command` that can be mixed in as an initial condition.

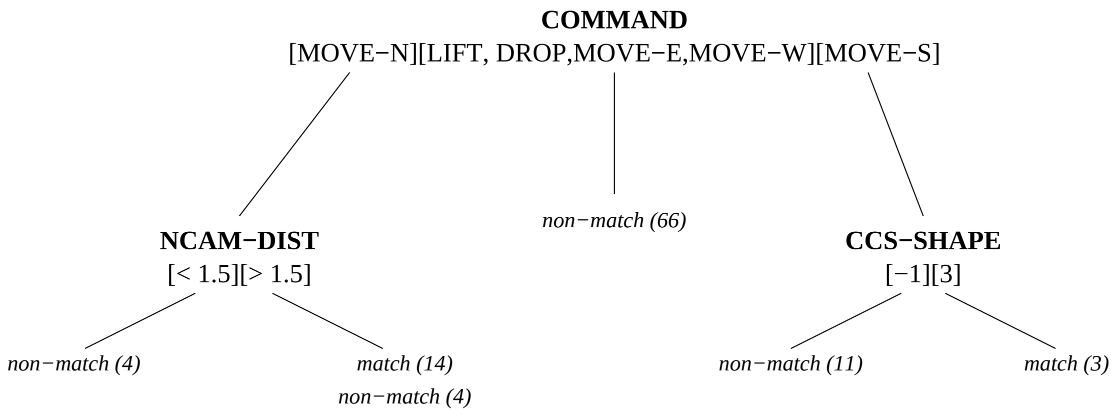

Figure 2.14. An initial condition tree generated using the local scheme on for the GridSim outcome `N-N-N-N-D-?-?-?-N-N-U-?-?-?-N-N`.

An initial condition tree built using the local scheme is pictured in figure 2.14. The tree is built on matches to the profile `N-N-N-N-D-?-?-?-N-N-U-?-?-?-N-N`. Note that the left subtree, corresponding to experiences with the action `move-n`, is a more general version of the tree built on the global scheme. For this more generalized outcome, the distinction between `ncam-dist` below and above 2.5 is no longer relevant to discriminating outcomes. The remainder of the tree is devoted to expressing that there is an alternative to moving north in order to produce the desired outcome. If the current cell sensor is reporting a shape with the code 3 in the current cell, then moving south will also achieve the desired outcome. The initial conditions using the local scheme are:

$$((\texttt{command} = \texttt{move-n}) \wedge (\texttt{ncam-dist} > 1.5))[78\%]\vee$$
$$((\texttt{command} = \texttt{move-s}) \wedge (\texttt{ccs-shape} = 3))[100\%]$$

The use of local information allows the initial conditions to be more concise, with higher estimated probabilities, and also allows the modeling system to identify alternate activities that can achieve the goal outcome.

CHAPTER 3
GENERATING ACTIVITIES

The core question of this dissertation is, *how can an agent compose primitive actions into sophisticated activities that achieve goals?* In section 1.3, we revealed that our system would be based on means-ends analysis and planning, but that many technologies were appropriate to the task. Among the lines of AI research that have applications in building behaviors are reinforcement learning, genetic algorithms and planning. Each of these schemes specifies behavior in its own way: reinforcement learning produces *policies*, genetic programming produces *control programs*, and planners produce *plans*. Each encodes either explicity or implicitly a sequence of operations designed to achieve a goal. We begin this chapter with a review of these three technologies. We will review some of the relevant systems in the literature and motivate our eventual selection of planning as the core technology of our developmental system. We will follow with a detailed description of our planner and an example of how our planner operates on the GridSim domain.

3.1 Related Work

Many algorithms designed for generating or learning behavior can be thought of as a search through a space of action sequences for one or more such sequences that achieve a goal. The size of the sequence space is going to be exponential in the length of the largest allowable sequence: if there are n actions and the longest legal sequence is m actions, the space of possible action sequences is n^m. Algorithms that effectively negotiate such a space rely on a core idea or set of ideas that somehow make search feasible. Innovation may occur in how the space is searched: more efficient searches allow the space of action sequences to be searched more thoroughly, for example, and thus allow for more robust or effective sequences to be found. Alternatively, the innovation may be in how the space of sequences is represented: a more succinct or expressive representation of an action space may also allow for more robust sequences to be generated more quickly than a more cumbersome one. With this in mind, there are three grounds on which we may judge a technology more or less appropriate for our task. First, it is preferable for an algorithm to conduct its search in a way that is efficient or robust enough to give that technology a clear cut advantage over others in its class. Clearly, effectiveness is a primary concern for any system we choose as a basic developmental mechanism in our system. Second, an algorithm would be especially well suited to our project if its representation language were agreeable to downstream learning components like language acquisition and concept formation. Finally, since we are interested in the nature of development, algorithms that have appeal on philosophical grounds are preferred. If there is evidence that a particular process exists in developing humans or animals, we must consider that evidence. We now look at genetic programming, reinforcement learning, and planning with these three criteria in mind.

Genetic programming refers to a class of algorithms in which action sequences, defined as computer programs (such as Lisp functions), are built up in a manner inspired by Darwinian evolution. The genetic programming recipe for a search through the space of all programs is as follows: First, generate a population of random programs. Programs are generated randomly from a grammar that will produce legal (but not necessarily sensible) programs. Next, evaluate each member of the population according to some *fitness criteria* that essentially rates each program according to how closely it approximates a target behavior or output. Third, carry out a *breeding cycle.* In the breeding cycle, pairs of programs are selected either randomly or weighted according to the programs' fitnesses. Program pairs are then combined using special operators for "mating" programs (which typically amount to splicing together pieces of the two mates) to produce a new generation of programs. When the breeding cycle completes, some of the older generation may be deleted from the population, and some retained. The cycle of breeding followed by evaluation continues until the overall population's fitness levels off. Generally, the most fit program or set of programs is taken as the output of the algorithm.

A significant literature indicates genetic programming is capable of solving problems in which a brute-force search through the space of programs is prohibitive. Koza [43] surveys several successful applications using genetic programming to produce programs written in a subset of the Lisp programming language that solve simple control problems, such as the cart-centering problem, in which a rolling cart is brought to halt in a target location through the application forces to the cart's sides, and ant foraging, in which optimal monitoring strategies evolve for an ant with limited sensory capabilities trying to find and consume food in a maze.

Although genetic programming has applications in control and is often considered as a solution to machine learning tasks, there is a fundamental difference between evolution and development. The key distinction is that evolution is a process that works over populations, and development is a process over individuals. A species evolves a mechanism that makes grasping possible, an individual develops a familiarity of how to control the mechanism reliably. There is strong evidence that evolution is indeed a real process, and thus genetic-programming is well motivated. However, there is no evidence that an evolutionary process underlies the development of an individual. Rather, the lengthy, computationally intensive, and largely random process that it genetic programming simulates presents feasibility issues for a single situated agent. How are the hundreds of generations of programs to be evaluated? Is there not a more focused search procedure than natural selection that would substitute good heuristics for a randomized generate-and-test process performed on large samples?

Genetic program seems not to hit the mark on two of our criteria: effectiveness, for it's heavy computational demands, and evidence in favor of its existence as a natural process in human or animal intelligence. Perhaps the most worthwhile idea of genetic programming for our purposes would be that a programming language is a natural choice for representing the activities of a mobile robot, for several reasons: Programs are readily executable by machine and generally readable by humans. Their structured, declarative nature also simplifies the task of later learning components like concept and language learning. One can envision forming concepts around `if-then` clauses of activities; the construct (`if` (vis-a-width $<$ 100) `then acquire`) could be interpreted as conditions for the concept of a *graspable* object, for example, as indicated by the applicability of the `acquire` procedure. Still, the computational impracticalities of genetic programming seem to outweigh the attractiveness of its output for our purposes.

This leads us to our second candidate technology for creating action sequences, reinforcement learning (RL). The idea behind reinforcement learning is that the interaction between an agent and its environment consists of not only sensory information, but also positive and negative reinforcement called rewards and penalties, respectively. Reward signals, in RL, are the basis of an agent's behavior; agents act to maximize rewards.

In order to maximize reward, a reinforcement learning agent must model the behavior of the reward signals it receives as it interacts with its environment. RL agents generally model rewards with a *value function* (Watkins' Q function [88], for example) that approximates the expected reward of taking a given action in a particular sensory context. The value function then serves as a basis for an action-taking *policy*, which typically amounts to hill-climbing on the surface of the value function, or greedily selecting the action that maximizes the expected future reward given the value function in the current context.

The key to a good policy is to learn as close an approximation of the environment's actual reward function as is possible given the sensory apparatus of the agent. The Bellman equations provide a mechanism for gradient descent learning of such an approximation incrementally, through situated activity and observing the resulting rewards [3]. Using the Bellman equations allows a learning agent to approximate the long-term rewards associated with action-taking, and thus produce a policy which maximizes rewards not only locally, but over the long term. The Bellman equations are guaranteed to converge in certain situations [17, 36], and thus as an RL agent acts in its environment, its ability to achieve reward improves. In practice, reinforcement learning works on a wide range of applications, even in some where it is not known if convergence is guaranteed. Some successful applications include simple control tasks like navigating a simple race car in the *racetrack problem* [27] to learning routing tables for TCP networks [7]. Others have extended the technology to more complex tasks, including robotic foraging and exploration tasks [52], [50]. The DYNA system extends reinforcement learning by introducing action modeling to speed up learning [79]. Still other work has approached the task of reinforcement learning in hierarchical actions spaces with mixed results [82], [76], [39].

Our first criterion for a good activity-generating component is effectiveness, and RL seems to perform well at control tasks. However, there would be a handful of hurdles to clear to make it work in our system. These hurdles are encountered in every RL system, and chief among them is that reward functions have to be implemented by the system designer. Reward functions can be very complex or very simple, but in the end the RL agent is doing hill climbing on the value function, which is an approximation of the reward function. Thus, the reward function indirectly determines the eventual behavior of the agent. How would an agent produce and approximate its own reward functions? Can one specify a universal reward function that would allow an agent to choose and learn the behaviors it needs? Furthermore, we would need to decide on a value function. Many early RL systems used a lookup table to associate state-action pairs with rewards. Once the RL paradigm showed promise, researchers began to apply it to increasingly difficult tasks, and as state spaces became larger and larger, lookup tables became increasingly inadequate. RL researchers began using various function approximation techniques, including artificial neural networks, to approximate the reward function mathematically. These approximations allow RL techniques to scale to more difficult problems, but in so doing, the representation used by RL systems becomes less well-understood. As a result, effectiveness, our first criterion for a good developmental basis comes at the cost of our second criterion of being a representation that is amenable to downstream development.

Reinforcement learning is well-motivated in accordance with our third criterion for a good developmental basis. The ideas underlying reinforcement learning stem from evidence that reinforcement plays a significant role in animal learning [77]. While it is indisputable that reinforcement plays a role in *what* an actor will do, it does not necessarily follow, though, that it influences *how* the rewards are achieved, as it does in RL. That is, reward signals may determine the goals of an agent while not actually influencing in any direct way the steps that leads up to the goal. The leap made by reinforcement learning, and indeed the innovative contribution of RL, is that the reward signal simultaneously determines what an agent seeks to do and how it seeks to get it done. While many systems succeed in this paradigm, it is not fully supported by what we know about developmental psychology that a reward signal determines both what and how goals are achieved. In other words, it may be possible to leverage the motivation behind RL – that reinforcement drives behavior – without using the underlying learning algorithms.

The effectiveness of RL, coupled with its well-motivated background, prompted us to run some preliminary experiments. These experiments frequently resulted in pathological behavior that we could not easily understand. The solution generally required tinkering with the reward structure to produce more acceptable behavior. Our experiments with hierarchical reinforcement learning proved equally frustrating and often turned into exercises in experimenter intervention, requiring a great deal of tinkering with state spaces and reward signals to produce acceptable results. These pilot studies, some of which are documented in [82], suggested that tinkering might be necessary each time our agent wanted to learn a new task. In addition, the sometimes opaque and complicated function approximators used by RL create challenges to downstream components like language and concept learning. Because the behavior is only implicitly represented by the value function, the structure of activity is hidden; even if concepts like *graspable* might be extracted from policies, they certainly do not fall out as simply as they might from an explicit representation.

Since the time of our initial foray into RL, there has been considerable effort placed on the use of hierarchical control structures, generalization, and abstraction in reinforcement learning policies [18, 32, 54]. Many of these approaches are relevant to the difficulties we encountered in our pilot studies, and we feel that in all likelihood, RL could be made into an adequate basis for development in a system like ours. In the end, though, we did explore a third candidate that we felt was superior according to two of our three criteria.

Planning is the third family of algorithms we reviewed to handle the problem of producing action sequences in our developmental system. The foundation of traditional AI planning dates back to 1963, when the idea of means-ends-analysis was first brought to bear in a published AI system. [59] The General Problem Solver (GPS) attempted to solve problems by means-ends reasoning. Given a representation of a problem state, a set of operators, and complete models of the operators' effects, GPS would solve problems by first identifying the differences between a goal state and its current state, and then selecting operators whose effects would resolve differences. Because some operators themselves have preconditions, the selection of an operator might introduce more differences, which would be resolved by recursively running GPS on the new set of differences. Once all differences are settled, this algorithm will have constructed a solution: a sequence of operators that, when applied in the initial state, bring about the goal state. This form of reasoning backward from a goal state to a system's current state, reducing differences as one goes, is known as *backward-chaining*, and is the underpinning of what we refer to as classical generative planning.

Since that time, some weaknesses of basic planning approach to complex tasks have been identified, such as the intractability of the general planning problem (probabilistic STRIPS planning is EXPTIME-complete [49]) and pathology called Sussman's anomaly, where planning operators and subgoals can interact and cause problems for linear planners [78]. Extensions to the basic algorithm have been made, to address these problems and others: planners starting with ABSTRIPS soon made use of hierarchical and abstract operator representations. Nonlinear planning was introduced as a solution to Sussman's anomaly [80]. Gaps between the simple propositional worlds of GPS and continuous, complex environments of mobile robots were bridged by ambitious projects like PRODIGY [9], and the idea of operators with temporal extent has been addressed by the ZENO planner [64, 84]. Still other planners integrate planning with learning (PRODIGY, FOR EXAMPLE. Some such systems attempt to learn heuristic information to assist the planner in choosing the best plan when several are possible in a given situation, as with the ROGUE module of PRODIGY [31], while others contain functionality for learning propositional operator models [87].

State of the art planning meets our system's criteria quite well. One need only go as far as the PRODIGY architecture for an example of a planner that can operate efficiently in a complex environment. This system has been implemented on mobile robots operating in real-world situations with good success. The utility of PRODIGY in complex domains indicates that planning meets our first criterion for a technology of our system: it works and can be scaled up to manage difficult tasks. The representations used by typical planning technologies are also well suited to our application. Plan structures are naturally compositional and hierarchical in nature. Typical planning languages are also declarative, easily interpretable by computer programs or human analysts. Like the structured programs that genetic programming produces, plans are more easily analyzed for later learning than implicit representations like those produced by reinforcement learning. Finally, there is clear evidence that intelligent beings engage in means-ends analysis and planning type processes to solve problems. This lends credibility to the idea that means-ends analysis might be the basis for the development of new activities.

This is not to say that an off-the-shelf planning system is the answer to all of our problems. While some of the aforementioned planners are significant advances toward our cause over GPS, no single system seems to address all the unique demands of our developmental system. While planning has been applied with success to robotics, it is still common practice to engineer perceptual spaces and sophisticated planning operators to make the complex tasks these robots plan for possible. While the kinds of things our agents will do are in most cases less sophisticated than what many modern planning agents do, reasoning in a continuous sensor space with learned operators presents a significant challenge. Nevertheless, the benefits of the representations used in planning, coupled with the strong evidence of planning in intelligent agents do suggest that planning is the strongest candidate of the three.

3.2 Planning as a Basis for Development

Planning is one of the longest-lived technologies in the field of AI research. At the time of this publication, more than 40 years have passed since GPS was first introduced. As such, it is not surprising that there are many variations on the basic planning formulation to choose from, and many ways in which these variations distinguish themselves from each other. Before we describe how our developmental system uses planning, it will be helpful to draw a distinction between two major categories in planning. The distinction lies in how planners produce plans, and divides the world of planning systems into two basic categories: generative planners and skeletal planners.

Generative planners are those that literally generate plans from scratch by stitching together sequences of planning operators to get an agent from a starting state to a goal. All of the planners cited in the literature are primarily generative planners in that they compose plans from operator models. Generative planners rely heavily on these planning operators, and the models must correctly specify operator preconditions (or initial conditions) and postconditions in order for the planner to create successful plans. Throughout the planning process, models are used to identify useful operators in the means-ends analysis step and to verify the correctness of candidate plans.

Skeletal planners, by contrast, select plans from a library of pre-existing templates. Templates can be retrieved according to the goals they achieve, and may be ready-to-use plans or only partially *instantiated*. Partially-instantiated plans leave details undefined and require the planner to fill in these details prior to execution. Often, a partially-instantiated plans will break down a single goal into a series of *subgoals*. When subgoals are encountered, the skeletal planner is recursively invoked to select from the library an appropriate template. This naturally hierarchical process bottoms out when the planner reaches into plan templates that operate at the level of actions, with no details to fill in. Some skeletal planners allow variables in partially-instantiated plans. Variables express the roles that objects or resources in the environment will play in the unfolding of a plan and allow plans to be more general by allowing the planner to instantiate the variable values prior to execution by selecting objects or resources that will make the plan work. The name *skeletal planning* stems from the idea that plans are specified with as many details left undecided as is practically possible. Plans are stored as "skeletons", and the details of their execution must be decided when the plan is retrieved.

One class of planners that straddles this line of distinction is called *case-based planning*, and provides for the storage and retrieval of ready-to-use or skeletal plans. Many case-based planners take plans created by a generative process and store them as skeletal plans, allowing a planning system to resort to generative planning when it is necessary, and skeletal planning when appropriate plans are available.

To this point, we have focused exclusively on generative planning. A generative component is required to explain where the beginnings of activity come from in a developmental system. Generative planners are able to produce novel sequences of behavior from in situations, where skeletal planners are limited to working with a pre-existing library. On the other hand, generative planners in their most basic form always start from scratch. There is no persistent notion of an *activity* at all if the agent is always generating behavior from scratch. Our intent in introducing skeletal or case-based planning at this point is to suggest that in a sensible developmental scheme, generative and skeletal planning work in tandem. Successful plans may be cached in a library for later use via skeletal planning, the less computationally taxing of the two approaches to planning. In novel situations, where no appropriate cached plans exist, an agent resorts to generative planning. Combining a simple plan-caching scheme with a generative planner is not a new idea; the STRIPS planner implemented a macro learning component to do this in one of its early incarnations [22] and the SOAR planner includes a "chunking" mechanism for doing just this [45].

Our system is built to utilize both generative and skeletal planning. The generative component is the centerpiece of the activity generation process, the skeletal component saves computation time and maintains structures and statistics for downstream learning. In the section that follows, we discuss the generative component of our system and include examples of its operation in the GridSim domain. We conclude this chapter with a description of the simple plan-caching scheme we use to handle storage, reuse, and removal of generated plans.

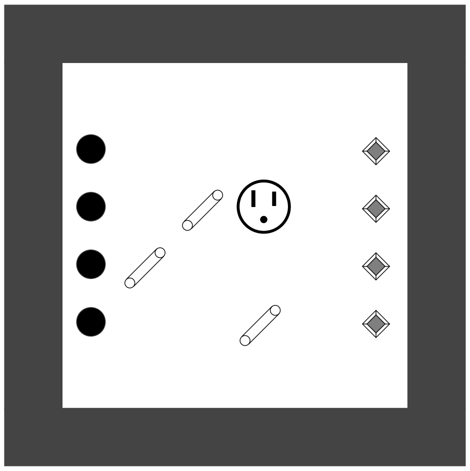
Figure 3.1. A rendering of the GridSim simulator.

3.2.1 Generative Planning

Our original concept for the generative planning component of our developmental system was to build a planner in the GPS or STRIPS mold and adapt it to the requirements of operating at the sensorimotor level. The operation of our proposed planning system is best described with an example in the GridSim domain.

For our discussions on planning, we will assume that our agent has sufficient models to build good models for planning. As we will show in the evaluation, at 2000 experiences, a GridSim has models that are generally accurate and stable. Our system selects outcomes as its goals, in accordance with the philosophy of planning to act, as introduced in section 1.3. Outcomes, and goals, are processed using the Delta-Simple filter, and we will refer to them in this discussion using the shorthand notation introduced in section 2.2.3.

Suppose that our GridSim agent is sitting just off the center of the 8× grid, as in figure 3.1. A typical goal outcome might be described in the Delta-Simple shorthand as follows:

```
U-U-N-N-N-N-N-N-N-N-N-N-N-N-D-D
```

Recall the semantics of the Delta-Simple shorthand. In the above outcome, the bay shape and color sensors both increase (denoted by `U`) and the ccs sensors both decrease (denoted by `D`). This outcome corresponds to lifting a piece of debris into the cargo bay. The bay sensors reflect the introduction of a new object, and the current cell sensors reflect the object disappearing from the current cell. One might call this outcome *lift-debris.*

For a human observer, generating a plan to get the agent from where it is pictured in figure 3.1 to the above outcome is simple once the semantics of the goal outcome are known. The goal outcome is simple to achieve, and can be achieved by a variety of plans., The simplest plan achieves the goal in only two steps.

1. `move-w` until an object appears in the ccs sensor

2. `lift`

How would a means-ends analysis (MEA) planner approach this goal? Traditional MEA planning starts by looking at a goal state and comparing it to the current state. Since we do not work with goal states under the planning to act scheme, our goal is an outcome, and not a state. Our planner must start by getting the initial conditions of the goal outcome from the modeling system. Here are the initial conditions for o_{lift}, taken from the initial condition tree shown in figure 3.2:

$$((\texttt{command}{\in}\{\texttt{lift}\}) \wedge (\texttt{ccs-shape}{=}\ 3))[100\%]$$

The system has only found one way to produce the desired outcome: to execute the `lift` command when the ccs shape sensor reports an object with shape code 3 in the current cell. The observed probability of the desired outcome given the above initial condition is 1.0. The MEA planner would identify the goal state as (`ccs-shape`= 3).

The next step is to compute the difference between the current state and the goal state. In the agent's current state, as pictured in figure 3.1, the current cell sensor detects nothing in the current cell, reporting a value of −1. The difference is

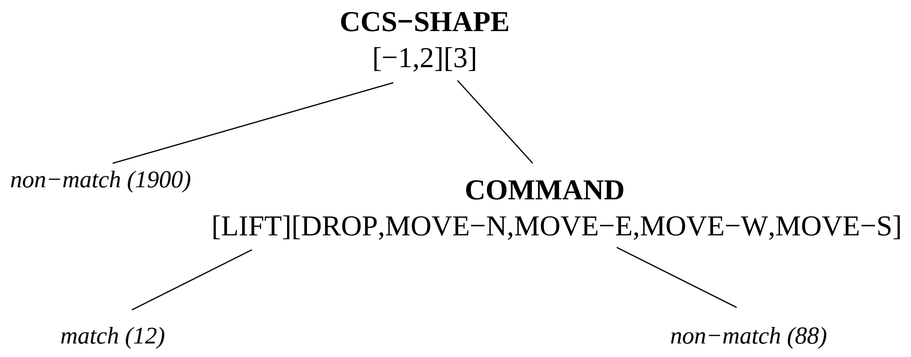

Figure 3.2. An initial condition tree built for the *lift-debris* outcome.

noted and added to a *goal stack*. The goal stack maintains a list of the outstanding differences between the agent's state and the necessary conditions for plan success at various stages of the planning process. The goal stack is used to verify whether or not a plan is *valid*. A plan is valid if when it is executed, each step in the plan will unfold as predicted by the planner, ultimately leading to the goal.

The next step of the planner is to search for operators that can resolve the outstanding conditions on the goal stack. Planners that work at the symbolic level simply select out operators that resolve one or more outstanding conditions by matching operator postconditions against the goal stack. This process is complicated by working at the sensorimotor level. A sensorimotor agent does not have operator models that state, "changes the value of `ccs-shape` to 3". They have experiences, with time series sensor readings, and discrete interpretations of the behavior of those sensor readings (provided by the SSC classifier). How could sensorimotor models be used to select appropriate plan steps? Here are two possibilities:

- Use the raw sensor readings and search for *goal crossings*. Goal crossings are points in the time series where outstanding goal conditions, like (`ccs-shape`= 3), are met.

- Use the SSC classification to select outcome classes that drive sensor values toward goal values. In our example, where the goal condition is (`ccs-shape`$=$ 3), and in the current state (`ccs-shape`$= -1$), the planner would search for outcomes that include regions where `ccs-shape` increases.

Both of these schemes have their drawbacks. The goal crossing technique is computationally expensive, requiring the planner to pore over time series, point by point. The second scheme works under the assumption that an operator that drives a sensor in the right direction can actually induce a goal crossing. This is not always true. Our `ccs-shape` goal offers a fine example of how this assumption can be violated when using the DELTA-SIMPLE filter. Consider two experiences, one in which a piece of debris with shape code 3 comes into view of the west-facing camera, and one in which a dropoff location with shape code 2 comes into view. Both experiences will be classified by the DELTA-SIMPLE filter as showing an increase in the `ccs-shape` sensor, but only one of them can actually achieve the goal of driving `ccs-shape` to the specific value 3. The assumption that driving a sensor in the right direction can achieve the desired state is easily violated. Because situations like this exist throughout the GRIDSIM domain and presumably others, the trend-based scheme cannot be counted on, and the goal crossing scheme should be preferred in spite of its heavy computational cost.

There are 82 experiences in our set of 2,000 that show a goal crossing at (`ccs-shape`$=$ 3). These consist mainly of movement actions where the agent moves over a debris object, as we would expect, but also `drop` actions in which debris is dropped from the cargo bay into the current cell. The modeling system can be queried for the initial conditions on these 82 experiences. The resulting tree is fairly complex, consisting of 15 decision nodes and 19 leaves. Of those 19 leaves, 7 contain positive instances of the goal crossing, and 5 of those 7 leaves show a probability of a goal crossing above 1%. The initial conditions above the 1% threshold are shown in figure 3.2.1.

$$((\texttt{command}\in\{\texttt{move-w}\}) \wedge (\texttt{wcam-dist}< 1.5)) \wedge (\texttt{wcam-shape}= 3))[100\%]$$
$$((\texttt{command}\in\{\texttt{move-w}\}) \wedge (\texttt{wcam-shape}\in \{2,3\}) \wedge (\texttt{wcam-color}= 3) \wedge (\texttt{ecam-shape}= 2))[100\%]$$
$$((\texttt{command}\in\{\texttt{move-e,move-n}\}) \wedge (\texttt{ncam-shape}= 3)) \wedge (\texttt{ecam-shape}= 3)) \wedge (\texttt{wcam-shape}\in \{2,3\}))[100\%]$$
$$((\texttt{command}\in\{\texttt{move-s,move-e}\}) \wedge (\texttt{ncam-shape}= 1)) \wedge (\texttt{ecam-shape}= 3)) \wedge (\texttt{wcam-shape}\in \{2,3\}))[64.5\%]$$
$$((\texttt{command}\in\{\texttt{move-s,move-n}\}) \wedge (\texttt{ecam-dist}= [2\ldots 3])) \wedge (\texttt{ecam-shape}= 2)) \wedge (\texttt{wcam-shape}\in \{2,3\}))[14.5\%]$$

Figure 3.3. Initial conditions for experiences crossing the goal (`ccs-shape`= 3).

Initial conditions with the highest observed probability are listed first. The first two are conditions in which `move-w` are predicted to bring about a goal crossing at (`ccs-shape`= 3). They both have an estimated probability of 100%. The first condition set is most interesting, and most intuitive. It states that if you move west with the west-facing camera reporting an object with the debris shape code, and the distance to that object is 1.5 or less, the agent will notice the debris appearing in the ccs sensor. This corresponds to the state of the agent in figure 3.1. The second condition set includes the `ecam-shape` sensor and removes the `wcam-dist` restriction. The number of instances supporting this leaf, however, is low. In these situations, an agent's confidence in this leaf should be low to reflect the dearth of data supporting it. The third condition set is interesting in that it includes conditions on three shape sensors oriented in different directions. In this grid world, noting the shape in three of the four cardinal directions is sometimes enough to accurately predict the exact cell that the agent is sitting in. If the agent knows its exact location, and the debris objects are fixed in their locations, then accurate predictions can be made. In some cases, the decision tree induction process will key on these "landmark" situations that allow it to bring a leaf in the tree into one-to-one correspondence with a cell in the GridSim domain. The third initial condition set above is an instance of this tree-building behavior.

The fourth and fifth condition sets are less interesting in that they are not 100% pure. Usually this is a result of the decision tree induction algorithm being unable to pin down the exact conditions for a given type of outcome. Consider that in this case, there are only 82 of 2000 experiences corresponding to our target behavior, and those 82 instances are mixed among 5 different possible actions, each of which has a distinct initial condition set. With more positive instances, the initial condition sets for the experiences that the last two conditions represent should evolve to look increasingly like the first set listed above, with 100% observed probability and conditions that correspond to real structure in the environment.

While it may seem obvious to us that simply selecting the first set of initial conditions and running with them provides an easy solution for our agent, rarely is it so simple and never is anything obvious to a planning algorithm. Instead, when confronted with a decision, the usual course of action for a planner is to treat the situation as a search for the best of the possible decisions. Our planner might build a tree to organize the search, as shown in figure 3.4. At the root of the tree is the ultimate goal: the *lift-debris* outcome. The children of any node in this type of plan tree are the initial conditions for operators that satisfy the conditions for their parent. Since there is only one set of initial conditions for *lift-debris*, there is only one branch coming out of the *lift-debris* node, leading to the node labeled "IC". But there are five sets of initial conditions that might lead to the goal crossing with (`ccs-shape`= 3), and so there are five children to the node labeled "IC" in the tree.

Paths through the tree correspond to plans. Each node in the tree has a unique copy of the goal stack that can be used to determine which nodes appear to be complete solutions or are more or less promising than the others. As long as there is no node in the tree with a satisfied goal stack, the planner continues to search by selecting a node to *expand* by computing its initial conditions and creating children for them. The search can be conducted by any one of many tree-search algorithms: breadth-first, depth-first, beam-search, and so on. Once there is a leaf with a completed goal stack, that leaf can be used to produce a plan. A goal stack is complete if its only

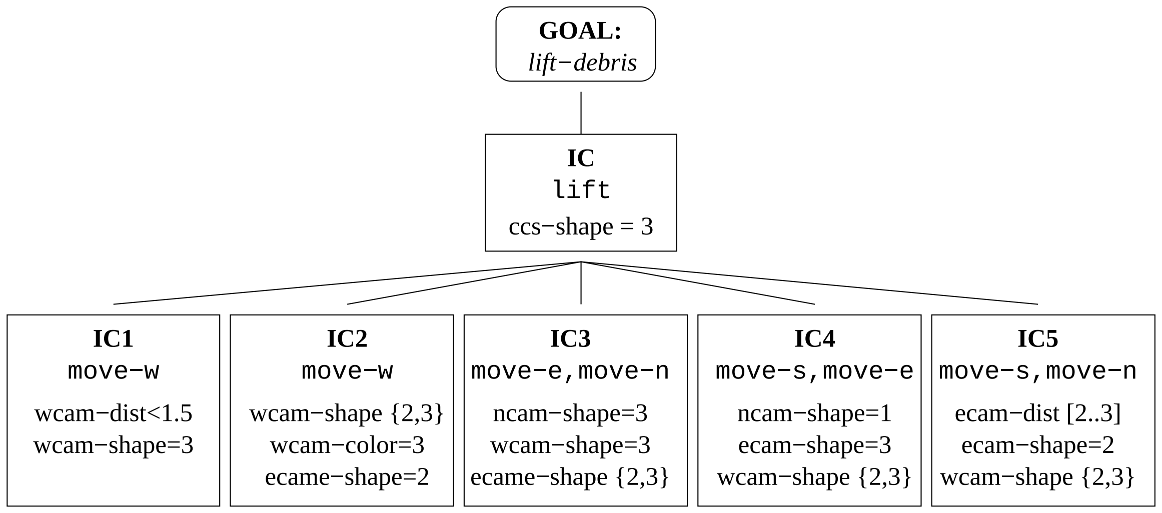

Figure 3.4. A plan tree for *lift-debris*, 2 levels deep.

outstanding conditions are satisfied in the agent's current state. In the case of our current plan, the node labeled "IC1" would have a completed goal stack; the `move-w` action satisfies the condition (`ccs-shape`= 3) and the initial conditions at "IC1" are satisfied in the current state. Each of the other leaves in figure 3.4 have goal stacks with outstanding conditions. For these nodes, the search can be continued to find alternate plans.

Checking a goal stack for a completed plan is another task that is complicated by operating at the sensorimotor level. Typically, to test for completeness, the planner must simulate the plan to ensure that what it expects to happen will actually happen. The planner simulates each plan step, transforming the starting state of the agent according to the operators it applies as it goes. It must first verify that each operator's preconditions are in fact met, then apply the postconditions to the current simulated state before going on to the next step. Plans must be verified in this manner because of a situation called *clobbering* in which operators inserted into a plan to satisfy certain conditions may invalidate downstream conditions previously considered satisfied by other operators. The process of state projection, or simulating the effects of a plan on the world state, allows the planner to verify that clobbering is not a problem.

For symbolic planners like STRIPS, simulation is relatively straightforward. State projection is a simple matter of maintaining a list of active predicates by running through operators' add and delete lists. So long as the operator models are accurate, the results of state projection are generally simple to compute and accurate themselves. For sensorimotor planners like ours, sensor readings must be projected from time series sensor data. Some sensors' behavior may be independent of the effect an operator was selected for. For example, the `move-w` action in the plan of figure 3.4 was included in the plan to drive `ccs-shape` to 3. But what will happen in the north and south facing sensors? These effects may be important to downstream planning steps. How can we project the sensory state in this case, or worse yet, how can we project sensory states for a complex, real-world sensorimotor agent like the Pioneer where the sensor behavior can be noisy and can change in the middle of an activity?

