Situation-Dependent Learning for Interleaved Planning and Robot Execution

Karen Zita Haigh

February 1998

CMU-CS-98-108

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Manuela M. Veloso, Chair
Tom Mitchell
Reid Simmons
R. James Firby (Neodesic Corporation)

This research is sponsored in part by (1) the National Science Foundation under Grant No. IRI-9502548, (2) by the Defense Advanced Research Projects Agency (DARPA), and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-95-1-0018, (3) the Natural Sciences and Engineering Council of Canada (NSERC), and (4) the Canadian Space Agency (CSA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, DARPA, Rome Laboratory, the U.S. Government, NSERC or the CSA.

Keywords: Artificial intelligence, robotics, PRODIGY, Xavier, interleaving planning and execution, execution monitoring, asynchronous goals, machine learning, situation-dependent rules, situation-dependent costs, plan quality, planning performance, execution performance, search control knowledge.

Abstract

This dissertation presents the complete integrated planning, executing and learning robotic agent ROGUE.

Physical domains are notoriously hard to model completely and correctly. Robotics researchers have developed learning algorithms to successfully tune operational parameters. Instead of improving low-level *actuator* control, our work focusses instead at the *planning* stages of the system. The thesis provides techniques to directly process execution experience, and to learn to improve planning and execution performance.

ROGUE accepts multiple, asynchronous task requests, and interleaves task planning with real-world robot execution. This dissertation describes how ROGUE prioritizes tasks, suspends and interrupts tasks, and opportunistically achieves compatible tasks. We present how ROGUE interleaves planning and execution to accomplish its tasks, monitoring and compensating for failure and changes in the environment.

ROGUE analyzes execution experience to detect patterns in the environment that affect plan quality. ROGUE extracts learning opportunities from massive, continual, probabilistic execution traces. ROGUE then correlates these learning opportunities with environmental features, thus detecting patterns in the form of *situation-dependent rules*. We present the development and use of these rules for two very different planners: the path planner and the task planner. We present empirical data to show the effectiveness of ROGUE's novel learning approach.

Our learning approach is applicable for any planner operating in any physical domain. Our empirical results show that situation-dependent rules effectively improve the planner's model of the environment, thus allowing the planner to predict and avoid failures, to respond to a changing environment, and to create plans that are tailored to the real world. Physical systems should adapt to changing situations and absorb any information that will improve their performance.

Acknowledgements

It's often said that the only important decision one makes in graduate school is who their advisor will be. I have never doubted my choice. Without Manuela's unending enthusiasm and unfailing support, I would never have completed my PhD. Manuela's vision and creativity made everything seem worthwhile. I thank her for giving me the freedom to pursue my ideas and interests, while providing me with guidance through the tough parts.

I am indebted to everyone in the Xavier and PRODIGY research groups, especially Joseph, Sven, Rich, Greg and Jim. Robotics research can be very frustrating, and Joseph's cynical humour made it a lot more fun. I thank Greg in particular for his many hours fixing the robot and following it around to help me collect data.

I also thank Henry, Po, Arup, Darrell and the SCS facilities staff for keeping my machines running, particularly after a few too many late nights and stupid errors.

Outside school, I want to thank the Dinner Co-op, particularly Sonia and Sanjiv, and Barry and Ev. They kept me sane (or insane, depending on how you look at it), and distracted me from the agony. Lisa kept me healthy, Will and Wayne kept me amused. Rob has been a rock through all the turbulent waters, his love and unconditional support the foundation upon which I build my dreams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robots that aim at fully operating in the real world need to be able to perform many tasks autonomously. Reliability and efficiency are key issues. The robot must be able to effectively deal with noisy sensors and actuators and consistently achieve its tasks. It must create high-quality plans, and it must act efficiently, in real time, to deal with unexpected situations.

Moreover, since most physical world domains are hard to model completely and correctly, the robot should be able to learn from its experiences. The robot should be able to adapt to a changing environment. A learning robot will be more flexible and adaptive than a pre-programmed system. An office delivery robot, for example, would be able to move from working in a university classroom building to a hospital with few, if any, design changes. Since Shakey the robot [Nilsson, 1984], researchers have been trying to build autonomous robots that are capable of planning and executing high-level tasks, as well as learning from the analysis of execution experience.

This thesis addresses the concrete technical challenge of building a complete planning, executing and learning robotic agent operating in the real world. We present a robotic system which creates and executes plans for multiple, asynchronous, interacting tasks. We aim at showing that a real robot can learn from execution experience to improve planning and execution models, and therefore its performance.

The specific research foci are to investigate:

- real execution in a fully autonomous robot,
- challenging the agent with multiple interacting tasks,
- using planning and real execution as a source for learning.

One of the important scientific questions is to understand the interaction between an autonomous agent and its environment, especially when there are many interdependent tasks to be performed. The second important scientific question is to understand the impact of using past experience to improve planning performance in a challenging domain. Our learning is applicable in any physical domain where the costs or probabilities of actions are hard to capture or may change over time.

Figure 1.1: Xavier the robot.

1.1 Approach

In this dissertation we explore the interaction of perception, cognition, action and learning in a complete integrated autonomous agent.

We have built a system called Rogue [Haigh & Veloso, 1997; Haigh & Veloso, 1998a; Haigh & Veloso, 1998b] that forms the task planning and learning layers for a real mobile robot, Xavier[1]. One of the goals of the project is to have the robot move autonomously in an office building, reliably performing office tasks, such as picking up and delivering mail and computer printouts, picking up and returning library books, and carrying recycling cans to the appropriate containers.

Xavier is a mobile robot being developed at Carnegie Mellon University [O'Sullivan *et al.*, 1997; Simmons *et al.*, 1997] (see Figure 1.1). It is built on an RWI B24 base and includes bump sensors, a laser range finder, sonars, a color camera and a speech board. The software controlling Xavier includes both reactive and deliberative behaviours, integrated using the Task Control Architecture (TCA) [Simmons, 1994]. Much of the software can be classified into five layers, shown in Figure 1.2: Obstacle Avoidance, Navigation, Path Planning, Task Planning (provided by Rogue), and the User Interface.

Rogue provides a setup where users can post tasks for which the planner generates

[1]In keeping with the Xavier theme, Rogue is named after the "X-men" comic-book character who absorbs powers and experience from those around her. The connotation of a wandering beggar or vagrant is also appropriate.

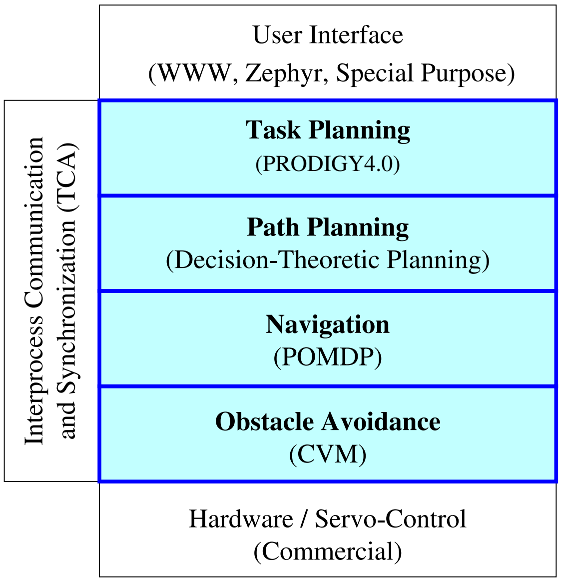


Figure 1.2: Xavier's primary software layers. Reproduced from Simmons *et al.* [1997].

appropriate plans, delivers them to the robot, monitors their execution, and learns from evaluation of execution performance.

ROGUE's task planner is based on the PRODIGY4.0 planning and learning system [Veloso *et al.*, 1995]. The challenges for a task planner in this domain are due to the asynchronous goals and the dynamics and uncertainty of the world. The task planner generates and executes plans for multiple interacting goals, which arrive asynchronously and whose structure is not known *a priori*. The task planner interleaves tasks, reasoning about task priority and task compatibility. ROGUE enables the communication between the planner and the robot, and controls the interleaved planning and execution process. ROGUE can detect execution failures, side-effects (including helpful ones), and opportunities. The task planner controls the execution of a real robot to accomplish tasks in the real world. The planning and execution capabilities of ROGUE form the foundation for a complete, learning, autonomous agent.

ROGUE uses a planner-independent learning approach that processes the execution data to improve planning. The challenges for a learning system in a physical world include (1) automatically extracting relevant learning information from the execution data, and (2) correlating that information with features of the domain to improve planning models. Our approach relies on examining the execution data to identify situations in which the planners' behaviour need to change. It then correlates features of the domain with the learning opportunities, and creates situation-dependent rules for each of the planners. The planners then use these rules to select between alternatives to create better plans. We demonstrate the generality of the approach in two different planners: Xavier's path planner, and the task planner. ROGUE learns *situation-dependent* rules that affect the planners' decisions.

ROGUE's overall architecture is shown in Figure 1.3. ROGUE exploits Xavier's reliable

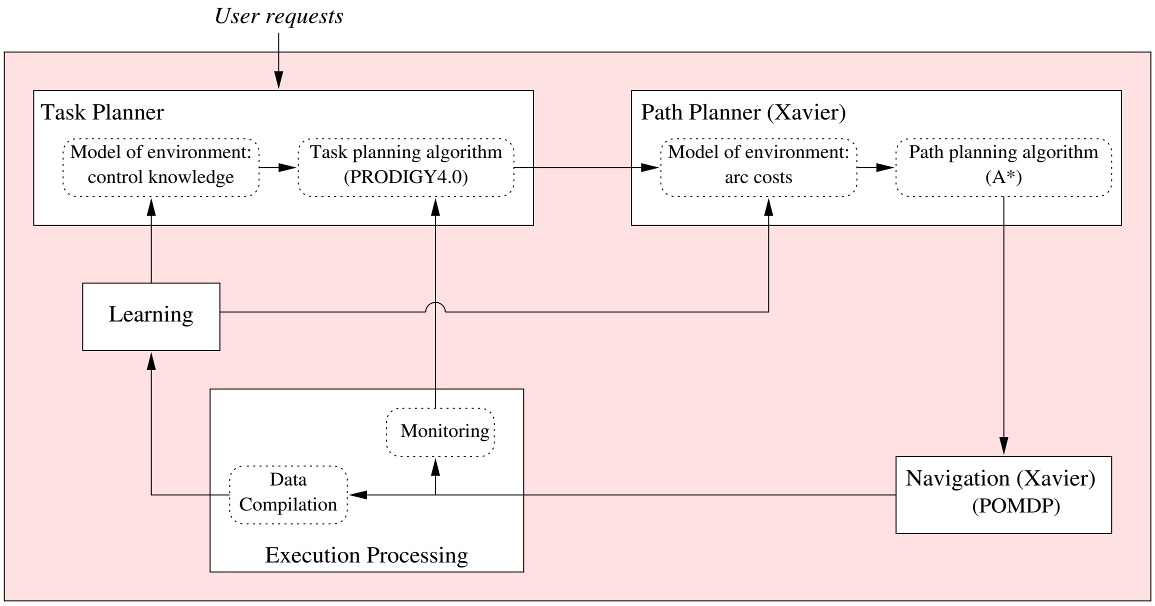


Figure 1.3: ROGUE architecture.

lower-level behaviours, including path planning, navigation, speech and vision. ROGUE provides Xavier with a high-level task planning component, and a learning component. The learning component extracts information from low-level execution data to improve high-level planning.

1.2 The Domain

The office delivery domain provides a reasonably rich and challenging environment for a robotic system, while remaining reasonably structured. Tasks include picking up printouts, picking up and returning library books, and delivering mail and packages within the building. User requests are, for example, *"Pickup a package from my office and take it to the mailroom before 4pm today."* In general, requests involve acquiring an item at some location, and then delivering it to another.

The domain requires a reliable, efficient, autonomous mobile robot. When the system is not reliable, tasks are not successfully achieved, and users will not utilize the system. When the system is not efficient, it misses deadlines and otherwise annoys users, and either users only request nonessential tasks, or the set of authorized users must be restricted.

Task requests arrive asynchronously, the locations and details of which are not known *a priori*. Plans for achieving tasks may interact; the task planner is responsible for finding an appropriate ordering to interleave and combine compatible tasks.

We can expect the environment to be dynamic at two levels. At the navigation level, temporary obstacles, including people and objects, may appear at any time. Permanent obstacles or changes may also occur; for example the hallways in our building were recently carpeted and several doors added. These changes may lead to changes in navigation efficiency, reliability or even achievability.

At the task planning level, temporary changes include, for example, people going to meetings, or changing work hours. More permanent changes might include new staff, people changing offices, or new task capabilities. For example, one goal of the system is to have the robot identify and collect aluminium cans for recycling.

Creating a pre-programmed model of these dynamics would be not only time-consuming, but very likely would not capture all relevant information. ROGUE can reduce the burden on the programmer because its learning capabilities modify the existing domain model to reflect real world experience. ROGUE extracts relevant information from the execution data to create situation-dependent rules to improve default cost or probability estimates. ROGUE learns patterns and identifies changes in the environment, creating situation-dependent rules that the planners can then use to improve plan quality.

1.2.1 Capabilities of the Robot

The software controlling Xavier includes both reactive and deliberative behaviours. The various software modules communicate with each other through the Task Control Architecture (TCA) [Simmons, 1994; Simmons *et al.*, 1990]. TCA provides facilities for scheduling and synchronizing tasks, resource allocation, environment monitoring and exception handling. The reactive behaviours enable the robot to handle real-time local navigation, obstacle avoidance, and emergency situations (such as detecting a bump). The deliberative behaviours include vision interpretation, maintenance of occupancy grids and topological maps, and path planning and global navigation (an A^* algorithm).

ROGUE exploits Xavier's reliable lower-level behaviours, including path planning, navigation, speech and vision. If Xavier were given other abilities, for example manipulation, elevator riding, or extended vision skills, they could be easily incorporated into ROGUE.

The path planner creates plans for moving from one location in the environment to another. The path planner uses a decision-theoretic A^* algorithm on a topological map with metric information [Goodwin, 1996]. The planner creates a plan with the best expected travel time, taking into account distance, blockage probability, traversal weight, and recovery costs (for, say, missing difficult turns).

ROGUE depends most heavily on Xavier's reliable navigation module, which reaches its destination approximately 95% of the time. Navigation is done using Partially Observable Markov Decision Process Models (POMDPs) [Simmons & Koenig, 1995]. In the period from December 1, 1995 to August 31, 1997 Xavier attempted 3245 navigation requests and reached its intended destination in 3060 cases, where on average each job required it to move 43 meters, for a total travel distance of over 125 kilometers. Detailed navigation results are presented elsewhere [Simmons *et al.*, 1997].

Xavier does *not* currently have the ability to manipulate objects itself. It therefore relies on humans in the environment to place or remove objects from its basket.

Xavier's vision system is minimally used by researchers in the group. Current abilities include face detection, door identification and door-label reading. The door identification skill is used by ROGUE only indirectly — the navigation module uses it to centre the robot in front of the door.

Xavier has a speech board that can convert ASCII English to recognizable accented speech. ROGUE uses this speech capability to interact with users, for example, asking for mail or verifying its location. To reply to ROGUE, users type on the keyboard.

In the near future, we expect that the robot will be able to autonomously ride the elevator (it currently does so with assistance), and thereby increasing the variety of tasks the system can perform.

1.3 Task Planning

The challenges for a task planner in this domain are due to the asynchronous goals and the dynamics and uncertainty of the world. ROGUE's task planner is based on PRODIGY4.0 [Veloso *et al.*, 1995], a domain-independent nonlinear state-space planner that uses means-ends analysis and backward chaining to reason about multiple goals and multiple alternative operators to achieve the goals. It has been extended to support real world execution of its symbolic actions [Haigh *et al.*, 1997b; Stone & Veloso, 1996]. ROGUE handles multiple asynchronous task requests and controls the real-world execution of Xavier to achieve tasks in this dynamic office delivery domain.

Any system operating in a dynamic world needs to be able to respond efficiently and effectively to changes in the environment. Actions may fail, actions may have unexpected side-effects (beneficial, irrelevant or harmful), and unexpected opportunities may arise. The system must have mechanisms to detect and respond to such failures and changes.

Asynchronous goals can have a serious effect on both planning and execution efficiency. The first important issue is that the system cannot delay execution until it has completed planning for all goals; it must instead *interleave planning and execution*.

Interleaving planning with execution not only allows the system to start executing tasks when requests arrive, but also allows the system to respond to changes in the environment as well as to reduce its planning effort. An interleaved framework provides the planner with feedback about execution, for example by pruning alternative outcomes of an action, or noticing opportunities. For example, the planner can notice limited resources such as battery power, or notice external events like doors opening and closing. The planner can remove planned actions when exogenous events or side-effects unexpectedly make them irrelevant.

ROGUE controls the task planner to interleave planning with execution. Each time PRODIGY4.0 generates an executable plan step, ROGUE maps the action into a sequence of navigation and other commands which are sent to the Xavier module designed to handle them. ROGUE then monitors the outcome of the action to determine its success or failure. ROGUE can detect execution failures, side-effects (including helpful ones), and opportunities.

Since the office delivery domain involves multiple users and multiple tasks, another important issue is that the task planner be able to interleave compatible tasks but not get so side-tracked that it gets nothing done. ROGUE provides PRODIGY4.0 with mechanisms to reason about task priority and task compatibility, and successfully and competently interleaves compatible tasks.

Because ROGUE interleaves planning with execution *and* handles asynchronous goals, ROGUE's ability to easily suspend and reactivate tasks is crucial. When important requests arrive, ROGUE suspends the execution of lower priority tasks. Once the important request has been fulfilled, ROGUE reactivates the less important task(s). Some systems respond to asynchronous goals by restarting the planner, losing planning effort as well as placing high demands on sensing to determine the current status of the environment and interrupted tasks [Pell *et al.*, 1997; Bonasso & Kortenkamp, 1996]. ROGUE, however, suspends and reactivates tasks efficiently, without losing any of the prior planning information. It monitors the environment to identify unexpected changes, such as side-effects and exogenous events, that can affect the validity and applicability of plans.

The office delivery domain involves multiple users and multiple tasks in a dynamic world. ROGUE interleaves planning and execution to create a task planner with the ability

- to integrate asynchronous requests,
- to prioritize goals,
- to suspend and reactivate tasks,
- to recognize compatible tasks and opportunistically achieve them,
- to execute actions in the real world, integrating new knowledge which may help planning, and
- to monitor and recover from failure.

ROGUE can control the execution of a real robot to accomplish tasks in the real world.

1.4 Learning

A complete autonomous agent must learn from its experiences. Most physical worlds are hard to model completely and correctly, and hence, regardless of the skill and thoughtfulness of its creator, the agent is bound to encounter situations that have not been specified in its design. The agent should adapt to these situations and absorb any information that will improve its performance. As completely embodied autonomous agents, robots generally deal with more complex environments than software or network agents do. The added modelling difficulty and greater dynamics of these environments make learning an even more critical component of a complete system.

The challenges for learning in a physical domain are primarily due to representation differences between the planners and the executors. It is hard to extract information from the execution data that will be relevant for planning, and hard to transform that data into useful planning knowledge. Moreover, it is hard to design a learning mechanism that will be flexible enough to acquire initial information about the environment, and then to modify it to incorporate future changes in the domain.

Prior Learning Efforts for Robotics. Learning has been applied to robotics problems in a variety of manners. Common applications include map learning and localization (e.g. [Koenig & Simmons, 1996; Kortenkamp & Weymouth, 1994; Thrun, 1996]), or learning operational parameters for better actuator control (e.g. [Baroglio *et al.*, 1996; Bennett & DeJong, 1996; Grant & Feng, 1989; Pomerleau, 1993]). Instead of improving low-level *actuator* control, our work focusses at the *planning* stages of the system.

Artificial intelligence researchers have explored this area extensively, but have generally limited their efforts to simulated worlds with no noise or exogenous events. AI research that most closely resembles ours has explored how to learn and correct action models (e.g. [Gil, 1992; Pearson, 1996; Wang, 1996]). These systems observe or experiment in the environment to correct action descriptions, which are then directly used for planning.

In the robotics community, closely related work comes from those who have explored learning costs and applicability of actions (e.g. [Lindner *et al.*, 1994; Shen, 1994; Tan, 1991]). These systems learn improved domain models and this knowledge is then used by the system's planner, as *costs* or *control knowledge*, so that the planner can then select more appropriate actions.

Situation-dependent Learning Approach. Current systems learn that each action has an associated *average* probability or cost. However, *actions may have different costs under different conditions.* Instead of learning a global description, *we would like the agent to learn the pattern by which these situations can be identified.* The agent needs to learn the correlation between features of the environment and the situations, so that its planners can predict and plan for those situations. Hence we introduce the concept of *situation-dependent rules* that determine costs or probabilities of actions.

We would like a *path planner* to learn, for example, that a particular highway is extremely congested during rush hour traffic. We would like a *network routing planner* to learn, for example, that packets are more easily lost at a particular router when the network is congested. We would like a *task planner* to learn, for example, that a particular secretary doesn't arrive before 10am, and tasks involving him can not be completed before then. We would like a *multi-agent planner* to learn, for example, that every Monday heavy packages arrive, requiring two agents to carry them. Once these patterns have been identified and correlated to features of the environment, the planner can then predict and plan for them when similar conditions occur in the future.

Learning consists of processing execution episodes situated in a particular task context, identifying successes and failures, and then interpreting this feedback into reusable knowledge. Our approach relies on examining the execution data to identify situations in which the planner's behaviour needs to change. Our approach requires that the execution agent defines the set of available situation *features*, $\mathcal{F}$, while the planner defines a set of relevant learning *events*, $\mathcal{E}$, and a *cost function*, $\mathcal{C}$, for evaluating those events.

Events are learning opportunities in the environment for which additional knowledge will cause the planner's behaviour to change. *Features* discriminate between those events, thereby creating the required additional knowledge. The *cost function* allows the learner to evaluate the event. We give some examples of events, costs and features in Table 1.1. The learner then creates a mapping from the execution features and the events to the costs:

$$\mathcal{F} \times \mathcal{E} \rightarrow \mathcal{C}.$$

For each event $\varepsilon \in \mathcal{E}$, in a given situation described by features $\mathcal{F}$, this learned mapping predicts a cost $c \in \mathcal{C}$ that is based on prior experience. We call this mapping a *situation-dependent rule.*

Once the rules have been created, the learner then gives the information back to the planners so that they will avoid re-encountering the problem events. When the current situation matches the features of a given rule, the planners will avoid (or exploit) the corresponding event when appropriate.

These steps are summarized in Table 1.2. Learning occurs incrementally and off-line; each time a plan is executed, new data is collected and added to previous data, and then all data is used for creating a new set of situation-dependent rules.

In this incremental way, the planners can not only detect patterns in the environment, but also notice when the environment *changes.* For example, the bottleneck router may be replaced by new hardware so that it can handle more packets. The secretary may change his work hours. The incremental learner can notice these changes and incorporate them into the rules, thereby responding to the changing environment.

The approach is relevant for all planners that would benefit from feedback about plan execution. Every planner can benefit from understanding the patterns of the environment

	$\mathcal{E}$	$\mathcal{F}$	$\mathcal{C}$
Path Planner *A highway is congested during rush hour.*	driving a highway	time-of-day day-of-week	traversal time gas consumption
Network Router *Packets are lost at a particular router when the network is congested.*	routing packets	traffic volume router	packet loss rate throughput time-to-destination
Task Planner *A particular secretary doesn't arrive until 10am.*	achieving tasks	location secretary time-of-day	success rate
Multi-Agent Planner *Heavy packages arrive on Mondays, requiring two agents.*	achieving tasks	number of agents package weight day-of-week	success rate time-to-completion

Table 1.1: Examples of Events, $\mathcal{E}$, Features, $\mathcal{F}$, and Costs, $\mathcal{C}$, for sample planners.

1. Create plan.
2. Execute; record the execution data and features $\mathcal{F}$.
3. Identify events $\mathcal{E}$ in the execution data.
4. Learn mapping: $\mathcal{F} \times \mathcal{E} \rightarrow \mathcal{C}$.
5. Create rules to update each planner.

Table 1.2: General approach for learning situation-dependent costs.

that affect task achievability. This situation-dependent knowledge can be incorporated into the planning effort so that tasks can be achieved with greater reliability and efficiency. Situation-dependent features are an effective way to capture the changing nature of a real-world environment.

The approach is also relevant for planners and executors whose data representations differ widely. Features are defined as by the executor and the task environment, while events and costs are defined by the planner. These are mapped into an intermediate data representation that is independent of both the executor and the planner. As a result, planners can be designed independently from their hardware, thereby allowing designers to select the best planner for a given task.

To demonstrate the effectiveness of the approach, we have implemented it in two planners: Xavier's path planner, and the task planner. Rogue processes execution data to create improved domain models for both of its planners, thereby allowing them to create better quality, more efficient plans. Rogue incorporates the situation-dependent learning approach to equip a real robot with the ability to learn from its own execution experiences.

1.4.1 Learning for the Path Planner

Knowledge in the path planner is represented as a topological map of the robot's navigation environment. The map is a graph with nodes and arcs representing office rooms, corridors, doors and lobbies, and is augmented with metric information. The path planner uses an estimate of the arcs' traversal costs to create path plans with the best expected travel time. By learning appropriate arc-cost functions, Rogue helps the path planner to avoid troublesome areas of the environment when appropriate. Therefore events, $\mathcal{E}$, for this planner are arc traversals; features, $\mathcal{F}$, include robot sensor data and high-level information such as date and desired route; and costs, $\mathcal{C}$, are travel time and position confidence.

Consider the following example. For Xavier, the most challenging region of its environment is the lobby of our building. Figure 1.4 shows the map of the main floor, and Figure 1.5 shows a closeup of the lobby area, with typical obstacles added for the reader's benefit (since they often change, the robot does not know where they are). The lobby contains two food carts, several tables, and is often full of people. The tables and chairs are extremely difficult for the robot's sonars to detect, and the people are (often malicious) moving obstacles. As a result, navigating through the lobby is challenging and expensive for the robot. During

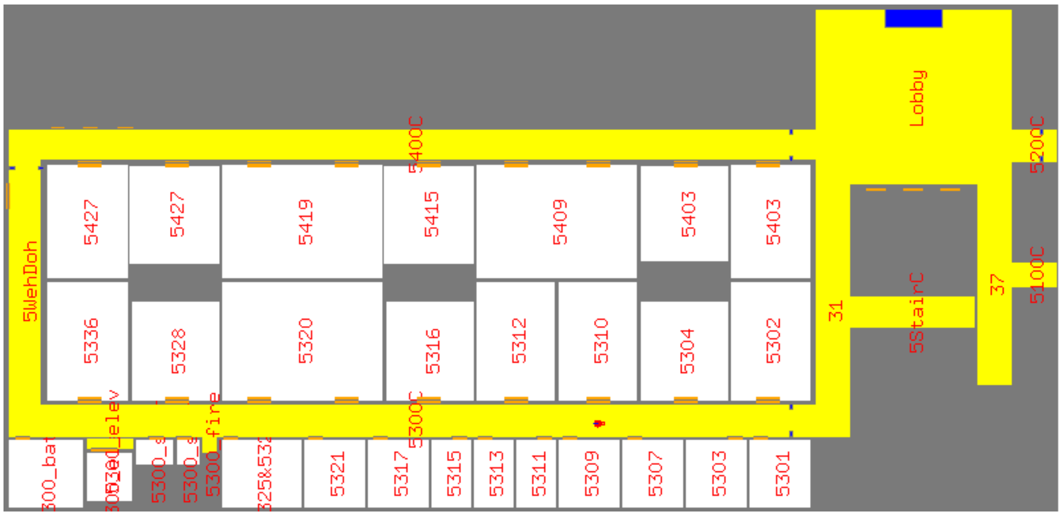


Figure 1.4: Robot's map (half of the 5th floor of our building).

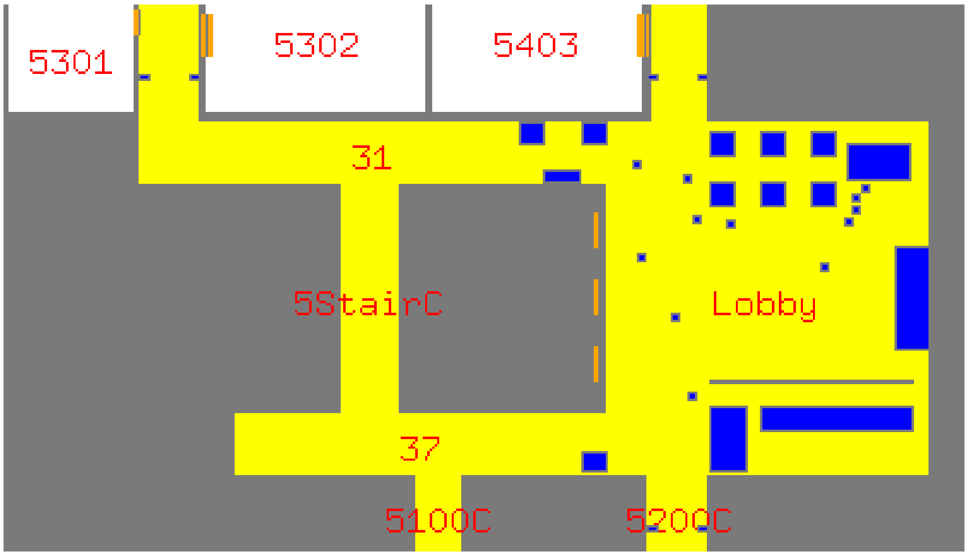


Figure 1.5: Closeup of map; typical obstacles added for the reader: small obstacles indicate people, while larger ones indicate tables and food carts.

peak hours (coffee and lunch breaks), it is virtually impossible for the robot to efficiently navigate through the lobby.

In this example, we would like Xavier to learn *when* to avoid the lobby *completely*. A direct path from the 5200 corridor to room 5409 is very short through the lobby, but when the lobby is crowded, the robot takes a lot of time to arrive at its destination. When the lobby is empty, the robot rarely has problems. A rule modifying the cost of the arc, such as the one shown in Figure 1.6, would force the planner to avoid the lobby during lunch break.

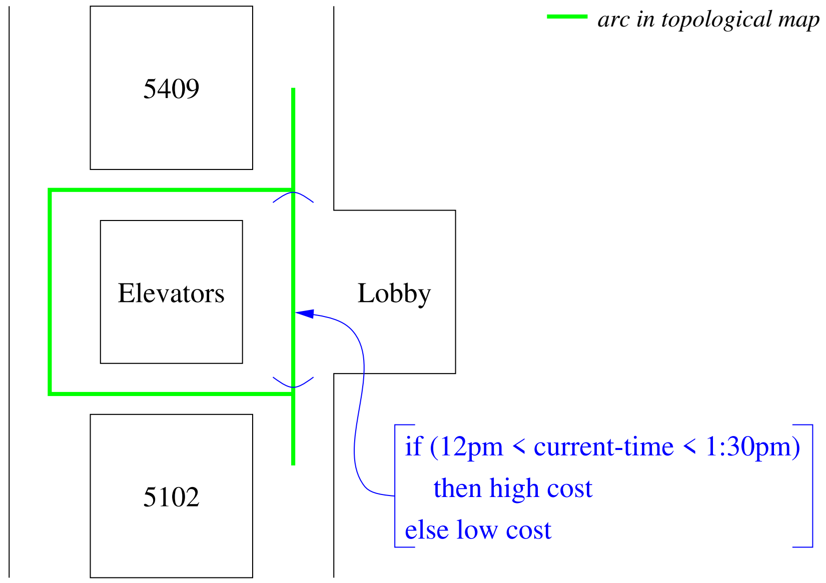


Figure 1.6: A high-level view of a sample learned rule for the path planner; ROGUE learns actual traversal costs.

1.4.2 Learning for the Task Planner

Knowledge at the task planner level is represented and manipulated as symbolic information. User requests are, for example, *"deliver mail to the main office."* By learning rules that govern the applicability of actions and tasks, ROGUE helps the task planner select, reject or delay tasks in the appropriate situations. Events, $\mathcal{E}$, useful for learning include missed deadlines and time-outs (e.g. waiting at doors); features, $\mathcal{F}$, include robot sensor data and high-level information such as date and other tasks; while costs, $\mathcal{C}$, can be defined by task importance, effort expended (travel plus wait time), and how much a deadline was missed by.

For ROGUE, an important aspect of achieving its tasks involves interacting with users. For example, ROGUE needs to request that a person place or remove the desired object in its carrying basket. If ROGUE has to wait for substantially long times before acquiring or delivering objects, ROGUE's efficiency is severely compromised.

In this example, we would like ROGUE to learn when people will be away from their offices, and then to avoid the task during those times. For example, a particular user might work 11am to 8pm, while another works 8am to 5pm. Rules guiding the task planner, like the ones shown in Table 1.7, would help the task planner avoid tasks at appropriate times.

1.5 Contributions

This dissertation presents the full implementation of an integrated planning, executing and learning robot system.

```
if (or (current-time < 11am)
       (current-time > 8pm))
then reject goals involving room1
```

```
if (or (current-time < 8am)
       (current-time > 5pm))
then reject goals involving room2
```

Figure 1.7: A high-level view of two sample rules learned for the task planner.

Before the addition of ROGUE to Xavier's architecture, Xavier reliably performed actions requested of it, but had no task planning or learning abilities. PRODIGY4.0, meanwhile, is a complex task planner that had never been used interleaved with execution in the real world; as such, it had never been used for asynchronous goals or in an environment where the state spontaneously changes. In combining PRODIGY4.0 and Xavier, the challenges for ROGUE included developing a communication mechanism for control and feedback, as well as extending the planner to handle the dynamics of a real-world task.

In extending ROGUE with learning capabilities, we have increased the flexibility and efficiency of the system because it can adapt to its current environment and also respond to changes in the environment. The challenges included developing techniques to overcome the representation differences between the execution module and the planning modules, as well as handling the massive, continual, probabilistic execution traces from this noisy domain.

The specific contributions of this thesis include:

- **Task Planner:**
  - ◇ The transparent incorporation of asynchronous goals into planning.
  - ◇ The ability to create plans for multiple interacting goals, taking into account task priority and compatibility.
  - ◇ The ability to suspend and reactivate tasks when necessary.
  - ◇ The ability to detect and respond to failures, unexpected side-effects of actions, and changes in the environment.
  - ◇ The development of an interleaved planning and real robot execution procedure, including the development of a communication mechanism between the planner and the executor.

- **Learning:**
  - ◇ The improvement of plans through examination of real-world execution data.
  - ◇ The introduction of *situation-dependent rules* which set action costs or probabilities at planning time as a function of situational features.
  - ◇ The design of a general framework for learning across representations, in which execution data representation differs widely from planning representations.
  - ◇ The implementation and proof-of-concept of the planner-independent approach for two different planners, along with extensive empirical results.
  - ◇ The demonstration of system adaptability to a changing domain.

Additional technical contributions are described in Chapter 6.

1.6 Reader's Guide

In Figure 1.8 we show which sections need to be read for full comprehension of each of the main contributions of this thesis.

Task Planner: In Chapter 2, we present the task planner. We describe how ROGUE handles multiple asynchronous goals to create plans that the robot executes. We describe ROGUE's mechanisms for determining task priority and compatibility, and for suspending and interrupting tasks. We present ROGUE's interleaved planning and execution paradigm, including the mechanisms ROGUE uses to monitor execution.

Situation-Dependent Learning: For an understanding of the general learning framework only, we suggest reading the following Sections:

- Overview: 1.4
- Features: 3.2 and 4.1
- Events: the first paragraphs of 3.3 and all of 4.2
- Costs: 3.4 and 4.3
- Learning: 3.5

Learning for the Path Planner: In Chapter 3, we instantiate the general learning framework for Xavier's path planner. We describe the mechanisms used to identify features,

Task Planning	**Learning for the Path Planner**	**Learning for the Task Planner**
Chapter 1	Chapter 1	Chapter 1
Chapter 2	**Chapter 3**	Section 2.2.2
(Section 5.1)	(Section 5.2)	Section 3.2
		(Section 3.4)
		Section 3.5
		Chapter 4
		(Section 5.2)

Figure 1.8: Reader's guide. For each of the three topics of this thesis, relevant sections are listed. Bold face indicates that the primary topic of the chapter matches that of the heading; parentheses indicate less critical sections.

events and costs and then present the learning algorithm. We present detailed empirical results showing the effectiveness of the system.

Learning for the Task Planner: In Chapter 4, we present our learning framework in a prototypical instantiation for the task planner. We present two manners by which our learning approach can be used for this planner: to improve planning performance, and to improve execution performance. We present the techniques used to identify learning events for this planner, and describe how events are evaluated. We present sample empirical results showing the applicability of the approach to this planner.

One of the contributions of the thesis is our *planner-independent* learning approach, therefore the structure of the chapter parallels that of Chapter 3. In Chapter 4, we emphasize the differences between the two implementations, and *do not* repeat overlapping technical content; cross references are provided where appropriate. Note that for full comprehension of this chapter, we suggest reading Sections 2.2.2 (planning), 3.2 (features), 3.4 (costs) and 3.5 (learning) beforehand.

In Chapter 5 we describe related work. In Chapter 6 we present our conclusions. We describe some areas of future work, and provide an analysis of the general applicability of the approach.

Chapter 2

The Task Planner

In this chapter, we focus on presenting the techniques underlying the planning and execution control in ROGUE. The planning and execution capabilities of ROGUE form the foundation for a complete, learning, autonomous agent.

ROGUE generates and executes plans for multiple interacting goals which arrive asynchronously and whose task requirements are not known *a priori*. ROGUE interleaves tasks and reasons about task priority and task compatibility. ROGUE enables the communication between the planner and the robot, allowing the system to successfully interleave planning and execution to detect successes or failures and to respond to them. ROGUE controls the execution of a real robot to accomplish tasks in the real world.

In Section 2.1, we present the ROGUE's planning and executing architecture. In Section 2.2, we describe PRODIGY4.0, describe how it plans for multiple asynchronous goals, and introduce ROGUE's mechanism for handling task priority and compatibility. We include a detailed example of the system's behaviour for a simple two-goal problem, when the goals arrive asynchronously. In Section 2.3, we present execution and monitoring, in particular how the system detects, processes and responds to failure. Finally we provide a summary of ROGUE's capabilities in Section 2.5. Related work can be found in Section 5.1.

2.1 Planning and Execution Architecture

ROGUE accepts tasks posted by users, calls the task planner, PRODIGY4.0, which generates appropriate plans, and posts actions to the robot, Xavier, for execution. Figure 2.1 shows the general architecture of the planning and execution part of ROGUE's system.

ROGUE interfaces with Xavier through the Task Control Architecture (TCA) [Simmons, 1994]. TCA provides the communication network between each of the processes controlling the robot's behaviour, as well as facilities for scheduling and synchronizing tasks, resource allocation, environment monitoring and exception handling. These processes include both reactive behaviours and deliberative behaviours. Reactive behaviours include local navigation, obstacle avoidance, and emergency situations (such as detecting a bump). Deliberative

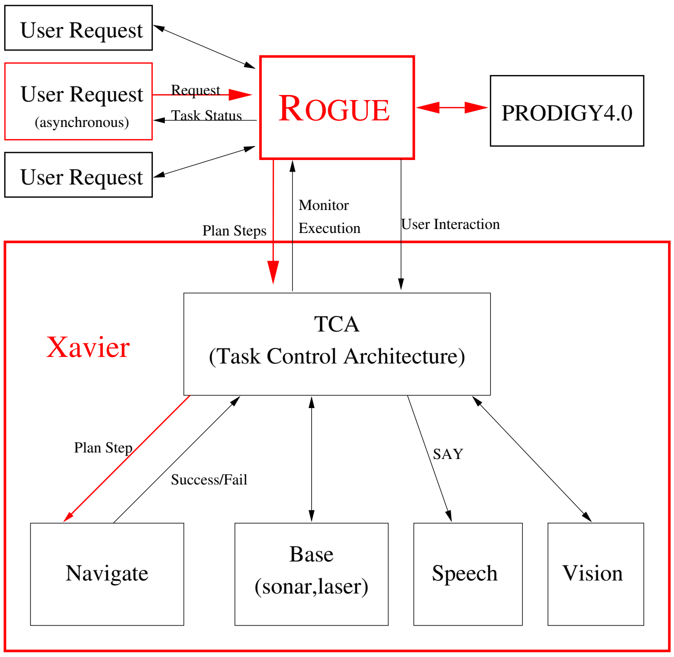


Figure 2.1: ROGUE task planning architecture.

behaviours include vision, occupancy grids and topological maps, and path planning and global navigation.

PRODIGY is a domain-independent planner that serves as a testbed for machine learning research [Carbonell *et al.*, 1990; Veloso *et al.*, 1995]. The current implementation, PRODIGY4.0, is a nonlinear planner that follows a state-space search guided by means-ends analysis and backward chaining. It reasons about multiple goals and multiple alternative operators to achieve the goals. It reasons about interacting goals, exploiting common subgoals and addressing issues of resource contention. ROGUE provides appropriate search control knowledge to the planner and monitors the outcome of execution.

There are several approaches for creating plans that can be executed. We take the approach of *interleaving planning with execution.* Interleaving planning with execution can create opportunities for the system as well as reduce the search space by removing alternative outcomes of actions.

Two features inherent in PRODIGY4.0 are key to allowing an interleaved planning and execution paradigm:

- PRODIGY4.0 is capable of generating partial plans for execution in a continuous way, and
- PRODIGY4.0 continuously re-evaluates the goals-to-be-achieved based on its current state information.

The first feature allows ROGUE to interrupt the planning cycle and send actions for execution. The second features allows ROGUE to incorporate sensor information from the real world so that PRODIGY4.0 can respond to changes in the environment.

ROGUE's interleaving of planning and execution can be outlined in the following procedure for accomplishing a set of tasks:

1. Each time a user submits a request, and ROGUE adds the task information to PRODIGY4.0's state.
2. PRODIGY4.0 creates a plan to achieve all current and new goals, constrained by ROGUE's priority and compatibility knowledge, taking into account any interactions between the goals.
   - As each action is selected for execution, ROGUE sends it to the robot for execution, first confirming that its preconditions are valid, and suspending planning during execution.
   - ROGUE confirms the outcome of each action. ROGUE incorporates any new knowledge into PRODIGY4.0's state. In particular, if the action fails, ROGUE notifies PRODIGY4.0 and forces replanning.
3. Continuously throughout planning, ROGUE monitors the environment for changes that may affect decisions, and updates PRODIGY4.0's state accordingly.

It is important to realize that PRODIGY4.0 does not continue planning while the robot is executing an action. ROGUE sends only one action at a time to TCA, and PRODIGY4.0 waits until the action has completed (Section 2.3 describes how actions are selected for execution). Requests, however, may enter the system while executing; ROGUE adds them to PRODIGY4.0's state description, but PRODIGY4.0 does not plan for them until planning resumes.

In this chapter, we introduce each of these steps in detail. ROGUE's scientific contribution includes the development of this procedure and using it with a real planner on a real executor for real user requests. Given that it currently cannot ride the elevator autonomously, it is very limited in the actual tasks that it can do. ROGUE has made actual deliveries for several users, but is not currently in general use in the department. ROGUE has been thoroughly tested in the simulator, and when Xavier is given the ability to ride elevators we fully expect an easy transition to the more complex environment.

2.2 Planning for Asynchronous Requests

The office delivery domain involves multiple users and multiple tasks. A planner functioning in this domain needs to respond efficiently to task requests, as they arrive asynchronously. One common method for handling these multiple goal requests is simply to process them in a first-come-first-served manner; however, this method leads to inefficiencies and lost opportunities for combined execution of compatible tasks [Goodwin & Simmons, 1992].

ROGUE is able to process incoming asynchronous goal requests, to prioritize them, and to suspend lower priority actions when necessary. It successfully interleaves compatible requests and creates efficient plans for completing all the tasks.

2.2.1 Receiving a Request

User requests are standard office delivery tasks. For example, a user might make the request: *"Pickup a package from my office and take it to the mailroom before 4pm today."* Important information includes the user, the item, the pickup and delivery locations, and the deadline. Users submit their task requests through one of three different interfaces: the World Wide Web [Simmons *et al.*, 1997], Zephyr [DellaFera *et al.*, 1988; Simmons *et al.*, 1997], or a specially designed graphical user interface (Figure 2.2) [Haigh & Veloso, 1996].

The slots in this last interface are automatically filled in with default information related to the task (e.g. FedEx delivery location) as well as information extracted from the user's plan file through a simple template-matching mechanism. The deadline time defaults to one hour in the future. The interface can be extended with additional tasks at any time.

The user interface forwards the request to ROGUE by TCA messages. Table 2.1 shows the data structures used by the user interface and the PRODIGY4.0 planner, along with an example request. Appendix B shows sample code used to generate multiple tasks; it shows the use of the data structures, and the TCA command used to create the request.

PRODIGY4.0 connects to TCA with the command sequence shown in Table 2.2. The first command sets up a PRODIGY4.0 interrupt, `(tcaProdigyCheckMessage)`, to check for new requests. A PRODIGY4.0 interrupt is a function that is called once during each decision cycle of the planner. It then connects to TCA, registers the request handler, and finally calls `(tcaProdigyListen)`, which is the top-level function that starts the planning cycle when the first request arrives. Appendix A shows the full code of this initialization sequence.

When each new request comes in, either in the interrupt or in `(tcaProdigyListen)`, ROGUE adds it to PRODIGY4.0's list of unsolved goals, and updates the task model, as shown in Table 2.3. The literal `(needs-item <user> <item>)` indicates that a request, sent by user `<user>`, is pending. G is PRODIGY4.0's list of *top-level goals*, the list of goals

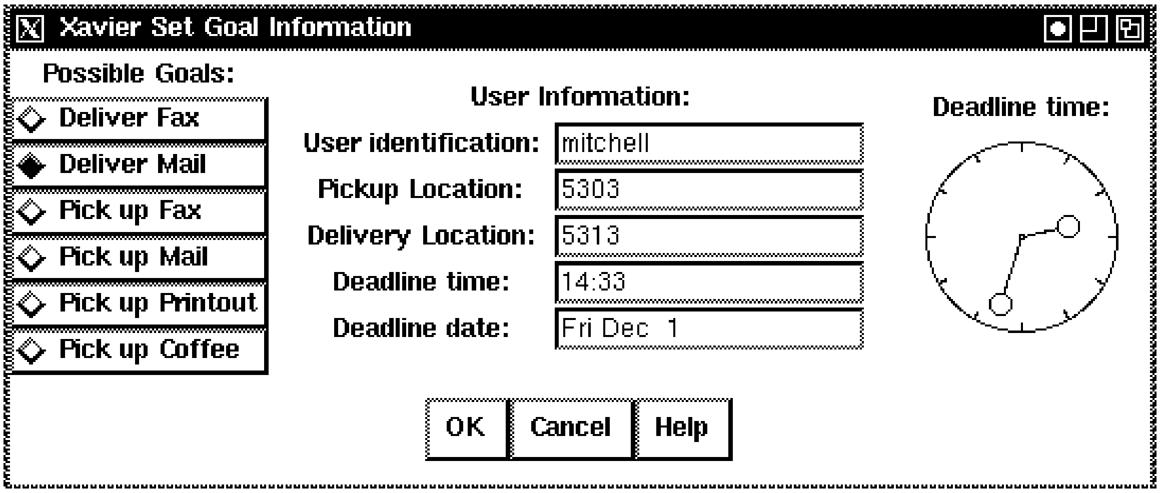


Figure 2.2: User request interface.

```
(tca::defstruct_tca (tcaRequest)
  (userid "" :type  string)
  (rank 0 :type int)
  (task "" :type string)
  (task-rank 0 : type int)
  (why "" :type string)
  (when-request "" :type string)
  (when-deadline "" :type string)
  (where-pickup "" :type string)
  (where-deliver "" :type string)
)
```

(a) Lisp.

```
mitchell
3
delivermail
2
""
Fri Dec 01 13:33
Dec 01 14:33
r-5303
r-5313
```

(b) Example.

```
struct  {
  char *userid;
  int rank;
  char *task;
  int taskrank;
  char *why;
  char *whenrequest;
  char *whendeadline;
  char *wherepickup;
  char *wheredeliver;
} prodigy_struct_ptrs;
```

(c) C.

Table 2.1: Request data structure for TCA, as defined for the C user interface and the Lisp planner.

```
;; install PRODIGY interrupt handler to check the socket
(define-prod-handler :always #'tcaProdigyCheckMessage)

;; register Request handler and connect to TCA
(tca::tcaConnectModule "Prodigy" (tca::tcaServerMachine))
(tca::tcaRegisterCommandMessage "Prodigy_PlanRequestCommand"
             "{string,int,string,int,string,string,string,string,string}")
(tca::tcaRegisterHandler "Prodigy_PlanRequestCommand"
                         "PlanRequestHandler" 'PlanRequestHandler)
(tca::tcaEnableDistributedResponses)
(tca::tcaWaitUntilReady)

;; wait for initial request to arrive
(tcaProdigyListen)
```

Table 2.2: Registering the request handler and connecting to TCA.

which need to be satisfied in the state before PRODIGY4.0 declares the planning cycle to be complete; the literal `(has-item <user> <item>)` becomes satisfied when the request is completed[1]. The function shown in Table 2.3 is domain-dependent because the literals added relate strictly to this domain; however, the structure would be identical for any other domain with asynchronous tasks.

It should be noted that new goals may arrive during execution, while PRODIGY4.0's planning cycle is suspended. PRODIGY4.0 will incorporate the new goals into the plan at its next decision point. The example in Section 2.2.4 illustrates ROGUE's behaviour when a

[1]Semantically, `(has-item <user> <item>)` might seem strange for a delivery task, or if a third person made the request. However, the meaning of the symbol is irrelevant to the computer; humans should consider it equivalent to "task for user `<user>` involving item `<item>` is complete, irrespective of who the actual recipient is or where the item is located.

Define: $C \leftarrow$ current state
Define: $G \leftarrow$ top-level goals

Let R be the list of pending unprocessed requests
For each $request \in R$, turn $request$ to goal:
- $C \Leftarrow C \cup \{$ `(needs-item` $request$`-userid` $request$`-object)`
  `(pickup-loc` $request$`-userid` $request$`-pickup-loc)`
  `(deliver-loc` $request$`-userid` $request$`-deliver-loc)`
  `(deadline` $request$`-userid` $request$`-when-deadline)` $\}$
- $G \Leftarrow$ `(and` G `(has-item` $request$`-userid` $request$`-object))`
- $request$-completed $\Leftarrow$ nil

Table 2.3: Integrating new task requests into PRODIGY4.0.

new goal arrives during execution.

