Learning Situation-Dependent Costs: Improving Planning from Probabilistic Robot Execution

Karen Zita Haigh
khaigh@cs.cmu.edu
http://www.cs.cmu.edu/~khaigh

Manuela M. Veloso
mmv@cs.cmu.edu
http://www.cs.cmu.edu/~mmv

Computer Science Department,
Carnegie Mellon University,
Pittsburgh, PA. 15213-3891

Abstract

Physical domains are notoriously hard to model completely and correctly, especially to capture the dynamics of the environment. Moreover, since environments change, it is even more important for the system to learn from its own experiences. Our work focusses on learning for the *planning* stages of a physical system, where our algorithm learns the costs and probabilities of operating the environment.

Since actions may have different costs under different conditions, we introduce the concept of *situation-dependent rules*, in which situational features are attached to the costs or probabilities, reflecting patterns and dynamics encountered in the environment.

In this article, we present Rogue, a robot that analyzes its execution experiences to detect patterns in the environment. Rogue extracts learning opportunities from massive, continual, probabilistic execution traces. It then correlates these learning opportunities with environmental features, creating situation-dependent costs for its actions. We present the development and use of these rules for a robotic path 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 create plans that are tailored to the real world, and to respond to a changing environment. Physical systems should adapt to changing situations and absorb any information that will improve their performance.

Contents

1 Introduction

A system operating in a physical world 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 system should adapt to these situations and absorb any information that will improve its performance.

The challenges for designing a learning for a physical system are often due to representation differences between its planners and its 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 that information to incorporate future changes in the domain.

In this article, we present a learning mechanism for a real indoor mobile robot. Our approach learns the costs and probabilities of operating in an environment, and is able to identify when the cost of an action depends on the current situation. Moreover, our approach is responsive to changes in the environment, and hence provides the robot with the ability to adapt to changes and continuously improve its performance.

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. 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 as appropriate.

These steps are summarized in Table 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 that affect task achievability. This situation-dependent knowledge can be incorporated into

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

Table 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 2: General approach for learning situation-dependent costs.

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 different planners for a real robot, a path planner and a task planner. This article describes the implementation for the path planner. The implementation for the task planner can be found elsewhere [Haigh, 1998]. Our situation-dependent learning approach processes execution data to create improved domain models for both of its planners, thereby allowing them to create better quality, more efficient plans. Our approach effectively equips a real robot with the ability to learn from its own execution experiences.

Reader's Guide. We present our application domain in Section 2, along with the system architecture and representations of the relevant software modules. In Section 3, we present the mechanisms ROGUE uses to create training data for the learning algorithm. We describe how ROGUE extracts and evaluates learning events, $\mathcal{E}$, from the execution trace. We also discuss features, $\mathcal{F}$, including the characteristics of a good feature.

In Section 4, 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 5, we briefly describe how the path planner uses these situation-dependent arc costs to create efficient paths. We present our experimental results in Section 6. Related work can be found in Section 7. We present our conclusions and lessons learned in Section 8.

2 Implementation Domain

Our research explores the interaction of perception, cognition, action and learning in a complete integrated autonomous agent. Towards this end, we have built a system called ROGUE [Haigh & Veloso, 1997; Haigh & Veloso, 1998b; Haigh, 1998] that forms the task planning and learning layers for a real mobile robot, Xavier. 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. 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.

Xavier is a mobile robot being developed at Carnegie Mellon University [O'Sullivan *et al.*, 1997; Simmons *et al.*, 1997] (see Figure 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 2: Obstacle Avoidance, Navigation, Path Planning, Task Planning, and the User Interface.

Users send task requests to the task planner, which generates plans and sends plan steps to the robot for execution. The task planner combines plans for multiple interacting goals, reasons about task priority and compatibility, and interleaves planning with execution. The path planner calculates the path between two locations with the best expected travel time. The navigation module uses a Partially Observable Markov model to navigate the selected path.

Figure 1: Xavier the robot.

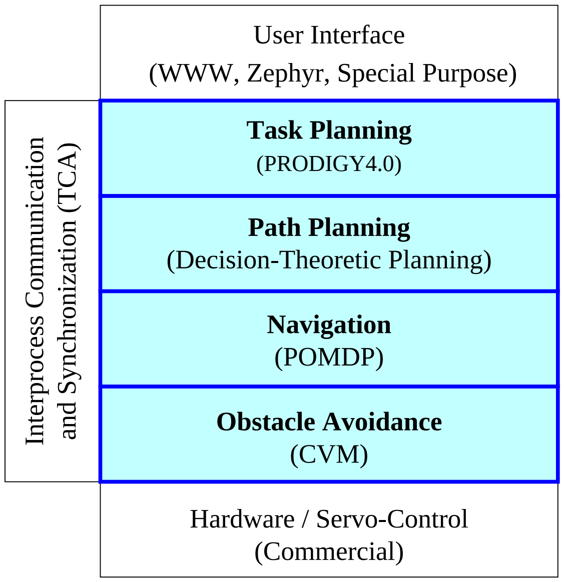


Figure 2: Xavier's primary software layers. Reproduced from Simmons *et al.* [1997].

Rogue adds to this architecture by providing a learning module. Rogue processes execution experience to help the task planner and the path planner improve the quality of their generated plans. In this article we focus on Rogue's learning capabilities as applied to the path planner.

We incorporate our situation-dependent learning approach in the Xavier architecture to capture patterns that affect costs of operating in the environment. For example, 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.

2.1 Situation-Dependent Learning Example

Consider the following example. For Xavier, the most challenging region of its environment is the lobby of our building. Figure 3 shows the map of the main floor, and Figure 4 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 peak hours (coffee and lunch breaks), it is virtually impossible for the robot to efficiently navigate through the lobby.

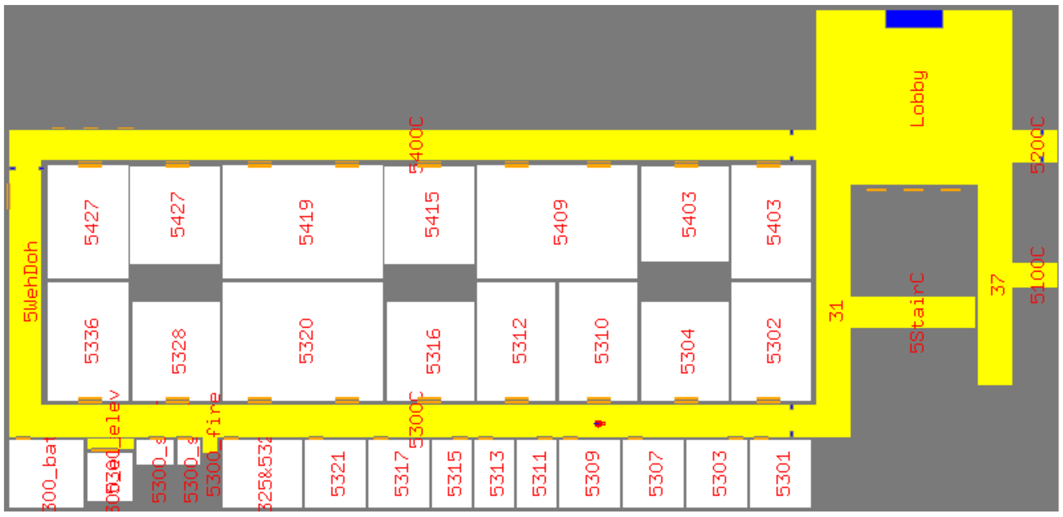


Figure 3: Robot's map (half of the 5th floor of our building).

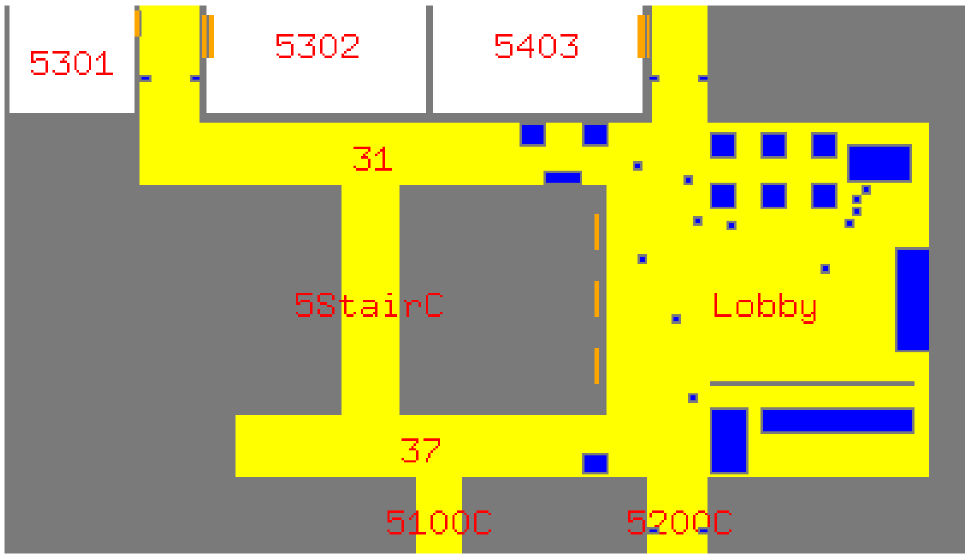


Figure 4: Closeup of map; typical obstacles added for the reader: small obstacles indicate people, while larger ones indicate tables and food carts.