Our experience implementing a generative, means-ends analysis planner unfolded much like this discussion. The adaptation of a planner built for a symbolic, propositional world to a continuous, sensorimotor world proved difficult in three ways. First, operator selection presented a challenge. Determining which operators can satisfy outstanding goal conditions became an expensive matter of searching for goal crossings in raw time series data. Second, the sometimes extensive results of the goal crossing algorithm often resulted in a plan tree with a high branching factor. The search for plans itself was a computationally taxing process. Finally, sensory projection, necessary to verify the correctness of a plan, was a decidedly difficult task with no straightforward solutions. Observations like these led us to the conclusion that it is preferable to avoid sensory projection and its related problems (and by extension classical generative planning) in a sensorimotor context.

A desire to move away from classical planning led us to the following insight. The goal of an agent is to recreate some experience. For any such goal, an agent may previously have had a similar experience once, several, or many times. But there

```
1. Collect all experiences that match the goal $g$ into $E_g$
2. For each experience $e_n \in E_g$ do
    (a) For i from 0 to max-plan-length do
        • if precursor($e_{n-i}$) matches the current state,
          then $e_{n-i} \ldots e_n$ constitutes a plan
```

Figure 3.5. The PFT planning procedure.

is always at least one such experience in memory, and the agent knows the trace of experiences that preceded each instance of the goal outcome. Could it be possible to analyze the previous instances of the goal outcome, along with the experience trace leading up to it, and recover a plan from that trace that applies in the current state? Doing so would allow the planner to skip over the details of sensory projection, since those constraints are inherently enforced in the experience trace.

We call our planner the Plans From Traces (PFT) planner, as it attempts to recover workable plans from experience traces that lead to a goal outcome. The general PFT planning process is written up in figure 3.2.1, and can be summarized as follows. First, select all experiences that match the goal outcome. Next, choose an experience called the *exemplar*, e_n, from the set of matches E_g, where e_n is the n^{th} experience in the agent's history, e_{n-1} is the experience that directly proceeded e_n, and so on. Now check to see if the precursor sensory state for e_n matches the current state. If it does, then we have found an *inroad* to the trace, and we simply execute the action that generated e_n. If it does not match, then consider the precursor to e_{n-1}. Continue searching backward through the trace of experiences leading up to e_n until a match is found or some predetermined number of steps have been taken. If a match has been made at e_{n-i}, then the sequence $e_{n-i} \ldots e_n$ comprises a plan. If the threshold is reached without a match, choose a new exemplar from E_g and repeat until a matching plan is found or the set of exemplars has been exhausted.

The key to this simple algorithm lies in the matching procedure that tests if the precursor of an experience in the trace matches the agent's current state. A straightforward test for equality is not likely to turn up many matches in most reasonably complex domains. Consider that the basic GRIDSIM 8×8 domain has 72 states per configuration of debris in the environment. Each time debris is moved, taken away, or introduced into the environment, more unique sensory states are introduced until that number reaches several hundred. The Pioneer robot has a practically unbounded number of sensory states since many of its sensors are real-valued. The innovation of PFT is in how it generalizes experience traces to find inroads when an exact match does not exist.

PFT finds inroads to experience traces by treating each experience trace leading to an exemplar as an instance of a plan. The PFT planner attempts to fill in the conditions that allowed the actions of the plan to unfold into a trace resulting in the goal outcome. PFT does this by trying to produce a plan structure called a *triangle table* for the experience trace. Triangle tables were introduced with various incarnations of the STRIPS planner and in that system, served two functions. The first function was as an execution-monitoring structure that allowed STRIPS to respond (albeit in a limited way) to unexpected effects of actions in the world. The second function was as an archival structure for STRIPS' plan-caching ability so that an executable version of the plan could be stored and retrieved for later use.

A triangle table is a lower-diagonal matrix whose columns are labeled by operators and numbers, ranging from 0 to n, where n is the size of the plan. The rows are numbered from top to bottom from 1 to n[1]. A non-empty cell at row i, column j indicates the initial conditions for operator i are satisfied by the j^{th} operator of the plan. Said differently, entries in the row to the left of the i^{th} operator are precisely the

[1]Triangle tables used in classical planning have $n + 1$ rows, since classical planners result in a goal state rather than a goal outcome.

	current−state 0	1	2
1	`wcam−dist < 1.5` `wcam−shape = 3`	**move−w**	
2	---	`ccs−shape = 3`	**lift**

Figure 3.6. A simple triangle table for *lift-debris*.

initial conditions of that operator, and the entries in the column below the j^{th} operator are precisely the conditions achieved by operator j that are needed by subsequent steps of the plan. Column zero contains conditions in the current state description that are initial conditions of the various operators of the plan. The entries in the bottom row are those achieved by various steps in the plan that are components of the goal state, in the case of classical planning, or in our case, initial conditions to the goal outcome. A sample triangle table, built from our STRIPS-style plan to execute the *lift-debris* operator, is shown in figure 3.6.

Triangle tables are efficient representations for simple plans. They express the action sequence that comprises a plan, the initial conditions to make each step possible, and they indicate which plan steps are responsible for satisfying later steps' initial conditions. The PFT planner generates plans by attempting to build working triangle tables for existing experience traces. PFT follows the general procedure outlined in figure 3.2.1, building partial triangle tables for experiences $e_{n-i} \ldots e_n$ as part of the matching process at the heart of the algorithm. When a triangle table with no unsatisfied initial conditions in any of its cells is built, then a working plan has been recovered.

Let us consider an example of the PFT planner in operation to illustrate how partial triangle tables are generated. The planner starts by retrieving a set of experiences E_g that match the goal. Let us assume again that the goal is *lift-debris*, and therefore E_g is a set of experiences that have the same DELTA-SIMPLE encoding as *lift-debris*.

The PFT planner then selects an exemplar from E_g. Suppose it is an experience labeled `exp-lift-1957`[2]. It considers this experience alone as the basis for a possible plan and attempts to build a triangle table as evidence that this plan will work. The triangle table for a single step plan has two columns and a single row, and is pictured in figure 3.7. The PFT planner must next compute the initial conditions for experiences of type `exp-lift-1957`. In our earlier discussion, the initial condition for *lift-debris* is (`ccs-shape`$= 3$). Assume for the sake of this discussion that the initial conditions have expanded to the following, a more accurate set of initial conditions for this domain:

$$((\texttt{ccs-shape}= 3)) \wedge (\texttt{bay-shape}= -1))[100\%]$$

This new addition to the initial conditions for *lift-debris* states that the bay must be empty prior to taking action in order for the lift to succeed. The PFT planner now places the initial conditions into the triangle table, one cell to the left of the operator they apply to: row 1, column 0. The corresponding triangle table is shown in figure 3.8. The PFT planner then attempts to validate the table by showing that all the initial conditions in all the cells satisfy the definition of a triangle stated above. That is, conditions in row (i, j) are satisfied by operator j in the plan and enable operator i to work. Note that for the plan described by the current triangle table to work, the conditions in cell $(1, 0)$ would have to be satisfied in the current state. The bay condition is satisfied, but the ccs condition is not. We indicate cell with conditions that violate the definition of a valid triangle table by shading the cell, and the condition(s) that violate the definition are printed in bold. A triangle table with an invalid cell does not express a valid plan, and PFT must continue on.

[2]This is an actual expeirence generated by our system as are the rest of the experiences referenced in this section.

The planner's next step is to expand the triangle table. It retrieves the experience recorded just prior to `exp-lift-1957`, which is an experience labeled `exp-move-s-1956`. An extra row and column are added to the triangle table, as shown in figure 3.9. Next, the initial conditions for each step are computed, and inserted in to the table one cell to the left of the operator they correspond to. First, the initial conditions for step 2, as shown in figure 3.10, then, the initial conditions for step 1, as shown in figure 3.11.

Again, the planner attempts to validate the triangle table. It works through the columns, right to left, to see if the operator taken at step j of the plan satisfies the conditions listed below it in column j of the table. Any condition in cell (i, j) that is not satisfied by step j of the plan is moved one cell to the left, to cell $(i, j-1)$, until the conditions are either satisfied by some operator, or they reach column zero, in which case those conditions must be satisfied in the current state for the table to be validated.

The table in figure 3.11 is checked as follows. First, cell $(2, 1)$ is tested to see if the experience `exp-move-s-1956` was responsible for satisfying either of the conditions listed in that cell. The goal crossing algorithm is used to perform the test, and in this case, it is determined that indeed the condition ((`ccs-shape`$=$ 3)) was satisfied by the experience `exp-move-s-1956`, but ((`bay-shape`$= -1$)) was not. The bay shape condition is moved one cell to its left, into $(2, 0)$, and cell $(2, 1)$ is validated. Next, cell $(2, 0)$ is checked. Cells in column zero are checked against the current state, and so the bay shape condition is compared to the agent's sensory state. The agent's bay is empty, and so cell $(2, 0$ is valid. Finally, cell $(1, 0)$ is considered. Referring to figure 3.1, note that one of the conditions in this cell, (`scam-shape`$=$ 3) is satisfied: there is a debris object to the south. However, the distance reported by the south-facing camera is greater than 1.5, and so cell $(1, 0)$ is not valid. The triangle table does not yet represent a working plan.

	current−state 0	1
1		**lift**

Figure 3.7. Step 1 of the PFT process: building an empty table for a one-step plan.

	current−state 0	1
1	**`ccs−shape = 3`** `bay−shape = −1`	**lift**

Figure 3.8. Step 2 of the PFT process: inserting the initial condition for the `lift` step.

The planner continues by again expanding the table to include three steps. It retrieves another preceding experience, this one labeled `exp-move-s-1955`, and inserts it into the table. The initial conditions for each step in the plan are acquired from the modeling system and inserted into the new triangle table, as shown in figure 3.13. Conditions that are satisfied remain in their current cell, and those that are not are moved left, as before, until they are satisfied or they reach column zero. The result of the validation process is the triangle table shown in figure 3.14: a fully validated table. The plan prescribes moving south until a piece of debris is less than 1.5 units to the south, then move south again until the ccs picks the debris up, and finally, execute a `lift` action to produce the *lift-debris* outcome.

While this plan is not the quickest route to the goal, it is a valid plan that will work as the GridSim domain is configured in figure 3.1. It is entirely possible that another trace in memory would lead to the shorter plan based on moving west. In any case, the PFT planner recovers plans from experience traces without having to address the difficult task of state projection.

	current−state 0	1	2
1		**move−s**	
2			**lift**

Figure 3.9. Step 3 of the PFT process: expand the triangle table to include a new action.

	current−state 0	1	2
1		**move−s**	
2		**`ccs-shape=3`** **`bay-shape=-1`**	**lift**

Figure 3.10. Step 4 of the PFT process: compute the initial conditions for step 2 of the plan.

	current−state 0	1	2
1	**`scam-dist < 1.5`** **`scam-shape=3`**	**move−s**	
2		**`ccs-shape=3`** **`bay-shape=-1`**	**lift**

Figure 3.11. Step 5 of the PFT process: compute the initial conditions for step 1 of the plan.

	current−state 0	1	2
1	**scam−dist < 1.5** scam−shape=3	**move−s**	
2	bay−shape=−1	ccs−shape=3	**lift**

Figure 3.12. Step 6 of the PFT process: test for condition satisfaction.

	current−state 0	1	2	3
1	**scam−dist > 1.5**	**move−s**		
2		**scam−dist < 1.5** **scam−shape=3**	**move−s**	
3			**ccs−shape=3** **bay−shape=−1**	**lift**

Figure 3.13. Step 7 of the PFT process: expand the triangle table to include a third action.

	current−state 0	1	2	3
1	scam−dist > 1.5	**move−s**		
2	scam−shape=3	scam−dist < 1.5	**move−s**	
3	bay−shape=−1		ccs−shape=3	**lift**

Figure 3.14. Step 8 of the PFT process: insert initial conditions and validate table cells.

The PFT planner incorporates elements from case-based planning with elements from generative planning. It is a case-based planner in the sense that examples of a path to the goal must exist in the experience history of an agent for planning to be possible. Each time it constructs a plan, the PFT planner is retrieving a plan from memory, whether or not the agent was acting under the influence of a plan at the time. In order to verify that a trace will work as a plan, though, the PFT planner must use means-ends type reasoning to determine how the steps in a trace work together to result in the goal. The planner is case-based in that it requires examples and generative in that it produces plans where there was no previous notion of a plan.

3.2.2 Plan Caching

As we outlined earlier, keeping successful plans around for reuse is, from a practical standpoint, a sensible thing to do. True case-based planning deals with how to efficiently store, organize, retrieve, and refine working plans. Considerable effort in this field has been devoted to the study of how to efficiently organize plans for retrieval when many exist, and how to retrieve plans that are most relevant to the task at hand when no exact match exists. Our needs our quite modest in comparison to the functionality that state of the art case-based planners provide. Since the number of goals our agent might pursue is fixed by the number of unique and repeatable outcomes in a given environment, our system has the luxury of using simple lookup tables or to do indexing. Our plans are indexed by their goal outcome and stored as triangle tables.

Since many goals can be satisfied a number of ways, a single goal lookup may result in one, none, or many cached plans. In the case where one or more plans are found, the retrieval system simply searches for one that applies in the current state. This procedure is a matter of looking for inroads into existing triangle tables. Recall that "inroads" are columns of triangle tables where, if the table were truncated at that column, the triangle table would be valid in the current state. Each candidate table retrieved can be scanned left-to-right for inroads.

When matching plans are retrieved, our system takes the opportunity to re-evaluate the plan during reuse. If the plan succeeds, then no further action is required by the planner. A count of successful uses is incremented and our agent goes on its way. If the plan fails, the execution system (described fully in chapter 4) returns the circumstances of the failure, and the caching system can either try to reconstruct the triangle table using updated initial conditions from the modeling system, or it can consider scrapping it altogether.

CHAPTER 4
PLAN EXECUTION

The fourth and final component of our developmental system is the execution module, the subsystem responsible for turning plans into real behavior in a physical or simulated world. A valid plan improperly executed is little better than a broken plan, and so proper execution is of vital importance to the success of our developmental system.The ability to abort plans when they go badly is also important, especially when uptime or unnecessary actions are costly. Each action an agent takes consumes resources. If we are working in simulated environments, each experience generated consumes time and memory. If we are working with a robot, additional resources are used, such as battery life and machine life. [1] If we allow plans to run beyond the point where there is no chance for success, then we are collecting experiences the agent doesn't necessarily need, and we are using time and energy that could be better spent some other way.

In this chapter, we will discuss how plans specified as triangle tables are executed by our system. We will also describe how our agent determines when plans are not going well, and the steps our system takes to abort plans with little chance of success. Before we go into the details of execution and monitoring, it is important to understand action-taking in the domains we work with.

This dissertation was designed around real-world platforms like the Pioneer mobile robot. These platforms come packaged with software controllers that allow a

[1]Robots have a limited lifetime, especially if they are allowed to crash into walls.

programmer to issue motor commands for the various actuators at any time. The Pioneer can be commanded to move forward, at 200mm/sec, and it will smoothly accelerate until it reaches the desired velocity or until another motor command is issued that tells it to stop or change its speed. Rotational, translational, and gripper controllers of the Pioneer can all be in operation at the same time. Clearly, allowing this kind of freedom would make planning and the execution of plans quite complex. Turning and moving together produces a controller with dynamics far different that either one alone, and varying the wheel speeds can result in a nearly boundless set of possible sensory patterns. In our developmental model, we limit the complexity of control by allowing only one active controller at a time. Furthermore, we discretize continuous controllers to operate at fixed levels: we only allow rotation at $\pm 50 deg/sec$, for example, or movement at $\pm 250 mm/sec$. These controllers are activated, allowed to run until a desired effect has been achieved, and deactivated.

For practical purposes, a time limit is placed on all experiences as well. The TURN and MOVE controllers of the Pioneer are allowed to run for a maximum of three seconds, and the gripper controllers are fixed in duration to the length of time it takes for the gripper to fully close or open. There are two basic reasons for placing maximum time limits on controllers. First, lengthy time series create a heavy computational burden throughout the system. We have found that smaller snippets of activity are more manageable and that longer traces do not necessarily offer any added utility for the purposes of modeling and planning. Second, we use several offline algorithms for execution monitoring. In the sections that follow, we shall see a suite of heuristics for judging whether a plan should be allowed to continue or be aborted. Some of these heuristics are based on how the execution of the plan is going and since a majority of available time series analysis schemes useful for execution monitoring work offline, the agent must periodically stop what it is doing in order to decide whether or not the plan should be aborted. Placing a time limit on controller operation allows the agent to take stock of its latest experiences to determine whether a plan is proceeding as expected. Note that controllers may be terminated before the time limit is reached, as is often necessary during plan execution.

The GridSim simulator was built to be compatible with the execution model of the Pioneer robot. The GridSim agent has controllers for moving, lifting, and dropping that may be activated and deactivated. Each of the move actions take approximately one second to traverse one cell in the grid, and the lift and drop operators take that same amount of time to complete the act of lifting or dropping debris. Each of the GridSim controllers are constrained to terminate in under one second.

4.1 Executing Plans

Executing the plans generated by our system is relatively straightforward and is based on the original STRIPS scheme. In STRIPS, steps of a plan are executed sequentially, and after each step, a test is made to verify that the preconditions for the next step have been achieved before moving on. This test is based on *kernals*, rectangular subarrays of the complete triangle table. The i^{th} kernal of a triangle table is defined as the cells bounded by $(i, 0)$, $(i, i-1)$, $(n, 0)$, and $(n, i-1)$, where n is the last row of the table. If all the expressions of the i^{th} kernal are satisfied, then the i^{th} *tail* of the plan is applicable, and execution can continue at step i in the plan. An example kernal is shown in figure 4.1. This figure shows the triangle table for *lift-debris* as introduced in section 3.2.1, with the 2^{nd} kernal highlighted. If all the cells in this rectangular subarray are satisfied in the current state, then execution can continue at step 2.

In simple domains with simple plans, it can be assumed that if after step i of a plan, the $(i+1)^{th}$ kernal should be satisfied and thus step $i+1$ could be taken next. If the kernal is not satisfied, it follows that something has gone wrong and the

	current−state 0	1	2	3
1	`scam-dist > 1.5`	**move−s**		
2	**`scam−shape=3`**	**`scam−dist < 1.5`**	**move−s**	
3	**`bay−shape=−1`**		`ccs−shape=3`	**lift**

Figure 4.1. The 2^{nd} kernal of the *lift-debris* triangle table.

planning agent can decide whether to abort or replan. With our planner and the sensorimotor domains we work with, there are a couple of other possibilities. First, if kernal $i + 1$ is not satisfied after step i is taken, the agent may need to execute step i again. Because we have placed an upper limit on the duration of an action, the controller of step i may have been deactivated before it fully achieved its desired effects. If this is the case, the $(i + 1)^{th}$ kernal will not be satisfied, but the i^{th} kernal will remain satisfied. A simple example of such a situation would be step 1 of the plan of figure 4.1; the purpose of this step is to drive the value of `scam-dist` below 1.5. The time limit allows the GridSim agent to drive this value down a maximum of 1 unit per experience. If in the starting state the distance to the debris to the south were 3 or more, then it would require multiple executions of the first step to drive `scam-dist` to the desired value.

The other possible contingency is that there are unnecessary steps in a triangle table that can and should be skipped. The PFT planner produces plans from experience traces, and these traces may have been generated by a variety of controllers. The agent may have been executing a plan to reach an unrelated goal, using the entropy-based L0 controller, or just wandering when the exemplar was produced. It is not uncommon for superfluous steps to appear in experience traces and make their way into a triangle table. In some of these cases, it is possible to eliminate these steps if they do not interact with other steps since their corresponding column in the triangle table will be empty, indicating that they do not achieve any later steps' preconditions. In other cases, these extra steps will find their way into triangle tables and must be dealt with at execution time.

All these contingencies can be handled by simply removing the assumption that plans move sequentially forward after each step in the plan is taken. Our execution module determines which step is next by taking the maximum value i for which the i^{th} kernal is satisfied in the current working triangle table. The value i may advance one or many steps, not change at all, or there may be no i for which there is a satisfied kernal. In the former case, the plan has created unexpected effects that have caused the agent to lose its place in the plan, and the agent has no choice but to start the planning process over.

This simple execution scheme allows our agent to repeat and skip steps as necessary, but it also introduces the possibility that the agent could become stuck on a single step in the plan without making any progress, or worse yet, the get caught in a cycle since the control scheme can potentially revert control from step j in a plan to some step i, where $i < j$. For this reason it is necessary to periodically test for reasons to abort a plan.

4.2 Aborting Plans

In our system, deciding whether or not to abort a plan during execution is a heuristic process. Rarely can the planner point conclusively to evidence that a plan is doomed to fail; bad plans are usually built on bad models that give them an apparent chance of success in spite of the weaknesses that will eventually cause them to fail. However, it is possible to predict failure in some circumstances once plan execution has begun. Each time an action is executed by our agent, it runs through a set of *abort reasons* to determine if there is evidence pointing toward failure. If any one of the reasons should predict failure, a the execution module aborts the plan, and returns the abort reason that predicted failure. The abort reasons can then be used by the plan caching system to diagnose the failure and decide on what to do with the aborted plan. Here are the abort reasons as currently implemented in our system:

LOST-THE-NIP This abort reason is invoked when the execution module can no longer find a kernal that is satisfied in the current triangle table. This happens when there is no step in the plan with satisfied initial conditions, usually because a step in the plan has produced unexpected results that clobber previously satisfied plan conditions and do not achieve any conditions for a later step. We call the highest-ranked satisfied kernal the *next instruction pointer* (NIP), and hence this abort reason is called LOST-THE-NIP.

GOING-NOWHERE This reason is invoked when a step in the plan is executed and has no discernible effect on the state of the world. An example might be attempting to lower the Pioneer robot's gripper when it is already down. If lowering the gripper is expected to achieve some conditions in the plan, then it is safe to abort, since the action is not bringing about the desired effect.

The GOING-NOWHERE abort reason works under the assumption that plans have no hidden state built into them. One can imagine situations in which this might be an unsafe assumption. An example might involve an action whose effects are delayed or unobservable (like pushing a button on a machine) but affect the outcome of later steps in a plan. Our sensorimotor agents have no way of representing hidden state, and we believe that solutions to these problems are beyond the scope of this dissertation, and may occur in later stages of development than we seek to model. In these later stages, it may be possible to infer hidden state variables, and to use the activities learned by our systems to

```
1. generate an initial condition tree for experiences with action $a_n$
2. add a state to the PFA for $a_n$ for each leaf in the initial condition tree
3. examine each experience $e$ with action $a_n$
    • if $e$ starts in PFA state $s_i$ and ends in state $s_f$, strengthen the directed edge
      $< s_i, s_f$ in the PFA
```

Figure 4.2. The procedure for generating outcome PFA for domain actions.

try and model the behavior of these conceptual variables as they intermediate the unfolding of plans, as has been suggested by Drescher [19].

PFA-Dead-End This abort reason is the most sophisticated and aggressive in our system. The basic idea is that when an agent is executing a step n in a plan, it has expectations of what will happen during that step. The expectations are expressed symbolically, as by the SSC outcome classifier. If it takes action a_n, it expects a certain outcome $\hat{o}(a_n)$ to unfold, and that it will satisfy the initial conditions of some future step in the plan. In general, the execution module will repeat step n until $\hat{o}(a_n)$ does unfold, and it has determined that the plan can proceed to a later step. If it were possible to detect when $\hat{o}(a_n)$ is unlikely to ever unfold given the current sensory state, then it would make sense to abort the plan. Suppose the Pioneer is in the middle of executing a plan step in which it is supposed to approach a red object by moving forward. If the robot were to hit a wall (that is not red), having hit the wall should be considered evidence that a red object is unlikely to be encountered by continuing to move forward.

The PFA-Dead-End abort reason uses probabilistic finite automata (PFA) to model which outcomes are likely to follow each other when an action is repeated. Finite automata are graphs in which the nodes represent sensory states and directed edges represent possible transitions between states. We create PFA from the experiences of action a_n using the procedure in figure 4.2.

First, a PFA state is created for each leaf in the initial condition tree for a_n. Then, transitions between the states are added for each pair of states $< s_i, s_f >$ for which there is an experience of type a_n that starts in PFA state s_i and ends in PFA state s_f. Each such transition $< s_i, s_f >$ is weighted by the number of experiences that make that transition divided by the total number of experiences that start in state s_i. The automata can then be used to determine if a particular outcome is likely to unfold as a result of repeated execution of a_n by performing reachability analysis from the current state to the state in which the desired outcome is most likely to unfold.

Suppose we want to build a PFA that models outcomes of the `drop` action in the GridSim simulator. The first step is to get the initial condition tree, which is shown in figure 4.3. The tree is simple, and reflects two possible outcomes of executing the `drop` action. If the cargo bay is full, the drop will result in the outcome shown in the leaf to the left. If the bay is empty, the action will result in no change in the sensory state, as expressed by the outcome in the leaf on the right. This tree prescribes two PFA states, one for each leaf. The two states of the PFA are shown in figure 4.4.

Next, the transitions are filled in. The `drop` action is very simple, and there are only two transitions possible in GridSim. They are shown in figure 4.5. This simple action illustrates clearly the situation that the PFA-Dead-End abort reason was designed to detect: dead ends in the PFA for an action. Once an agent finds itself in the state to the left of the PFA (where performing a `drop` has no observeable effects), it will never be able to get out with the `drop` action, and it is time to break from the plan.

A more complex PFA is shown in figure 4.6. This is a PFA built for the `move-w` action after 500 experiences in the 8×8 grid. Outcome labels associated with PFA states and transition strengths are omitted for brevity. Note that the initial

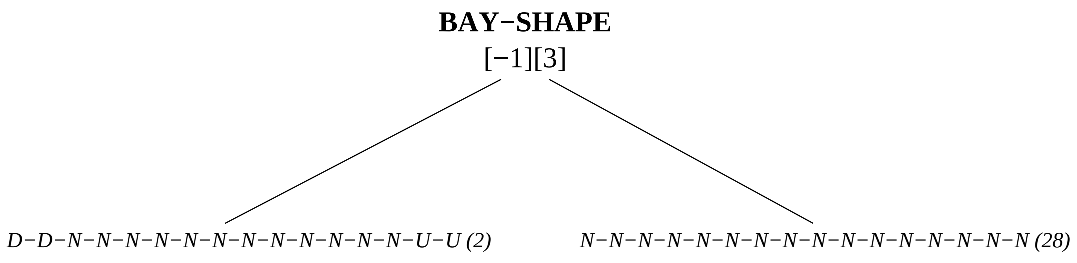
BAY−SHAPE
[−1][3]
D−D−N−N−N−N−N−N−N−N−N−N−N−N−N−U−U (2)
N−N−N−N−N−N−N−N−N−N−N−N−N−N−N−N−N (28)

Figure 4.3. An initial condition tree for the `drop` action in GridSim.

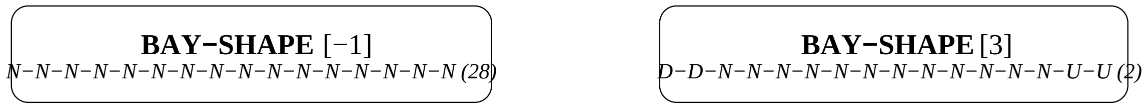
BAY−SHAPE [−1]
N−N−N−N−N−N−N−N−N−N−N−N−N−N−N−N−N (28)
BAY−SHAPE [3]
D−D−N−N−N−N−N−N−N−N−N−N−N−N−N−U−U (2)

Figure 4.4. Two PFA states corresponding to the two initial condition leaves for the GridSim `drop` action.

conditions generated by the modeling system make heavy use of landmarks to the north and south to determine the outcome of an action, as is reflected by the preponderance of conditions using the north and south facing cameras in the PFA. Like the `drop` PFA, the `move-w` PFA contains a dead-end state, useful for the PFA-Dead-End abort reason.

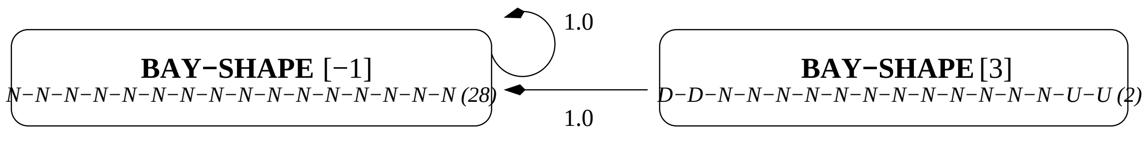
1.0
BAY−SHAPE [−1]
N−N−N−N−N−N−N−N−N−N−N−N−N−N−N−N−N (28)
BAY−SHAPE [3]
D−D−N−N−N−N−N−N−N−N−N−N−N−N−N−U−U (2)
1.0

Figure 4.5. The PFA for `drop`, fully constructed.

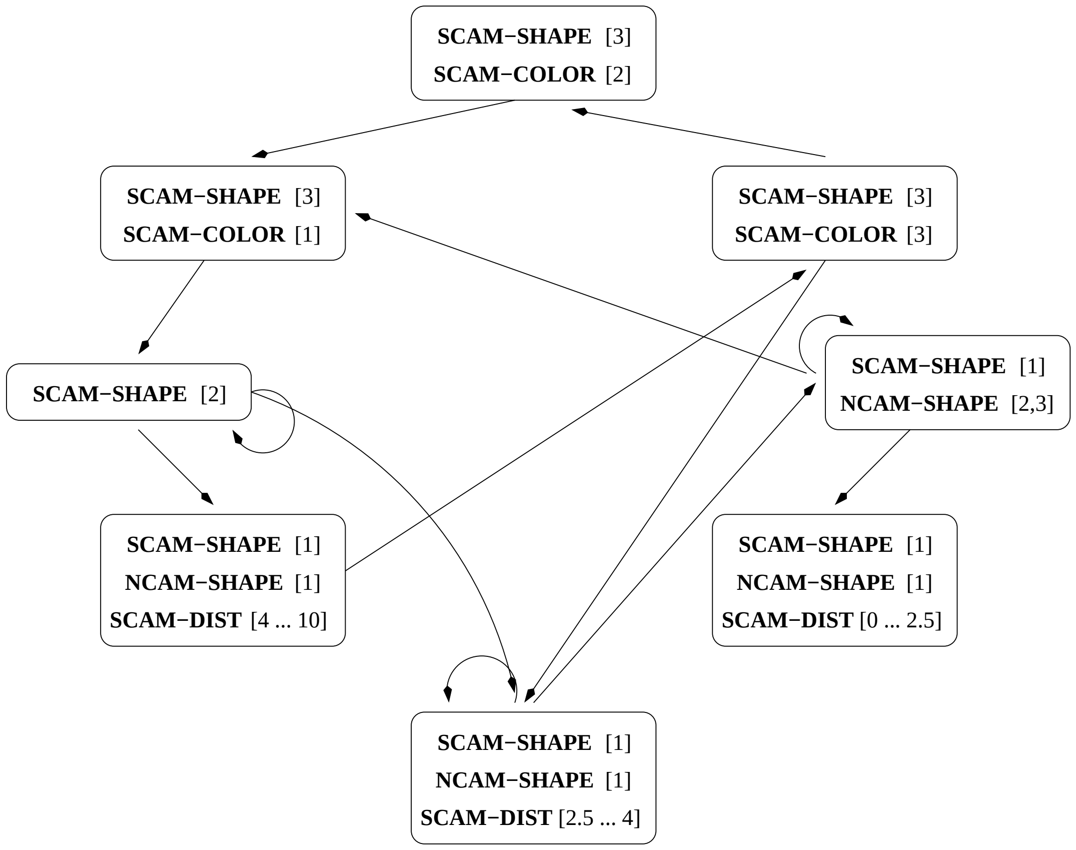

Figure 4.6. A PFA for `move-w`.

CHAPTER 5
GOAL SELECTION

Most classical planning systems pursue goals that are generated exogenously, typically from a user. We are interested in autonomous development, though, and so having goals fed to our planner from a human user is unacceptable. At the stages of development we are considering, it is not assumed that even the most rudimentary language understanding is present. We require an explanation for how an agent not only achieves, but also generates its own goals. This is the task of the *goal selection* system.

The goal selection system determines what the agent actually spends its time doing (or attempting to do), and ultimately shapes the types of activities that the agent will learn. This component to our developmental system can literally make or break the process of development [1]. As such, we would like the goal selection module to generate goals which meet the following criteria:

- **Plausibility**: the system should produce goals that are possible for the agent to achieve.
- **Sustainability**: the system should attend to the basic needs of the agent. This includes vegetative needs like hunger, thirst, and fatigue, if they apply, as well as related needs like aversion to pain. The goal selection scheme must keep an agent out of trouble.

[1]Running a robot into a wall repeatedly tends to break it.

- **Learning**: the system should provide an agent with opportunities to learn about its environment.

- **Purposefulness**: the system should behave with purpose. Let us define an agent that acts *purposefully* as one in which the decision to do something is always based on some criteria, and not simply random.

Recall that we adopt the *planning to act* philosophy in our system. Planning to act suggests that at the highest level of control, agents plan to *act* rather than to achieve states. This idea is motivated by Piaget and others who developed theories of development based on the acquisition and repetition of *schemas* (repeatable chunks of activity). These theories implies that it is the activity, not the state, that is rewarding. Our discussion of planning in chapter 3 assumed that goals were generated in accordance with planning to act, and consequently we ensured that all goals met the first of the four criteria listed above. Planning to act constrains the space of goals to encompass only situations that *have been achieved before*, and consequently can be assumed to meet the plausibility constraint.

In the remainder of this chapter, we describe a goal selection scheme that chooses among the goals that are available to an agent. Our goal selection scheme comprises three parts: a basic exploratory controller, called the level-zero L0 controller, a high-level control scheme called a *motivational system* that weighs the needs of the agent in making a decision about what to do, and a system of *reflexes* for biasing the developmental trajectory toward important results. We describe each of these systems, and show that our goal selection system satisfies the remaining three criteria: sustainability, learning, and purposefulness.

5.1 Acting With Purpose

One of the criteria for the goal-selection component of our system is that it guides an agent to always act with *purpose*. An intelligent agent will always have a reason for the things it does, whether that reason seems reasonable or not. We may break down activity into two types: *exploitative* and *exploratory*. Exploitative activity attempts to utilize learned models to reap the rewards an environment has to offer. Exploratory action seeks to improve models, so that exploitation might be more effective in the future.