There is currently no explicit mechanism for a user to rescind a request; however PRODIGY4.0 will no longer plan for (or attempt to apply operators for) the associated top-level goal if it is simply removed from G. Implementation details, such as reversing partially executed plans when necessary, are left for future work.

2.2.2 Planning in PRODIGY4.0

PRODIGY4.0 creates a plan for its unsolved goals by selecting operators whose effects achieve those goals. It continues adding operators to the incomplete plan until a solution to the problem is found.

Planning involves specifying a task model including operators and search control rules. Below, we describe the operator representation, and then present the planning algorithm, and finally describe how control rules are used to guide the planner's decisions.

2.2.2.1 Operators

A PRODIGY4.0 operator is defined by its *preconditions* and *effects*, described by *literals* that may contain variables. Variables may be typed, or may be constrained by arbitrary functions. Preconditions in the operators can contain conjunctions, disjunctions, negations, and both existential and universal quantifiers. Effects may be conditional. Variables may also have delayed bindings, where the value is not selected until the operator is applied.

The operators in ROGUE's task planning domain rely heavily on Xavier's existing behaviours, including path planning, navigation, vision and speech. ROGUE does not reason, for example, about which path the robot takes to reach a goal, or about obstacles in its way. By abstracting each request to the robot, such as which path the robot takes, ROGUE can more fully address issues arising from multiple interacting tasks, such as efficiency, resource contention, and reliability.

Table 2.4 shows the primary operators used in this domain. In Table 2.4d, for example, the robot cannot deliver a particular item unless it (i) has the item in question, and (ii) is in the correct location. In Table 2.4a, we show how an operator represents that the robot will not go to a pickup location unless it needs to pickup an item there. It does not matter where the robot's current location is; the variable `<current-location>` is only instantiated when the operator is applied, namely when ROGUE knows where the robot is.

The representation of the operators, for example (GOTO-PICKUP-LOC) and (GOTO-DELIVER-LOC), is not intrinsic to the task, but it can be relevant to planning efficiency. We have an implementation of the domain with a single (GOTO-LOC) operator with less constrained preconditions, which leads to more backtracking while the planner selects the correct order of desired locations. We can also create a search control rule to guide the planning choices (see below for a description); this is logically equivalent to separating the operators, but with some additional match cost.

```
(operator GOTO-PICKUP-LOC
  (params <user> <new-room>)
  (preconds ((<user> PERSON)
             (<item> ITEM)
             (<newloc> ROOM))
    (and (needs-item <user> <item>)
         (not (robot-has-item <user> <item>))
         (pickup-loc <user> <new-room>)))
  (effects ((<current-location> ROOM))
    ((del (robot-in-room <current-location>))
     (add (robot-in-room <new-room>)))))
```

(a) Goto pickup location.

```
(operator GOTO-DELIVER-LOC
  (params <user> <new-room>)
  (preconds ((<user> PERSON)
             (<item> ITEM)
             (<new-room> ROOM))
    (and (needs-item <user> <item>)
         (robot-has-item <user> <item>)
         (deliver-loc <user> <new-room>)))
  (effects ((<current-location> ROOM))
    ((del (robot-in-room <current-location>))
     (add (robot-in-room <new-room>)))))
```

(b) Goto deliver location.

```
(operator ACQUIRE-ITEM
  (params <room> <user> <item>)
  (preconds ((<user>PERSON)
             (<item> ITEM)
             (<room> ROOM))
    (and (needs-item <user> <item>)
         (not (robot-has-item <user> <item>))
         (pickup-loc <user> <room>)
         (robot-in-room <room>)))
  (effects ()
    ((add (robot-has-item <user> <item>)))))
```

(c) Acquire Item.

```
(operator DELIVER-ITEM)
  (params <room> <who> <item>)
  (preconds ((<who> PERSON)
             (<item> ITEM)
             (<room> ROOM))
    (and (needs-item <user> <item>)
         (robot-has-item <user> <item>)
         (deliver-loc <user> <room>)
         (robot-in-room <room>)))
  (effects ()
    ((add (has-item <user> <item>))
     (del (needs-item <user> <item>))
     (del (robot-has-item <user> <item>)))))
```

(d) Deliver Item.

Table 2.4: The primary operators in ROGUE's task planning domain.

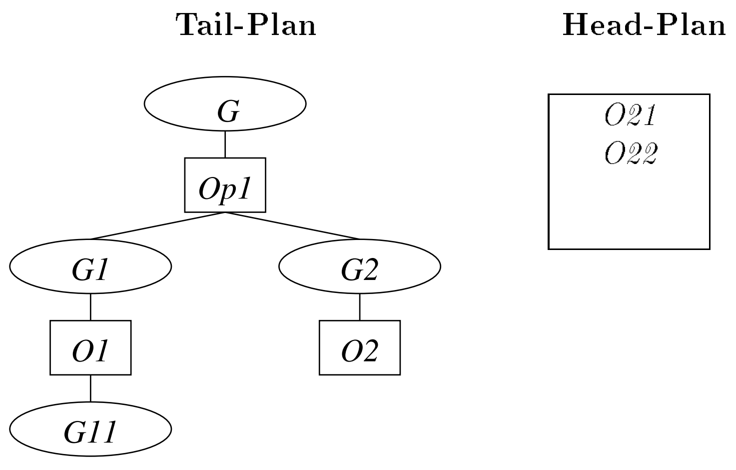
2.2.2.2 Building the Plan

PRODIGY4.0 creates a plan for its unsolved goals by selecting operators whose effects achieve those goals. It continues adding operators to the incomplete plan until a solution to the problem is found. In Figure 2.3 we show a simple incomplete plan. An incomplete plan consists of two parts, the *head-plan* and the *tail-plan* [Fink & Veloso, 1994].

The tail-plan is built by a backward-chaining algorithm, which starts from the list of goals, G, and adds operators, one by one, to achieve its *pending goals*, i.e., to achieve preconditions of other operators that are not satisfied in the current state. Adding operators to the tail-plan is known as *subgoaling*.

When all the preconditions of a given operator are satisfied in the current state, PRODIGY4.0 can simulate the effects of the action by *applying* the operator, or moving an operator from the tail-plan to the head-plan. The head-plan is a *valid total-order plan*, that is, a sequence of operators that can be executed in the initial state.

Each time an operator is applied, the current state is updated with the effects of the action, effectively *simulating* the effects of the action. PRODIGY4.0 terminates planning when each of the goals in G are satisfied in the current *simulated* state. In ROGUE, we use this simulation step to to actually *execute* the action, and maintain the simulated state as closely as possible to the actual state; we describe this process in Section 2.3.2, along with other possible methods for deciding when to execute.



Op1	pre: G1, G2 add: G del: —	*O1*	pre: G11 add: G1 del: —	*O2*	pre: G21, G22 add: G22 del: —	*O21*	pre: — add: G2 del: —	*O22*	pre: — add: G21 del: —

Figure 2.3: Example representation of an incomplete plan in PRODIGY4.0. G is the top-level goal, and $Op1$ is the operator that achieves it. $G1$ and $G2$ are two preconditions of $Op1$ that are not satisfied in the current state, and are achieved by $O1$ and $O2$ respectively. Lines can be viewed as causal links.

The planning cycle involves several decision points, including

- whether to *apply* or to *subgoal*,
- *which goal* to select from the set of pending goals, and
- *which applicable operator* to apply.

Table 2.5 shows the PRODIGY4.0 planning algorithm, with its main decision consisting of whether to subgoal or apply an operator. **Back-Chainer** shows the subgoaling decisions made while back-chaining on the plan, and **Operator-Application** shows how an operator is applied.

ROGUE runs under PRODIGY4.0's *SABA* mode (Subgoal Always Before Apply) [Stone *et al.*, 1994]. SABA delays operator application until all subgoals have been expanded. Essentially, this behaviour is equivalent to planning as far in advance as possible, but note that the plan may not be complete, since parts of the plan may depend on having applied other operators.

In ROGUE's office delivery domain, PRODIGY4.0 takes the top level goal, `(has-item <user> <item>)`, and selects an operator that will achieve it. It continues building the plan

PRODIGY4.0
1. If the goal statement G is satisfied in the current state, terminate.
2. Either (A) Subgoal: add an operator to *Tail-Plan* (**Back-Chainer**), or
   (B) Apply: move an operator from *Tail-Plan* to *Head-Plan* (**Operator-Application**).
   Decision point: Decide whether to apply or to subgoal.
3. Recursively call **PRODIGY4.0** on the resulting plan.

Back-Chainer
1. Pick an unachieved goal or precondition g.
   Decision point: Choose an unachieved goal.
2. Pick an operator *op* that achieves g.
   Decision point: Choose an operator that achieves this goal.
3. Add *op* to *Tail-Plan*.
4. Instantiate the free variables of *op*.
   Decision point: Choose an instantiation for the variables of the operator.

Operator-Application
1. Pick an operator *op* in *Tail-Plan* which is an *applicable operator*, that is the preconditions of *op* are satisfied in the current state.
   Decision point: Choose an operator to apply.
2. Move *op* from *Tail-Plan* to *Head-Plan*.
3. Update the current state with the effects of *op*.

Table 2.5: PRODIGY4.0 algorithm and decision points, adapted from Veloso *et al.* [1995].

recursively, adding operators for each precondition that is not satisfied in the state, until all of the operators in the leaf nodes have no unsatisfied preconditions, yielding a network of plan steps and goals such as the one shown in Figure 2.4.

The variable <`current-location`> in Table 2.4 is known as a *delayed binding*. PRODIGY4.0 binds all free variables in step 4 of **Back-Chainer** (Table 2.5), *when they appear in the list of preconditions*. By placing variables in the effects list of the operator, ROGUE forces PRODIGY4.0 to delay binding them until the operator is applied, thereby effectively reducing backtracking effort.

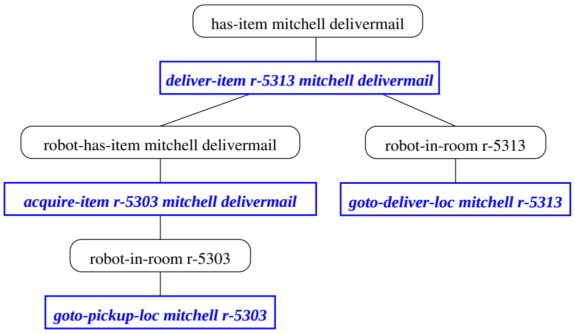


Figure 2.4: Plan for single task problem. Goal nodes are shown in ovals, selected operators are shown in rectangles.

2.2.2.3 Search Control Rules

PRODIGY4.0 provides a method for creating *search control rules* that reduces the number of choices at each decision point in Table 2.5 by pruning the search space or suggesting a course of action while expanding the plan.

Control rules are *if-then* rules that indicate which choices should be made (or avoided) depending on the current state and other meta-level information. In particular, control rules can *select*, *prefer* or *reject* specific planning choices at every decision point [Carbonell *et al.*, 1992]. Control rules can be used to focus planning on particular goals and towards desirable plans. ROGUE primarily uses two types of control rules: those that control *goal* decisions, and those that control *applicable operator* decisions.

In Chapter 4, we describe mechanisms to learn control rules that aid the planner in making decisions that reflect actual experiences encountered in the real world.

Goal Selection Rules: Each time PRODIGY4.0 examines the set of unsolved pending goals, it fires its *goal selection* search control rules to decide which goal to expand. ROGUE interacts with PRODIGY4.0 by providing the set of control rules used to constrain PRODIGY4.0's decisions. Table 2.6 shows ROGUE's goal selection control rule that calls two

```
(control-rule SELECT-TOP-PRIORITY-AND-COMPATIBLE-GOALS
   (if (and (candidate-goal <goal>)
            (or (ancestor-is-top-priority-goal <goal>)
                (compatible-with-top-priority-goal <goal>))))
   (then select goal <goal>))
```

Table 2.6: Goal selection search control rule.

functions, forcing PRODIGY4.0 to select the goals with high priority as well as the goals that can be opportunistically achieved (without compromising the main high-priority goal).

The test functions in a control rule are known as *meta-predicates*. The meta-predicate `(ancestor-is-top-priority-goal)` calculates whether the goal is required to solve a high-priority goal. ROGUE prioritizes goals according to a modifiable metric. In the current implementation, this metric involves looking at the user's position in the department, the type of request and the deadline: $Priority = PersonRank + TaskRank + DeadlineRank$, where $DeadlineRank$ is defined as shown in Figure 2.5. This function could easily be replaced with alternatives (e.g. [Williamson & Hanks, 1994]).

When the deadline is reached, the goal is removed from PRODIGY4.0's pending goals list; otherwise even an extremely low priority task would eventually be attempted after all other pending tasks have been completed.

The meta-predicate `(compatible-with-top-priority-goal)` allows ROGUE to determine when different goals have similar features so that it can opportunistically achieve lower priority goals while achieving higher priority ones. For example, if multiple people whose offices are all in the same hallway asked for their mail to be picked up and brought to them, ROGUE would do all the requests in the same episode, rather than only bringing the mail for the most "important" person. Compatibility is defined by physical proximity (*"on the path of"*) with a fixed threshold for being too far out of the way. PRODIGY4.0 uses the path planner to calculate the route(s) to the next location(s) of the top priority goal(s), and then adds any other goals whose routes are compatible.

$$DeadlineRank = \begin{cases} \frac{R_{max}}{t_1 - t_0} \times (t - t_0) & t_0 \leq t \leq t_1 \\ 0 & \text{otherwise} \end{cases}$$

Where: t is the current time
R_{max} is the maximum possible rank value
t_1 = deadline − expected_execution_time
t_0 = deadline − 2× expected_execution_time

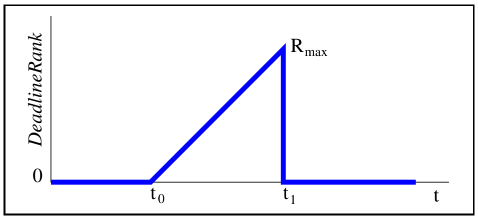


Figure 2.5: Calculating the priority rank of the deadline.

These rules select which *goals* PRODIGY4.0 will focus on. PRODIGY4.0 will suspend planning for goals that are low priority and too far out of the way.

It is possible that these rules will select too many compatible tasks, become "side-tracked," and therefore fail on the high-priority task. A preset threshold would serve as a pragmatic solution to this problem. We also do not deal with the issue of thrashing, i.e., receiving successively more important tasks resulting in no forward progress, because it has not been an issue in practice. Again, a preset threshold would also handle this potential problem. Learning techniques could also be applied.

The learning mechanisms described in Chapter 4 learn goal selection rules to improve planning performance. Essentially, they refine the models of when tasks and actions can be achieved, avoiding them when they cannot be achieved.

Applicable Operator Rules: ROGUE also provides PRODIGY4.0 with a search control rule that selects a good execution order of the applicable actions. Recall that when an action is applied, ROGUE sends it directly to Xavier for execution. (Other heuristics for deciding when to execute an action are described in Section 2.3.2, and Chapter 4 describe one method of learning when to execute.)

This control rule is an execution-driven heuristic, which tries to minimize the expected total traveled distance from the current location. The heuristic uses the nearest-neighbour approximation to the travelling salesman problem (TSP). The heuristic selects the next closest location from the current location of the robot, where the distance estimates are calculated by the path planner. The heuristic performs well in our environment. Table 2.7 shows one of the applied operator control rules.

When all n locations are known before-hand, this heuristic has been shown to be within $\frac{1}{2}\lceil \lg n \rceil$ of optimal [Rosenkrantz *et al.*, 1977]. However, the asynchronous requests in our environment mean that some locations are not known before-hand; therefore each of ROGUE's decisions depend only on what it knows *at that time.* Plans are efficient with respect to the order that task requests arrive. By using the SABA delaying strategy, ROGUE's execution decisions are made with the maximum possible information.

ROGUE's goal selection rules work in concert with its applicable operator rules to control

```
(control-rule SELECT-GOTO-CLOSEST-LOCATION-4
   (if (and (candidate-applicable-op
                      (GOTO-DELIVER-LOC <user1> <item1> <loc1> <curloc1>))
            (candidate-applicable-op
                      (GOTO-DELIVER-LOC <user> <item> <loc> <curloc>))
            (diff <loc1> <loc>)
            (true-in-state (robot-in-room <roboroom>))
            (closer <roboroom> <loc1> <loc>)))
   (then reject apply-op (GOTO-DELIVER-LOC <user> <item> <loc> <curloc>)))
```

Table 2.7: A control rule to select execution order.

PRODIGY4.0's behaviour. The goal selection rules prune the search space to create plans for high priority and compatible goals. Then the applicable operator rules fire to select amongst pending applicable operators for the selected goals. A lower priority, incompatible task will not have pending applicable operators.

The learning mechanisms described in Chapter 4 learn applicable operator rules to improve execution performance.

2.2.3 Suspending and Interrupting Tasks

ROGUE needs to be able to respond quickly when new tasks arrive and also when priorities of existing tasks change. PRODIGY4.0 supports these changing objectives by making it easy to suspend and reactivate tasks.

PRODIGY4.0 grows the plan incrementally, meaning that each time it selects a goal to expand, the remainder of the plan is unaffected. The system can therefore easily suspend planning for one task while it plans for another. ROGUE evokes this behaviour in PRODIGY4.0, through its control rules: when a rule rejects a particular goal, that goal is effectively suspended, and when a rule selects a particular goal, other goals are suspended.

The planning already done for the suspended goals remains valid until PRODIGY4.0 is able to return to them. When PRODIGY4.0 does in fact return to the suspended actions, it validates their preconditions in the state, expanding the plan if necessary, or continuing execution if appropriate.

Generally, the plans for the interrupted goals will not be affected by the planning and execution for the new goal. (Because of the delayed bindings[2] in the `(GOTO)` actions, moving around the building does not affect planning for tasks; moving only affects the ordering of applicable actions.)

Occasionally, however, actions executed to achieve the new goal might undo or achieve parts of the interrupted plan. For example, the robot might have finished its new task in the pickup location of the interrupted task, allowing PRODIGY4.0 to ignore a `(GOTO)` action. There are also occasions in which exogenous events may change the state, such as if a user passed the robot in the corridor and took his mail at that time, in which case PRODIGY4.0 could remove the actions relating to delivering the mail.

In cases like these, ROGUE's execution monitoring algorithm will update PRODIGY4.0's state information and PRODIGY4.0 will know which preconditions it needs to re-achieve or to ignore. In Section 2.3.3 we discuss in more detail how side-effects of actions and exogenous events may affect interrupted or pending plans.

2.2.4 Example: Asynchronous Requests

We now present a detailed example of how PRODIGY4.0, ROGUE and Xavier interact in a two goal problem: `(has-item mitchell delivermail)` and `(has-item jhm deliverfax)`.

[2] See Section 2.2.2.2 for the description of delayed bindings.

The second goal is higher priority, and arrives while ROGUE is executing the first action for the first goal.

Figure 2.6 shows the tail-plan generated by PRODIGY4.0. We describe below the details of how it is generated. This example illustrates:

- how PRODIGY4.0 generates plans for task requests,
- how ROGUE's search control rules affect PRODIGY4.0's selections,
- how an asynchronous task request affects the plan, and
- how the planner interacts with the executor.

We assume for the purposes of this example that no failures occur during execution. The example is perhaps overly detailed for a reader familiar with back-chaining planners; those readers could skip to the next section without loss of continuity.

We show the algorithmic sequence of steps of PRODIGY4.0. At each step, we show the lists of pending goals, *PG*, applicable operators, *Applicable-Ops*, and executed operators, *Executed-Ops*.

The plan shown in Figure 2.6 corresponds to PRODIGY4.0's *tail-plan*, while the *Executed-Ops* correspond to PRODIGY4.0's *head-plan*. Recall that operators are executed at the operator application phase of planning, that is, when they are moved from the tail-plan to the head-plan.

1. Request `(has-item mitchell delivermail)` arrives. ROGUE adds this goal to PRODIGY4.0's pending goals list, *PG*, and adds the following knowledge to PRODIGY4.0's state information:

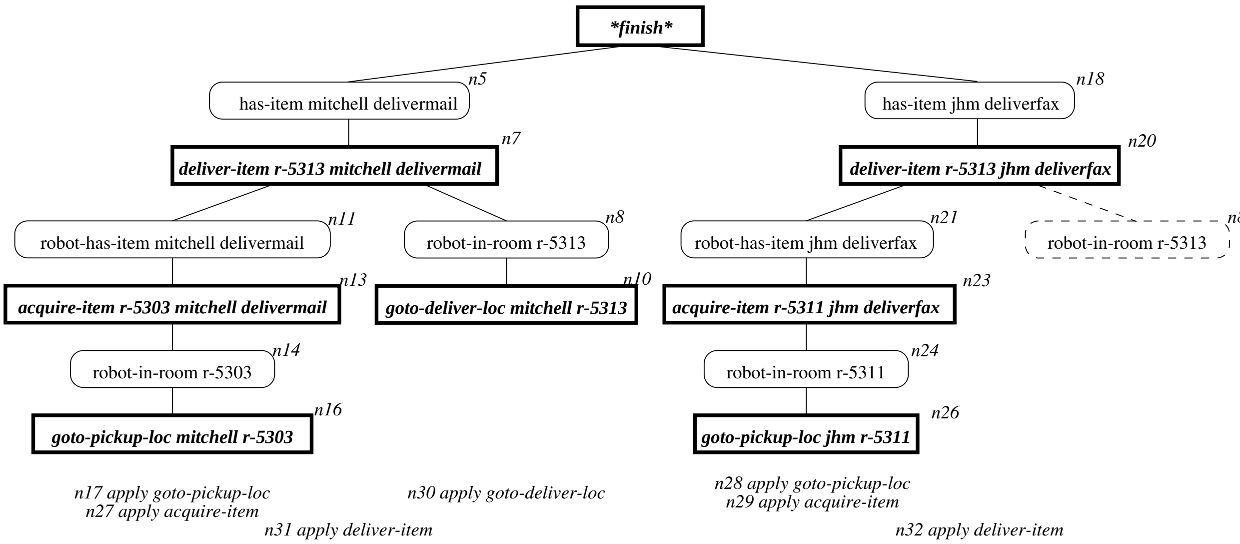


Figure 2.6: Plan for a two-task problem; goal nodes are in ovals, required actions are in rectangles. Nodes are labelled with their node number; missing numbers correspond to uninstantiated operators.

```
(needs-item mitchell delivermail)
(pickup-loc mitchell r-5303)
(deliver-loc mitchell r-5313)
```

PG is (has-item mitchell delivermail)
Applicable-Ops is nil
Executed-Ops is nil

2. PRODIGY4.0 fires its goal-selection search control rules, which selects this goal (node 5 of Figure 2.6) as the highest priority goal (since it is the only choice). PRODIGY4.0 examines this goal to find an appropriate operator. It finds (DELIVER-ITEM <user> <room> <object>), and instantiates the variables: <user> := mitchell, <room> := r-5313 and <object> := delivermail, yielding the instantiated operator shown in node 7. Using means-ends analysis, PRODIGY4.0 identifies two preconditions not satisfied in the state: (robot-has-item mitchell delivermail) and (robot-in-room r-5313). PRODIGY4.0 adds these preconditions to the pending goals list.

   PG is (and (robot-has-item mitchell delivermail)
   (robot-in-room r-5313))
   Applicable-Ops is nil
   Executed-Ops is nil

3. PRODIGY4.0 continues expanding the plan for this task, yielding nodes 5 through 16. At this moment, two operators in the plan have all their preconditions met in the current state. Node 10 is not applied when it is expanded because of the SABA delaying strategy.

   PG is nil
   Applicable-Ops is (and (GOTO-DELIVER-LOC mitchell r-5313)
   (GOTO-PICKUP-LOC mitchell r-5303))
   Executed-Ops is nil

4. PRODIGY4.0 examines the set of *Applicable-Ops*, and based on ordering constraints (goal clobbering), selects (GOTO-PICKUP-LOC mitchell r-5303) to apply. ROGUE takes the applied operator, moves it to the head plan, and sends it to the robot for execution. (It does not need to verify preconditions in the real world since none can be changed by exogenous events.)

   PG is nil
   Applicable-Ops is (GOTO-DELIVER-LOC mitchell r-5313)
   Executed-Ops is nil

5. Request `(has-item jhm deliverfax)` arrives. ROGUE adds this goal to *PG*. *Note that* PRODIGY4.0 *does not plan for it.*
   ROGUE does not interfere with the currently executing action, namely `(GOTO-PICK-UP-LOC mitchell r-5303)`. Goodwin [1994] discusses methods to decide when to interfere.

   PG is `(has-item jhm deliverfax)`
   Applicable-Ops is `(GOTO-DELIVER-LOC mitchell r-5313)`
   Executed-Ops is `nil`

6. The navigation module finally indicates completion of the action. ROGUE verifies the outcome (post-conditions) of the action, i.e., that it has arrived at the location `r-5313` (see Section 2.3 for a description of this verification step). Now the action `(ACQUIRE-ITEM r-5303 mitchell delivermail)` is applicable.

   PG is `(has-item jhm deliverfax)`
   Applicable-Ops is `(and (GOTO-DELIVER-LOC mitchell r-5313)`
   `(ACQUIRE-ITEM r-5303 mitchell delivermail))`
   Executed-Ops is `(GOTO-PICKUP-LOC mitchell r-5303)`

7. PRODIGY4.0 fires ROGUE's search control rules, which select the new goal (since it is higher priority than the current task) (node 18). It expands the plan as for the first task, except that instead of selecting an additional operator to achieve the goal `(robot-in-room r-5313)`, it notices that the operator `(GOTO-DELIVER-LOC mitchell r-5313)` has the same effect, and does not redundantly add a new operator.

   PG is `nil`
   Applicable-Ops is `(and (GOTO-DELIVER-LOC mitchell r-5313)`
   `(ACQUIRE-ITEM r-5303 mitchell delivermail)`
   `(GOTO-PICKUP-LOC jhm r-5311))`
   Executed-Ops is `(GOTO-PICKUP-LOC mitchell r-5303)`

8. Since the robot is standing in front of room 5303, a control rule fires to select the applicable operator `(ACQUIRE-ITEM r-5303 mitchell delivermail)` (node 27). Note that while `(GOTO-PICKUP-LOC jhm r-5311)` is higher priority, this action is compatible with it.
   ROGUE verifies `(ACQUIRE-ITEM)`'s preconditions and then sends it to the robot for execution. When the action is complete, ROGUE verifies the postconditions to check that it now has mitchell's mail.

   PG is `nil`
   Applicable-Ops is `(and (GOTO-DELIVER-LOC mitchell r-5313)`
   `(GOTO-PICKUP-LOC jhm r-5311))`
   Executed-Ops is `(GOTO-PICKUP-LOC mitchell r-5303)`
   `(ACQUIRE-ITEM r-5303 mitchell deliver-mail)`

9. The execution constraint control rule now selects `(GOTO-PICKUP-LOC jhm r-5311)` as the next applicable operator (node 28). ROGUE sends it to Xavier for execution and monitors its outcome.

   PG is `nil`
   Applicable-Ops is `(and (GOTO-DELIVER-LOC mitchell r-5313)`
   `(ACQUIRE-ITEM r-5311 jhm deliverfax))`
   Executed-Ops is `(GOTO-PICKUP-LOC mitchell r-5303)`
   `(ACQUIRE-ITEM r-5303 mitchell deliver-mail)`
   `(GOTO-PICKUP-LOC jhm r-5311)`

10. ROGUE then acquires the fax.
11. ROGUE then goes to room 5313.
12. ROGUE delivers both items.

The final execution order described in this example is shown in Table 2.8, where the second request arrives during the execution of the first. This example illustrates the asynchronous handling of goals in ROGUE, and our interleaved planning and execution paradigm.

```
Solution:
    <GOTO-PICKUP-LOC mitchell r-5303>
  [arrival of second request]
    <ACQUIRE-ITEM r-5303 mitchell delivermail>
    <GOTO-PICKUP-LOC jhm r-5311>
    <ACQUIRE-ITEM r-5311 jhm deliverfax>
    <GOTO-DELIVER-LOC mitchell r-5313>
    <DELIVER-ITEM r-5313 jhm deliverfax>
    <DELIVER-ITEM r-5313 mitchell delivermail>
```

Table 2.8: Final execution sequence.

2.3 Execution and Monitoring

In this section we describe how ROGUE mediates the interaction between the planner and the robot. There are three places where ROGUE controls the robot's execution:

- when decisions are made about actions or tasks,
- when actions are executed, and
- when the environment or actions are monitored.

ROGUE's control rules may use sensing actions to help make planning decisions. Below, we describe how sensing may be integrated into planning, and discuss some of the limits placed on execution at this phase.

Each time PRODIGY4.0 generates a plan step, ROGUE translates the abstract action description into a sequence of commands for the robot. In this section, we show how PRODIGY4.0's symbolic action descriptions are turned into robot commands.

Before executing an action, ROGUE monitors the environment to verify its preconditions and postconditions. Also, ROGUE continuously monitors changes in the environment that may affect current goals. We describe below how robot sensor data is incorporated into the planner's knowledge base so that the planner can compensate for changes in the environment or unexpected failures of its actions.

The key to this communication model is based on a pre-defined language and model translation between PRODIGY4.0 and Xavier. The procedures to do this translation are manually generated, but are in a systematic format and may be extended at any time to augment the actions or sensing capabilities of the system. It is an open problem to automate the generation of these procedures because it is not only challenging to select what features of the world may be relevant for replanning, but also how to detect those features using existing sensors.

2.3.1 Sensing in Control Rules

While PRODIGY4.0 is expanding a plan, search control rules[3] fire to guide the planner's decisions. These rules traditionally rely on state information contained in PRODIGY4.0's simulated state, but there is no inherent limitation preventing ROGUE from sensing directly from the external environment.

For example, current external state can be used to decide whether or not to execute an action. One of ROGUE's control rules could sense the battery power levels to decide whether to return to a recharging station. The camera or sonars could be used to detect whether a room is occupied, thereby deciding whether or not to try and acquire or deliver an item. In Chapter 4, we show an example of this kind of rule, and how it can be learned from experience.

Currently, none of the control rules that use sense the environment directly modify PRODIGY4.0's internal state. However, there may be times when it would be beneficial to store such information, and there is no inherent limitation on designing rules that do so.

While there are no implementation limitations on what execution can happen while firing a control rule, conceptually the rule should *not modify* the external state. Such behaviour should be relegated to operators. Control rules should only sense the state, and it is the designer's responsibility to ensure that they do not modify it.

2.3.2 Executing Actions

Each action that PRODIGY4.0 selects must be translated into a form that Xavier will understand. ROGUE translates the high-level abstract action into a command sequence appropriate for execution.

[3]See Section 2.2.2.3 for a description of control rules.

2.3.2.1 PRODIGY4.0's Mechanisms for Supporting Execution

PRODIGY4.0 allows arbitrary procedural attachments that are called during the operator application phase of the planning cycle [Haigh *et al.*, 1997b; Stone & Veloso, 1996]. Typically, we use these functions to give the planner additional information about the state of the world that might not be accurately predictable from the model of the environment. For example, this new information might show resource consumption or action outcomes.

ROGUE extends this information-gathering capability because, instead of *simulating* operator effects, ROGUE actually sends the commands to the robot for *real world* execution. Actually executing the planner's actions in this way increases system reliability and efficiency because the system can respond quickly to unexpected events, and the planner knows the exact outcome of its uncertain actions, reducing the planning effort because the planner does not need to plan for multiple outcomes.

In order to execute operators, several mechanisms were added to PRODIGY4.0 [Haigh *et al.*, 1997b; Stone & Veloso, 1996]. The new PRODIGY4.0/EXECUTE algorithm is shown in Table 2.9. First, the designer needs to define the execution behaviour of the operator (**Change-state-on-execute**). Second, the designer needs to define when to execute the operator (**Automatically-decide-to-execute**). Finally, the designer needs to decide whether to simulate the effects of execution at the apply state, so as to allow backtracking before committing to execution (**Change-state-on-apply**).

2.3.2.2 Deciding When to Execute

It is an important problem to decide when it is safe to execute an operator. The default behaviour of PRODIGY4.0/EXECUTE is to interactively exploit the user's intuition about the domain [Stone & Veloso, 1996]. ROGUE, on the other hand, *eagerly* executes actions, primarily because we want the system to function autonomously.

Most domains fall into this second category; asking the user or defining a complex decision-making function is more effective when (i) the domain is dangerous and modelling is difficult, or (ii) a lot of backtracking occurs to find the correct ordering of applicable operators. It is an important open problem to design a domain-independent heuristic to select execution points that will not backtrack. Some initial research efforts are described in Section 5.1.

ROGUE solves these two problems through the use of control rules. Dangerous operator effects are avoided with *prefer* rules: prefer a safer alternative until there are no other choices. The correct ordering of applicable operators is defined by the TSP control rule (described in Section 2.2.2.3).

2.3.2.3 ROGUE's Execution Behaviour

The eager execution behaviour of ROGUE simplifies the PRODIGY4.0/EXECUTE algorithm. Every operator is executed immediately after it is applied. **Automatically-decide-to-execute** returns "`t`", and **Change-state-on-apply** does nothing. **Change-state-on-execute**

PRODIGY4.0/EXECUTE
1. If the goal statement G is satisfied in the current state, terminate.
2. Either (A) Subgoal: add an operator to *Tail-Plan* (**Back-Chainer**), or
   (B) Apply: move an operator from *Tail-Plan* to *Head-Plan* (**Operator-Application**), or
   (C) Execute: execute an operator previously applied (**Operator-Execution**).
   Decision point: Decide whether to apply, subgoal, or execute.
   Calls **Automatically-decide-to-execute**.
3. Recursively call **PRODIGY4.0/EXECUTE** on the resulting plan.

Automatically-decide-to-execute
User-defined. Default behaviour is to ask the user.

Operator-Application
1. Pick an operator *op* in *Tail-Plan* which is an *applicable operator*, that is the preconditions of *op* are satisfied in the current state.
   Decision point: Choose an operator to apply.
2. Move *op* from *Tail-Plan* to *Head-Plan*.
3. Simulate undetectable state changes (**Change-state-on-apply**).
4. Update the current state with the effects of *op*.

Change-state-on-apply
User-defined. Default behaviour is to do nothing.

Operator-Execution
1. Pick an operator *op* in *Head-Plan* which has already been *applied*.
   Decision point: Choose an applied operator to execute.
2. Execute the operator and update the current state with the real-world effects of *op* (**Change-state-on-execute**).

Change-state-on-execute
User-defined. Default behaviour is to do nothing.

Table 2.9: The PRODIGY4.0/EXECUTE search algorithm, reproduced from Haigh *et al.* [1997b]. The PRODIGY4.0 algorithm from Table 2.5 is modified to decide whether to apply (**Operator-Application**), or to subgoal (**Back-Chainer**), or to execute (**Operator-Execution**). **Back-Chainer** remains unchanged from Table 2.5.

contains all relevant execution behaviour. Effectively, these two definitions reduce the PRODIGY4.0/EXECUTE algorithm to the original PRODIGY4.0 algorithm, with the added behaviour of executing the operator during the application phase of planning.

In the function **Change-state-on-execute**, each operator has a predefined behaviour. In general, this behaviour is

1. to verify the preconditions of the operator,
2. to execute an associated command sequence,
3. to verify the postconditions of the operator, and
4. if necessary, to attempt to recover from simple failures.

These execution behaviours resemble schemas [Georgeff & Ingrand, 1989; Hormann *et al.*, 1991] or RAPs [Firby, 1989; Gat, 1992; Pell *et al.*, 1997], in that they specify how to execute the action, what to monitor in the environment, and some internal recovery procedures. ROGUE's execution behaviours, however, do not contain complex recovery or monitoring procedures, which we believe the planner should explicitly reason about.

The two verification steps are part of the *action monitoring* sequence described in Section 2.3.3, where we explain how ROGUE monitors the outcome of the action, and how failures may cause replanning or affect plans of interrupted tasks.

Below, we describe executing the command sequence, and internal failure recovery.

Executing an Action. Each operator has an associated command sequence for executing the action. This command sequence may be executed directly by ROGUE (e.g. a command like `finger` to determine an office location), or sent via the TCA interface to the Xavier module designed to handle the command. The action `(ACQUIRE-ITEM <room> <user> <item>)`, for example, is mapped to a sequence of commands that allows the robot to interact with a human. The action `(GOTO-PICKUP-LOC <user> <room>)` is mapped to the commands shown in Table 2.10, extracted from an actual trace: (1) Announce intended action, (2) Ask Xavier's path planner to find the coordinates of a door near the room, and (3) Navigate to those coordinates. The outcome is then verified by the action monitors. The example in Section 2.3.4 shows some additional operators.

Only one action at a time is sent for execution, and while the robot is executing, the planner waits for execution feedback. Hence, the planner remains fully in control of the robot at all times. Some planners send the robot command sequences for multiple actions or even complete plans, thereby assuming that action failures, beneficial side-effects, and exogenous events will be rare. Alternative approaches to planning and execution are discussed in Sections 2.4 and 5.1.

In ROGUE, there are two possible methods for permitting execution of parallel actions. The first is through an explicit definition of a "continuous action," which would have two parts: a "start" operator and an "end" operator. The planner would then be able to continue planning during execution.

The second method is through an environment monitor, which would, for example, monitor for objects to recycle. When such an object is noticed, execution could be halted, the

```
<GOTO-DELIVER-LOC MITCHELL R-5309>

SENDING COMMAND (tcaExecuteCommand "C_say" "Going to room 5309")
ANNOUNCING: Going to room 5309
SENDING COMMAND (tcaQuery "nearRoomQ" "5309")
   ...Query returned #(TASK-CONTROL::NEARROOMREPLY 567.0d0 3483.0d0 90.0d0)
SENDING COMMAND (tcaExpandGoal "navigateToG" #(TASK-CONTROL::MAPLOCDATA 567.0d0 3483.0d0))
   ...waiting...
   ...Action NAVIGATE-TO-GOAL finished (SUCCESS).
SENDING COMMAND (tcaQuery "visionWhereAmI")
   ...Query returned #(TASK-CONTROL::VISIONWHEREAMI "5309")
```

Table 2.10: The set of actions taken for executing the PRODIGY4.0 operator `<GOTO-DELIVER-LOC mitchell r-5309>`.

object acquired, and then execution continued. ROGUE briefly used this technique for incorporating new requests that arrived during the execution of a long continuous action: when a new request came in, ROGUE checked to see if the new goal might, in any way, have affected the executing action. It might have higher priority, or be compatible with existing tasks. When the new goal did affect the executing action, that action was halted, and PRODIGY4.0 was notified that it had failed. PRODIGY4.0 then incorporated the new goal in its continued planning. This behaviour was removed from ROGUE because it was hard to make claims about the system's performance when there are frequent requests.

Internal Failure Recovery. After the action has been executed, the action monitors verify the outcome of the action. If the execution failed, then a simple recovery procedure is invoked; this recovery procedure is completely internal to the action. Internal recovery procedures handle common, known failures with simple, directly applicable behaviour.

For example, when the navigation module fails, the `(GOTO)` operators will simply request a new path plan, and then reinvoke the `navigateToGoal` command. The command performs well given incomplete or incorrect metric information about the environment and in the presence of noisy effectors and sensors, arriving at its destination approximately 95% of the time [Simmons & Koenig, 1995].

At the scene of a pickup or delivery, if ROGUE times-out while waiting for a response to a query, ROGUE will prompt for a user a second time before failing.

ROGUE's execution behaviours do not contain complex recovery or monitoring procedures, since we feel that it is more appropriate for the planner to reason about when they should be used. Different recovery procedures may have different costs, reliabilities, or relevance, and it may be important to reason about the tradeoffs. For example, we prefer the planner to *plan* whether to undo side effects — there may be situations when it is important to undo them, and other situations in which they can be ignored. Extracting complex recovery procedures from the actions allows the planner to reason about tradeoffs that would be hard to explicitly enumerate (see Section 5.1 for a discussion on this point).

If the internal recovery procedures cannot recover from the failure, ROGUE deletes the desired effects from PRODIGY4.0's state, thereby forcing PRODIGY4.0 to replan to achieve those effects.

2.3.3 Monitoring

In this section, we describe how ROGUE monitors the environment, and how changes in the environment may cause replanning. There are two types of events that ROGUE needs to monitor in the environment.

The first centres around *actions*. Each time ROGUE executes an action, it needs to verify its outcome because actions may have multiple outcomes or fail unexpectedly. ROGUE may need to invoke replanning, or select actions at a branching condition. ROGUE also needs to verify the preconditions of an action before executing it because the world may change, invalidating one of the system's beliefs. ROGUE uses a layered verification process, incrementally calling methods with greater cost and accuracy, until a predefined confidence threshold is reached. Action monitors are invoked only when the action is executed. A detailed example is presented in Section 2.3.5.

The second centres around exogenous events in the *environment*. Certain events may cause changes in the environment that affect current goals, or opportunities may arise that ROGUE can take advantage of. Environment monitors are invoked when relevant goals are introduced to the system. For example, ROGUE can monitor battery power, or examine camera images for open doors or particular objects.

Both types of monitoring procedures specify (1) what to monitor and (2) the methods that can be used to monitor it. *Action monitors* monitor the preconditions and effects of the action, while *environment monitors* are a function of the goals the system can achieve. Since action monitors are based on the planning domain model, they provide a focus for execution monitoring. It is an open problem to autonomously decide what exogenous events to monitor that will be relevant for planning.

Although action monitoring is sequential and of limited time-span, while environment monitoring is parallel and continuous, the two sets of procedures have similar effects on planning.

Once ROGUE has done the required monitoring, ROGUE needs to update PRODIGY4.0's state description as appropriate. In execution monitoring, the update occurs when the object is detected, or when battery power falls below a certain threshold. In action monitoring, the critical update is when the *actual* outcome of the monitoring does not meet the *expected* outcome. These updates will force PRODIGY4.0 to re-examine its plan, adding or discarding operators as necessary.

If the primary effect of an action has been unexpectedly satisfied, ROGUE adds the knowledge to PRODIGY4.0's state description and PRODIGY4.0 does not attempt to achieve it. For example, if a user passed the robot in the corridor and took his mail at that time, ROGUE could delete the goal from the state, allowing PRODIGY4.0 to remove actions relating to delivering the mail. Observing the environment and maintaining a state description in this way improves the efficiency of the system because it will not attempt redundant actions.

If a required precondition is no longer satisfied as a side-effect of some other action or detected by environment monitoring, ROGUE deletes the relevant precondition from PRODIGY4.0's state. PRODIGY4.0 will therefore replan in an attempt to find an action that will re-achieve it. For example, if learning delays the execution of an acquire or deliver action, then the precondition `(robot-in-room <room>)` is deleted when the robot moves. PRODIGY4.0 will need to re-navigate to the required room.

In action monitoring, if the action fails, ROGUE will first try the internal recovery methods. These recovery methods are very simple; more complex ones are treated as separate operators for PRODIGY4.0 to reason about explicitly. For example, ROGUE will try calling the navigation routine a predefined number of times before deciding that the action completely failed. If, despite these built-in recovery methods, ROGUE determines that the action has completely failed, ROGUE will delete the effect from PRODIGY4.0's state description, and PRODIGY4.0 will replan to achieve it. PRODIGY4.0 will not simply re-apply the same action, since the backtracking rules would not allow it.

Occasionally during environment monitoring, knowledge will unexpectedly be added to the state that causes an action to become executable, or a task to become higher priority. Each time PRODIGY4.0 makes a decision, it re-examines all of its options, and will factor the new action or goal into the process.

Autonomously deciding which preconditions and effects need to be verified is an important open problem. In ROGUE, all effects need to be verified since real world execution may fail. However, the only preconditions that need to be verified are those which can change externally. For example, the precondition `(needs-item <user> <item>)` is completely internal to PRODIGY4.0; if a user were to delete the request, then an *environment* monitor would delete the literal from the state and PRODIGY4.0 would no longer plan for it.

In this manner, ROGUE is able to detect execution failures and compensate for them, as well as to respond to changes in the environment. The interleaving of planning and execution reduces the need for replanning during the execution phase and increases the likelihood of overall plan success because the planner is constantly being updated with information about changes in the world. It allows the system to adapt to a changing environment where failures can occur.

2.3.4 Example: Sensing to Make Planning Decisions

One of the benefits of interleaving planning with execution is that sensor information can be used to prune the search space directly. In this example, we show how ROGUE deliberately senses the environment at a branching operator.

We illustrate the incorporation of perception information from execution into planning through an example corresponding to one of the events from the AAAI 1996 robot competition. The environment consists of three conference rooms and several offices. The director wishes to schedule a meeting in a conference room. ROGUE needs to find an empty conference room and then inform all the meeting attendees which room the conference will be in, and at what time. The task requires ROGUE to incorporate observation knowledge into planning in order to accurately and efficiently complete the task.

When ROGUE receives the task request, it spawns a PRODIGY4.0 run, giving PRODIGY4.0 relevant task information such as who are the attendees and which rooms are potential conference rooms. The path planner knows the topological layout of the rooms, but does not know the exact location of the doors.

PRODIGY4.0 starts creating a plan by alternating considering the goals and their subgoals and the different ways of achieving them. When PRODIGY4.0 finds that there are several possible conference rooms, a control rule fires to check the closest room for availability[4].

Table 2.11 shows a partial trace of a run. When PRODIGY4.0 applies the `(GOTO-ROOM)` operator in its simulated environment model (see node 17), ROGUE sends the command to Xavier for execution. Each line marked "`SENDING COMMAND`" indicates a direct command sent through the TCA interface to one of Xavier's modules.

This trace uses three TCA commands: `navigateToG`(oal), `C_observe` and `C_say`. `C_observe` is a direct perception action. The observation routine can vary depending on the kind of information needed. It can range from an actual interpretation of some of Xavier's sensors or its visual images, to specific input by a user. During the competition, we used motion detection and face detection routines [Simmons *et al.*, 1996]. The command `C_say` sends the string to the speech board.

Once the navigate module has successfully completed, ROGUE tells Xavier's vision module to observe the room. In this example, the conference room is occupied, and ROGUE updates PRODIGY4.0's state by setting `(room-empty 5309)` to false, and `(room-occupied 5309)` to true. At this stage, the head-plan contains the two executed operators, while the tail-plan contains the expanded search tree, as shown in Figure 2.7.

Because there are no operators in the domain model that can be used to empty a room, replanning forces PRODIGY4.0 to use another conference room for the meeting (i.e. PRODIGY4.0 backtracks to node 6 and selects a different conference room, node 19). Figure 2.8 shows the partial plan at this stage.

The run proceeds until ROGUE finds a conference room that is empty or until it exhausts all the available conference rooms. The goal `(people-informed)` is expanded after a meeting room has been selected, and then ROGUE proceeds to tell the attendees where their meeting will be. The announcement is made by navigation to each individual room. The final plan executed by Xavier is shown in Figure 2.9. Xavier stops at all the conference rooms until it correctly identifies an empty one, and then tells all the attendees when and where the meeting will be (within 3.5 minutes in 5311). This behaviour was developed in Xavier's simulator and then applied successfully on the real robot.

---

[4]This control rule corresponds to the applicable operator TSP rule, Section 2.2.2.3, but at a much earlier stage of planning, since the room location needs to be known well in advance.

```
    n2 (done)
    n4 <*finish*>
      n5 (mtg-scheduled)
      n6 schedule-meeting
Firing prefer bindings LOOK-AT-CLOSEST-CONF-ROOM-FIRST #<5309> over #<5311>
      n7 <schedule-meeting 5309> [1]
        n8 (conference-room 5309)
        n10 <select-conference-room 5309>
          n11 (at-room 5309)
          n13 <goto-room 5309>
          n14 (room-empty 5309)
          n16 <observe-room 5309>
          n17 <GOTO-ROOM 5309>

SENDING COMMAND (tcaExecuteCommand "C_say" "Going to room 5309")
ANNOUNCING: Going to room 5309
SENDING COMMAND (TCAEXPANDGOAL "navigateToG" #(TASK-CONTROL::MAPLOCDATA 567.0d0 3483.0d0))
   ...waiting...
Action NAVIGATE-TO-GOAL finished (SUCCESS).

          n18 <OBSERVE-ROOM 5309>

SENDING COMMAND (tcaExecuteCommand "C_observe" "5309")
DOING OBSERVE: Room 5309 conf-room
   ...waiting...
Action OBSERVE finished (OCCUPIED).

SENDING COMMAND (tcaExecuteCommand "C_say" "This room is occupied")
ANNOUNCING: This room is occupied

  6   n6 schedule-meeting
  7   n19 <schedule-meeting r-5311>
```

Table 2.11: Partial trace of ROGUE interaction, in which direct observation is used to make planning decisions.

Observing the real world allows the system to adapt to its environment and to make intelligent and relevant planning decisions. Observation allows the planner to update and correct its domain model when it notice changes in the environment. For example, it can notice limited resources (e.g. battery), notice external events (e.g. doors opening/closing), or prune alternative outcomes of an operator. In these ways, observation can create opportunities for the planner and it can also reduce the planning effort by pruning possibilities. Real-world observation creates a more robust planner that is sensitive to its environment.