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 5, would force the planner to avoid the lobby during lunch break.

Creating a pre-programmed model of these dynamics would be not only time-consuming, but very likely would not capture all relevant information, particularly in a changing environment. ROGUE can reduce the burden on the programmer because its learning capabilities

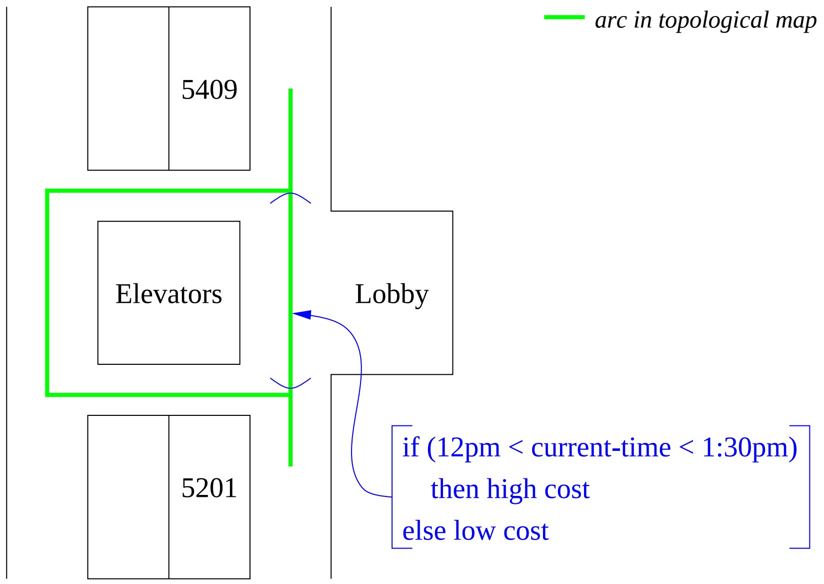


Figure 5: A high-level view of a sample learned rule for the path planner; Rogue learns actual traversal costs.

modify the existing domain model to reflect real world experience. Rogue extracts relevant information from the execution data to learn patterns and identify changes in the environment. Rogue then creates situation-dependent rules that the planners can use to improve plan quality.

2.2 Learning for the Path Planner

When applying our situation-dependent learning algorithm to Xavier's path planner, our concern is to improve the reliability and efficiency of selected paths. Figure 6 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, 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.

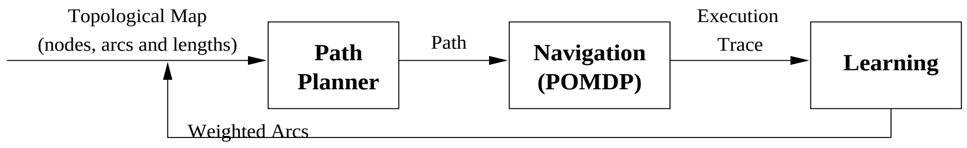


Figure 6: Learning for the path planner.

By learning appropriate arc cost functions, ROGUE helps 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. Features, $\mathcal{F}$, include both robot sensor data and high-level features such as date and goals. ROGUE's situation-dependent learning algorithm will then create a mapping from features and events to costs: $\mathcal{F} \times \mathcal{E} \rightarrow \mathcal{C}$. The path planner can then use these *situation-dependent costs* to create better estimates of a path's expected execution time.

Execution traces 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 challenge of our learning approach is to create arc costs that depend on high-level features of the environment. In the implementation for Xavier's path planner, an additional challenge is to process vast amounts of uncertain, continual navigation data. Note that our situation-dependent learning 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 processes execution data to improve plan quality.

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

2.2.1 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*.

[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 3, page 6) [Cassandra *et al.*, 1994; Lovejoy, 1991]. Reasoning about blockage probabilities and recovery costs is also notably easier in the topological map.

- 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.

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.

2.2.2 Navigation

Navigation on the robot is done using Partially Observable Markov Decision Process models (POMDPs) [Simmons & Koenig, 1995; Koenig, 1997]. 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 probabilistic models describe variance on its metric information. This information is automatically compiled into a POMDP model.

Table 3 shows the Bayesian probability update calculation. Figure 7 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

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; o_t is the observation received at time t.
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: Bayesian probability updates.

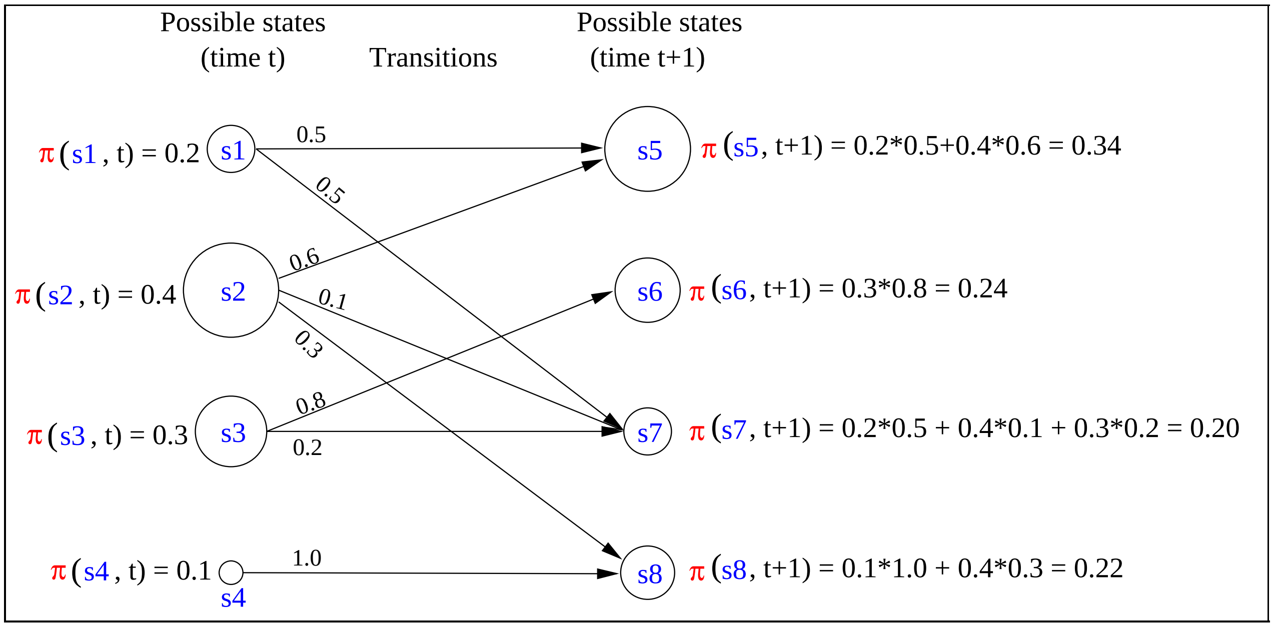

Possible states (time t)
Transitions
Possible states (time t+1)
π(s1, t) = 0.2
s1
π(s2, t) = 0.4
s2
π(s3, t) = 0.3
s3
π(s4, t) = 0.1
s4
0.5
0.5
0.6
0.1
0.3
0.8
0.2
1.0
s5
s6
s7
s8
π(s5, t+1) = 0.2*0.5+0.4*0.6 = 0.34
π(s6, t+1) = 0.3*0.8 = 0.24
π(s7, t+1) = 0.2*0.5 + 0.4*0.1 + 0.3*0.2 = 0.20
π(s8, t+1) = 0.1*1.0 + 0.4*0.3 = 0.22

Figure 7: 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.

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

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

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.

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 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 8 demonstrates this change. Figure 8a shows the probability distribution before the robot sees the wall at the end of the corridor, while Figure 8b shows the distribution after.