Exploitative behavior is always purposeful. An agent engages in foraging behavior to find food to eat, or rests to recharge its batteries. Exploratory behavior, however, is not always traditionally purposeful. Many early systems, when the decision was made to explore, engaged in random behavior for some period of time. While the high-level aim of an agent in these systems is to explore, there is no explicit connection between each action and the goal of exploration. These agents, in acting randomly, may or may not actually explore in the sense that they may or may not encounter new experiences that allow them to improve their models as a result of what they do. They do not act with *purpose*.

In this section, we will outline a baseline controller that our agent can engage when it has no models to exploit or no need for exploitative behavior. We call this controller the Level 0 (L0) controller. It is a simple, greedy exploratory controller that recommends actions about which the modeling system has the least certainty. The agent, by taking the actions recommended by the L0 controller, acts with the purpose of improving its weakest models.

Our L0 controller is based on the *entropy* statistic. Entropy has many forms in the various sciences. The one we use here is based on Boltzmann's entropy, sometimes denoted by h.

$$h = -k\Sigma_i(p_i log(p_i)) \tag{5.1}$$

Boltzmann derived this equation in the service of statistical mechanics, and it was later proved by Claude Shannon to satisfy the needs of information theory in expressing uncertainty about messages sent from an unreliable source [75]. In the information theory context, $k = 1$, and p_i is a probability value for a given bit of a message.

This same entropy statistic can be used to express uncertainty in any system that makes probabilistic predictions. In this case, i iterates over the possible predictions, and p_i is the probability of i coming true. Note that in cases where a prediction can be made with absolute certainty, there is only one possible prediction, with probability 1, and $p_i log(p_i)$ evaluates to zero. In the worst case, the probability distribution over possible outcomes will be uniform. That is, p_i is the same for all possible outcomes. If there are two possible outcomes, the maximum entropy will be where both values for p_i are the same, and $h = -(.5log(.5) + .5log(.5))$, or 1. If there are three possible outcomes, the maximum entropy is approximately 1.58, with four outcomes, the maximum entropy is 2, and so on.

We use the entropy statistic to compute *predictive uncertainty* at the leaves of initial condition trees. Consider the tree generated using Pioneer-2 data collected during MOVE actions shown in figure 5.1. Each leaf of the tree corresponds to a region in the state space where, when a MOVE action was taken in the past, one or more distinct outcomes were observed. The outcomes are listed with their observed probability, and these results can be used to compute uncertainty about what will happen if the Pioneer-2 executes a MOVE action from any sensory state. For example, if the Pioneer-2 is sitting in a state where the `vis-x` sensor is reporting a value of -10, the agent is in a state represented by the second leaf from the left in the tree of figure 5.1. There have been four observed outcomes of MOVE in that region of the state space. The entropy calculation is

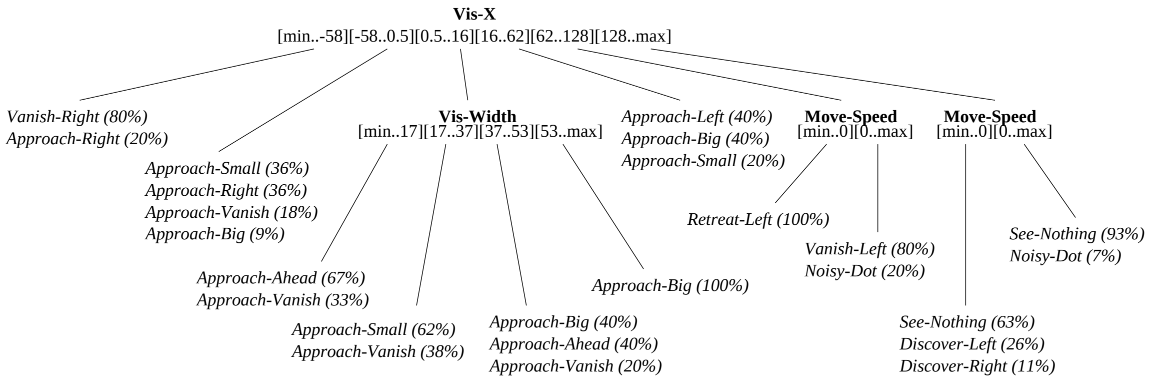

Figure 5.1. An initial condition tree generated using precursor phase data from the Pioneer-2.

$$-(.36log(.36) + .36log(.36) + .18log(.18) + .09log(.09))$$

which is approximately 1.64. Each action has its own such tree, and each state of the Pioneer-2 corresponds to a single leaf in the trees, so in any context, an entropy value can be calculated for each action. These values represent the uncertainty in the current models, and the L0 controller can simply recommend the action with the greatest entropy value. This allows the agent to focus exploration based on where the models are most uncertain, and to act with purpose all the time.

5.2 Motivation and Goals

In the very early stages of development, opportunities for exploration are overabundant. Nearly any action in any state can provide valuable information and consequently an agent's models change rapidly during this early period. Due to these unstable and inaccurate models, opportunities for exploitation are comparatively few. During this stage, the L0 controller is of great use in selecting the most profitable of the available opportunities for exploration.

Eventually, though, models will begin to stabilize and become accurate. Prediction, planning, and exploitation become possible, and an agent is offered the opportunity to reason about outcomes rather than actions. The idea of planning to act finally comes into play, and goal selection becomes a balancing act of deciding which of all the outcomes an agent knows about will be the best to pursue or whether a situation is appropriate for some opportunistic exploration. The details of how one manages the balancing act between exploration and exploitation determine how well the criteria of sustainability and learning are attended to.

We manage the balance between exploitation and exploration (sustainability and learning) with a set of models that comprise a *motivational system*. The motivational system enumerates the various motives an agent might have for acting, such as hunger, pain, fatigue, and curiosity, and rolls them into a model that can be used to evaluate and select the goals of our system.

The motivational system can be thought of as defining a preference relation over the outcomes that an agent knows about. Let $\mathcal{O}$ denote the set of all outcomes that our agent knows about. The preference relation for agent x, $\psi_x(o_1, o_2, s_t)$, holds iff in sensory state s_t agent x prefers the outcome o_1 over o_2. The goal of the system at time t, then, is o_g such that $\forall o_{n \neq g}(\psi_x(o_g, o_n, s_t))$.

If we were interested only in the learning criterion, we could base ψ on a metric such as information gain or maximum entropy as the L0 controller does. Every decision that agent x would make would be based on the amount of information to be gained about a particular outcome. Agent x would be constantly planning to achieve outcomes for the purpose of exploration. Likewise, if the agent had some single, all-encompassing vegetative concern, such as keeping its battery from running out, we could base ψ solely on the expected loss or gain of battery power. This agent would constantly be building plans that would maintain or increase its battery level. While both of these formulations of ψ satisfy the purposefulness criterion, both will also suffer the drawbacks of ignoring one of the four criteria for effective goal selection. The exploratory agent will likely run out of battery power, defeating the sustainability of the motivational system, while the agent obsessed with its batteries

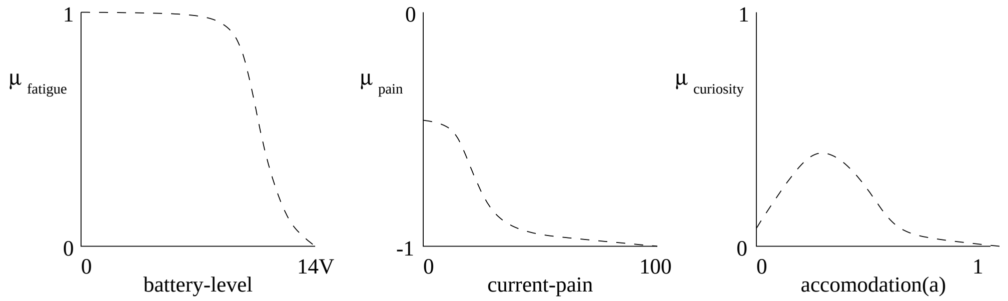

Figure 5.2. Coefficient functions for each of the Pioneer-2's primary motivational factors.

may never explore enough to find the most efficient policy for keeping its batteries running strong, stifling the learning aspect of development.

An agent whose goal is to truly explore the affordances of its environment, all the while ensuring to stay alive, must attend to a variety of factors that motivate its behavior. The relation ψ should reflect the variety of these factors. For a mobile robot exploring the surface of Mars, for example, ψ may reflect three distinct factors: *fatigue*, the need of the robot to maintain battery level, *crash avoidance*, the need of the robot to keep from colliding with other objects at high speeds, and *science*, or the requirements of the Mars rover to collect data about the surface of Mars by collecting samples and exploring. Were the rover a learning agent, we might add in *curiosity*, a factor that would compel the rover to sometimes take actions that improve its performance through learning about the dynamics of operating on Mars. We call each of these components of behavior a *motivational factor*. Individually, each expresses a basic need of the agent, and collectively, they comprise behavior.

In our model of motivation, we represent each motivational factor F with a *drive coefficient* $\mu_F(s_t)$. The drive coefficient expresses the relative importance of F in state s_t; the importance of the fatigue factor, for instance, may increase as the battery voltage decreases. Drive coefficients represent internal states of the agent, and may or may not be reflected in the sensory aparati. For this reason, when we refer to states in this chapter, we are referring to not just sensory configurations, but also any internal state measures relevant to motivation. An agent may have a motivational factor for *frustration*, for example. Frustration is not reflected anywhere in the sensory aparati of an agent. Rather, it is an internal state. Perhaps frustration is the ratio of broken plans to successful plans over the last 20 minutes of operation. For the purposes of learning planning operators, and using those operators in planning, internal states have no utility, but for the purpose of selecting goals, internal states like frustration form the basis of motivation. States referred to in this chapter should be thought of composite sensory/internal states.

Each action outcome o_n will have an expected motivational payoff $E_\Delta(o_n, s_t)$ were o_n to be achieved from state s_t. The product of the drive coefficient and the expected payoff yields an expected *factor value* of F for o_n. The sum of these products over $F \in \mathcal{F}$, where $\mathcal{F}$ is the set of all motivation factors, is the *desirability* of outcome o_n in state s_t, denoted by ρ.

$$\rho(o_n, s_t) = \textstyle\sum_{F \in \mathcal{F}} \mu_F(s_t) E_\Delta(o_n, s_t) \tag{5.2}$$

and

$$\psi(o_n, o_m, s_t) \iff (\rho(o_n, s_t) > \rho(o_m, s_t)) \tag{5.3}$$

It is worth noting that we consider each factor to comprised an internal, hard-wired (perhaps genetically determined) component, $\mu_F(s_t)$, and a component contributed by the environment which must be learned, the quantity $E_\Delta(O_n, s_t)$. Figure 5.2 shows some possible coefficient functions for three sample motivational factors designed for the Pioneer-2 mobile robot. The motivational factors pictured are:

- **Fatigue** The coefficient function for fatigue depends on the robot's battery level, and is near zero when the battery is fully charged at 14V, indicating that fatigue plays no part in desirability in the fully charged state. As the battery level drops, though, the coefficient rises, most dramatically in the range 10V-12V. Fatigue becomes an issue in this range, as the robot's effectors start to become unreliable when its voltage drops below around 10V.

- **Pain** We use a simulated pain sensor for the Pioneer-2 mobile robot. Sudden contact with immovable objects produces surges of activity in the pain sensor. The coefficient function for pain, in contrast to fatigue, starts in the negative range, indicating that the possibility of pain reduces the desirability of an activity regardless of whether the robot has recently experienced pain, and when the robot has recently experienced pain, the inhibiting effect of pain avoidance only increases.

- **Curiosity** The curiosity coefficient depends on the activity under consideration, and a measure called *accommodation*. The role of the curiosity drive is to provide an analog for the action entropy metric used in the L0 controller to action outcomes. As the utility of the L0 controller drops off, the drive to explore and learn is picked up by the curiosity component, which attempts to identify action outcomes about which relatively little is known. Accommodation is a measure of the agent's familiarity with a particular outcome, and comprises a measure of the uncertainty in the initial conditions for the particular outcome and the number of times an outcome has been experienced. Accommodation is based on something we call the *moderate novelty effect*. Outcomes that are very unfamiliar or very uncertain can daunting to a developing agent, and curiosity to explore them is muted. Similarly, outcomes that are accurately modeled or outcomes that are uncertain but have been experienced frequently no longer generate interest. It is the space between utter unfamiliarity and complete familiarity – the space where an outcome is moderately novel – where a developing agent is most interested in exploration. The result of accommodation on curiosity is that if an activity is moderately novel, then accommodation is low, and curiosity makes a positive contribution to that activity's desirability. As the agent exercises the activity, and builds better models of how the activity works, accommodation increases, and the contribution of curiosity to the desirability of the activity tapers off. The curiosity coefficient reacts to the novelty of an activity according to a bell shaped curve, in a manner consistent with infant accommodation studies [70]. The curiosity factor asserts that all other things equal, an agent prefers to engage in activities it can learn most about.

Each outcome of the Pioneer-2 will have an expected payoff for each of the above factors. They are computed each time an experience is generated by differencing the internal states associated with the motivational factors before and after an outcome is experienced. The motivational payoff is stored along with each outcome profile with a little help from the modeling system, and may be retrieved as an expected value as needed by the motivational system to compute the desirability of any given outcome.

The preference relation ψ, based on desirability and motivation, gives an agent a means for selecting a goal. It can simply choose the outcome that maximizes d to beat out all other outcomes in ψ. Next, we will evaluate at the motivational system in terms of the criteria for goal selection outlined at the beginning of this chapter.

5.3 How Motivation Drives Development

The two primary jobs of the motivational system are to keep the agent out of trouble and to provide learning opportunities for the operator modeling process. These two priorities are expressed implicitly in the motivational system for an agent, and the design of the motivational system is such that the balance between exploration and exploitation (or, variously, sustainability and learning) is managed automatically.

Our motivational system manages this tradeoff in much the same way that counter-based directed exploration policies studied in the reinforcement learning literature do [81]. That is, all other things equal, an agent has a slight preference for improving deficiencies in known activity models over exploiting them for rewards. When it is clear that pursuing some goal will result in information gain without putting the agent at risk, it will usually go for it. When vegetative needs become pressing, they will express themselves through their respective drive coefficients and dominate the preference relation. So long as an agent has the capacity to address its vegetative needs through planning, it will do so, and then the slight preference toward learning will become the primary influence over the preference relation.

The questions we wish to answer here are, "can the tradeoff be managed efficiently by our motivational system?" and, "does the curiosity factor properly direct exploration?" We have devised a set of experiments to answer these questions, but before we present these experiments, it is important to draw attention to an aspect of exploration in our model of development that is not common to all learning systems.

In most reinforcement learning systems, an agent has access to the entirety of its action and state spaces, and those things do not change over time. This makes it possible to reason about areas of the state space it has not visited or actions it has not yet executed in certain contexts. This simplifies exploration because an agent can reason explicitly about what *it does not know* or has not yet encountered. This is an important and powerful faculty. Our system reasons about outcomes during planning, and these outcomes are not known a priori. Agents in our model cannot reason about outcomes until they have experienced them. Thus there are two aspects to exploration in our system: first, the agent must *discover* the outcomes available to it in its environment, and second, an agent must effectively *improve* uncertain outcome models until they become accurate,

Since our motivational system is limited to reasoning about things it has already experienced, it is not clear that it addresses the discovery aspect of exploration. But we posit that in many cases, exploration can be an indirect result of exploitation done with incomplete or incorrect models. Said differently, a planner working with poor models will generate plans that create unexpected results that an agent can learn from. Our claim is that our system manages to effectively discover action outcomes, improve models of those outcomes, and exploit them for rewards.

5.3.1 Experiments

We have run two sets of experiments to illustrate how our motivational system manages the tradeoff between exploration and exploitation. The first is based on a simple block painting robot, and the second is based on a generalized set of randomly generated Markov decision processes.

5.3.1.1 Block Painting Domain

The block domain consists of a simulated robot working in a parts preparation factory. The robot has a gripper and a paint gun, and sensors to detect whether or not it is holding something, whether or not what it holds has been painted, its current fatigue level, and its current paint level. As parts pass by the robot on a conveyor belt, the robot may attempt to pick a block up, open its gripper to drop what it is holding, fire its paint gun at whatever is in its gripper, rest, or activate a refill mechanism to refill its paint gun.

The outcome of each of its five actions is based on the observable changes in the four sensors; if the robot sprays its paint gun, it may paint a block, it may repaint a block, it may paint its own gripper, or if it is out of paint nothing will happen at all, for example. Each of these outcomes has a unique DELTA-SIMPLE profile, which for the purposes of this discussion we will denote with more informative titles like *lift-block* and *paint-block*.

This particular robot acts under the influence of 3 motivational factors: curiosity, fatigue, and *reward*. The first two are the same as those we described for the Pioneer-2. Curiosity compels the robot to exercise its outcomes. Excessive fatigue may cause some actions to fail unexpectedly. Reward is an exogenous signal used to indicate to the robot when it has done something good (like paint and release a block) or bad (like attempt a refill when its paint gun is already full).

We can use this simple domain to illustrate how exploration can occur when an agent attempts to exploit inaccurate models. An example often occurs within the first few experiences of this robot. Suppose the agent, right after being turned on, executes the `grasp` command. It picks up a block, and receives feedback that it has experienced a new outcome, which for this discussion, we will call *pick-up*. Immediately, this outcome will be the most highly rated by our motivational system, not only because it is the only available outcome, but because of its novelty. The goal selection system will suggest that the robot try and reproduce it. Due to its poor model of how *pick-up* works, the planner will assume that there are no initial conditions to *pick-up*, and that activating the `grasp` controller will produce the desired outcome. Since the robot is already holding something, the `grasp` controller experiences a different outcome. Attempting to grasp with the gripper already engaged will produce the *pu-fail* outcome. The robot has learned a new outcome due to planning with an incomplete model.

5.3.1.2 Random MDP Domain

Rather than hand-build increasingly sophisticated domains such as the block painting domain to show generality and scalability, we decided to apply our motivational system to randomly generated Markov decision processes. A Markov decision process (MDP) consists of a set of n_s states (denoted Q) and a set of n_a finite actions (denoted Σ). An agent occupies a single state in Q, and can move from state to state by executing actions in Σ. Uncertainty is allowed in MDP action execution, but here we will deal with deterministic action outcomes. MDPs must satisfy the *Markov property*: any transition from state s_i to state s_j taken as a result of action execution must depend only on the state of the system at s_i and the action taken. The MDP representation is one often used in planning, since the representation facilitates the use of many graph algorithms such as reachability analysis. We use MDPs as a general model to test the extensibility of our motivational system to exploration in increasingly sophisticated domains. Our MDP agent responds to two motivational factors: *novelty* and *reward*. Novelty is as described previously, and reward is simply a positive reinforcement signal that comes to the agent as a result of experiencing certain outcomes in the MDP.

The procedure we use for generating random MDPs is shown in figure 5.3, and can be outlined as follows. A user specifies the complexity of the MDP in terms of the number of states, n_s, and actions, n_a, available to the agent. A set of n_s states is generated and they are broken down into randomized subgroups. Then, transitions are added to the graph. Each transition in the graph corresponds to a single action outcome in the domain, and is randomly assigned to one of the actions. Transitions are added systematically to ensure each state is reachable from any other state and that there are no dead ends. States within subgroups are generally tightly connected, with a few transitions in each subgroup leading to a new subgroup. Default outcomes are added to the graph (which do not cause state transitions) until each action has an outcome in each state. Once a legal graph has been generated where all states are reachable, rewards and penalties are added to the graph.

The MDP structures that we generate are intended to resemble a typical domain for activity learning. The small, tightly connected subgroups correspond to simple,

1. Generate a state variable with a user-specified number of states.
2. Generate a user-specified number of generic actions.
3. Break the states up into subcycles of random size.
4. Generate transitions within the subcycles.
5. Generate transitions between the subcycles such that there are no dead-ends or unreachable states in the process.
6. Generate an outcome for each transition and assign it randomly to an action.
7. For each state-action pair that does not yet have a unique outcome, assign a "default outcome" that does not change the state of the agent.
8. Assign "reward" values for each outcome. In our experiments, there is a 10% chance of an outcome having a negative reward in the range $[-0.5 \ldots 0]$, and a 10% chance of an outcome having a positive reward in the range $[0 \ldots 0.5]$.

Figure 5.3. The procedure for generating random MDPs with motivational payoffs.

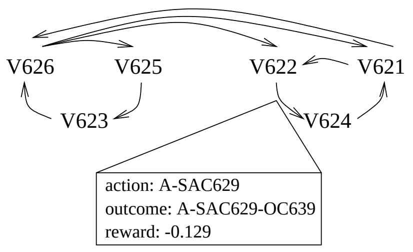

Figure 5.4. A sample, randomly generated MDP.

repeatable cyclic activities, and the longer, intergroup transitions represent bottleneck outcomes that change the focus of an agent to new subgroup activities. A small, randomly-generated MDP is shown in figure 5.4. Outcome labels are omitted for clarity, as are self-transitions that denote the default outcomes of step 7 in the generation procedure. A single outcome label assigned to one of the transitions is shown, along with its reward value. In the full MDP, each transition is assigned such an outcome label. Note that each state in the MDP is reachable from any other state via a legal sequence of actions.

5.3.1.3 Results

Our experiments with these two learning domains were intended to show two things. First, that exploration could be efficiently and effectively managed by our motivational system. This includes both senses of exploration: discovering the unknown outcomes and exercising the outcomes to build strong models once they are found. Second, we wanted to show that the transition from exploration to exploitation would be smooth, and that the transition to full-on exploitation would not occur until strong models for all operators had been learned. It is also our goal to show that the long-term exploitative behavior is optimal or near-optimal with respect to the amount of reward that can be achieved by an agent in the domain. To do this, we let the motivational system control simulated agents in each domain until behavior became fully exploitative and stable for several hundred steps. Each time a plan is generated to achieve an outcome, we record the desirability levels of each of the known outcomes and increment outcome counts each time one is experienced. This allows us to examine the long-term behavior of the system in terms of which outcomes are dominating ψ over time (and thus are selected as goals most often).

For the purposes of these experiments, we implemented a simple planner and learning schedule to isolate the effects of the motivational system from nuances in the full-blown planning and modeling systems described in previous chapters. The planner here is a simple generate-and-test planner that uses the simulator to evaluate whether or not it thinks a plan will succeed. It is purposely allowed to generate illegal plans if it has not yet experienced the outcomes that would be necessary for the planner to identify them as illegal. An example of an allowable but illegal plan would be the `grasp` plan described in the description of the block painting domain. The agent does know about the *pick-up* outcome, and thus it can use it in plans, but since it has not experienced *pu-fail*, it cannot verify that the plan would fail. Once it has experienced the *pu-fail* outcome, the planner will be able to recognize that simply executing a grasp will fail to produce the *pick-up* result in this simple example. In each experiment, the agent was initialized with no knowledge of its domain, and the motivational system was engaged.

Plots of the desirability of various outcomes in the block painting domain are shown in figure 5.5. In figure 5.6, one sees occurrence counts for the outcomes in the block painting domain. The results represent a single run of the experiment, but results are similar across trials and different starting configurations. By 60 steps into the simulation, all 19 outcomes have been experienced by the simulated agent. By about 100 steps into the simulation, the effects of novelty for 16 of the 19 outcomes have dropped to levels sufficient to make them undesirable except as steps in a plan to achieve some other outcome. The effects of novelty can be seen in each of the outcome plots of figure 5.5; after the outcome is discovered, desirability rises to a sharp peak and then tapers off in the typical bell-curve shape. We call the effect of novelty being driven down to levels where an outcome becomes undesirable to exercise *habituation*. Habituation is reflected in the desirability chart when the desirability plot for an outcome is driven down and becomes dominated by the desirability of another outcome. This is easily observed in the time period between about 300 and 430 steps of figure 5.5 where the desirability of *repaint* sinks below that of *drop-*

completed-block. The desirability of repaint continues to oscillate as the paint levels of the gun change (recall that attempting a refill on a full paint gun results in a penalty), but never actually exceeds the desire to achieve rewards by executing *drop-completed-block*. At around 300, exploration has for all practical purposes stopped, and the robot has settled into a routine of painting and dropping blocks. This is reflected in the plot of outcome counts, where the steps required for dropping a painted block (*pick-up*, *paint-block*, and *drop-completed-block*[2]) continue to be reinforced. Note that *refill* and *rest* also continue to be executed even though their individual desirabilities are low. This is because they are periodically included in plans when fatigue and paint level become a factor in the outcome of the plan for *drop-completed-block*.

Figures 5.7 and 5.7 show plots of desirability versus time and outcome counts versus time for a 5 action, 10 state, 22 outcome MDP. Outcome labels, which are randomly generated tokens like `A-SAC346-OC373`, have been omitted from the graph for clarity. As in the block painting domain, the 22 outcomes have all been discovered by step 100, and the most desirable outcome has distinguished itself by 200 steps into the simulation. Beyond that point, only six outcomes continue to be executed as plan components to exercising the outcome with the highest E_{Δ}^{reward}.

Figures 5.9 and 5.10 show desirability and outcome counts versus time for a larger MDP with 10 actions, 20 states, and 44 outcomes. In this problem, our simple planner was bogged down with a very large search space of 44 operators, and due to time constraints, we cut the trials short after 210 steps. Still, the system managed to find 39 of the 44 possible outcomes after only 140 steps, and had become focused on a sequence of two rewarding outcomes shortly thereafter. Interestingly, as the graph of outcome counts shows, one of the highly rewarding outcomes had been discovered almost instantly. Over the course of exploring novel outcomes for the next 140 steps,

[2]The plots for *drop-completed-block* and *paint-block* are virtually the same in figure 5.6, and appear to be a single plot in the graph.

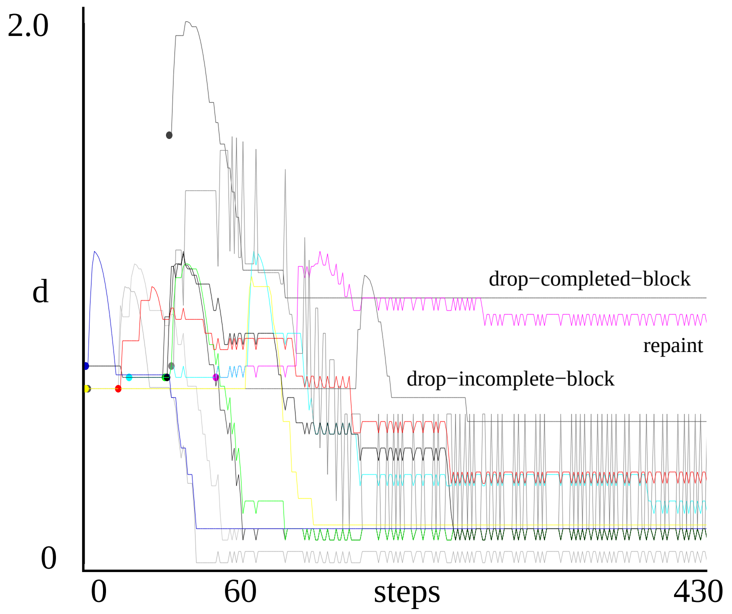

Figure 5.5. Desirability levels versus time for each of the unique action outcomes in the block paining robot domain.

the agent came across a new rewarding outcome. Soon after, the agent entered into a policy of alternatively planning for the original outcome and the new outcome, the two of which form a cycle, one leading into the other.

Empirical data from both the block painting domain and the randomly generated MDP domain indicate that the combination of our motivational system and a classical planner can effectively explore a domain, and as necessary (or in the absence of anything to explore) effectively exploit the affordances of the domain.

5.4 Reflexes

As we scaled up the complexity of the randomly generated MDPs in our experiments with the motivational system in the previous section, complete exploration became increasingly difficult for our agent. There are two reasons why domain complexity presents a challenge to exploration. First, there are simply more outcomes that must be discovered and modeled. Second, and perhaps more importantly, out-

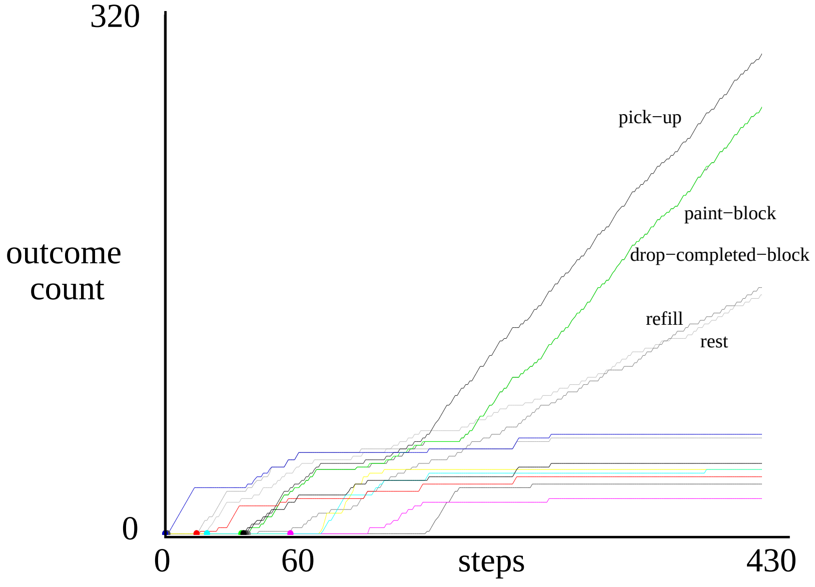

Figure 5.6. Outcome counts versus time for each of the unique action outcomes in the block painting robot domain.

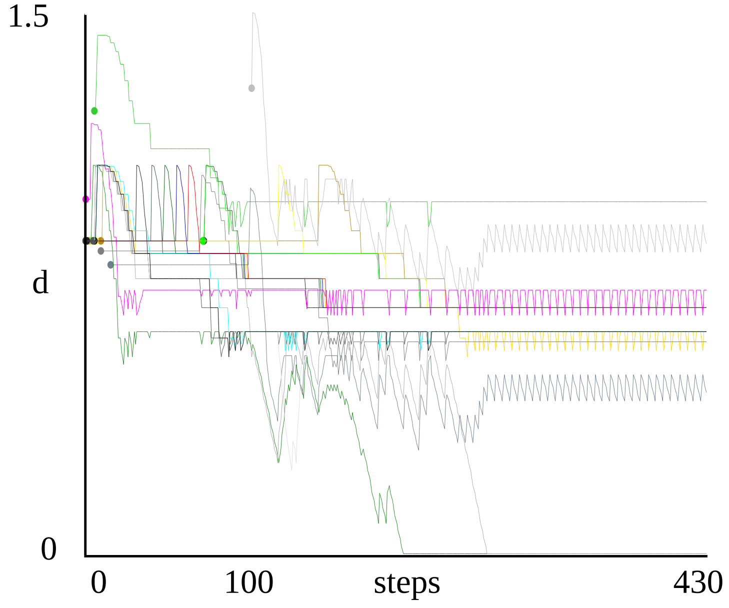

Figure 5.7. Desirability levels versus time for outcomes in a randomly generated Markov process domain with 5 actions, 22 outcomes, and 10 states.

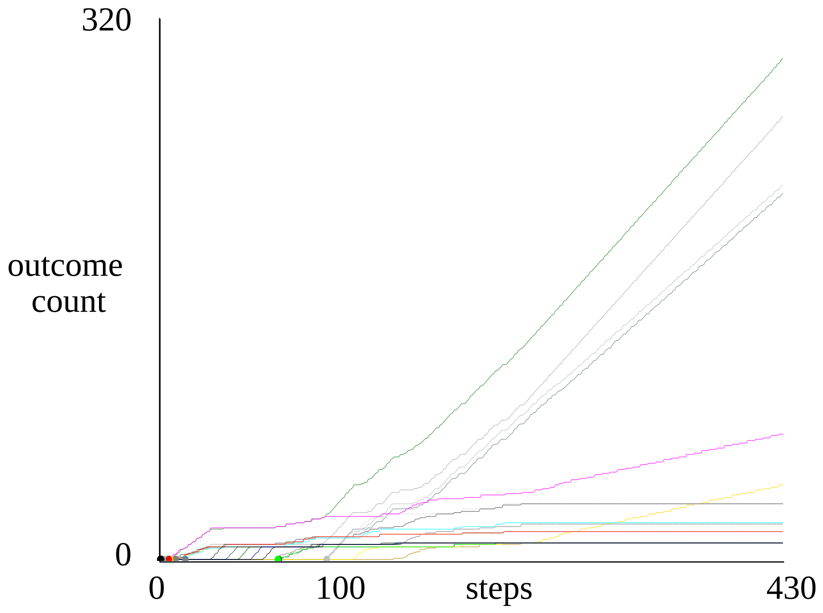

Figure 5.8. Occurrence counts for the outcomes in a randomly generated Markov process domain with 5 actions, 22 outcomes, and 10 states.

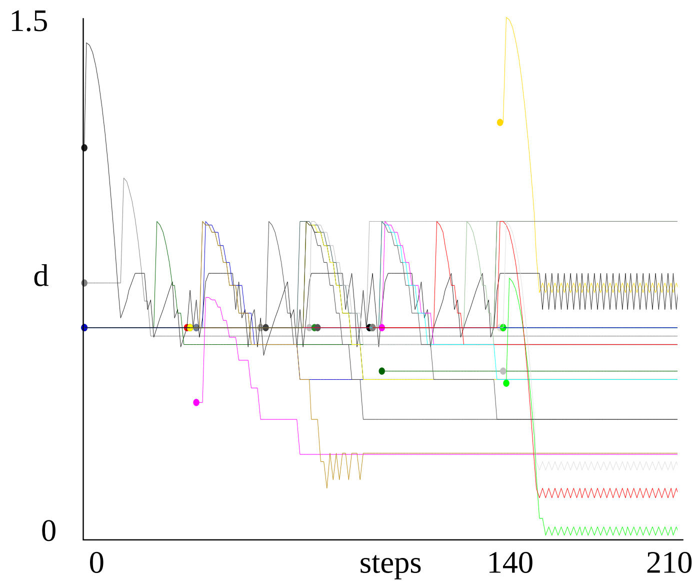

Figure 5.9. Desirability levels versus time for the outcomes in a randomly generated Markov process domain with 10 actions, 44 outcomes, and 20 states.