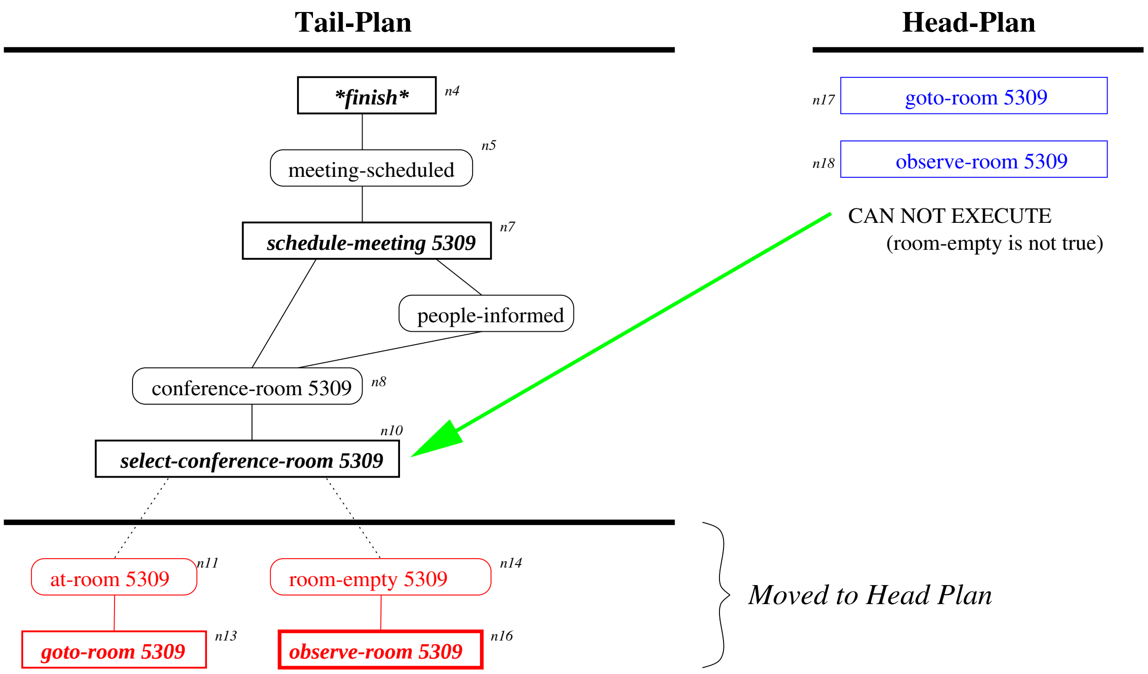


Figure 2.7: Partial plan after room 5309 has been observed.

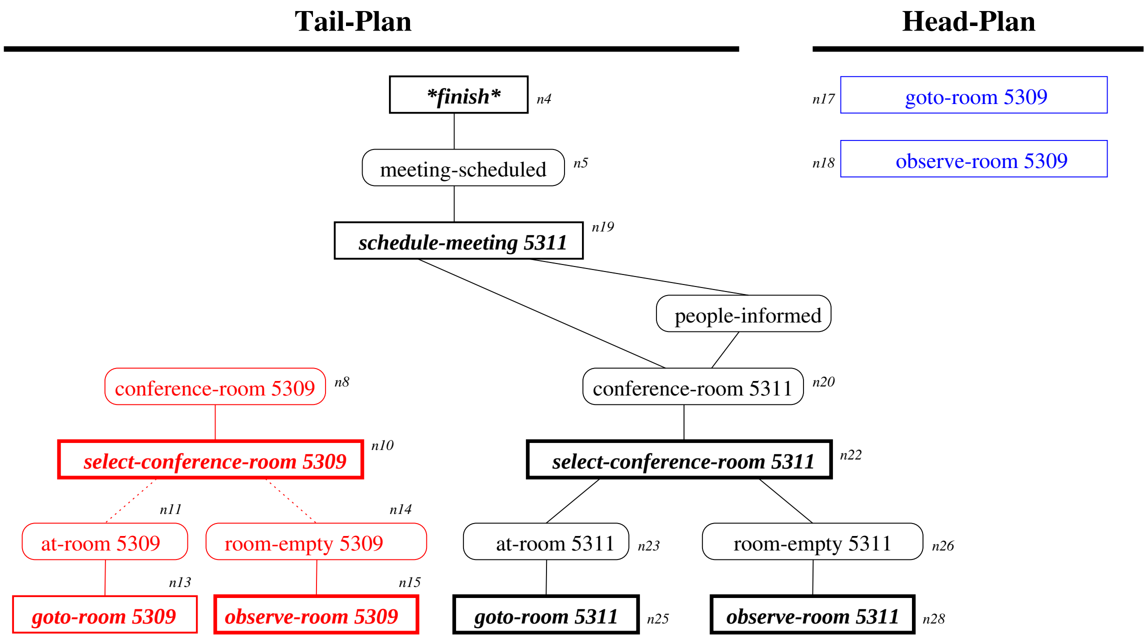


Figure 2.8: Partial plan after backtracking occurred and immediately before room 5311 has been observed.

```
<goto-room 5309>
<observe-meeting-room 5309>
<goto-room 5311>
<observe-meeting-room 5311>
<select-meeting-room 5311>
<goto-room 5307>
<inform-person-of-meeting director 3.5 5311>
<goto-room 5303>
<inform-person-of-meeting professor-G 3.5 5311>
<goto-room 5301>
<inform-person-of-meeting professor-S 3.5 5311>
```

Figure 2.9: Executed plan

2.3.5 Example of how Rogue Handles Failures

One of Xavier's actions that Rogue monitors is the `navigateToGoal` command, used by both the `(GOTO-PICKUP-LOC <user> <room>)` and the `(GOTO-DELIVER-LOC <user> <room>)` operators. `navigateToG` reports a success when the robot arrives at the requested goal. `navigateToG` may fail under several conditions, including detecting a corridor or door blockage, or lack of forward progress. The module is able to autonomously compensate for certain problems, such as obstacles and missing landmarks. Navigation is done using Partially Observable Markov Decision Process models [Simmons & Koenig, 1995], and the inherent uncertainty of this probabilistic model means that the module may occasionally report success even when it has not actually arrived at the desired goal location.

When `navigateToG` reports a failure or a low-probability success, Rogue verifies the location. Rogue first tries to verify the location autonomously, using its cameras. The vision module looks for a door in the general area of the expected door, and finds the room label, and reads it. If this module fails to find a door, fails to find a label, or returns low confidence in its template matching, Rogue falls back to a second verification procedure, namely using the speech module to ask a human. We assume that verification step gives complete and correct information about the robot's actual location; other researchers are focussing on the open problem of sensor reliability [Hughes & Ranganathan, 1994; Thrun, 1996].

If Rogue detects that in fact the robot is *not* at the correct goal location, Rogue updates the navigation module with the new information and re-attempts to navigate to the desired location. If the robot is still not at the correct location after a constant number of tries (three in our current implementation), Rogue updates PRODIGY4.0's task knowledge to reflect the robot's actual position, rather than the expected position.

In general, PRODIGY4.0 has several different operators that can achieved a particular effect, and will successfully replan for the failure. In this case, however, there are no other alternative methods of navigating, and PRODIGY4.0 declares that the task can not be suc-

```
goto-location( num-times, room )
    if (num-times > max-retries)
        report failure to execute, remove all planning info
    else
        (navigateToG room)
        if  (or ((where-am-i-now) ≠ room)
                ((best-prob-markov-state) < threshold ))
            actual-room ← (verify-location room))
            if (actual-room ≠ room))
                remove literal (robot-in-room room) from state
                add literal (robot-in-room actual-room) to state
                goto-location( num-times+1, room )

verify-location( room )
    vision-room ← (visionWhereAmI confidence-threshold)
    if (vision-room)
        return vision-room
    else return (ask-human "Where am I?")
```

Table 2.12: An outline of the monitoring and recovery procedure used for the navigation operators. Courier font is used to indicate calls to Xavier.

cessfully achieved. ROGUE removes the task from PRODIGY4.0's goals lists G and PG which effectively kills the task.

The commands used for these navigation operators are outlined in Table 2.12. A short trace appeared in Table 2.10.

2.3.6 Example of how ROGUE Handles Side-effects

Occasionally, suspending one task for a second one will mean that work done for the first will be undone by work done for the second. ROGUE needs to detect these situations and plan to re-achieve the undone work. Consider a simple situation that illustrates this re-planning process:

Task one:	**Task two:**
1a. goto 5301	2a. goto 5409
1b. pick up mail	2b. pick up fed-ex package
1c. goto 5315	2c. goto 4320
1d. drop off mail	2d. drop package off

Many possible interleaved planning and execution scenarios may occur; below are two possibilities.

- **[Normal:]** ROGUE executes 1a and 1b. While executing, the request for task two arrives. ROGUE decides that task two is more important. Task one is suspended; step 1c is pending. ROGUE plans for and executes task two. ROGUE returns to step 1c, verifies that it is still needed to complete the task and can still be done, then does 1c and 1d.

- **[Undone Action:]** ROGUE executes 1a. While executing 1b, the request for task two arrives. 1b times-out, indicating that the mail-room person wasn't there to give the robot the mail. ROGUE decides task two is more important, and suspends task one; step 1b is pending. ROGUE plans for and executes task two. ROGUE returns to step 1b, discovers that a precondition is not true: `(robot-in-room <5301>)`. ROGUE re-plans to achieve it, and then re-executes step 1a, and then finishes the task as expected.

When ROGUE's environment monitors detect a change in world state, either from an exogenous event or as a side-effect of some other action, ROGUE modifies PRODIGY4.0's internal state description.

- When ROGUE *deletes* a relevant *precondition*, PRODIGY4.0 is forced to create a plan for it.
- When ROGUE *adds* a relevant *precondition*, PRODIGY4.0 will not execute any actions that try to achieve it.
- When ROGUE *deletes* a primary *effect* of an already-executed action, PRODIGY4.0 is forced to replan for it.
- When ROGUE *adds* a primary *effect* of a required-to-execute action, PRODIGY4.0 will not execute that action.

ROGUE thus gives PRODIGY4.0 the power to improve system efficiency and correctness by removing redundant actions and responding to detrimental changes in the environment.

2.4 Alternative Approaches

ROGUE sends one action at a time for execution, and while the robot is executing, the planner is suspended, waiting for execution feedback. New goals can be incorporated at this time, but the planner does not reason about them.

An alternative design for ROGUE would have been to allow PRODIGY4.0 to continue planning while actions were being executed. Essentially, this approach would require

- explicitly monitoring for action completion, and then verifying its postconditions and attempting internal failure recovery,
- adding *significant* machinery to PRODIGY4.0 to support delayed action failures,
- setting (**automatically-decide-to-execute**) to return "t" when the execution of the previous action has completed,
- modifying PRODIGY4.0 to continue monitoring execution after a complete plan has been generated, instead of returning to the command line.

This approach would, for example, have allowed PRODIGY4.0 to create plans for each possible outcome of an action, without "wasting time" after an action fails. In domains with dangerous side-effects and no control rules to avoid them, PRODIGY4.0 would be able to simulate their effects before execution, thereby directly eliminating them from the plan. In this domain, however, dangerous side-effects are rare, and it takes very little time to plan a failure recovery. Hence, the effort needed to implement the basic requirements did not appear to have sufficient benefit.

A second alternative design for ROGUE would have been to send actions to Xavier without waiting for them to return. TCA would then maintain sequencing constraints amongst them. While this approach would have the same benefits as the first, it would require ROGUE

- to autonomously add failure monitors and internal recovery procedures to the TCA hierarchy each time an action is sent,
- to retract actions already sent to TCA when side-effects make them unnecessary, or failures require replanning or reordering of actions, or a compatible asynchronous request is received.

The added overhead of these two requirements reduce the benefits of an interleaved paradigm. Moreover, the learning mechanisms for improving plan execution (described in Chapter 4) would have much less benefit: they may reorder applicable operators at any time, and ROGUE would occur significant overhead to notify TCA.

2.5 Summary

ROGUE can successfully run errands between offices in our building.

This chapter has presented the integrated planning and execution aspect of ROGUE. We have described how PRODIGY4.0 gives ROGUE the power

- to integrate asynchronous requests,
- to prioritize goals,
- to suspend and reactivate tasks,
- to recognize compatible tasks and opportunistically achieve them,
- to execute actions in the real world, integrating new knowledge which may help planning, and
- to monitor and recover from failure.

ROGUE represents a successful integration of a classical artificial intelligence planner with a real mobile robot. The complete planning and execution cycle for tasks can be summarized as shown in Table 2.13. ROGUE handles multiple goals, interleaving the individual plans to maximize expected overall execution efficiency.

Figure 2.10 summarizes the information exchanged between users, PRODIGY4.0, and Xavier under ROGUE's mediation. ROGUE constrains PRODIGY4.0's decisions through

In Parallel:

1. ROGUE receives a task request from a user, and adds the information to PRODIGY4.0's state.

2. ROGUE requests a plan from PRODIGY4.0.

   Sequential loop; terminate when all top-level goals are satisfied:

   (a) Using up-to-date state information, PRODIGY4.0 generates a plan step, considering task priority, task compatibility and execution efficiency.
   (b) ROGUE translates and sends the planning step to Xavier.
   (c) ROGUE monitors execution and identifies goal status; in case of failure, it modifies PRODIGY4.0's state information.

3. ROGUE monitors the environment for exogenous events; when they occur, ROGUE updates PRODIGY4.0's state information.

Table 2.13: The complete planning and execution cycle in ROGUE. Note that Steps 1 to 3 execute in parallel.

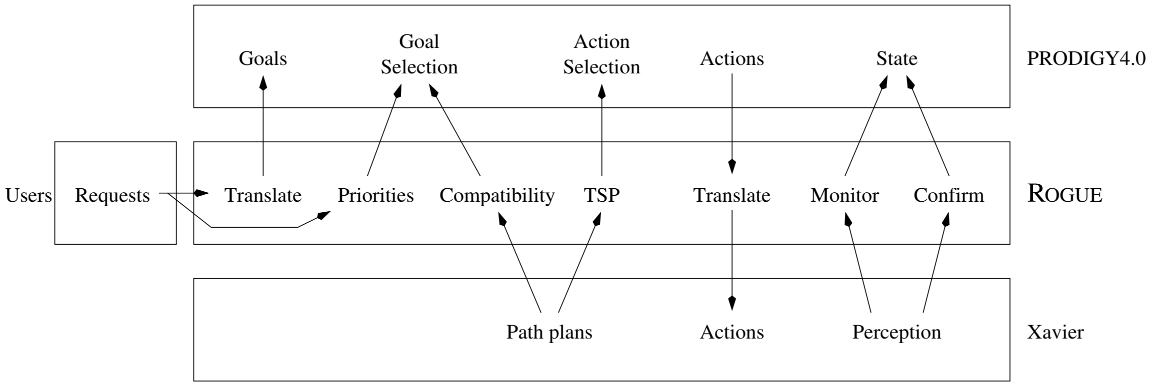


Figure 2.10: Summary of ROGUE's mediation between users, PRODIGY4.0 and Xavier.

calculations on task priority, task compatibility, and execution efficiency. ROGUE translates PRODIGY4.0's symbolic action descriptions into Xavier commands, and also translates Xavier's perception information into PRODIGY4.0 domain description.

The contributions of our work to the Xavier project are in the high-level reasoning parts of the system, allowing the robot to efficiently handle multiple, asynchronous interacting goals, and to effectively interleave planning and execution in a real world system. Execution monitoring based on a planning model allows the systematic identification of environment monitors: literals that appear as preconditions need to be monitored either in an environment monitor or in an action monitor; literals that appear as postconditions need to be verified in the action monitors.

Rogue advances the state of the art of the integration of planning and execution in robotic agent. In a unique novel way, Rogue is designed as the integration of two independently developed platforms. prodigy4.0 is a general-purpose planner and Xavier can be viewed as a general-purpose navigation robot. Rogue merges the functionality of these two systems in a real implementation that demonstrates the feasibility of connecting both systems in a rich task environment, namely the achievement of asynchronous user requests. (Rogue therefore also shows how the prodigy4.0 planner and the TCA approach in Xavier are in fact robust architectures.)

Strictly looking at Rogue only from the viewpoint of the integration of planning and execution, Rogue compares well with other integrated planning and execution systems such as NMRA [Pell *et al.*, 1997] and 3_T [Bonasso & Kortenkamp, 1996] (Section 5.1 discusses this comparison in more detail). Given the general-purpose character of the prodigy4.0 planner, Rogue could easily be applied to other executing platforms and tasks by a flexible change of prodigy4.0's specification of the domain.

Interleaving planning with execution enhances a deliberative robot system in numerous ways. One such benefit is that the system can sense the world to acquire necessary domain knowledge in order to continue planning. For example, it could actively ask directions, or passively use control rules to check whether it needs to recharge its batteries, or whether doors are open or closed. Another benefit is reduced planning effort because the system does not need to plan for all possible failure contingencies; instead, it can execute an action to find out its actual outcome.

The interleaved planning and execution portion of Rogue provides an appropriate platform to collect execution data for the learning portion.

Chapter 3

Learning for the Path Planner

The goal of learning in our system is to identify events ($\mathcal{E}$) during execution that do not meet expectations, and to then correlate situational features ($\mathcal{F}$) to those events. Events are then evaluated by a cost function ($\mathcal{C}$). The learner will then create a mapping from features and events to costs:

$$\mathcal{F} \times \mathcal{E} \rightarrow \mathcal{C}.$$

These *situation-dependent costs* are used to improve the quality and reliability of generated plans. The challenges are to automatically extract relevant information from the execution data in order to improve cost estimates, and to correlate that information to high-level features of the domain.

In this chapter, we present the learning algorithm as it applies to Xavier's path planner, where our concern is to improve the reliability and efficiency of selected paths. ROGUE demonstrates the ability to *learn situation-dependent costs* for the path planner's arcs. It extracts relevant training data from the massive, continual, probabilistic execution traces, and creates appropriate situation-dependent costs that the path planner can use to create more efficient paths. Figure 3.1 shows how our algorithm fits into the framework of the Xavier architecture.

The path planner uses a A* algorithm on a topological map that has additional metric information [Goodwin, 1996]. Knowledge in the path planner is represented as a topological map of the robot's navigation environment. The map is a graph with nodes and arcs representing office rooms, corridors, doors and lobbies. Learning appropriate arc-cost functions will allow the path planner to avoid troublesome areas of the environment when appropriate. Therefore we identify events, $\mathcal{E}$, for this planner as arc traversals, and costs, $\mathcal{C}$, as travel time

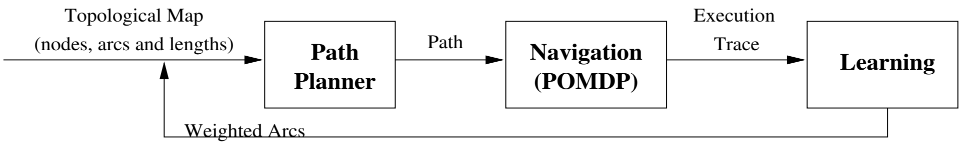


Figure 3.1: Learning for the path planner.

and position confidence. Features, $\mathcal{F}$, include both robot sensor data and high-level features such as date and goals.

The execution traces from which learning can occur are provided by the navigation module. Navigation is done using Partially Observable Markov Decision Process Models (POMDPs) [Simmons & Koenig, 1995]. The execution trace includes observed features of the environment as well as the probability distribution over the Markov states at each time step. Identifying the path planner's events from this trace is challenging because the execution traces contain a massive, continual stream of probabilistic data. At no point in the robot's execution does the robot know where it *actually* is. It maintains a probability distribution, making it more robust to sensor and actuator errors, but making the learning problem more complex because the training data is not guaranteed to be correct.

The primary challenges of our learning approach include:

- processing vast amounts of uncertain, continual navigation data,
- creating arc costs that depend on high-level features of the environment.

The approach is valid for *any* path planner paired with *any* navigation module. If Xavier were to directly plan paths within the POMDP, then Rogue would learn situation-dependent transition probabilities between Markov states. The important point is that Rogue must process the execution data to extract information relevant for planning, and then correlate that information with features of the domain. The designer must specify how to extract relevant learning opportunities from the execution data, and how to use the learned information within the planner.

We present the representations of the path planner and the navigation module in Section 3.1. In Section 3.2, we discuss features $\mathcal{F}$, including the characteristics of a good feature. In Section 3.3, we present events $\mathcal{E}$. We briefly describe the execution trace, and then describe how we extract and identify the robot's traversal from the uncertain data. Costs $\mathcal{C}$ are presented in Section 3.4, along with the mechanism Rogue uses to create an *events matrix* of training data for the learning algorithm.

In Section 3.5, we present the learning mechanism we use to create the mapping from situation features ($\mathcal{F}$) and arc traversals ($\mathcal{E}$) to arc costs ($\mathcal{C}$).

In Section 3.6, we briefly describe how the path planner uses these situation-dependent arc costs to create efficient paths. We present our experimental results in Section 3.7, and then finally summarize the main points of the chapter in Section 3.8. Related work can be found in Section 5.2.

3.1 Architecture and Representation

Our goal is to learn situation-dependent arc costs to improve the efficiency of constructed paths. The training data for this learning is provided by the navigation module.

The path planner uses an A* algorithm with an arc/node representation [Goodwin, 1996]. Navigation, meanwhile, uses a Markov state representation inside a Partially Observable Markov Decision Process (POMDP) model [Simmons & Koenig, 1995; Koenig, 1997].

There is, unfortunately, no clear correspondence between the representation used in the navigation module and the representation used in the path planning module. Hence, the representation gap between the path planner and navigation is one of the challenges in this research. More discussion on this point follows in Section 3.3.2 (page 69).

We now describe in detail the representations of the path planning and navigation modules.

3.1.1 The Path Planner

The path planner determines how to travel efficiently from one location to another. The environment is modelled as a topological map with nodes and arcs. *Nodes* represent junctions, such as those between corridors or at doors. *Arcs* represent connections between junctions. Topological arcs are augmented with length estimates.

Plans are generated using a decision-theoretic A* search strategy [Goodwin, 1996]. The path planner operates on the augmented topological map rather than using the POMDP model directly.[1]

The path planner creates a path with the best expected travel time. The travel time of a complete path is calculated as a function of four parameters: *distance*, *traversal weight*, *blockage probability* and *recovery costs*.

- The distance is an estimate of the straight-line length of the arc. It is an *estimate* because topological maps are not necessarily generated from building blue-prints: they may be hand sketched or learned.
- The traversal weight describes the difficulty of the route (e.g. door arcs are more expensive than corridor arcs).
- Blockage probability indicates the probability a given arc cannot be traversed (e.g. a closed door).
- Recovery costs estimate the difficulty of recovering from a failure, such as missing a turn or discovering a closed door. These costs estimate local recovery costs, i.e. for each missed turn.

Recovery costs are an important part of the expected time calculation. Actuator and sensor uncertainty means that the robot may not be able to accurately follow a path, and the shortest distance path is not necessarily the fastest. Consider, for example, the two paths from A to B shown in Figure 3.2. Although path 1 is shorter than path 2, the robot might miss the first turn on path 1 and have to backtrack. This problem cannot occur on the other path, since the end of the corridor prevents the robot from missing the turn. In this particular example, the recovery cost of path 1 is very high, and so the path planner determines that the longer path might take less time on average.

[1] It is infeasible to determine optimal POMDP solutions given our real-time constraints and the size of our state spaces (over 3000 states for the map shown in Figure 1.4, page 11) [Cassandra *et al.*, 1994; Lovejoy, 1991]. Reasoning about blockage probabilities and recovery costs is also notably easier in the topological map.

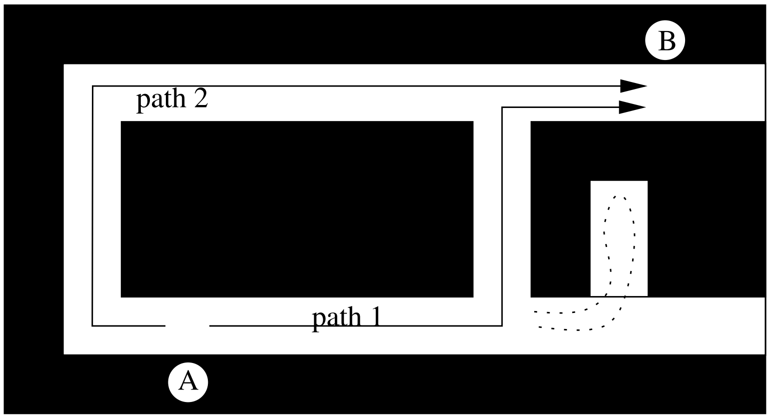


Figure 3.2: Two paths from A to B. Although path 1 is shorter, the robot might miss the turn and get trapped in the dead end. Reproduced from Simmons *et al.* [1997].

Xavier currently travels in a restricted environment, namely three of the floors in our office building. The weights of the topological map of this environment have been hand-tuned and provide a good initial approximation of the unoccupied environment. However, these default costs do not capture the variations created by human use. The patterns describing these variations can be detected.

ROGUE learns traversal weights (or *costs*) that depend on high-level features of the situation. These learned weights effectively modify the estimated traversal time to reflect experienced traversal time. Learning situation-dependent costs will allow the path planner to respond to patterns and changes in the environment.

3.1.2 Navigation

Navigation on the robot is done using Partially Observable Markov Decision Process models (POMDPs) [Simmons & Koenig, 1995]. The navigation module estimates the robot's current location, determines the direction the robot should be heading at that location to follow the path, and then sets a directional heading.

The navigation module estimates the robot's current location by maintaining a probability distribution over the robot's current *pose* (position and orientation). Given the current pose distribution and new sensor information, the navigation module uses Bayes' rule to update the pose distribution. The updated probabilities are based on probabilistic models of the actuators, sensors, and the environment. In Xavier, the primary actuators are the wheels, for which the probabilistic models describe the robot's dead-reckoning skills. Xavier's primary sensors are its sonars, whose probabilistic models describe the likelihood of observing given features in the sonar data. The environment is the map, where the models describe variance on its metric information. This information is automatically compiled into a POMDP model.

The metric variance of the map alters the structure of the Markov model. In our system,

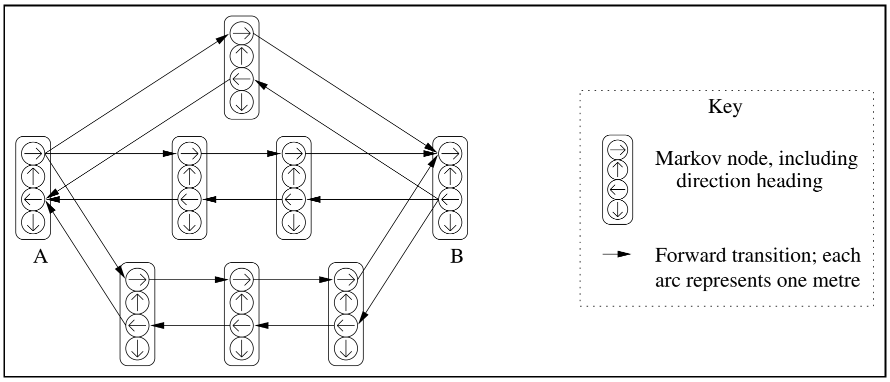


Figure 3.3: Corridor representation which captures length uncertainty for the navigation module. Each transition corresponds to 1 metre, and hence this corridor is represented as being 2, 3 or 4 metres long. Only *forward* transitions are marked. Reproduced from Simmons & Koenig [1995].

we use *parallel Markov chains*, where each corresponds to one of the possible lengths of the edge [Simmons & Koenig, 1995]. Figure 3.3 illustrates an example for a corridor that may be two, three or four metres long. This representation is an effective way to represent worlds in which lengths are not known with certainty.

Table 3.1 shows the probability update calculation. Figure 3.4 shows an example of how Bayes' rule is used to update state probabilities (for a *forward* action, disregarding observations). At time t, states $s_1, ..., s_4$ have the marked probabilities, and for a given action, the marked transition probabilities to $s_5, ..., s_8$. Denote $\pi(s_i, t)$ to be the probability of state i at time t; denote $A_a(s_i, s_j)$ to be the transition probability between s_i and s_j for a given action a; denote $\mathcal{O}(s_i, o_t)$ to be the probability of observing o_t in state s_i. At time

Define S to be the set of all Markov states; Let $s, s' \in S$.
Define A to be the probability distribution over successor states; $A_a(s, s')$ is the transition probability for an action a between state s and state s'.
Define $\mathcal{O}$ to be the probability distribution over observations; $\mathcal{O}(s, o)$ is then the probability of obverving o in state s.
Define π to be the probability distribution over S; $\pi(s, t)$ is then the probability of the robot being in state s at time t. (Technically, $\pi(s, t)$ is shorthand for $\pi(s, t \mid o_0, ..., o_t, a_0, ..., a_{t-1}, \pi(s, 0))$ for the observation sequence $o_0, ..., o_t$ and the action sequence $a_0, ..., a_{t-1}$.)

At time $t = 0$:
$\forall s \in S$, let $\pi(s, 0) =$ initial state distribution.

For time $t + 1 \geq 1$, action a was selected, and then observation o_{t+1} was made:
$\forall s' \in S$, $\pi(s', t+1) = \sum_{s \in S} \pi(s, t) \times A_a(s, s') \times \mathcal{O}(s', o_{t+1})$.

Table 3.1: Bayesian probability updates.

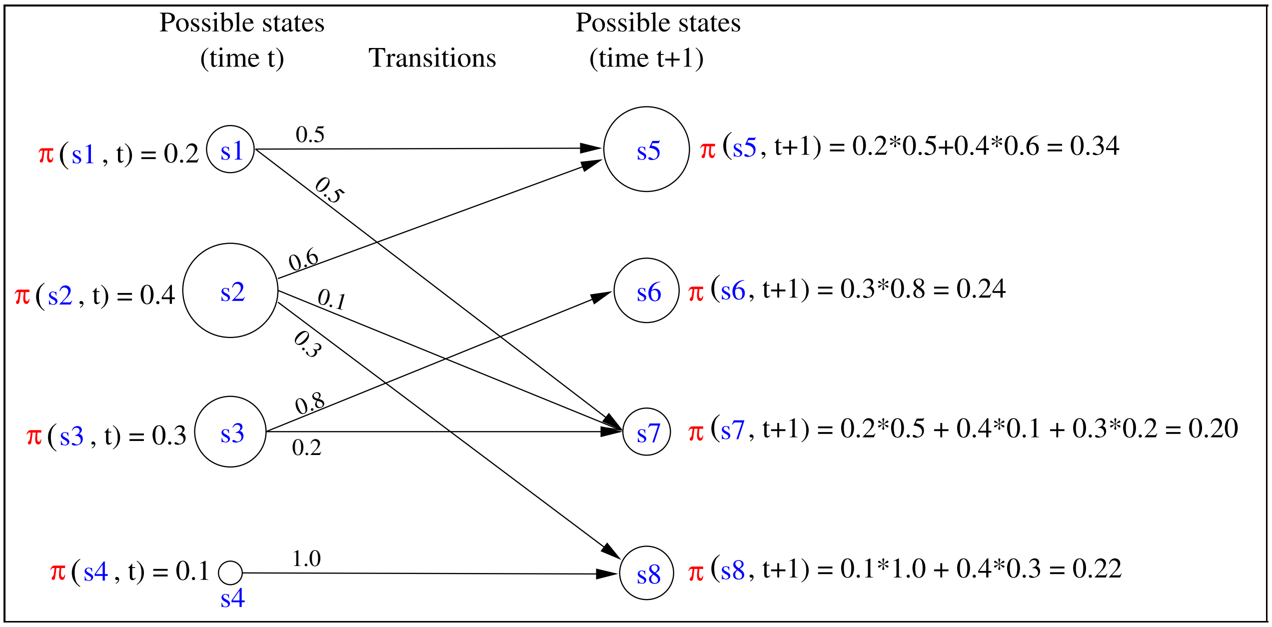


Figure 3.4: An example of POMDP transition calculations (for a *forward* action, disregarding observations). $\pi(s_i, t)$ indicates the probability of the state (circle size is proportional to probability). At time $t+1$, POMDP state probabilities are calculated as the sum of all incoming transitions.

$t+1$, for a given action a, the POMDP's Bayesian probabilities are calculated as:

$$\pi(s_j, t+1) = \sum_i \pi(s_i, t) \times A_a(s_i, s_j) \times \mathcal{O}(s_j, o_{t+1}). \tag{3.1}$$

Each of the states at time $t+1$ has updated probabilities that are calculated as the sum of all incoming transitions.

Observations of the world help prune unlikely states from the probability distribution. Observations can help prune unlikely states because a low probability observation will make a low probability state essentially impossible[2], while a high probability observation will improve confidence in medium or high probability states. Table 3.2 shows some sample observation probabilities; note that none of the observation probabilities are zero. In general there is significant probability of sensor error; however it is very unlikely the robot will "hallucinate" the end-of-corridor (`EOC`).

Regular observations can keep the robot fairly certain of its location. However, if the robot does not receive any observations for a long time (e.g. in a long featureless corridor), the probability distribution may spread over many states, making it impossible to determine with any precision the robot's exact location.

Note that a new observation may significantly change the probability distribution. For example, when the robot observes the end of a corridor, that state is extremely likely. At the

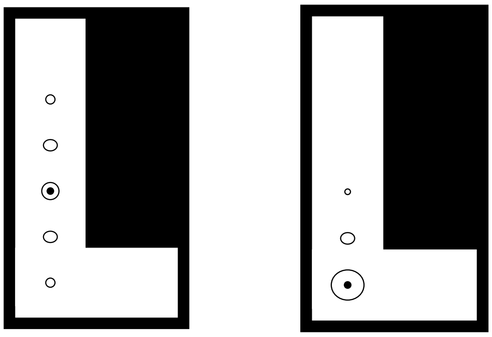
[2]In the implemented algorithm, all states with less than 10^{-9} probability are reset to zero.

$P($ EOC | Wall $) = 0.98$
$P($ Nothing | Wall $) = 0.02$
$P($ EOC | Open $) = 0.15$
$P($ Nothing | Open $) = 0.85$
$P($ Wall | Wall $) = 0.65$
$P($ SmallOpening | Wall $) = 0.10$
$P($ MediumOpening | Wall $) = 0.10$
$P($ LargeOpening | Wall $) = 0.10$
$P($ Nothing | Wall $) = 0.05$

Table 3.2: Sample observation probabilities. $P(o|s) = \mathcal{O}(s, o)$, which is the probability that the sensor will report o given that the state is of type s.

previous time step, however, the robot might have had a very poor estimate of its location, in which the probability distribution was very flat and centred some distance from the end of the corridor. Figure 3.5 demonstrates this change. Figure 3.5a shows the probability distribution before the robot sees the wall at the end of the corridor, while Figure 3.5b shows the distribution after.

The navigation module is very reactive to unexpected sensor reports, since probabilities are maintained for *all* possible poses, not just the most likely pose. Thus, if the robot strays from the nominal path, it will automatically execute corrective actions once it realizes its mistake. Consequently, the navigation module can gracefully recover from sensor noise and misjudgments about landmarks. The drawback to this approach is that the robot is never *completely* sure where it is. This introduces uncertainty into the learning data, and therefore further complicates the learning algorithm because the training data may contain errors.



(a) (b)

Figure 3.5: Markov state probability distribution, (a) before and (b) after observing the wall at the end of the corridor. Circles indicate probability distribution; large circles have high probability. At each time step, the most likely state is marked with a dot.

3.2 Features

Features, $\mathcal{F}$, of the environment are used to discriminate between different learning events. It is crucial to find a good set of relevant features, since the hypothesis space can only be described in terms of the available features. If critical features are omitted, then the learner will be unable to converge on the correct target function. It is an important open problem to autonomously determine a good set of features.

Features are defined by the robot architecture and the environment. Usually they are not dependent on the tasks. For this reason, the execution module defines and collects features.

Features available in Xavier include speed, time of day, sonar observations (walls, openings), camera images (which could also be abstracted to indicate "empty," "crowded," "cluttered," etc.), other goals, and the desired route. For example, travelling too fast past a particular intersection might lead to missing a turn. Images with lots of people might indicate difficult navigation.

Characteristics that make a good feature include:

- it is easy to detect (in terms of accessibility and cost),
- it is informative, and
- it is projective.

Easy detection is important because features are recorded frequently, usually once per time step in the trace. The system cannot spend most of its time calculating and recording feature values, nor can it spend all its time gathering the information before making decisions. We briefly explore the effect of feature costs on learning in Section 4.6.3.

Informative features are ones that contain information relevant to learning. A good learning system will be able to prune out irrelevant features, but we do not want the system expending effort to collect data that will later be ignored.

By a "projective" feature, we mean one for which the gathered information at one moment can help the system make decisions about the future. Usually these features are "high-level," that is, they *do not* depend exclusively on execution. For example, a feature like time can be easily projected into the future. Similarly, a feature such as the goal location will not change for the duration of the task. "Robot-level" features can be projective when we can control them; for example, the robot's speed can affect the reliability of navigation because the robot misses fewer openings and travels more smoothly.

Most execution-level features, such as sonar readings or images, are not usually projective because what the system sees *now* may have little or no bearing on what it sees in the *future*. It is not often the case that current sonar readings relate to future sonar readings at a different location.

There are rare cases in which execution-level features may be projective. For example, if the robot saw many people in the lobby, it could predict that in a few minutes, the classroom corridor would be crowded and that it should avoid tasks in that corridor for a little while. It would be possible to use camera images, $\mathcal{I}$, from all locations, $\mathcal{L}$, at all times, t, as features. However, learning this correlation would be extremely computationally expensive, and therefore we do not store this information.

There are also features which may be projective with respect to *execution*, but not projective with respect to *planning*, such as travel direction. Travel direction can have direct impact on the cost of an arc; for example, an arc near a corridor intersection may be very expensive when making a turn, but when travelling straight from within the corridor, may be much cheaper. Travel direction, however, cannot be predicted before planning, and hence the path planner needs to carefully consider each route.

For the experiments in this chapter, we only use the high-level features such as time of day, route and other goals, along with execution-level features we can control such as the robot's speed. We incorporate sonar readings as one of the features for the learning in Chapter 4, where the current reading (whether or not a door is open) affects the next immediate decision.

3.3 Events

Events ($\mathcal{E}$) in any planner can be identified by asking the question: "*What will change the planner's behaviour?*" In Rogue, we would like the path planner to predict and avoid areas of the environment which are difficult to navigate (and similarly, exploit areas that are easy to navigate). Improved cost estimates on arcs will cause the path planner to select more appropriate plans. Learning events are therefore arc traversals that do not meet expectations.

The available execution data is generated by the navigation module, and is therefore stored using the probability distribution over Markov states. Our algorithm examines the execution trace, identifies the most likely path that the robot traversed, and then identifies the corresponding path planner arcs. It then maps situational features to the arc traversals to create situation-dependent costs.

An *execution trace* from the robot includes:

- the features describing the situation,
- the sequence of actions executed by the robot, and
- the probability distribution over the Markov states at each time step.

In particular, an execution trace does *not* include arc traversals. We therefore need to extract the traversed arc sequence from the Markov state distributions. The steps in this process are:

1. Identify the robot's most likely traversed sequence through the Markov states.
2. Calculate the most likely traversed sequence through the path planner's arcs.

The POMDP navigation module keeps track of the *most likely states* but not the *most likely sequence of states*. Viterbi's algorithm is guaranteed to find the single best state sequence with the highest probability, given the actions, observations and initial state distribution [Rabiner & Juang, 1986]. However, Viterbi's algorithm *was not* designed for use in a

Markov model that represents uncertain length information. We extend Viterbi's algorithm to compensate for this uncertainty, giving us a powerful way to identify likely paths through the environment.

Once these likely state sequences have been identified, we then need to identify the corresponding arc sequences. The environment representations used by the navigation module and the path planner are different enough that the mapping is not direct.

Finally, once the arc sequences have been identified, Rogue can calculate cost estimates for the arcs, and then correlate those costs with the available features, thereby creating situation-dependent arc costs.

This process can be described pictorially as in Figure 3.6. As the robot wanders down the corridor, it sees doors at time steps 6 and 8. The Markov state distribution changes as shown. In order to modify the arc cost estimates for the path planner, Rogue needs to determine which arcs the robot travelled, and for how long.

Below, we describe the workings of Viterbi's algorithm and our extension of it. We then present the techniques used to calculate the arc sequence so that arc traversal events can be identified.

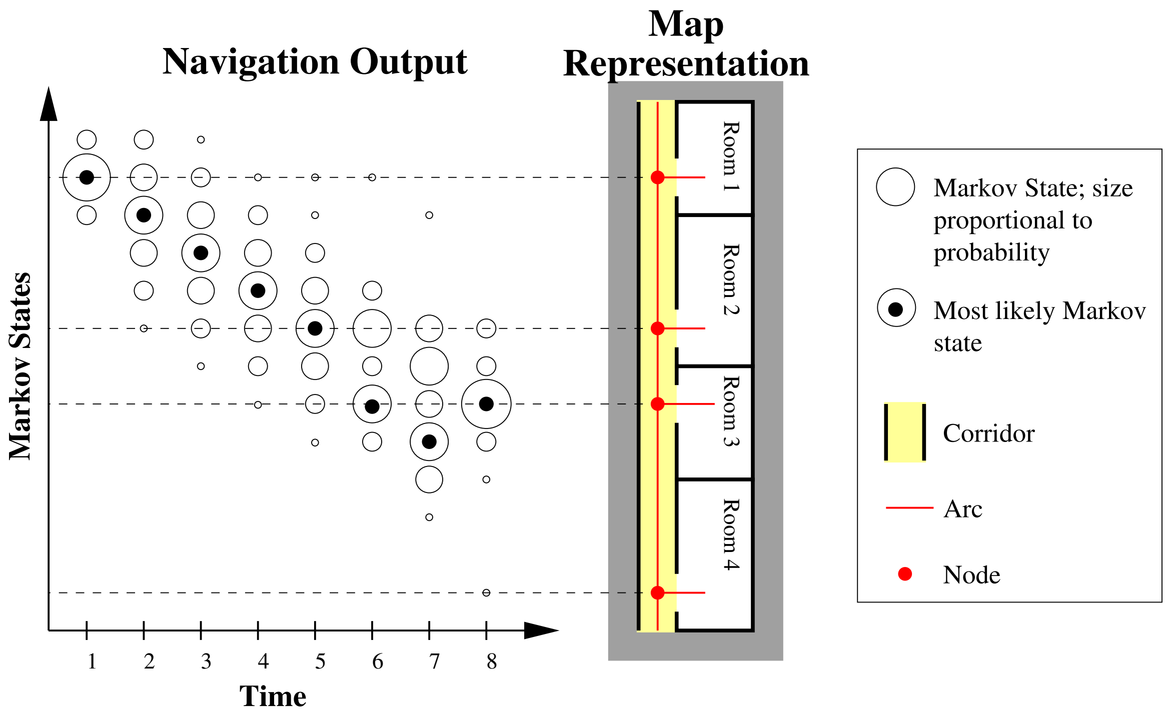


Figure 3.6: Extracting arc traversals from Markov state distributions.

3.3.1 Identifying the Most Likely Traversed Markov Sequence

Since the robot does not know where it is at any given moment, it consequently cannot identify with certainty its path. In order to reconstruct the arc traversal sequence, we must first reconstruct the Markov state traversal sequence.

The algorithm to calculate this sequence is known as Viterbi's algorithm [Rabiner & Juang, 1986]. The algorithm is reproduced in full in Table 3.3. In step 1, variables are initialized. In step 2, Viterbi's algorithm maintains an estimate of which state the robot was in at the previous time step, for each possible state. In step 3, the algorithm calculates the complete Viterbi sequence by recursing backwards through time.

Define S to be the set of all markov states; $s, s' \in S$.
Define A to be the probability distribution over successor states; $A_a(s, s')$ is the transition probability for an action a between state s and state s'.
Define $\mathcal{O}$ to be the probability distribution over observations; $\mathcal{O}(s, o)$ is the probability of obverving o in state s.
Define π to be the POMDP probability distribuion over S; $\pi(s, t)$ is the probability of the robot being in state s at time t.
Define δ to be the Viterbi probability distribution over S; $\delta(s, t)$ is the probability of the sequence ending at s at time t.
Define $\Psi(s, t)$ to be the unique state from time $t-1$ that most likely leads to state s.
Define Seq_T to be the most likely sequence generated from time T; $s = Seq_T(t)$ is the state at time t in Seq_T.

1. At time $t = 0$:
   $\forall s \in S$, let $\delta(s, 0) =$ initial state distribution $= \pi(s, 0)$
   let $\Psi(s, 0) =$ NULL

2. For time $t + 1 \geq 1$, action a was selected, and observation o_{t+1} was made:
   $\forall s \in S$, $\Psi(s, t+1) = s'$ such that s' gives $\mathrm{MAX}_{\forall s' \in S}\left[\delta(s', t) \times A_a(s', s)\right]$.
   $\delta(s, t+1) = \frac{1}{k}\delta(\Psi(s, t+1), t) \times A_a(\Psi(s, t+1), s') \times \mathcal{O}(s, o_{t+1})$.
   where k is a normalization factor.

3. To calculate the most likely sequence at time T, Seq_T:
   $Seq_T(T) = s$ such that s gives $\mathrm{MAX}_{\forall s \in S}\left[\delta(s, T)\right]$,
   i.e. the most likely Viterbi state at time T.
   $\forall t, 0 \leq t < T$ $Seq_T(t) = \Psi(Seq_T(t+1), t+1)$.

Table 3.3: Viterbi's Algorithm, reproduced from Rabiner & Juang [1986].

Viterbi's algorithm is a slight modification to the standard POMDP algorithm used for navigation. The primary difference is that:

the POMDP algorithm calculates the most likely *states*, while
Viterbi's algorithm calculates the most likely *state sequence*.

The two may differ, for example, when there are multiple parallel corridors that the robot may have travelled down. Consider Figure 3.7, where the robot travelled from X to Y, along either path 1 or 2. When the robot nears Y, the most likely *states* reflect the possibility of having arrived along *either* route, while the most likely *state sequence* is only *one of* the two routes.

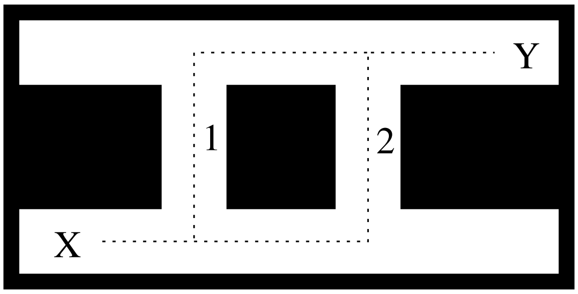


Figure 3.7: A map showing why the most likely state sequence may be different from the most likely states.

The POMDP algorithm is well-suited to most robotics tasks because it is very important for the robot to have a good idea where it is. Viterbi's algorithm, on the other hand, is more commonly used in applications where the whole sequence is needed. For example, it is widely used in speech recognition, where the most likely sentence is desired, rather than simply the most likely last word.

For our robot learning application, we need the complete path of the robot, and hence use Viterbi's algorithm. Viterbi's algorithm, however, was not designed for use when the desired trajectory is actually an *abstraction* of the Markov states. Our models represent length uncertainty, and hence we need an estimate of the trajectory that integrates over the length variable. We extend Viterbi's algorithm to compensate for this representation difference.

Mathematically, the POMDP algorithm calculates the transition probability as a *sum* of the probabilities on connecting states, that is, looking at *all possible* ways of arriving at a particular state. Viterbi's algorithm, on the other hand, finds the *single most likely* prior state, so as to reconstruct a path. (Note that Viterbi's algorithm does not use π, the standard POMDP state probability distribution, but instead uses δ, the probability of the sequence.)

Figure 3.8 illustrates the difference between the standard POMDP calculations and the calculations in Viterbi's algorithm. In this figure, the $\delta(s, t = 0)$ probabilities equal the $\pi(s, 0)$ probabilities of Figure 3.4, and the transition probabilities, A, are also the same.

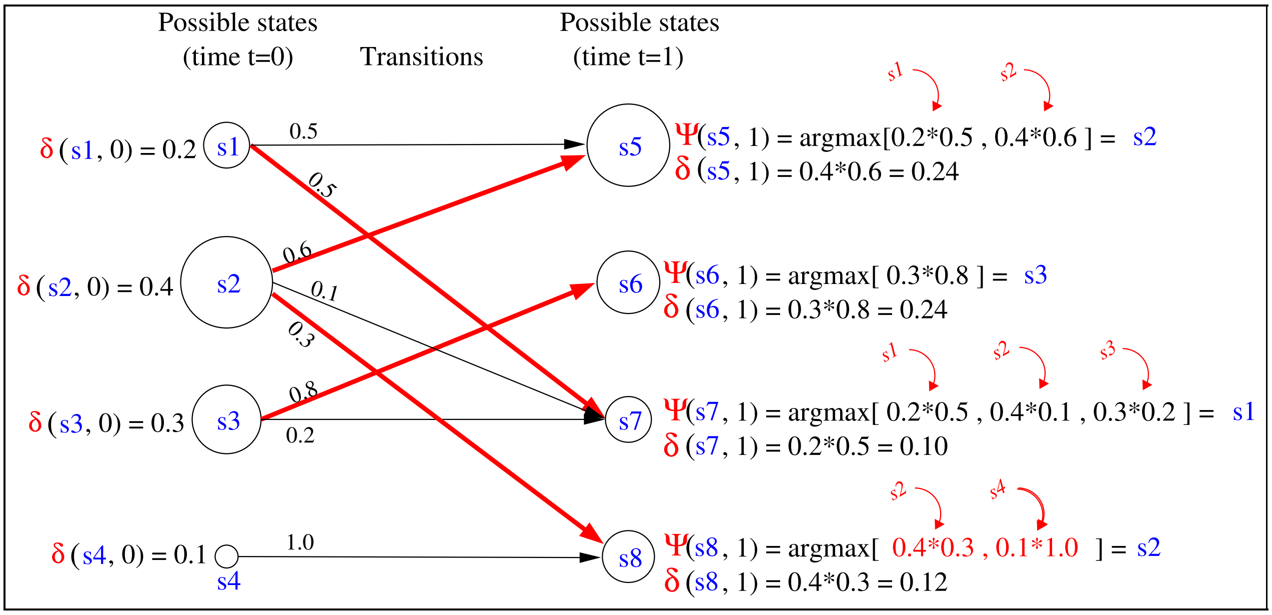


Figure 3.8: Viterbi transition calculations (for a *forward* action, disregarding observations). $\delta(s_i, t)$ indicates the Viterbi probability of the sequence ending in s_i at time t (circle size is proportional to probability); $\Psi(s_i, t)$ indicates the most likely prior state (thick line shows the selected transition). At time $t+1$, Viterbi sequence probabilities are calculated as the most likely prior sequence times the transition probability (and then normalized).