The metric variance (length uncertainty) of the map alters the structure of the Markov model. In our system, we use *parallel Markov chains*, where each corresponds to one of the possible lengths of the edge. Figure 9 illustrates an example for a corridor that may be two, three or four metres long. This representation is an effective way to model worlds in which lengths are not known with certainty.

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

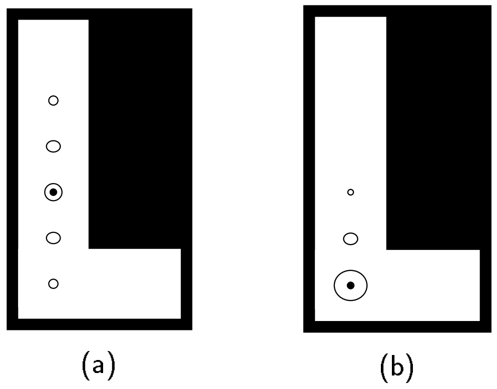


Figure 8: Markov state probability distribution, (a) before and (b) after observing the wall at the end of the corridor. Circles indicate probability distribution; large circles have high probability. At each time step, the most likely state is marked with a dot.

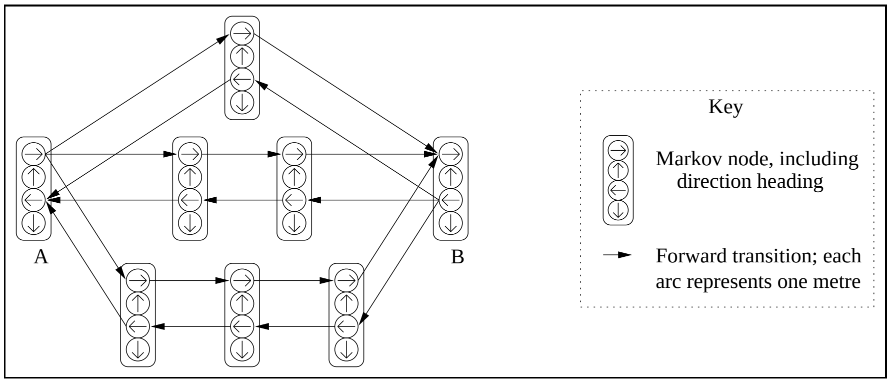

A
B
Key
Markov node, including direction heading
Forward transition; each arc represents one metre

Figure 9: 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].

3 Training Data

Our situation-dependent learning algorithm correlates situational features to learning opportunities so that planners can predict and avoid similar situations in the future. The algorithm requires a list of the learning events, $\mathcal{E}$, the evaluation of each of the events, $\mathcal{C}$, and the values of each of the situational features during the event, $\mathcal{F}$. Each event, its cost, and its feature values are then placed in an *events matrix* for use by the learning algorithm.

3.1 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. Rogue examines the execution trace, identifies the most likely path that the robot traversed, and then identifies the corresponding path planner arcs. Rogue 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.

This process can be described pictorially as in Figure 10. 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.

The POMDP navigation module keeps track of the *most likely states* but not the *most likely sequence of states*. The algorithm to calculate this sequence is known as Viterbi's algorithm [Rabiner & Juang, 1986]. Viterbi's algorithm is guaranteed to find the single best state sequence with the highest probability, given the actions, observations and initial state distribution. 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.

Below, we describe the workings of Viterbi's algorithm and our extension of it. We then

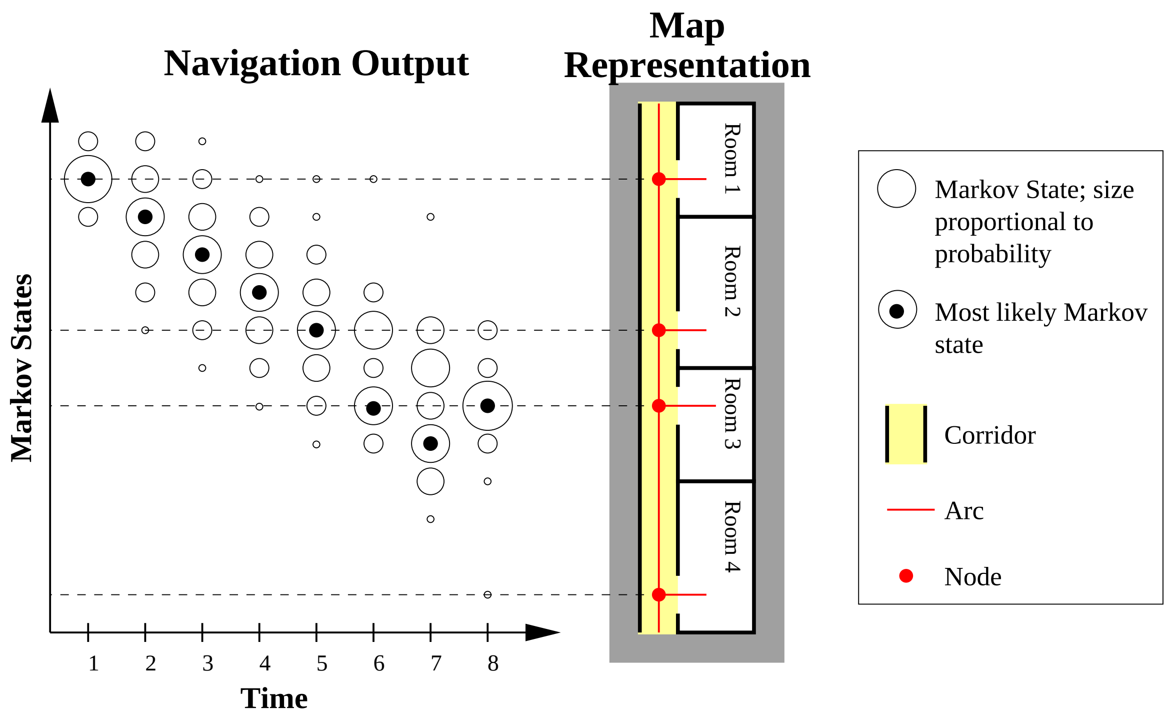


Figure 10: Extracting arc traversals from Markov state distributions.

present the techniques used to calculate the arc sequence so that arc traversal events can be identified.

3.1.1 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 4. 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 $\pi$ 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 $\delta$ 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 $\mathrm{MAX}_{\forall s \in S}[\delta(s, T)]$,
                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 4: Viterbi's Algorithm, reproduced from Rabiner & Juang [1986].

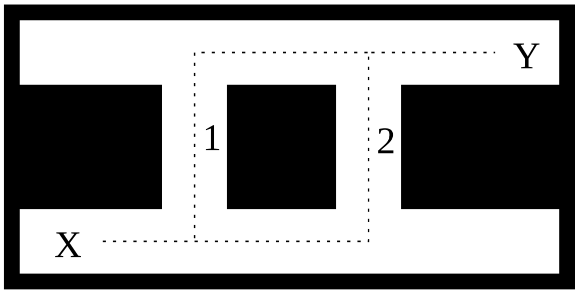


Figure 11: A map showing why the most likely state sequence may be different from the most likely states.

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 11, 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.

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 ignores length uncertainty. 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 12 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 7, and the transition probabilities, A, are also the same. Recall that POMDP probabilities are calculated as shown in equation 1. Viterbi's algorithm,