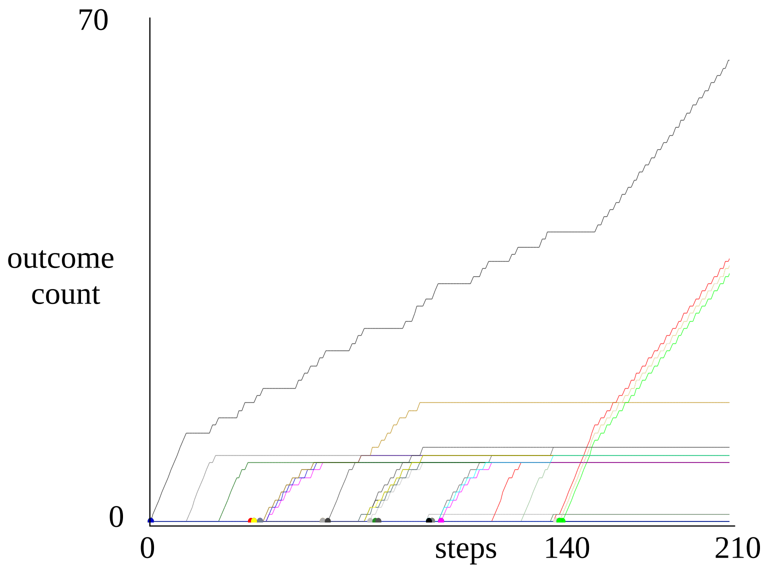

Figure 5.10. Outcome counts for the outcomes in a randomly generated Markov process domain with 10 actions, 44 outcomes, and 20 states.

comes can become increasingly isolated in the larger sensory spaces. The reason is best illustrated with an example MDP domain.

Consider the MDP of figure 5.11. It is nearly the same as the sample MDP of figure 5.11, except that two sates have been added. Note that state V627 can only be reached through one particular action taken in state V624. State V628 can only be reached through one particular action taken in state V626. Furthermore, state V627 has two transitions leading out to states distant in the graph. The result is that state V628 is "isolated" in the graph; during a random walk, this state is less likely to be visited on average than any other state.

In some cases, infrequent events or outcomes are unimportant. In these cases, we must accept that they may not be visited. Without a priori knowledge that they exist, there is no principled way to guarantee they will be discovered. In some cases, isolated outcomes may be important. Suppose the outcome that leads from V627 to V628 is the only outcome with a motivational payoff for some survival factor such as fatigue. It is vitally important for the agent to learn this outcome if it is ever to

$$\begin{tikzcd}
V626 \arrow[rr, bend left] \arrow[rrrr, bend left] \arrow[rrrrrr, bend left] & & V625 \arrow[dl] & & V622 \arrow[dr] & & V621 \arrow[ll] \arrow[llllll, bend right] & & \\
& V623 \arrow[ul] & & & & V624 \arrow[ur] \arrow[rr] & & V627 \arrow[ul] \arrow[r] \arrow[llllll, bend left] & V628 \arrow[lllllll, bend left]
\end{tikzcd}$$

Figure 5.11. A sample domain illustrated as an MDP.

attend to its fatigue drive. Similarly, suppose state V628 is a "bottleneck" state, and the only way to a large portion of the state space is through V628.

An example of a bottleneck situation is the *lift-debris* outcome in the GridSim domain. All of the outcomes associated with dropping debris, including dropping debris into a dropoff cell (which might produce a reward) are only possible after a successful *lift-debris* experience. The *lift-debris* is an important experience for an agent operating in the GridSim world to have, and it is an isolated action. The agent must be in a grid cell with debris with the bay empty for it to work, and only one of the six effectors will produce the result. On a random walk in an 8×8 grid, with 3 piles of debris, the *lift-debris* has less than 2% chance of occurring on any given step.

Now consider a robot domain, or worse yet, the world of a human infant. The possibilities for exploration are nearly limitless, and the stakes are greater if particular outcomes are not encountered. In these cases, a behavioral bias toward important outcomes provides a distinct advantage. In humans, we call these behavioral biases *reflexes*. Reflexes are automatic, unlearned reactions to stimuli from the environment, and human infants exhibit many that serve a variety of purposes in development. Here are a few reflexes common to human infants:

- **Sucking** When an object is near a healthy infant's lips, the infant will begin sucking immediately. This reflex helps the child get food, and is an instinct generally necessary for survival. The reflex also gives the infant experience sucking so that it can reproduce the behavior when this reflex disappears, usually by three weeks of age.

- **Grasp** The palmar grasp reflex is observed when the infant's palm is touched and when a rattle or another object is placed across the palm. The infant's hands will grip tightly. This reflex disappears the first three or four months after birth, and seems to be aimed at providing experience with grasping, and important bottleneck activity later in life.

- **Stepping** This reflex can also be observed in normal full term babies. When the infant is held so that the feet are flat on a surface, the infant will lift one foot after another in a stepping motion. This reflex usually disappears two months after birth and reappears toward the end of the first year as learned voluntary behavior. This too is apparently an evolutionary mechanism for strengthening important musculature and providing automatic experience with another activity important later in life.

Healthy human infants have many such reflexes. They provide for the infant's survival early on, produce behavioral bias for learning, and exercise specific musculatures so that they will be ready when behaviors that need them are learned. Our developmental schedule is remarkable, with reflexes that appear and disappear according to predictable timetables, and some, like the sneezing and blinking reflexes, that stay with us throughout our lives.

Reflexes provide such a developmental advantage where outcomes are unlikely or isolated that we include them in our model. An analog to the grasping reflex can be implemented for the GridSim agent and the Pioneer-2. They are specified as simple stimulus-response pairs and are integrated into the motivational system inside the action-taking cycle. Our simulated reflexes interrupt plans, as human reflexes do, and may appear and disappear as human reflexes do.

CHAPTER 6
EVALUATION

In chapter 1.4, we established two critical measures of performance for our system design. First, we established a metric for rating the *accuracy* of the system's models. Given a state s, and an action a, the model accuracy for an agent is the probability that the agent can predict the outcome of a taken in state s. Next, we established a metric called *facilitation* that reflects how well the planner works. Given a goal outcome o, and a set of initial conditions $IC(o)$ for o, facilitation measures the probability that the agent can arrive at $IC(o)$. Once the agent establishes $IC(o)$, the agent will achieve its goal outcome to whatever accuracy the models are working at.

We further refined the facilitation metric to isolate two major factors that influence overall facilitation. The statistic *coverage* measures facilitation as a percentage of the overall state space from which a given outcome o can be facilitated. The statistic *capability* measures the percentage of the total set of outcomes that can be facilitated from a given state.

An agent that becomes competent at acting in its environment should see improvements in accuracy, coverage, capability, and overall facilitation as it collects experiences interacting with its environment. In the remainder of this chapter, we will look first at how accuracy changes over time, and once it is established that a reasonable baseline for accuracy can be achieved by our system, we move on to coverage and capability. In each of the studies, we will implement our developmental architecture on a NESW agent operating in the GridSim domain (described in section 1.1.1) as a testbed. With this simulator, we can reduce and control sources of variance exogenous to our system (such as sensor noise and nondeterminism), as well as enumerate the effective state space. This will allow us to get truer measures of accuracy and facilitation, as well as make performance claims with fewer caveats pertaining to the domain.

6.1 Model Accuracy

A plan built by our system is only as good as the models it is built on. Ultimately, the final step in a plan (the *goal step*), designed to bring about a goal outcome, only has a chance to succeed if the initial conditions for that last step are accurate. In this section, we evaluate the modeling system to verify that models do improve as the system collects experiences, and that a level of accuracy can be obtained to warrant further experimentation with the planning system.

Recall the following definitions introduced in section 1.4:

- $o(s, a)$ refers to an actual outcome of taking action a in state s
- $p(o|s, a)$ refers to the true probability that outcome o will result when action a is taken in state s
- $\hat{p}_m(o|s, a)$ refers to the estimate of $p(o|s, a)$ made by some model m
- $\hat{o}_m(s, a)$ refers to an outcome predicted by model m ($\arg\max_o \hat{p}_m(o|s, a)$)

Now recall the concept of *disparity* between a predicted outcome $\hat{o}(s, a)$ and an actual outcome $o(s, a)$. In section 1.4, we defined a *winner-take-all* (WTA) disparity metric as:

$$d_{wta}(\hat{o}(s,a), o(s,a)) = \begin{cases} 1 & \text{if } \hat{o}(s,a) = o(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

We now define two additional disparity metrics based on the SSC schemes described in section 2.1.2. Recall that these schemes redescribe an outcome by analyzing and producing a symbol string for each of the sensor streams in an experience. A symbolic description of a GridSim experience, generated by the Delta-Simple filter, might

(bay-shape up)	(bay-color up)	
(n-camera-color nc)	(n-camera-shape nc)	(n-camera-dist nc)
(e-camera-color nc)	(e-camera-shape nc)	(e-camera-dist nc)
(s-camera-color nc)	(s-camera-shape nc)	(s-camera-dist nc)
(w-camera-color nc)	(w-camera-shape nc)	(w-camera-dist nc)
(ccs-shape down)	(ccs-color down)	

Table 6.1. A DELTA-SIMPLE outcome description.

appear as shown in table 6.1. This table lists each sensor along with the SSC-generated symbolic label for that sensor time series in the given experience. The symbolic description expresses that the `bay-shape` and `bay-color` sensors exhibit upward trends, `ccs-shape` and `ccs-color` both show downward trends, and the directional cameras all exhibit no change. This is the characteristic DELTA-SIMPLE description for the experience of a GRIDSIM agent lifting an item into its cargo bay.

A convenient shorthand for outcomes described with the DELTA-SIMPLE filter is to use a single letter abbreviation for each sensor trend, seperated by dashes, in the order presented in table 6.1. The following string is shorthand for the description of table 6.1:

```
U-U-N-N-N-N-N-N-N-N-N-N-N-N-D-D
```

The two disparity metrics are based on comparing symbolic profiles, sensor by sensor. The first we call *Hamming disparity*, and is an adaptation of Hamming distance to our symbolic representation.[1] The Hamming disparity between two outcome strings is defined as:

$$d_h(\hat{o}(s,a), o(s,a)) = \frac{\sum_{x \in \mathcal{S}} d_{wta}(\hat{o}_x(s,a), o_x(s,a))}{|\mathcal{S}|} \qquad (6.2)$$

Where $o_x(s,a)$ refers to the x^{th} symbol of outcome o and the winner-take-all disparity between two symbols is 0 if they match, 1 if they do not. Hamming sums

[1]Richard Hamming. Coding and Information Theory. Prentice-Hall, 1980. ISBN 0-13-139139-9.

this sensorwise disparity over all sensor pairs, essentially counting the number of sensorwise mismatches. This count is divided by the number of sensors to limit Hamming disparity to the range $[0 \ldots 1]$. As an example, consider the following two outcomes:

```
U-U-N-N-N-N-N-N-N-N-N-N-N-N-D-D
D-D-N-N-N-N-N-N-N-N-N-N-N-N-N-N
```

These outcomes differ in 4 of the sensor encodings (the first two and the last two), and so the summation in equation 6.2 sums to 4. There are 16 sensors in the outcomes, and so the Hamming disparity between these two outcomes is $\frac{4}{16}$, or 0.25.

The second disparity metric extends the idea of Hamming disparity to consider only those sensors in which there is behavior of note. This stems from the philosophy described in section 2.1.2 that inspired the SSC classifiers: for the purposes of planning and acting, "interesting" regions of time series are those sections that have non-zero slope and predictable trend. We call this second metric HI distance, and it is defined as follows:

$$d_h(\hat{o}(s,a), o(s,a)) = \frac{\sum_{x \in \mathcal{S}} d_{wta}(\hat{o}_x(s,a), o_x(s,a))}{\sum_{x \in \mathcal{S}} I(\hat{o}_x(s,a), o_x(s,a))} \tag{6.3}$$

where

$$I(s_i, s_j) = \begin{cases} 1 & \text{if } (s_i \neq \texttt{N} \wedge s_j \neq \texttt{N}) \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

HI disparity effectively ignores sensor matches in which nothing happens in a particular sensor for either experience. The HI disparity of the two experiences in the two experiences above is 1.0, since all of the sensors of interest are mismatched.

Each of the three disparity metrics has its place in our evaluation. Winner-take-all represents the accuracy with which a model can produce an *exact* outcome. When an agent posts an outcome as a goal, it is trying to elicit that outcome, and not some variation. Therefore, winner-take-all accuracy provides a good gauge for a maximal rate that an agent could achieve its goals if its plans were perfect. Hamming and HI distance represent the modeling system's ability to get outcomes partially correct. For the purposes of planning, it is often necessary to change some part of the sensory state, and thus it is not always necessary to elicit an exact outcome as long as the pertinent sensors change as anticipated. In partially observable domains, it is not always possible even with good models to predict every sensor change with 100% accuracy. Hamming disparity, then, provides a measure of accuracy with partial credit, and HI disparity gives a measure that ignores uninteresting predictions. Since planning concerns itself mainly with change, HI distance is perhaps the best overall measure of model accuracy for models used in planning.

In our experiments, we track disparity measures as our agent collects experiences. Disparity curves are generated via the following procedure. A training set of experiences is collected as an agent takes actions according to some control policy. At regular intervals, the agent stops collecting training data, and enters into a testing phase. In the testing phase, the agent engages in a suite of predict-and-verify trials in which the agent makes a prediction $\hat{o}(s,a)$ for its current state and some action a, executes a, and records the resulting experience $o(s,a)$. Disparity statistics are calculated and recorded. With GridSim, the state space for a particular grid configuration can be enumerated. The state space consists of each of the traversable 36 cells on the 8×8 grid, with the cargo bay empty or occupied, for a total of 72 states. The agent generates and verifies a prediction for each of the 6 actions in each of the 72 states for a total of 432 trials per test suite. In domains where the state space cannot be enumerated a priori, test suites may be generated by random sampling.

6.1.1 Trial Descriptions

Between the our system and the environment, there are many parameters that can affect both the steepness and lower bound of disparity curves. Our goal is to isolate the most relevant factors, control for them in our experiments, and understand the nature of their influence on accuracy so that we may make statements about the accuracy of the modeling system under varied conditions. In so doing, we can get a handle on how difficult the process of building reliable behaviors will be. Here are the primary factors with influence over model accuracy:

- The **domain**:
  - The number of possible outcomes
  - Nondeterminism and partial observability
  - The degree to which outcomes can be generalized (the ratio of unique sensory states to possible outcomes)
- The **high-level controller** and any biases on exploration it provides.
- The **modeling system**:
  - The outcome classification scheme; both in the number of outcomes it produces and its sensitivity to noise and the possibility of over and under-generalization
  - The ability of the inductive (tree-building) component to produce predictive models

In the experiments that follow, we attempt to get a handle on these effects. It is worth describing the various trial conditions we use to do that.

We use the GridSim domain in each of the trials. This is a deterministic simulator in the sense that for a given start state and action, there is a single process that generates the resulting sensor patterns. There are partially observeable situations, however. Consider the four cameras of a GridSim agent when the agent moves in a particular direction, perhaps north. The north-facing camera will continue to track whatever is ahead of the agent. Similarly, objects to the south will get more distant, but whatever objects are to the north and south tend to stay within view of the cameras facing those directions. To the east and west, however, objects appear on one side of the visual field and pass through to the other side. Imagine riding in an automobile and looking out one of the side windows. Objects appear in the visual field and pass through. There is little in the way of visual cues to tell you that a traffic sign is about to appear in on the left-hand side of your visual field, then pass through and disappear off to the right. Without some kind of clue as to what might appear in the east and west facing sensors as it moves north, our GridSim agent cannot predict the behavior of those sensors either. In general, the sensors oriented -90 and 90 degrees from the direction of movement are difficult to predict.

In response to the partial observeability problem, we offer two flavors of the GridSim simulator: a *static* configuration and a *dynamic* configuration. In the static configuration, debris always appears in the same cells. In the dynamic configuration, debris can appear in any cell that is not already occupied. These conditions offer an interesting contrast; in the static condition, the modeling system can learn "landmarks" that can be used to predict the side-facing camera behavior. Consider the GridSim configuration pictured in figure 1.3. This is the static configuration. If the agent were to move south, the west-facing camera would lose sight of the debris currently adjacent to the agent, and then pick up the next piece that is currently to the south and west. In the static configuration, the agent can count on the second piece of debris being there, and it can use the first piece as a cue to predict its appearance. Specifically, the agent can use the conditions

$$\texttt{(scam-dist 2)} \wedge \texttt{(scam-shape 1)} \wedge \texttt{(wcam-dist 1)} \wedge \texttt{(wcam-shape 2)}$$

to predict the appearance of the second piece of debris. In the dynamic environment, the debris to the south and west is not guaranteed to be there, and so the agent cannot be expected to predict the appearance of a new piece of debris to the west. In the dynamic case, the agent cannot use landmarks. It can use fixtures, such as the objects along the east and west walls, and the walls themselves, but since the location of debris is not fixed, it no longer can be used as a cue for predicting the peripheral cameras' behavior. On the other hand, operator models learned in the dynamic case should be more general since they will not generally be based on landmarks. This should mean that the models learned in the dynamic configuration will transfer better to changes in scenery. We run trials with both the static and dynamic configuration of GridSim.

When the debris location is dynamic, the number of outcomes also increases. It allows a wider variety of interaction between the agent, its cameras, and the debris. Similarly, if as we increase the size of the GridSim grid, we increase the number of possible sensory states as well as the number of possible outcomes. We run trials with the basic 8×8 grid as well as a larger 14×14 grid.

6.1.2 Random Walk

We begin our accuracy evaluation with the most primitive conditions. The agent is placed in the 8×8 GridSim environment in its static configuration. $2,000$ training experiences are generated by repeatedly selecting actions at random and executing them. Wandering randomly in this fashion is called a *random walk*, and we will refer to the set of experiences generated by random walk as RW data hereafter. Experience outcomes are generated using the Delta-Simple filter. Each time the agent collects 100 experiences, the random walk is stopped, and accuracy test is performed, and then the random walk is resumed.

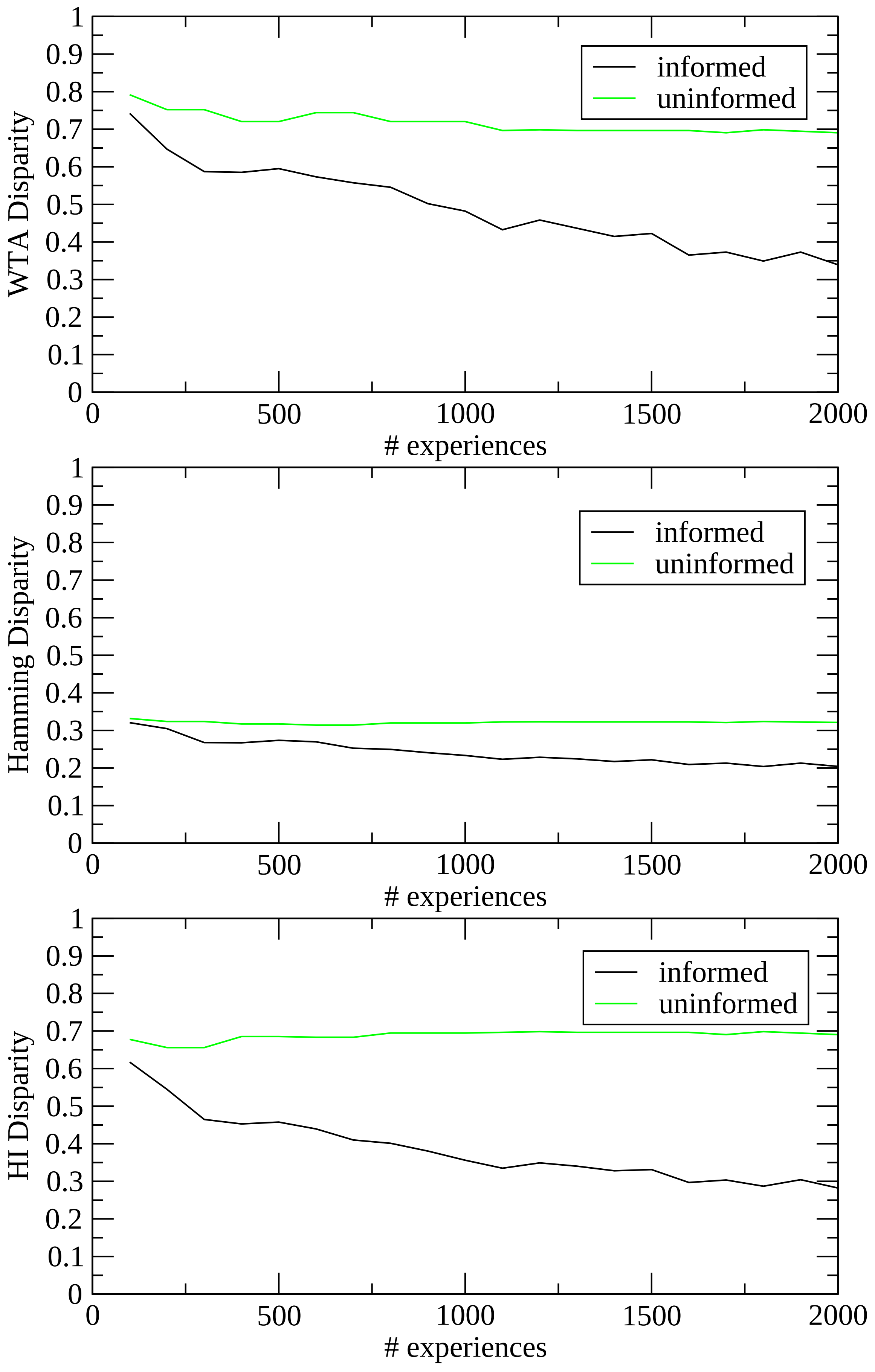

Figure 6.1. WTA, Hamming, and HI disparity measured at regular intervals during a random walk of the 8×8 GridSim simulator with the NESW agent.

Figure 6.1 shows each of the three disparity metrics, as they change over time, for RW data. From left-to-right, they are WTA, Hamming, and HI disparity. Each plot shows the disparity of an informed agent, using our modeling system, versus that of an uninformed agent that simply predicts the most prevalent class. In each case, there is a marked advantage to building models of the relationship between start state and outcome. The WTA plot shows that after $2,000$ randomly generated experiences, our agent can predict 65% of the outcome classifications over the space of 432 action/state pairs. Looking at sensor-by-sensor outcome classification, as we do in the Hamming disparity metric, performance is at a level beyond the 65%, indicating that while 35% of outcomes cannot be predicted exactly (WTA SCORE, the behavior of certain sensors in those outcomes can be accurately predicted. After $2,000$ experiences, approximately 80% of all individual sensor classifications are correctly predicted. Finally, HI disparity shows that after $2,000$ experiences, better than 70% of all sensors that change during the course of an experience can be accurately predicted.

Before moving on to other experimental conditions, it is worth delving into the RW data a bit further to make two observations. First, consider the plot of Hamming disparity of figure 6.1. Note that the uninformed performance here appears to be fairly good. This is because for each action, some number of sensors will exhibit no change on a regular basis. For example, in each of the move actions, the bay sensor cannot change. These sensors are “uninteresting” in the SSC sense because they do not change, but they are an easy way for the uninformed modeler to get some sensors correct and improve its Hamming score. Since the uninformed modeler will get the sensors that change correct far less frequently, the true weaknesses of its performance are exposed in the WTA and HI disparity plots. For this reason, we will skip Hamming plots as we progress through different experimental conditions. Instead, we will focus exclusively on HI disparity, since it most closely targets the measure we are interested in: whether or not a modeler can predict those sensors that change.

Next, consider the GridSim domain and its actions. The effects of lift and drop actions are fully observeable, while each of the move actions is potentially not. That is, the sensory state prior to acting may not be enough to predict the resulting behavior in all 16 sensors. It is useful, then, to analyze the prediction accuracy of the individual actions seperately, as we have in figure 6.2. Clockwise, from upper-left, are plots of move-w, move-e, lift, drop, move-s, and move-n. Note the marked difference in the nature of the lift and drop actions versus the move actions. The lift and drop actions have each only a few observeable Delta-Simple outcomes; lift has two and drop has three. Thus, the models required to generate them are simple and can be converged on quickly. Their effects are fully observeable and so a disparity of zero is possible. On the other hand, there are in the range of 20-25 Delta-Simple outcomes per move action. The outcome is highly dependent on starting state and may require extensive use of landmarks in the visual sensors to disambiguate all the observeable effects. In many cases, a particular outcome of a move action is only attainable from a single grid cell in the simulation. The models required can be extensive, the modeler is more suceptible to overfitting with a shortage of examples, and in some cases a full outcome specification can never be fully disambiguated from the start state. In each of the subsequent accuracy experiments, we will break down accuracy by action.

6.1.3 Entropy-based Exploration

A random walk will more or less guarantee a uniform distribution of actions taken over a period of time. If our agent has enough time, it will eventually get representatives of every outcome class, and should have the data it needs to build good models. But the ease with which the modeling system can pin down the lift and drop actions indicate that a random walk is not necessarily the best policy for generating examples

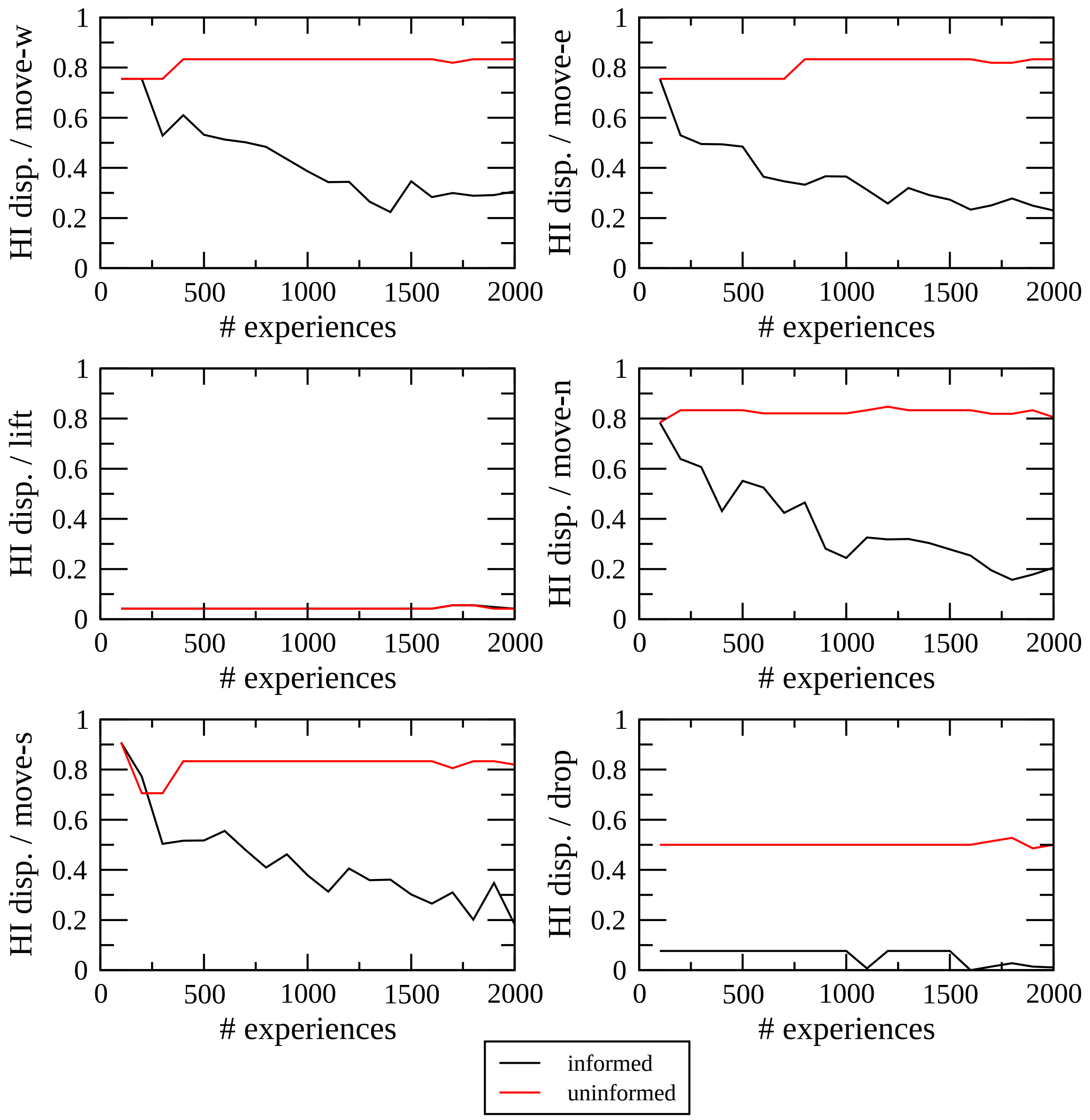

Figure 6.2. HI disparity versus experience count, broken down by action, for the 8×8 GridSim random walk data.

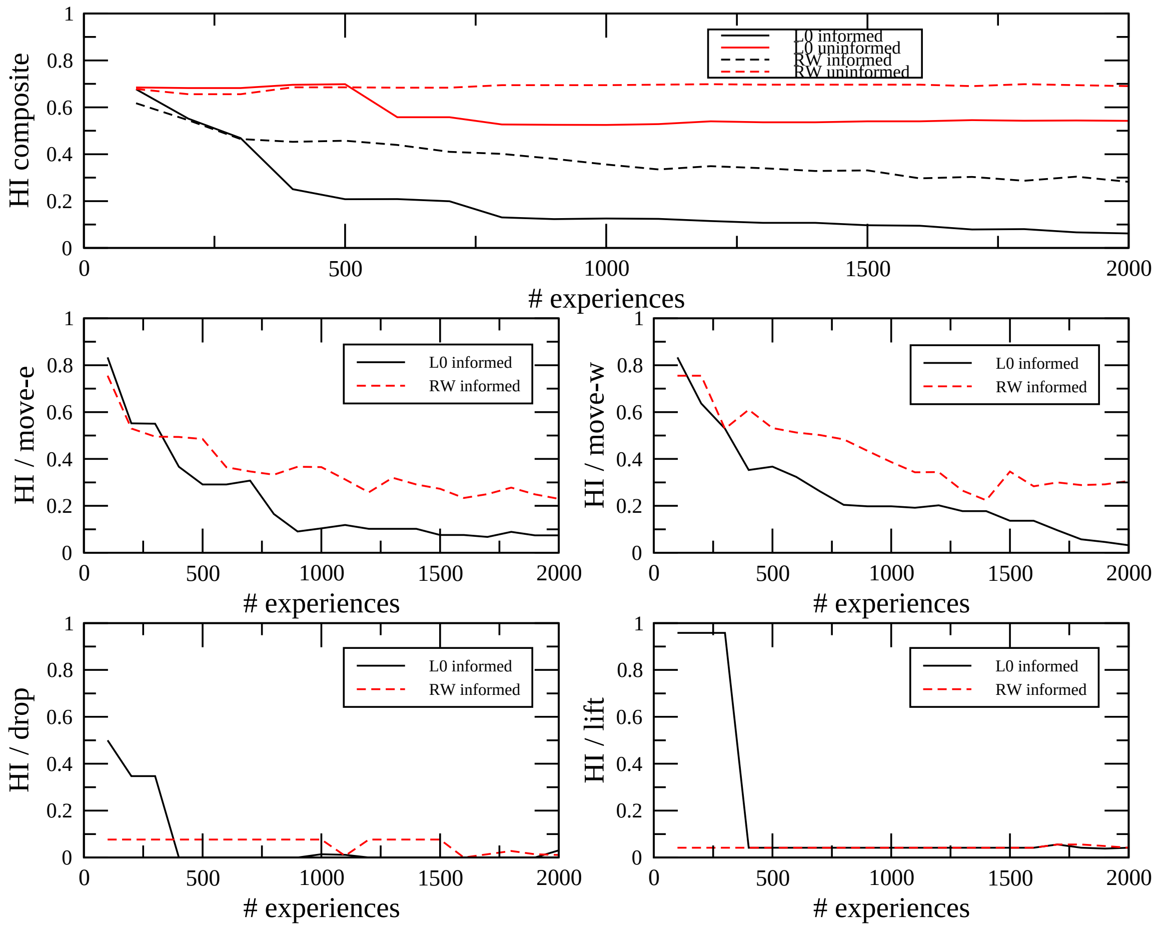

Figure 6.3. HI disparity versus experience count, for an agent in the 8×8 GridSim udnder the L0 and random walk exploration policies. At the top, composite HI disparity using the L0 controller data versus random walk data. Below, HI disparity for four representative actions.

to learn from. Furthermore, if the agent is in a grid cell where the model for move-e has a 100% accuracy, then executing another move-e might be a waste of time.

To test the hypothesis that more efficient learning should be possible, we generated $2,000$ experiences using the L0 control policy first introduced in section 5.1. To recap, L0 control is a three-tiered control policy:

1. **if** a reflex is active, execute the response
2. **else** execute the maximum-entropy action $\arg\max_a h(\hat{o}(s, a))$
3. **if** there is a tie for maximum entropy, select an action at random

The addition of reflexes guarantees that important outcomes are experienced, while the entropy-based action selection focuses exploration on the areas with higher entropy in the distribution of predicted outcomes, driving the system towards disambiguation. Figure 6.3 shows time series HI disparity measures of the modeling system using experiences generated by the L0 controller.

The graph on the left shows overall HI disparity as it changes with the number of collected experiences, for both the informed and uninformed cases. The dashed lines are the same measures for the RW dataset, taken from figure 6.1. The four graphs to the right of figure 6.3 break down the HI measures by action; clockwise from upper left they are graphs for RW data versus L0 data for the move-w, move-e, drop, and lift.

As anticipated, the L0 controller appears to accelerate learning in every case except lift and drop. Overall HI disparity was at 35% in under 500 experiences with the L0 controller, a level that took the random controller $2,000$ experiences to achieve. After $2,000$ experiences, the models generated using L0 data were missing less than 10% of all predicted sensor outcomes. The combination of reflexes, which bias the agent toward important outcomes, and the bias toward taking high-entropy actions, the controller effectively focuses the agent towards opportunities to learn and refine models.