Recall that POMDP probabilities are calculated as shown in equation 3.1 (page 56). Viterbi's algorithm, meanwhile, maintains the probability distribution of the sequence, δ, calculated as:

$$\delta(s_j, t+1) = \frac{1}{k}\delta(\Psi(s_j, t+1), t) \times A_a(\Psi(s_j, t+1), s_j) \times \mathcal{O}(s_j, o_{t+1}), \tag{3.2}$$

where k is a normalization factor[3] and $\Psi(s_j, t+1)$ is the most likely sequence at time $t+1$. $\Psi(s_j, t+1)$ is calculated from the transition probability and the probability of the most likely sequence at time t:

$$\begin{aligned} \Psi(s_j, t+1) &= s_i \text{ such that } s_i \text{ gives } \mathrm{MAX}_{\forall s_i \in S}\left[\delta(s_i, t) \times A_a(s_i, s_j)\right] \\ &= \mathrm{ARGMAX}_{\forall s_i \in S}\left[\delta(s_i, t) \times A_a(s_i, s_j)\right]. \end{aligned} \tag{3.3}$$

Viterbi's algorithm finds the sequence at time t that contributed the most probability to the sequence at time $t+1$. For example, the most likely prior state for state s_7 is s_1, because s_1 contributed 0.10 (0.2×0.5) probability, while s_2 contributed 0.04, and s_3 contributed 0.06. Note that, in hind-sight, Viterbi's algorithm eliminates the possibility that the robot was in state s_4 at time t, while the two paths it generates from states s_5 and s_8 converge, both passing through s_2.

[3] If k is not used, δ reflects the exact probability of the sequence; however, round-off error causes serious miscalculations when these numbers become very small.

3.3.1.1 Problems with the Viterbi Sequence

Viterbi's algorithm is guaranteed to find the most likely sequence of Markov states [Rabiner & Juang, 1986]. However, the Markov models we use for robot navigation differ from standard Markov models used by speech systems: we represent length uncertainty.

This representation change leads to a serious problem that needs to be addressed before using the information in our learning algorithm: **the fact that a given node may "fan-out" leads to information loss and a poor estimate of the best path.** Essentially, we want the algorithm to identify the robot's trajectory in the topological map, which is an abstract representation of the Markov model. The fan-in/fan-out representation of the model effectively captures the length uncertainty of the environment, but Viterbi's algorithm is unable generate a good estimate of the abstracted trajectory.

For example, consider Figure 3.9. Because node $s1$ splits into three parallel Markov chains, while the lower probability state, $s2$, splits into two, Viterbi's algorithm selects $s2$ as the most likely previous state for $s3$. In a Markov model that does *not* represent length uncertainty, Viterbi's algorithm would correctly identify $s1$ as the more likely previous state.

Consider the reverse situation, shown in Figure 3.10, in which one outgoing arc has a greater weight than other outgoing arcs, such as when a node is connected to a door. Although it is clear that the robot travelled to $s2$ rather than $s3$, Viterbi's algorithm selects $s2$ as the most likely generating state. In this situation, since room states have high-probability self-transitions, Viterbi's algorithm will very often never correct itself, instead claiming that

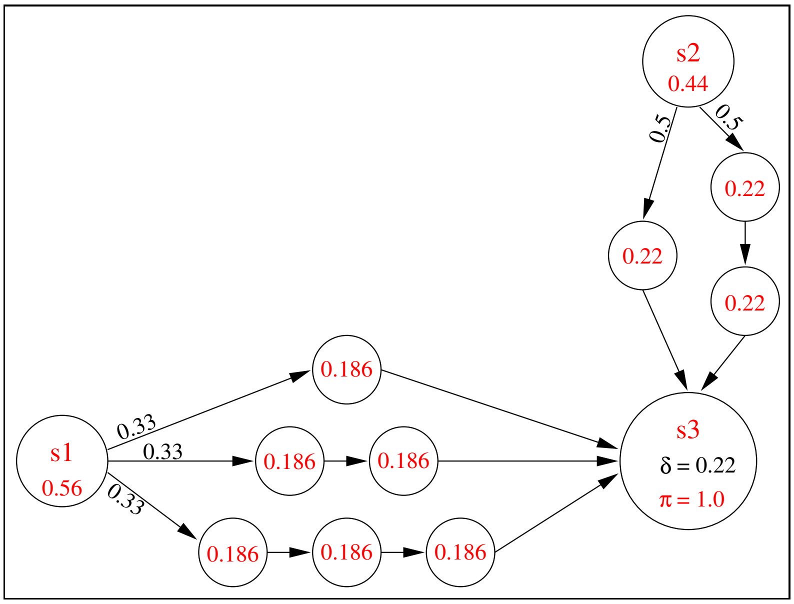


Figure 3.9: Fan-in: Example of how the map representation affects Viterbi's algorithm. Although it is more likely that the robot passed through $s1$, the Viterbi sequence generated from $s3$ passes through $s2$ instead.

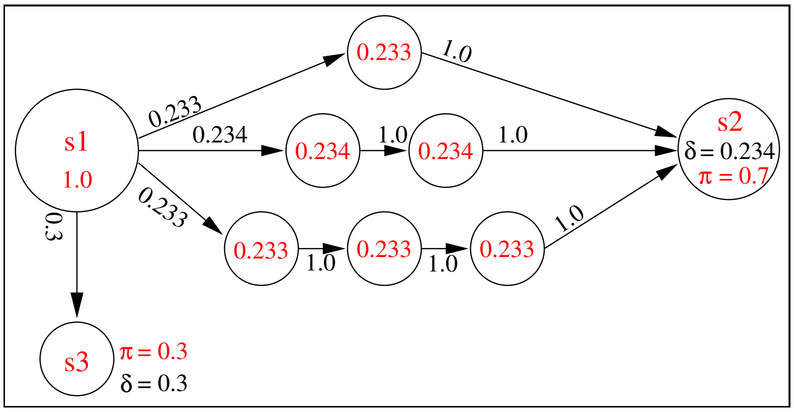


Figure 3.10: Fan-out: Example of how the map representation affects Viterbi's algorithm. Although $s2$ has a greater π probability than $s3$, Viterbi's algorithm selects $s3$ as the sequence-generating state.

the robot's most likely path was only within the room.

The problem continues to compound so that after a long execution run, Viterbi's algorithm selects sequences that are *extremely* unlikely according to the standard POMDP calculations. In fact, in most cases, the final state in the most likely sequence did not even appear in the list of possible POMDP states π, which prunes out extremely low probability states, i.e. $\nexists s \in S$ such that $\pi(s,T) > 0$ and $s = Seq_T(T) = \text{ARGMAX}_{\forall s' \in S}\left[\delta(s',t)\right]$..

Table 3.4 shows a small example of the π and δ distributions. These distributions were collected approximately 4 minutes (70 time steps) after starting the robot (recall that at $t = 0$, the π and δ distributions are identical). The maximum δ state, number 86, has no π probability because the POMDP prunes out low probability states. In fact, the top *five* δ states have zero probability in the π distribution.[4] The set of states for which $\delta(s,t) > 0$ is always a superset of the states for which $\pi(s,t) > 0$.

3.3.1.2 Possible Modifications to Viterbi's Algorithm

Recall that our primary goal is to evaluate arc traversals. In order to do that, we need to determine the best estimate for the robot's trajectory. Ideally, we would like Viterbi's algorithm to correctly identify the robot's trajectory in the *topological map*, rather than directly in the Markov model. Essentially, Viterbi's algorithm would have to identify a fan-in situation, and correctly sum probabilities over those edges. However, our Markov model representation does not lend itself to easy detection of these situations (see Section 6.3.2), and so we instead use an approximate method.

Viterbi from $\pi(s_{max}, T)$: Our first modification to Viterbi's algorithm is to use the most likely POMDP state as the sequence generator. We know that the π distribution is always

[4]Recall that all states with less than 10^{-9} probability are reset to zero.

	δ			π	
State	Probability	Rank	State	Probability	Rank
86	0.4422119	(1)			
94	0.0796982	(5)			
170	0.0148035	(8)			
174	0.1639655	(3)			
270	0.0159902	(7)	270	0.4202660	(1)
318	0.0060332	(11)			
498	0.0796982	(4)			
510	0.1639655	(2)			
698	0.0063961	(9)	698	0.1344851	(3)
702	0.0183418	(6)	702	0.3856558	(2)
902	0.0000153	(14)	902	0.0001228	(5)
906	0.0000200	(13)	906	0.0000555	(6)
1914	0.0060332	(10)			
1918	0.0000050	(15)	1918	0.0000831	(7)
2058	0.0028218	(12)	2058	0.0593317	(4)

Table 3.4: Small example of δ and π probability distributions, collected after four minutes of execution.

a better estimate of the robot's current location than the δ distribution, since these probabilities are based on *all possible ways* of reaching a given state. In other words, instead of using the default generating state

$$Seq_{T}(T) = \text{ARGMAX}_{\forall s \in S}\left[\delta(s,T)\right] \tag{3.4}$$

(the most likely Viterbi state at time T), we use

$$Seq_{T}(T) = \text{ARGMAX}_{\forall s \in S}\left[\pi(s,T)\right] \tag{3.5}$$

(the most likely POMDP state at time T). Effectively, this change forces Viterbi's algorithm to use the POMDP position estimate as an "oracle" of the final state. The intuitive justification for this change is that if the final state selected sequence has a high π probability, then the generated sequence is more likely to reflect the actual traversal sequence. In speech recognition, for example, this modification would be equivalent to having a good estimate of the last word of the sentence; instead of calculating the most likely sentence, $P(s)$, we calculate the most likely sentence given the most likely last word, $P(s|w)$.

Although the Viterbi sequence generated from $\pi(s_{max}, T)$ improves the sequence estimate, it still falls short in two ways. First, it can still be mislead by the fan-out problem described above. Second, the most likely π state at the end of the trace might have a low probability, such as if the robot stopped in the middle of a long corridor, or in the example shown in Table 3.4, where the best two states have similar probabilities. Hence, the sequence generated from this state might not be as reliable as we would like.

Viterbi from $\text{MAX}[\pi(s,t)]$: To solve the low probability problem, we could use the Viterbi sequence generated from the highest probability recent POMDP state. That is, instead of using equation 3.5 we select $t \leq T$ such that

$$Seq_t(t) = \text{ARGMAX}_{\forall s \in S, \forall t \leq T}\left[\pi(s,t)\right]. \tag{3.6}$$

We would then ignore all data between t and T. This sequence more probably reflects the actual sequence, but may still suffers from the fan-out problem above, in which only the most likely Viterbi sequence is identified. This method also suffers in that t may be much smaller than T and hence much data is lost, or that a threshold must be set for the smallest value of t.

Multiple Viterbi from $\forall s, [\pi(s,T)]$: Another alternative method is to use the *set* of Viterbi sequences generated from the complete set of POMDP states:

$$\forall s \in S,\ [Seq_{s,T}(T) = \pi(s,T)] \tag{3.7}$$

Each sequence could then be weighted for the learning algorithm by the probability of the generating state. Figure 3.11 illustrates the concept: each state in the final π distribution is used as a generating state; there will then be several Viterbi sequences used for learning. This method eliminates captures the entire probability distribution at the last time step. It therefore covers more of the likely paths. However, this method still suffers from the fan-out problem above.

3.3.1.3 Multi/Markov Viterbi

Each of the possible extensions described above solves a small part of the problem. By using the best π state as the sequence-generating state, the sequence is more likely to reflect the robot's actual trajectory. If we set a threshold for that state's probability, we are even more confident in the generated sequence. By using multiple sequences, we again improve odds that the actual sequence will be captured. However, we still need to solve the ambiguity problem raised by the fan-out representation.

ROGUE uses the Viterbi sequences generated from *many* high probability states *throughout* the trace:

$$\begin{aligned} \forall t \leq T, \qquad & Seq_t(t) = \text{ARGMAX}_{\forall s \in S}\left[\pi(s,T)\right] \\ \wedge \quad & Seq_t(t) > \tau, \end{aligned} \tag{3.8}$$

where τ is a threshold to select high probability sequences and eliminate low probability ones. We call the modified algorithm *Multi/Markov Viterbi* because we use *multiple* trajectories generated from the most likely POMDP (*Markov* state. By using many sequences, Multi/Markov Viterbi collects evidence for the most likely actual trajectory, and thereby compensates for the poor estimates made by Viterbi's algorithm.

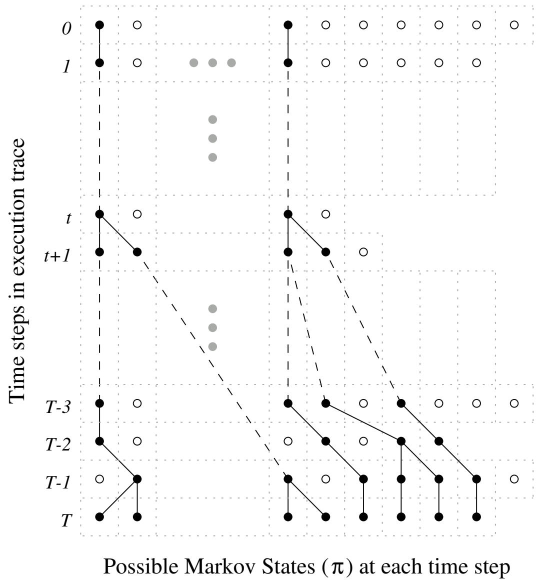


Figure 3.11: The set of last sequences: Viterbi sequences generated from each of the possible Markov states at the last time step in the execution trace, T. Each table entry contains a Markov state with probability > 0.0; connections between entries indicate the Viterbi sequence. (Note that the number of possible states may vary, and that not all possible states were believed possible in hind-sight (light circles). If there are time steps with very few or highly confident states, most sequences will include that node, and converge for all earlier time steps, such as at time step t.)

Consider the probabilities and transitions shown in Figure 3.9 (page 64). Unmodified, Viterbi's algorithm would generate a sequence passing from $s3$ through $s2$ to the initial state. Our modified Viterbi's algorithm uses that path *as well as* the sequence generated from $s1$. By using both sequences, ROGUE is more likely to capture the robot's actual traversal sequence.

For a second example, consider the map shown in Figure 3.12. Imagine that the robot travels up one of the central corridors, and then turns right towards point C. Assume the

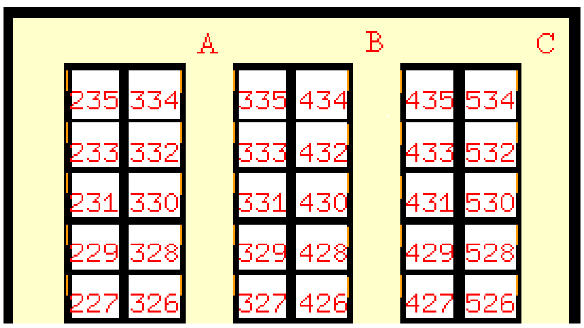


Figure 3.12: Map used in the example of how multiple sequences are used.

robot initially believes it is heading towards point A, in the "300" corridor. Because of position uncertainty, it might be in the "400" corridor, heading towards point B. When the sonars detect a wall in front of the robot, the robot becomes very certain that it has arrived at the end of the corridor. The probability masses around points A and B. Point A has a higher probability, say 0.60, while point B is 0.30 and other places with the remaining 0.10. The sequence generated at this moment (from point A) is then used for learning. Later in the episode, the robot arrives at point C with 0.90 probability. The Viterbi sequence generated from here shows that it is more likely that the robot travelled up the "400" corridor, going through point B. This second sequence is *also* used for learning. Neither of the two sequences is necessarily correct: imagine that the robot had not reached point C, but instead that an obstacle had been placed in the corridor directly above room 435, which the robot believed to be the end of the corridor. If the trace had ended at this point, and ROGUE only used the second sequence for learning, the system would learn incorrectly. Using both sequences allows ROGUE to cover both possibilities.

By recording each of these multiple sequences as training data for the learner, ROGUE is in some sense "hedging its bets." It knows that the robot traversed only one unique path through the environment, but it does not know *which*. By recording all possibilities, ROGUE gathers a body of evidence that collectively captures the robot's actual path.

In the cases that a later sequence subsumes an earlier sequence, the earlier sequence has more evidence of being correct. Throughout an execution trace, an early sequence may acquire a substantial amount of corroborating evidence. Moreover, since arc sequences are generalizations of Markov sequences, minor variations in the Markov sequence will appear as minor variations in time estimates of the arcs. It is then the responsibility of the learning algorithm to generalize the data, by grouping similar data and eliminating noise. Enough evidence of the correct path will allow ROGUE to learn situation-dependent rules that correctly reflect the dynamics of the environment.

To summarize, Viterbi's algorithm finds the most likely sequence of Markov states that the robot traversed. However, we need the most likely trajectory in the *topological map*, rather than the most likely trajectory in the Markov model. Since our Markov models represent length uncertainty, Viterbi's algorithm can become misled by the fan-out/fan-in nature of the representation. To get a good estimate of the robot's actual state sequence, we use the most likely π state as the sequence generator. We also utilize multiple sequences, thus eliminating ambiguity raised by the fan-out representation. Multi/Markov Viterbi is summarized in Table 3.5.

3.3.2 Identifying the Planner's Arcs

Once the set of most-likely Markov sequences has been constructed, we need to identify which of the path planner's arcs the robot traversed. The representation of the path planner and of the POMDP are significantly different and the mapping is not direct. Only the need to reverse-engineer the data for learning has identified this representation gap. Although the details of this process are dependent on our particular implementation, the representation

Define τ to be the threshold for a high probability state.
Define $\mathcal{V}$ to be the set of selected Viterbi sequences; $Seq_t \in \mathcal{V}$ is then
    the most likely sequence generated from the most likely π state at
    time t; $s = Seq_t(t')$ is the state at time t' in Seq_t, for $0 \leq t' \leq t$.

Calculate π, δ and Ψ as in Tables 3.1 and 3.3. Recall that Ψ depends on δ.
Let $\mathcal{V} = \emptyset$.
Foreach t, $0 < t \leq T$:
    Let $s_{max} = \text{ARGMAX}_{\forall s \in S} \pi(s,t)$
    if $\pi(s_{max}, t) > \tau$
        Let $Seq_t(t) = s_{max}$
        Foreach t', from $t-1$ down to 0
            $Seq_t(t') = \Psi(Seq_t(t'+1), t'+1)$
        Let $\mathcal{V} = \mathcal{V} \cup Seq_t$

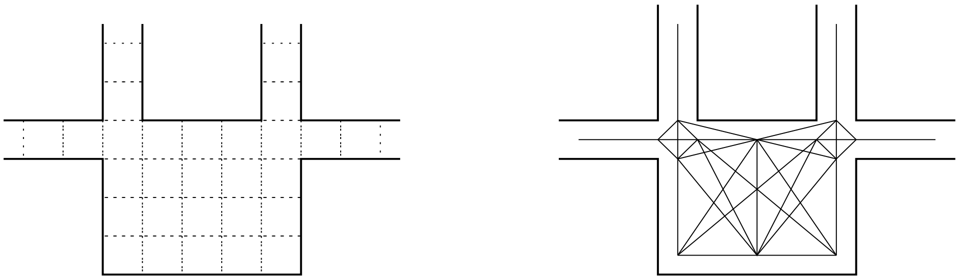
Table 3.5: Multi/Markov Viterbi: Viterbi's algorithm for generating abstract trajectories in Markov models with a high degree of fan-in/fan-out. It takes into account the state probability distribution, π, and uses multiple sequences to eliminate ambiguities created by the data representation.

gap problem is a general one. Each module in a given architecture may require a special-purpose representation that is well suited for its task, and mapping the information between layers may be non-trivial. Careful design of the architecture may reduce the representation gap, but it is extremely unlikely that the problem will be entirely eliminated.

The POMDP represents the world in a set of discrete square blocks. In our environment, 1 meter squares have been found to be empirically reliable while remaining efficiently computable. The path planner, on the other hand, represents the world in a set of arcs, where nodes correspond to topological junctions like doors and corridors.

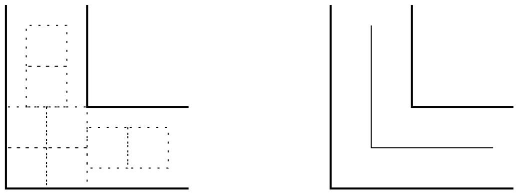
Although these representations clearly make sense for each module, there is no direct correspondence between the Markov states and the arcs. The original Xavier system was designed to create the Markov model from the topological map, not to extract the topological map from the Markov model. Figure 3.13 demonstrates the difference for a lobby area. There is no clear mapping from the Markov nodes to the path planners' arcs.

A similar problem exists at junctions in corridors. Our hallways are wider than the Markov precision, but along the corridor, we do not represent the full width. This decision increases efficiency while maintaining reliability. At junctions, however, we need finer control of the robot, and therefore finer location estimates, and therefore we retain the full representation. Figure 3.14 shows the different representations. There is a direct mapping from the four Markov states in the junction to a *node* in the topological map; however, the path planner does not consider nodes to have "space," and so we need to assign nodes to arcs. Unfortunately, it is unclear which Markov states correspond precisely to which arcs.



a) Markov Representation b) Path Planner Arc Representation

Figure 3.13: Different representations of a foyer.



a) Markov Representation b) Path Planner Arc Representation

Figure 3.14: Different representations of junctions in corridors.

We have addressed these problems by calculating the path using a greedy heuristic based on expected execution times.

First we calculate all the arcs that could possibly correspond to a single Markov node. For example, the four nodes at junctions in corridors would correspond to all the path planner arcs that meet there. Hence, there are often Markov nodes associated with multiple arcs. This fact complicates the reconstruction of the arc sequence because a single Markov sequence may map to multiple arc sequences.

We then reduce the number of possible arc sequences by permitting only the arcs that correspond to the *transition* between sequential Markov states in the Viterbi sequence. However, for a single Viterbi sequence, we are still left with many possible arc sequences.

The mapping function then assigns states to arcs in a greedy manner, based on expectation times. Consider Figure 3.15, in which arc_1 corresponds to $s_1, ..., s_j$, while arc_2 corresponds to $s_i, ..., s_n$. If we have an expected time $e(arc_i)$ to traverse arc_i, and timestamps on each state s_k, $t(s_k)$, then we say that states s_1 through s_k correspond to arc_1

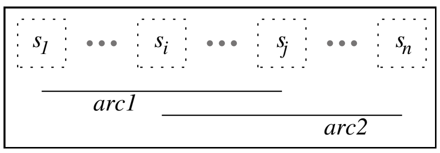


Figure 3.15: Multiple arcs corresponding to multiple Markov nodes. arc_1 corresponds to $s_1, ..., s_j$, and arc_2 corresponds for $s_i, ..., s_n$.

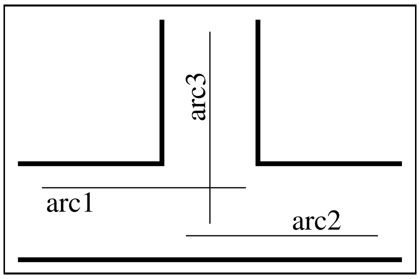


Figure 3.16: An example of when the greedy heuristic may fail.

for:

$$k = \begin{cases} i-1, & \text{if } t(s_{i-1}) - t(s_1) > e(arc_1), \text{ or} \\ l, & \text{if, for some } l \text{ such that } i \le l < j,\ t(s_l) - t(s_1) \le e(arc_1) < t(s_{l+1}) - t(s_1), \\ j, & \text{if } t(s_j) - t(s_1) \le e(arc_1). \end{cases}$$

arc_2 then corresponds to states s_{k+1} through s_n. We greedily add states to arcs until the Viterbi sequence is exhausted, thereby creating the complete arc sequence. We do this mapping for each of the Viterbi sequences returned by Multi/Markov Viterbi.

Experiments show that selecting arc sequences in this greedy manner yields good results. There are occasions however when the heuristic may fail. For example, imagine that the corridor intersection in Figure 3.16 contains many obstacles. If most of the execution traces contained paths from arc_1 to arc_2, then ideally, the excess traversal weight of the intersection should be evenly distributed between them. Instead, the heuristic will make the weight of arc_1 smaller, closer to the default value of an empty corridor, while arc_2 would be much larger, containing all the weight of the difficult intersection.

Any newly generated paths that pass through both arc_1 and arc_2 would have the correct total weight. However, any new paths passing through arc_3 and only one of arc_1 and arc_2 would have a poor estimate of the true traversal weight.

Empirically, this problem has not occurred. In general, the paths used as training data are a fair representation of the paths used at execution: if ROGUE travels certain typical routes, then it is likely that it will continue to do so. Moreover, the incremental nature of the learning algorithm means that ROGUE will self-correct with additional experience: if ROGUE starts travelling new routes, new data will be collected, and the combined body of evidence will create more accurate estimates of costs.

The probabilistic representation of the navigation module creates significant challenges in reconstructing the robot's path through the environment. ROGUE needs to estimate the most likely sequence of Markov states that the robot passed through, which can be done through a merging of the Bayesian POMDP state probabilities and Viterbi's algorithm. Then ROGUE needs to reverse-engineer the path planner's arcs from the Markov states. ROGUE collects each of the possible sequences into one body of data that collectively describes the robot's true path.

The process for extracting arc traversal events can be summarized as follows:

1. Apply Multi/Markov Viterbi; i.e. accumulate likely sequences of traversed Markov states.
2. Apply the heuristic to break the representation-gap; i.e. map the Markov state sequences into topological arc sequences.

These arc traversal events, $\mathcal{E}$, become input for the learning algorithm after they have been evaluated.

3.4 Costs

Once arc traversal events have been identified from the execution trace, updated costs need to be calculated. These costs become the value predicted by the learning algorithm as a function of the situational features. The learned costs are used by the path planner as traversal weights.

Rogue uses the cost evaluation function, $\mathcal{C}$, to determine the degree of success or failure of an event. When learning for the path planner, $\mathcal{C}$ yields an updated arc traversal weight for each arc traversal event $\varepsilon \in \mathcal{E}$.

In our current implementation, this weight is equal to the product of the *desired velocity* on that arc and the *actual time* spent traversing it, divided by the *modelled length*:

$$\mathcal{C}(\varepsilon) = vt/l.$$

This cost represents the experienced difficulty of the arc traversal. When the robot travels in a straight line at the desired speed, the cost is 1.0, indicating that the default cost estimate was correct.

Weights may be greater than one for the following reasons:

- The robot travels in a straight line more slowly than desired.
- The robot travels along a sinuous path at the desired speed.
- The modelled length of the arc were shorter than the actual length.

Weights may be less than one for the following reasons:

- The modelled length of the arc is longer than the actual length. (For the experiments conducted in our environment, the modelled length is 10% longer than the actual length.)
- The heuristic incorrectly assigns traversal times to arcs.

Another possible function includes position confidence. There may be occasions when it is more important for the robot to know where it is, than to move quickly through the environment. An expensive arc would then be one for which the robot's position estimates are very poor. Position confidence can be situation-dependent because of transitory obstacles that occlude landmarks or affect sensor reliability.

The data is stored in a matrix along with the cost evaluation and the environmental features observed when the event occurred. Those environmental features which change during the traversal are averaged. Table 3.6 shows a sampling from an *events matrix* generated by ROGUE.

This collection of feature-value vectors is presented in a uniform format for use by any learning mechanism. Additional features from the execution trace can be trivially added; this particular matrix was recorded for the path planner experiments described in Section 3.7, while sonar readings were added for the task planner experiments described in Chapter 4.

The events matrix is grown incrementally; most recent data is appended at the bottom. Each time the robot is idle, the execution trace is processed and new events are added to the matrix. The learning algorithm then processes the entire body of data, and creates a new set of situation-dependent rules by compressing the many examples. By using incremental learning, ROGUE can notice changes and respond to them on a continual basis.

The process for identifying and storing arc traversal events from the trace is summarized in Table 3.7. Step 1 corresponds to Section 3.3.1, step 2 corresponds to Section 3.3.2, and step 3 corresponds to Section 3.4. Each arc traversal event is stored in the events matrix along with the relevant situational features and the cost evaluation. The matrix is then used as input for the learning algorithm, described next.

The A* path planner uses these learned weights to improve its estimate of traversal time, as described in Sections 3.1.1 and 3.6.

ArcNo	Weight	CT	Speed	PriorArc	Goal	Year	Month	Date	DayOfWeek
233	0.348354	38108	34.998001	234	90	1997	06	30	1
192	0.777130	37870	33.461002	191	90	1997	06	30	1
196	3.762347	37816	34.998001	195	284	1997	06	30	1
175	0.336681	37715	34.998001	174	405	1997	06	30	1
168	1.002090	60151	34.998001	167	31	1997	07	07	1
246	0.552367	60099	34.998001	247	253	1997	07	07	1
201	1.002090	64282	34.998001	202	379	1997	07	07	1
134	16.549173	61208	34.998001	234	262	1997	07	09	3
238	0.640905	54	34.998001	130	379	1997	07	10	4
169	0.429588	39477	27.998402	168	31	1997	07	13	0
165	1.472222	8805	34.998001	164	379	1997	07	17	4
196	5.823351	3983	34.608501	126	253	1997	07	18	5
194	1.878457	85430	34.998001	193	262	1997	07	18	5

Table 3.6: Events matrix; each feature-value vector (row of table) corresponds to an arc traversal event $\varepsilon \in \mathcal{E}$. *Weight* is arc traversal cost, $\mathcal{C}(\varepsilon)$. The remaining columns contain environmental features, $\mathcal{F}$, valid at the time of the traversal: *CT* is CurrentTime (seconds since midnight), *Speed* is velocity, in cm/sec, *PriorArc* is the previous arc traversed, *Goal* is the Markov state at the goal location, *Year, Month, Date* and *DayOfWeek* form the date of the traversal.

Foreach time step $t < T$ in the execution trace
Let $s_{max} = \text{ARGMAX}_{\forall_{s \in S}} \pi(s,t)$
If $\pi(s_{max}, t) > \tau$, for some threshold τ
1. Let Seq_t be the Viterbi sequence generated from s_{max}:
$Seq_t(t) = s_{max}$
Foreach t' from $t-1$ down to 0
$Seq_t(t') = \Psi(Seq_t(t'+1), t'+1)$
2. Calculate the arc sequence that corresponds to Seq_t
3. For each arc traversal event $\varepsilon \in \mathcal{E}$ in the arc sequence
Estimate the cost of ε from $\mathcal{C}$: $\mathcal{C}(\varepsilon) = vt/l$
Store the arc traversal event ε, the features $\mathcal{F}$, and the weight $\mathcal{C}(\varepsilon)$ in the *events matrix*

Table 3.7: Identifying arc traversal events $\mathcal{E}$ from the execution trace.

3.5 Learning Algorithm

We now present the learning mechanism that creates the mapping from situation features $\mathcal{F}$ and events $\mathcal{E}$ to costs $\mathcal{C}$.

The input to the algorithm is the events matrix described in Section 3.4. The desired output is situation-dependent knowledge in a form that can be used by the planner.

We selected *regression trees* [Breiman *et al.*, 1984] as our learning mechanism because

- the data often contains *disjunctive descriptions*,
- the data may contain *irrelevant features*,
- the data might be *sparse*, especially for certain features,
- the learned costs are *continuous values*.

Bayesian learning would not successfully handle disjunctive functions, k-Nearest Neighbour algorithms would not handle irrelevant features well, neural networks would not generalize well for sparse data, and standard decision trees do not handle continuous valued output particularly well [Mitchell, 1997; Quinlan, 1993]. Other learning mechanisms may be appropriate in different robot architectures with different data representations.

We selected an off-the-shelf package, namely S-PLUS [Becker *et al.*, 1988], as the regression tree implementation. A regression tree is created for each event, in which features are splits and costs are learned values.

A regression tree is fitted for each arc using *binary recursive partitioning*, where the data is successively split until data is too sparse or nodes are pure. A *pure* node has a deviance below a preset threshold. Deviance of a node is calculated as $D = \sum(y_i - \mu)^2$, for all examples i and predicted values y_i within the node[5]. Section 3.7.2 presents experiments with different

[5]The average deviance, $\frac{1}{n}\sum(y_i - \mu)^2$, is not used because we want a node to be split when sufficient evidence accumulates; there is more value in splitting leaves with large numbers of examples.

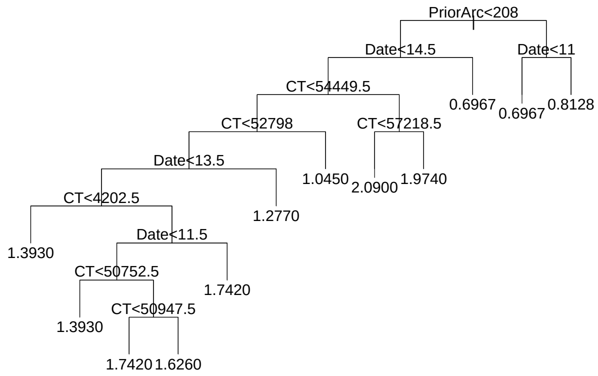


Figure 3.17: Learned tree for arc 208 from the Exposition world described in Section 3.7.1. Leaves represent learned costs (traversal weights); CT is current time, in seconds since midnight.

preset thresholds.

Splits are selected to maximize the reduction in the deviance of the node. Chambers & Hastie [1992] discuss the method in more detail. Figure 3.17 shows one sample learned regression tree built with our data. Each internal node in the tree represents one feature comparison. The left subtree indicates data for which the feature was less than the comparison value; the right subtree contains data for which the feature was greater than the comparison value. Leaf nodes show the arc's learned costs. For comparison, Table 3.8 shows the text version, which provides additional information including the number of examples covered by each node, as well as the deviance of each node.

We prune the tree using *10-fold random cross validation*, in which a tree is built using 90% of the data, and then the remaining 10% of the data is used to test the tree, resulting in the relationship between tree size and misclassification rates. This calculation is done 10 times, each time holding out a different 10% of the data. The results are averaged, giving us the best tree size so as not to over-fit the data. The least important splits are then pruned off the tree until it reaches the desired size.

The graph in Figure 3.18 shows the cross validation results for the tree learned for arc 208. Figure 3.19 shows the learned tree after pruning, and Table 3.9 shows the text version. Note again that the text version provides additional information including the number of examples covered by each node, as well as the deviance of each node.

This pruned tree represents the situation-dependent arc costs of arc 208. (This rule was generated from 340 examples from the Exposition world described in Section 3.7.1. Arc 208 appears in corridor 2.) When coming from the direction of arc 209, arc 208 has cost 0.7548. Otherwise, in the second half of the month, arc 208 has cost 0.6967. In the first half of the month, from midnight to 14:39:58 the traversal weight is 1.5610. From 14:39:59 to 15:07:29, it costs 1.0450 and for the rest of the day its traversal weight is 2.0320.

Although there are significant advantages to using an off-the-shelf package for learning, there is a disadvantage in that certain capabilities described in the literature may not be available. In this particular implementation, the selection criterion for a split is fixed: the

```
node), split, number of examples, deviance, learned value
      * denotes terminal node

  1) root 60 1.084e+001 1.4240
    2) PriorArc<208 52 6.671e+000 1.5270
      4) Date<14.5 48 3.681e+000 1.5970
        8) CT<54449.5 40 1.834e+000 1.5100
         16) CT<52798 36 8.749e-001 1.5610
           32) Date<13.5 32 5.124e-001 1.5970
             64) CT<4202.5 4 3.698e-032 1.3930 *
             65) CT>4202.5 28 3.236e-001 1.6260
              130) Date<11.5 24 2.607e-001 1.6060
                260) CT<50752.5 4 3.698e-032 1.3930 *
                261) CT>50752.5 20 4.315e-002 1.6490
                  522) CT<50947.5 4 0.000e+000 1.7420 *
                  523) CT>50947.5 16 2.847e-030 1.6260 *
              131) Date>11.5 4 0.000e+000 1.7420 *
           33) Date>13.5 4 0.000e+000 1.2770 *
         17) CT>52798 4 0.000e+000 1.0450 *
        9) CT>54449.5 8 2.697e-002 2.0320
         18) CT<57218.5 4 0.000e+000 2.0900 *
         19) CT>57218.5 4 3.698e-032 1.9740 *
      5) Date>14.5 4 9.244e-033 0.6967 *
    3) PriorArc>208 8 2.697e-002 0.7548
      6) Date<11 4 9.244e-033 0.6967 *
      7) Date>11 4 0.000e+000 0.8128 *
```

Table 3.8: Text version of the learned tree for arc 208.

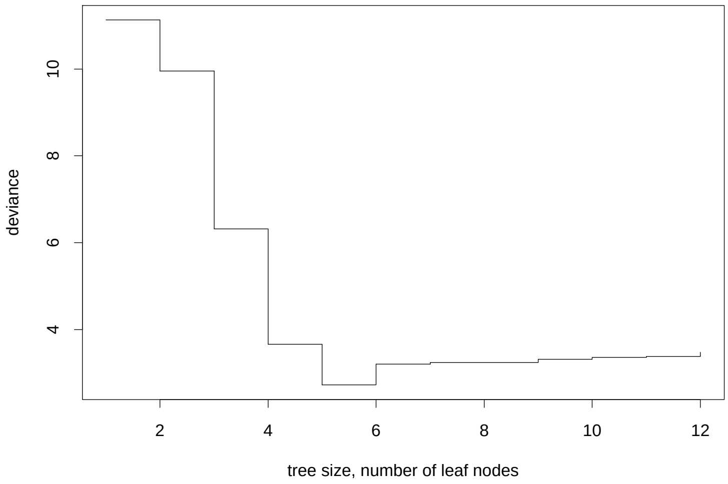


Figure 3.18: The cross validation results for arc 208.

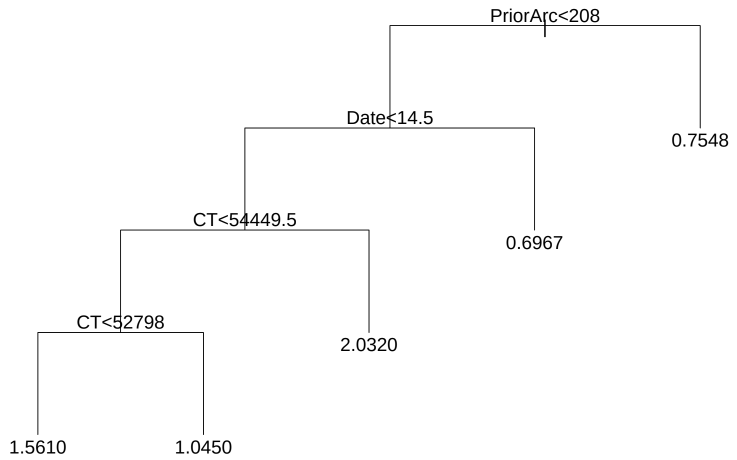


Figure 3.19: The learned tree from arc 208 after pruning.

```
node), split, number of examples, deviance, learned value
      * denotes terminal node

 1) root 60 1.084e+001 1.4240
   2) PriorArc<208 52 6.671e+000 1.5270
     4) Date<14.5 48 3.681e+000 1.5970
       8) CT<54449.5 40 1.834e+000 1.5100
        16) CT<52798 36 8.749e-001 1.5610 *
        17) CT>52798 4 0.000e+000 1.0450 *
       9) CT>54449.5 8 2.697e-002 2.0320 *
     5) Date>14.5 4 9.244e-033 0.6967 *
   3) PriorArc>208 8 2.697e-002 0.7548 *
```

Table 3.9: Text version of the learned tree from arc 208 after pruning. CT is current time, in seconds since midnight.

selected split is the one that maximizes the reduction in deviance of the node.

This restriction was somewhat limiting because real world domains often have different costs associated with different features, and we would like to have the system select splits with some consideration of cost. A good example is Ming Tan's technique of selecting the feature with the maximum *marginal information gain*: I^2/C where I is the information gain and C is the cost of measuring the feature [Tan, 1991].

In Section 4.4 we describe our re-implementation of the regression tree analysis to cope with feature costs. Feature costs were not an issue for the path planner experiments because the selected features were all high level, with a constant cost to acquire their values.

Section 3.7 presents the results of using regression trees to learn situation-dependent costs for path planner arcs. Our experiments show that regression trees adequately describe the situations found in Xavier's environment, and that situation-dependent costs are a feasible extension to the path planner, and significantly enhance the system.

3.6 Updating the Path Planner

Once the regression trees have been created (one for each arc), they are ready for use by the path planner. Each path from the root node of the tree to each leaf of the tree can be viewed as a situation-dependent rule.

The path planner requests the new arc costs from the update module each time it is preparing to generate a path. These costs are generated by matching the current situation against each arc's learned tree.

The update module parses the learned tree, matching each feature against the calculated or current value. When it reaches a leaf node, it updates the path planner with the learned value.

The mechanism for extracting the value of the feature from the current situation is provided *a priori*. For robot-dependent situation data, such as speed and vision, the update module monitors TCA messages from the other executing modules, and makes explicit information requests when necessary.

Using the A* algorithm described in Section 3.1.1, the path planner then uses the updated costs to calculate the best path. If the updated arc cost is high, then the path planner is more likely to avoid using that arc in a route. In this way, the path planner can successfully predict and avoid areas of the environment that are difficult to navigate.

In the event of a failure during navigation, for example a closed door, the path planner is re-invoked, at which point it re-requests the learned arc costs. A particular set of arc costs is valid for the calculation of a single path; any replanning forces an update of the costs.

Several changes were made to the path planner to support learned traversal weights. The original path planner used a constant traversal weight that was a function of the type of arc (corridor-corridor, corridor-room, etc). The arc data structure was extended to include a traversal weight, and the planner now uses that value if one exists (if not, it uses the default value). TCA commands were added to support dynamic changing of the probabilities and traversal weights of the arcs. Appendix C enumerates in more detail the changes made to the path planner.

3.7 Experimental Results

We have conducted four sets of experiments. The first three sets involve a simulated world for controlled experiments, while the fourth set were on the real robot. The first simulated-world set demonstrates that ROGUE can learn patterns. The second set explores rule stability and data generalization. The third set explores learning rates and the ability to detect a change in the environment. The final was run on the real robot, validating the algorithm and the need for it.

Xavier's simulator is primarily used to test and debug code before running it on the real robot. The simulator allows software to be developed, extensively tested and then debugged off-board before testing and running it on the real robot. The simulator closely approximates the real robot: it creates noisy sonar readings, it has poor dead-reckoning abilities, and it gets stuck going through doors. Most of these "problems" model the actual behaviour of the robot, allowing code developed on the simulator to run successfully on the robot with no modification [O'Sullivan *et al.*, 1997]. The simulator allows the tight control of experiments, to ensure that the learning algorithm is indeed learning appropriate situation-dependent costs.

3.7.1 Simulated World 1: Learning Patterns

The first environment tests ROGUE's ability to learn situation-dependent costs. Figure 3.20 shows the *Exposition World*: an exposition of the variety one might see at a conference. Rooms are numbered; corridors are labelled for discussion purposes only. Figure 3.20a shows the simulated world, complete with a set of possible obstacles. Figure 3.20b shows the topological map used by the path planning module; this map displays everything the robot "knows" about its environment.

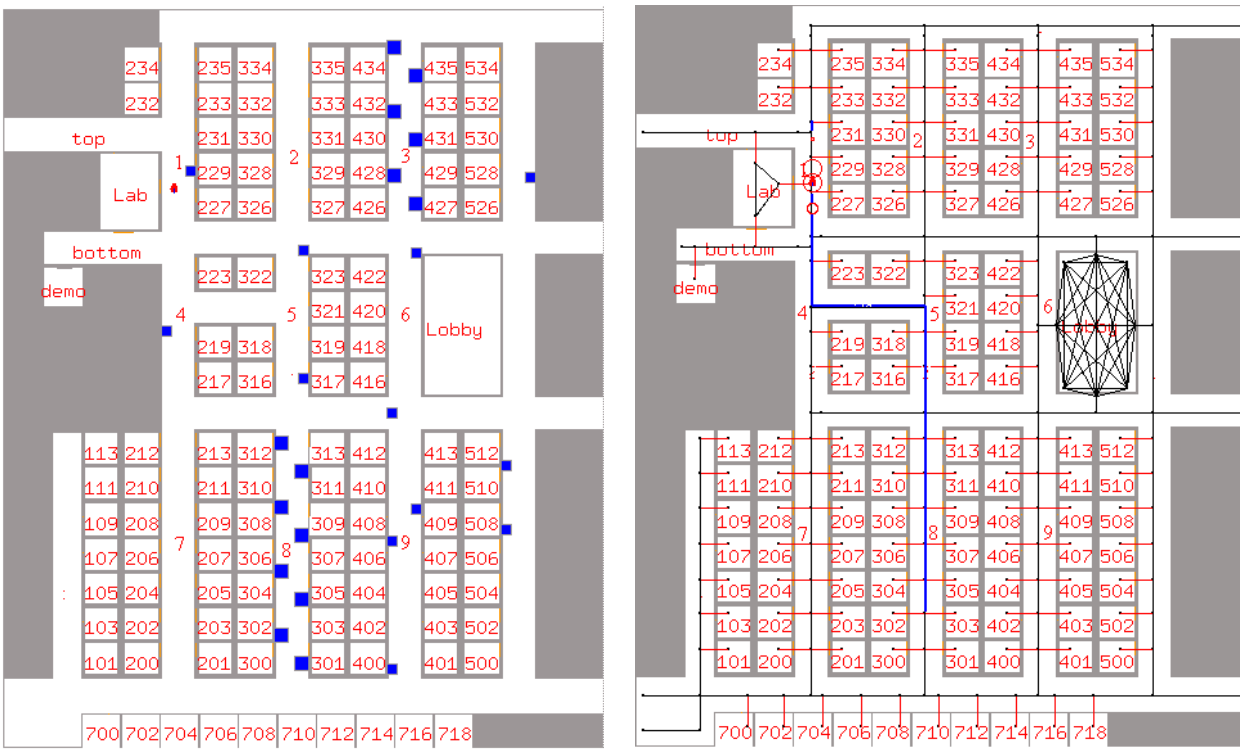


Figure 3.20: Exposition world. (a) Simulator: operating environment. Obstacles marked with dark boxes. (b) Path Planner: topological map. Arcs shown in light grey, a sample path shown darker.

The simulator has limited capabilities for dynamism: currently doors can only be opened and closed only at the whim of the user, and obstacles are static. For our experimental stage, we needed the robot to be operating in a dynamic world. We added dynamism by running each experiment in a *variation* of the map shown in Figure 3.20a. The position of the obstacles in the simulated world changes according to the following schedule:

- corridor 2 always clear
- corridor 3 with obstacles
  - EITHER Monday, Wednesday, or Friday between (midnight and 3am) and between (noon and 3pm)
  - OR one of the other days between (1 and 2am) and (1 and 2pm)
- corridor 8 always with obstacles
- remaining corridors with random obstacles (approximately 10 per map)

In each map, we ran a fixed path through the environment: from corridor 1 to booth 303 to 411 to 327 to 435 to 210, collecting the execution trace. (We ran random routes for the experiments in Section 4.6.1, and actual user requests in Section 3.7.4.)

This set of environments allowed us to test whether Rogue would successfully learn:

- permanent phenomena (corridors 2 and 8),
- temporary phenomena (random obstacles), and
- patterns in the environment (corridor 3).

The events matrix was generated as described in Sections 3.3 and 3.4, and then processed as described in Section 3.5.

3.7.1.1 Data and Rule Learning

Over a period of two weeks, 651 execution traces were collected. Almost 306,500 arc traversals were identified, creating an events matrix of 15.3 MB. The average training value of the arc traversals was 1.65. Figure 3.21 shows the frequency of arcs for a given cost.

The 17 arcs with fewer than 25 traversal events were discarded as insignificant, leaving 100 arcs for which the system learned trees. (There are a total of 331 arcs in this environment, of which 116 are doors, and 32 are in the lobby.) Trees were generated with as few as 25 events, and as many as 15,340 events, averaging 3060. A low number of traversals usually indicates that the robot strayed from the nominal path, while a large number indicates that the robot went over that arc more than one time. Generated trees had an average size of 18.04 total nodes and 9.02 leaf nodes.[6]

Figure 3.22 shows a sampling of learned trees. All arcs shown are from corridor 3. Both *DayOfWeek* and *CT* are prevalent in all the trees. (CT is *CurrentTime*, in seconds since midnight.) In Arc 244, for example, before 02:08:57, *DayOfWeek* is the dominant feature. In

[6]All presented data is for deviance = 0.10; see Section 3.7.2 for a discussion of deviance.

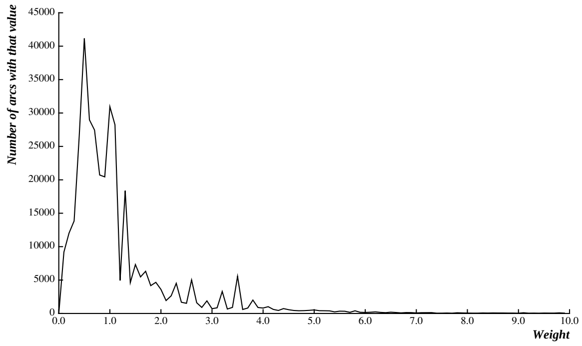


Figure 3.21: Arc cost frequency: most arcs in the training set have a cost close to 1.0, the default value.

Arc 240, between 02:57:36 and 12:10:26, there is one flat cost for the arc. After 12:10:26 and before 15:00:48, *DayOfWeek* again determines costs. Lack of data accounts for *Date* being used as a feature in this tree and *DayOfWeek* not being used before 3am.

Figure 3.23 shows the cost, averaged over all the arcs in each corridor, as it changes throughout the day. ROGUE has correctly identified that corridor 3 is difficult to traverse between midnight and 3am, and also noon and 3pm. During the rest of the day, it is close to default cost of 1.0. This graph shows that ROGUE is capable of learning patterns in the environment. Corridor 8, meanwhile, is *always* well above the default value, while corridor 2 is slightly below default, demonstrating that ROGUE can learn permanent phenomena. Minor variations in the value are a result of noise in the training data.

Table 3.10 shows the overall average cost of each of the three types of corridor: one that never has obstacles, one that occasionally contains random obstacles, and one that always contains obstacles. This data shows that ROGUE successfully separates different types of phenomena.

Corridor 2	Empty	0.73
Corridor 4	Random Obstacles	1.13
Corridor 8	Many Obstacles	3.28

Table 3.10: The average cost of all the arcs in each type of corridor.

Figure 3.24 to 3.26 shows learned arc costs for Wednesday at 01:05am. As expected, corridor 2 is considered inexpensive, while corridors 3 and 8 are considered expensive. As the cost threshold increases, fewer arcs are considered expensive. Arcs near turns can be more expensive, because the robot may be recovering from the turn. Also, short arcs may be more affected by an error in the heuristic mapping from the Multi/Markov Viterbi sequence.

These figures are closeups of the complete maps shown in Figure 3.27. When costs are only *slightly* more expensive than default (costs > 1.25, Figure 3.27a), numerous arcs are highlighted, demonstrating that the system has successfully identified areas where obstacles may appear: corridors 3 and 8, plus most of the arcs in which random obstacles may appear.