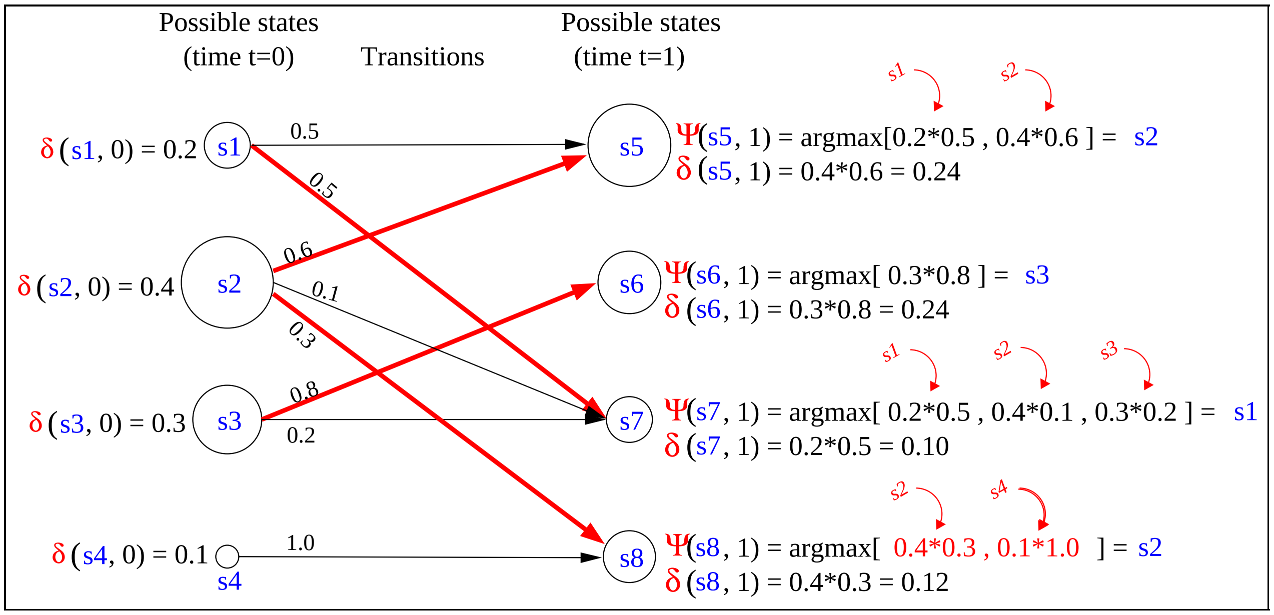

Possible states (time t=0)
Transitions
Possible states (time t=1)
δ (s1, 0) = 0.2
δ (s2, 0) = 0.4
δ (s3, 0) = 0.3
δ (s4, 0) = 0.1
s1
s2
s3
s4
s5
s6
s7
s8
0.5
0.5
0.6
0.1
0.3
0.8
0.2
1.0
Ψ(s5, 1) = argmax[0.2*0.5 , 0.4*0.6] = s2
δ (s5, 1) = 0.4*0.6 = 0.24
Ψ(s6, 1) = argmax[0.3*0.8] = s3
δ (s6, 1) = 0.3*0.8 = 0.24
Ψ(s7, 1) = argmax[0.2*0.5 , 0.4*0.1 , 0.3*0.2] = s1
δ (s7, 1) = 0.2*0.5 = 0.10
Ψ(s8, 1) = argmax[0.4*0.3 , 0.1*1.0] = s2
δ (s8, 1) = 0.4*0.3 = 0.12

Figure 12: 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).

meanwhile, maintains the probability distribution of the sequence, δ, calculated as:

$$\delta(s_j, t+1) = \frac{1}{k}\delta(\Psi(s_j, t+1), t) \times A_a(\Psi(s_j, t+1), s_j) \times \mathcal{O}(s_j, o_{t+1}), \tag{2}$$

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:

$$\begin{aligned}\Psi(s_j, t+1) &= s_i \text{ such that } s_i \text{ gives } \mathrm{MAX}_{\forall s_i \in S}\left[\delta(s_i, t) \times A_a(s_i, s_j)\right] \\ &= \mathrm{ARGMAX}_{\forall s_i \in S}\left[\delta(s_i, t) \times A_a(s_i, s_j)\right].\end{aligned} \tag{3}$$

Viterbi's algorithm finds the sequence at time t that contributed the most probability to the sequence at time $t+1$. In Figure 12 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.

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.

In our models, Viterbi's algorithm finds the most likely sequence that reflects *trajectory and length*; we would like an algorithm that finds the most likely *trajectory*. In other words, we want the algorithm to identify the robot's trajectory in the topological map, which is an abstract representation of the Markov model.

Essentially, the fact that a given node may "fan-out" leads to information loss and a poor estimate of the best path. The fan-in/fan-out representation of the model effectively captures the length uncertainty of the environment, but Viterbi's algorithm is unable to generate a good estimate of the abstracted trajectory.

For example, consider Figure 13. Because node $s1$ splits into three parallel Markov chains, while the lower probability state, $s2$, splits into two, Viterbi's algorithm selects the sequence through $s2$ as the most likely sequence from $s3$. 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 14, 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

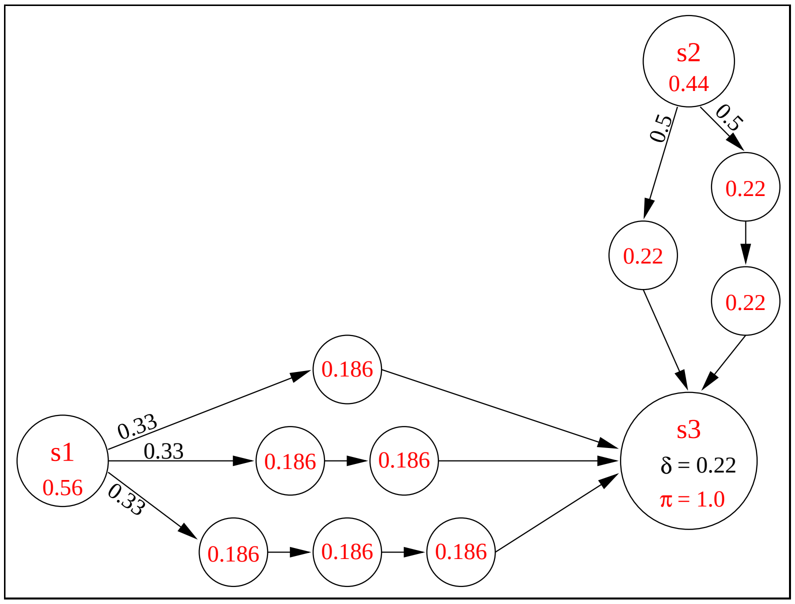


Figure 13: Fan-in: Example of how the map representation affects Viterbi's algorithm. Although it is more likely that the robot passed through $s1$, the Viterbi sequence generated from $s3$ passes through $s2$ instead.

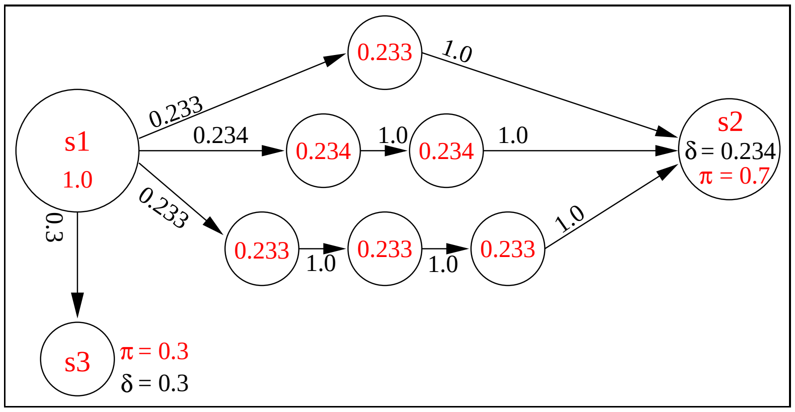


Figure 14: Fan-out: Example of how the map representation affects Viterbi's algorithm. Although $s2$ has a greater π probability than $s3$, Viterbi's algorithm selects $s3$ as the sequence-generating state.

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 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 $s = Seq_{_T}(T) = \text{ARGMAX}_{\forall s' \in S}\,[\delta(s',t)]$.

Multi/Markov Viterbi

Ideally, we would like Viterbi's algorithm to ignore length uncertainty and 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[4], and so we instead use an approximate method.

We modify Viterbi's algorithm in three ways:

1. Rogue uses the most likely POMDP state as the sequence generator. We know that the π distribution is always 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

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

(the most likely Viterbi state, δ, at time T), Rogue uses

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

[4]Note to the reviewers: We can provide a discussion paragraph or reference.

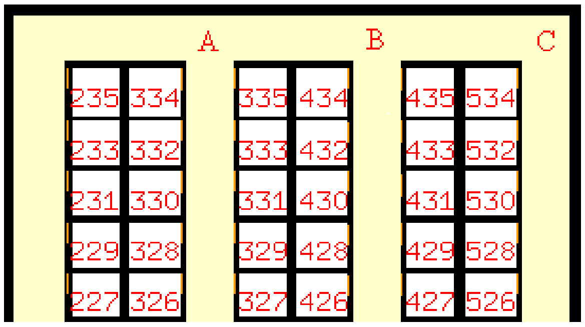


Figure 15: Map used in the example of how multiple sequences are used.

(the most likely POMDP state, π, at time T). Effectively, this change forces Viterbi's algorithm to use the standard 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, $P(s|w)$.

2. Rogue sets a threshold for the minimum probability for the generating state. That is, Rogue selects $t \leq T$ and a threshold τ such that $Seq_t(t) > \tau$. In this way, we can ensure that we select high probability sequences and eliminate low probability ones.