6.1.4 Dynamic Configurations

To understand the possible advantages that learning in a statically configured environment might give the modeling system, we conducted accuracy tests for the dynamic case. We generated $2,000$ experiences with the L0 control policy in an 8×8 grid in which debris was randomly placed. Every 400 experiences during the collection of training experiences, the debris locations would be reset to new, random locations. In each dynamically configured grid, there were the same three pieces

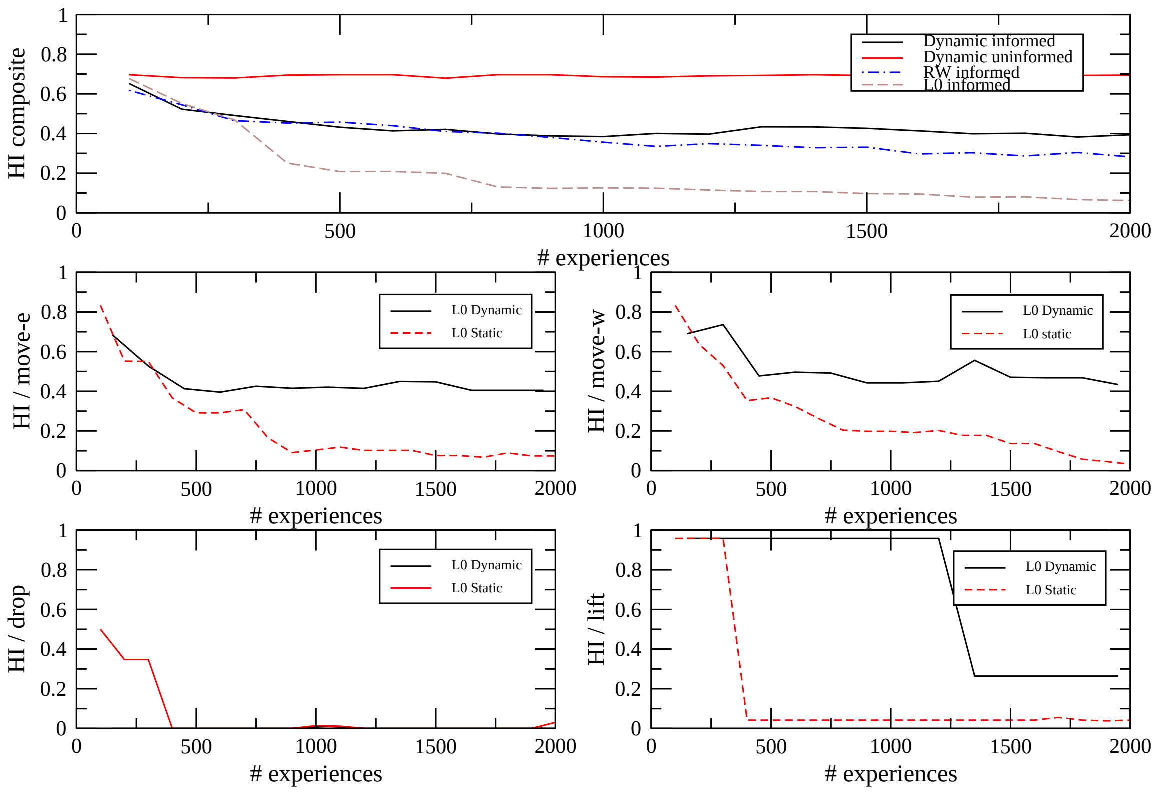

Figure 6.4. HI disparity versus experience count, for an agent in the 8×8 GridSim trained under the L0 exploration policy in the dynamic configuration. At the top, composite HI disparity for the dynamic set versus random walk and L0 sets previously described. Below, HI disparity broken down by action for four representative actions.

of debris, only their locations were changed. Accuracy tests were performed in the statically configured environment, so that the both test conditions would be evaluated against the same environment.

Figure 6.4 shows HI disparity plots for the dynamic case versus the static case. The leftmost plot shows composite disparity. For training on dynamically configured domains, disparity tends to converge at around 0.4, indicating that over the domain, the dynamic based models can predict about 60% of sensor behavior. Rate of learning was similar to that of the static based random walk, although the random walk agent continued to learn after the dynamic agent leveled off. The static based L0 agent outperformed the dynamic based agent by an edge of 0.3 in disparity after 2,000 experiences.

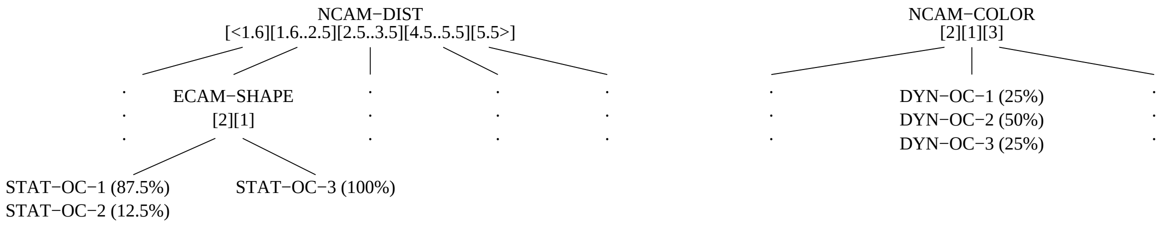

Figure 6.5. Sample outcome classification trees. On the left, a tree built using experiences in the statically configured domain. On the right, a tree trained in the dynamic configuration.

The intuition behind comparing dynamically and statically configured environments was that a statically configured environment would provide "landmarks" for an agent to make predictions that could otherwise not be made. We identified the move actions in the GridSim simulator as having the qualities of partial observability. That is, the cameras facing 90 and -90 degrees from the direction of movement cannot be predicted with 100% accuracy because objects that could not be previously sensed might appear into the view of those cameras. By keying off of landmarks in the environment, though, these appearances might be predicted in the static configuration. In the dynamic configuration, debris cannot be used as landmarks.

Plots of HI disparity for individual actions to the right of figure 6.4 solidify the intuition further; The plots for the move actions show a distinct seperation between static and dynamic training. Disparity bottoms out at about 0.4 in the dynamic case, and under 0.1 in the static case. To illustrate the difference more clearly, figure 6.5 shows representative portions of initial condition trees built on 105 move-n experiences [2]. To the left is an excerpt from the static tree, to the right is an excerpt from the dynamic tree. Note that the static tree, even with only 105 experiences to work with, has elaborated the initial condition space more fully than in the dynamic case. The static tree has brought into consideration the shape of any object sensed

[2]The entire trees are not shown for the sake of brevity. The fully elaborated static tree has 11 leaves and 5 decision nodes. The fully elaborated dynamic tree has 4 leaves and 2 decision nodes.

	BAY-COLOR	BAY-SHAPE	SCAM-COLOR	SCAM-SHAPE	SCAM-DIST	NCAM-COLOR	NCAM-SHAPE	NCAM-DIST	WCAM-COLOR	WCAM-SHAPE	WCAM-DIST	ECAM-COLOR	ECAM-SHAPE	ECAM-DIST	CCS-COLOR	CCS-SHAPE
DYN-OC-1	N	N	N	N	U	N	N	D	N	N	N	N	N	N	N	N
DYN-OC-2	N	N	N	N	U	N	N	D	N	N	N	D	U	D	N	N
DYN-OC-3	N	N	N	N	U	N	N	D	D	D	N	U	U	D	N	N

Figure 6.6. The three outcomes shown in the tree generated from the dynamic configuration of figure 6.5.

by the east facing camera to disambiguate outcomes. It has brought into play a landmark of the static configuration. The tree built in the dynamic configuration cannot resort to such tactics, and so the tree is less developed.

Figure 6.6 shows the actual Delta-Simple descriptions of the outcome classes represented in the leaf shown in the tree built on data the dynamically configured domain. The sensors that cannot be resolved are the east and west facing cameras; those 90 and −90 degrees from the direction of movement. Results are similar for all four move actions. In each, there are unresolvable ambiguities in the camera sensors facing 90 and −90 degrees to the direction of movement. There are 3 sensors in each direction, accounting for 37.5% of the available sensors. This accords with the apparent 0.4 lower bound on HI disparity in the dynamic case.

6.1.5 SSC **Variants**

As noted in section 2.1.2, sensitivity to noise or overfitting in the classification system can manifest as outcome class definitions that are too general or too specific. If overly general class definitions prevail, experiences generated by two or more different processes (and thus having different initial conditions) will take the same label. If

	BAY–COLOR	BAY–SHAPE	SCAM–COLOR	SCAM–SHAPE	SCAM–DIST	NCAM–COLOR	NCAM–SHAPE	NCAM–DIST	WCAM–COLOR	WCAM–SHAPE	WCAM–DIST	ECAM–COLOR	ECAM–SHAPE	ECAM–DIST	CCS–COLOR	CCS–SHAPE
DELTA SIMPLE	N	N	D	D	U	N	N	N	N	N	D	U	U	N	D	D
DELTA	N	N	2D	D	4U	N	N	N	N	N	D	2U	U	N	4D	3D
SSC1	N	N	D	D	_⌐‾	N	N	N	N	N	D	U	U	U	‾¬_	D

Figure 6.7. A single move-w experience in the GridSim simulation, represented by each of the three ssc outcome classifiers, Delta-Simple, Delta, and ssc1.

overly specific class definitions prevail, experiences generated by the same process will have different class labels. In either case, the result is loss of accuracy.

The accuracy test gives us our first opportunity to compare and test the ssc outcome classifiers. The intuition behind these classifiers is to reduce the opportunity for overfitting and sensitivity to variations in sample size, outside parameters, or noise. To this point, all of our tests are with the Delta-Simple filter, and the achievable accuracy levels indicate that this filter produces outcome classifications that are fairly close to the true generating processes behind the GridSim simulator. Here, we revisit the Delta and ssc1 filters and consider the accuracy levels that are possible with their outcome classifications.

Recall the three ssc schemes described in section 2.1.2. Delta-Simple assigns a symbol to each sensor of an experience based on whether its value goes up or down over the course of an experience. Delta elaborates on the simple scheme by including a magnitude rating indicating *how much* each sensor changes. ssc1 uses an even further enhanced representation based on piecewise linear fitting that allows the filter to express arbitrary sensory *shapes*, made up of segments that increase, decrease, level off, or include discontinuities. Figure 6.7 shows each of the three outcome classifiers' interpretations of a single GridSim experience.

The experience in figure 6.7 is a move-w experience in which the agent moves away from a piece of debris. This is reflected in the behavior of the ccs (current cell sensor), and typifies the differences between the three classifiers. The DELTA-SIMPLE filter simply notes that as the agent moves off of the debris, its ccs loses track of the debris, and the shape and color sensors decrease to their baseline values[3]. The DELTA filter is more expressive, classifying the ccs changes as `4D` and `3D` indicating a decrease of 4 units and 3 units, respectively. DELTA outcome labels allow more subtle distinctions between colors and shapes to be made. Finally, the SSC1 classifier introduces the idea of a *discontinuity* – a sharp increase or decrease in value – in the ccs-color sensor.

Additional expressiveness will allow an agent to make more detailed distinctions during planning. The additional expressiveness of the DELTA formulation would allow the planner to distinguish moving onto *red* debris from *black* debris. If there were some utility in transporting the black stuff and not the red stuff, then the DELTA scheme provides a relative utility over DELTA-SIMPLE. However, if moving to a more expressive representation impacts the model building scheme negatively, then the tradeoff may not be worth it. Therefore, we are interested in the accuracies of these three schemes.

Figure 6.8 shows a comparison of the three schemes. On the left, the number of unique outcomes produced by the individual classifiers. Both DELTA and DELTA-SIMPLE top out in the neighborhood of 100 unique class labels for the 8×8 grid. SSC1, however, exceeds 200 by $2,000$ experiences, and is possibly not done. Not only does the increased expressiveness allow the SSC1 filter to make more distinctions, but the piecewise linear fit algorithm introduces a source of variance. This can be verified empirically, as the SSC1 algorithm may parse multiple experiences generated from a single action taken from a fixed start state in different ways. This effectively

[3]Recall that shape and color are encoded as integer scalars in the GRIDSIM domain, with 0 indicating the absence of anything visible to report.

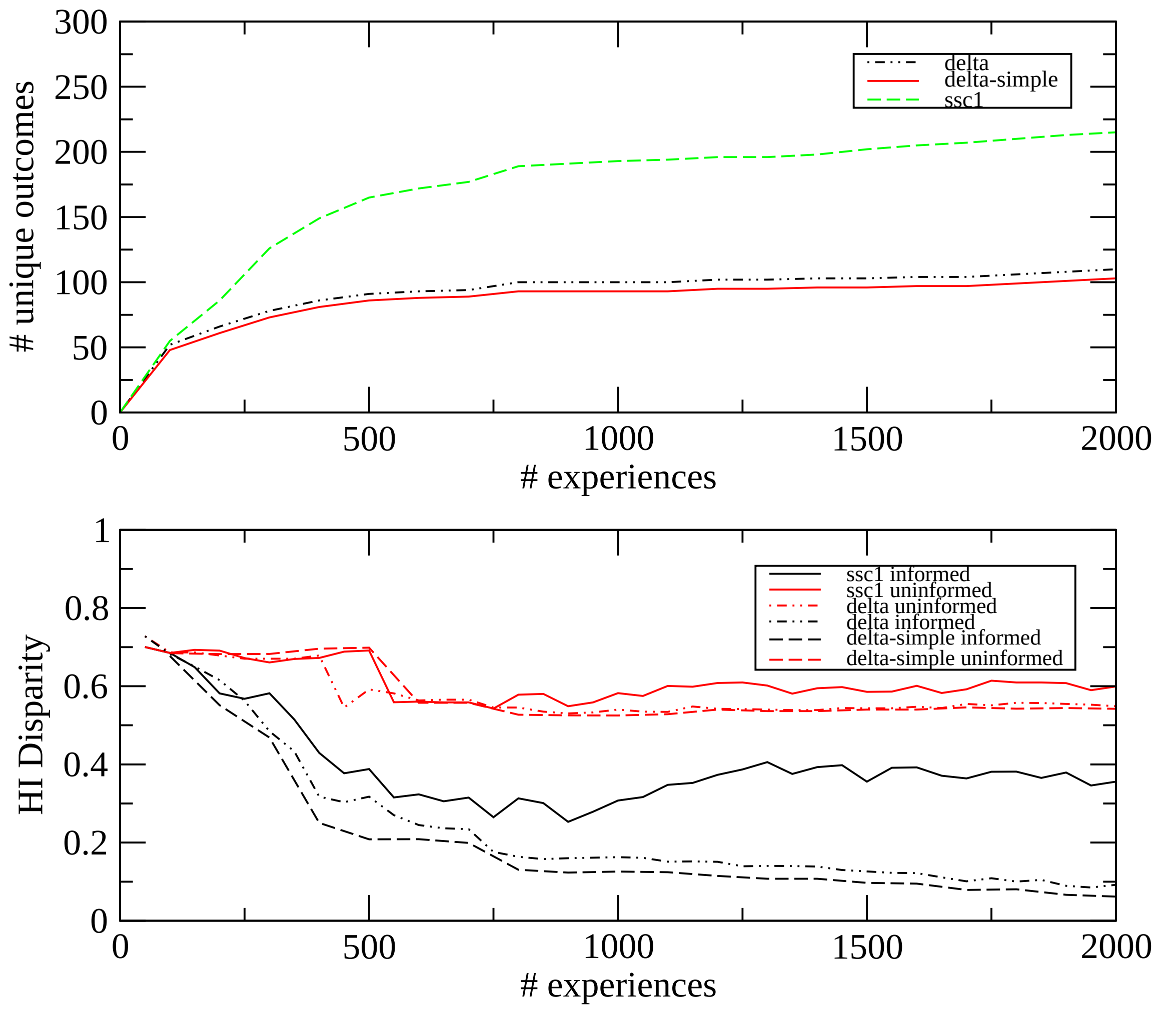

Figure 6.8. A comparison of the three outcome classification filters: DELTA-SIMPLE, DELTA, and SSC1. Above, the number of unique outcomes produced by each filter run versus experience count. Below, informed and uninformed HI disparity scores for each filter over the same 2,000 experience run.

introduces ambiguity into the modeling system, which is evident in the plot to the right of figure 6.8. This graph plots HI disparity for the three outcome classifiers versus the number of experiences in the statically configured environment. For the first 700 experiences, the performance of SSC1 stays within a reasonable margin of the two delta schemes. But, as the number of experiences increases, so increase the number of opportunities for the piecewise linear fitting algorithm to interpret two similar experiences two different ways. By $1,000$ experiences, the disparity plot for SSC1 starts to diverge, actually taking a significant turn for the worse before stabilizing at about 0.4.

These graphs show that there is a cost to increased discrinability. The most expressive classifier, SSC1, performs the worst in accuracy tests. In the GRIDSIM domain, it is not clear that the increase in expressiveness would give an agent an advantage during planning, since GRIDSIM outcomes are relatively simple. The additional expressiveness is unnecessary and a detriment in this case and for the purposes of our experiments with the GRIDSIM simulator, we will use the DELTA-SIMPLE classifier. However, in more sophisticated domains, we still suspect that there may be cases in which qualitatively distinct outcomes would be classified together by the DELTA and DELTA-SIMPLE schemes. Where this is the case, accuracy and planning may be limited by the modeling system's tendency to group these outcomes together. Moving between levels of expressiveness during development is a distinct possibility for future work, should it ever become evident that a lack of expressiveness is limiting performance.

6.1.6 A Larger Space

With the eventual goal of implementing our develpmental system on increasingly sophisticated platforms, scalability is a concern, particularly of the modeling system. Some of the properties of decision tree induction have been studied and the effects of input parameters such as dataset size and class complexity are known [51, 58]. In this section we consider the scalability of our modeling component.

We implemented a larger, 14×14 grid similar to the 8×8 grid described throughout our experiments. Where there are 36 grid cells that an agent can manuever into in the smaller grid, there are 144 in the larger one. Where there are 3 pieces of debris in the smaller, statically-configured grid, there are 6 in the larger grid. The action space, and locations of the dropoff cells are similar for the two domains.

Figure 6.9 shows HI disparity measures for an agent operating under the L0 control policy in the larger grid. At right of the figure are the raw disparity against the number of experiences as the agent collected 8,000 experiences. The curve appears similar in shape to that of the smaller grid. By 2,000 experiences, the agent has reduced average disparity to under 0.2, indicating that over 80% of the agent's sensor behavior can be accurately predicted across the grid. By 8,000 experiences, accuracy reaches the neighborhood of 95%. To the right of figure 6.9, HI disparity plots for the 8×8 grid and the 14×14 grid are overlaid, with the number of experiences for the larger grid normalized by the 300% increase in grid size. The plots are virtually indistinguishable, with the larger grid showing an apparent advantage due to a greater opportunity to generalize in the larger domain.

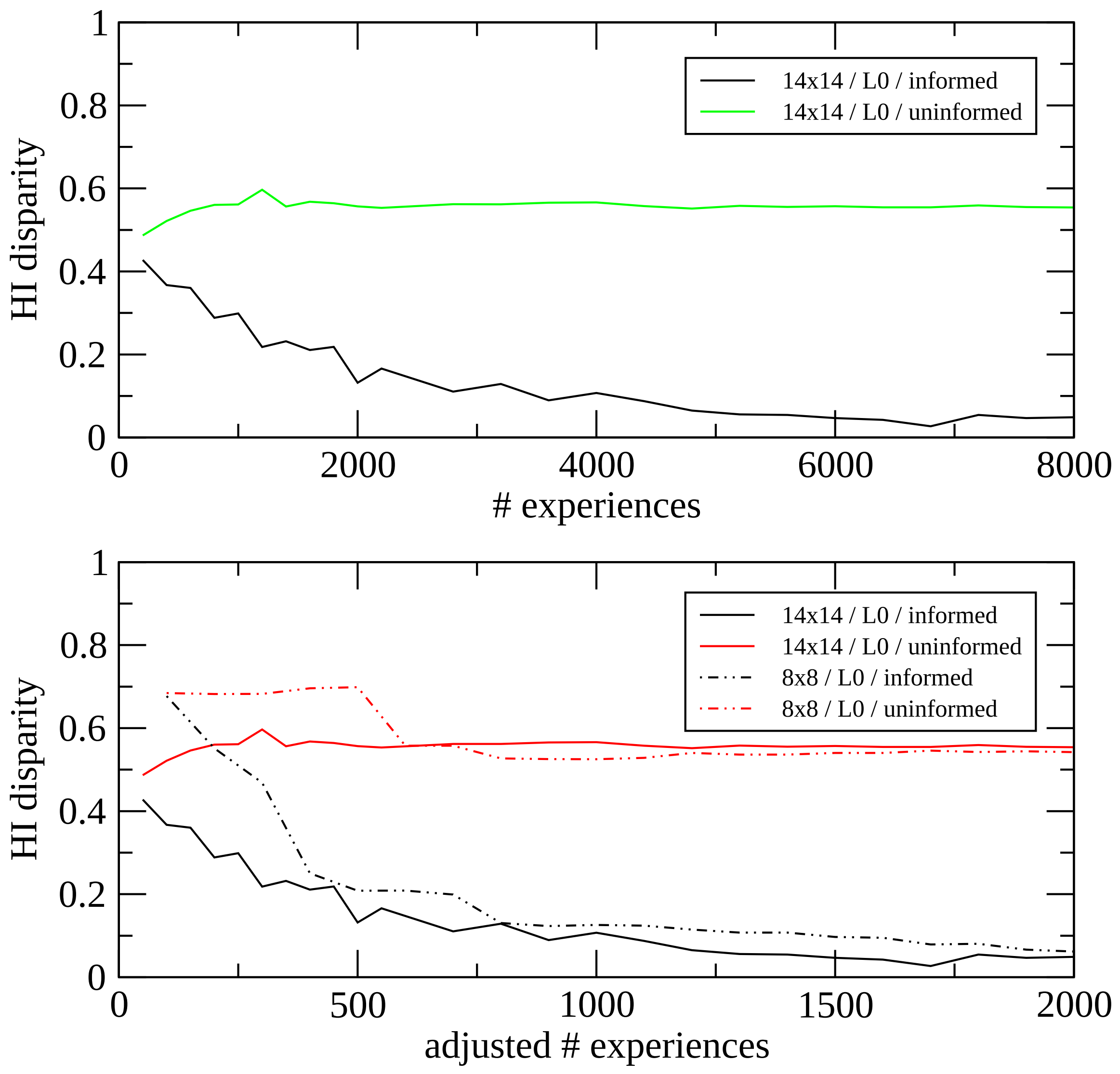

Figure 6.9. HI disparities for an agent operating in a larger 14×14 grid. Above are HI scores for an agent operating under the L0 control policy, as they change over the first $8,000$ experiences of exploration. Below, the same plot overlaid onto the plot of figure 6.3 for the 8×8 grid, with the X axis for the 14×14 data scaled by .25 to account for the increase in the number of states.

6.2 Facilitation

The accuracy statistic tells us how well our system can predict an outcome given an action and a starting state. Conversely, we can predict the probability that some action will produce a goal outcome given an agent is in a particular state. If our agent is in a state that maximizes the probability of its goal outcome, then the accuracy metric tells it everything it needs to know about its expectations of achieving its goal. This is not always the case; often an agent needs to string together a sequence of two or more actions to get it into a state in which its goal is a high probability result. This is the basis of our system: that an agent can identify regions of the state space where a goal is a highly probable outcome, produce a sequence of actions that will get it there, and package them up into an activity that can be used and reused any time the goal comes up. In this section, we will consider those situations in which plans are necessary, and the planner that produces them. Specifically, we will ask how well the planner is able to *facilitate* goal outcomes by putting an agent into a state where they are likely to unfold.

In our discussion of facilitation, we will use the p and o notation from the previous section on accuracy. Recall that

- $o(s, a)$ refers to an actual outcome of taking action a in state s
- $p(o|s, a)$ refers to the true probability that outcome o will result when action a is taken in state s
- $\hat{p}_m(o|s, a)$ refers to the estimate of $p(o|s, a)$ made by some model m
- $\hat{o}_m(s, a)$ refers to an outcome predicted by model m ($\arg\max_o \hat{p}_m(o|s, a)$)

Three additional concepts that were introduced in the facilitation portion of section 1.4 will be used in this discussion.

- $\wp(s_s, s_g)$ is called a *prefix.* A prefix is a sequence of actions, produced by the planner, that are intended to move an agent from state s_s to state s_g.

- Prefixes are generally followed by *goal steps.* The goal step is an action a_g chosen as the action that maximizes the chance of a goal outcome o_g: $\arg\max_a p(o_g|s_g, a)$. A prefix followed by a goal step is a *plan.*

- $o(s_s, \wp(s_s, o_g), a_g)$ refers to the outcome resulting from taking a_g, the goal step, after having executed the prefix $\wp$ as described in chapter 4.

We can again use the various *disparity* statistics introduced in section 6.1 to compare the outcome of a plan's goal step to a target outcome. *Plan disparity* is referred to by $d(o_g, o(s_s, \wp(s_s, o_g), a_g))$. Disparity can be used as the basis for different measures of facilitation. We can define *absolute facilitation* as the complement of the disparity between a target outcome and the outcome of a plan.

$$1 - d(o_g, o(s_s, \wp(s_s, o_g), a_g)) \tag{6.5}$$

Or, we can define *relative facilitation* as the ratio of outcome disparity with a plan prefix to outcome disparity without the prefix. Specifically, relative facilitation is the measure we introduced in section 1.4:

$$1 - \frac{d(o_g, o(s_s, \wp(s_s, o_g), a_g))}{d(o_g, o(s_s, a_g))} \tag{6.6}$$

The relative facilitation score for a plan will be 1 when the disparity between the goal outcome and the plan outcome is zero, regardless of the value of the denominator; we will define facilitation to be 1 even when $d(o_g, o(s_s, a_g))$ is zero. Relative facilitation will be zero when the disparity after executing a prefix is the same as simply executing the goal step from the start state. Note, though, that this formulation of relative facilitation can also enter the negative range, even approaching $-\infty$. This happens in cases where executing a plan results in an increase in disparity over that of simply executing the goal step from the start state. Since this may cause problems in aggregate measures of facilitation, we will use absolute facilitation and disparity scores in the section to evaluate the planner. Recall these definitions of aggregate facilitation metrics from section 1.4:

Coverage is the percentage of the state space for which a planning actor can facilitate a particular income o_g.

Capability is the percentage of possible goal outcomes that can be facilitated from a fixed point or region of the state space.

We will use the statically-configured GridSim domain, as described in the discussion of accuracy, to evaluate facilitation. This domain will allow us to enumerate the full state space and get complete coverage statistics by generating plans from each starting point in the state space. In this evaluation we are most interested in coverage. Coverage gives us a concentrated look at a single activity, and how facilitation for that activity changes during development.

In our facilitation experiments, we track absolute facilitation for one or more goal outcomes as our agent learns from experiences in its environment. Experiences are collected as an agent acts under to some control policy. At regular intervals, the agent stops collecting training experiences, and enters into a testing phase. In the testing phase, the agent engages in a series of generate-and-test trials to determine the coverage for some outcome o_g after having collected n experiences to build models and plans with. The coverage test is performed by enumerating the states in the simulator and iterating through them, with the agent attempting to plan from each of the possible start states to achieve o_g. At each state, a plan is generated, and executed. The result code from the execution module is recorded, along with the disparity between the actual outcome and the goal outcome.

We can visualize coverage as it changes throughout development in two basic ways. First, we can compute average facilitation scores over the entire starting state space each time a testing phase is performed, and view it as a time series. These learning curves allow us to observe the gross dynamics of coverage over time to tell whether or not coverage is improving, by roughly how much, and over what time intervals significant change occurs. This gives us a sense of whether or not the basic facilitation claim is being satisfied (the agent is improving). We cannot, however, see fine details of what is going on in each of the individual starting states using only learning curves. When it is necessary to look at successes and failures encountered in individual starting states, we will use *coverage plots*.

The coverage plot is a visualization technique suited for simple 2D domains such as the GridSim simulator. A coverage plot is essentially a map of the domain with success and failure information superimposed onto it. A sample coverage plot is shown in figure 6.10, using the GridSim configuration introduced in section 1.1.1. Note that the grid contents, including the walls, debris, and dropoff locations, are the same as those in figure 1.3, except that the agent has been removed and some of the cells are shaded gray. The shading of the internal cells represents coverage information for the following outcome, after 300 training experiences:

```
N-N-D-D-U-N-N-U-N-N-N-N-N-D-N-N
```

This outcome corresponds to a `move-w` experience in which the distance to the nearest visible object in the west decreases, the distance to the east increases, a nearby object to the north leaves the visual field, and nothing changes to the south.

We will refer to cells on a coverage plot by their distance from the top, left cell on the map, which is $(0, 0)$. Note that $(0, 0)$ contains a wall, cell $(1, 2)$ contains a dropoff point, cell $(4, 3)$ contains debris, and so on. Cells along the border of the grid are walls, and are colored black in the coverage plot to indicate that a facilitation test

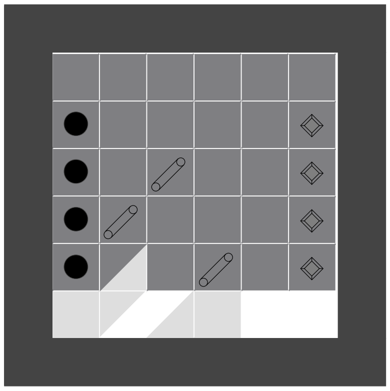

Figure 6.10. A sample coverage plot for the 8 × 8 GridSim domain.

is not possible from that cell. Cells inside the rectangle formed by the points $(1, 1)$ and $(6, 6)$ are traversable by the agent, and facilitation tests can be made starting at any of these cells. The level of shading for any cell in the coverage plot indicates what the agent is able to do with the goal outcome when starting in that particular cell. Those locations that are shaded medium-gray are cells from which the agent either cannot build a plan, or builds a plan that breaks. The lighter gray shaded cells are cells from which the agent can build a plan, execute it as intended, but fails to achieve the goal outcome. Unshaded cells correspond to starting states on the map from which the agent is able to build and execute a plan that successfully achieves the goal. Additionally, each cell is broken up into two triangular areas to differentiate starting states in which the cargo bay is empty from those in which the bay is full. The upper-left triangle of each cell corresponds to the start state where the bay is full, while the lower-right triangle corresponds to the start state in which the bay is empty. Note that cell $(2, 5)$ is shaded medium-gray on top and light-gray on the bottom, indicating that when the agent starts there with its bay full, it cannot build or execute a plan, but if the bay is empty, it builds a plan it thinks will work but does not.

This particular coverage plot illustrates the general utility of such a visualization. Notice that there are two cells from which the agent can successfully achieve the goal outcome regardless of the cargo bay's status. They are $(5,6)$ and $(6,6)$. Note also that the agent can generally only build plans when it starts on the bottom row of the grid. It happens that this particular outcome is only possible when the agent moves west out of cell $(6,6)$. The coverage plot shows that the agent has a hard time achieving the goal when it is far from the grid cell in which the outcome is easiest to achieve. It is an intuitive result made obvious by the coverage plot.

We now turn to individual coverage experiments. Our goal here is to verify the facilitation claims with learning curves and coverage plots, and in the process determine the factors that contribute to or limit the performance of our system. We will present a series of experiments and detail what they suggest about the performance of our system, highlighting what we consider to be the primary factors that influence facilitation.

6.2.1 Random Walk

To serve as a baseline, we ran coverage tests with random walk data. In this baseline experiment the agent operates under a random walk policy for 300 training steps, learning is suspended while a complete coverage test is run for a random sample of four unique outcomes, and the process is repeated. The same four outcomes are tested after each batch of 300 training steps. The four outcomes in the experiment we describe are:

```
N-N-D-D-U-N-N-N-N-N-N-N-D-U-N-N
N-N-U-U-D-N-N-U-N-N-N-N-N-D-N-N
N-N-U-U-D-N-N-D-N-N-N-N-N-U-N-N
N-N-N-N-D-N-N-N-N-N-U-N-D-U-N-N
```

From top to bottom, these labels represent: a move north along the east wall in which the agent moves from one dropoff cell to another one, a move west in which a nearby object comes into view to the north, another move north in which an object disappears from view in the west facing camera, and a move east in which the agent moves into the southeast corner of the grid.

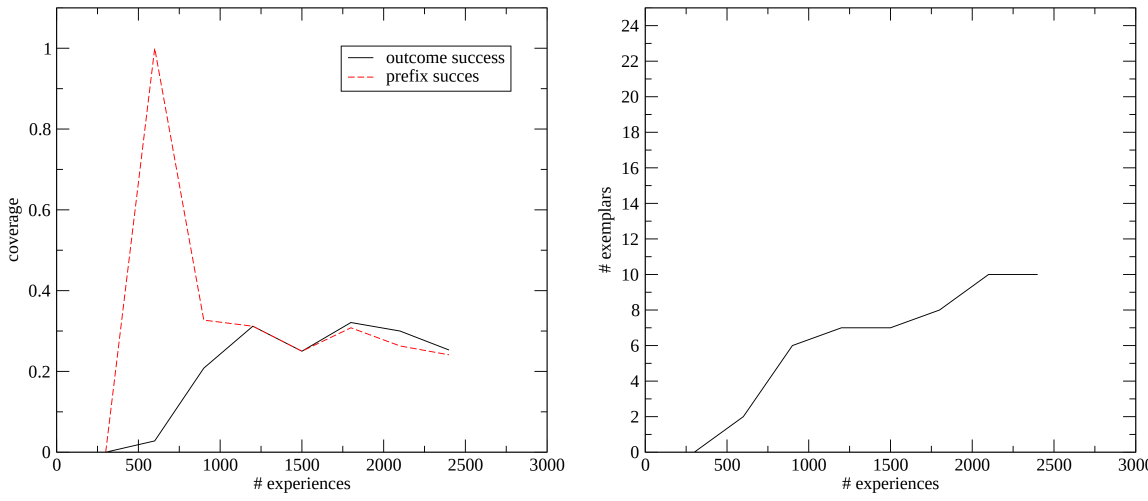

Figure 6.11. Coverage statistics for the outcome `N-N-D-D-U-N-N-N-N-N-N-N-D-U-N-N` under the control of a random walk in the static domain.