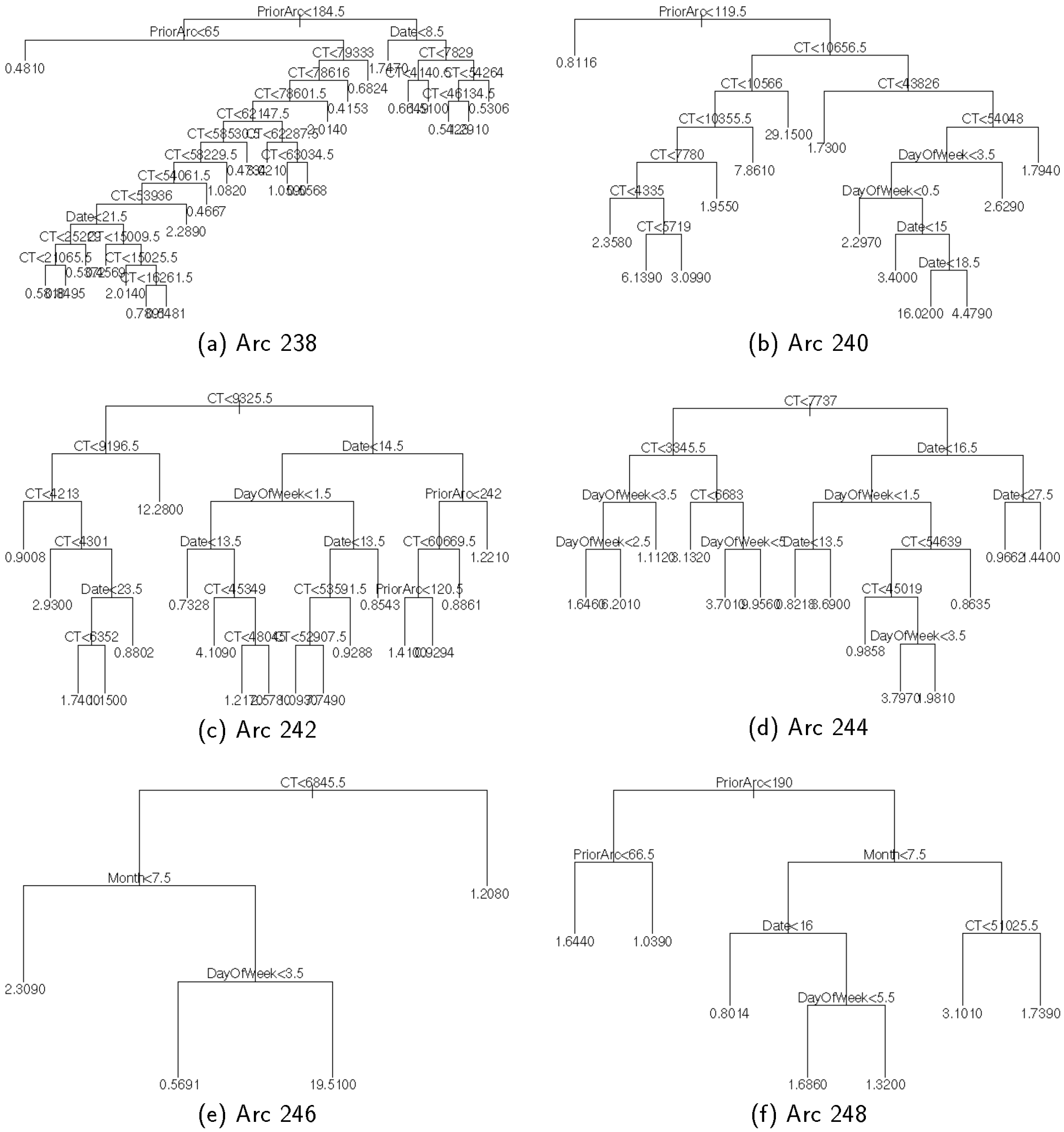
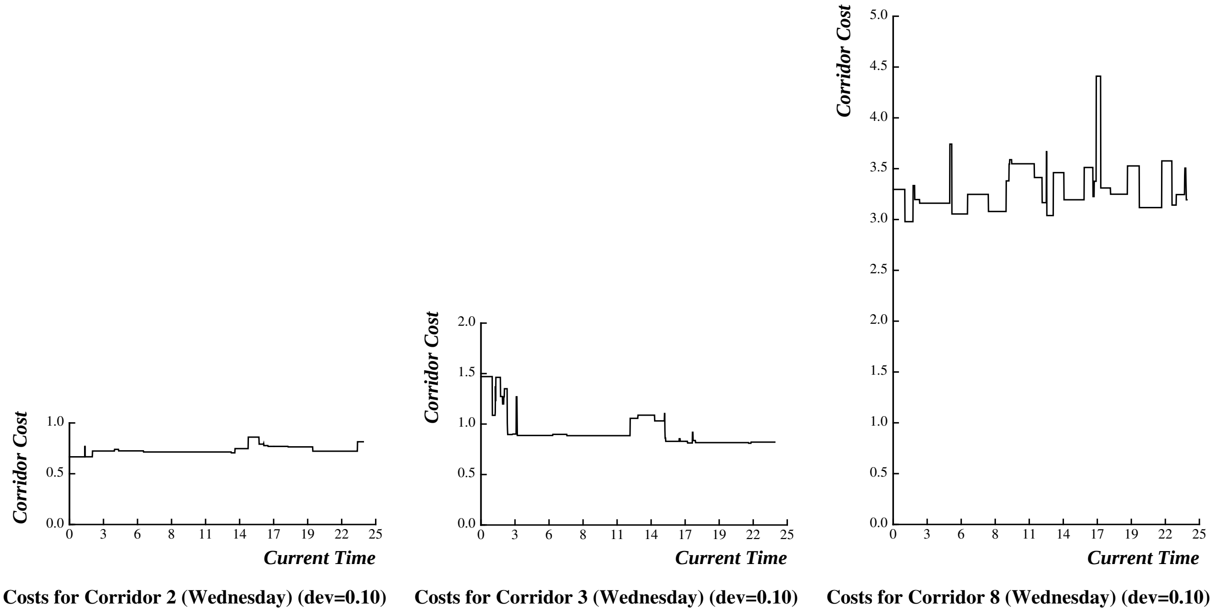


Figure 3.22: Learned trees for the six arcs in corridor 3.



(a) Corridor 2 (b) Corridor 3 (c) Corridor 8

Figure 3.23: Learned corridor cost (average over all arcs in that corridor) for Wednesdays.

Figure 3.27b shows those arcs *somewhat* more expensive than default, and Figure 3.27c shows those arcs *much* more expensive than default. Again, note that as the cost threshold increases, fewer arcs are considered expensive. Note that all arcs containing random obstacles have been eliminated from these images; only extremely difficult turns continue to be marked.

For comparison, Figure 3.28 shows learned costs for Tuesday at 09:45am. Note that corridor 3 is not considered expensive at any time.

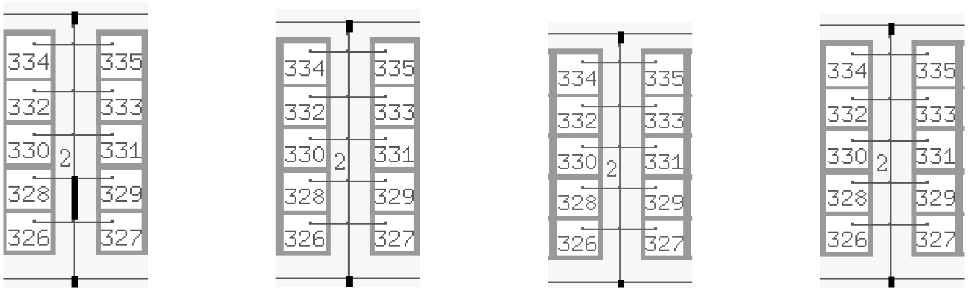
The data collected for this experiment has shown that ROGUE's learning algorithm successfully identified patterns in the environment. ROGUE also successfully identified both permanent and temporary phenomena.

3.7.1.2 Effect on Path Planner

Figure 3.29 illustrates the effect of learning on the path planner. The goal is to have ROGUE learn to avoid expensive arcs (those with many obstacles). Figure 3.29a shows the path normally generated. Figure 3.29b shows the path generated by the path planner after learning; note that the expensive arcs have been avoided.

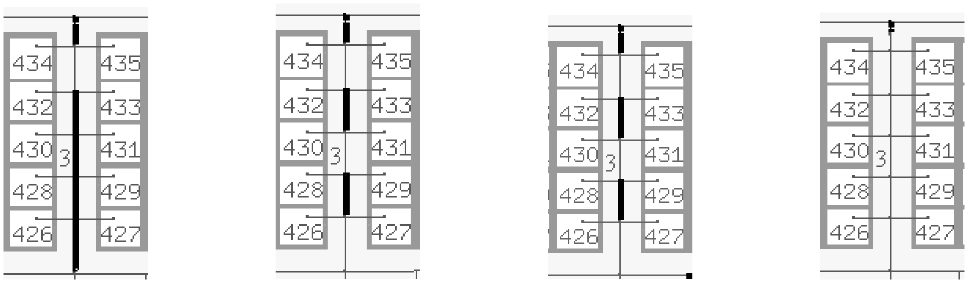
Table 3.11 shows a sample path calculation, for a path from room 231 to room 319. It shows the default path, evaluating it with both the default cost values and the learned costs. It also shows the new path, evaluated with the learned values. Assuming the learned costs closely reflect reality, the new path is 60% of the cost of the default path.

Table 3.12 shows the total *weight* $\times$ *length* values for several routes, using the learned costs to evaluate both the default path and the new path. The new path is consistently better than the default path.



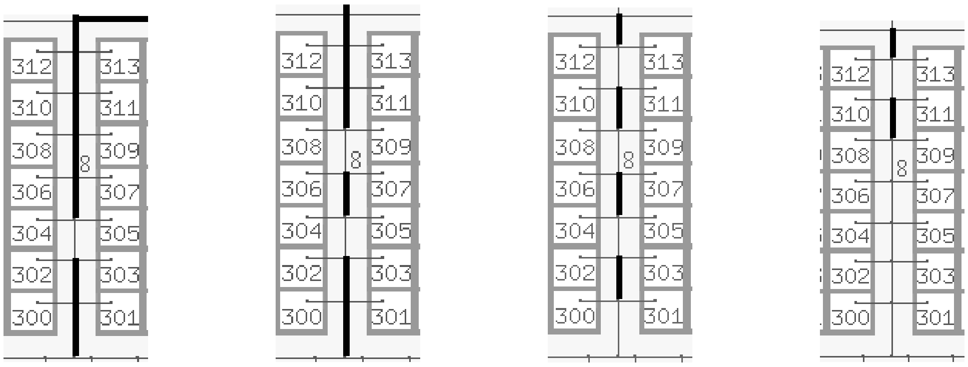
(a) Costs > 1.25 (b) Costs > 2.00 (c) Costs > 2.50 (d) Costs > 5.00

Figure 3.24: Expensive arcs in corridor 2 for situation: Wednesday, 01:05am. Note that this corridor is not considered expensive. (Dark, thick edges are expensive.)



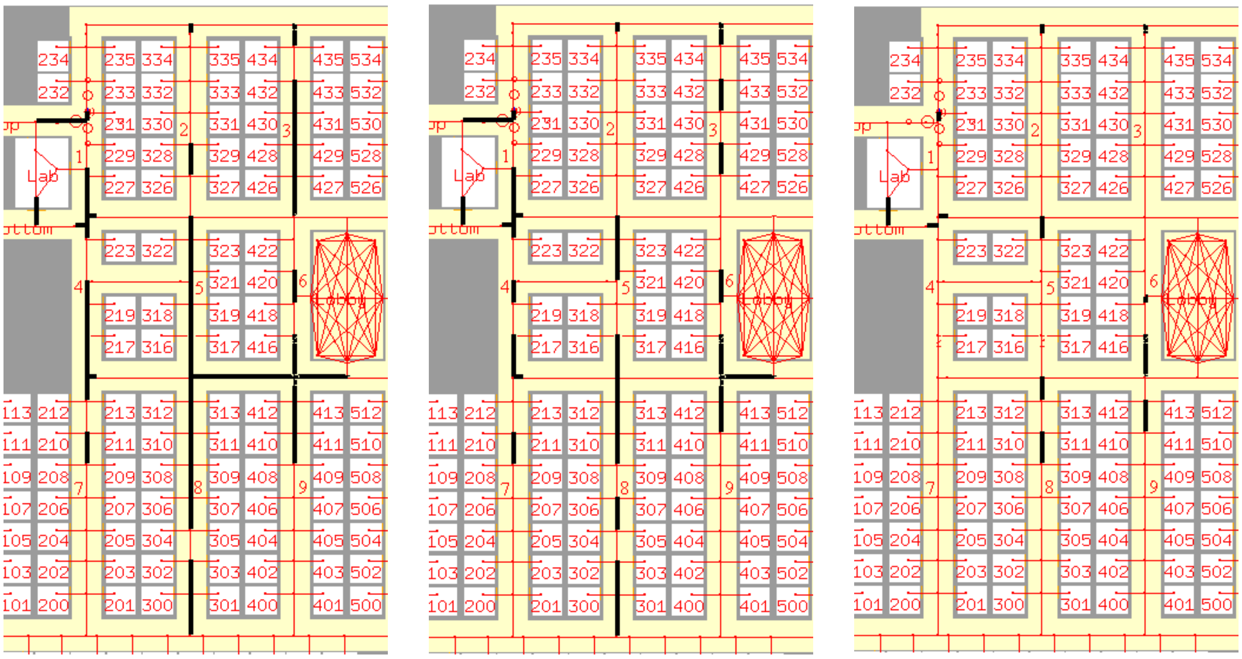
(a) Costs > 1.25 (b) Costs > 2.00 (c) Costs > 2.50 (d) Costs > 5.00

Figure 3.25: Expensive arcs in corridor 3 for situation: Wednesday, 01:05am. Note that most arcs are expensive until the cost threshold is very high. (Dark, thick edges are expensive.)



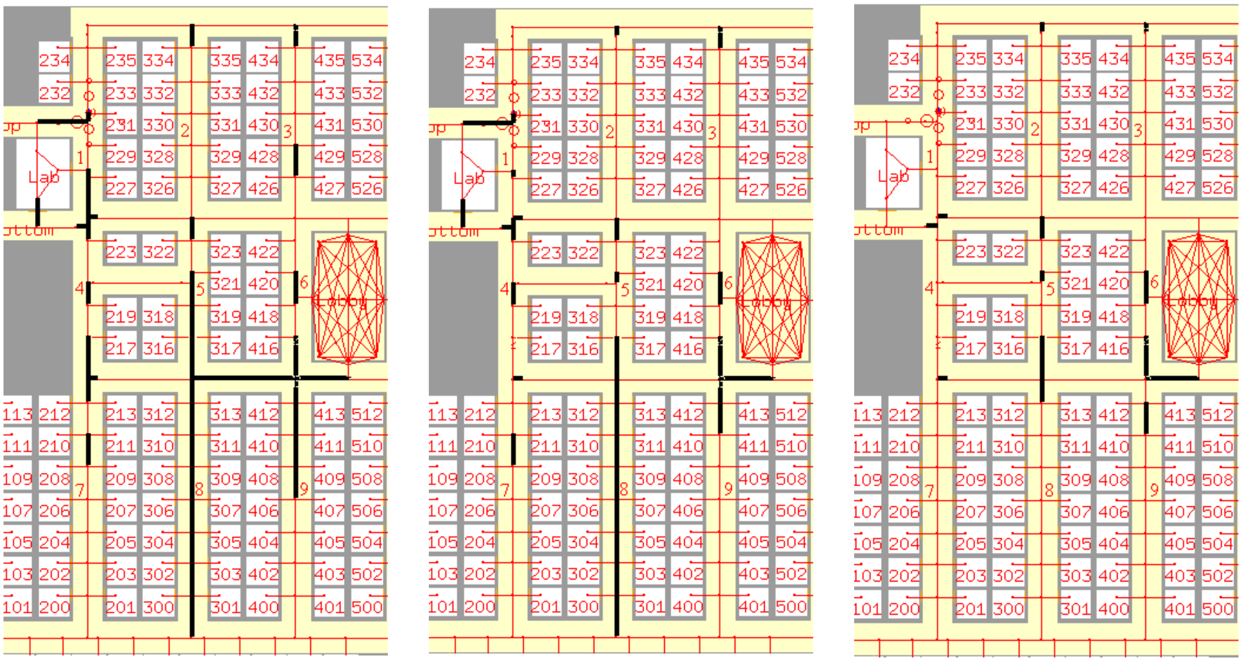
(a) Costs > 1.25 (b) Costs > 2.00 (c) Costs > 2.50 (d) Costs > 5.00

Figure 3.26: Expensive arcs in corridor 8 for situation: Wednesday, 01:05am. Note that most arcs are expensive until the cost threshold is very high. (Dark, thick edges are expensive.)



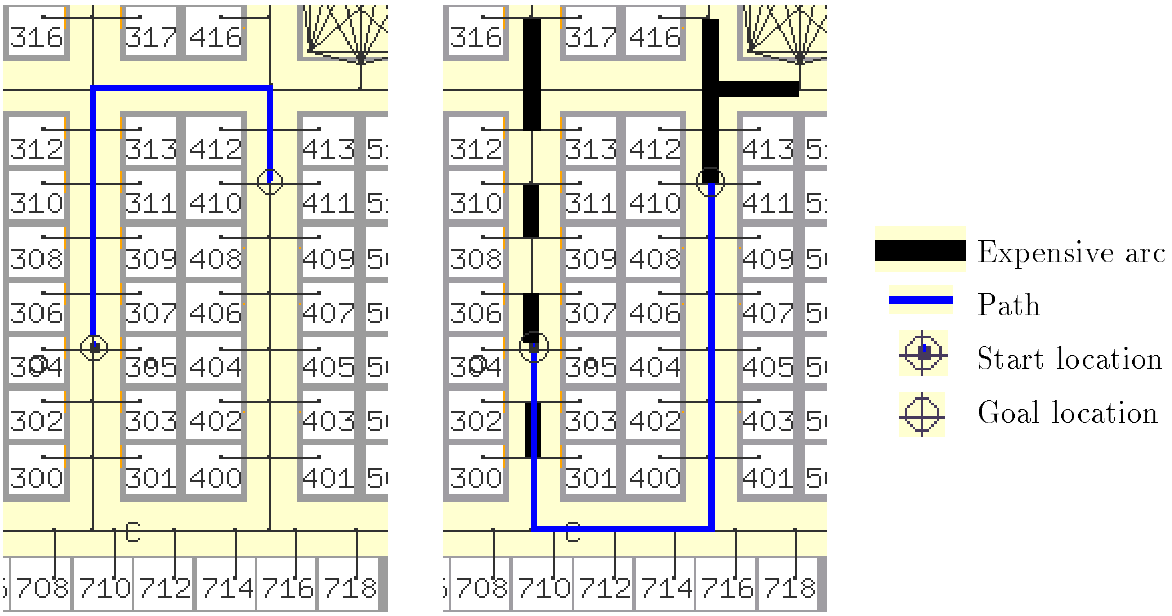
(a) Costs > 1.25 (b) Costs > 2.00 (c) Costs > 5.00

Figure 3.27: Expensive arcs for situation: Wednesday, 01:05am. Note that corridors 3 and 8 are expensive, along with arcs containing random obstacles and difficult turns. (Dark, thick edges are expensive.)



(a) Costs > 1.25 (b) Costs > 2.00 (c) Costs > 5.00

Figure 3.28: Expensive arcs for situation: Tuesday, 09:45am. Note that corridor 3 is not considered expensive.



(a) Default Path (b) New Path

Figure 3.29: Comparison of path planner's behaviour before and after learning. (a) Default path (when all corridor arcs have default value). (b) New path (when corridor arcs have been learned) on Wednesday 01:05am; note that the expensive arcs have been avoided (arcs with cost > 2.50 are denoted by very thick lines).

Default Path Default Costs				**Default Path Learned Costs**				**New Path Learned Costs**			
Arc	W	L	W * L	Arc	W	L	W * L	Arc	W	L	W * L
176	1.00	82.00	82.00	176	1.14	82.00	93.32	176	1.14	82.00	93.32
175	1.00	189.00	189.00	175	0.53	189.00	100.40	175	0.53	189.00	100.40
174	1.00	205.00	205.00	174	0.44	205.00	89.52	174	0.44	205.00	89.52
173	1.00	69.00	69.00	173	1.45	69.00	100.05	173	1.45	69.00	100.05
172	1.00	347.50	347.50	172	1.69	347.50	585.88	172	1.69	347.50	585.88
171	1.00	82.50	82.50	171	2.57	82.50	212.11	265	0.61	796.50	483.63
170	1.00	108.00	108.00	170	3.53	108.00	381.35	205	1.18	190.50	225.55
169	1.00	355.50	355.50	169	3.34	355.50	1185.95	203	0.84	272.00	227.23
291	1.00	749.50	749.50	291	1.00	749.50	749.50	202	1.61	83.50	134.18
201	1.00	190.50	190.50	201	1.00	190.50	190.88	201	1.00	190.50	190.88
199	1.00	274.00	274.00	199	1.43	274.00	392.09	199	1.43	274.00	392.09
		Total:	2652.50			*Total:*	4081.05			*Total:*	2622.74

Table 3.11: Path length calculation for a path between room 231 and room 319. W is *weight* and L is *length*. The path chosen after learning is 60% the total learned cost of the default path, or a 40% improvement.

Start Room	Goal Room	Situation	Default Path Default Costs	Default Path Learned Costs	New Path Learned Costs	Percent Improvement
231	303	Mon, 15:40	4503.50	6481.96	5969.99	8%
303	411	Mon, 15:40	2908.00	6753.12	3768.66	44%
411	327	Mon, 15:40	3343.00	5438.67	5438.67	0%
327	435	Mon, 15:40	2683.00	2759.07	1274.97	55%
435	210	Mon, 15:40	4969.50	6502.58	5595.47	14%
			Total:	27423.43	22047.76	20%
231	303	Wed, 01:00	4503.50	6433.49	5586.48	13%
303	411	Wed, 01:00	2908.00	6250.80	3768.66	40%
411	327	Wed, 01:00	3343.00	5002.09	5002.09	0%
327	435	Wed, 01:00	2683.00	8902.85	1280.35	86%
435	210	Wed, 01:00	4969.50	12351.17	5305.65	57%
			Total:	38940.40	20943.23	46%
231	303	Thu, 01:00	4503.50	6432.49	5586.18	13%
303	411	Thu, 01:00	2908.00	6090.72	3768.67	38%
411	327	Thu, 01:00	3343.00	4842.02	4842.02	0%
327	435	Thu, 01:00	2683.00	3447.87	1280.34	63%
435	210	Thu, 01:00	4969.50	6896.18	5305.66	23%
			Total:	27709.28	20782.87	25%

Table 3.12: Path length calculation for a variety of paths under three different situations. We show the default estimate of path length, evaluate the default path with the learned costs, and the length of the path that A* finds with the learned costs. Finally, we show the percent improvement in path length between the default path and the new path.

The data we have presented here demonstrates that ROGUE successfully learns situation-dependent arc costs. It correctly processes the execution traces to identify situation features and arc traversal events. It then creates an appropriate mapping between the features and events to arc traversal weights. The path planner then correctly predicts the expensive arcs and creates plans that avoid difficult areas of the environment.

3.7.2 Simulated World 2: Stability and Generalization

In Section 3.5 we described how the learned regression trees are built. In particular, a node is split when its deviance grows beyond a preset threshold. This section presents experiments to explore different thresholds and their effect on data generalization and rule stability. One important consideration is that data can be over-generalized. It is important to find a reasonable level of data generalization so that rules are neither over-specific nor over-general.

Using the Exposition World, we collected four sets of trees, with the maximum node deviance set to each of 0.00, 0.10, 0.25 and 0.50. A deviance of 0.00 corresponds to a tree exactly fitting the data. A larger deviance means that more training examples will be classified together. For example, two training examples with weights of 0.75 and 1.0 will be

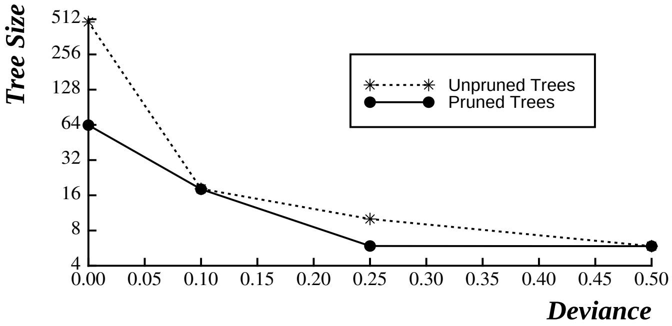


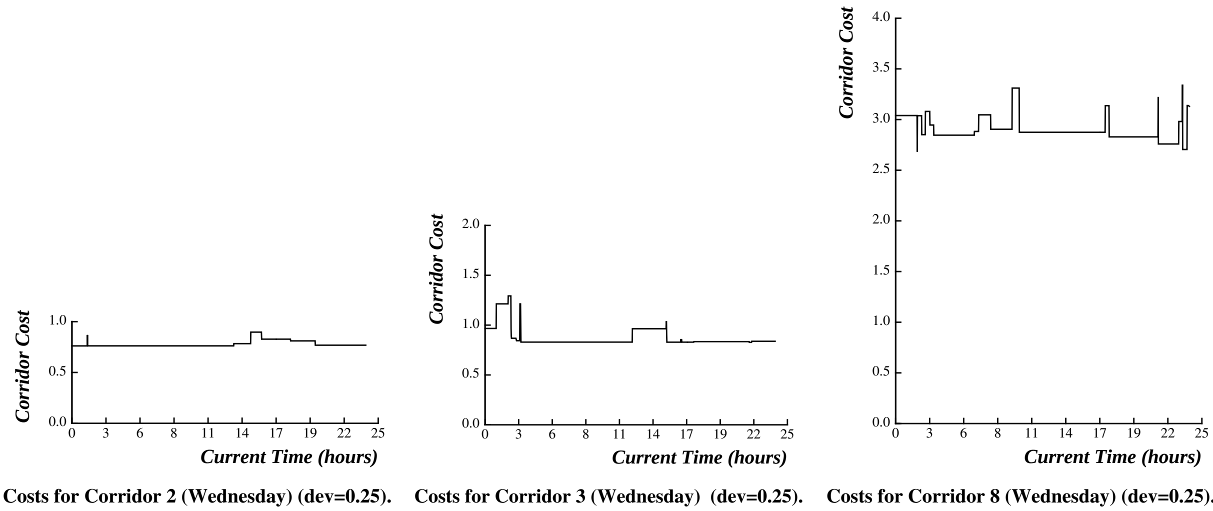
Figure 3.30: Maximum node deviance vs. learned tree size.

classified together for deviance greater than 0.031, but separated into two nodes for deviance less than or equal to 0.031. If there were ten examples of each weight, then they would be classified together only when maximum node deviance greater than 0.31.

As deviance grows, generalization rates grow and rule stability increases. Figure 3.30 shows how tree size is affected by deviance; smaller trees provide greater generalization.

Greater generalization means fewer nodes in the tree, and hence less variation in arc costs. Figure 3.31 shows the average corridor costs for deviance equal to 0.25, for comparison with Figure 3.23 (page 84) which shows costs for deviance equal to 0.10. Note that these graphs show much less variation.

Figure 3.32 shows the arcs in corridor 3 for deviance equal to 0.25. Note that they are considerably smaller and more generalized than the trees presented in Figure 3.22 (page 83), which shows trees generated for deviance equal to 0.10. These trees have over-generalized



(a) Corridor 2. (b) Corridor 3. (c) Corridor 8.

Figure 3.31: Corridor cost (average over all arcs in that corridor) for Wednesdays. Trees generated with deviance = 0.25. Note that graphs are smoother than for deviance = 0.10 (Figure 3.23, page 84).

the data. None of them capture the *DayOfWeek* feature required to correctly fit the data.

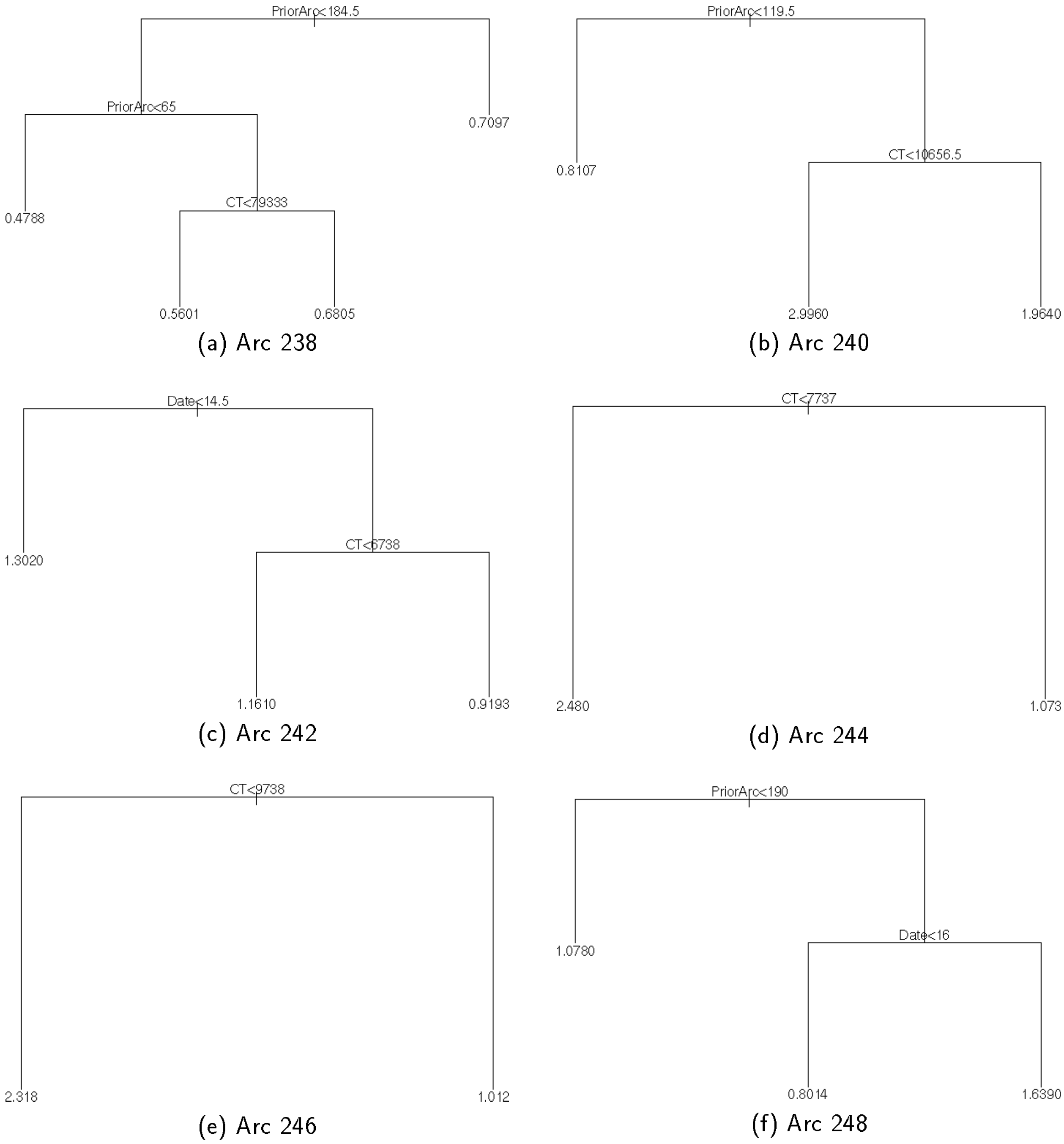


Figure 3.32: Learned trees for the six arcs in corridor 3. Deviance is 0.25. Compare to Figure 3.22 (page 83), which displays trees for deviance set to 0.10. Note that these over-generalize the data because the feature *DayOfWeek* is completely ignored, and the *CT* is oversimplified.

3.7.3 Simulated World 3: Learning Rates

In this experiment, we explored (i) the system's learning rate, and (ii) the need to forget old data.

Learning rate: ROGUE is an incremental learning system. Each time it collects execution data, it adds the newly experienced events to the events matrix, generates a new set of learned rules, and then uses those new rules for its next execution.

Issue 1: Stability. We need to determine how much data is required before ROGUE has a reasonably stable concept of its environment.

Issue 2: Learning Rate. We also need to determine how quickly ROGUE can absorb and respond to changes in its environment.

Data Relevance vs. Processing Power: As data ages, it may become less and less relevant for current planning. For example, a conference may lead to a week of very crowded corridors and data showing expensive arc traversals. Although the learning system will correctly create situation-dependent rules that state "*if current-date-is-during-conference, then cost-is-high*," these rules will never match the current situation, and hence the effort expended to create these rules is wasted.

The massive amount of data we collected for the above experiments could lead to a lot of wasted effort when old, irrelevant data is processed. For this reason, the system will need to have some scheme for "forgetting" data. However, it is important not to forget *everything*, since long-term patterns would never be detected. For example, unless the system maintains data over a span of years, it will never detect annual patterns such as New Year's Day; instead such patterns will be treated as noise.

Issue 3: Forgetting. We need to determine the effect of forgetting data on learning patterns in the environment.

3.7.3.1 Data

We collected data to explore these three issues in the *corridor-switch world*, shown in Figure 3.33. We collected 74 execution traces in which the robot did laps around the environment (five laps in one direction per trace, changing directions between traces). In the first 34 runs, corridor A was filled with obstacles while corridor B was clear. In the remaining 40 traces, corridor A was cleared while corridor B contained the obstacles shown.

We then analyzed the data in the form of "windows" of size n, in which only n consecutive execution traces were analyzed to form events matrices. For a working system, window size is equivalent to maintaining an execution history of n traces. We then plotted a graph of the average cost of each corridor for that window, shown in Figure 3.34. For a window size of n,

- *Region X* is the first $(34 - n)$ traces, which contain only data showing corridor A to be expensive,

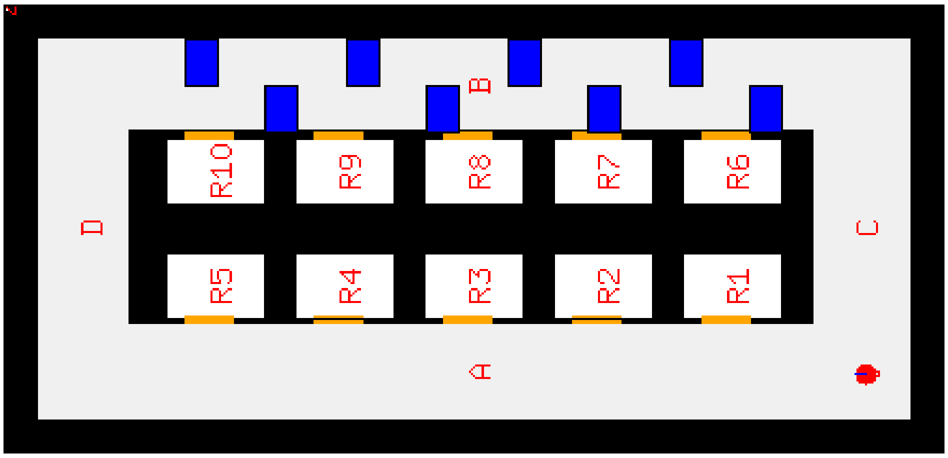


Figure 3.33: Corridor-switch world.

- *Region Y* is the last $(40 - n)$ traces, which contain only data showing corridor B to be expensive, and
- *Region Z* is the central n traces, which contain data of both types.

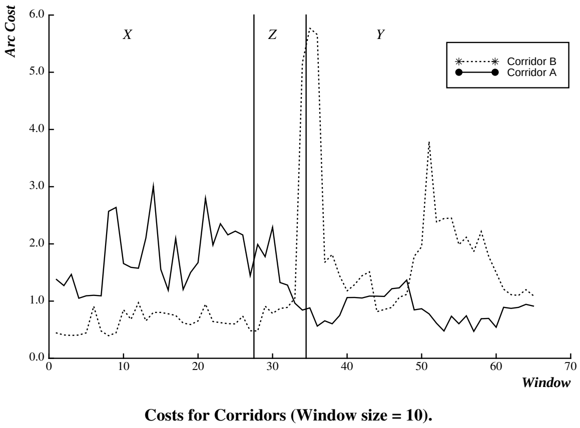
The plot in regions X and Y demonstrates the *stability* of the learned rules: substantial changes from point to point indicate that the system is highly susceptible to noise. The plot in region Z demonstrates the *learning rate* of the system: the crossover point shows how quickly the system has identified the environmental change. The plot in region Y shows the effect of *forgetting* data: all data showing corridor A to be expensive has been lost.

It should be noted that the graphs shown in Figure 3.34 are somewhat misleading: plotted points are an average, *for all features and for all arcs*, of the cost of the corridor. The learned rules, however, *remain* situation-dependent: Figure 3.35 shows a "typical" rule taken from the learned rules of a window in region Z, where some of the data shows that corridor A is more expensive, and some of the data shows that corridor B is more expensive. In particular corridor A was more expensive for data collected between September 9 through September 19, while data collected after September 20 showed that corridor B was more expensive. This rule correctly identifies the change.

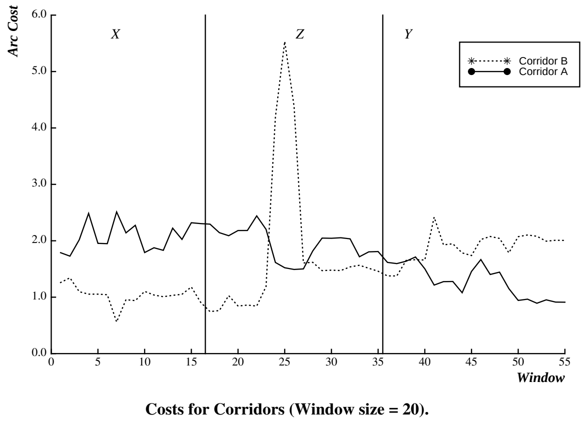
All tested window sizes show the same trend: that corridor A is more expensive than B in region X, corridor B is more expensive than A in region Y, and they crossover in region Z. Noise in the data accounts for the short unexpected overlaps, and is more noticeable in the smaller window sizes. Noise and a minor bug in the simulator[7] account for the unexpected peaks; the robot gets stuck often enough that Rogue can detect the pattern.

Figure 3.34a shows the graph for a window size of 10. It is very unstable, in that exchanging one execution trace for another leads to substantial changes in the estimate of the corridor cost. The small amount of data also means that Rogue responds to the change

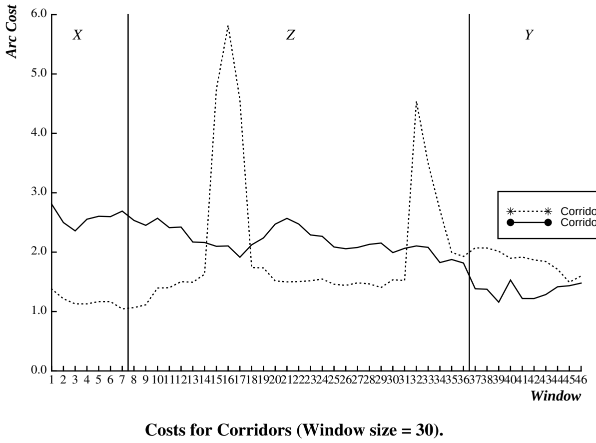
[7] The simulator bug is that when the robot bumps into an obstacle, it can become "stuck" more easily than the real robot; this problem raises the value of the learned arc costs.



(a) Window size is 10.



(b) Window size is 20.



(c) Window size is 30.

Figure 3.34: Effect of window size on stability, learning rate and forgetting data.

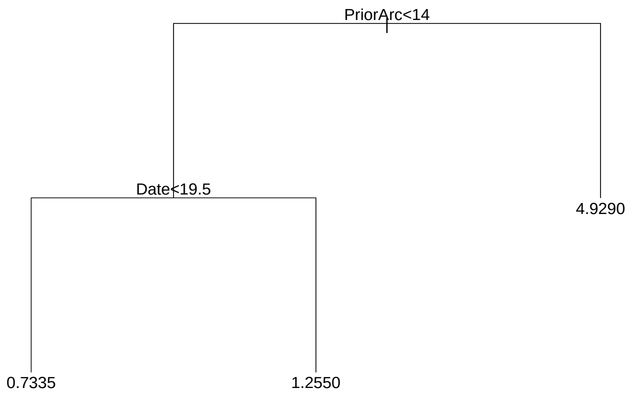


Figure 3.35: Typical rule inside the crossover region, Z. This rule is for an arc in corridor B, which is more expensive after September 20, or while recovering from a turn.)

very quickly; one execution trace corresponds to 10% of the data. Windows 40 through 48 contain two traces in which the robot had problems in corridor A, accounting for the unexpected rise in cost.

Figure 3.34b shows the graph for a window size of 20. This graph is considerably more stable than the previous one, but noise can still strongly affect the data. Figure 3.36 shows how corridor cost changes as a function of current time for the first 6 windows of size 20. These graphs show that exchanging one execution trace can immediately and significantly change the average estimate of the cost of the corridor. Figure 3.34c shows the average corridor costs for a window size of 30. This plot is very stable, in that point-to-point estimates of cost are generally much smaller. The large peaks in all three graphs come from

The graphs in Figures 3.36 and 3.37 shows cost estimates in consecutive windows. Each consecutive window changes exactly one file, so that in Figure 3.36, 5% of the data is changed, while in Figure 3.37, 3.3% of the data changes. Notice that the graphs in Figure 3.37 show much more stability than those in Figure 3.36, in that each window is very similar to the previous one. For example, notice that the change from Figure 3.36c to Figure 3.36d is quite dramatic: costs between 1am and 7am drop by roughly 0.5, while costs after 7am increase by roughly 1.0. The change from Figure 3.36d to Figure 3.36e is again quite dramatic. Meanwhile, each window in Figure 3.37 is very similar to the one before.

The graphs in Figure 3.34 demonstrate clearly that the system should use as large a history as physically and computationally possible. Any data that is explicitly forgotten will never again influence learned rules, and hence small window sizes means that long-term patterns will never be detected. Any pattern whose period is greater than the window size will be considered permanent by the system.

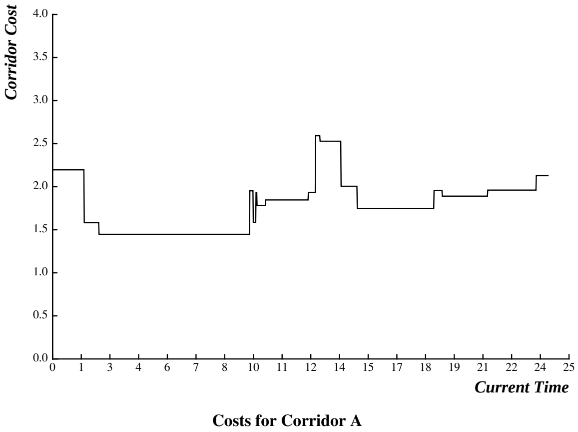
Larger windows create greater rule stability and confidence in the validity of the environmental knowledge captured. Rogue can learn situation dependent rules that separate old data from recent data, thereby successfully identifying temporary phenomena, and it can do so in a fairly small number of execution traces.

3.7.4 Real Robot

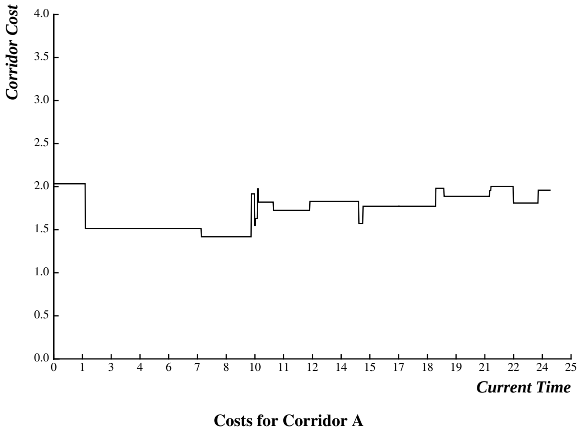
The final set of data was collected from real Xavier runs on the fifth floor of our building (part of which was shown previously in Figure 1.4, page 11).

Goal locations and tasks were selected by the general public through Xavier's web page, `http://www.cs.cmu.edu/~Xavier`. This data has allowed us to validate the need for the algorithm in a real environment, as well as to test the predictive ability given substantial amounts of noise.

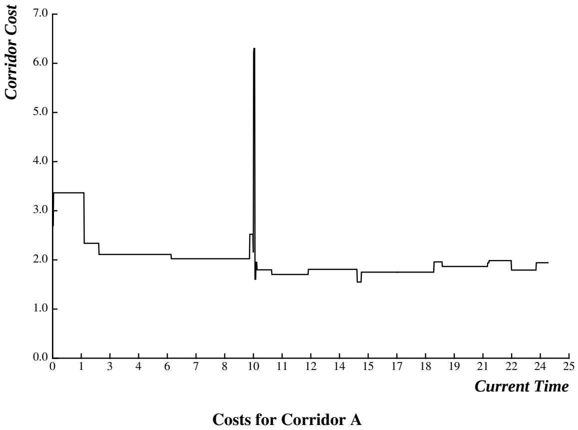
We show the incremental nature of Rogue through an analysis of the data at two snapshots in time.



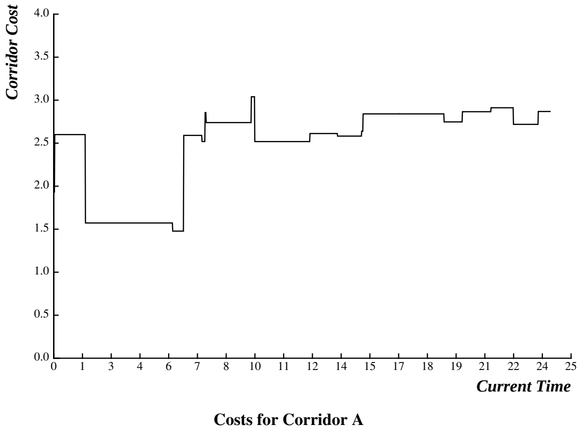
(a) Corridor A, Window 1.



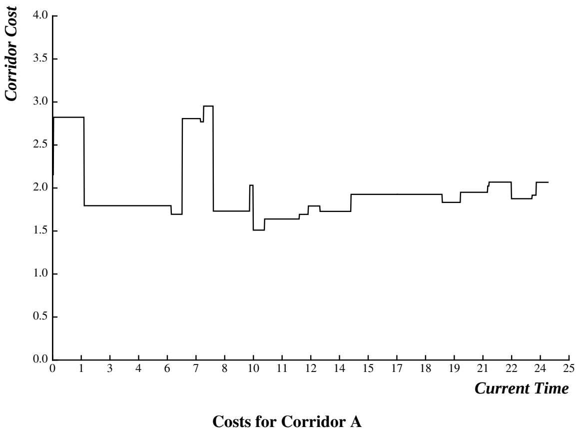
(b) Corridor A, Window 2.



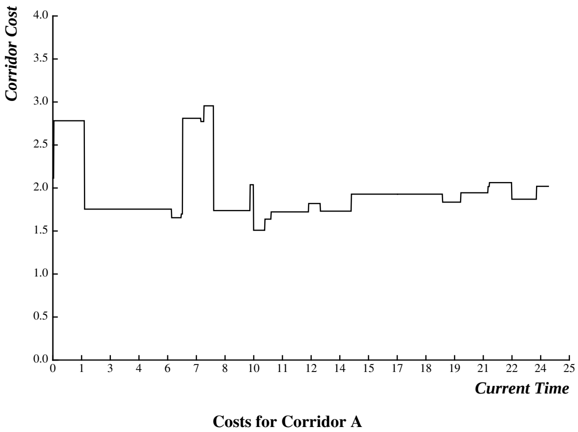
(c) Corridor A, Window 3.



(d) Corridor A, Window 4.

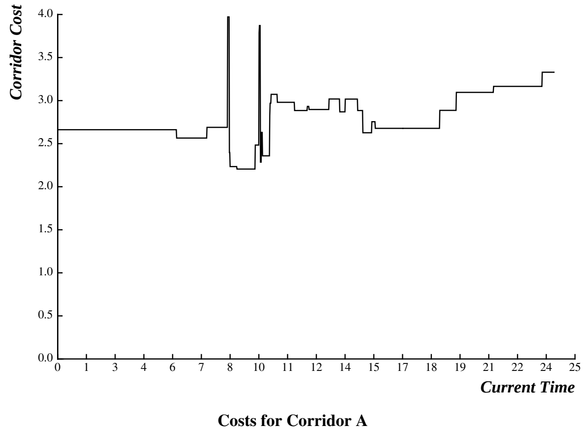


(e) Corridor A, Window 5.

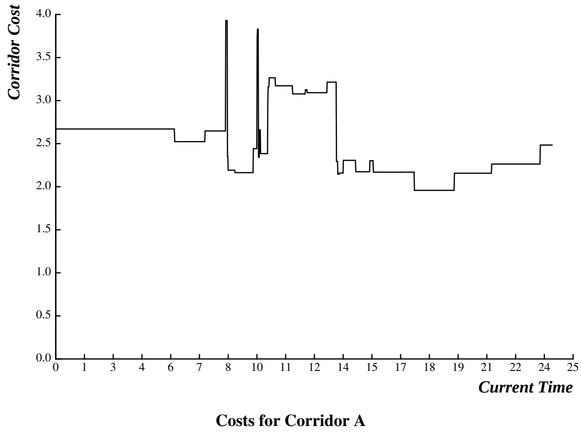


(f) Corridor A, Window 6.

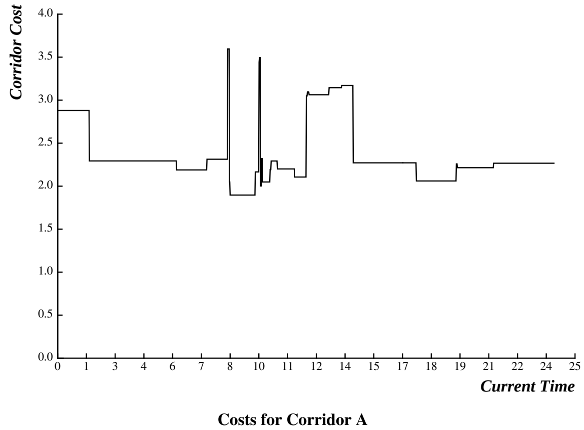
Figure 3.36: Window size: 20 trial runs. Note that window 3 and window 4 have quite different shapes, as do window 4 and window 5.