3. Rogue uses the Viterbi sequences generated from *many* high probability states *throughout* the trace:

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

By using many sequences, Rogue collects evidence for the most likely actual trajectory, and thereby compensates for the poor estimates made by Viterbi's algorithm.

We call the modified algorithm *Multi/Markov Viterbi* because we use *multiple* trajectories generated from the most likely *Markov* state.

Reconsider the probabilities and transitions shown in Figure 13. 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 15. Imagine that the robot travels up one of the central corridors, and then turns right towards point C. Assume the 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 one, the later sequence provides corroborating evidence for the earlier one. 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 5.

3.1.2 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 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

```
Define $\tau$ 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 $\pi$ state at
        time $t$; $s = Seq_t(t')$ is the state at time $t'$ in $Seq_t$, for $0 \leq t' \leq t$.

Calculate $\pi$, $\delta$ and $\Psi$ as in Tables 3 and 4. Recall that $\Psi$ depends on $\delta$.
Let $\mathcal{V} = \emptyset$.
Foreach $t$, $0 < t \leq T$:
      Let $s_{max} = \mathrm{ARGMAX}_{\forall s \in S} \pi(s,t)$
      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 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.

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, one metre 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 16 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.

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, each node at a corridor junction 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.

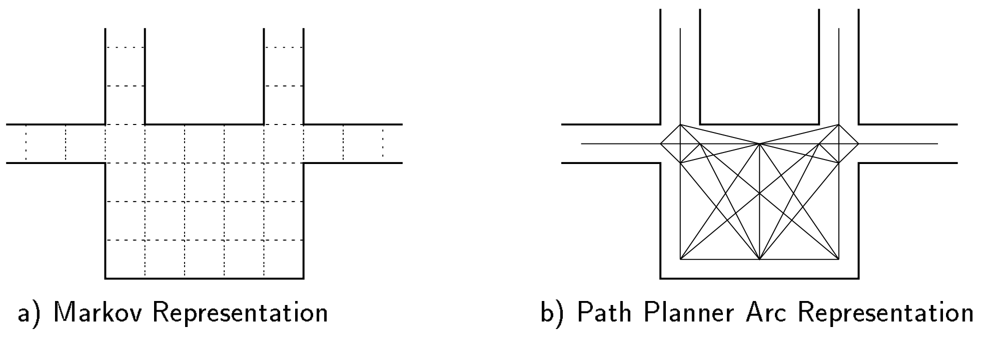


Figure 16: Different representations of a foyer.

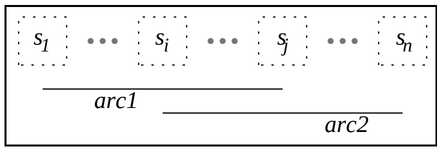


Figure 17: Multiple arcs corresponding to multiple Markov nodes. arc_1 corresponds to $s_1, ..., s_j$, and arc_2 corresponds for $s_i, ..., s_n$.

The mapping function then assigns states to arcs in a greedy manner, based on expectation times. Consider Figure 17, 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 time-stamps on each state s_k, $t(s_k)$, then we say that states s_1 through s_k correspond to arc_1 for:

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

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 18 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

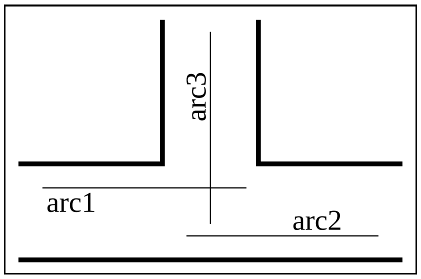


Figure 18: An example of when the greedy heuristic may fail.

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.

3.1.3 Summary of Event Identification

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.

3.2 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.

The cost evaluation function, $\mathcal{C}$, for the path planner 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 is 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.

3.3 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.

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. "Execution-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 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 article, 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 when learning for the task planner [Haigh, 1998], where the current reading (whether or not a door is open) affects the next immediate decision.

3.4 Events Matrix

Each training event is stored in a matrix along with its cost evaluation and the environmental features observed when the event occurred. Those environmental features which change during the traversal are averaged. Table 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 experiments described in Section 6, while sonar readings were added for the task planner experiments [Haigh, 1998].

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 complete process for identifying, evaluating and storing arc traversal events from

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 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 $\tau$
      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 $\varepsilon$ from $\mathcal{C}$: $\mathcal{C}(\varepsilon) = vt/l$
             Store the arc traversal event $\varepsilon$, the features $\mathcal{F}$, and
                 the weight $\mathcal{C}(\varepsilon)$ in the events matrix
```

Table 7: Identifying arc traversal events $\mathcal{E}$ from the execution trace.

the trace is summarized in Table 7. Step 1 corresponds to Section 3.1.1, step 2 corresponds to Section 3.1.2, and step 3 corresponds to Section 3.2. 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.

4 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 3.2. 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]

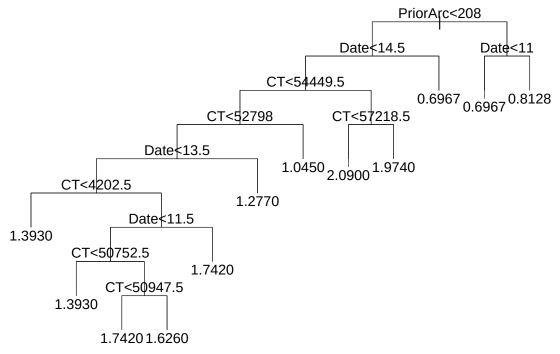
Splits are selected to maximize the reduction in the deviance of the node. Chambers & Hastie [1992] discuss the method in more detail.

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.

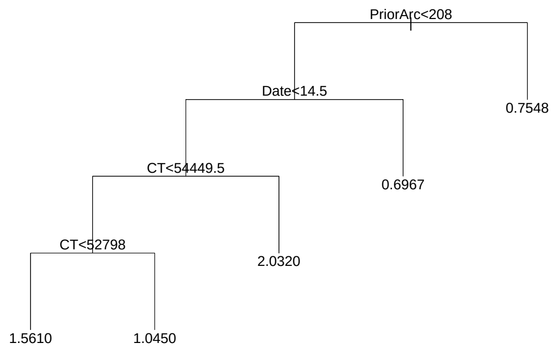
Figure 19 shows a learned tree, before and after pruning. The pruned tree represents the situation-dependent arc costs of arc 208.[6] Each internal node in the tree represents one feature comparison. The left subtree indicates data for which the feature was less than the comparison value; the right subtree contains data for which the feature was greater than the comparison value. Leaf nodes show the arc's learned costs.

[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.

[6]Arc 208 appears in corridor 2 of the Exposition world described in Section 6.1.



(a) Before pruning.



(b) After pruning.

Figure 19: A learned tree. Leaves represent learned costs (traversal weights); CT is current time, in seconds since midnight.

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.

Section 6 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.

5 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 a 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 2.2.1, 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.

6 Experimental Results

We will present two sets of experiments in this article. The first simulated-world set demonstrates that ROGUE can learn patterns. The second was run on the real robot, validating the algorithm and the need for it. We have also performed experiments testing rule stability, data generalization, and learning rates [Haigh, 1998].

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.

6.1 Simulated World: Learning Patterns

The first environment tests ROGUE's ability to learn situation-dependent costs. Figure 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 20a shows the simulated world, complete with a set of possible obstacles. Figure 20b shows the topological map used by the path planning module; this map displays everything the robot "knows" about its environment.