Coverage curves for each of the four target outcomes are presented in figures 6.11 through 6.14. Each figure contains two graphs. The graph to the left of each figure shows the results of coverage tests taken every 300 experiences. In each of these graphs, two coverage measures are shown: *prefix success* and *outcome success*. Recall that in a coverage test, an agent is asked to achieve a target outcome from each of the enumerable starting states in the environment. Starting in cell (x, y), an agent will first build a plan, and then execute it. If a plan can be successfully generated and executed to completion from cell (x, y), that is scored as a prefix success. If, in addition, the target outcome is achieved by the plan, then it is scored an outcome success. Each point in the left-hand graphs of figures 6.11 through 6.14 represent a percentage of the state space from which a prefix or outcome successes have been generated. A prefix success coverage value of 0.5 indicates that a plan was successfully

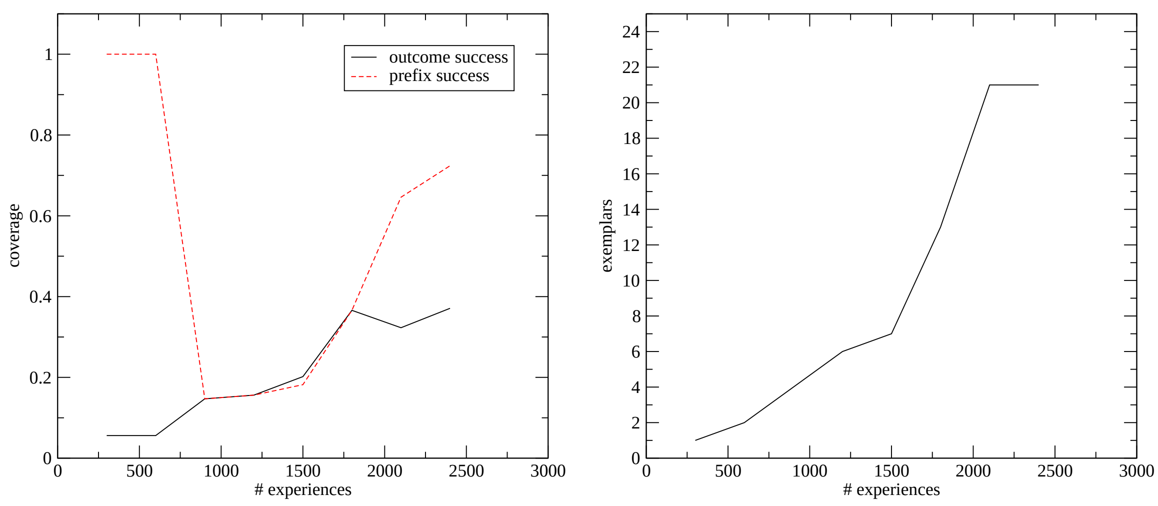

Figure 6.12. Coverage statistics for the outcome `N-N-U-U-D-N-N-U-N-N-N-N-N-D-N-N` under the control of a random walk in the static domain.

generated and executed to completion from 50% of the possible starting states at some point during development, for example.

The ideal is always to achieve both a prefix success and an outcome success. This indicates that a plan was generated, it could be executed, and it facilitated the goal outcome. Note, however, that it is possible to generate a prefix success that does not produce an outcome success. This is often the case for plans that are too general; a legal plan is generated and executed but does not achieve the goal. It is also possible to fail in the prefix phase and then produce an outcome success. This is because the goal step of each plan is executed whether or not the plan is perceived by the system to have succeeded. A prefix failure followed by an outcome success is rare in the GridSim domain, and generally indicates a broken plan that manages to coincidentally achieve the goal.

The plot to the right of figures 6.11 through 6.14 shows the number of exemplars for the given goal outcome versus the total number of experiences the agent has collected. Recall that an exemplar is an experience that matches the target outcome.

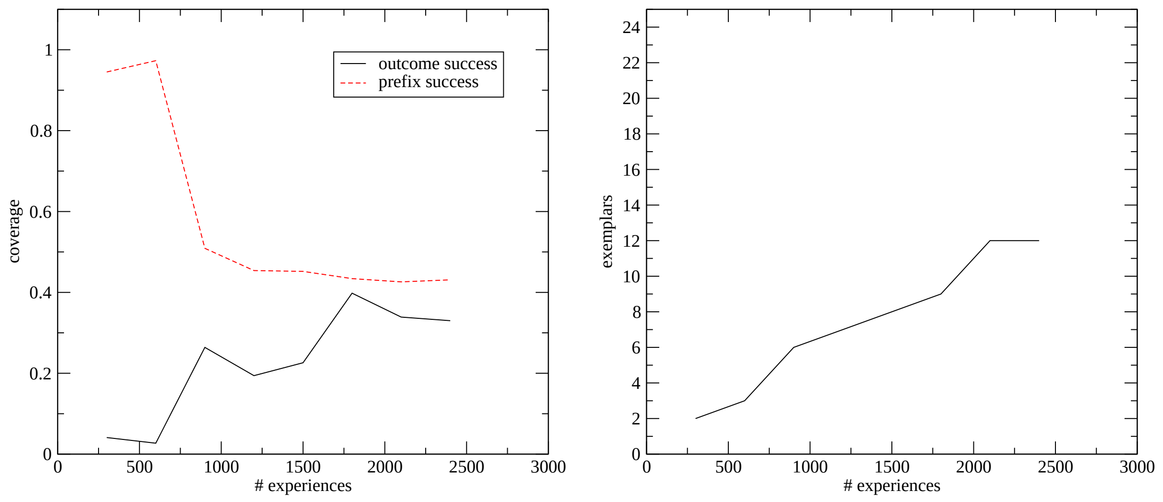

Figure 6.13. Coverage statistics for the outcome `N-N-N-N-D-N-N-N-N-N-U-N-D-U-N-N` under the control of a random walk in the static domain.

Exemplar counts are included because our planning system is case-based; it stands to reason that the number of viable exemplars to work with has an effect on coverage.

This baseline study illuminates several worthwhile results. Note that in each graph, prefix success coverage starts much higher than outcome success coverage. In many cases, prefix success spikes to near 1.0, indicating that the planner always thinks that it is generating and executing good plans. In contrast, outcome success starts low. These two curves then slowly converge upon each other. This phenomenon is almost universal in all coverage tests, and is a result of early instability and inaccuracy in the modeling system. In early development, the modeling system has weak and overly-general models. These models produce initial conditions that are easy (and sometimes trivial) to satisfy but rarely reflect the true conditions necessary for success. As models get better, the planner simultaneously improves at correctly identifying plans that will and will not work. When the planner, with the help of the modeling system, can identify with perfect accuracy whether a plan will or will not work, prefix success and outcome success converge. In three of the four graphs, prefix and

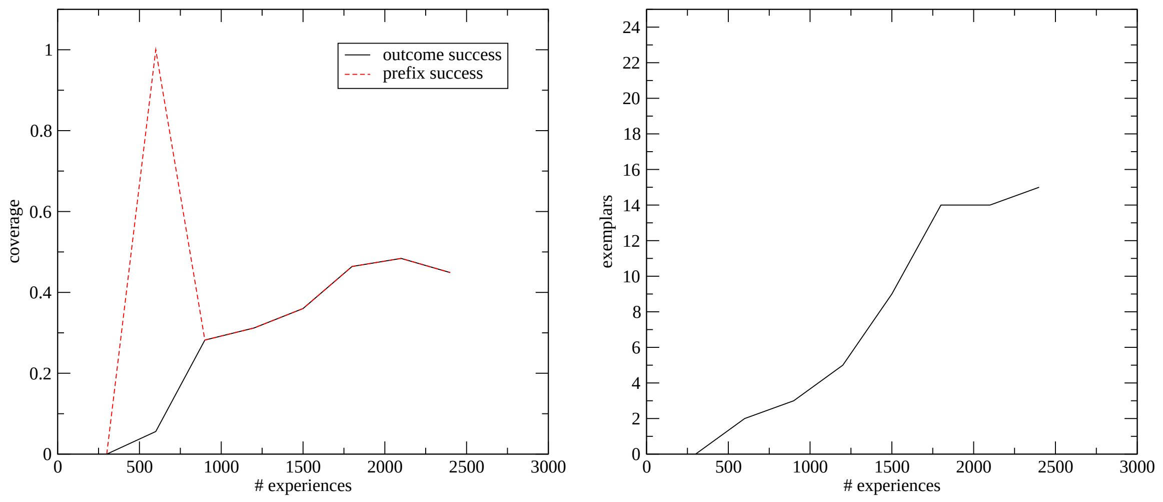

Figure 6.14. Coverage statistics for the outcome `N-N-N-N-N-N-N-U-N-U-D-N-N-D-N-N` under the control of a random walk in the static domain.

outcome success actually do converge. In the fourth, convergence seems likely with additional training experiences Comparison of these plots to the accuracy plots of section 6.1 shows that convergence occurs approximately when model accuracy starts to stabilize. This does not preclude later divergence, however. In figure 6.12, prefix and outcome success do diverge, as a result of a sudden surge in exemplars shown on the corresponding graph of exemplar counts. This is another common feature of coverage; sudden increases in exemplar counts induce instability in the models that are felt throughout the system in the form of temporary periods in which faulty plans may be generated.

Coverage plots in figure 6.15 reflect this "destabilizing" effect schematically. In the leftmost coverage plot, the agent has 300 experiences. The planner produces plans that cover the entire states space, though in only two cells can it actually produce a working plan. In the center plot, after 1800 experiences, the agent's prefix and success curves have converged. Though the range of cells from which the agent can achieve the goal is limited, it never fails to achieve its goal when it produces a

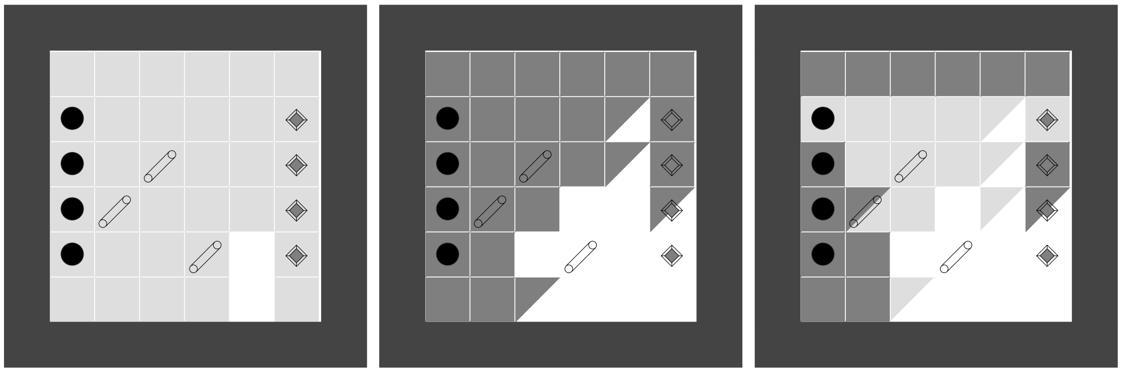

Figure 6.15. Coverage plots for the outcome `N-N-U-U-D-N-N-U-N-N-N-N-N-D-N-N` as underlying models change after 300, 1800, and 2400 experiences.

plan. Finally, in the rightmost plot, the introduction of a slew of new exemplars at 2400 experiences has destabilized the models. As a result, the agent begins starts producing nonworking plans for cells from which it was previously unable to produce a plan for at all.

The coverage plots in figure 6.15 also illustrate an emerging trend in facilitation for the GridSim domain: that some outcomes can only be experienced by taking a single action in one or just a few states. When an outcome is achievable only by taking a certain action when in a single state or small number of states, we call these special states the outcome's *home states*. Coverage initially includes only the home states, indicating that the modeling system can identify the correct conditions for achieving the goal with only a single action, but the system cannot yet build good plans. In this initial batch of coverage tests, coverage seems to "grow" outward from the home states until coverage tends to reach a maximum radius. In figure 6.15, the radius seems to be somewhere around 3 cells. The planner, in this case, seems to have an effective range of 3-4 steps from the home state. We call this range in which a planner is capable of producing a working plan the planner's *effective radius*. Depending on where on the grid the home state is, this effective radius may limit the maximum coverage scores we will see for a particular goal.

The data from this baseline study, coupled with observations made from the coverage plots, allow us to form some hypotheses about the primary factors involved in determining facilitation, coverage, and the rates at which they change. Here are the primary hypotheses that these data suggest:

- There is an initial "ramp up" period in which the modeling component is primarily responsible for changes in facilitation and coverage. During this period models are unstable but improving. Poor initial models cause a wide disparity between prefix success and outcome success, indicating that faulty plans are being generated. As the modeling process is bootstrapped, prefix and outcome success levels converge. Convergence occurs when the models involved in a particular outcome stabilize.

- Exemplar count seems to influence coverage in two ways. First, accurate initial condition models can only be built when an adequate number of positive instances of an outcome are available. Second, we are using a case-based planner that must find inroads to old traces leading to the goal. When more exemplars are being generated, in general, there will be more traces through the state space for our planner to catch an inroad to.

- The number of home states, or states from which a goal is trivially achievable, plays a primary role in how coverage for that goal will develop. This is especially true when home states are remote or improbable in a domain.

- The planner horizon – the maximum number of steps in a trace our case-based planner will consider in the search for a plan – appears to limit coverage. Consider that the effective planning radius for a typical goal is roughly 3, using traces generated by a random walk and a planning horizon of 7. Since a random walk will rarely produce traces that go directly to a goal, we can expect superfluous steps in any trace that reduce the effective radius of the planner.

- The control policy used to generate training experiences should have some effect on facilitation. A structured control policy might improve coverage in a number of ways. For example, the L0 control policy has the effect of accelerating the stabilization of the modeling system, and as such, this control policy could have the effect of increasing the rate at which prefix and outcome success levels converge.

We will use these hypotheses to prescribe the next round of experiments. We will start by selecting a fixed set of five outcomes to test each time we change the experimental setup to test these hypotheses. We will refer to these outcomes as o_1, o_2, o_3, o_4, and o_5. To control for exemplar size and outcome frequency, we chose these outcomes by sorting the set of all outcomes by their frequency in a $2,500$ step random walk, and selecting outcomes from this at fixed intervals: the 17th, 33rd, 50th, 67th, and 83rd percentiles. The outcomes, followed by their frequency in the $2,500$ step random walk, are listed below:

o_1: `N-N-N-N-N-N-N-D-D-D-U-N-N-U-N-N (31)`

o_2: `N-N-D-D-U-N-N-U-N-N-N-N-N-D-N-N (10)`

o_3: `D-D-N-N-N-N-N-N-N-N-N-N-N-N-N-N (9)`

o_4: `N-N-N-N-N-D-N-D-N-N-N-N-N-N-N-N (6)`

o_5: `N-N-N-N-U-N-N-N-D-D-U-N-D-U-U-U (4)`

It is worth noting that the frequency range of these outcomes is relatively small, between 31 and 4. This is primarily due to the fact that the large majority of outcomes in a random walk fit the following profile:

```
N-N-N-N-N-N-N-N-N-N-N-N-N-N-N-N
```

This is the outcome in which no sensor values change, and it corresponds to a wide variety of home states and actions: moving into a wall produces it, as does executing a `drop` action with nothing in the cargo bay. It occurs $1,024$ times in the $2,500$ step random walk. There are 99 additional outcome classes that occur between 4 and 36 times in the random walk. The above outcomes, from top to bottom, represent the following outcomes: an outcome where the agent moves east and something disappears from view to the south (o_1), an outcome where the agent moves west out of the southeast corner of the grid (o_2), an outcome where the agent drops an object that is not picked up by the current cell sensor (o_3) [4], an outcome where the agent moves north in the field of recharge cells along the west wall (o_4), and an outcome where the agent moves south over a piece of debris that appears in the current cell sensor (o_5).

Using the same set of outcomes over each our experiments will allow us to better compare the results for facilitation experiments run under different conditions. By selecting outcomes $o_1 \ldots o_5$ as we have (by their frequencies in a random walk) we are attempting to observe the apparent effects of exemplar frequency on facilitation throughout the experimental process.

We will now present the results of four additional experiments. First, we will briefly reexamine the random walk for the new set of outcomes introduced above to reproduce a baseline example. Next, we will look at a planning based control policy in which the agent pursues only the outcomes $o_1 \ldots o_5$ during the training periods to determine if a policy biased towards reproducing the outcomes being evaluated will improve performance beyond what we saw as the effective range of our planner in the random walk. In a third set of experiences, we extend the planning horizon to 9 steps, and also introduce a profile relaxation modification to the planning process. The relaxation procedure attempts to seek out greater levels of generalization during the planning process so that a single exemplar can facilitate a goal from a broader

[4] An object that is dropped may not appear in the ccs if it is dropped into a cell where there is a dropoff receptacle or there is an existing object that obscures the newly introduced object.

portion of the state space than previously possible. Finally, we introduce a *cheat walk* control policy to determine if the greatest factor contributing to coverage is a wide variety of good exemplars.

6.2.2 Random Walk Revisited

Baseline coverage and exemplar curves for the five new outcomes $o_1 \ldots o_5$, taken during a random walk, are shown in figures 6.16 through 6.20. These results fit the general pattern established in the first random walk experiment. That is, prefix success starts at an inflated value as the planner produces plans that are easy to execute, but outcome success is low, as those plans rarely succeed. Over the first 1000 experiences, model improvement allows prefix and outcome success to more or less converge. Beyond 1000 experiences, these success rates may increase (at a slower rate) or level off, generally reaching values in the range of $20 - 40\%$. Note that the two most infrequent outcomes, o_4 and o_5, are harder to come by in a random walk; for this reason, the exemplar curve in figures 6.19 and 6.20 level off, capping outcome success coverage for these two curves at about 20%. These two outcomes have only two home states each, a single cell on the grid with the bay either empty or full, making them fairly rare in any given random walk. Rarity, in the cases of o_4 and o_5, appears to have an adverse effect on coverage.

The worst coverage, however, was experienced with outcome o_2. Coverage for this outcome never exceeds 10% and the combination of the modeling system and planner is never able to drive prefix success into convergence with outcome success. Interestingly, this outcome does not seem to be as rare in a random walk as o_4 and o_5 are. Like o_4 and o_5, o_2 has only two home states, but infrequency does not appear to be the primary factor behind its lack of coverage. This outcome corresponds to moving west out of the southeast corner. This outcome, while showing up a few more times over the first 2500 training experiences, has one of the most remote home states on the grid, and this seems to produce poor results in coverage tests.

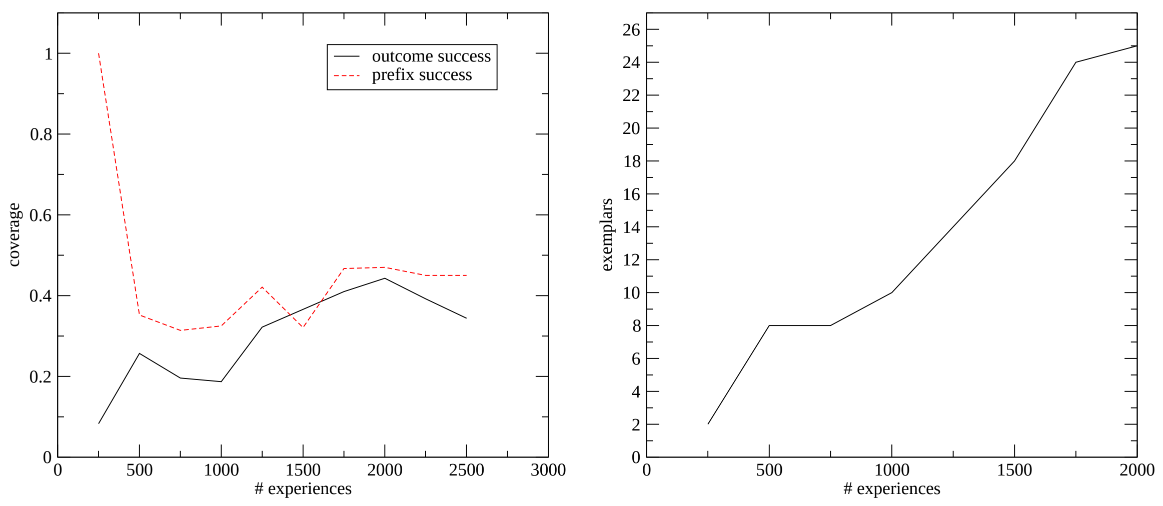

Figure 6.16. Coverage statistics for o_1 `(N-N-N-N-N-N-N-D-D-D-U-N-N-U-N-N)` on random walk data.

Figures 6.21 through 6.22 show coverage plots for $o_1 \ldots o_5$ after 2500 training experiences generated via random walk[5]. Recall that unshaded cells are facilitated, lightly shaded grid cells correspond to prefix successes but not outcome successes, and heavy shading represents the case where no plan could be generated. Coverage tends to "radiate" out from the home states of outcomes in these figures. Note that in o_2, there is only a single covered cell, its home state in the southeast corner of the grid. In the coverage plot for o_3, the upper half of many cells are facilitated but the lower half is not, indicating that this outcome is facilitated more readily when the agent starts with cargo in its bay. Recall that o_3 corresponds to a drop experience in which the cargo bay is emptied. Most such `drop` outcomes are more readily facilitated starting with the bay full since the `lift` experience that loads the cargo bay may precede the

[5]Each of the five outcomes are tested using the same 2500 random walk experiences.

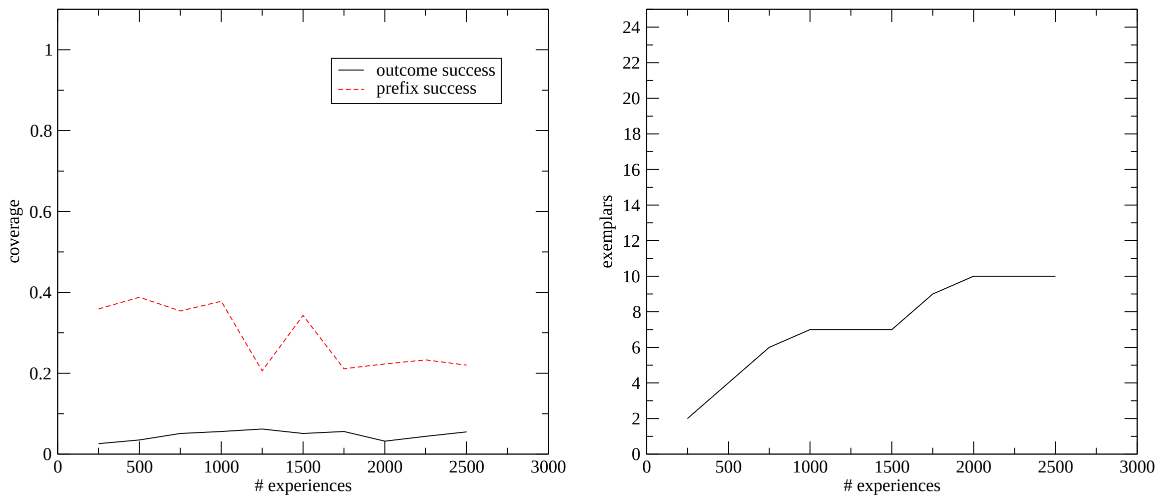

Figure 6.17. Coverage statistics for o_2 (`N-N-D-D-U-N-N-U-N-N-N-N-N-D-N-N`) on random walk data.

actual exemplar used in planning by many steps. This critical `lift` operation will often be outside the planning horizon, causing the planner to fail to reconstruct a plan when starting with an empty bay. For many of the `drop` outcomes that rely on the bay being full, lifting debris is a bottleneck outcome, as described in section 5.4. Thus, not only is it critical to identify bottlenecks for the purposes of modeling, but also to identify them for their possibly pivotal (and distal) roles in plans.

6.2.3 A Greedy Training Policy

Random walk data serves as a baseline for how we can expect learning curves, and coverage plots, to develop over time with the modeling system and planner working on experiences generated with an unbiased control policy. The learning curves suggest that at least three of the candidate factors contributing to the development of plans that we identified are actually in play under a basic random walk: bootstrapping of the models, exemplar count, and the number and location of home state(s).

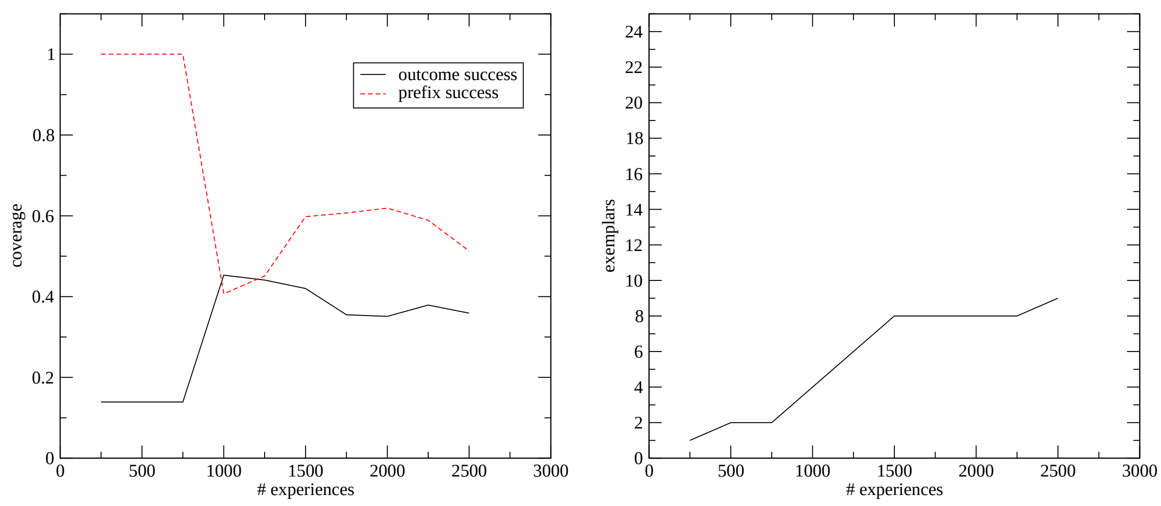

Figure 6.18. Coverage statistics for o_3 (D-D-N-N-N-N-N-N-N-N-N-N-N-N-N-N) on random walk data.

We are interested both in confirming the influence of the other primary factors on facilitation (control policy, planning parameters, and more detailed aspects of exemplar count) and identifying ways of improving performance. While facilitation rates of 40% are certainly not disappointing, there is reason to believe that much higher rates are possible. Evidence in the accuracy evaluation suggested that a structured control policy can improve performance. In this section, we will look at one such control policy and its effect on the development of plans in our system.

We have designed a control policy, called the LPQ controller, that works in two phases. The controller starts by operating the L0 controller, as described in section 5.1 over the first $1,000$ experiences of training. The L0 phase is introduced to bootstrap stable, accurate models more quickly than the random walk controller. Once stability is achieved, control is gradually handed over to a second system called the LP controller. The LP controller will choose from $o_1 \ldots o_5$ at random, build a plan to achieve it, and execute it. Essentially, this controller greedily pursues training examples pertinent to the outcomes we are interested in. The idea here is that we can

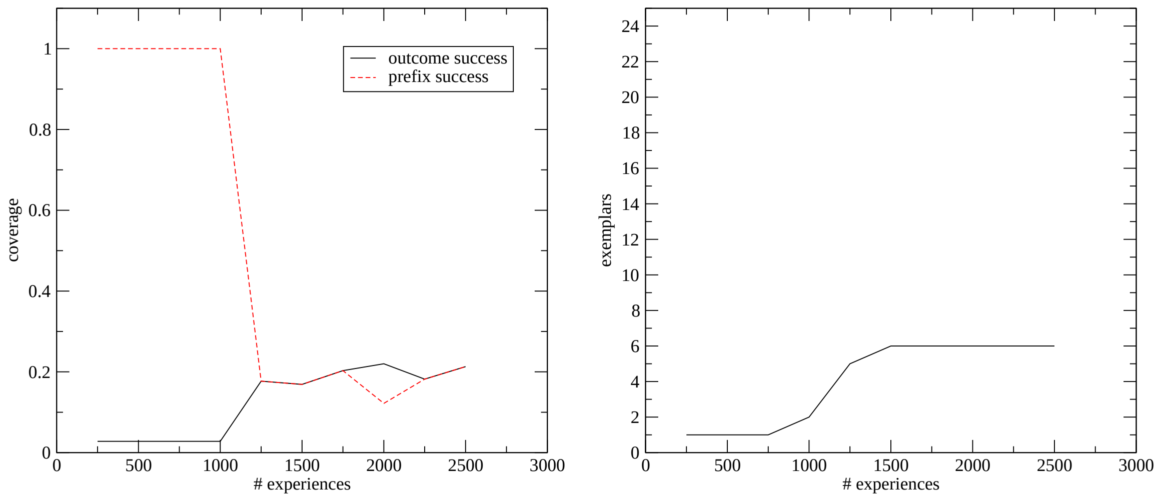

Figure 6.19. Coverage statistics for o_4 `(N-N-N-N-N-D-N-D-N-N-N-N-N-N-N-N)` on random walk data.

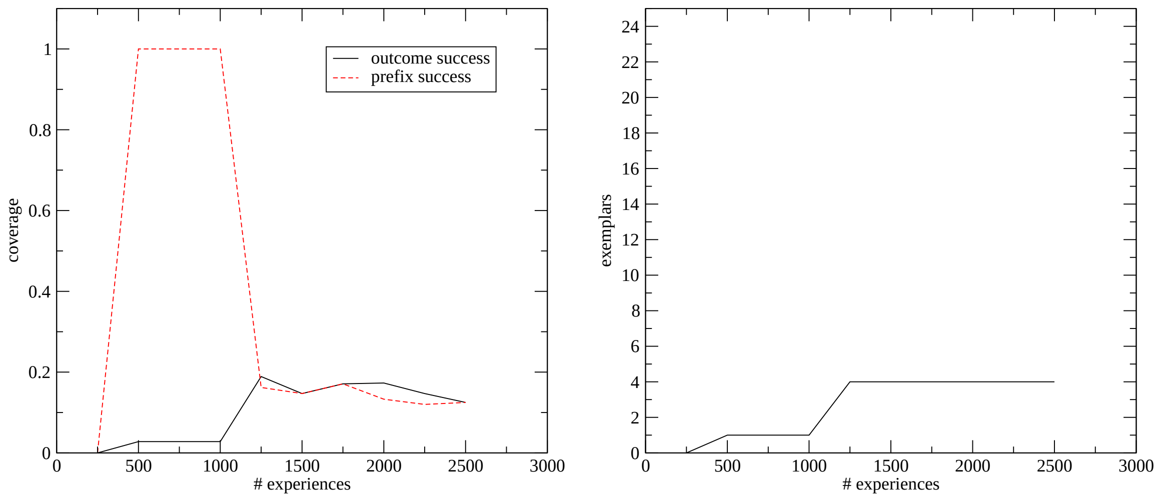

Figure 6.20. Coverage statistics for o_5 `(N-N-N-N-U-N-N-N-D-D-U-N-D-U-U-U)` on random walk data.

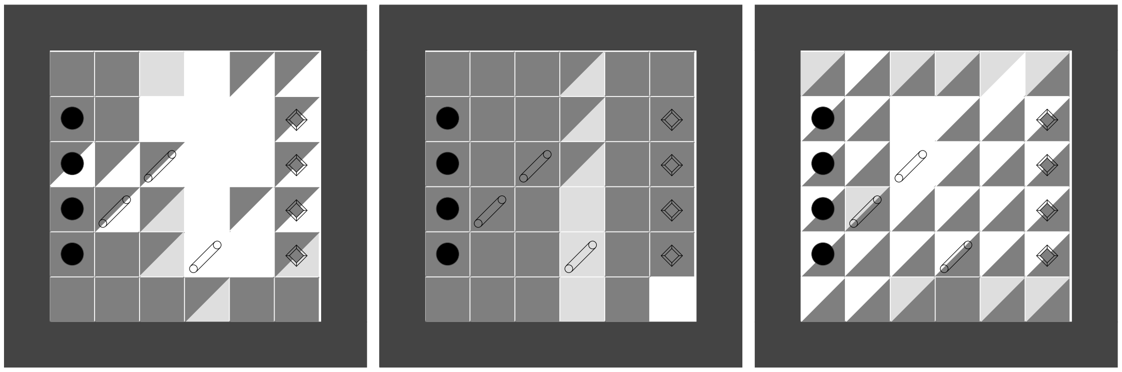

Figure 6.21. Coverage plots for o_1, o_2, and o_3 after 2500 experiences of random walk training.

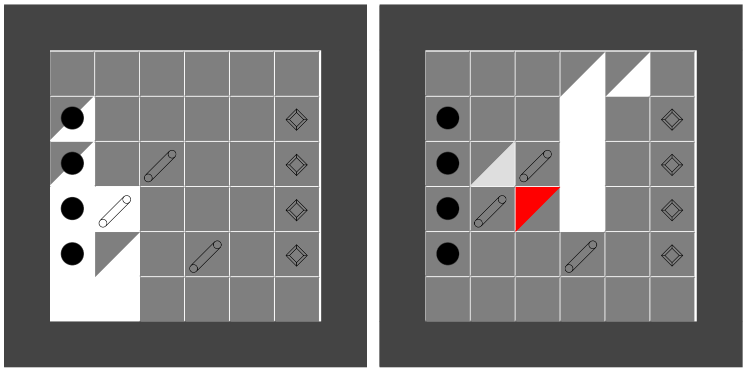

Figure 6.22. Coverage plots for o_4 and o_5 after 2500 experiences of random walk training.