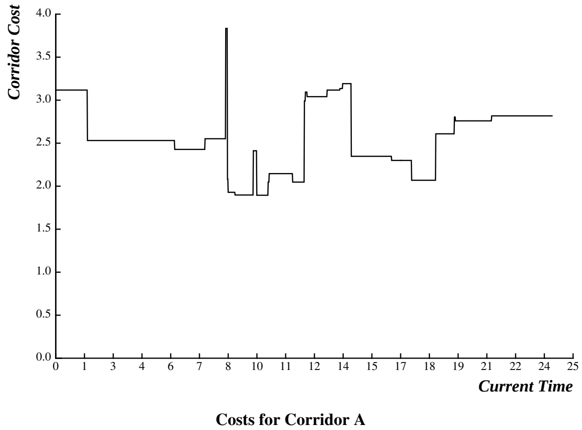
(a) Corridor A, Window 1.



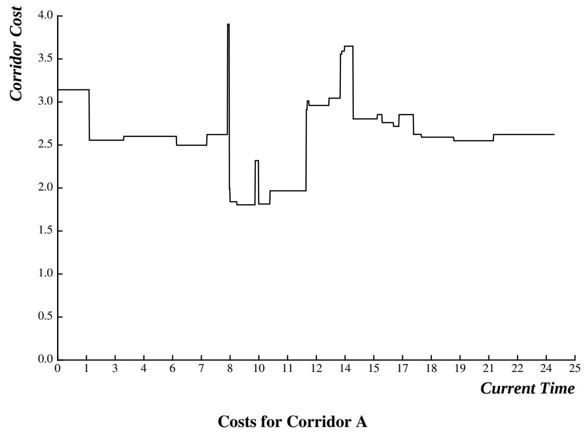
(b) Corridor A, Window 2.



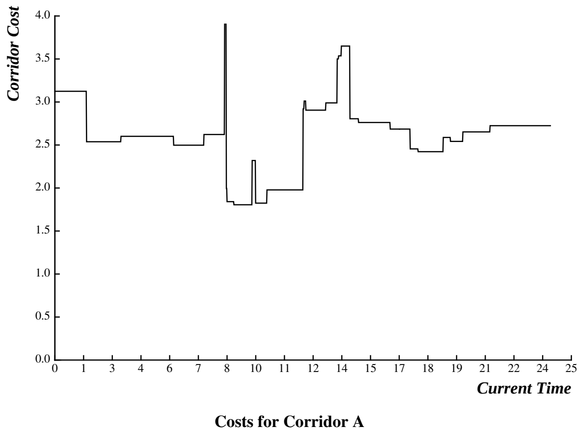
(c) Corridor A, Window 3.



(d) Corridor A, Window 4.



(e) Corridor A, Window 5.



(f) Corridor A, Window 6.

Figure 3.37: Window size: 30 trial runs. Note that the shape of the curve from each window to the next changes only minimally.

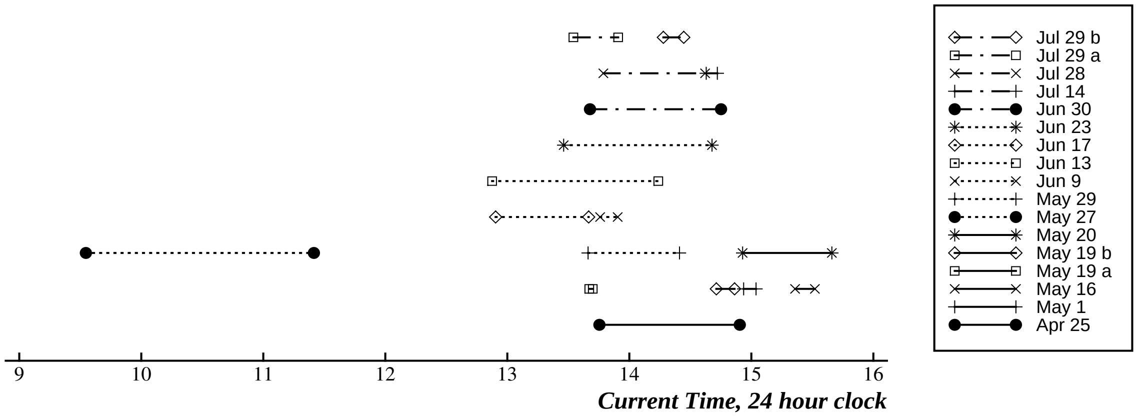


Figure 3.38: Distribution and length of robot running times, April-July 1997.

3.7.4.1 31 July 1997

Over a period of three months, 17 robot execution traces were collected. These traces were run between 9:30 am and 3:40pm and varied from 10 minutes (0.35 MB) to 82 minutes (14 MB). Figure 3.38 shows the distribution and length of running times.

More than 15,000 arc traversal events were recorded, for a total of 766 KB of training examples. Trees were learned for 89 arcs from an average of 169 traversals per arc. At deviance = 0.25, the average tree size was 10.2 nodes (5.1 leaf nodes). At deviance = 0.10, the average tree size was 20.4 nodes (10.2 leaf nodes).

Figure 3.39 shows the average learned costs for all the arcs in the lobby. The histogram below the graph shows number of execution traces per time step (calculated from Figure 3.38). Note that the graph is shown for a particular Wednesday; date might be a relevant feature. Values differentiated by other features were averaged. The system correctly identified lunch-time as a more expensive time to go through the lobby. The minimal morning data was not significant enough to affect costs, and so the system generalized, assuming that morning costs were reflected in the earliest lunch-time costs.

3.7.4.2 31 October 1997

During the subsequent three months, an additional 42 traces were collected, yielding a total of 59 execution traces. Figure 3.40 shows the distribution and length of running times.

An additional 57,249 arc traversal events were recorded, for a total of 72,516 events and 3.6 MB of data in the events matrix. Trees were learned for 115 arcs from an average of 631 traversal events per arc (min 38, max 1229). Data from nine arcs were discarded because they had fewer than 25 traversal events. At deviance = 0.25, the average tree size was 16.3 nodes (8.1 leaf nodes). At deviance = 0.10, the average tree size was 23.1 nodes (11.5 leaf nodes).

Figure 3.41 shows the average learned costs for all the arcs in the lobby. The histogram shows the number of execution traces per time step (calculated from Figure 3.40). Note that

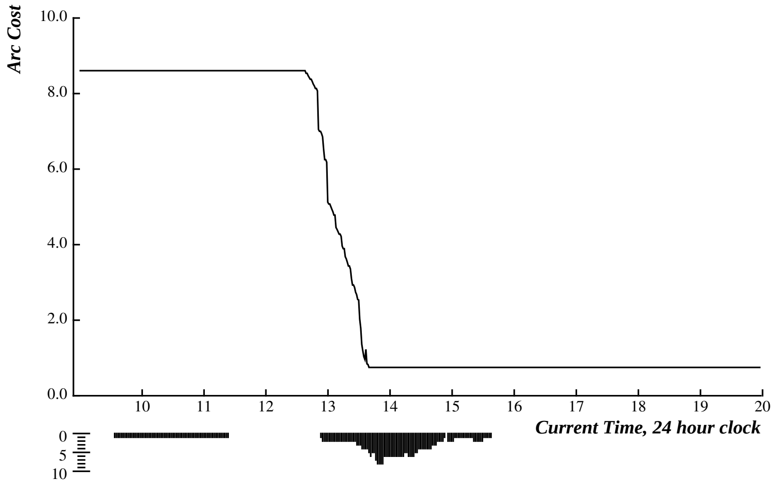


Figure 3.39: Learned costs for Wean Hall lobby on Wednesday, August 6. (Data from April-July 1997.) The histogram below the graph indicates volume of training data, in terms of number of execution traces; most data was collected between 1:30pm and 2:45pm.

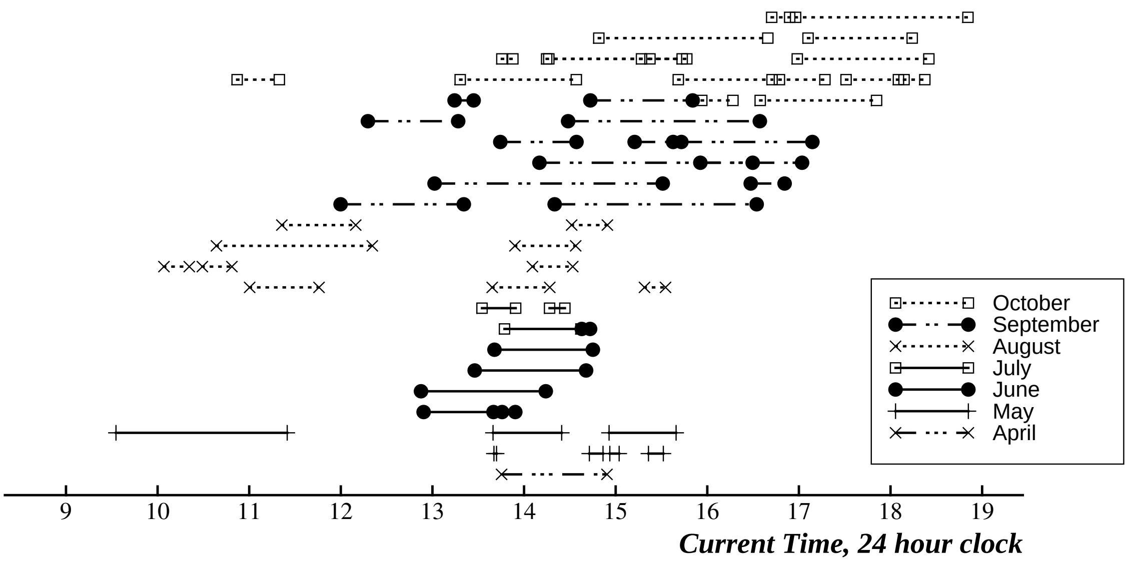


Figure 3.40: Distribution and length of robot running times, April-October 1997.

the graph is shown for a particular Wednesday; date might be a relevant feature. Values differentiated by other features were averaged.

This graph shows that the system is still confident that the lobby is expensive to traverse during the lunch hour. The greater volume of data reduced the cost estimate, but the morning data was still not sufficient to reduce the morning cost. To our surprise, the graph

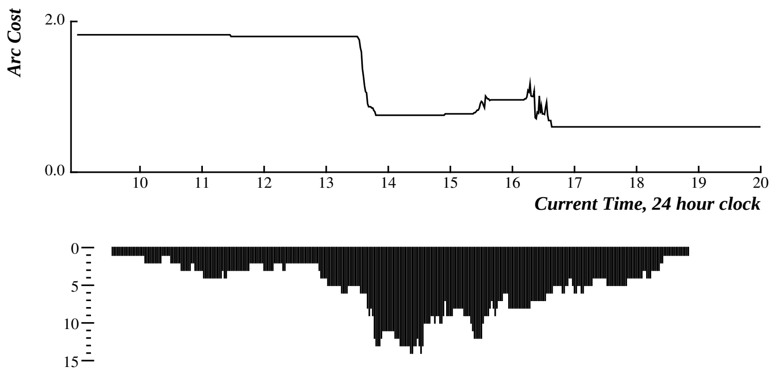


Figure 3.41: Learned costs for Wean Hall lobby on Wednesday, November 11. (Data from April-October 1997.) The histogram below the graph indicates volume of training data, in terms of number of execution traces; most data was collected between 1pm and 6pm.

shows a slightly higher cost during the late afternoon.[8] Investigation reveals that it reflects a period when afternoon classes have let out, and students come to the area to study and have a snack.

This data shows ROGUE's robustness to a changing world, even in an environment where many of the default costs were tediously hand tuned by the researchers. The added flexibility of situation-dependent arc costs increases the reliability and efficiency of the overall robot system.

3.8 Summary

We have presented a general framework for learning situation-dependent rules. These rules are extracted from execution data, and then used by a planner to improve the quality of generated plans.

We instantiated this framework with Xavier's path planner, presenting a learning robot with the ability to learn from its own execution experience. ROGUE uses predictive features of the environment to create *situation-dependent costs* for the arcs in the topological map used by the path planner to create routes for the robot. ROGUE effectively identifies arc traversal events, $\mathcal{E}$, from the execution trace so that the learning algorithm can correlate them with situational features, $\mathcal{F}$, and create updated costs, $\mathcal{C}$. These costs, represented as learned regression trees, will reflect the patterns detected in the environment, and the path planner will know which areas of the world to avoid (or exploit), and therefore find the most efficient path for each particular situation.

ROGUE processes the execution trace generated by the navigation module to extract events relevant for learning. The execution trace contains a massive, continual stream of probabilistic, low-level data. To identify which arcs the robot traversed in the topological

[8] Note that the April-July data did not contain many traces during this time period.

map, we modified Viterbi's algorithm to operate directly in the Markov model; Multi/Markov Viterbi effectively generates abstract trajectories in Markov models with a high degree of fan-in/fan-out. In this manner, ROGUE effectively abstracts the information in the execution trace to identify arc traversals. Each of these arc traversals is then evaluated, and the cost recorded along with the situational features existing at the time of the traversal event.

This data is then correlated by a regression tree algorithm to create situation-dependent arc costs for each of the traversed arcs. Finally, the path planner uses the updated costs to create efficient, situation-dependent routes for the robot. The algorithm works incrementally, improving the situation-dependent rules after each run of the robot. The process as used for the path planner is summarized in Table 3.13 (compare to Table 1.2, which outlines the process for a general planner).

We presented empirical data from both controlled, simulated environments as well as from the real robot. Our data demonstrates the effectiveness and utility of our approach.

1. Create route plan.
2. Navigate route; record the execution trace.
3. Identify events $\varepsilon \in \mathcal{E}$: arc traversals.
4. Learn mapping: $\mathcal{F} \times \mathcal{E} \rightarrow \mathcal{C}$.
5. Create rules to update arc costs.

Table 3.13: General learning approach as instantiated for the path planner.

Chapter 4

Learning for the Task Planner

ROGUE's learning goal is to extract from execution knowledge which will help a planner make better decisions. The general situation-dependent learning approach involves extracting learning events, $\mathcal{E}$, from the execution trace, evaluating them with a cost function, $\mathcal{C}$, and then correlating those events with situational features of the environment, $\mathcal{F}$. The learned information can then be used by a planner to improve the quality and reliability of generated plans.

In Chapter 3, the general approach was used to calculate *action costs* for an A* robotic *path planner*. The path planner used these situation-dependent action costs to create plans plans with the best expected execution time in the given situation.

In this chapter, we instantiate the general approach to calculate *action probabilities* for a symbolic *task planner*. ROGUE creates situation-dependent search control rules that guide the task planner towards actions with higher probability of success. ROGUE collects execution data to record the success or failure of learning events, $\mathcal{E}$. Events in the task planner result from operator applications, such as moving from one location to another, or delivering an item. Each event is evaluated with a cost function, $\mathcal{C}$, that determines the probability of action success, including for example missing deadlines or having timeouts while waiting for someone. The learner then correlates situational features, $\mathcal{F}$, to the learning events to create PRODIGY4.0 search control rules.

Two features of ROGUE's task planner make learning both effective and useful.

- ROGUE is able to interleave the plans for multiple compatible tasks. This ability allows ROGUE to create control rules to improve the compatibility estimates and also the interleaved order of different tasks.
- ROGUE interleaves planning with execution. This ability allows ROGUE to create rules that improve the order of executed actions.

These two features lead to two distinct goals for learning inside the task planner. The first is to *create a better plan* given the set of requests from users. This process primarily includes using the learned situation-dependent control rules to avoid tasks when they cannot be achieved. The rules depend on *high-level features* of the environment that can be detected well before actions need to be executed.

1. Interleave planning and execution for asynchronous requests Record execution events $\varepsilon \in \mathcal{E}$ and features $\mathcal{F}$
2. Learn mapping: $\mathcal{F} \times \mathcal{E} \rightarrow \mathcal{C}$
3. Create control rules to guide planner

Table 4.1: ROGUE's learning approach as instantiated for the task planner.

The second goal is to *execute the plan more effectively.* This goal does not exist for the path planner since the POMDP navigation module is completely disconnected from the planner; navigation occurs after the path planner has created the complete plan[1]. The task planner, however, determines when to send a plan step for execution, and hence can benefit from using learned experience to improve the execution performance of the robot. These rules depend on features in the environment that can only be detected when the action is about to be executed; we call these features *execution-level features.*

The learning approach used by ROGUE is identical to that described in Chapter 3. However, ROGUE needs to identify the task planner's events from execution data, and to process the learned trees to form search control rules for the task planner. The common learning approach is one of the contributions of this research. The approach as used in the task planner is summarized in Table 4.1 (compare to Table 1.2, page 10, which outlines the approach for a general planner).

The primary reason we implemented our learning framework in the task planner was to show the general applicability of the approach; the implementation described in this chapter is a *prototype.* We also extended the number and type of features available for learning.

In each of the sections below, we describe only the parts of ROGUE which are *different* from the path planner, making references to areas of overlap. In particular, the feature definition, $\mathcal{F}$, is essentially the same as for the path planner, while events, $\mathcal{E}$, and costs, $\mathcal{C}$, are planner-dependent. In Section 4.1 we describe the mechanism ROGUE uses to acquire features from Xavier. We also extend the set of available features to incorporate those from the execution-level. In Section 4.2 we describe the events relevant for learning in the task planner. In Section 4.3 we describe the cost function.

$\mathcal{F}$, $\mathcal{E}$ and $\mathcal{C}$ are transformed into an events matrix as for the path planner, and ROGUE uses the same learning algorithm to process the data. In Section 4.4, we present the one extension made to the learning algorithm so that it correctly handles execution-level features. In Section 4.5 we describe the mechanism use to transform the learned information into PRODIGY4.0 control rules.

We present experimental data in Section 4.6, and summarize the main contributions in Section 4.7. Related work can be found in Section 5.2.

[1]Our learning approach could be applied to the navigation module; see Section 6.2.

4.1 Features

In the feature discussion of Section 3.2, we described the features available in Xavier, including speed, time of day, sonar observations, camera images, other goals, and the desired route. We argued that execution-level features are generally not useful for learning in a planner because they are not projective. For that reason, the feature set used in the experiments of Chapter 3 included only high-level features.

However, the task planner controls the execution of the plan it creates (whereas the path planner's routes are executed by the navigation module). Hence, execution-level features such as sonar and camera become relevant. In particular, there are a number of occasions when the robot needs to stand in one place and *wait*. On these occasions, the current value of the execution-level feature clearly correlates to a future value (they essentially become projective in the short-term). For this reason, we extended the stored feature values to incorporate sensors.

The execution trace that Rogue analyzed to create situation-dependent arc costs for the path planner was created by the navigation module. The task planner, however, records its own execution trace since much of the detail recorded by the navigation module is not necessary. However, it is still the navigation module which has access to and defines the set of available features. To make them accessible to other modules, we implemented a TCA query that can be used to collect all current execution features.

The current list of features includes:

- robot odometer readings
- robot speed and acceleration
- robot sonar readings
- the Markov state probability distribution
- time and date

It can, of course, be incrementally expanded to incorporate new features. Appendix D shows the data structures and code used to implement the TCA query to report execution features.

4.2 Events

The two goals of learning control knowledge for the planner is to have Rogue learn when tasks and actions can and cannot be easily achieved. To change the planner's behaviour, we need to create control rules that tell the planner when to attempt or avoid a task. Therefore learning events, $\mathcal{E}$, for this planner are actions related to task achievement, for example, missing or meeting a deadline, or acquiring or not acquiring an object.

Careful analysis of the domain model yields these learning opportunities. Most events correspond directly to operator applications, although execution monitors may also record events. For example, if the user meets the robot in the corridor and takes his mail, Rogue's execution monitor does two things: (i) it would indicate task completion in PRODIGY4.0's domain knowledge, and (ii) it would record an event of premature task completion. Additional events, not explored in this thesis, might include checking preconditions and postconditions, or other things relating to task completion that do not directly correspond to single operators.

Each time the task planner records the event in an execution trace, it requests current situational features from the execution module using the query described above.

It should be noted that events for the task planner are *much* more rare than for the route planner. Events can be collected constantly for the route planner, whereas the task planner only has relevant events when an action succeeds or fails. Progress towards the goal, such as acquiring a necessary object, is recorded as a successful event. Lack of progress is recorded when the corresponding action fails.

4.3 Costs

One possible approach for assigning event costs in a task planner involves considering task and user importance, and effort expended (travel plus wait time). For an event $\varepsilon \in \mathcal{E}$, a possible function could be:

$$\mathcal{C}(\varepsilon) = \begin{cases} [t_{dist} + t_{wait}] \times \frac{1}{Rank_{task} + Rank_{user}} & \text{if } \varepsilon \text{ was a success} \\ \\ k \times [t_{dist} + t_{wait}] \times [Rank_{task} + Rank_{user}] & \text{if } \varepsilon \text{ was a failure} \end{cases}$$

$Rank_{task}$ and $Rank_{user}$ are extracted from the task knowledge, and correspond directly to the event. The travel time, t_{dist}, would cover the time dedicated solely to this one event. k is a factor weighting failures much more expensively than successes; k's value would depend on the particular domain and how critical it would be to avoid failures in the future. Other possible functions might include penalties for missed deadlines, especially in domains when deadlines are more critical. Learned rules would be of the form *if cost-is-unreasonable, then avoid-task*.

In many planners, including ROGUE's task planner, this function can be reduced to a binary function, in which successes are assigned a cost of zero, while failures are assigned a cost of one:

$$\mathcal{C}(\varepsilon) = \begin{cases} 0 & \text{if } \varepsilon \text{ was a success} \\ 1 & \text{if } \varepsilon \text{ was a failure} \end{cases}$$

Rules are then of the form *if cost-is-one, then avoid-task*. The abstraction provided by this function simplifies the learning task, but loses some forms of knowledge; for example, we can no longer represent the concept *if taskA-is-easier-than-taskB, attempt taskA*.

The loss of representation is not important in our domain because ROGUE rarely has enough tasks that it needs to differentiate between them at such a fine-grained level. Moreover, the priority and compatibility calculations prune out the set of tasks considered by the learned rules.

In the same manner as for the path planner, the event is stored in an events matrix along with the cost evaluation and the environmental features observed when the event occurred.

4.4 Learning Algorithm

Following the general situation-dependent learning approach, we use regression trees to learn PRODIGY4.0 control rules for the task planner, as we did for Xavier's path planner. The regression tree algorithm is presented in Section 3.5. The input to the algorithm is an events matrix and the output is a set of learned regression trees from which control rules can be built.

For the task planner, we made one extension to the basic regression tree algorithm: we incorporate the cost of measuring the value of a feature.

Incorporating sensor values into the feature set raised the problem that low-level execution features are more predictive of performance than high-level features. However, the cost of calculating a value for an execution-level feature is *much* greater than for a high-level feature. For example, to calculate a sonar value, the robot would have to navigate to the location in which they are valid.

The regression tree algorithm therefore needed to consider feature costs when making a decision about splitting a node. Standard techniques exist to do this, e.g. Tan [1991], however S-PLUS does not support changing the default split function, which only considers node deviance.

We therefore re-implemented the regression tree analysis to cope with feature costs for the experiments described in Section 4.6.3. We select splits greedily to minimize total *Deviance*Cost* in the tree[2,3]. Since S-PLUS provides more data analysis tools and a graphical user interface, we prefer to use this package whenever possible. We therefore use S-PLUS to create the regression trees of Sections 4.6.1 and 4.6.2, because the costs of the features are constant within the category.

4.5 Creating Control Rules for the Task Planner

Once the set of regression trees has been created (one for each type of event), each tree needs to be translated into PRODIGY4.0 search control rules. Control rules can be used to focus planning on particular goals and towards desirable plans, and to prune out undesirable actions, as was described in Section 2.2.2.3.

There are two locations where learned control rules can be useful in ROGUE. The first type decide which tasks to focus on achieving; these are *goal selection* control rules. Goal selection rules aim at creating better plans. The second type decide what order to achieve

---

[2] This function is the closest regression-tree application of Tan's decision-tree technique.

[3] When costs of features do not vary, minimizing total *Deviance*Cost* is mathematically equivalent to S-PLUS's strategy of maximizing reduction in *Deviance*.

actions; these are *applicable operator* rules. Applicable operator rules aim at executing the plan more efficiently.

A control rule is created at each leaf node; it corresponds to the path from the root node to the leaf. We decide whether to make a rule a *select*, *prefer-select*, *prefer-reject* or a *reject* based on the learned value of the leaf node.

The training data identified the success or failure of an event, indicating each with a value of 0.0 or 1.0 respectively. Leaf nodes with values close to 0.0 are considered *select* rules, and those close to 1.0 are considered *reject* rules. Intermediate values become *prefer* rules. Table 4.2 summarizes the important decisions made when generating the rules.

Table 4.3 shows a sample tree learned for this domain. The tree indicates that between 10:00 and 20:00, tasks are more likely to succeed than at night (recall that CT is *Current-Time*, in seconds since midnight). There are four control rules created for this tree: one for each leaf node. They are shown in Table 4.4. Rules `auto-timeout-0` and `auto-timeout-3` are *reject* rules, while rule `auto-timeout-1` is a *select* rule. Rule `auto-timeout-2` is a *prefer-reject* rule because <G2> is preferred over <G>.

Tests in the search control rules are generated directly from the branch nodes of the tree. For example `(current-time LT 71749)` is generated from node 6 of Table 4.3, while (location <G> GT 5314.0000) is generated from node 13. A given rule will have the same number of tests as the depth of the corresponding leaf node. Notice that "redundant" tests may appear in a rule (`auto-timeout-3`); these occur when a given feature is used multiple times in the tree, each time with different values.

Each split in the tree is of the form *feature-comparison-value*. A meta-predicate function needs to be provided that will perform this comparison. For each feature in the domain, the corresponding meta-predicate determines how to extract the feature from the world. Table 4.5 shows the meta-predicate function that tests current time. Table 4.6 shows the

Let *execution-tests* be the set of all tests that may only occur in an *apply-op* rule
Let *leafval* be the learned value of the leaf node generating the rule. In the training data, *leafval* = 1.0 is a failure, *leafval* = 0.0 is a success.
Let *rule* be the set of tests generated by each of the branch nodes in the tree.

1. if ($execution\text{-}tests \cap rule$)
   /* this is an *apply-op* rule */
   else
   /* this is a *goal* rule */

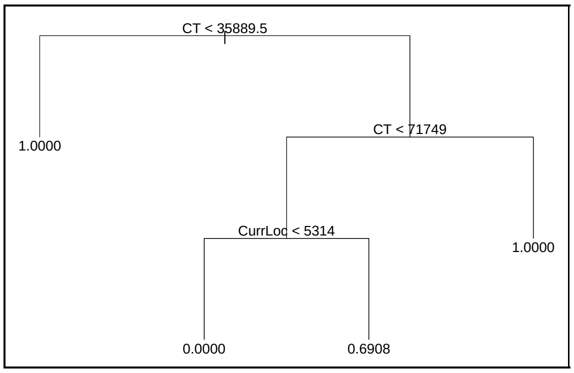
2. if ($0 \leq leafval < 0.25$)
   /* this is a *select* rule */
   else if ($0.25 \leq leafval < 0.50$)
   /* this is a *prefer-select* rule */
   else if ($0.50 \leq leafval < 0.75$)
   /* this is a *prefer-reject* rule */
   else
   /* this is a *reject* rule */

Table 4.2: Important tests for generating PRODIGY4.0 control rules from learned trees.

```
node), split, n, deviance, learned value
      * denotes terminal node

 1) root 856 186.70 0.6787
   2) CT<35889.5 264   0.00 1.0000 *
   3) CT>35889.5 592 147.30 0.5355
     6) CT<71749 418  94.08 0.3421
      12) CurrLoc<5314 211   0.00 0.0000 *
      13) CurrLoc>5314 207  44.21 0.6908 *
     7) CT>71749 174   0.00 1.0000 *
```

(a)



(b)

Table 4.3: A sample tree. (a) The text version, which lists the number of examples n, deviance value and learned value for each node. (b) The graphical version.

```
;;;Deviance is 0.0000 on value of 1.0000
;;;264 examples
(CONTROL-RULE auto-timeout-0
   (if (and (real-candidate-goal <G>)
            (current-time LT 35889)))
   (then reject goal <G>))

;;;Deviance is 44.2099 on value of 0.6908
;;;207 examples
(CONTROL-RULE auto-timeout-2
     (if (and (real-candidate-goal <G>)
              (current-time GT 35889)
              (current-time LT 71749)
              (location <G> GT 5314.0000)
              (real-candidate-goal <G2>)
              (diff <G> <G2>)))
     (then prefer goal <G2> <G>))
```

```
;;;Deviance is 0.0000 on value of 0.0000
;;;211 examples
(CONTROL-RULE auto-timeout-1
   (if (and (real-candidate-goal <G>)
            (current-time GT 35889)
            (current-time LT 71749)
            (location <G> LT 5314.0000)))
   (then select goal <G>))

;;;Deviance is 0.0000 on value of 1.0000
;;;174 examples
(CONTROL-RULE auto-timeout-3
     (if (and (real-candidate-goal <G>)
              (current-time GT 35889)
              (current-time GT 71749)))
     (then reject goal <G>))
```

Table 4.4: Learned PRODIGY4.0 control rules for the tree in Table 4.3.

meta-predicate function that tests sonar readings.

In addition to the *cost* of calculating a value for an execution-level feature, we must also consider the possibility that they would *change the state*. As discussed in Section 2.3.1, it is important to ensure that the external state is not modified while firing a control rule. For example, to collect a sonar value, the robot might have to change locations, potentially affecting the plan. For this reason, we do *not* use them in goal-selection rules; rules containing execution-level features are only used at the operator application stage.

```
(defun current-time (comparetype value)
  (let ((currtime nil))
    (multiple-value-bind (sec min hr date month year dayOfWeek)
                         (get-decoded-time)
        (setf currtime (+ (+ (* 3600 hr) (* 60 min)) sec)))
    (format t "~%Testing time ~S ~S ~S" currtime comparetype value)
    (if (eq comparetype 'user::LT)
      (if (< currtime value) t)
      (if (> currtime value) t))))
```

Table 4.5: The meta-predicate function for current time.

```
(defun sensor-test (sensor-num comparetype value)
  (let ((sonar (tca::SONAR_STATUS_TYPE-data SonarDataCache)))
    (format t "~%Testing (sensor ~S) ~S ~S ~S"
               sensor-num (aref sonar sensor-num)
               comparetype value)
    (if (eq comparetype 'LT)
      (if (< (aref sonar sensor-num) value) t)
      (if (> (aref sonar sensor-num) value) t))))
```

Table 4.6: The meta-predicate function for sonar readings.

4.6 Experimental Results

We conducted three experiments to test the ability to learn from execution experience to create knowledge for the task planner. Each experiment explores one of the classes of learned search control rules.

The first experiment explores using only execution-level features to create applicable operator rules. These rules are designed to improve *execution performance* of the task planner.

The second set explores using only high-level features to create goal selection control rules. These rules improve *planning performance* of the task planner.

The final set explores using both high-level and execution-level features to create both types of control rules: applicable operator and goal selection. We use feature costs to select splits in the regression trees.

All experiments were conducted in the simulator.

4.6.1 Experiment 1: Execution Features

The first experiment was designed to improve the task planner's *execution performance.* ROGUE uses execution-level features to create *applicable operator* search control rules. Recall that the planner selects which plan step to execute by choosing which plan step to apply. Hence, the task planner uses applicable operator rules to decide when to execute an action. Using current execution-level features of the domain allows the task planner to make execution more efficient.

The goal was to have ROGUE autonomously determine *"Should I wait?"* by detecting closed doors and inferring that the task could not be completed. In particular, we wanted ROGUE to learn an applicable operator search control rule that avoided applying an `(ACQUIRE-ITEM)` or `(DELIVER-ITEM)` operator when it sees that the door is closed.

We generated training data that had the robot fail at a task every time the door to a required location was closed. A map was generated for each trial run in which each door was opened with 50% probability. In each trial run, the task planner handled five user requests, randomly generated as described in Appendix B. Every door closure was treated as a timeout event for learning. ROGUE recorded a timeout event every time the average length of the front three sonars was less than 1.5m:

$$\frac{\text{Sonar23} + \text{Sonar0} + \text{Sonar1}}{3} < 150\text{cm}$$

Table 4.7 shows a sampling from the events matrix generated for this task. A total of 3987 events were recorded, of which 2419 (61%) were door-open events and 1568 (39%) were door-closed events. The *Status*, or cost, variable indicates whether the door was open; a value of 1 indicates that it was closed. Notice that *Status* does not depend on features such as time or location. A close examination of the data reveals that it depends on the three front-most sonar values.

The regression tree algorithm created rules that depend primarily on Sonar0 (the front-most sonar). Sonar1 appears regularly in the *unpruned* trees, as do several other sonar values. In the pruned trees, only Sonar0 appears; all other features were ignored, for all deviance levels. Figure 4.1 shows the unpruned and pruned trees generated for three deviance levels. Leaf values correspond to the learned value of the event in the situation described by each split. In the pruned trees, then, when sensor 0 has a reading of less than 150.144cm, the door is closed 81.96% of the time. When sensor 0 has a reading greater than 150.144cm, the door is open 99.76% of the time.

Table 4.8 shows the control rules generated from the pruned trees for PRODIGY4.0 according to the method described in Section 4.5. Notice that each leaf in the pruned tree forms a rule, and that the single split value on the path from the root is now a meta-predicate. The generated rules were used at operator application time, i.e. when the operator is released for execution, since sonar readings are relevant only at the robot's current location.[4]

Table 4.9 shows part of a trace generated by PRODIGY4.0 for two requests when using the control rules of Table 4.8. The door to 5302 was closed on the first visit (after node 31), and the control rule rejects applying the operator `<ACQUIRE-ITEM mitchell delivermail r-5302>`. The robot goes to room 5304, where the reject control rule does not fire[5], so PRODIGY4.0 applies the operator `<ACQUIRE-ITEM jhm deliverfax r-5304>`. The robot

[4] See Sections 2.3.1, 3.2, and 4.5 for more discussion on this point.

[5] The *select* control rule does not fire when it would not change PRODIGY4.0's default decision.

Status	CurrLoc	Year	Month	Date	DayOfWeek	CT	Who	WhoRank	Task	TaskRank	PickupLoc
0	5301	1997	9	29	1	81982	JAN	2	DELIVERFAX	2	5310
1	5301	1997	10	1	3	12903	WILL	5	DELIVERFAX	2	5302
0	5303	1997	10	1	3	23526	MITCHELL	3	PICKUPMAIL	6	5303
1	5336	1997	10	3	5	1200	JEAN	2	DELIVERMAIL	3	5336
0	5321	1997	10	4	6	8522	MCOX	4	DELIVERMAIL	3	5321
1	5304	1997	10	4	6	76641	THRUN	3	DELIVERFEDEX	1	5304
1	5415	1997	10	5	0	48733	REIDS	3	PICKUPCOFFEE	8	5415
0	5304	1997	10	6	1	9482	MMV	3	DELIVERMAIL	3	5304
0	5310	1997	10	6	1	76701	THRUN	3	PICKUPFAX	5	5320
1	5301	1997	10	7	2	22746	SKOENIG	5	DELIVERFAX	2	5313
1	5409	1997	10	8	3	1320	REIDS	3	PICKUPMAIL	6	5427
0	5427	1997	10	8	3	81802	JEAN	2	DELIVERFEDEX	1	5321
0	5303	1997	10	9	4	4141	SKOENIG	5	DELIVERMAIL	3	5301
1	5313	1997	10	10	5	83303	MMV	3	DELIVERFEDEX	1	5313
0	5409	1997	10	10	5	29768	JBLYTHE	5	DELIVERFEDEX	1	5409
1	5321	1997	10	11	6	78501	JRS	4	PICKUPMAIL	6	5310
0	5427	1997	10	12	0	19745	KHAIGH	5	PICKUPMAIL	6	5427
1	5307	1997	10	12	0	29888	MCOX	4	DELIVERMAIL	3	5307

DeliverLoc	Sensor0	Sensor1	Sensor2	Sensor3	...	Sensor20	Sensor21	Sensor22	Sensor23
5301	596.676	224.820	224.820	93.756	...	173.004	163.860	163.860	773.460
5301	81.564	84.612	84.612	90.708	...	166.908	157.764	157.764	154.716
5311	599.724	87.660	87.660	87.660	...	166.908	157.764	157.764	154.716
5302	78.516	78.516	78.516	84.612	...	261.396	243.108	243.108	243.108
5328	456.468	166.908	166.908	169.956	...	87.660	84.612	84.612	84.612
5313	148.620	148.620	148.620	160.812	...	96.804	90.708	90.708	87.660
5321	102.900	99.852	99.852	105.948	...	154.716	139.476	139.476	133.380
5303	773.460	133.380	133.380	142.524	...	115.092	108.996	108.996	108.996
5310	773.460	733.836	733.836	182.148	...	102.900	96.804	96.804	96.804
5301	81.564	81.564	81.564	87.660	...	169.956	154.716	154.716	154.716
5409	81.564	78.516	78.516	84.612	...	169.956	157.764	157.764	157.764
5427	773.460	773.460	773.460	90.708	...	176.052	169.956	169.956	173.004
5303	642.396	136.428	136.428	136.428	...	121.188	115.092	115.092	112.044
5313	130.332	136.428	136.428	148.620	...	112.044	105.948	105.948	105.948
5317	773.460	773.460	773.460	773.460	...	179.100	163.860	163.860	160.812
5321	145.572	148.620	148.620	160.812	...	96.804	90.708	90.708	90.708
5311	773.460	773.460	773.460	84.612	...	182.148	169.956	169.956	169.956
5415	148.620	148.620	148.620	163.860	...	93.756	90.708	90.708	87.660

Table 4.7: Sampling from the events matrix for the "Should I wait?" task. *Status* (cost) indicates whether the door was open; 1 is closed.

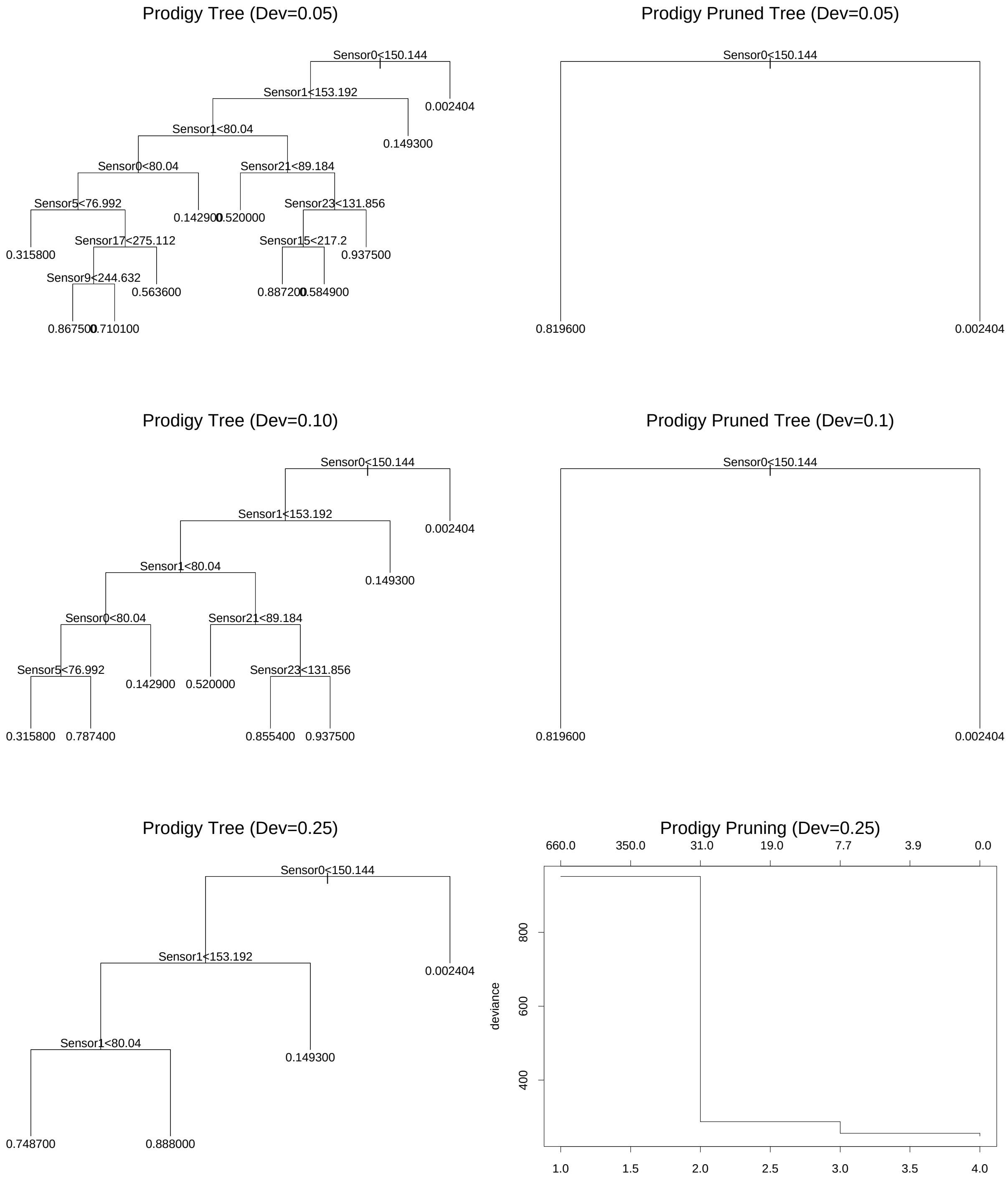


Figure 4.1: Regression trees learned for the “Should I wait?” task.

```
;;;Deviance is 208.40 on value of 0.827200 (1907 examples)
(CONTROL-RULE auto-timeout-0
        (if (and (candidate-applicable-op <OP>)
                 (or (inst-op-name-eq <OP> ACQUIRE-ITEM)
                     (inst-op-name-eq <OP> DELIVER-ITEM))
                 (sensor-test 0 LT 150.144)))
        (then reject apply-op <OP>))

;;;Deviance is 4.99 on value of 0.002404 (2080 examples)
(CONTROL-RULE auto-timeout-1
        (if (and (candidate-applicable-op <OP>)
                 (or (inst-op-name-eq <OP> ACQUIRE-ITEM)
                     (inst-op-name-eq <OP> DELIVER-ITEM))
                 (sensor-test 0 GT 150.144)))
        (then select apply-op <OP>))
```

Table 4.8: PRODIGY4.0 control rules generated for the learned pruned trees.

then returns to 5302, where the door has been opened, and it successfully acquires Mitchell's mail. If the door were still closed, ROGUE would deliver jhm's mail and return. ROGUE would continue returning until either the door was open, or the deadline was reached, or there were no more pending tasks.

Any action that is rejected by an applicable operator control rule will be reselected later in the trace, *assuming there are other pending tasks.* If there are other pending tasks, the planner will reconsider this task at each decision point. When it is no longer rejected by the control rule, the planner will attempt the action. When there are no other pending tasks, the planner will fail on the plan, rather than waiting for conditions to be more favourable; the implementation details are left for future work.

This experiment shows that ROGUE can use real-world execution data to learn when to execute actions. As a result, ROGUE learns to execute plans more effectively.

4.6.2 Experiment 2: High-level features

This experiment was designed to test ROGUE's ability to identify and use high-level features to learn to *create better plans.* The goal was to have ROGUE identify times for which tasks could not be completed, and then create goal selection rules of the form "reject task until..."

For training data, we generated two maps for the simulator. Between 10:00 and 19:59, all doors in the map were open. At other times, all doors were closed. When a door was closed, we defined the task as incompletable. We used a single route: from the starting location of 5310, go to room 5312 then to room 5316. The user remained constant and tasks were selected randomly from a uniform distribution. Table 4.10 shows a sampling of the data in the events matrix. The primary difference between this dataset and the dataset shown

```
Message from tca: "mitchell" 3 "delivermail" 2 "Oct 24 16:16" "Fri Oct 24 17:15"
                  "r-5302" "r-5316"
Message from tca: "jhm" 1 "deliverfax" 1 "Oct 24 16:16" "Fri Oct 24 17:15"
                  "r-5304" "r-5312"
  2 n2 (done)
  4 n4 <*finish*>
  5   n5 (has-item mitchell delivermail)
  7   n7 <deliver-item r-5316 mitchell delivermail>
  8   n8 (has-item jhm deliverfax)
 10   n10 <deliver-item r-5312 jhm deliverfax>
 11     n11 (robot-has-item jhm deliverfax)
 13     n13 <acquire-item r-5304 jhm deliverfax>
 14       n14 (robot-in-room r-5304)
 16       n16 <goto-pickup-loc jhm r-5304>
 17     n17 (robot-in-room r-5312)
 18     n18 goto-pickup-loc ...no choices for bindings (I tried)
 19     n20 <goto-deliver-loc jhm r-5312>
 22     n23 <acquire-item r-5302 mitchell delivermail>
 23       n24 (robot-in-room r-5302)
 25       n26 <goto-pickup-loc mitchell r-5302>
 26     n27 (robot-in-room r-5316)
 27     n28 goto-pickup-loc ...no choices for bindings (I tried)
 28     n30 <goto-deliver-loc mitchell r-5316>
 29       n31 <GOTO-PICKUP-LOC MITCHELL R-5302>
SENDING COMMAND (TCAEXPANDGOAL "navigateToG"
                          #(TASK-CONTROL::MAPLOCDATA 748.0d0 2083.0d0))
          Waited:2144 Total-to-wait:600000
          ...
          Waited:39032 Total-to-wait:600000
Action NAVIGATE-TO-GOAL-ACHIEVED finished.
SENDING COMMAND (TURN-TO-FACE R-5302)
Asking room location: "5302"
```

SonarDataCache:
#(127.536d0 142.776d0 188.496d0 389.664d0 648.744d0 319.56d0 209.832d0 164.11d0 142.776d0 136.68d0 ...)
Testing (sensor 0) 127.536d0 LT 150.144...T
Firing reject applied-operator AUTO-TIMEOUT-0 to remove
#<ACQUIRE-ITEM MITCHELL DELIVERMAIL R-5302>

(continued...)

Table 4.9: PRODIGY4.0 trace using the control rules of Table 4.8. The door of 5302 was closed at the first visit; PRODIGY4.0 returns to the room at node 35 (next page), and finds that the door has been opened.

```
30      n32 <GOTO-PICKUP-LOC JHM R-5304>
SENDING COMMAND (TCAEXPANDGOAL "navigateToG"
                               #(TASK-CONTROL::MAPLOCDATA 748.0d0 2672.0d0))
          Waited:619 Total-to-wait:600000
          ...
          Waited:13406 Total-to-wait:600000
Action NAVIGATE-TO-GOAL-ACHIEVED finished.
SENDING COMMAND (TURN-TO-FACE R-5304)
Asking room location: "5304"

 31       n33 (robot-in-room r-5302)
 33       n35 <goto-pickup-loc mitchell r-5302>

   SonarDataCache:
        #(596.928d0 87.912d0 84.864d0 90.96d0 100.104d0 124.488d0
                     176.304d0 456.72d0 596.928d0 310.416d0 ...)
   Testing (sensor 0) 596.928d0 LT 150.144...NIL

34     n36 <ACQUIRE-ITEM R-5304 JHM DELIVERFAX>
          ... "Please place Jim Morris's fax delivery on my tray."
          ... "Please indicate on my keyboard when you are finished."
          ... "Are you finished placing Jim Morris's fax delivery on my tray?"

 35       n37 <GOTO-PICKUP-LOC MITCHELL R-5302>
SENDING COMMAND (TCAEXPANDGOAL "navigateToG"
                               #(TASK-CONTROL::MAPLOCDATA 748.0d0 2083.0d0))
          Waited:809 Total-to-wait:600000
          ...
          Waited:21031 Total-to-wait:600000
Action NAVIGATE-TO-GOAL-ACHIEVED finished.
SENDING COMMAND (TURN-TO-FACE R-5302)
Asking room location: "5302"

   SonarDataCache:
        #(441.48d0 209.832d0 188.496d0 444.528d0 441.48d0 301.272d0 209.832d0
                    170.208d0 148.872d0 142.776d0 ...)
   Testing (sensor 0) 441.48d0 LT 150.144...NIL

 36     n38 <ACQUIRE-ITEM R-5302 MITCHELL DELIVERMAIL>
          ... "Please place Tom Mitchell's mail delivery on my tray."
          ... "Please indicate on my keyboard when you are finished."
          ... "Are you finished placing Tom Mitchell's mail delivery on my tray?"
```

Table 4.9: (cont) PRODIGY4.0 trace using the control rules of Table 4.8.

Status	CurrLoc	Year	Month	Date	DayOfWeek	CT	Who	WhoRank	Task	TaskRank
0	5312	1997	10	17	5	56235	KHAIGH	5	PICKUPFAX	5
0	5312	1997	10	17	5	56595	KHAIGH	5	PICKUPCOFFEE	8
1	5316	1997	10	18	6	660	KHAIGH	5	DELIVERFEDEX	1
1	5312	1997	10	18	6	1200	KHAIGH	5	DELIVERMAIL	3
1	5316	1997	10	19	0	86183	KHAIGH	5	DELIVERFEDEX	1
0	5312	1997	10	20	1	42851	KHAIGH	5	DELIVERFAX	2
0	5316	1997	10	20	1	42911	KHAIGH	5	DELIVERFAX	2

PickupLoc	DeliverLoc	Sensor0	Sensor1	Sensor2	Sensor3	Sensor4	Sensor5
5312	5316	96.804	99.852	99.852	108.996	108.996	139.476
5312	5316	773.460	105.948	105.948	112.044	112.044	136.428
5312	5316	81.564	81.564	81.564	87.660	87.660	115.092
5312	5316	72.420	72.420	72.420	78.516	78.516	93.756
5312	5316	84.612	81.564	81.564	87.660	87.660	108.996
5312	5316	90.708	93.756	93.756	102.900	102.900	136.428
5312	5316	72.420	72.420	72.420	78.516	78.516	99.852

Sensor6	Sensor7	...	Sensor18	Sensor19	Sensor20	Sensor21	Sensor22	Sensor23
139.476	182.148	...	313.212	148.620	148.620	142.524	142.524	139.476
136.428	166.908	...	291.876	145.572	145.572	133.380	133.380	133.380
115.092	142.524	...	188.244	166.908	166.908	154.716	154.716	154.716
93.756	115.092	...	212.628	182.148	182.148	166.908	166.908	166.908
108.996	133.380	...	200.436	166.908	166.908	154.716	154.716	154.716
136.428	179.100	...	169.956	151.668	151.668	148.620	148.620	148.620
99.852	124.236	...	221.772	328.452	328.452	166.908	166.908	163.860

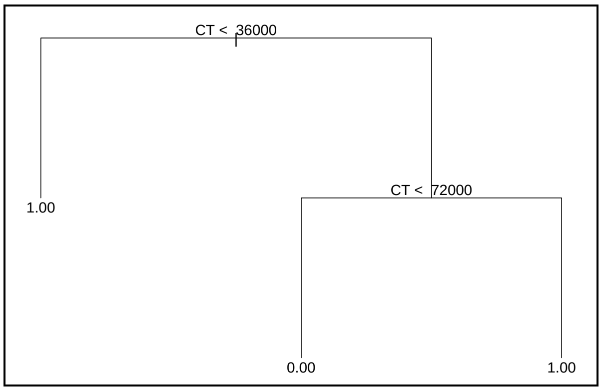
Table 4.10: Sampling from the events matrix for the "Reject until..." task. (Note: CT=36000 is 10:00, and CT=72000 is 20:00.)

in Table 4.7 is that the *Status*, or cost, variable was dependent on time, rather than being random.