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 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)

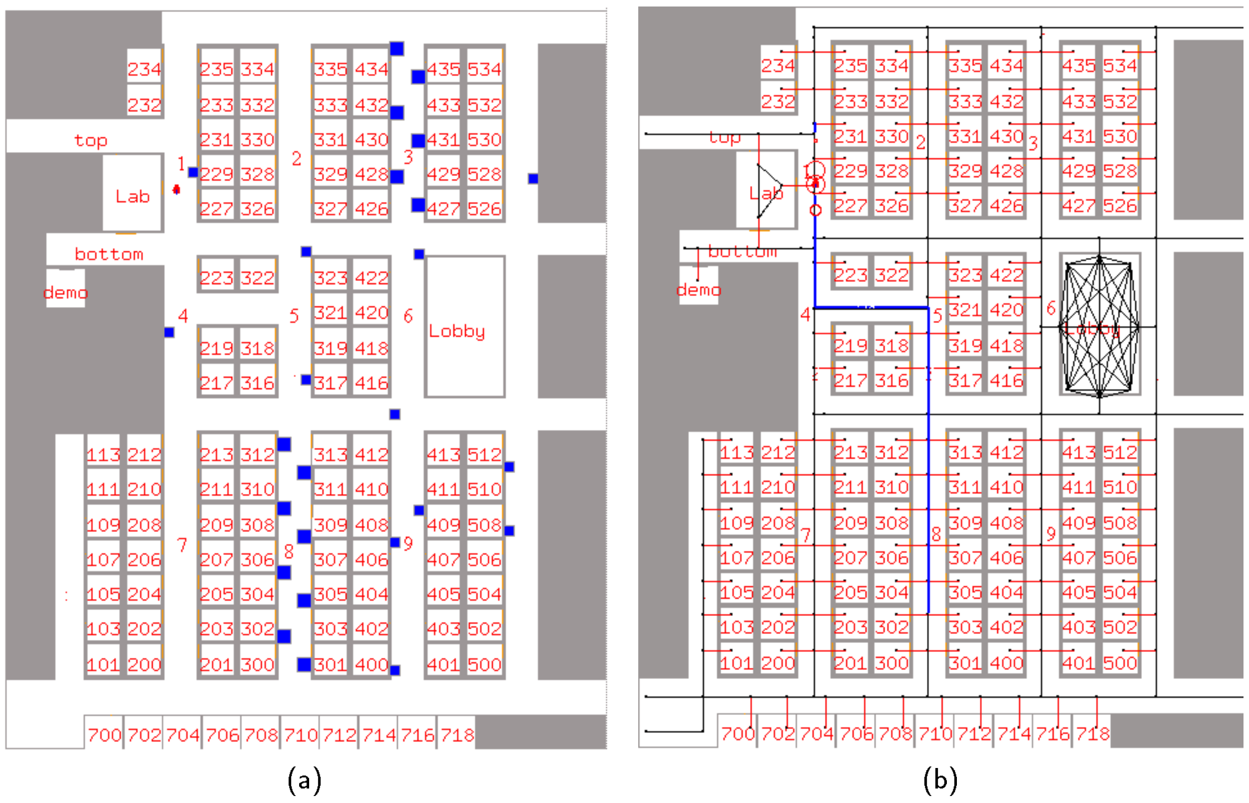


Figure 20: Exposition world. (a) Simulator: operating environment. Obstacles marked with dark boxes. (b) Path Planner: topological map. Arcs shown in light grey, a sample path shown darker.

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 actual user requests in Section 6.2.)

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 3.1 and 3.2, and then processed as described in Section 4.

6.1.1 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 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,

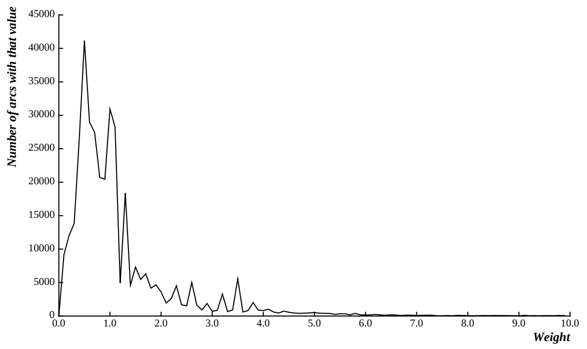


Figure 21: Arc cost frequency: most arcs in the training set have a cost close to 1.0, the default value.

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.

Figure 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 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.

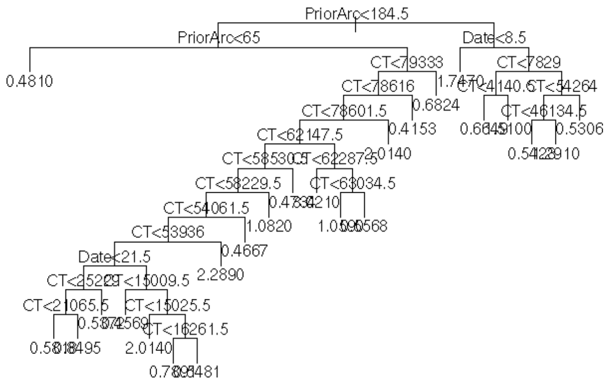
Figure 23 shows the cost, averaged over all the arcs in each corridor, as it changes throughout the day. Rogue has correctly identified that corridor 3 is difficult to traverse between midnight and 3am, and also noon and 3pm. During the rest of the day, it is close to default cost of 1.0. This graph 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 8 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.

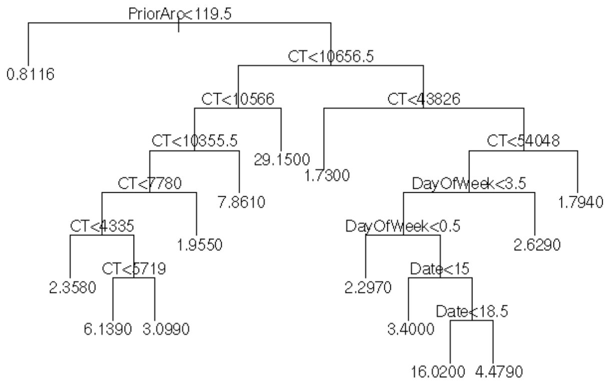
Figure 24 shows learned expensive arcs for Wednesday at 01:05am. As expected, corridor 2 is considered inexpensive, while corridors 3 and 8 are considered expensive. As the cost

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

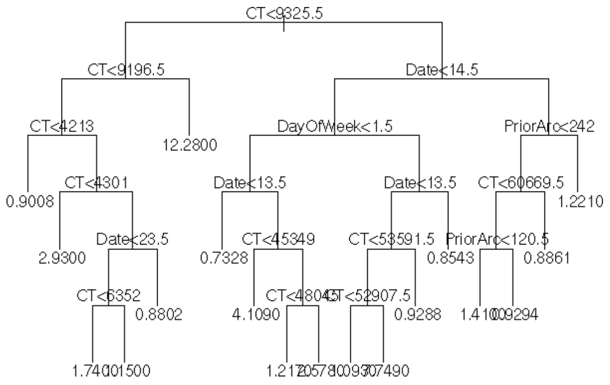
Table 8: The average cost of all the arcs in each type of corridor.