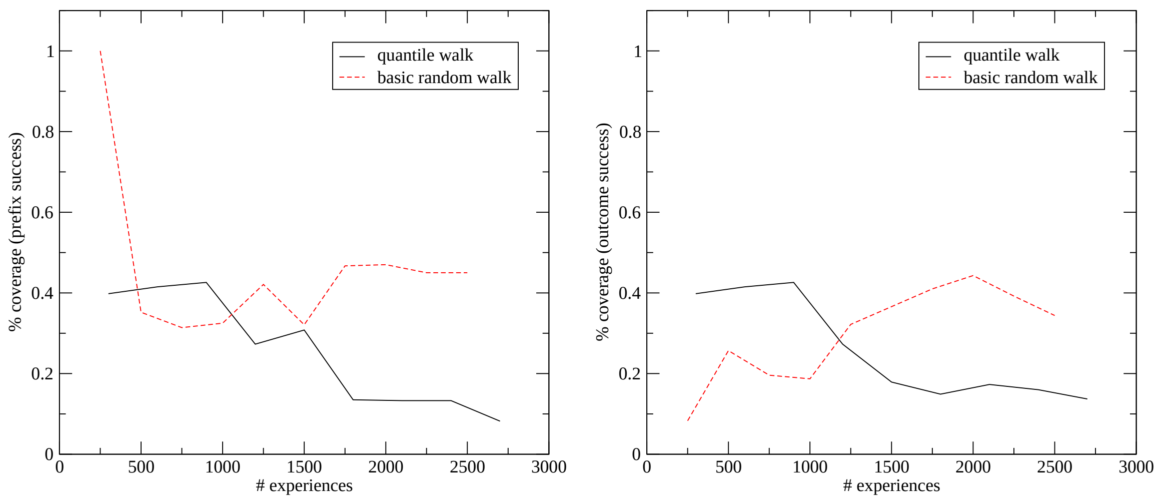

Figure 6.23. Coverage statistics for o_1 `(N-N-N-N-N-N-N-D-D-D-U-N-N-U-N-N)` using training data collected with the LPQ controller.

focus training experiences toward behavior we are looking to improve, and away from peripheral outcomes and behavior that may be superfluous to the tasks the system is being evaluated on.

Coverage curves are shown in figures 6.23 through 6.27. Each of these figures contains two graphs. On the left is an overlay of prefix success for the LPQ and random walk (baseline) controllers, and on the right, outcome success for the LPQ and random walk controllers.

The results shown in these figures can be summarized as follows.

- The L0 controller produces faster convergence between prefix success and outcome success in all five outcomes tested. The L0 controller also produces an accelerated learning curve for outcome success over the first $1,000$ experiences of training.

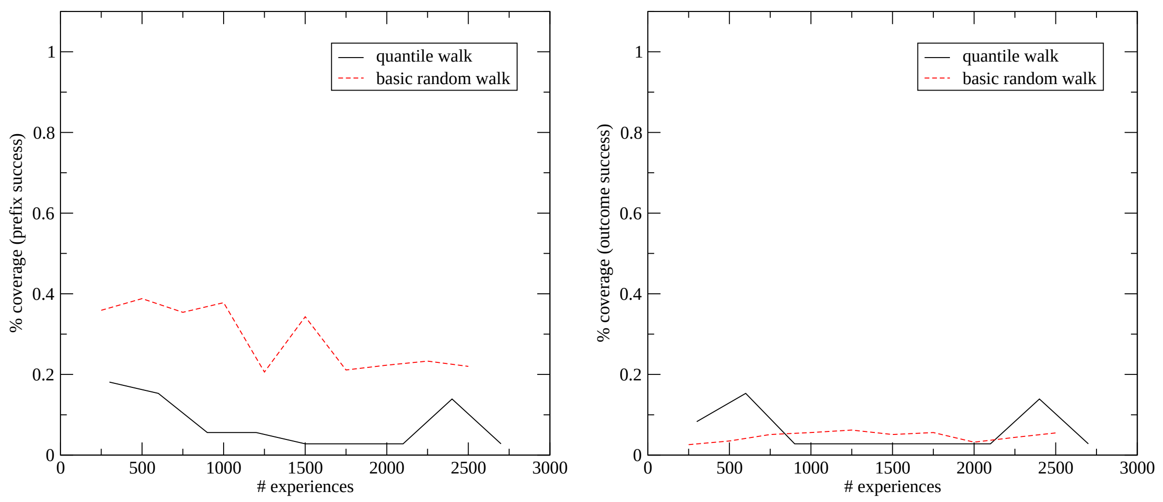

Figure 6.24. Coverage statistics for o_2 (N-N-D-D-U-N-N-U-N-N-N-N-N-D-N-N) using training data collected with the LPQ controller.

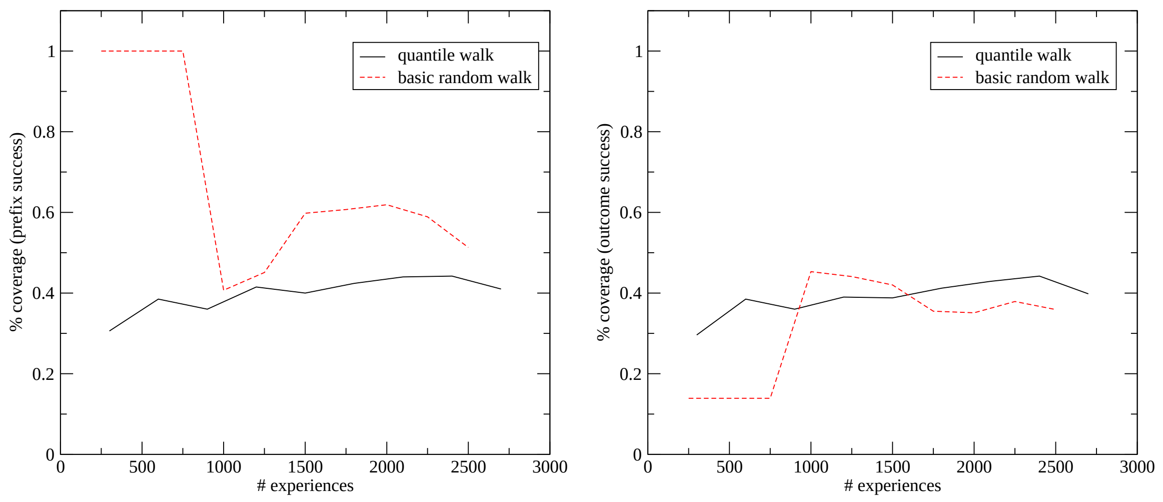

Figure 6.25. Coverage statistics for o_3 (D-D-N-N-N-N-N-N-N-N-N-N-N-N-N-N) using training data collected with the LPQ controller.

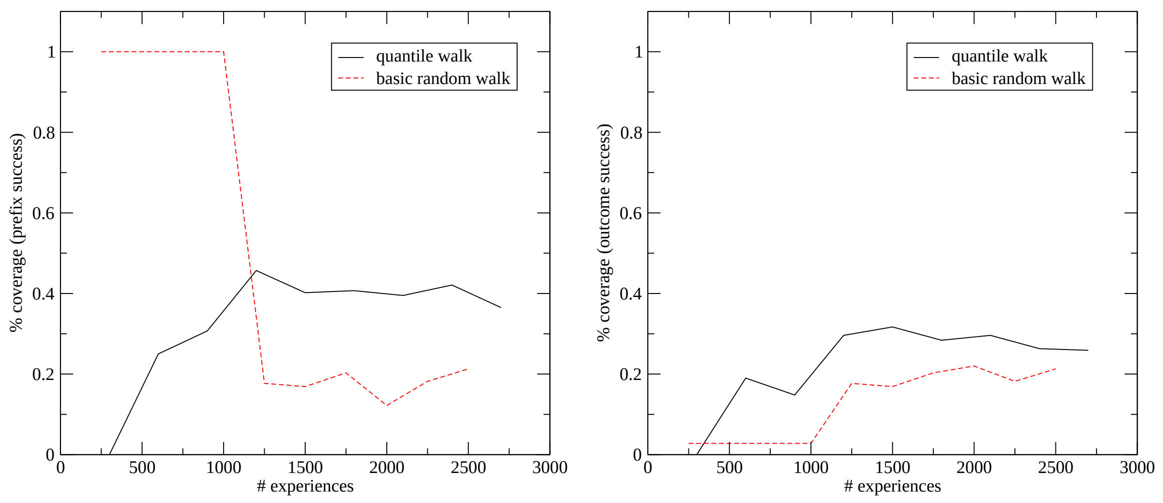

Figure 6.26. Coverage statistics for o_4 `(N-N-N-N-N-D-N-D-N-N-N-N-N-N-N-N)` using training data collected with the LPQ controller.

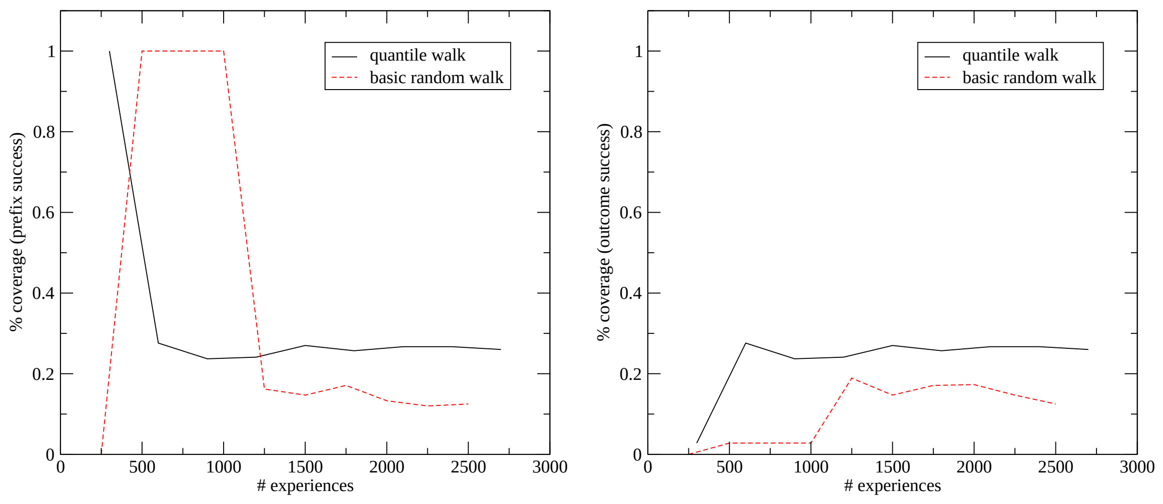

Figure 6.27. Coverage statistics for o_5 `(N-N-N-N-U-N-N-N-D-D-U-N-D-U-U-U)` using training data collected with the LPQ controller.

- The LP controller appears to provide a slight advantage in overall outcome coverage in the two lowest-frequency outcomes (o_4 and o_5). These two outcomes' performance improved by a factor of roughly 50% over random walk control.

- Training with the LP controller does not provide a distinct advantage in the coverage of o_2 or o_3.

- Training with the LP controller appears to *hurt* coverage for o_1.

These results for the LP controller are not altogether intuitive, but can be explained. For outcomes that are frequent, a random walk can easily produce exemplars, arriving at them from a variety of trajectories through the state space, with good regularity [6]. In this case, since exemplars are plentiful in the random walk, the idea that they be arrived at from a variety of different trajectories affords the planner many varied experience traces to work with. The random distribution of traces allows the planner to facilitate more starting states. By contrast, when using the LP controller, goal states are revisited under control of the planner, and so they are generally revisited by repeating one or a small number of fixed trajectories through the grid. The effect of the LP controller is to stifle continued exploration that might occur under random walk or L0 control.

For low frequency outcomes, a random walk will provide few exemplars. Depending on the home state, these exemplars may only be approachable along one or two trajectories. In these cases, an LP controller can both increase the number of exemplars and simultaneously move the agent into the vicinity of the low-frequency outcome's home state, improving its chances of being revisited in the course of unrelated behavior. For these low-frequency outcomes, the LP controller has the effect of *stimulating* exploration.

[6]By *trajectory*, we mean a sequence or path through the grid. The south east corner of the grid may be approached on a trajectory from the north or west, for example.

Outcomes exist on a spectrum between high frequency, accessible home states, and low-frequency, remote home states. In one corner, high-frequency, accessible outcomes like o_1, are stunted by the LPQ controller. In the opposite corner, infrequent, remote outcomes like o_2 are relatively unaffected because neither the random walk nor the LPQ walk can generate very useful exemplars. However, for the range of medium to low-frequency outcomes, the LPQ controller can provide additional exemplars useful in planning, providing a range of improvement shown in figures 6.25 through 6.27.

6.2.4 Opening Up the Planner

The planner itself plays a great role in determining in the performance of our system. While we are committed to the case-based nature of the planner (for reasons described in section 3.2), the planner does have some heuristics and parameters that might be changed to produce improvements in coverage.

The most noteworthy planning parameter we have identified is the planning horizon: the length of experience traces that the case-based planner will consider when searching for plans. Extending the planning horizon should allow the planner to produce plans in some instances where with a shorter horizon it would be impossible. Two immediate examples are cases in which many superfluous steps clutter a workable experience trace and when the starting state is distant from the goal's home state, and a longer plan is necessary as a result.

We will also consider a variation on the PFT planner. Recall that the PFT algorithm attempts to build a triangle table that explains how a series of experiences moved the agent from one state to the goal outcome. The basis of building these triangle tables is the production of initial conditions for steps in the experience trace and making connections to prior experiences that satisfy these initial conditions. Initial conditions are generated with the decision tree induction scheme described in section 2.2, using the SSC profiles for the actual outcomes in the experiential trace.

In the course of running many coverage experiments, we noticed that both outcome and prefix success were capped at around 40%. We wondered whether the planning process, and in particular the initial condition induction phase, was proving restrictive and not producing plans where generalizations were possible. If the initial condition induction step could be made more general, then plans might become more general, creating better coverage. We will now introduce a variant of PFT we call the *relaxation planner*. This variant attempts to produce more general plans through a relaxed initial condition induction phase. Instead of using the full SSC profiles of experiences in a trace, we generalize these profiles as suggested in section 2.2.3. Profiles are generalized by wildcarding any sensor that does not appear later in the triangle table.

The relaxed planning process is best understood by example. Suppose we have a goal outcome o_g, and we are building a triangle table for a trace that ends in an experience, e_g. The first step is to produce initial conditions for experiences of type o_g. We will call this set of conditions I_{o_g}. The next step is to compare the current state against I_{o_g} to determine if the plan can start as is. Let us assume that I_{o_g} is not satisfied in the current state, and that more planning is required. The next step is to pull the experience prior to e_g (which we will call e_{g-1}) off the experience trace and to insert it into the triangle table. The PFT planner would start by producing the SSC profile for e_{g-1} (denoted by o_{g-1}) and generating initial conditions for that profile. The relaxed PFT planner, however, would start by generalizing o_{g-1}. Suppose o_{g-1} has the following profile:

```
U-U-N-N-N-N-N-N-N-N-N-N-N-N-D-D
```

This new variant of the PFT planner will *relax* this profile before generating its initial conditions. Each sensor that does not appear later in the triangle table will be replaced by a wildcard. Since only I_{o_g} appears later in the triangle table, only the sensors mentioned in I_{o_g} will remain specified in the relaxed profile. Assume that I_{o_g} comprises the following conditions:

$$(\text{bay-color} > 1) \wedge (\text{bay-shape} > 1)$$

Then the relaxed version of o_{g-1} will be as follows:

```
U-U-?-?-?-?-?-?-?-?-?-?-?-?-?-?
```

This indicates that what is important about e_{g-1} is that the bay sensor changes. What happens in the visual sensors is considered superfluous. When the modeling system builds initial conditions for o_{g-1}, it labels all experiences in which the bay sensors increase as positive instances of the desired behavior, not just those experiences which are exact matches. The hypothesis is that steps in the plan will become more general and that it will be easier to find an inroad to relaxed plans.

Again, the intention in this section is to make nontrivial changes to the planner by increasing the planner horizon and relaxing the initial condition generation phase of the PFT process to produce farther-reaching and more generalized plans. The planner obviously plays a pivotal role in our coverage evaluation, and this condition is meant to determine how robust or sensitive the planner is to these types of changes.

Coverage curves for o_1 and o_4 are shown in figures 6.28 and 6.29. These graphs plot the coverage for prefix and outcome success against the basic random walk outlined in section 6.2.2. Graphs for the remaining three outcomes o_2, o_3, and o_5 exhibit similar behavior and are ommitted for brevity. There is in fact no discernable difference in coverage performance as a result of moving from a planning horizon of 7 to a horizon of 9 and mixing profile relaxation into the PFT planner.

These results do not suggest that planning horizon has no effect on performance; rather, they suggest that a small increase in horizon will not yield a small increase in coverage. It is still our belief that improved coverage could be achieved by increasing

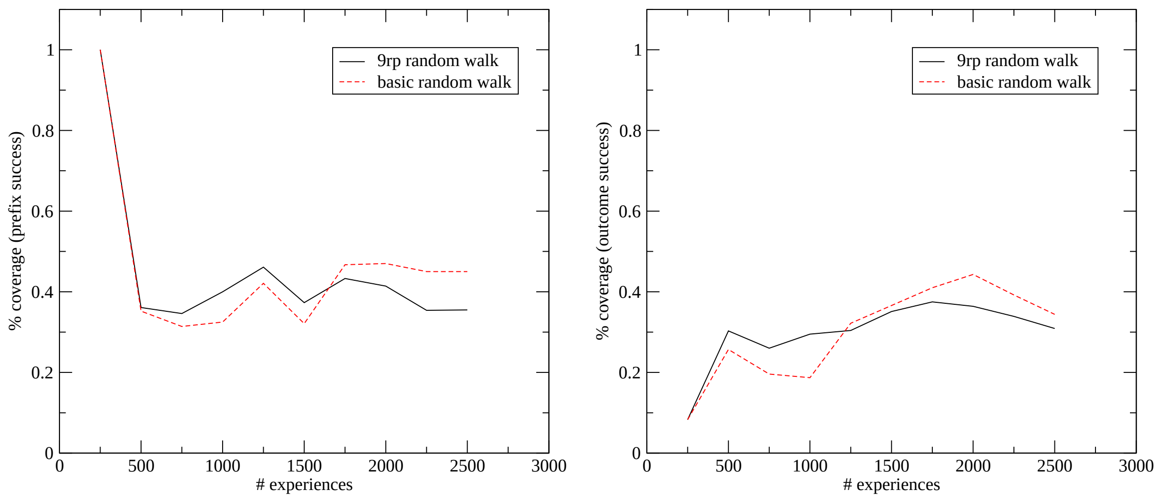

Figure 6.28. Coverage statistics for o_1 (`N-N-N-N-N-N-N-D-D-D-U-N-N-U-N-N`) on random walk data with a planning horizon of 9 and the relaxed planner.

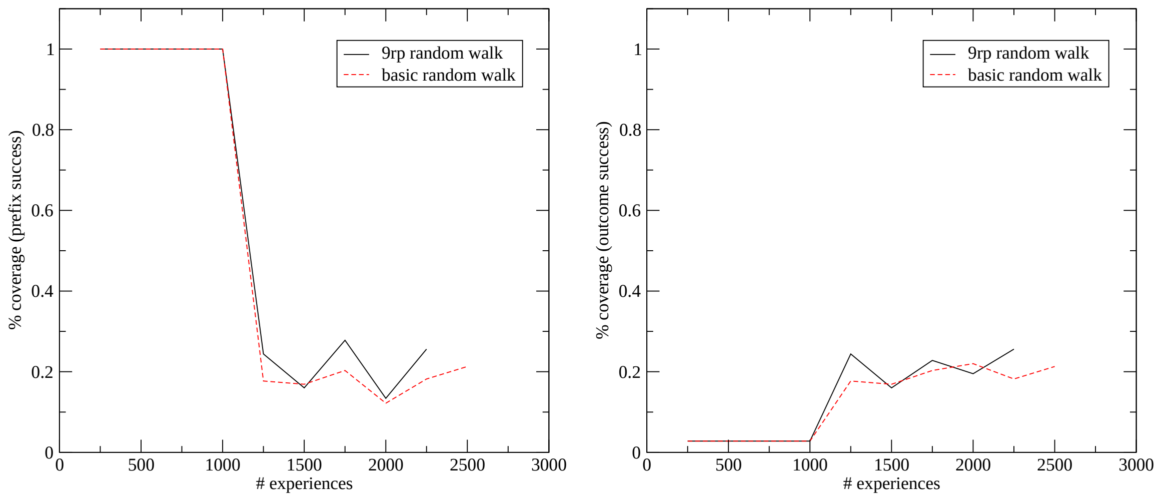

Figure 6.29. Coverage statistics for o_4 (`N-N-N-N-N-D-N-D-N-N-N-N-N-N-N-N`) on random walk data with a planning horizon of 9 and the relaxed planner.

horizon, but under random walk conditions, it may be necessary to expand the horizon significantly (perhaps by 4-6 steps) before a small improvement in coverage will be observed. This is because for each useful plan step in a random trace, we are likely to encounter 3 or more superfluous steps, as well as steps that are counterproductive in reaching the ultimate goal. As we move further from a goal's home state, these additional steps pile up and add to the computational complexity of the planning process.

Likewise, adding in the relaxation step does not have the effect of producing generalized plans by lifting some of the constraints of the original PFT planner. Rather, in experimenting with the relaxed planner, we found that sometimes the opposite was true. Recall that initial conditions are generated by building a decision tree in which experiences that match a target profile are labeled as positive instances and those that do not are marked negative. These experiences are then classified using their precursory sensory states as distinguishing features. Relaxing the target profile has the effect of increasing the number of positive instances and reducing the number of negative instances. Also recall that the decision tree induction algorithm has a tendency to key on "landmarks" that can distinguish one precursory state from another (landmarks are discussed in section 6.1). By generalizing the target profile, we mix a number of different specific profiles into one more general profile. The decision tree induction algorithm can no longer key on landmarks as a result. In cases where landmarks are the most succinct way of expressing initial conditions, initial condition trees will become larger and initial conditions more cumbersome as a result of relaxation. Instead of producing more general plans with easier-to-achieve steps, the opposite can become true. This effect is not terribly common, but the effect is that coverage does not improve as a result of the relaxation process.

In sum, parameter adjustments and steps taken to improve the generality of the planner seem to have little effect on coverage. The planner seems relatively robust against such tinkering.

6.2.5 The Cheat Walk

Each of the previous experiments suggest that we can make minor improvements in performance by following one control policy as opposed to another or by tuning parameters such as planning horizon. They also suggest that characteristics of the goals, such as the number and accessibility of home states, play a role in determining their eventual coverage once the system has run for a while and built good models. We have run additional experiments (which are not included here) to test other aspects of the system such as the execution module and its use of abort reasons. Many of these experiments are ommitted because they suggested similar results: that they probably play a role in coverage, but when viewed in the larger context of 100 potential goals and 72 possible starting states, their contribution is small.

For our final experiment, we will look at a condition in which the agent is given good exemplars, from a variety of starting states, at regular intervals throughout a random walk. We call this condition the *cheat walk*, as the agent does not have to manage to reach the goal outcome on its own in order to get access to traces useful in planning. Every 50 experiences throughout development, the regular random walk is suspended. The agent is then given a sequence of actions to execute that take it to a new starting state, and subsequently it is given another action sequence that will result in an exemplar of a target outcome. In this way, the cheat walk seeds the experience trace with good exemplars. Here we will test the theory that with good exemplars, high levels of coverage can be attained quickly.

Coverage curves and exemplar plots are shown in figures 6.30 through 6.34. Each of these plots shows that exemplars are introduced at a near constant rate (they are

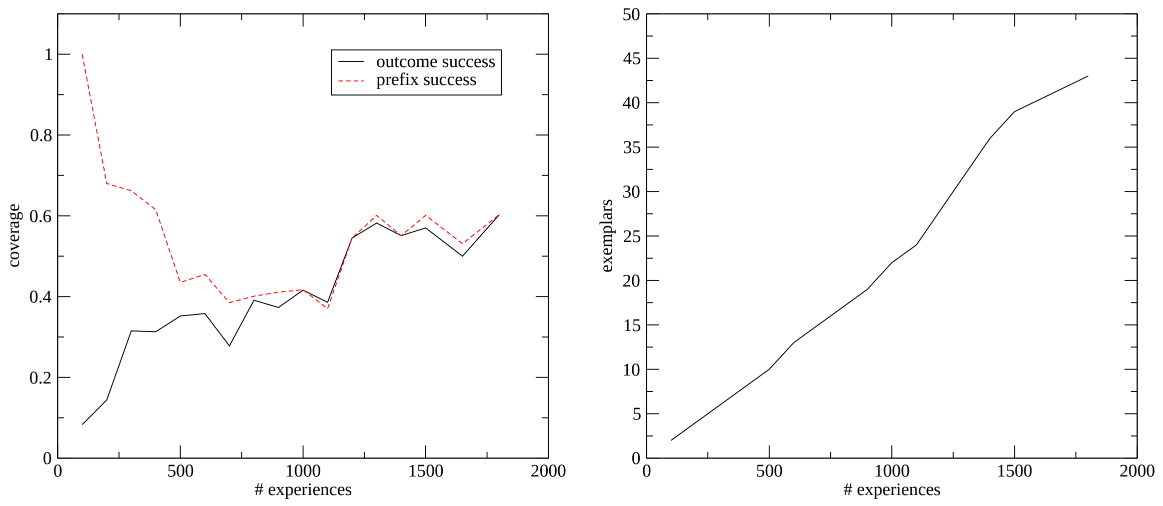

Figure 6.30. Coverage statistics for o_1 (`N-N-N-N-N-N-N-D-D-D-U-N-N-U-N-N`) on random walk data seeded with exemplars.

also encountered during the random walk) and that performance is well above that of a normal random walk. Each of the target outcomes achieves coverage of 60% or greater, with some exceeding 70%. Furthermore, the effects of home states, or their remoteness, appears to be mitigated in the cheat walk. The result appears to be that if useful exemplars exist, the PFT planner will find them.

Coverage plots are shown for two of our target outcomes in figures 6.35 and 6.36. They show the development of excellent coverage centered around the home state of o_2 and o_3 as the agent collects experiences. Tests on outcome o_2, whose remote home state made coverage difcicult in our previous experiments, show an accelerated coverage curve, resulting in high levels of coverage after only 300 experiences seeded with good exemplars. As the models destabilize and restabilize due to the rapid introduction of exemplars, there are periods in which there is a loss of coverage, as shown in the coverage plot at center of figure 6.35. Once the models have become accurate and stable, as they are after 1500 experiences, the existence of good exemplars allows for coverage over a large region of the starting state space. Figure 6.36 shows a similar

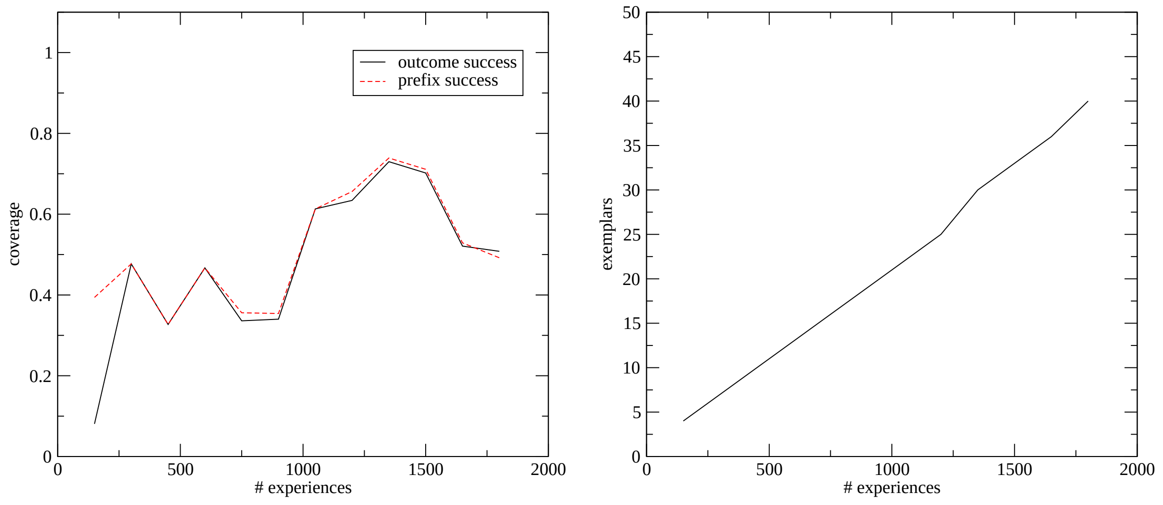

Figure 6.31. Coverage statistics for o_2 (`N-N-D-D-U-N-N-U-N-N-N-N-N-D-N-N`) on random walk data seeded with exemplars.

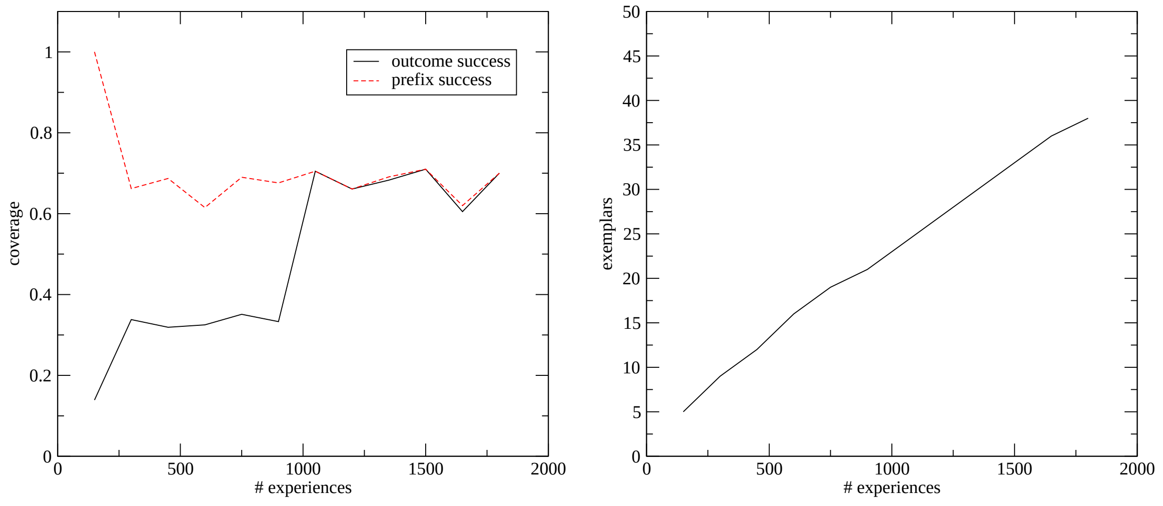

Figure 6.32. Coverage statistics for o_3 (`D-D-N-N-N-N-N-N-N-N-N-N-N-N-N-N`) on random walk data seeded with exemplars.

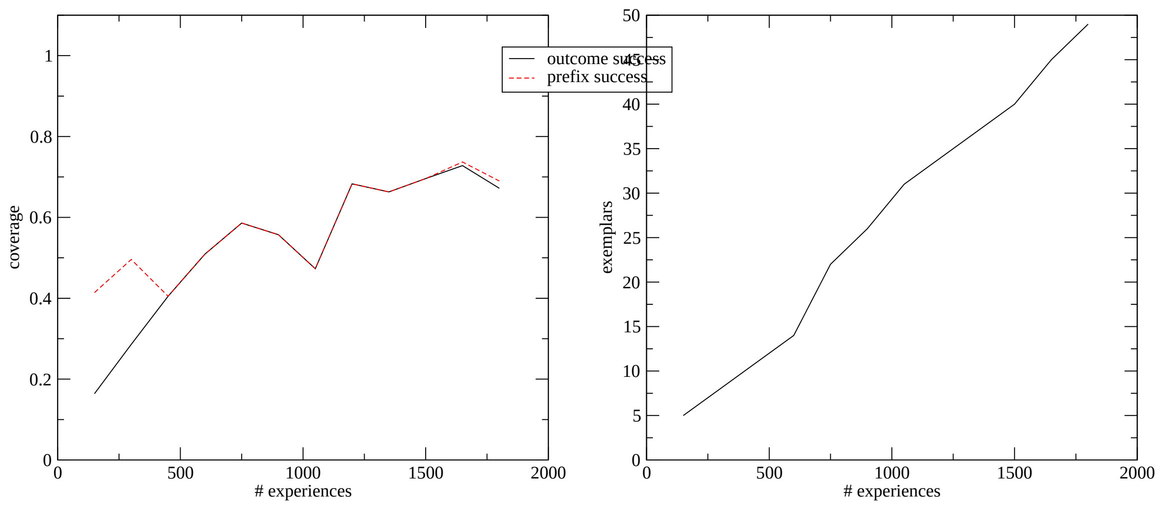

Figure 6.33. Coverage statistics for o_4 `(N-N-N-N-N-D-N-D-N-N-N-N-N-N-N-N)` on random walk data seeded with exemplars.

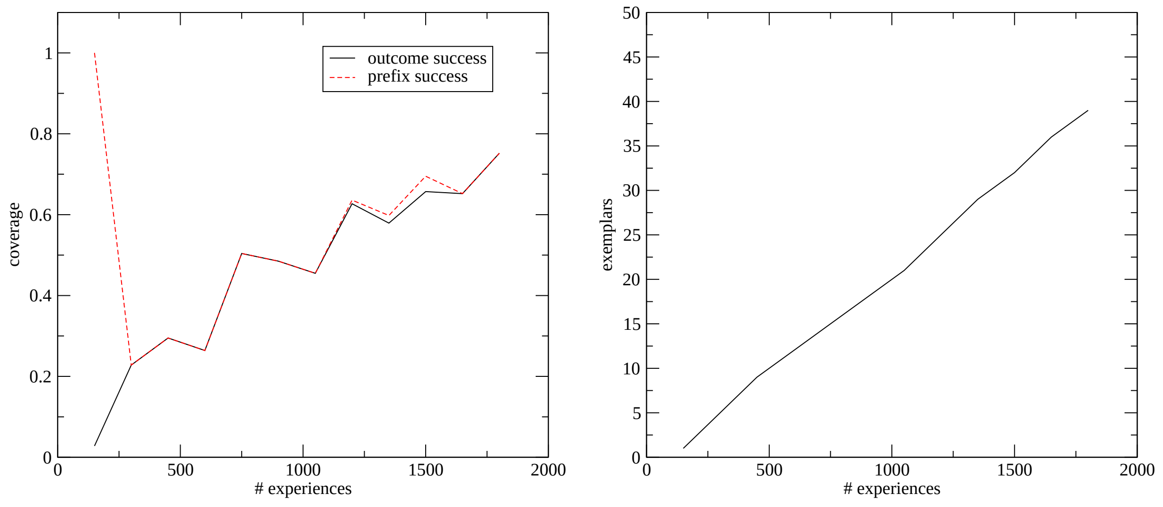

Figure 6.34. Coverage statistics for o_5 `(N-N-N-N-U-N-N-N-D-D-U-N-D-U-U-U)` on random walk data seeded with exemplars.