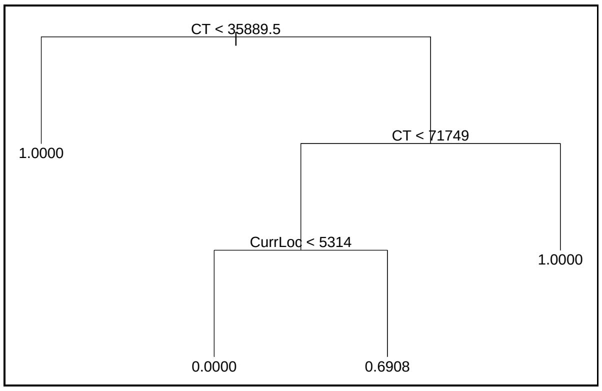
We ran the data through the learner, allowing the tree to be built using only high-level features of the environment.

We were expecting the learned tree to resemble the example shown in Figure 4.2a, in which time was the only feature used to build the tree. Figure 4.2b shows the actual regression tree learned for this data (the same pruned tree was created for all tested deviance levels).

The unexpected appearance of the feature *CurrLoc* caused us to re-examine the data. We found that there was indeed a difference between room 5312 and 5316. The Markov



(a) Expected Tree.



(b) Learned Tree.

Figure 4.2: Expected and actual trees for door-open times. (Note: CT=36000 is 10:00, and CT=72000 is 20:00.)

Room	Timeouts	No timeouts	Percentage Timeouts
5312	0	211	0%
5316	145	64	69%

Table 4.11: Timeout data between 10:00 and 20:00. The robot often does not find the door of 5316, and so the training data records a timeout event.

navigation module stopped the robot several feet away from the door of 5316 approximately 69% of the attempts, while centering the robot perfectly in front of 5312. Standing in front of a wall rather than at an open door caused the system to record a closed door and hence a failed task. Table 4.11 shows the exact success/fail data[6]. In the real robot, this failure rate is considerably reduced because the vision module is used to help centre the robot correctly on the door.

ROGUE created four control rules for PRODIGY4.0, one for each leaf node, shown in Table 4.4 (page 107). PRODIGY4.0 uses the reject control rules (0 and 3) to reject tasks before 09:58:09 and after 19:55:59. Rule 1 is used to select tasks between those times involving rooms "less than" 5314... namely room 5312. The prefer-reject control rule (rule 2) is used to prefer tasks *other* than those involving room 5316.

This experiment shows an inherent bias of the learning approach. In particular, the "true" feature that *should* have been used for learning is "distance-travelled-along-this-corridor." The learner would then have created a rule stating that the further the robot travels along a long, featureless corridor, the more likely that it will not stop at exactly the right location. Since this "true" feature was not part of the available feature set, the learner could not learn the "true" rule, and instead learned the best approximation. Automatically identifying features from the execution trace is an important open problem.

[6]There are more events at room 5312 because the robot occasionally gets trapped somewhere and does not arrive at room 5316.

This experiment also shows that ROGUE is able to use high-level features of the domain to learn situation-dependent control rules for Xavier's task planner. These control rules guide the planner's decisions towards tasks that can be easily achieved, while guiding the task planner away from tasks that are hard or impossible to achieve.

4.6.3 Experiment 3: Feature Costs & Combining High- and Execution-level Features

This experiment was designed to test ROGUE's response to feature costs. In Section 4.6.2 we explicitly eliminated execution-level features from the dataset. In these experiments, we tested preset costs for execution-level features, and allowed ROGUE to determine automatically whether rules were to be used as goal selection rules or applicable operator rules, as described in Section 4.5.

We expanded the dataset of Section 4.6.2 to include the five most probable Markov nodes, and collected additional data. We expected the learned trees to look much like the one in Figure 4.2b, with additional features splitting the most ambiguous node, as in Figure 4.3. We expected these additional split(s) to involve execution-level features, most probably sonar values or Markov states.

We set the cost of all features to be 1.0, and ran the modified regression tree algorithm described in Section 4.4. The learned, pruned tree is shown in Table 4.12. The primary split is on the value of Sonar 0 (the front-most). While it is clear that the robot's sensor values affect task completion, this rule is not valuable for the task planner since it cannot be used to make goal decisions.

We then increased the value of the execution-level features. At a cost value of 2.0, the learned pruned tree eliminated all execution-level features below the root node, but Sonar 0 remains the most important feature (Table 4.13). A cost of 3.0 yields the same tree.

A cost of 3.25 yielded the tree shown in Table 4.14. It is essentially the same tree shown

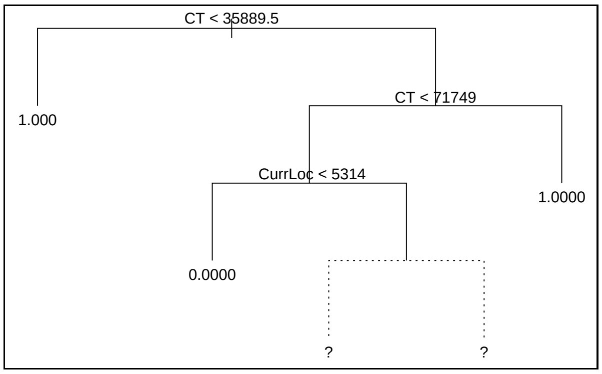


Figure 4.3: Expected tree for door-open times with all features. We expected the dotted split to contain execution-level features such as Markov nodes or sonar values. (Note: CT=36000 is 10:00, and CT=72000 is 20:00.)

```
node), split, (n, deviance), learned value
      * denotes terminal node

  1) root   (n=2670, dev=379.4267)   0.8285
   2) Sensor0 < 86.136   (n=2265, dev=74.3813)   0.9660
     4) MN1prob < 0.24371   (n=1939, dev=0.9995)  0.9995       *
     5) MN1prob > 0.24371   (n=326, dev=58.2821)   0.7669
       10) MN4 < 47   (n=16, dev=0.0000)  0.0000               *
       11) MN4 > 47   (n=310, dev=48.3870)   0.8065
         22) Sensor17 < 238.536  (n=210, dev=19.6952)  0.8952  *
         23) Sensor17 > 238.536  (n=100, dev=23.5600)  0.6200  *
   3) Sensor0 > 86.136   (n=405, dev=22.5777)   0.0593         *
```

Table 4.12: Learned tree for both high-level and execution-level features at cost 1.0.

```
node), split, (n, deviance), learned value
      * denotes terminal node

1) root   (n=2670, dev=379.4267)   0.8285
 2) Sensor0 < 86.136   (n=2265, dev=74.3813)   0.9660
   4) CT < 37930   (n=1335, dev=0.0000)  1.0000                 *
   5) CT > 37930   (n=930, dev=70.6250)   0.9172
     10) CT < 71629   (n=316, dev=58.2374)   0.7563
       20) CurrLoc < 5314   (n=17, dev=0.0000)  0.0000          *
       21) CurrLoc > 5314   (n=299, dev=47.9599)   0.7993
         42) Date < 5.5   (n=35, dev=8.1714)  0.6286            *
         43) Date > 5.5   (n=264, dev=38.6326)   0.8220
           86) TaskRank < 3.5   (n=94, dev=17.3723)   0.7553   *
           187 TaskRank > 3.5   (n=170, dev=20.6117)   0.8588 *
     11) CT > 71629   (n=614, dev=0.0000)  1.0000               *
 3) Sensor0 > 86.136   (n=405, dev=22.5777)   0.0593            *
```

Table 4.13: Learned tree for high-level features at cost 1.0, execution-level features at cost 2.0.

in Figure 4.2b; in particular it does not involve and execution-level features as had been expected. (The slightly higher value for the leaf corresponding to $CurrLoc > 5314$ reflects the additional data accurately.)

Although this experiment did not show any conclusive results, it is important for a learning algorithm to consider feature costs. The learning algorithm could then trade off the benefit of the information with the cost of acquiring the feature's value. There may indeed be situations when a little effort would give the planner a lot of benefit.

It is still important to remember, however, that a control rule *can not* change the external state. All actions that change the external state should be relegated to operator descriptions so that the planner can explicitly reason about their effect on plans.

```
node), split, (n, deviance), learned value
      * denotes terminal node

1) root   (n=2670, dev=379.4267)   0.8285
 2) CT < 36130   (n=1326, dev=0.0000)  1.0000                  *
 3) CT > 36130   (n=1344, dev=301.9224)   0.6592
   6) CT < 71989.5   (n=733, dev=171.8278)   0.3752
     12) CurrLoc < 5314   (n=368, dev=0.0000)  0.0000          *
     13) CurrLoc > 5314   (n=365, dev=67.8081)   0.7534        *
   7) CT > 71989.5   (n=611, dev=0.0000)  1.0000               *
```

Table 4.14: Learned tree for high-level features at cost 1.0, execution-level features at cost 3.25.

4.7 Summary

ROGUE is the only system we are aware of that learns through interaction with a real environment with noisy sensors and actuators, and exogenous events.

The regression tree algorithm is well-suited for learning search control rules for PRODIGY4.0 in this domain. Through its statistical analysis of the data, it is less sensitive to noise and exogenous events. Moreover, the symbolic representation of the features in the trees leads to an easy translation to search control rules.

Our experiments show that situation-dependent rules are a useful extension to the task planner. They create better plans because they guide the planner away from hard-to-achieve tasks, and towards easy-to-achieve tasks. They also reduce execution effort by learning when to execute the action.

The experiments in this chapter illuminate three important issues. The first is that the learning algorithm can learn an incorrect rule if the correct feature is not available since the hypothesis space can only be described in terms of the available features. Automatically extracting additional features from the data is an important open problem.

The second issue is that execution-level features are useful for making decisions about when to execute an action, but it is important to ensure that they do not change the state.

The third issue is that the learning algorithm should not create rules that ignore the cost of calculating the current value of a feature. Some features, for example vision processing, are considerably more expensive to acquire than others. The learned rules should make a reasonable tradeoff between the information gained and the cost of acquiring that information.

In addition to highlighting these issues, the specific contributions of this chapter are

- to demonstrate that our situation-dependent learning approach is planner-independent;
- to show how execution data can be incorporated into a symbolic task planner;
- to highlight the difference between learning to create a better *plan* and learning to *execute* the plan more effectively; and
- to demonstrate one method for deciding when to execute an action.

Chapter 5

Related Work

This section describes research closely related to that presented in this thesis. Our work contributes to the planning community, the machine learning community and the robotics community. In Section 5.1, we present work related to the task planner. In Section 5.2 we present research related to our learning framework.

5.1 Task Planning

There are a few approaches to creating plans for execution. Shakey [Nilsson, 1984] was the first system to use a planning system on a robot. This project was based on a classical planner which ignored real world uncertainty [Fikes *et al.*, 1972] and followed a deterministic model to generate a single executable plan. When execution failures occurred, replanning was invoked.

This pioneering approach has been acknowledged as partly successful, but also has been criticized for its lack of reactivity, and has led to significant research into planning systems that can handle the uncertainty of the real world. *Conditional planning* is one approach that aims at considering in the domain model all the possible contingencies of the world and plan ahead for each individual one [Atkins *et al.*, 1996; Mansell, 1993; Pryor, 1994; Schoppers, 1989]. In most complex environments, the large number of possible contingencies means that complete conditional planning becomes infeasible, but may nevertheless be appropriate in particularly dangerous domains.

Probabilistic planning takes a more moderate approach in that it only creates conditional plans for the most likely problems [Blythe, 1994; Dean & Boddy, 1988; Gervasio & DeJong, 1991; Kushmerick *et al.*, 1993]. It relies on replanning when unpredictable or rare events take place. Although this approach generates fast responses to most contingencies, it may miss potential opportunities that arise from changes in the world. It should be noted that none of these systems have ever been applied to a real robotic system.

Another moderate approach is that of *parallel planning and execution*, in which the planner and the executor are decoupled [Drummond *et al.*, 1993; Lyons & Hendriks, 1992; McDermott, 1992; Pell *et al.*, 1997]. The executor can react to the environment without a plan. The planner continually modifies the behaviour of the executor to increase the goal satisfaction probability. This approach leads to a system with fast reactions, but a set of default plans need to be pre-prepared, and in some situations may lead away from the desired goal. Furthermore, the planner creates its plans based on assumptions about the world that may have changed during planning time.

We take a third approach: that of *interleaving planning and execution*, as do several other researchers [Ambros-Ingerson & Steel, 1988; Dean *et al.*, 1990; Georgeff & Ingrand, 1989; Nourbakhsh, 1997]. Interleaving planning with execution allows the system to reduce its planning effort by pruning alternative possible outcomes immediately, and also to respond quickly and effectively to changes in the environment. For example, the system can notice limited resources such as battery power, or notice external events like doors opening and closing. In these ways, interleaving planning with execution can create opportunities for the system while reducing the planning effort.

One of the main issues raised by interleaved planning and execution is when to stop planning and start executing. Dean *et al.* [1990] selects between alternative actions by selecting the one with the highest degree of information gain, but is therefore limited to reversible domains. Nourbakhsh [1997] on the other hand, executes actions that prefix all branches of a conditional plan created after making simplifying assumptions about the world. The assumptions are built so that the planner always preserves goal reachability, even in an irreversible world. Gervasio & DeJong [1991] take a slightly different approach by creating general complete plans upfront, and then at execution time uses sensor information to select between alternative actions.

ROGUE has three methods for selecting when to take an action. The first method selects an action when it is the first in a chain of actions that are known to lead towards the goal. PRODIGY4.0 uses means-ends analysis to build plans backwards, working from the goal towards the initial state. Each action is described in terms of required preconditions and possible effects; actions are added to the plan when their effects are desirable. When all the preconditions of an action are believed to be true in the current state, ROGUE executes the action. Since PRODIGY4.0 already has a partial plan from the initial state to the goal state, the action ROGUE selects is clearly relevant to achieving the goal. Actions whose failures may lead to irreversible states are avoided until it has exhausted all other possible ways of reaching the goal.

The second method is used when there are multiple actions available for selection. ROGUE selects between these actions to maximize overall expected execution efficiency.

The third method ROGUE uses to decide when to execute an action is through *learning*. It collects execution data and statistically estimates when it would be beneficial to execute the action.

When ROGUE selects an action for execution, it executes a procedure that confirms the preconditions of the action, then executes the action, and finally confirms the effects. In addition to the explicit confirmation of preconditions and effects of actions, our system also monitors events that may affect goals. Each goal type has a set of associated monitors that are invoked when a goal of that type enters the system. These monitors run parallel to planning and may modify the planner's knowledge at any time. A given monitor may, for example, monitor battery power or examine camera images for particular objects.

The ability to handle asynchronous goals is a basic requirement of a system executing in the real world. A system that only handles asynchronous goals in a first-come-first-served manner is inefficient and loses many opportunities for combined execution. ROGUE easily incorporates asynchronous goals into its system without losing any context of existing tasks, allowing it to take advantage of opportunities as they arise. By intelligent combining of compatible tasks, ROGUE can respond quickly and efficiently to user requests.

Amongst the other interleaving planners, only PRS [Georgeff & Ingrand, 1989] handles multiple asynchronous goals. ROGUE however abstracts much of the lower level details that PRS explicitly reasons about, meaning that ROGUE can be seen as more reliable and efficient because system functionality is suitably partitioned [Pell *et al.*, 1997; Simmons *et al.*, 1997]. NMRA [Pell *et al.*, 1997] and 3_T [Bonasso & Kortenkamp, 1996] both function in domains with many asynchronous goals, but both planners respond to new goals and serious action failures by abandoning existing planning and restarting the planner. As stated by Pell *et al.* [1997], establishing standby modes prior to invoking the planner is "a costly activity, as it causes [the system] to interrupt the ongoing planned activities and lose important opportunities." Throwing out all existing planning and starting over not only delays execution and but also can place high demands on sensing to determine current status of partially-executed tasks.

RAPs [Firby, 1994; Firby, 1989], like TCA, is an architecture that enables a library of behaviours and reactions to be controlled by a deliberative system. RAPs and TCA have been used as the underlying control mechanism on a variety of robots, from indoor mobile robots [Gat, 1992; Simmons *et al.*, 1997] to outdoor legged robots [Simmons, 1991] to planetary rovers [Krotkov *et al.*, 1995] and spacecraft [Bonasso & Kortenkamp, 1996]. TCA provides facilities for scheduling and synchronizing tasks, resource allocation, environment monitoring and exception handling. RAPs allow you to specify the methods and context for actions, and can therefore be constructed to provide the same facilities as TCA.

Neither architecture inherently contains a planner. RAPs in particular was explicitly designed to interact with a planner:

> It defines a well-structured, flexible and extensible mechanism for describing modular behaviors that can be both executed and reasoned about... the ability to reason about them independently provides a hook for interfacing the system to more deliberative planning and problem solving processes.
>
> [Firby, 1989]

To build a set of RAPs, the programmer must explicitly account for all goal interactions, pre-determine all preference rankings between actions, and eliminate all accidental paths to dangerous states. Attaching a planner to RAPs gives the system the ability to simulate actions, to reason about goal interactions, preferences and dangerous states. These abilities reduce the programmer's effort because the programmer can specify the building blocks from which a RAP can be constructed. A RAP can be viewed as a fully worked out plan; Rizzo *et al.* [1997][1998] present methods to automatically translate PRODIGY4.0 plans intro RAPs.

The behaviours demonstrated by ROGUE under TCA could be easily transferred to another robot control architecture.

5.2 Learning

Although there is extensive machine learning research in the artificial intelligence community, very little of it has been applied to real-world domains. Common applications include map learning and localization (e.g. [Koenig & Simmons, 1996; Kortenkamp & Weymouth, 1994; Thrun, 1996]), or learning operational parameters for better actuator control (e.g. [Baroglio *et al.*, 1996; Bennett & DeJong, 1996; Pomerleau, 1993]). Instead of improving low-level *actuator* control, our work focusses instead at the *planning* stages of the system.

In this section, we describe some of the work related to our learning approach. There are three primary groups of related work:

- learning action costs from a real-world environment,
- learning symbolic descriptions of actions, and
- learning plan quality.

5.2.1 Learning Action Costs

The situation-dependent rules that ROGUE learns for the path planner determine arc traversal costs. Other researchers have also explored the area of learning action costs.

CSL [Tan, 1991] and Clementine [Lindner *et al.*, 1994] both learn sensor utilities, including which sensor to use for what information. CSL represents very early work in the area, since its "sensors" were actually features of the object, e.g. the "height-sensor." The approach, however, is general, and it is clear that learning is a good method for predicting sensor reliability. Clementine explicitly uses utility theory to define the tradeoff between sensor cost and sensor reliability, and is applied to multiple sensors on a mobile robot. Even though they explicitly state "the ultrasonic sensors were reliable for other settings, they are less desirable for sensing [glass]," they do not incorporate situation-dependent features in their utility estimates.

LIVE [Shen, 1994] learns a model of the environment, as well as the costs of applying actions in that environment. For example, it can learn that a particular corridor has a higher-than-average cost. It does not, however, consider the possibility that costs may change according to a predictable pattern.

Haigh *et al.* [1997a] used situational features in a case-based reasoning system to assign costs to cases. Their route planning system used these costs to select a good set of cases for planning under the given conditions. Our current approach essentially assigns costs at a finer-grained level, that of the actions rather than of a set of consecutive actions.

Reinforcement Learning (overviewed by Kaelbling *et al.* [1996]) learns the value of being in a particular state, which is then used to select the optimal action. This approach can be viewed as learning the integral of action costs. However, most Reinforcement Learning techniques are unable to generalize learned information, and as a result, they have only been used in small domains.

Recently, several research have been exploring techniques for allowing generalization in Reinforcement Learning [Baird, 1995; Boyan & Moore, 1995; McCallum, 1995]. Essentially, these systems replace Reinforcement Learning's standard table-lookup mechanism with alternative function approximation techniques, such as decision trees or neural networks. Experimentally, these algorithms seem to produce reasonable policies. However, they may be very computationally intense since a single generalization might require the entire space to be recalculated.

Moreover, Reinforcement Learning techniques typically learn a universal action model for a *single* goal. Our situation-dependent learning approach learns knowledge that will be transferrable to other similar tasks.

5.2.2 Learning Symbolic Descriptions of Actions

Situation-dependent rules control the applicability of actions as a function of the current features of the environment. In the artificial intelligence community, several researchers have explored techniques for learning or changing action models. Most of these systems rely on complete and correct sensing, in simulated environments with no noise or exogenous events.

OBSERVER [Wang, 1996] and ARMS [Segre, 1991] learn action models by observing another agent's solution; they rely on complete observation of the environment and external agents or noise. Learning is assumed to be correct and irreversible. EXPO [Gil, 1992] learns operators by experimentation; it designs experiments, and explicitly monitors effects in environment. It also assumes complete and immediate sensing with no external events or noise.

Learning in real world domains, however, cannot utilize techniques that rely on closed-world assumptions such as complete observation, single agents, or exogenous events.

LIVE [Shen, 1994], like EXPO, also uses experimentation to learn a model of the environment. It extends EXPO's abilities by learning stochastic effects from incomplete sensing, but does not handle environments with noise or exogenous events.

IMPROV [Pearson, 1996] is one system which relaxes the assumption about complete and correct sensing, but still manages to learn operator descriptions. The planner learns through experimentation, by trying alternative operators until it achieves a success. It then compares the successful episode with the failures, and modifies operators to compensate for the errors.

Performance in IMPROV degrades dramatically with the noise introduced from sensing, but remains better than the system without learning of any kind. Part of the reason for this degradation is because the system uses only training data generated from the most recent version of the operator. Changing the operator means that old data is invalidated, and hence must be ignored. As a result, the system cannot explicitly identify and eliminate noise through analysis of long term trends in the data. In ROGUE, the operators remain constant, while search control rules change. As a result, data remains valid over the lifetime of the robot, and ROGUE can statistically identify and eliminate noise from the large body of data.

Although both IMPROV and LIVE aim at relaxing the closed-world assumptions made by most artificial intelligence learning systems, neither has been applied to a real-world robotics domain. The difficulties posed by real-world domains have generally limited learning to action parameters, such as manipulator widths, joint angles or steering direction. For example, Grant & Feng [1993] built a system that also tunes parameters in for grasping actions; Zhao *et al.* [1994] use genetic algorithms to find an optimal sequence of base positions and manipulator configurations to perform a series of different manipulation tasks on a mobile manipulator; Pomerleau [1993] uses neural networks to select good steering directions in an autonomous land vehicle. Bennett & DeJong's [1996] *permissive planning* paradigm tunes parameters in actions.

Salganicoff & Ungar[1995] built another system that learns action parameters for a manipulator arm. It uses a decision tree approach similar to ours, in that it uses perceptual features to discriminate between actions, with the goal of maximizing action probability. However, their features are strictly a function of the object being grasped, such as height, length, and width, while our system uses high-level features of the domain.

ROGUE learns patterns in the environment that affect planning. Mitchell *et al.* [1994] also built a system that learned patterns from the environment. The CAP system scheduled meetings after learning patterns in the environment for determining meeting location, duration and day of week.

5.2.3 Learning Plan Quality

The above-mentioned systems all learn action models, focussing on operator correctness rather than planning efficiency or plan quality. ROGUE does not learn action models; it assumes that actions are correct, but that their costs or applicability may vary according to the task and the environment. This applicability function is implemented in the form of search control rules. Search control rules can be viewed as equivalent to learning preconditions for action models, but for the reasons outlined in the above description of IMPROV, we feel that control rules are more appropriate in this domain.

Most of the research towards learning search control rules has focussed on making planning more efficient, rather than on making better quality plans. In the robot control domain, execution efficiency is extremely important, while planning efficiency is much less so. As pointed out by Kibler [1993], the major concern for real-world problems is the quality of the solution and not the speed at which the solution is reached.

PYRRHUS [Williamson & Hanks, 1994] supports the generation of high quality plans through the use of utility functions. Hand-built domain-dependent control rules use the utility function to determine which plans to expand and which flaws to fix. The system learns neither the control rules nor the utility function.

QUALITY [Pérez, 1995] learns control rules to generate high quality plans, where quality can be defined in terms of execution cost, reliability or user satisfaction, and operators may have different costs. It relies on a comparison of pairs of complex plans to learn control rules that bias the planner towards the higher quality plan. New learned knowledge overrides previous knowledge, but noise is not accounted for.

HAMLET [Borrajo & Veloso, 1994] learns control rules that improve planning efficiency and the quality of plans generated. It assumes that all operators have equivalent cost. It relies on training the system with simple problems for which it can find optimal solution(s), and then uses bounded explanation and induction to learn control rules. Rules are incrementally refined and with more training examples will converge towards a possibly disjunctive set of correct rules. Noise is also not accounted for in this system.

CHEF [Hammond, 1987], PRODIGY/ANALOGY [Veloso, 1994] and Haigh & Veloso [1997a] use analogical reasoning to create plans based on past successful experiences, where the belief is that past success might help lead to future success. Only Haigh & Veloso's route planning system *explicitly* aims at creating better quality plans; it assigns situation-dependent costs to cases with the goal of selecting the best case for the given user under the given traffic conditions. Noise and exogenous events are not handled in any of these systems; all successful cases are stored.

ROGUE uses statistical analysis of real world execution to create situation-dependent control rules for the task planner. These control rules guide the planner towards more efficient plans in which failures can be predicted and avoided. Statistical analysis and incremental learning allow ROGUE to explicitly account for noise in both its sensors and its actuators. Exogenous events that affect planning are explicitly identified and incorporated into the search control rules.

Chapter 6

Conclusion

In this thesis we have examined the problem of combining planning, execution and learning within a real-world environment.

A planning and executing agent can perform more tasks than a simple reactive executing agent because it can reason about tradeoffs between various system requirements. Learning abilities increase the flexibility and efficiency of the system because it can autonomously respond to changes in the environment.

We presented a robotic system, ROGUE, that creates and executes plans in a complex, real-world environment, and then learns from its experiences to improve both planning and execution performance.

ROGUE's task planner handles asynchronous requests from multiple users in an office delivery environment. ROGUE reasons about task priority and task compatibility to create interleaved plans that minimize execution cost. ROGUE monitors the execution of plan steps, and detects and responds to action failures, exogenous events in the environment, and unexpected side-effects of actions. ROGUE represents a specific instantiation of a general framework for interleaving planning and execution in a complex, dynamic domain. The specific scientific contributions of the task planner include:

- ◇ The transparent incorporation of asynchronous goals into planning.
- ◇ The ability to create plans for multiple interacting goals, taking into account task priority and compatibility.
- ◇ The ability to suspend and reactivate tasks when necessary.
- ◇ The ability to detect and respond to failures, unexpected side-effects of actions, and changes in the environment.
- ◇ The development of an interleaved planning and real robot execution architecture, including the development of a communication mechanism between the planner and the executor.

ROGUE's learning system collects data from the real-world execution of plans to improve the quality of generated plans. The planner-independent approach relies on extracting learning opportunities from the execution traces, evaluating them according to a pre-defined cost function, and then correlating them with features of the environment. ROGUE learns situation-dependent rules that affect the planners' decisions. ROGUE's learning approach was demonstrated for two planners: a path planner and the task planner. Extensive empirical results were presented, demonstrating both the effectiveness and utility of the approach. Specific scientific contributions of our learning approach include:

- ◇ The improvement of plans through examination of real-world execution data.
- ◇ The design of *situation-dependent rules* which set action costs at planning time as a function of situational features.
- ◇ The implementation and proof-of-concept of the domain-independent approach for two different planners, along with extensive empirical results.
- ◇ The design of a general framework for learning across representations, in which execution data representation differs widely from planning representations.
- ◇ The demonstration of system adaptability to a changing domain through incremental refinement of learned rules.
- ◇ The development of techniques for handling noisy, probabilistic training data.
  - A modification to Viterbi's algorithm that that generates abstract sequences in Markov models with additional uncertainty variables such as time or length.
- ◇ The distinction between learning to improve planning and learning to improve execution.

A single learning paradigm successfully learns arc costs for the path planner and also learns control knowledge for the task planner. It guides both planners away from hard-to-achieve tasks, and towards easy-to-achieve tasks. Better quality plans are thus generated, leading to greater system efficiency and effectiveness.

Since the learning approach is planner-independent, it is usable from any execution module to any planner, regardless of data representations. The designer must specify how to extract relevant learning opportunities from the execution data, and how to use the learned information within the planner.

6.1 Important Issues

Several important issues were identified through the research conducted for this dissertation.

6.1.1 Planning

Which module is in control: the planner, or the executor. In ROGUE, the task planner remains in control of the robot at all times. It sends *one action at a time* to the robot, deciding which action to execute next after the previous one has completed. The alternative approach is to commit earlier to the execution order, sending a *sequence* of actions to the executor, which then constrains their execution ordering.

The primary benefit of this interaction is that *dynamic reordering of actions and eventual replanning* is extremely easy: the task planner does not have to retract actions when side-effects make them redundant, or when failures require replanning or reordering of actions, or when a compatible asynchronous request is received. Another benefit of this approach is that a learning opportunity is created: the system can learn *when to send an action for execution.*

An additional benefit is that the task planner can reduce planning effort because it doesn't have to explicitly plan for contingencies; it can wait for feedback from execution.

One of the main drawbacks of this approach is that it is hard to represent actions that need to occur in *parallel.* The designer would need to put them all in one operator, since one action is sent at a time. If the multiple actions don't need to be explicitly synchronized, then another possibility would be to represent them each with instantaneous "start" and "end" operators.

A second drawback is that, since the planner is not exploring contingencies until they occur, it may have a slower response time than an executor with a complete plan.

Because of the asynchronous goals in our domain, the ability to dynamically reorder actions is *very* important. Since our domain is not very dangerous, and does not have hard real-time constraints, the potentially slower response time and difficulty representing parallel actions are non-critical issues.

How much planning to do before executing. ROGUE's task planner creates as much of the plan as possible before executing. ROGUE then uses control rules, both pre-coded and learned, to decide on the execution order of available actions, and also to incorporate any new asynchronous goals. The task planner will not plan beyond a point where the outcome of an action affects planning, such as at conditional branches.

Other methods include (i) eagerly executing, namely executing an action at the first opportunity, effectively making it a reactive system, (ii) creating a fully conditional plan up-front, and invoking replanning if unexpected conditions arise, (iii) ensuring that all branches of a conditional plan have the same initial actions, and (iv) executing a default plan, allowing the planner to update it in an "anytime" manner.

Which method to choose depends on the type of planner and the requirements of the domain. A dangerous domain, for example, would need more foresight than an eager execution policy would provide. A rapidly changing domain may need a very tight interaction between the planner and the executor, such as in the anytime method. A conditional planner may not be able to find a universally common operator domain with many possible actions, and hence be unable to make any forward progress.

ROGUE's method works well in our domain because there are few dangerous actions and they can be easily avoided, and it is acceptable to have a slower reaction time to action failures.

Whether to plan while executing. ROGUE's task planner does not plan while executing (although it will accept new goals while executing). The main benefit of this ability is to reduce overall planning and execution time: the planner can explore contingency actions, thereby reducing its response time when the outcome of an action is determined.

In a more dangerous domain, or if it takes a long time to plan a failure recovery, a planner would need to have this ability.

What knowledge should be put into control rules and what into operators. It could be argued that control rules should contain only meta-level control that *only depends on properties of the domain*, not on the current state. For example, train travel is preferred between cities, while vehicle travel is preferred within a city. Operators, meanwhile, would encode all information that depends on the current state.

ROGUE's task planner instead describes operators in very abstract terms, using control rules to reorder them or refine their applicability. For example, the task planner puts the TSP information into a control rule, since it affects the *quality* of the plan but not the *correctness*. However, since the rule depends on the current state of the robot, alternative approaches might directly put that information into the operators.

We believe that control rules are useful for both *planning* efficiency as well as *execution* efficiency. This belief is reflected in the fact that ROGUE learns *control rules* for planning and execution quality, instead of modifying operators.

Putting execution into control rules. In PRODIGY4.0, search control rules have traditionally relied on the *internal* model of the state. ROGUE, however, often senses directly from the external environment, as described in Chapter 4.

While there are no implementation limitations on what execution can happen while firing a control rule, conceptually the rule should *not modify* the external state. Such behaviour should be relegated to operators so that the planner can explicitly reason about their interaction with the rest of the plan. Control rules should only sense the state, and it is the designer's responsibility to ensure that they do not modify it.

How much "recovery" to put into an operator. The execution paradigm in ROGUE's task planner does not allow complex recovery mechanisms, since we believe the planner should reason about when they should be used, and how they will affect the remaining plan. Different recovery procedures may have different costs, reliabilities, or relevance, and it may be important to reason about the tradeoffs.

Incorporating complex recovery procedures into the command sequence for an action requires the designer to explicitly account for all goal interactions, pre-determine all preference rankings between recovery methods, and ensure the elimination of all accidental paths to dangerous states.

By extracting complex recovery methods, the designer can reduce his effort to build the domain. The designer can specify the individual *building blocks*, while the planner reasons about their interactions.

6.1.2 Learning

How to extract learning opportunities, and designing the system to exploit them. Learning opportunities for any planner can be identified by asking the question: "*What will change the planner's behaviour?*"

The path planner makes decisions based on estimates of the arc's length, blockage probability and traversal weight. Therefore improved estimates of these factors would improve the planner's performance. The task planner, meanwhile, makes decisions based on operator descriptions and control rules that affect goal and action selection. Therefore improved descriptions – correctness, costs, or probabilities – about tasks and actions would aid the planner in improving plans.

It is important to design the planner so that learned information can be seamlessly incorporated. Adding control rules to PRODIGY4.0, required no changes to the internal algorithm. When we added learned arc costs to the path planner, however, we had to modify some of its internal structures (data and control) to support the changes. Adding learned sensor reliabilities to the POMDP navigation module would require a massive effort to change the way these probabilities are stored and used in the code. One of the lessons learned in this thesis is that it is important to make the critical components accessible to external modules.

How to identify features for learning. Features of the environment are used to discriminate between different learning events. It is crucial to find a good set of relevant features, since the hypothesis space can only be described in terms of the available features.

A good feature will have the following characteristics: it is *easy to detect*, in terms of accessibility and cost; it is *informative*, so that the system doesn't waste time gathering information about irrelevant features; and it is *projective*, in that gathered information at one moment can help the system make decisions about the future.

If critical features are omitted, then the learner will not converge on the correct target function. It is an important open problem to autonomously extract relevant features from the data.

It is also important to design the system so that new features can be added at any time. In ROGUE there are several missing features, including the distance travelled since the last turn, the length of time since the battery was last recharged, and the length of time since the batteries (or other equipment) were last replaced. As we identify additional relevant features, the learner should seamlessly incorporate them into the data and learned information. This design consideration will be more important when systems are capable of autonomously identifying relevant features.

Putting execution-level features into control rules. By definition, execution-level features need to have their value calculated through direct sensing of the environment. They may be very useful for learning, but since they *can not* modify the environment, it is very important to consider *how they will be used.* Moreover, most execution-level features are not projective when making decisions at planning time. A sonar value in one location will not tell the planner whether a corridor is blocked elsewhere.

ROGUE's task planner uses execution-level features to decide when to execute an action. In other words, these features *become* projective in the short-term: the current value of the feature is correlated to a the value when the action is executed. When the planner is about to send the robot through a door, checking whether it is open can tell the planner whether to make the attempt.

Forgetting data. The massive amount of data that can be collected in a system that interacts with a real environment could lead to a lot of wasted effort when old, irrelevant data is processed. For this reason, it has been argued that the system will need to have some scheme for "forgetting" data.

However, our experiments show that the system should use as large a history as physically and computationally possible. Any data that is explicitly forgotten will never again influence learned rules, and hence a short history means that long-term patterns will never be detected. Unless the system maintains data over a span of several years, it will never detect annual patterns such as New Year's Day; instead such patterns will be treated as noise. Moreover, any situation that lasts longer than the history length will be considered permanent.

A longer history improves confidence in the validity of the environmental knowledge captured. Our situation-dependent learning approach learns rules that separate old data from recent data, thereby successfully identifying temporary phenomena, and it can do so in a fairly small number of execution traces.

6.2 Other Applications

Situation-dependent rules are useful in any domain where actions have specific *costs*, *probabilities*, or *achievability criteria* that depend on a complex definition of the state.

The approach is generally applicable in domains where:

- the environment changes according to some predictable pattern,
- action costs or probabilities change as function of world state,
- it is hard to pre-specify costs or probabilities, or patterns are likely to change over time, and
- a planner will benefit from increased knowledge of the environment.

Methods that learn an *average* cost or probability for an action will improve a system's behavior *on average*. If there are many patterns in the domain, however, there may be times when the system's *default* behaviour is actually *better* than the *learned* behaviour. Situation-dependent rules will change the cost or probability of an action according to the current environment. The system will not only be able to respond effectively to *changes* in the environment, but also behave in a manner that is directly *tailored* to their environment.

Some possible applications include:

Learning operator or action costs for planners that try to optimize total plan execution cost. (ROGUE learns action costs for the route planner.) A Martian path planner might decide on one route when there is a dust storm and different route otherwise. A network routing planner may select one route when congestion is high, and another otherwise.

Learning operator probabilities for probabilistic or conditional planners, such as for Weaver [Blythe, 1994] or U-PLAN [Mansell, 1993], or Xavier's navigation module. In Xavier's navigation module, the transitions between Markov states are currently assigned default probabilities; situation-dependent probabilities would probably improve performance of the system,. (ROGUE's control rule learning for the task planner can be viewed as a form of learning operator probabilities.)

Learning sensor probabilities or reliabilities, in any system (planner or otherwise) that relies on sensor information. For example, Xavier's navigation module uses a default value for $P(observation|state)$, where *state* is a very simple state description.

Learning sensor costs and utilities, in any system (planner or otherwise) that relies on sensor information. For example, under certain conditions some sensors may be easier or better to use than others. Medical domains are a good example of when the utility of different tests may change according to each patient's symptoms.

Learning case costs in case-based reasoning systems for which quality of the final solution depends on the current environment. In such systems, different cases may be more appropriate than others. For example, Haigh *et al.*'s route planner [1997a] selected cases depending on likely traffic congestion.

6.3 Future Research Directions

This thesis has opened up several areas for future research, both in the task planning framework and in the learning framework.

6.3.1 Improvements to the Task Planner

There is room for many extensions to the task planner. Mentioned in Chapter 2 were:

- adding the ability to rescind requests,
- improving the ability to reason about deadlines,
- adding the ability to identify when too many compatible tasks are being attempted,
- adding the ability to avoid thrashing issues as progressively more important tasks arrive, and
- exploring representations for continuous and parallel actions.

It will also be interesting to research the task planner's behaviour in a more complex domain. For example, as Xavier acquires more abilities, such as elevator riding or manipulation, the robot's autonomy will need to noticeably increase. As a result, it will be necessary to adapt the task planner to reason about multiple action-verification procedures and to rely somewhat less on human users. ROGUE is designed to make it easy to incorporate new abilities: the designer would need to specify (i) an abstract action description for the planner's use, (ii) a map from that action to Xavier-level commands, and (iii) when that action would be beneficial for plans.

Another future research direction is to examine the possibility of using PRODIGY4.0's environment simulation abilities in greater depth. Currently, ROGUE sends an action for execution when PRODIGY4.0 applies it in the current state. In more dangerous domains, operators will model dangerous side-effects, and this greedy execution heuristic will lead to task failure. Extending ROGUE to support parallel planning and execution will give the system the ability to simulate such dangerous effects.

This change to ROGUE's planning and execution approach will also require researching domain-independent heuristics for when to execute actions.

6.3.2 Improvements to the Learning Architecture

One area of possible research involves extending the cost evaluation function for events. In particular, the cost function for arc traversals in the path planner currently involves velocity, time and length. It would be interesting to extend this function to incorporate position confidence and other metrics, because they would aid in showing the applicability of the approach.

Another valuable research direction would be to explore methods to have Viterbi's algorithm correctly sum probabilities over fan-out edges. Our approximate algorithm gives good results, but an exact algorithm would likely do better. We see two possible approaches for making this change: (i) to reverse the Viterbi/ArcIdentification phases, and the (ii) to change the Markov model to indicate fan-out groupings.

In the first case, Viterbi's algorithm would then be applied directly to the topological map. Challenges in this approach would include developing techniques to evaluate traversal time correctly, and to appropriately amalgamate Markov state probabilities into arc probabilities.

In the second case, Viterbi's algorithm would have to be modified to sum over groups of incoming transitions, while selecting the maximum amongst different groups. The Ψ data structure and the forward calculations of δ would have to be modified to correctly capture the robot's motion. Another challenge in this approach would come from the same data representation differences described in Section 3.3.2. Appendix E describes AmalgamViterbi, an algorithm that meets exactly these requirements, developed after the bulk of the thesis work. However, AmalgamViterbi is also a heuristic method, for reasons explained in the Appendix.

Another area for research is to extend the types and number of events learned, for both planners. For example, the path planner would benefit from improved probability estimates on arc traversals, since it reasons about trade-offs between path reliability and path efficiency. Currently, ROGUE learns only arc traversal weights, affecting the planner's estimate of path efficiency.

It would also be interesting to see our learning approach implemented with another learning algorithm. Regression trees were well-adapted to our domain and our data; neural networks or Bayesian learning might be more suited to other other domains.

Learned environment costs would also be useful for customizing the environment. Humans already customize environments greatly for children and the handicapped. It seems only appropriate to also consider customizing the environment for our future co-workers: robots. Areas of the environment that the learning system identifies as being difficult or expensive to achieve tasks could be modified to improve system performance. An example modification would be to include location information in the packets sent from radio ethernet connections.

Another area of possible research is to have the system identify what areas of the environment need to be explored. Currently, ROGUE will only un-learn information when it is forced to re-execute an action it would otherwise avoid. For example, if ROGUE learns that a particular corridor is extremely expensive, ROGUE will only go into that corridor when a task demands that it must. It would also be useful for ROGUE to explore the environment where data is particularly sparse.

A last area of possible research, and perhaps with the greatest potential for improving the performance of learning systems, is to automatically decide what features to add to the data set. Klingspor *et al.* [1996] have already designed techniques for learning high-level feature concepts from low-level data. It remains an open research problem to automatically incorporate those features into learning.

Appendix A

Setting up PRODIGY4.0 with TCA

The two initialization programs in this Appendix set up the Lisp process. `init.lisp` is called to create a binary image of Lisp after loading PRODIGY4.0, TCA and the Wean Hall 5th floor map. `short-init.lisp` is called every time the binary image is loaded; it sets flags and loads the domain...code which changes frequently.

A.1 init.lisp

`init.lisp` shows what code is needed to

- to load PRODIGY4.0,
- to load TCA
- to load the Wean Hall 5th floor map,
- to load several "helper" functions,
- to create a binary image of the process, so that restarts are faster.