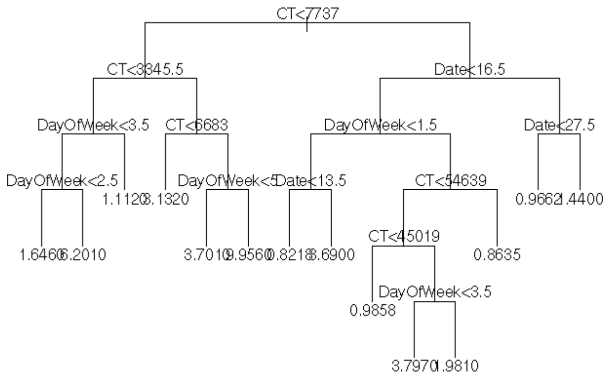
(a) Arc 238



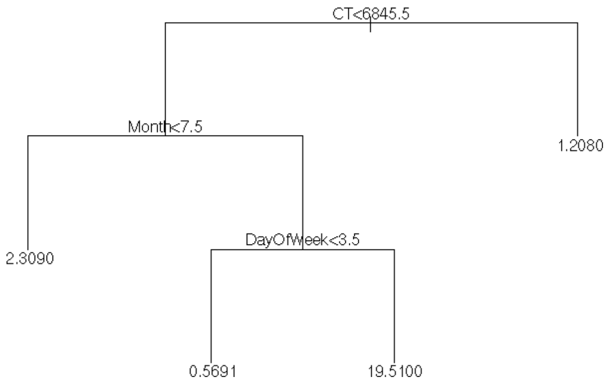
(b) Arc 240



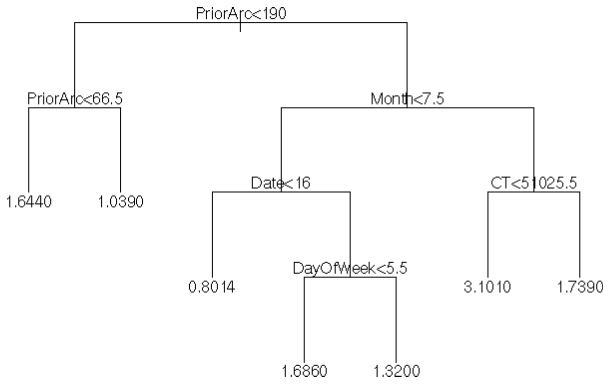
(c) Arc 242



(d) Arc 244

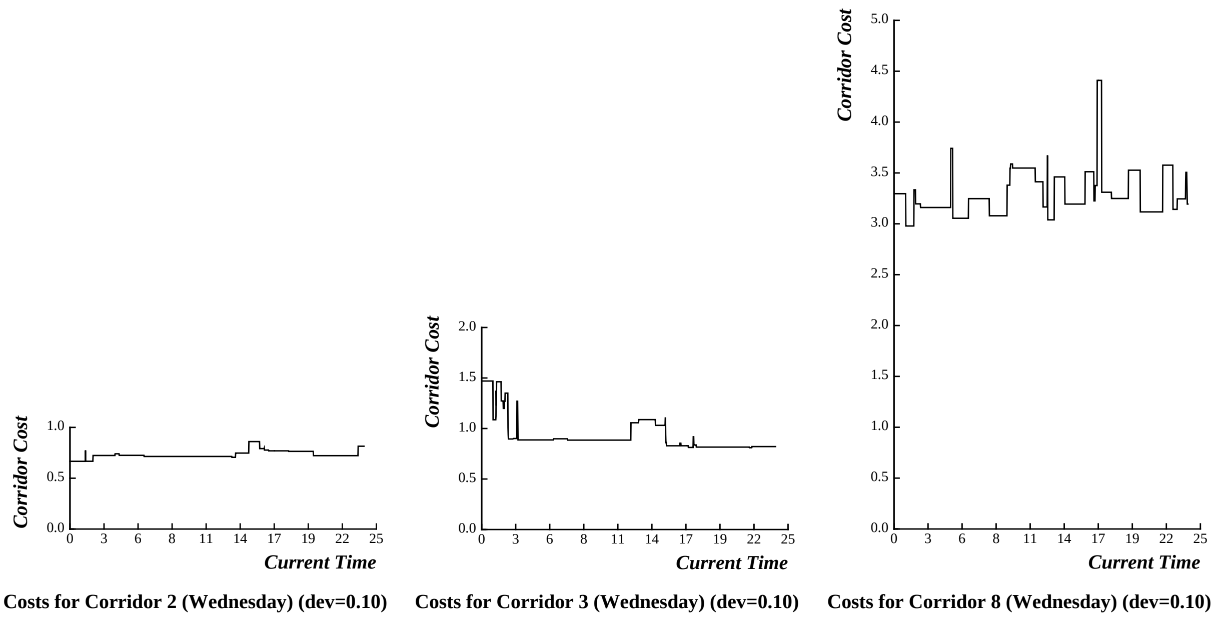


(e) Arc 246



(f) Arc 248

Figure 22: Learned trees for the six arcs in corridor 3.



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

Figure 23: Learned corridor cost (average over all arcs in that corridor) for Wednesdays.

threshold increases, fewer arcs are considered expensive, and in particular all arcs containing random obstacles have been eliminated. 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.

For comparison, Figure 25 shows learned costs for Tuesday at 09:45am. Note that corridor 3 is not considered expensive at any time.

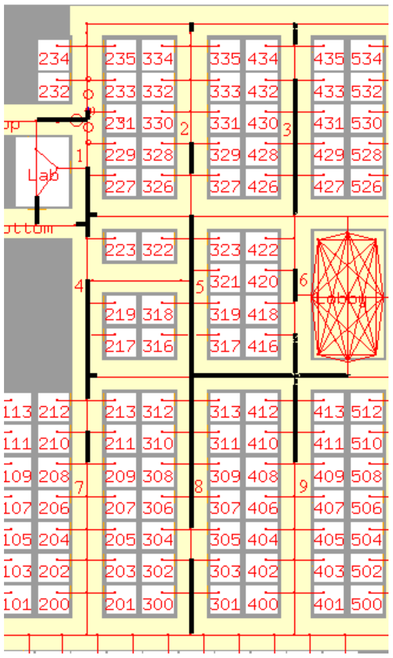
The data collected for this experiment has shown that ROGUE's learning algorithm successfully identified patterns in the environment. ROGUE also successfully identified both permanent and temporary phenomena.

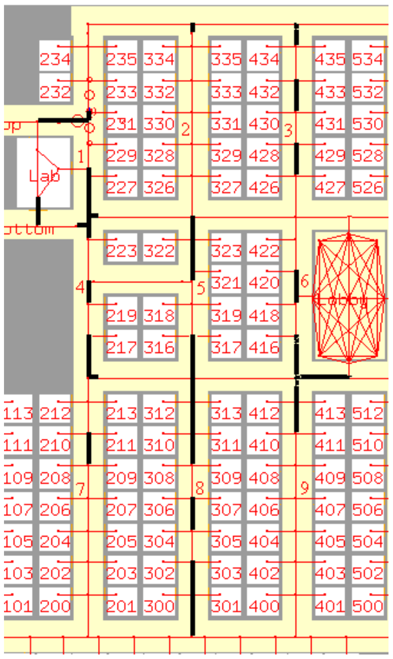
6.1.2 Effect on Path Planner

Figure 26 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 26a shows the path normally generated. Figure 26b shows the path generated by the path planner after learning; note that the expensive arcs have been avoided.

Table 9 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 10 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


(a) Costs $>$ 1.25


(b) Costs $>$ 2.00

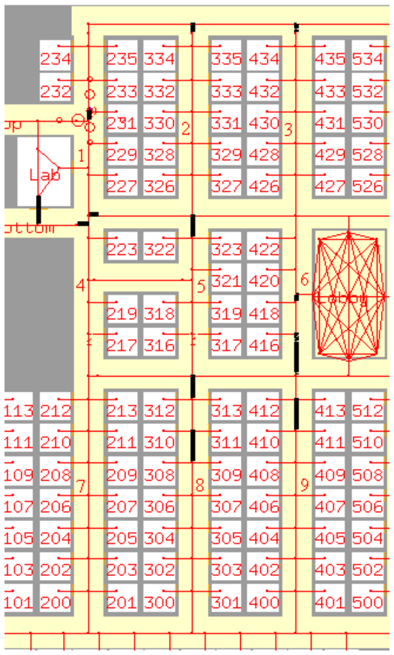
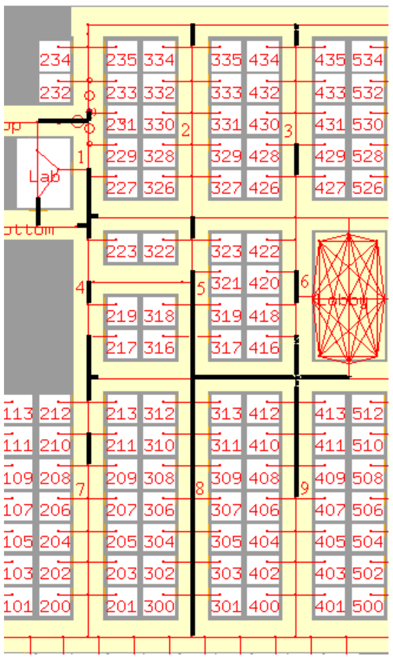
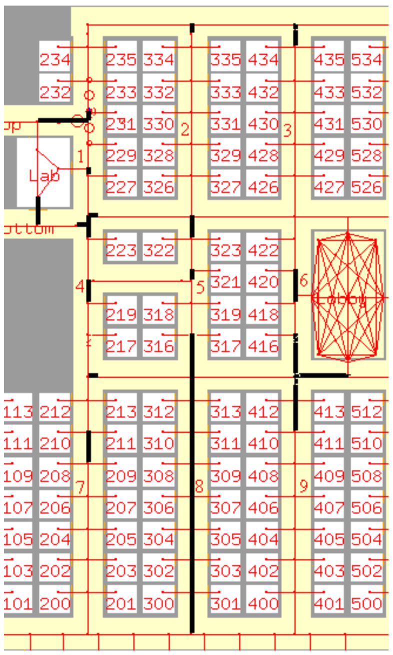