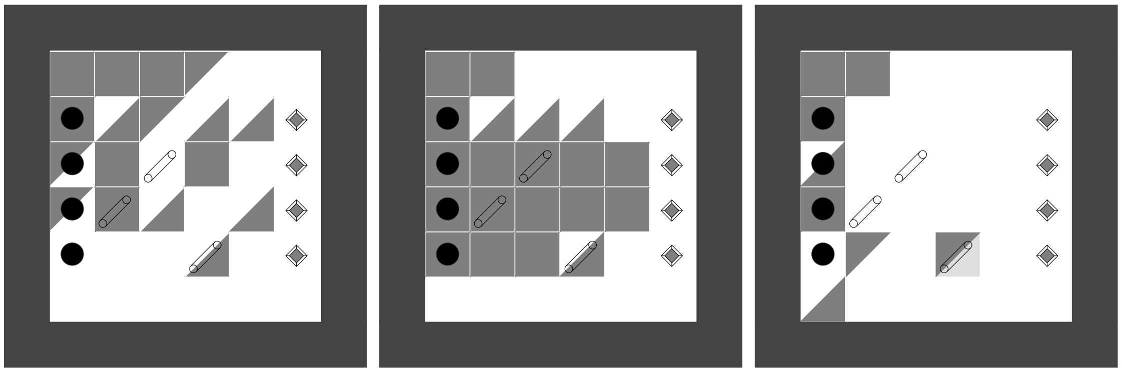
Figure 6.35. Coverage plots for outcome o_2 during a cheat walk after 300, 900, and 1500 experiences, from left to right.

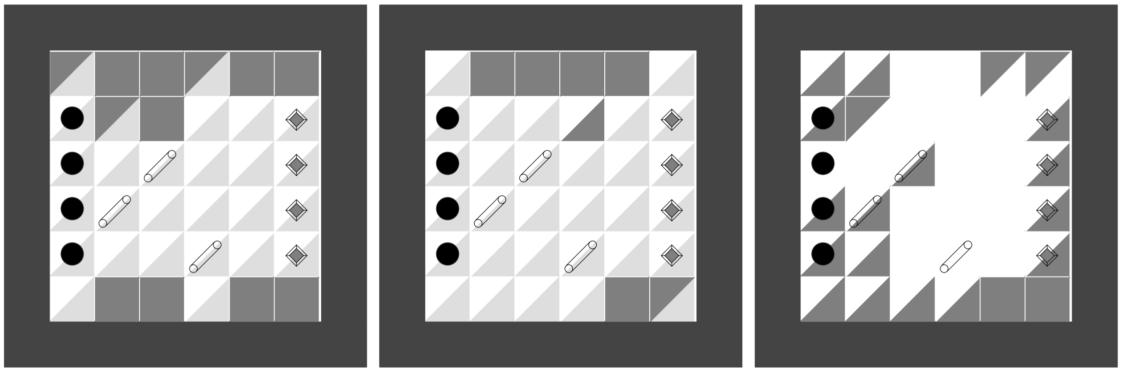
Figure 6.36. Coverage plots for outcome o_3 during a cheat walk after 300, 900, and 1500 experiences, from left to right.

pattern for outcome o_3 in which the agent drops debris into an occupied cell. Note that coverage ramps up quickly for starting states in which the bay is already full; the experience trace is seeded with these exemplars early. After 1500 experiences, exemplars in which the bay is empty at the outset have been experienced, and as a result, more complete coverage is reflected in the rightmost plot of figure 6.36.

6.3 Discussion

Our evaluation targets the two components that are responsible for generating knowledge that runs our system: the modeling system, which produces models relating sensory conditions to action outcomes, and the planner, which produces actionable plans in the form of triangle tables. We evaluated these two system using measures of accuracy and facilitation, respectively, under a number of conditions indended to elucidate the primary contributors to our system's overall success or failure. Many of these conditions were aimed at understanding the link between control policy and the development of activity.

Our accuracy tests suggest several findings. First and foremost, the combination of a reasonably coarse outcome classifier (such as Delta-Simple or Delta) and decision tree induction provide predictive models with accuracy sufficient to serve as a basis for our developmental system. Even in the absence of a sensible control policy, our models are adequate to score 50% at the winner-take-all prediction task with under $1,000$ experiences over all 432 state/action pairs in the GridSim domain. If we break the accuracy measure down to sensor-by-sensor predictions, by looking at HI disparity, that number goes up to 65% when the agent has $1,000$ experiences to go by. Adding a hill-climbing control policy that executes the action about which the least is known, which we call the L0 controller, accelerates learning by focusing on opportunities to learn. Under this control policy, 80% HI accuracy is possible with only 500 experiences - only a few more than the 432 possible state/action pairs. That accuracy number rises to better than 90% by $1,000$ experiences. Ultimately, the questions we sought to answer with the accuracy test were: "What level of prediction accuracy can we expect from our modeling system?" and "Is that level good enough to suggest that a planner based on those models might have a resaonable chance of achieving its goals?" The answer to the first question is that for an arbitrary state/action pair, in a domain where noise is not a significant problem, the modeling system is sufficiently accurate to predict sensor trends at least 95% of the time after 2,000 experiences under the L0 policy. This suggests an affirmative answer to the second question.

Peripheral questions that the accuracy tests were designed to answer dealt with how the modeling system reacts to properties of the environment and how sensitive prediction accuracy is with regards to the outcome classifier. Our experiments with a larger simulation showed that with a considerably larger grid, accuracy did not suffer, although the learning rate did slow down proportionately with the increase in state/action pairs. We also looked at partial-observability issues in the GridSim simulator. Our modeling system will opportunistically use fixtures in the environment to improve predictive accuracy, as witnessed in the statically configured GridSim domain. Where partial observability cannot be skirted, impure decision tree leaves will inevitably appear in the underlying model. Impure leaves suggest a distribution of outcomes, and their observed frequencies are known so that probabilistic predictions can be made. In many cases, the sensors whose behavior cannot be predicted can be determined by analysis of the outcomes. Through this analysis it is possible to make partial outcome predictions with 100% accuracy. We might call these incomplete predictions "generalized operators", and in some cases these operators can still be useful in planning.

Finally, we considered the impact of outcome classification on model quality. The ssc classifiers offer three levels of granularity. We found that the Delta and Delta-Simple, which are based on simple mathematics and hard limits, offer no resistance to the initial condition induction process in the GridSim dp,aom, allowing the highest levels of accuracy. Their simple expressiveness is limited to describing outcomes by their net result. The subtleties of "shape" in sensor behavior are lost. For the purposes of planning, it is not a terrible tradeoff to be made, since planning operators are traditionally specified in this way (by net change) anyways. The ssc1 classifier does capture the subtleties of shape at the expense of introducing hypothesis tests in the piecewise linear fit algorithm. The combination of a greedy algorithm and the possibility of overfitting result in mutliple interpretations for experiences produced by a single generating process. The result is expressiveness at the cost of accuracy. In the end, it may be possible to move between levels of granularity as necessary to navigate the tradeoff between expressiveness and accuracy.

Our facilitation experiments with the GridSim simulator show that performance hinges first on the stabilization of good models over an initial period determined by the modeling system in conjunction with a reasonable exploration policy. Once the modeling system has settled down, the primary influence on facilitation is the existence of good exemplars for the PFT planner to work with.

A good exemplar takes the agent from a unique state to a goal state without any superfluous actions along the way; superfluous or coutnerproductive actions have the effect of pushing distant start states outside the planning horizon, where it will be impossible for our planner to successfully retrieve a plan. All the other candidate influences that we identified at the outset of our facilitation experiments seem to be related through their association with good exemplars: the control policy, planning horizon, exemplar count, and even characteristics of the goal outcome like accessibility and number of home states. The control policy determines how focused experience traces will be in moving towards a goal outcome. Planning horizon can trade off the ability to wade through superfluous steps to recover plans from "messy" experience traces at the cost of additional computation time. Goal characteristics simply make the generation of useful exemplar traces more or less likely under a random walk or even a greedy exploration policy like lpq.

In the absence of any way to consistently produce good experience traces from unique starting states (our "cheat walk" flies in the face of any disciplined or believable approach to development), we can reasonably expect to achieve moderate levels of facilitation coverage (in the 30-40% range) in and around the home states of most outcomes. A combination of guided exploration, such as the L0 controller provides, and exploitation such as the PFT planner, seems to be the most sensible approach. The L0 controller provides a way to improve models and produce unique trajectories through the state space, and the planner provides a way to test and optimize existing experience traces that can be used to achieve goals. The combination of these two policies in a balanced controller such as our motivational system should provide a mechanism for extending the range of our planner beyond what was experienced under any one control policy. In cases where good experience traces can be experienced to cover a wide variety of starting states, we can expect facilitation coverage scores in the 60-70% range, or perhaps beyond.

This final conclusion – that the system must produce good exemplars originating from all over the state space – is not at odds with any of the theoretical foundations of this dissertation. It is, in fact, in close accordance with Piaget's theories and the interactionst philosophy that we drew upon for our thesis. Developing agents repeat and exercise behaviors with subtle variations to increase coverage for their developing activities. This type of repeated experimentation and integration of smaller activities into larger ones is what Piaget called a primary circular reaction.

One of the assumptions of our system when we chose to use the case-based PFT planner was that whatever control policy we used, be it the motivational system described in chapter 5, a random walk, or any other control system, combined with generalizability in the domain, would enable coverage to spread throughout the state space. What we found with our simple 8×8 GridSim domain was that generalization was not terribly prevalent. Most SSC outcomes in the GridSim domain had only 2 or 4 home states. As a result, outcome models are not very general, and neither are the types of plans the PFT planner builds. In order for PFT to generate a plan from any given experience trace, the agent more or less needs to be sitting somewhere along that trace. This is why a wide variety of succinct exemplar traces plays such a crucial role in the performance of our system.

Our system lacks two key components that could potentially close the loop on a Piagetian type of development. First, we lack a control policy that can consistently set up the primary circular reaction. A simple example of a primary circular reaction in a human infant would be *mouthing*. Mouthing is a behavior in which a child takes an object and puts it (or part of it) into her mouth. Between the environment and the whims of the child, opportunities for mouthing are everywhere. Anything within reach is fair game, and a child can put down an object and easily return to it to attempt a new and innovative variation on mouthing that same old object. Opportunities, at least in the GridSim environment, are limited. Many outcomes exist only in a single grid cell, and once the outcome has been encountered, setting up a novel situation in which to exercise that outcome may require as intricate (or more intricate) a plan as will be required to exercise the original outcome. Second, our system lacks a generative component that can *extend* plans generated by the PFT planner. We have found that the PFT planner can build a plan if it can find an expereince trace that leads to the goal and has an inroad at the current state. But executing this plan does not produce any new procedural knowledge. The PFT planner is purely exploitative. A generative component would use operator models to produce novel experience traces. We found full-blown generative planning fraught with difficulties in chapter 3; backward-chaining was a particular trouble because of the complexities of state projection in a dynamic, continuous state space. But if the generative component were limited to one-step plans, then state projection would not be necessary. That is, suppose PFT could find no experience trace with an inroad at the current state, but it could find an experience trace which came within a single initial condition of matching the current state. Perhaps there is an operator that could achieve that single condition in a single invocation. This would not require state projection to determine. The operator could be executed, and then PFT could reattempt to find an inroad into the experience trace. If the operator succeeded in connecting to an inroad, the execution would have the effect of "extending the exemplar". This process, repeated whenever it was opportune, would have the effect of widening coverage until the planning horizon is reached; an extremely simple generative component combined with a case-based planner, and plenty of practice time, could result in eventual coverage of the starting state space.

CHAPTER 7
CONCLUSION

This dissertation describes a model of the development of activities for situated agents. Our system comprises four distinct components that handle different parts of the developmental process: a modeling system that learns the context-dependent effects of actions, a planner that uses experience traces to reproduce desirable activities, a motivational system that decides among the many things an agent might spend its time doing, and an execution component that controls the implementation of plans into physical activity.

The primary contributions of this work stem from the distinction which we draw between learning and development. Learning is the process of acquiring and refining a particular knowledge structure. An agent learns how to grasp an object or what it means to be graspable. Development is an ongoing process of posing, solving, and later refining learning tasks.

Many examples of capable learning systems exist in literature. Few systems can be considered developmental systems according to definition above because they cannot pose their own learning problems. A system might choose from a prespecified list of goals or it may learn the dynamics of a reward signal in order to learn a particular behavior. The centerpiece of our system is the Piaget-inspired concept of *planning to act*. The planning to act philosophy states that activity itself is rewarding and consequently, the goals of an agent are to engage in and exercise activities, not to achieve arbitrary sensory or perceptual states. Planning to act, in conjunction with our motivational system, allows our system to define, select, and pursue its own goals.

While it is a simple idea, this idea of planning to act is what allows us to make the leap from planning and learning to a system that develops activities.

A system that not only learns but develops activities is a significant contribution for two major reasons. The first and most practical reason is that a developing agent will learn whatever activities an environment affords, in the absence of supervision. As it interleaves exploration and exploitation it develops a corpus of activities that will frequently be transferable to supervised tasks that may arise later in its lifetime, even if the sensor and effector implementation of the agent is changed. A human experimenter may choose to allow an agent to explore and achieve some level of competence on its own before exerting pressure (through the motivational system) to solve specific domain tasks that have general utility (like dropping debris in receptacles) once it has a reasonable chance of doing so.

The second benefit of a developmental system is that it can serve as a platform for later conceptual development. Our system generates two types of operational knowledge: domain models and activities in the form of plans. We believe that the combination of these two forms of knowledge are foundational to *classes, concepts* and *language*. Our initial conditions contain conceptual information that, together with outcome classes, could be considered the beginnings of concept, for example. Consider the outcome of lifting debris into the cargo bay in the GridSim simulator. The initial conditions may specify that objects of a certain shape will result in a successful lift whereas other objects will not. Those conditions that are attached to the object in the current cell sensor form an intensional class of things that allow a lift to succeed. They define the concept of what is *liftable*. Later developmental systems have the opportunity to attach language to meanings rooted in the domain and activity models of our system.

Our evaluation is designed to validate the primary contributions of this dissertation: that planning to act allows us to make the leap from learning to development, that planning can serve as the basis of development, that our system generates useful, declarative models of its environment, and that in the end our system does develop a library of useful and reusable activities. The evaluation itself is significant for methodological reasons. By moving to the highly controlled GridSim environment, we were able to perform a systematic empirical analysis of development and the factors that determine success or failure of planning and modeling in our system. These analyses allow us to draw a number of conclusions. First, our modeling system creates models of the environment that improve over time. Accuracy in a controlled, deterministic domain can approach 100% in some cases, and the declarative models our system generated reflect actual structure present in the environment that we believe can form the basis for conceptual structures such as language and concepts. Second, planning to act allows an agent that can identify the qualitatively distinct outcomes of its actions to pursue outcomes as goals in an unsupervised manner. This allows an agent to consider a fixed number of goals, and assign value to them, in cases where the state space is unbounded or virtually so. Our ssc filters provide examples of one way an agent can distinguish among the different outcomes of its actions, and our facilitation results show that arbitrary goals can be pursued and that facilitation does develop with experience. Third, experiments showed that our case-based PFT planner was able to generate plans that allowed our agent to reproduce outcomes. Importantly, our experiments with different control policies show that facilitation for some outcome *o* can improve regardless of whether exploration is biased toward developing *o* or not. Although assuring good traces leading to *o* is the single most important factor in improving facilitation, moderate levels of facilitation can be achieved for most outcomes by simply setting an agent out on a random walk. The fact that good traces in the experiences of the robot are the most significant factor to performance at once suggests two things: that our system is for the most part robust against small changes in parameters such as planning horizon, and that the most fruitful lines of future work will be those that involve either directed exploration or simple ways of extending existing PFT plans (to produce more good experience traces).

7.0.1 Limitations and Future Work

The evaluation of our system in the GridSim domain was in general very positive in that we demonstrated that our system can engage in a developmental process in which an agent becomes increasingly able to achieve its goals. At the same time, some general limitations of our system became apparent in the course of our evaluation, and our results were most suggestive that there are a great many opportunities to extend this system in future work.

The major limitations fall into two categories. Technical limitations are related to our choice of algorithms. That our system makes frequent use of landmarks to locate its exact location in the grid is a limitation that stifles the generation of general initial conditions. This limitation is likely a result of a domain with few opportunities for generalization and a greedy tree building algorithm that selects sensory features that most accurately differentiate action outcomes. Another technical limitation imposed by the tree-building component is that the initial condition space is carved up by axis-parallel splits only. [1]

Another type of limitation is representational. The results reported in this dissertation are also of a slightly less ambitious nature than the proposal that precedes it. In section 1.2, we considered the economy of hierarchical activity representations. The work in this dissertation represents the development of what we called *flat* representations, in which each step of a plan is a primitive action. Our conviction that the economy of hierarchical representations is an integral part of intelligent behavior

[1]This is a limitation present in most decision tree inductions, but lifted in a few, such as the algorithms described in [57] and [35].

remains. The work in this dissertation can be seen as a first step to learning more complicated activities that are made up of sophisticated hierarchies of behavior.

The limitations and contributions of this work suggest three distinct lines of research for future work: improving performance, integrating concept and language learning into our developmental framework, and developing activities in new and more challenging domains.

Performance improvements could be achieved a number of ways. We are most optimistic that adding a very simple generative planning component to the PFT planner, as described in section 6.3, will ensure that an agent is able to expand its effective planning radius over time by extending short, working PFT plans one step at a time. Also, we note that the results presented in this dissertation are without plan caching scheme. In addition to an improvement in computation time, we believe that cached plans can themselves be utilized by the simple generative component, allowing our planner to produce hierarchical activities, and effectively increasing the planning radius at the same time.

That our system is producing declarative models of its environment and the activity it affords suggests that opportunities for integrating concept and language learning already exist. We intend to expound upon the results of this work by building grounded concept and language learning systems on top of it. Related work on concept learning [68, 69] and language learning [62] share in our interactionist philosophy. We envision a system which simultaneously tackles these three aspects of development at once.

Finally, we would like to broaden the application of our system to new agents and domains. It has always been our contention that richer activities and concepts are afforded by more sophisticated domains. Our experiments hinted that this was true, as GridSim appears to be a poorly generalizable domain with limited opportunities for sophisticated interactions. Our initial experiments with the Pioneer mobile robot indicate that such domains can be approached by a system such as ours. These domains, while more challenging to model, and more challenging to execute plans competently in, offer rewards in the way of more sophisticated emergent behavior and conceptual knowledge.

APPENDIX

TESTING FOR SLOPE DIFFERENCES

To classify the effects of an action on sensor readings, our system uses piecewise linear regression to discretize sensor time seris into strings of segment descriptions. The piecewise linear fit algorithm is top-down, and is based around a test that compares the slope of the sensor series before and after after a candidate split point.

The comparison works as follows. Let $before_i$ be the slope for sensor i before the action, and $after_i$ be the slope of ten values from sensor i after the action and a short delay. (The delay gives the action time to have an effect. Presently, the delay is fixed at 1000 ms, although in future work, the robot may learn an appropriate delay for each sensor.) To classify the effects of an action, the sensor monitor calculates $before_i - after_i$ for each sensor i, and converts the result to a class label, '+', '−', or '0', representing an increase, a decrease, or no change in slope, respectively. More precisely, the monitor tests the hypothesis that $before_i = after_i$ with the following statistical comparison of the slopes of two regression lines:

$$t = \frac{before_i - after_i}{\hat{\sigma}_{b_1 - b_2}}, \tag{A.1}$$

where $\hat{\sigma}_{b_1-b_2}$ is the estimate of the standard error for the difference of slopes from two regression lines. This estimate is computed from the pooled variance, or sums of squares of residuals from each least-squares line:

$$\hat{\sigma}_{b_1-b_2} = \sqrt{\hat{s}^2_{pooled}\left(\frac{1}{SS_{X_1}} + \frac{1}{SS_{X_2}}\right)} \tag{A.2}$$

$$\hat{s}^2_{pooled} = \frac{SS_{residual_1} + SS_{residual_2}}{df} \tag{A.3}$$

$$SS_{residual} = \left(1 - r^2\right) SS_Y \tag{A.4}$$

In Eqs. (A.2)-(A.4), SS_X and SS_Y are the sums of squares for the independent and dependent variables (respectively, time and sensor i), and r^2 is the goodness of fit for the regression line. Finally, the t statistic is compared to the t distribution with $n_1 + n_2 - 4$ degrees of freedom to see whether the hypothesis of equal slope should be rejected [42]. In our experiments, $n_1 = n_2 = 10$, because each slope is based on ten observations.

BIBLIOGRAPHY

[1] Atwood, M.E., and Polson, P.G. A process model for water jug problems. *Cognitive Psychology 8* (1976), 191–216.

[2] Barto, A., Bradtke, S., and Singh, S.T. Learning to act using real-time dynamic programming. *Artificial Intelligence 72*, 1 (1995), 81–138.

[3] Bellman, Richard. *Dynamic Programming*. Princeton University Press, 1957.

[4] Benson, Scott. Action model learning and action execution in a reactive agent. In *Proceedings of the 1995 International Joint Conference on AI* (August 1995).

[5] Benson, Scott. Inductive learning of reactive action models. In *Proceedings of the Twelfth International Conference on Machine Learning* (1995), pp. 47–54.

[6] Bishop, C. M. *Neural Networks for Pattern Recognition*. Clarendon Press, 1995.

[7] Boyan, Justin, and Littman, Michael. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in Neural Information Processing Systems 6*, J.D. Cowan, G. Tesauro, and J. Alspector, Eds. Morgan Kaufmann Publishers, Inc., 1994.

[8] Breiman, L., Friedman, J., Olshen, R., and Stone, C. *Classification and Regression Trees*. Wadsworth International, 1984.

[9] Carbonell, Jaime G., Blythe, Jim, Etzioni, Oren, Gil, Yolanda, Joseph, Robert, Kahn, Dan, Knoblock, Craig, Minton, Steven, Perez, Alicia, Reilly, Scott, Veloso, Manuela, and Wang, Xuemei. PRODIGY4.0: The manual and tutorial. School of Computer Science CMU-CS-92-150, Carnegie Mellon University, June 1992.

[10] Cheeseman, P., Freeman, D., Kelly, J., Self, M., Stutz, J., and Taylor, W. Autoclass: a bayesian classification system. In *Proceedings of the Fifth International Conference on Machine Learning* (1988), Morgan Kaufmann.

[11] Clark, P., and Niblett, T. The CN2 induction algorithm. *Machine Learning 3*, 4 (1989), 261–283.

[12] Cohen, Paul R., and Jensen, David. Overfitting explained. Submitted to the Fourteenth International Conference on Machine Learning, 1997.

[13] Corman, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. MIT Press, 1990.

[14] Craven, M.W. *Extracting Comprehensile Models from Trained Neural Networks.* PhD thesis, University of Wisconsin-Madison, 1996.

[15] Craven, M.W., and Shavlik, J.W. Learning symbolic rules using artificial neural networks. In *Proceedings of the Tenth International Conference of Machine Learning* (1993), Morgan-Kaufmann, pp. 73–80.

[16] Cybenko, G. Continuous valued neural networks with two hidden layers are sufficient, 1988.

[17] Dayan, Peter, and Sejnowski, Terrence J. Td(λ) converges with probability 1. *Machine Learning 14(3)* (1994).

[18] Dietterich, Thomas G. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research 13* (2000), 227–303.

[19] Drescher, Gary L. *Made-Up Minds.* MIT Press, 1991.

[20] Duda, Richard O., and Hart, Peter E. *Pattern Recognition and Scene Analysis.* John Wiley & Sons, Inc., 1973.

[21] Everitt, Brian. *Cluster Analysis.* John Wiley & Sons, Inc., 1993.

[22] Fikes, Richard E., Hart, Peter E., and Nilsson, Nils J. Learning and executing generalized robot plans. *Artificial Intelligence 3*, 4 (1972), 251–288.

[23] Fisher, D. H., and McKusick, K. B. An empirical comparison of id3 and backpropogation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (1989), pp. 788–793.

[24] Fisher, Douglas H. Knowledge acquisiution via incremental conceptual clustering. *Machine Learning 2* (1987), 139–172.

[25] Francois, G.R. Le. *Theories of human learning.* Cole Publishing Co., 1995.

[26] Funahashi, K. the approximate realization of continuous mappings by neural networks, 1989.

[27] Gardner, M. Mathematical games. *Scientific American* (December 1979), 20–24.

[28] Gil, Yolanda. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, Carnegie Mellon University, 1992.

[29] Gil, Yolanda. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning* (1994), pp. 87–95.

[30] Ginsberg, Herbert P., and Opper, Sylvia. *Piaget's Theory of Intellectual Development.* Prentice Hall, Englewood Cliffs, NJ, 1988. 3rd Edition.

[31] Haigh, Karen Zita. *Learning Situation-Dependent Costs: Improving Planning from Probabilistic Robot Execution.* PhD thesis, Carnegie Mellon University, 1998.

[32] Hauskrecht, Milos, Meuleau, Nicolas, Kaelbling, Leslie Pack, Dean, Thomas, and Boutilier, Craig. Hierarchical solution of Markov decision processes using macro-actions. In *Uncertainty in Artificial Intelligence* (1998), pp. 220–229.

[33] Hornik, K., Stinchcombe, M., , and White, H. Multilayer feedforward networks are universal approximators. *Neural Networks 2* (1989).

[34] Intrator, Orna, and Intrator, Nathan. Interpreting neural-network results: A simulation study.

[35] Ittner, A., and Schlosser, M. Non-linear decistion trees – ndt. In *Proceedings of the Thirteenth International Conference on Machine Learning* (1996), pp. 252–257.

[36] Jaakkola, Tommi, Jordan, Michael I., and Singh, Satinder P. Convergence of stochastic iterative dynamic programming algorithms. In *Advances in Neural Information Processing Systems*, Jack D. Cowan, Gerald Tesauro, and Joshua Alspector, Eds., vol. 6. Morgan Kaufmann Publishers, Inc., 1994, pp. 703–710.

[37] Jelinek, Frederick. *Statistical Methods for Speech Recognition.* MIT Press, 1997.

[38] Jensen, David, and Schmill, Matthew D. Adjusting for multiple comparisons in decision tree pruning. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining* (1997), pp. 195–198.

[39] Kaelbling, Leslie Pack. Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning* (1993), Morgan Kaufmann, pp. 167–173.

[40] Keogh, Eamonn, and Pazzani, Michael J. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *Working Notes of the AAAI-98 workshop on Predicting the Future: AI Approaches to Time-Series Analysis* (1998), pp. 44–51.

[41] Kononenko, I. *Current trends in knowledge acquisition.* IOS Press, Amsterdam, 1990, ch. Comparison of inductive and naive Bayesian Learning approaches to automatic knowledge acquisition.

[42] Kotz, S., and Johnson, N.L., Eds. *Encyclopedia of Statistical Sciences.* Wiley, New York, 1982-1989.

[43] Koza, J. R. *Genetic Programming.* MIT Press, Cambridge, MA, 1992.

[44] Kruskall, Joseph B., and Liberman, Mark. The symmetric time warping problem: From continuous to discrete. In *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison* (1983), Addison-Wesley.

[45] Laird, John E., Rosenbloom, Paul S., and Newell, Allen. Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning 1* (1986), 11–46.

[46] Lakoff, George. *Women, Fire, and Dangerous Things.* University of Chicago Press, 1984.

[47] Lakoff, George, and Johnson, Mark. *Metaphors We Live By.* University of Chicago Press, 1980.

[48] Lang, K.J, and Hinton, G.E. The development of the time-delay neural network architecture for speech recognition. Tech. Rep. CMU-CS-88-152, Carnegie Mellon University, 1988.

[49] Littman, M. Probabilistic strips planning is exptime-complete. Tech. Rep. CS-1996-18, Duke University Department of Computer Science, 1996.

[50] Mahadevan, Sridhar, and Connell, Jonathan. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence 55*, 2–3 (1992), 189–208.

[51] Martin, J., and Hirschberg, D. the complexity of learning decision trees, 1996.

[52] Mataric, M.J. Reward functions for accelerated learning. In *ML94* (1994), Morgan Kaufmann, pp. 181–189.

[53] McCulloch, W.S., and Pitts, W. A logical calculus of the ideas immanent in nervous activity. In *Bulletin of Mathematical Biophysics* (1943), vol. 5.

[54] McGovern, Amy, and Barto, Andrew G. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the 18th International Conference on Machine Learning* (2001), pp. 361–368.

[55] Melnik, O. Representation of information in neural networks, 2000.

[56] Mitchell, Tom. *Machine Learning.* McGraw Hill, 1997.

[57] Murthy, S.K., Kasif, S., and Salzberg, S. A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research 2* (1994).

[58] Murthy, Sreerama K., and Salzberg, Steven L. Investigations of the greedy heuristic for classification tree induction.

[59] Newell, Allen, and Simon, H.A. GPS: A program that simulates human thought. In *Computers and Thought*, E. A. Feigenbaum and J. Feldman, Eds. McGraw-Hill, New York, 1963, pp. 279–293.

[60] Newell, Allen, and Simon, H.A. *Human Problem Solving.* Prentice Hall, 1972.

[61] Nilsson, N.J. *Learning Machines.* McGraw-Hill, 1965.

[62] Oates, James T. *Grounding Knowledge in Sensors: Unsupervised Learning for Language and Planning.* PhD thesis, Department of Computer Science, University of Massachusetts, 2001.

[63] Oates, Tim, Schmill, Matthew D., and Cohen, Paul R. Identifying qualitatively different outcomes of actions: Experiments with a mobile robot. In *Working Notes of the IJCAI-99 Workshop on Robot Action Planning* (1999).

[64] Penberthy, J., and Weld, D. S. Temporal planning with continuous change. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (Menlo Park, CA, 1994), AAAI/MIT Press, pp. 1010–1015.

[65] Pomerleau, D.A. Alvinn: An autonomous land vehicle in a neural network. In *Advances in Neural Information Processing*, D. Touretzky, Ed., vol. 1. Morgan-Kaufmann Publishers, 1989.

[66] Quinlan, J. R. *C4.5 : programs for machine learning.* Morgan Kaufmann Publishers, Inc., 1993.

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

[68] Rosenstein, Michael, and Cohen, Paul R. Concepts from time series. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence* (1998), AAAI Press, pp. 739–745.

[69] Rosenstein, Michael T., and Cohen, Paul R. Continuous categories for a mobile robot. Submitted to Sixteenth National Conference on Artificial Intelligence, 1999.

[70] Ruff, H.A., and Rothbart, M.K. *Attention in early development: Themes and variations.* Oxford University Press, New York, 1996.

[71] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature 323* (1986), 533–536.

[72] Sankoff, David, and Kruskal, Joseph B., Eds. *Time Warps, String Edits, and Macromolecules: Theory and Practice of Sequence Comparisons.* Addison-Wesley Publishing Company, Reading, MA, 1983.

[73] Schmill, Matthew D. Predicting outcome classes for the experiences of a mobile robot. EKSL Memorandum #99-33, February 1999.

[74] Schmill, Matthew D., Rosenstein, Michael T., Cohen, Paul R., and Utgoff, Paul. Learning what is relevant to the effects of actions for a mobile robot. In *Proceedings of the Second International Conference on Autonomous Agents* (1998), pp. 247–253.

[75] Shannon, Claude. A mathematical theory of communication. *Bell System Technical Journal 27* (1948), 379–423.

[76] Singh, Satinder P. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence* (1992).

[77] Skinner, B.F. *The behavior of organisms: An experimental analysis.* Appleton-Century, 1938.

[78] Sussman, Gerald J. *A Computer Model of Skill Acquisition.* American Elsevier, New York, 1975.

[79] Sutton, Richard S. Dyna, an integrated architecture for learning, planning and reacting. *SIGART Bulletin 2*, 4 (August 1991), 160–163.

[80] Tate, A. Project planning using a hierarchical non-linear planner. Tech. Rep. Dept. of Artificial Intelligence Report 25, Edinburgh University, 1976.

[81] Thrun, Sebastian B. The role of exploration in learning control. In *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*, David A. White and Donald A. Sofge, Eds. Van Nostrand Reinhold, Florence, Kentucky 41022, 1992.

[82] Utgoff, Paul, and Cohen, Paul R. Applicability of reinforcement learning. In *The Methodology of Applying Machine leraning Problem Definition, Task Decompostion and Technique Selection Workshop, ICML-98* (1998), pp. 37–43.

[83] Utgoff, Paul E., Berkman, N.C., and Clouse, J.A. Decision tree induction based on efficient tree restructuring. *Machine Learning 29* (1997), 5–44.

[84] Veloso, Manuela, Carbonell, Jaime, Perez, Alicia, Borrajo, Daniel, Fink, Eugene, and Blythe, Jim. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical AI 7*, 1 (1995).

[85] Wallace, C.S., and Dowe, D.L. Intrinsic classification by mml: the snob program. In *Proceedings of the Seventh Australian Joint Conference on Artificial Intelligence* (1994).

[86] Wang, Xuemei. Learning planning operators by observation and practice. In *Artificial Intelligence Planning Systems* (1994), pp. 335–340.

[87] Wang, Xuemei. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning* (1995).

[88] Watkins, C.J.C.H, and Dayan, P. Q-learning. In *Machine Learning 8* (1992), pp. 279–292.

[89] Werbos, P.J. Backpropogation through time: What is does and how to do it. *Proceedings of the IEEE 78* (1990), 1550–1560.

[90] Wisotzki, Christel, and Wysotzki, Fritz. Prototype, nearest neighbor, and hybrid algorithms for time series classification. In *Proceedings of the Eighth European Conference on Machine Learning* (1995), pp. 364–367.