Ignore any error messages with a `:cont 0` command.

```
;;==============================================================================
;;                               init.lisp
;;==============================================================================
;; this file contains all the setup needed to hook up to
;;    load prodigy
;;    load TCA
;;    load the Xavier map
;;    create a binary with all this stuff
;; It runs on Allegro CL 4.2.beta2.0, which supports foreign functions
;; To load the Xavier domain (with code that changes a lot), load short-init.lisp

(load "/afs/cs/project/prodigy/version4.0/working/loader.lisp")
(load "/afs/cs/project/prodigy-1/khaigh/src/print-rules")
(load "/afs/cs/project/prodigy-1/khaigh/src/merge-static")
```

```
(load "/afs/cs/project/prodigy-1/khaigh/tca/tca.lisp")
(load "/afs/cs/project/prodigy-1/khaigh/tca/tca-commands.lisp")

(excl:chdir "/afs/cs/project/prodigy-1/khaigh/domains/xavier")

(setf *always-remove-p* t)
(setf *world-path* "/afs/cs/project/prodigy-1/khaigh/domains/")
(load "/afs/cs/project/prodigy-1/khaigh/domains/xavier/weh5th-obj")
(load "/afs/cs/project/prodigy-1/khaigh/domains/xavier/weh5th")

;;----------
;; Execute code
(defun use-execute ()
  (load "/afs/cs/project/prodigy-1/khaigh/src/execute.lisp"))
(use-execute)

;;----------
;; a helper function :)
(defun print-xavier (state)
  (if (eq 'state (car state))
      (print-xavier (cadr state))
      (dolist (s state)
        (if (and (not (eq s 'and))
                 (not (eq (car s) 'coords))
                 (not (eq (car s) 'robot-home-orientation))
                 (not (eq (car s) 'robot-position))
                 (not (eq (car s) 'robot-home-position))
                 (not (eq (car s) 'connected-room))
                 (not (eq (car s) 'connected))
                 (not (eq (car s) 'in-room))
                 (not (eq (car s) 'close-door))
                 (not (eq (car s) 'open-door))
                 (not (eq (car s) 'robot-radius))
                 (not (eq (car s) 'robot-orientation)))
            (format t "~%     ~S" s)))))

;;----------
;; These functions provide wrappers to C functions that
;; maintain a trace file.
;;    open a trace file
;;    call navigate to get the execution features,
;;    write the execution features,
;;    write arbitrary string
;;    close the trace file
;; make sure the directory is correct
```

```
(load "~/xavier/src/markov/obj/learningfeatures.o"
      :foreign-files '("/afs/cs/project/robocomp/tca/lib/libDevUtils.a"))
(ff:defforeign 'mc_print_execution_features_wrapper :return-type :void)
(ff:defforeign 'mc_get_execution_features_wrapper :return-type :fixnum)
(ff:defforeign 'mc_open_features_file :arguments '(string) :return-type :fixnum)
(ff:defforeign 'mc_close_features_file :return-type :void)
(ff:defforeign 'mc_write_string_to_features_file :arguments '(string)
               :return-type :fixnum)

;; These functions were written in C because writing lisp for some things
;; is just too agonizing.
(load "~/xavier/src/karen/obj/prodigyWrappers.o")
(ff:defforeign 'is_front_sonar_free
               :return-type :fixnum)
(ff:defforeign 'place_on_path
               :arguments '(string string)
               :return-type :fixnum)

;;----------
;; stores a Lisp binary with all the loaded stuff
(excl:dumplisp :name "~/Prodigy-tca-lisp")
;;================================================================================
```

A.2 short-init.lisp

`short-init.lisp` is the file that needs to be loaded every time Lisp is reloaded. It defines several flags and functions and loads the domain; this code changes much more frequently than the code defined in `init.lisp`. Run `~/Prodigy-tca-lisp`, then the first two commands are:

```
(load "short-init.lisp")
(connect_tca)
```

Calling `(connect_tca)` without the optional argument will cause it to connect to TCA and wait for a user request. Typing `(connect_tca nil)` will cause it to connect to TCA, but return to the command line without waiting for requests. Disconnect from TCA before re-calling `(connect_tca)`, even if the TCA server died.

```
;;================================================================================
;;                          short-init.lisp
;;================================================================================
;; This file relies on running ~/Prodigy-tca-lisp
;;
```

```
(defvar *do-execution* t) ;; set to nil if you want to simulate
(defvar *debug-Q* 0)
(defvar *debug-goto-location* 0)
(defvar *debug* 0)
(defvar *debug-new-requests* 1)
(defvar *debug-similar-goal* 0)
(defvar p4::*debug-goals-and-actions* 0)
(defvar *new-request* nil)
(defvar *time-of-last-execution* nil)
(defvar *max-wait-time* (* 600 internal-time-units-per-second))
(defvar *max-wait-for-user-time* (* 60 internal-time-units-per-second))
(defvar *waiting-for-stop-to-complete* nil)
(defvar *xavier-execution-queue* nil)
(defvar *xavier-executed* nil)

(use-execute)
(domain 'xavier)
(set-running-mode 'saba)

;; What to do when all tasks are done.
(setf XAVIER_FINISHED_MESSAGE
    `((tca::tcaExecuteCommand ,tca::SAY_COMMAND "I'm finished. Returning to Lab")
      (goto-location 1 r-5310 nil)))

(defun register_handlers ()
  (format t "Registering Handlers~%")
  (tca::tcaRegisterInformMessage "tapNavToGoal"  tca::NAVIGATE_TO_FORMAT)
  (tca::tcaRegisterHandler "tapNavToGoal" "tap_NavigateToG" 'tap_NavigateToG)
  (tca::tcaTapMessage tca::WhenAchieved tca::NAVIGATE_TO_GOAL "tapNavToGoal")
  (tca::tcaRegisterCommandMessage  "Prodigy_PlanRequestCommand"
                  "{string,int,string,int, string,string,string,string,string}")
  (tca::tcaRegisterHandler "Prodigy_PlanRequestCommand"
                          "PlanRequestHandler" 'PlanRequestHandler))

(defun connect_tca (&optional (startwait t))
  (reset-vars)
  (set-new-goal-point 'always)
  (define-prod-handler :always #'tcaProdigyCheckMessage)
  ;;open log file
  (mc_open_features_file "/usr0/khaigh/prodigy-output")

  (format t "~%Connect...~%")
  (tca::tcaConnectModule "Prodigy" (tca::tcaServerMachine))
  (register_handlers)
  (tca::tcaEnableDistributedResponses)
```

```
  (tca::tcaWaitUntilReady)
  (if startwait
      (tcaProdigyListen)))

(defun disconnect_tca ()
  (setf *waiting-for-stop-to-complete* nil)
  (setf *new-request* nil)
  (setf *time-of-last-execution* nil)
  (clear-prod-handlers)
  (tca::tcaClose))

(defun kill-handlers
  (mp:process-kill (mp:process-name-to-process "Listen to TCA")))

;;----------
;; this is my tcaModuleListen
;; it is the one used whenever prodigy is not running.
;;
;; if I call it in the first line of allegro, it will wait until a request
;; comes in, process that request, then invoke a (run). When the (run) is
;; complete, this function will resume, and wait until the next request comes
;; in. if requests come in *while* prodigy is running, then
;; tcaProdigyCheckMessage will deal with it.
;;
(defun tcaProdigyListen ()
  (loop
   (format t "~%-----~%New loop of tcaProdigyListen:~%")
   ;;check if there's a message, wait 5 seconds
   (tca::tcaHandleMessage 5)
   (when *new-request*
         (setf *new-request* nil)
         (format t "~%Handled an incoming request~%")
         ;; if prodigy is running, it should pick it up by itself
         (when
          (or (not (boundp 'p4::*prodigy-running*))
              (not p4::*prodigy-running*))
          (setf finished-executing nil)
          (format t "~%Spawning a Prodigy Run~%")
          (problem 'default)
          (setf p4::*execution-queue* nil)
          (run :output-level 3 :same-objects t :depth-bound 500)
          (setf p4::*prodigy-running* nil)
          (when *do-execution*
                (dolist (op XAVIER_FINISHED_MESSAGE)
                        (apply (car op) (cdr op))))))
```

```
   (when *xavier-execution-queue*
         (check-why-not-finished-executing))))

;;----------
;; this function is the one used whenever prodigy is already running.
;; prodigy will invoke it at every interrupt handler point
;; and whenever it's waiting for an action to finish
(defun tcaProdigyCheckMessage (signal)
  (let ((result))
    (when (> *debug* 0)
      (format t "~%Prodigy Checking Socket ~S" signal))

    (if (eq signal 'timeout)
        (setf result (tca::tcaHandleMessage 1))
      (setf result (tca::tcaHandleMessage 0)));;0 = just check
    (when (and *xavier-execution-queue* (not (eq signal 'no-timeout)))
      (check-why-not-finished-executing))
    result))

;; checks for timeouts
(defun check-why-not-finished-executing ()
  (when (and (not *waiting-for-stop-to-complete*) *time-of-last-execution*)
    (cond ((not p4::*robot-working-on-op*)
           (send-*execution-queue*-operator))
          ((> (- (get-internal-real-time) *time-of-last-execution*)
              *max-wait-time*)
           (format t "~%Option B")
           (when (> *debug-Q* 1)
                 (format t "  Q: ~S~%" *xavier-execution-queue*))
           (report-timeout))
          (t (format t "~%Waited:~S Total-to-wait:~S"
                     (- (get-internal-real-time) *time-of-last-execution*)
                     *max-wait-time*)))))

(defun reset-vars ()
  (setf *current-executing-action* nil)
  (setf *top-level-goals-priorities* nil)
  (setf p4::*execution-ops* nil)
  (setf *xavier-execution-queue* nil)
  (setf *xavier-executed* nil)
  (setf *requests* nil))

;;----------
;; some interesting functions to trace
(trace ancestor-is-top-priority-goal)
```

```
(trace similar-to-top-priority-goal)
(trace p4::completed-action)
(trace p4::remember-action-for-goal)
(trace p4::change-state-on-execute)
(untrace)

(problem 'mail-fax1)
(set-new-goal-point 'always)
(format t "~%Current Directory: ~S~%" (excl:current-directory))
;;==============================================================================
```

Appendix B

Sending Task Requests

This program was used to generate the training data for the experiments described in Chapter 4. It generates five requests between pairs of rooms. There are 21 users to select randomly from; none have multiple requests. There are 8 possible tasks, and 22 rooms from the `weh5th.param` environment; these may be repeated.

```
/* ============================== DATA ==================================== */
#include <stdio.h>
#include <tca/tcaDev.h>
#include <time.h>

#define NUM_REQUESTS 5
int used_users[NUM_REQUESTS];

/*    USER PRIORITY (small is good)
# 6 = undergrad student
# 5 = grad student
# 4 = research staff
# 3 = faculty
# 2 = secretary
# 1 = dept head
*/
#define NUM_USERS 21
struct {
  char *name;
  int rank;
} users[NUM_USERS] = {{"illah", 3}, {"jan", 2}, {"jblythe", 5},{"will", 5},
                      {"jean", 2}, {"jgc", 3}, {"jhm", 1}, {"joan", 2},
                      {"johnson", 3}, {"josullvn", 5}, {"jrs", 4},
                      {"khaigh", 5}, {"mcox", 4}, {"mitchell", 3},
                      {"mmv", 3}, {"reddy", 1}, {"reids", 3}, {"thrun", 3},
                      {"robd", 5}, {"satya", 3}, {"skoenig", 5}};
```

```
#define NUM_TASKS 8
struct {
  char *name;
  int rank;
} tasks[NUM_TASKS] = {{"deliverfax", 2},   {"delivermail", 3},
                      {"pickupcoffee", 8}, {"pickupfax", 5},
                      {"pickupfedex", 4},  {"deliverfedex", 1},
                      {"pickupmail", 6},  {"pickupprintout", 7}};

#define NUM_ROOMS 22
char *rooms[NUM_ROOMS] = {"r-5301","r-5303","r-5307","r-5309","r-5311",
                          "r-5313","r-5315","r-5317","r-5321","r-5302",
                          "r-5304","r-5310","r-5312","r-5316","r-5320",
                          "r-5328","r-5336","r-5427","r-5419","r-5415",
                          "r-5409","r-5403"};

char *WHY = "nothing";
char request_date[20];
char deadline_date[25];

struct  {
  char *userid;
  int rank;
  char *task;
  int taskrank;
  char *why;
  char *whenrequest;
  char *whendeadline;
  char *wherepickup;
  char *wheredeliver;
} prodigy_struct_ptrs;
/* ============================== CODE ==================================== */
void SendRequestCmd()
{
  int i,id,j;
  struct timeval tv;
  struct timezone tz;
  struct tm* tm1;
  time_t tt;
  (void) gettimeofday( &tv, &tz );
  (void) srand( tv.tv_usec );
  tt = time( NULL );
  tm1 = localtime(&tt);
```

```
  for (i=0 ; i< NUM_REQUESTS; i++) {
    j=0;
    do { /* make sure no repeated users */
      id = rand() % NUM_USERS;
      for (j=0; j<i; j++)
        if (id == used_users[j]) break;
    } while (j<i);
    used_users[i] = id;
    prodigy_struct_ptrs.userid =  (users[id].name);
    prodigy_struct_ptrs.rank = (users[id].rank);
    id = rand() % NUM_TASKS;
    prodigy_struct_ptrs.task = (tasks[id].name);
    prodigy_struct_ptrs.taskrank = (tasks[id].rank);
    prodigy_struct_ptrs.why =  (WHY);
    strftime(request_date,20,"\"%B %d %H:%M\"", tm1);
    prodigy_struct_ptrs.whenrequest = (request_date);
    tm1->tm_hour += 1;
    strftime(deadline_date,25,"\"%a %b %d %H:%M\"", tm1);
    prodigy_struct_ptrs.whendeadline = (deadline_date);
    id = rand() % NUM_ROOMS;
    prodigy_struct_ptrs.wherepickup = (rooms[id]);
    id = rand() % NUM_ROOMS;
    prodigy_struct_ptrs.wheredeliver = (rooms[id]);
    printf("Requesting for user %s (%d) to go to rooms %s / %s for %s purpose (%d)\n",
           prodigy_struct_ptrs.userid,      prodigy_struct_ptrs.rank,
           prodigy_struct_ptrs.wherepickup,  prodigy_struct_ptrs.wheredeliver,
           prodigy_struct_ptrs.task,        prodigy_struct_ptrs.taskrank);
    tcaExecuteCommand( "Prodigy_PlanRequestCommand", &prodigy_struct_ptrs);
    printf("Done Requesting\n");
  }
}
static void registerAll(void)
{
}
int main(int argc, char *argv[])
{
  static const char *reqRequires[] = {"Prodigy", NULL};
  TCA_connect("Test Move", registerAll, NULL, NULL, NULL, reqRequires);
  SendRequestCmd();
  return 1;
}
/* ======================================================================= */
```

Appendix C

Changes to the Path Planner

This appendix outlines the changes made to the path planner to support learned traversal weights and probabilities. In the original code, traversal weight was calculated as a function of the arc: all corridor-corridor arcs had a certain default cost, all corridor-room arcs had a certain default cost, and so on.

1. Added `traversalWeight` to `QMAP_ARC_TYPE` data structure. (The default values were constant.)

2. Modified the `traversalWeight()` function in `qmap.c` to return the updated cost if it exists; if not, it returns the default value.

3. Added a TCA command `UPDATE_QMAP_ARC_PROBABILITY_COMMAND` which updates arc probabilities.

```
UPDATE_QMAP_ARC_PROBABILITY_TYPE arc Data;
arc Data.arcid = arc;
arcData.prob_low = low;
arcData.prob_high = high;
tcaExecuteCommand(UPDATE_QMAP_ARC_PROBABILITY_COMMAND, &arcData);
```

4. Added a TCA command `UPDATE_QMAP_ARC_WEIGHT_COMMAND` which updates arc traversal weights.

```
UPDATE_QMAP_ARC_WEIGHT_TYPE arcData;
arcData.arcid = arc;
arcData.traversalWeight = node.yval;
tcaExecuteCommand(UPDATE_QMAP_ARC_WEIGHT_COMMAND, &arcData);
```

Appendix D

Collecting Execution Features

This appendix contains the current implementation of the TCA query to get current execution features. The data structures are defined in `markov/learningfeatures.h`, and the current definition is shown in Section D.1. The file `markov/learningfeatures.c` contains several library functions that can be used to manipulate the data. Section D.2 shows a sample piece of code that asks for the execution features, and then uses the library function `mc_print_execution_features` to store the data to the trace file. Section D.3 shows the implementation of the `MC_REPORT_EXECUTION_FEATURES` query. The query is handled in the Xavier file `markov/mcpos.c`.

D.1 Data Structure for Execution Features

Below is the definition for the current set of available execution features. The definition appears in `markov/learningfeatures.h`.

```
#define SEND_MARKOV_NODES 50
typedef struct {
  MarkovNodeIndexType mn;
  ProbabilityType prob;
} SIMPLIFIED_MARKOV_STATES_TYPE;

typedef struct {
  MOTION_TYPE motion_data;
  int time_data_hour;
  int time_data_min;
  int time_data_sec;
  int num_markov_states;
  SIMPLIFIED_MARKOV_STATES_TYPE markov_states[SEND_MARKOV_NODES];
  CMS x, y, orientation;
  CMS sonarData[24];
} NAV_FEATURES_TYPE;
```

D.2 Querying for Execution Features

This sample piece of code queries the navigation module for the execution features, and then uses the library function `mc_print_execution_features` to store the data to the trace file.

```
FILE *trace_fp;
NAV_FEATURES_TYPE LispFeatures;

if (tcaQuery(MC_REPORT_EXECUTION_FEATURES, NULL, &LispFeatures) == NullReply) {
  return 0;
} else {
  mc_print_execution_features( trace_fp, &LispFeatures );
  return 1;
}
```

D.3 Execution Feature Query Handler

Below is the current implementation of the query handler which returns the current set of execution features.

```
/* ========================================================================== */
void mc_get_execution_features_query_handler( TCA_REF_PTR ref, void *data )
{
  FILE *fp;
  NAV_FEATURES_TYPE features;
  PROBABILITY_ARRAY_PTR currentProbs;
  int i;
  SONAR_STATUS_TYPE sonarStatus;

  time_t tp1;
  struct tm *tm1;
  tp1 = time(&tp1);
  tm1 = localtime(&tp1);

  bzero(&features,sizeof(features));

  tcaQuery(CTR_MOTION_REPORT_QUERY, NULL, &(  features.motion_data ));

  features.time_data_hour = tm1->tm_hour;
  features.time_data_min = tm1->tm_min;
  features.time_data_sec = tm1->tm_hour;

  features.num_markov_states=0;
```

```
  currentProbs = GetProbabilities();
  DO_PROBS(currentProbs, markov_index, prob,
           {
           if (features.num_markov_states >= SEND_MARKOV_NODES) {
             printf("Warning: there are more than %d relevant markov nodes.\n",
                    SEND_MARKOV_NODES);
             printf("         Change SEND_MARKOV_NODES in mcpos.h\n");
             continue;
           }
           if (prob > 0.0005) {
             features.markov_states[features.num_markov_states].mn = markov_index;
             features.markov_states[features.num_markov_states].prob = prob;
             (features.num_markov_states)++;
           }
         });

  tcaQuery( CTR_SONAR_STATUS_QUERY, NULL, &sonarStatus);
  features.x = sonarStatus.x;
  features.y = sonarStatus.y;
  features.orientation = sonarStatus.orientation;
  for (i=0; i<24; i++) {
    features.sonarData[i] = sonarStatus.data[i];
  }

  tcaReply(ref, &features);
  tcaFreeData(tcaReferenceName(ref), (void *)data);
}
/* ========================================================================== */
```

Appendix E

AmalgamViterbi

This appendix represents recent work done to create an exact version of Viterbi's algorithm for Markov models with a high, local branching factor that effectively capture additional uncertainty variables such as time or length. In Section 3.3.1, we presented Multi/Markov Viterbi, a heuristic method that works reasonably well in our environment.

Ideally, however, we would like an exact algorithm that identifies when multiple trajectories should be abstracted into a single one.

This appendix presents AmalgamViterbi, an algorithm that does this calculation for our Markov models. AmalgamViterbi calculates an extremely good estimate of the robots trajectory, but, as we discuss in Section E.4, AmalgamViterbi is also a heuristic method.

We present a comparison between Viterbi's algorithm, Multi/Markov Viterbi and AmalgamViterbi in three different environments.

E.1 Markov Models with High Branching Factors

Viterbi's algorithm is guaranteed to find the most likely sequence of Markov states [Rabiner & Juang, 1986]. Unfortunately, Viterbi's algorithm was not designed for use in Markov models with additional uncertainty factors that change the structure of the model.

In the context of navigation, standard Markov models represent only position uncertainty. Our models represent length uncertainty too; in other words, our models are a probability distribution over position, p, and length, l, while standard models only represent a probability distribution over p.

Representing length uncertainty is conceptually the same as if speech models incorporated the length of time to say a word; e.g. a slow Southern Drawl vs. rapid New York speech. Speech researchers do not model time because they claim it would be too computationally complex.

Recall that we use *parallel Markov chains* to represent length uncertainty (Section 3.1.2.) The length uncertainty in our Markov models causes Viterbi's algorithm to lose information, become confused, and generate a poor estimate of the best path. Viterbi's algorithm picks the *single* most likely incoming transition; when a node "fans-out" into a set of parallel Markov chains, Viterbi's algorithm is unable to identify them as being related.

As a result of this problem, we have developed a modification to Viterbi's algorithm that compensates for the "fan-out" problem introduced by modelling this additional uncertainty variable, in our case, length. Our modification amalgamates the probabilities from within each group of parallel chains, hence correctly identifying the robot's most likely trajectory.

E.2 AmalgamViterbi

In order to make a good estimate of the robot's trajectory, we need to identify when multiple possible trajectories should be abstracted into a single one. For our Markov model, we do this by amalgamating the probabilities from all the incoming transitions from a single group of parallel Markov chains. Our Markov models are a probability distribution over position, p, and length, l. AmalgamViterbi essentially integrates over l, giving a probability distribution over p.

Table E.1 shows the complete algorithm. For a given Markov node s, AmalgamViterbi separates all the incoming transitions into groups, g_s, where a *group*, $g \in \mathcal{G}$, corresponds to one parallel chain. In other words, in a long corridor, each node might have only one incoming group from each of the two directions. At intersections, there will be a group corresponding to each incoming direction. Multiple nodes at a corridor intersection will also be considered a single group. For example, consider Figure E.1. Nodes j, k, and l are one group, as are m, n, o, and p.

Then, AmalgamViterbi finds the probability of group g incoming to s, $P(g_s) = P(\mathrm{X}_y)$, by summing all the transitions $A_a(s', s)$ in that group (line 1). Then, AmalgamViterbi sets the δ probability of s as the probability of the maximum incoming group (line 2). In Figure E.1, AmalgamViterbi sets $\delta(o,t) = \mathrm{MAX}\,[P(\mathrm{B}_o), P(\mathrm{M}_o)]$; $P(\mathrm{M}_o)$ is the sum of the transitions from m and p, $P(\mathrm{B}_o)$ is the sum of the transitions from d and h.

For each group, g, AmalgamViterbi then calculates its set of the incoming groups, G (line 3). For example, at a four-node corridor intersection, each node may have two incoming groups (one from a parallel Markov chain, and one from other nodes in the intersection). All the states in a group, however, may have jointly many more incoming groups. Node b has only one incoming group: group A. Similarly, node d also has only one incoming group: group B. Therefore group B as a whole has two incoming groups: groups A and B. Similarly, group M has three incoming groups: groups B, J and M.

AmalgamViterbi then calculates the total probability of each of these incoming groups, $g' \in G$ (line 4). For group M, then, AmalgamViterbi calculates the incoming probabilities of groups B, J and M, $P(\mathrm{B_M})$, $P(\mathrm{J_M})$, and $P(\mathrm{M_M})$.

AmalgamViterbi finally sets the the Δ probability of each group g to be the maximum of all the incoming groups (line 5). For group M, $\Delta(\mathrm{M}, t) = \mathrm{MAX}\,[P(\mathrm{B_M}), P(\mathrm{J_M}), P(\mathrm{M_M})]$. It then sets Ψ of the group accordingly (line 6).

To generate the estimate of the robot's trajectory, we recurse through the Ψ data struc-

Define $\mathcal{G}$ to be the set of all groups, $s \in g \in \mathcal{G}$.
Define δ to be the AmalgamViterbi probability distribution over S; $\delta(s,t)$ is the probability of the best sequence ending at s at time t.
Define Δ to be the AmalgamViterbi probability distribution over $\mathcal{G}$; $\Delta(g,t)$ is the probability of the best sequence ending at g at time t.
Define $\Psi(g,t)$ to be the group $g' \in \mathcal{G}$ at time $t-1$ that most likely leads to g.
Define $Seq_{_T}$ to be the most likely sequence generated from time T; $g = Seq_{_T}(t)$ is the group at time t in $Seq_{_T}$.

1. At time $t = 0$:
   $\forall s \in S$, let $\delta(s,0) =$ initial state distribution $= \pi(s,0)$
   $\forall g \in \mathcal{G}$, let $\Psi(g,0) =$ NULL

2. For time $t+1 \geq 1$, action a was selected, and observation o_{t+1} was made:
   Let $\mathcal{G}(s)$ be the set of groups *incoming* to node s; $g_s \in \mathcal{G}(s) \in \mathcal{G}$.
   $\forall s \in S, \forall g_s \in \mathcal{G}(s),\ P(g_s) = \sum_{s' \in g_s} [\delta(s',t) \times A_a(s',s) \times \mathcal{O}(s, o_{t+1})]$ 1
   $\delta(s,t+1) = \mathrm{MAX}_{g_s \in \mathcal{G}(s)} P(g_s)$ 2
   $\forall g \in \mathcal{G}$, Let G be all the groups incoming to g, i.e. $G = \bigcup_{s \in g} g_s$. 3
   $\forall g' \in G,\ P(g'_g) = \sum_{s' \in g'} P(g_{s'})$ 4
   $\Delta(g,t+1) = \mathrm{MAX}_{g' \in G} P(g'_g)$ 5
   $\Psi(g,t+1) = \mathrm{ARGMAX}_{g' \in G} P(g'_g)$ 6

3. To calculate the most likely sequence at time T, $Seq_{_T}$:
   $Seq_{_T}(T) = \mathrm{ARGMAX}_{\forall g \in \mathcal{G}} [\Delta(g,T)]$ 7
   $\forall t, 0 \leq t < T,\ Seq_{_T}(t) = \Psi(Seq_{_T}(t+1), t+1)$. 8

Table E.1: AmalgamViterbi.

ture, in the same manner as for the standard Viterbi algorithm. The sequence $Seq_{_T}$ however contains only group information; we no longer consider the concept of "what *node* robot was in at what time," we instead consider "what *group* the robot was in at what time."

AmalgamViterbi does dynamic programming in the same basic manner as Viterbi's algorithm, but instead of finding the maximum over *singleton* incoming transitions, it finds the maximum incoming *group*.

E.3 A Comparison of Viterbi Algorithms

To demonstrate the effectiveness of AmalgamViterbi, we collected data in several environments, both in the simulator and from a real robot. We compare the performance of Viterbi's algorithm with AmalgamViterbi through illustrative examples.

The first example is generated from a trace on Xavier. Figure E.2 shows a comparison

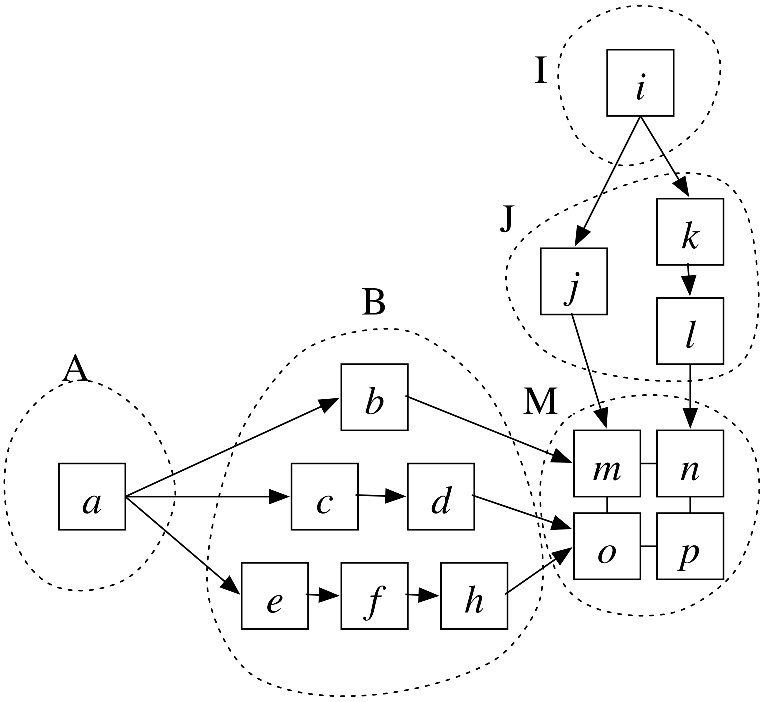


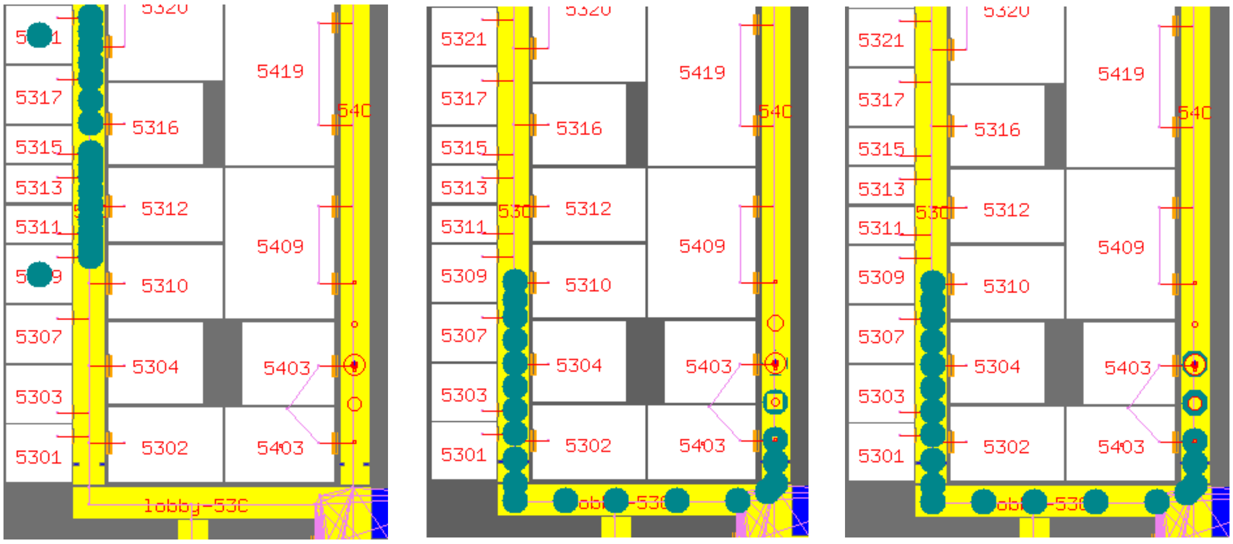
Figure E.1: Groups, $\mathcal{G}$ of Markov states.

between the most likely route selected by each of the algorithms, after the robot has travelled from outside room 5310 to outside room 5403. Figure E.2a shows the most likely route selected by the unmodified Viterbi's algorithm, while Figure E.2b shows the most likely route selected by AmalgamViterbi. Viterbi's algorithm provides a good estimate of the robots' trajectory the robot makes a turn; by the time the robot has made two turns and reached its goal, Viterbi's algorithm has been completely mislead.

The second example was generated in a maze world: a world we use to test the behaviour of the POMDPs. Execution traces were collected in the simulator; with 20% length uncertainty, Viterbi's algorithm is unable to track the robot through its zigzag route from the upper left to the lower right (Figure E.3).

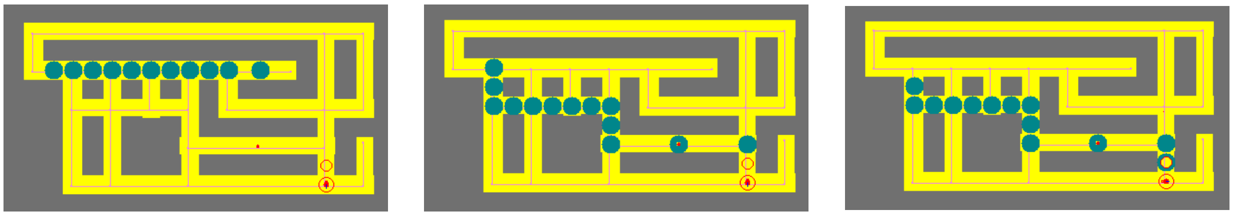
The third example was generated in the exposition world of Section 3.7.1. Execution traces were collected in the simulator which closely approximates the behaviour of the real robot: it creates noisy sonar readings, it has poor dead-reckoning abilities, and it gets stuck going through doors. With 20% length uncertainty, Viterbi's algorithm is unable to track the robot through its route from booth 231 to booth 311 (Figure E.4).

We could not find a convincing example when AmalgamViterbi provides better results than Multi/Markov Viterbi, hence showing that Multi/Markov Viterbi was adequate for our learning task *in our environment*, where uncertainty is constant in all corridors. However, we believe that AmalgamViterbi will be more accurate when it must disambiguate between many possible trajectories, in an environment where uncertainty is more varied.



(a) Standard (b) Multi/Markov (c) Amalgam

Figure E.2: Processing data from a trace collected on the real robot, after 175 actions; from room 5310 to 5403. (Large filled circles indicate most likely sequence.) (a) Standard Viterbi's algorithm: most likely route is from inside room 5309 to inside room 5321. (b) Multi/Markov Viterbi: most likely route closely approximates true route. (c) AmalgamViterbi: most likely route closely approximates true route.

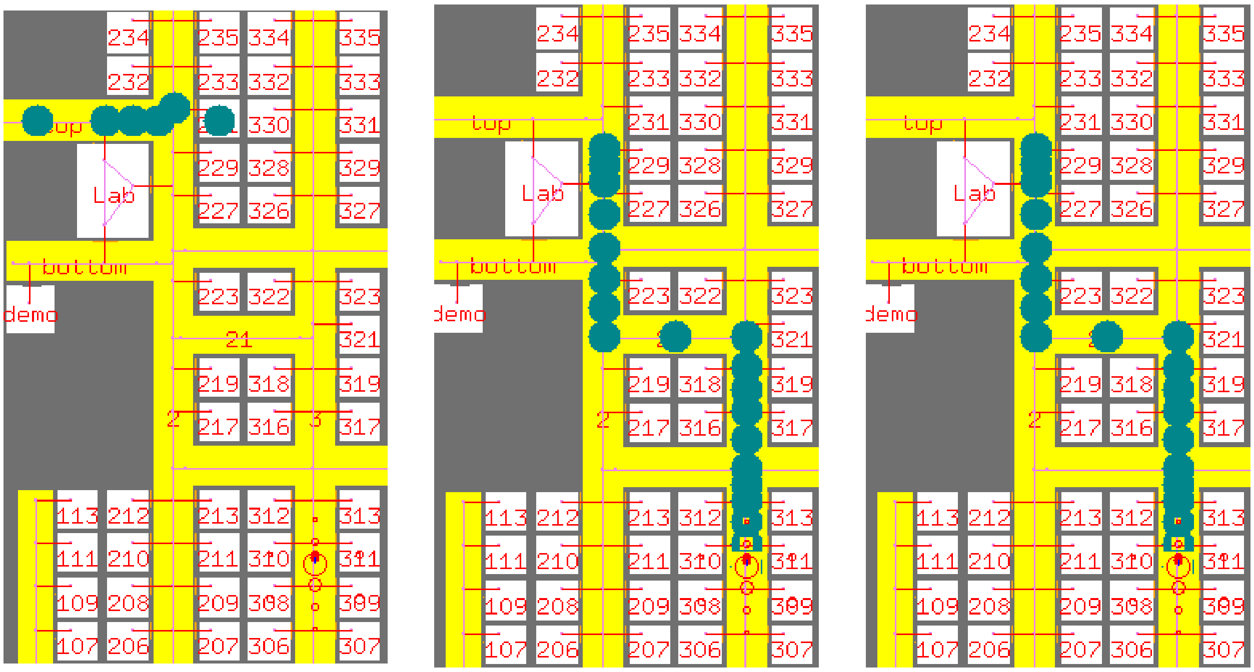


(a) Standard (b) Multi/Markov (c) Amalgam

Figure E.3: Viterbi's algorithm in the Maze World, after 250 actions. (a) Standard Viterbi's algorithm: most likely route does not detect any turns. (b) Multi/Markov Viterbi. (c) AmalgamViterbi.

E.4 AmalgamViterbi as a Heuristic

It is clear that AmalgamViterbi makes a better estimate of the robot's most likely trajectory than the standard Viterbi's algorithm. However, it is not guaranteed to be correct. For example, consider Figure E.5. Assume that the distance from A to B is two metres, while B to C may be two or four metres long. The initial probability distribution is $\delta(A,0) = \delta(B,0) = 0.5$. At time $t = 4$, AmalgamViterbi calculates $\delta(C,4)$ as the sum of the two paths: A-v-B-w-C and B-x-y-z-C, even though they should be considered distinct paths. The Ψ connections will arbitrarily select one of the two paths as the route estimate, but



(a) Standard (b) Multi/Markov (b) Amalgam

Figure E.4: Viterbi's algorithm in the Exposition World, after 125 actions; from booth 231 to booth 311. (a) Standard Viterbi's algorithm: most likely route goes along the "top" corridor, back, and into booth 231. (b) Multi/Markov Viterbi. (c) AmalgamViterbi.

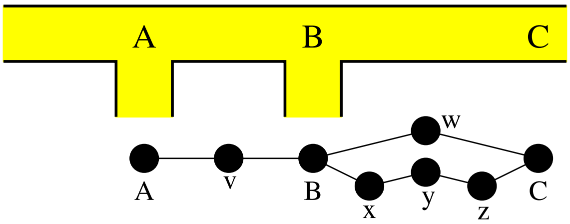


Figure E.5: An example of when AmalgamViterbi incorrectly estimates the most likely path. Below the corridor are the Markov states and transitions, indicating that the corridor segment from B to C may be two or four metres long.

its δ probability will be double the actual value. There is no way to correctly calculate the probability while still maintaining the dynamic programming property of the algorithm.

E.5 Summary

For the purposes of learning, Viterbi's algorithm does not provide an accurate enough estimate of the robots trajectory. Multi/Markov Viterbi is an improvement on the standard algorithm and works well in our environment where uncertainty is uniformly distributed.

AmalgamViterbi is another, more accurate, method of estimating the robot's trajectory through the environment.

Bibliography

[Ambros-Ingerson & Steel, 1988] José A. Ambros-Ingerson and Sam Steel (1988). Integrating planning, execution and monitoring. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, pages 83–88, St. Paul, MN. (Menlo Park, CA: AAAI Press).

[Atkins *et al.*, 1996] Ella M. Atkins, Edmund H. Durfee, and Kang G. Shin (1996). Detecting and reacting to unplanned-for world states. In *Papers from the 1996 AAAI Fall Symposium "Plan Execution: Problems and Issues"*, pages 1–7, Boston, MA. (Menlo Park, CA: AAAI Press).

[Baird, 1995] Leemon C. Baird (1995). Residual algorithms: Reinforcement learning with function approximation. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference (ICML95)*, pages 30–37, Tahoe City, CA. (San Mateo, CA: Morgan Kaufmann).

[Baroglio *et al.*, 1996] C. Baroglio, A. Giordana, M. Kaiser, M. Nuttin, and R. Piola (1996). Learning controllers for industrial robots. *Machine Learning*, 23:221–249.

[Becker *et al.*, 1988] Richard A. Becker, John M. Chambers, and Allan R. Wilks (1988). *The New S Language*. (Pacific Grove, CA: Wadsworth & Brooks/Cole). Code available from `http://www.mathsoft.com/splus/`.

[Bennett & DeJong, 1996] Scott W. Bennett and Gerald F. DeJong (1996). Real-world robotics: Learning to plan for robust execution. *Machine Learning*, 23:121–161.

[Blythe, 1994] Jim Blythe (1994). Planning with external events. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 94–101, Seattle, WA. (San Mateo, CA: Morgan Kaufmann).

[Bonasso & Kortenkamp, 1996] R. Peter Bonasso and David Kortenkamp (1996). Using a layered control architecture to alleviate planning with incomplete information. In *Proceedings of the AAAI Spring Symposium "Planning with Incomplete Information for Robot Problems"*, pages 1–4, Stanford, CA. (Menlo Park, CA: AAAI Press).

[Borrajo & Veloso, 1994] Daniel Borrajo and Manuela Veloso (1994). Incremental learning of control knowledge for improvement of planning efficiency and plan quality. In *Working* *notes from the AAAI Fall Symposium "Planning and Learning: On to Real Applications"*, pages 5–9, New Orleans, LA.

[Boyan & Moore, 1995] Justin A. Boyan and Andrew W. Moore (1995). Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 369–76, Cambridge, MA. The MIT Press.

[Breiman *et al.*, 1984] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone (1984). *Classification and Regression Trees.* (Pacific Grove, CA: Wadsworth & Brooks/Cole).

[Carbonell *et al.*, 1992] Jaime G. Carbonell, and the PRODIGY Research Group (1992). PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University.

[Carbonell *et al.*, 1990] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton (1990). PRODIGY: An integrated architecture for planning and learning. In K. VanLehn, editor, *Architectures for Intelligence.* Erlbaum, Hillsdale, NJ. Also available as Technical Report CMU-CS-89-189, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

[Cassandra *et al.*, 1994] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1023–1028, Seattle, WA. (Menlo Park, CA: AAAI Press).

[Chambers & Hastie, 1992] John M. Chambers and Trevor Hastie (1992). *Statistical models in S.* (Pacific Grove, CA: Wadsworth & Brooks/Cole).

[Dean *et al.*, 1990] Thomas Dean, Kenneth Basye, Robert Chekaluk, Seungseok Hyun, Moises Lejter, and Margaret Randazza (1990). Coping with uncertainty in a control system for navigation and exploration. In *Proceedings of the Eigth National Conference on Artificial Intelligence (AAAI-90)*, pages 1010–1015, Boston, MA. (Cambridge, MA: MIT Press).

[Dean & Boddy, 1988] Thomas L. Dean and Mark Boddy (1988). An analysis of time-dependent planning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-88)*, pages 49–54, St. Paul, MN. (Menlo Park, CA: AAAI Press).

[DellaFera *et al.*, 1988] C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Sommerfeld (1988). The Zephyr notification service. In *Proceedings of the USENIX Winter Conference*, pages 213–219, Dallas, TX. (Berkeley, CA: USENIX Association).

[Drummond *et al.*, 1993] Mark Drummond, Keith Swanson, John Bresina, and Richard Levinson (1993). Reaction-first search. In *Proceedings of the Thirteenth International Joint*

Conference on Artificial Intelligence (IJCAI-93), pages 1408–1414, Chambery, France. (San Mateo, CA: Morgan Kaufmann).

[Fikes *et al.*, 1972] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):231–249.

[Fink & Veloso, 1994] Eugene Fink and Manuela Veloso (1994). PRODIGY planning algorithm. Technical Report CMU-CS-94-123, Deparment of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Firby, 1989] Robert James Firby (1989). *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Yale University, New Haven, CT.

[Firby, 1994] R. James Firby (1994). Task networks for controlling continuous processes. In K. Hammond, editor, *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS-94)*, pages 49–54, Chicago, IL. (Menlo Park, CA: AAAI Press).

[Gat, 1992] Erann Gat (1992). Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 809–815, San Jose, CA.

[Georgeff & Ingrand, 1989] Michael P. Georgeff and François F. Ingrand (1989). Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 972–978, Detroit, MI. (San Mateo, CA: Morgan Kaufmann).

[Gervasio & DeJong, 1991] Melinda T. Gervasio and Gerald F. DeJong (1991). Learning probably completable plans. Technical Report UIUCDCS-R-91-1686, University of Illinois at Urbana-Champaign, IL, Urbana, IL.

[Gil, 1992] Yolanda Gil (1992). *Acquiring domain knowledge for planning by experimentation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Also available as Technical Report CMU-CS-92-175.

[Goodwin, 1994] Richard Goodwin (1994). Reasoning about when to start acting. In K. Hammond, editor, *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS-94)*, pages 86–91, Chicago, IL. (Menlo Park, CA: AAAI Press).

[Goodwin, 1996] Richard Goodwin (1996). *Meta-Level Control for Decision-Theoretic Planners.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-96-186.

[Goodwin & Simmons, 1992] Richard Goodwin and Reid G. Simmons (1992). Rational handling of multiple goals for mobile robots. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS-92)*, pages 86–91, College Park, MD. (San Mateo, CA: Morgan Kaufmann).

[Grant & Feng, 1989] E. Grant and Cao Feng (1989). Experiments in robot learning. In *Proceedings of IEEE International Symposium on Intelligent Control 1989*, pages 561–5, Albany, NY.

[Haigh *et al.*, 1997a] Karen Zita Haigh, Jonathan Richard Shewchuk, and Manuela M. Veloso (1997a). Exploiting domain geometry in analogical route planning. *Journal of Experimental and Theoretical Artificial Intelligence*, 9:509–541.

[Haigh *et al.*, 1997b] Karen Zita Haigh, Peter Stone, and Manuela M. Veloso (1997b). Execution in PRODIGY4.0: The user's manual. Technical Report CMU-CS-97-187, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

[Haigh & Veloso, 1996] Karen Zita Haigh and Manuela Veloso (1996). Interleaving planning and robot execution for asynchronous user requests. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 148–155, Osaka, Japan. (New York, NY: IEEE Press).

[Haigh & Veloso, 1997] Karen Zita Haigh and Manuela M. Veloso (1997). Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*. In press.

[Haigh & Veloso, 1998a] Karen Zita Haigh and Manuela M. Veloso (1998a). Learning situation-dependent costs: Improving planning from probabilistic robot execution. In Katia P. Sycara, editor, *Proceedings of the Second International Conference on Autonomous Agents*, Minneapolis, MN. (Menlo Park, CA: AAAI Press). In Press.

[Haigh & Veloso, 1998b] Karen Zita Haigh and Manuela M. Veloso (1998b). Planning, execution and learning in a robotic agent. In R. Simmons, M. Veloso, and S. Smith, editors, *Artificial Intelligence Planning Systems: Proceedings of the Fourth International Conference (AIPS-98)*, Pittsburgh, PA. (Menlo Park, CA: AAAI Press). Submission.

[Hammond, 1987] Kristian J. Hammond (1987). Learning and reusing explanations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 141–147, Irvine, CA.

[Hormann *et al.*, 1991] Andreas Hormann, Wolfgang Meier, and Jan Schloen (1991). A control architecture for and advanced fault-tolerant robot system. *Robotics and Autonomous Systems*, 7(2-3):211–225.

[Hughes & Ranganathan, 1994] Ken Hughes and N. Ranganathan (1994). Modeling sensor confidence for sensor integration tasks. *International Journal of Pattern Recognition and Artificial Intelligence*, 8(6):1301–1318.

[Kaelbling *et al.*, 1996] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

[Kibler, 1993] Dennis Kibler (1993). Some real-world domains for learning problem solvers. In *Proceedings of KCSL93, 3rd International Workshop on Knowledge Compilation and Speedup Learning (in ICML93)*, Amherst, MA.

[Klingspor *et al.*, 1996] Volker Klingspor, Katharina J. Morik, and Anke D. Rieger (1996). Learning concepts from sensor data of a mobile robot. *Machine Learning*, 23:305–332.

[Koenig, 1997] Sven Koenig (1997). *Goal-Directed Acting with Incomplete Information*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-97-199.

[Koenig & Simmons, 1996] Sven Koenig and Reid G. Simmons (1996). Passive distance learning for robot navigation. In Lorenza Saitta, editor, *Machine Learning: Proceedings of the Thirteenth International Conference (ICML96)*, pages 266–274, Bari, Italy. (San Mateo, CA: Morgan Kaufmann).

[Kortenkamp & Weymouth, 1994] David Kortenkamp and Terry Weymouth (1994). Topological mapping for mobile robots using a combination of sonar and vision sensing. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 979–984, Seattle, WA. (Menlo Park, CA: AAAI Press).

[Krotkov *et al.*, 1995] Eric Krotkov, Martial Hebert, and Reid Simmons (1995). Stereo perception and dead reckoning for a prototype lunar rover. *Autonomous Robots*, 2(4):313–331.

[Kushmerick *et al.*, 1993] Nick Kushmerick, Steve Hanks, and Dan Weld (1993). An algorithm for probabilistic planning. Technical Report 93-06-03, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

[Lindner *et al.*, 1994] John Lindner, Robin R. Murphy, and Elizabeth Nitz (1994). Learning the expected utility of sensors and algorithms. In *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, pages 583–590. (New York, NY: IEEE Press).

[Lovejoy, 1991] W. Lovejoy (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28(1):47–65.

[Lyons & Hendriks, 1992] D. M. Lyons and A. J. Hendriks (1992). A practical approach to integrating reaction and deliberation. In J. Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS-92)*, pages 153–162. (San Mateo, CA: Morgan Kaufmann).

[Mansell, 1993] Todd Michael Mansell (1993). A method for planning given uncertain and incomplete information. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pages 250–358, Washington, DC. (San Mateo, CA: Morgan Kaufmann).

[McCallum, 1995] Andrew Kachites McCallum (1995). *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY.

[McDermott, 1992] Drew McDermott (1992). Transformational planning of reactive behavior. Technical Report YALE/CSD/RR#941, Computer Science Department, Yale University, New Haven, CT.

[Mitchell, 1997] Tom M. Mitchell (1997). *Machine Learning.* (New York, NY: McGraw Hill).

[Mitchell *et al.*, 1994] Tom M. Mitchell, Rich Caruana, Dayne Freitag, John P. McDermott, and David Zabowski (1994). Experience with a learning personal assistant. *CACM*, 37(7):80–91.

[Nilsson, 1984] Nils J. Nilsson (1984). Shakey the robot. Technical Report 323, AI Center, SRI International, Menlo Park, CA.

[Nourbakhsh, 1997] Illah Nourbakhsh (1997). *Interleaving Planning and Execution for Autonomous Robots.* (Dordrecht, Netherlands: Kluwer Academic). PhD thesis. Also available as technical report STAN-CS-TR-97-1593, Department of Computer Science, Stanford University, Stanford, CA.

[O'Sullivan *et al.*, 1997] Joseph O'Sullivan, Karen Zita Haigh, and G. D. Armstrong (1997). *Xavier.* Carnegie Mellon University, Pittsburgh, PA. Manual, Version 0.3, unpublished internal report. Available via `http://www.cs.cmu.edu/~Xavier/`.

[Pearson, 1996] Douglas John Pearson (1996). *Learning Procedural Planning Knowledge in Complex Environments.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI. Available as Technical Report CSE-TR-309-96.

[Pell *et al.*, 1997] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams (1997). An autonomous spacecraft agent prototype. In *Proceedings of the First International Conference on Autonomous Agents*, pages 253–261, Marina del Rey, CA. (New York, NY: ACM Press).

[Pérez, 1995] M. Alicia Pérez (1995). *Learning Search Control Knowledge to Improve Plan Quality.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-95-175.

[Pomerleau, 1993] Dean A. Pomerleau (1993). *Neural network perception for mobile robot guidance*. (Dordrecht, Netherlands: Kluwer Academic).

[Pryor, 1994] Louise Margaret Pryor (1994). *Opportunities and Planning in an Unpredictable World*. PhD thesis, Northwestern University, Evanston, Illinois. Available as Technical Report number 53.

[Quinlan, 1993] J. Ross Quinlan (1993). *C4.5: Programs for Machine Learning*. (San Mateo, CA: Morgan Kaufmann).

[Rabiner & Juang, 1986] L. R. Rabiner and B. H. Juang (1986). An introduction to hidden Markov models. *IEEE ASSP Magazine*, 6(3):4–16.

[Rizzo *et al.*, 1997] Paola Rizzo, Manuela Veloso, Maria Miceli, and Amedeo Cesta (1997). Personality-driven social behaviors in believable agents. In *AAAI Fall Symposium on Socially Intelligent Agents*, pages 109–116, Cambridge, MA. (Menlo Park, CA: AAAI Press).

[Rizzo *et al.*, 1998] Paola Rizzo, Manuela M. Veloso, Maria Miceli, and Amedeo Cesta (1998). Goal-based personalities and social behaviors in believable agents. *Applied Artificial Intelligence*. Submitted Jan 1998.

[Rosenkrantz *et al.*, 1977] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal of Computing*, 6(3):563–581.

[Salganicoff & Ungar, 1995] Marcos Salganicoff and Lyle H. Ungar (1995). Active exploration and learning in real-valued spaces using multi-armed bandit allocation indices. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference (ICML95)*, pages 480–487, Tahoe City, CA.

[Schoppers, 1989] Marcel Joachim Schoppers (1989). *Representation and Automatic Synthesis of Reaction Plans*. PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL. Available as Technical Report UIUCDCS-R-89-1546.

[Segre, 1991] Alberto Segre (1991). Learning how to plan. *Robotics and Autonomous Systems*, 8(1-2):93–111.

[Shen, 1994] Wei-Min Shen (1994). *Autonomous Learning from the Environment*. (New York, NY: Computer Science Press).

[Simmons, 1991] Reid Simmons (1991). Concurrent planning and execution for a walking robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 300–305, Sacramento, CA. (New York, NY: IEEE Press).

[Simmons, 1994] Reid Simmons (1994). Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43.

[Simmons *et al.*, 1997] Reid Simmons, Rich Goodwin, Karen Zita Haigh, Sven Koenig, and Joseph O'Sullivan (1997). A layered architecture for office delivery robots. In W. Lewis Johnson, editor, *Proceedings of the First International Conference on Autonomous Agents*, pages 245–252, Marina del Rey, CA. (New York, NY: ACM Press).

[Simmons & Koenig, 1995] Reid Simmons and Sven Koenig (1995). Probabilistic robot navigation in partially observable environments. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1080–1087, Montréal, Québec, Canada. (San Mateo, CA: Morgan Kaufmann).

[Simmons *et al.*, 1990] Reid Simmons, Long-Ji Lin, and Chris Fedor (1990). Autonomous task control for mobile robots. In *Proceedings of the IEEE Symposium on Reactive Control*, pages 663–668, Philadelphia, PA.

[Simmons *et al.*, 1996] Reid Simmons, Sebastian Thrun, Greg Armstrong, Richard Goodwin, Karen Haigh, Sven Koenig, Shyjan Mahamud, Daniel Nikovski, and Joseph O'Sullivan (1996). Amelia. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, page 1358, Portland, OR. (Menlo Park, CA: AAAI Press). Introduction for robot competition.

[Stone & Veloso, 1996] Peter Stone and Manuela M. Veloso (1996). User-guided interleaving of planning and execution. In *New Directions in AI Planning*, pages 103–112. (Amsterdam, Netherlands: IOS Press).

[Stone *et al.*, 1994] Peter Stone, Manuela M. Veloso, and Jim Blythe (1994). The need for different domain-independent heuristics. In K. Hammond, editor, *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS-94)*, pages 164–169, Chicago, IL. (Menlo Park, CA: AAAI Press).

[Tan, 1991] Ming Tan (1991). *Cost-sensitive robot learning*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-91-134.

[Thrun, 1996] Sebastian Thrun (1996). A Bayesian approach to landmark discovery in mobile robot navigation. Technical Report CMU-CS-96-122, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Veloso, 1994] Manuela M. Veloso (1994). *Planning and Learning by Analogical Reasoning*. Springer Verlag, Berlin, Germany. PhD Thesis, also available as Technical Report CMU-CS-92-174, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

[Veloso *et al.*, 1995] Manuela M. Veloso, Jaime Carbonell, M. Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe (1995). Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120.

[Wang, 1996] Xuemei Wang (1996). *Leaning Planning Operators by Observation and Practice.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-96-154.

[Williamson & Hanks, 1994] Mike Williamson and Steve Hanks (1994). Optimal planning with a goal-directed utility model. In K. Hammond, editor, *Artificial Intelligence Planning Systems: Proceedings of the Second International Conference (AIPS-94)*, pages 176–180, Chicago, IL. (Menlo Park, CA: AAAI Press).

[Zhao *et al.*, 1994] Min Zhao, Nirwan Ansari, and Edwin S. H. Hou (1994). Mobile manipulator path planning by a genetic algorithm. *Journal of Robotic Systems*, 11(3):153–153.