(c) Costs $>$ 5.00

Figure 24: 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

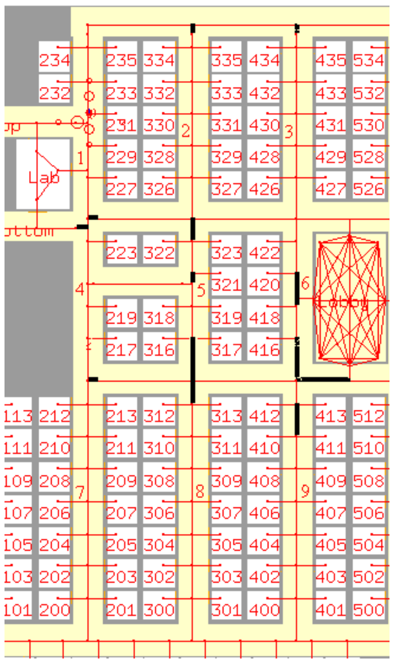
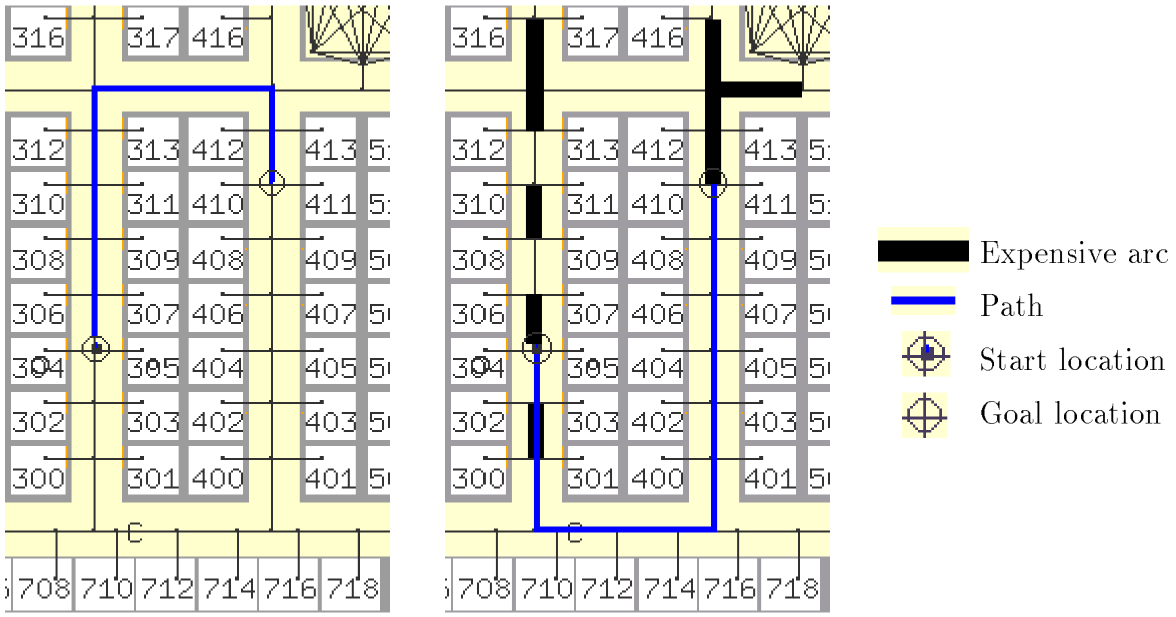

(c) Costs $>$ 5.00

Figure 25: Expensive arcs for situation: Tuesday, 09:45am. Note that corridor 3 is not considered expensive.



(a) Default Path (b) New Path

Figure 26: 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 9: 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 10: 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.

than the default 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.

6.2 Real Robot

The second set of data was collected from real Xavier runs on the fifth floor of our building (part of which was shown previously in Figure 3).

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.

6.2.1 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 to 82 minutes.

More than 15,000 arc traversal events were recorded. Trees were learned for 89 arcs from an average of 169 traversals per arc. The average tree size was 20.4 nodes (10.2 leaf nodes).

Figure 27 shows the average learned costs for all the arcs in the lobby on a particular Wednesday. Values differentiated by other features were averaged[7]. The histogram shows the number of execution traces per time step. 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.

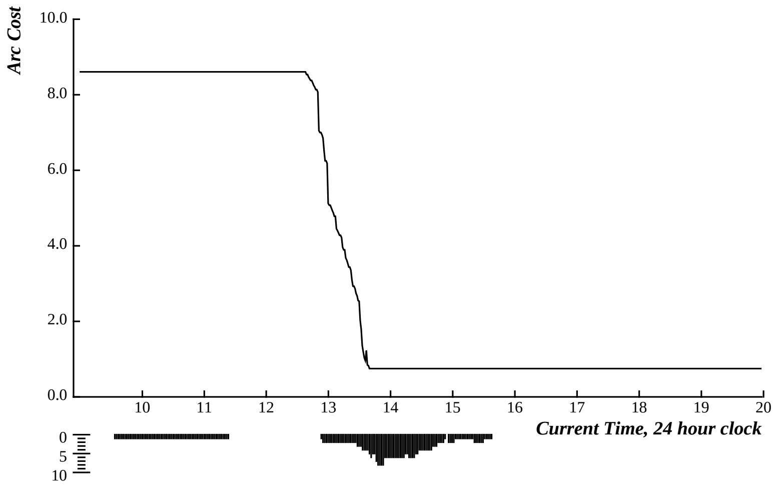


Figure 27: Learned costs for Wean Hall lobby on Wednesday, August 6. (Data from April-July 1997.) The histogram below the graph indicates volume of training data, in terms of number of execution traces; most data was collected between 1:30pm and 2:45pm.

6.2.2 31 October 1997

During the subsequent three months, an additional 42 traces were collected, yielding a total of 59 execution traces, containing a total of 72,516 events. 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. Average tree size was 23.1 nodes (11.5 leaf nodes).

Figure 28 shows the average learned costs for all the arcs in the lobby on a particular Wednesday. Values differentiated by other features were averaged. The histogram shows the number of execution traces per time step.

[7]Note that since the robot operates in a less controlled environment, many features may affect the cost of an arc. In the exposition world, other features do not appear in the trees.

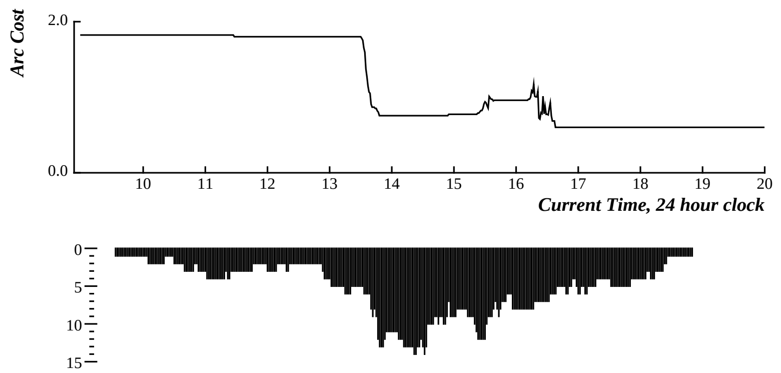


Figure 28: Learned costs for Wean Hall lobby on Wednesday, November 11. (Data from April-October 1997.) The histogram below the graph indicates volume of training data, in terms of number of execution traces; most data was collected between 1pm and 6pm.

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 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.

7 Related Work

This section describes research closely related to that presented in this article. Our work contributes to the machine learning community and the robotics community.

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,

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

- learning symbolic descriptions of actions, and
- learning plan quality.

7.1 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.

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.

7.2 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] also relaxes the assumption about complete and correct sensing, while still managing 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.

7.3 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.

Much of the research towards plan quality has focussed on learning search control rules.

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.

Most of the remaining 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.

ROGUE's situation-dependent costs guide the path 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.

8 Conclusion

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. 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. Planners can then use these situation-dependent rules to make better decisions.

We instantiated this framework with Xavier's path planner, creating 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 that the path planner uses to create routes for the robot. ROGUE effectively identifies relevant training data, i.e. arc traversal events, $\mathcal{E}$, in the execution trace. ROGUE then correlates the events with situational features, $\mathcal{F}$, to create updated costs, $\mathcal{C}$. These costs, represented as learned regression trees, 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 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.

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

8.1 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, 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.

We have applied our approach to Xavier's symbolic task planner, successfully learning action probabilities and creating control knowledge to guide the planner's decisions [Haigh, 1998]. Other 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.

Since the learning approach is planner-independent, it is usable from any execution module to any planner, regardless of data representations. The important point is that the system 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.

8.2 Important Issues

Several important issues need to be considered when incorporating situation-dependent costs into a system.

How to extract learning opportunities, and how to design 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. It is important to make the critical components accessible to external modules.

How to identify and add 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.

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 [Haigh, 1998]. 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.

8.3 Future Research Directions

This work has opened up several areas for future research.

One area of possible research involves extending the cost evaluation function for events. In particular, the cost function for arc traversals 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 discuss possible approaches elsewhere [Haigh, 1998].

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.

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.

References

[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).

[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).

[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).

[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, 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.

[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, 1998] Karen Zita Haigh, *Situation-dependent Learning for Interleaved Planning and Execution.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-98-108.

[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, 1997] Karen Zita Haigh and Manuela M. Veloso (1997). Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots.* 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.

[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).

[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.

[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.

[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.

[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.

[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).

[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.

[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, 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).

[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.

[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.

[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.