Situation-Dependent Learning from Uncertain Robot Data

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

August 29, 1997

Abstract

Real world robot tasks are so complex that it is hard to hand-tune all of the domain knowledge, especially to model the dynamics of the environment. Many researchers have therefore been exploring Machine Learning techniques. Most of the existing work falls into map learning, sensor/action mapping, and vision. The work presented in this paper explores Machine Learning techniques for robot planning.

Our system collects execution traces, and extracts relevant information to improve the efficiency of generated plans. In this article, we present the representation of the path planner and the navigation modules, and describe the execution trace. We show how training data is extracted from the execution trace. We describe the learning issues, including in particular how we handle the massive stream of continuous, probabilistic data.

We introduce the concept of *situation-dependent costs*, where situational features can be attached to the costs used by the path planner. In this way, the planner can generate paths are appropriate for a given situation. We present experimental results from a simulated, controlled environment as well as from data collected from the actual robot.

1 Introduction

Robots operating in the real world have many tasks they need to perform autonomously. Reliability and efficiency are key issues when designing real systems. The robot must be able to effectively deal with noisy sensors and actuators, as well as incomplete and changing knowledge about the environment. It must act efficiently, in real time, to deal with dynamic situations. In most current systems, the behaviour of an autonomous robot is completely dependent on the predictive ability of the programmer. Most real world environments are hard to model completely and correctly. Moreover, the world is dynamic and models need to account for changes. Any system that repeats its errors cannot be considered reliable. To be *truly* autonomous, the robot needs to be able to use

accumulated experience and feedback about its performance to improve its behaviour. It needs to *learn*.

Learning consists of processing execution episodes situated in a particular task context and interpreting feedback from successes and failures into reusable knowledge. 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, Pomerleau, 1993]). Instead of improving low-level *actuator* control, our work focusses at the *planning* stages of the system. A few other researchers have explored this area as well, learning and correcting action models (e.g. [Klingspor *et al.*, 1996, Pearson, 1996]), or learning costs and applicability of actions (e.g. [Lindner *et al.*, 1994, Shen, 1994, Tan, 1991]). Our work falls into the latter category.

These systems learn improved domain models and this knowledge is then fed back into the system's planner, as *costs* or *control knowledge*, so that the planner can then select more appropriate actions. CSL [Tan, 1991] and Clementine [Lindner *et al.*, 1994] both learn sensor utilities, including which sensor to use for what information. LIVE [Shen, 1994] learns a model of the environment, as well as the costs of applying actions in that environment.

We do not believe, however, that it is enough to simply learn that a particular action has a certain average probability or cost. Actions may have different costs under different situations. Instead of learning a global description, we would rather that the system learn the pattern by which these situations can be identified. The system needs to learn the correlation between features of the environment and the situations, thereby creating *situation-dependent costs*.

We would like a path planner to learn, for example, that a particular corridor gets crowded and hard to navigate when classes let out. 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 one of its agents has a weaker arm and can't pick up the heaviest packages. Once these problems have been identified and correlated to features of the environment, the planner can then predict and avoid them when similar conditions occur in the future.

A similar concept was used by Mitchell *et al.* (1994) for scheduling meetings. The CAP system learned patterns in the environment for determining meeting location, duration and day of week. Haigh *et al.* (1997) used situational features in a case-based reasoning system to select good cases

for planning.

We have been developing a robot architecture which aims at equipping a real robot with the ability to learn from its own execution experiences. It processes uncertain navigation data to create improved domain models for its planners, successfully abstracting numeric sensor information into symbolic planner information.

Our approach relies on examining the execution traces of the robot to identify situations in which the planner's behaviour needs to change. Our approach requires that the robot executor defines the set of available situation *features*, $\mathcal{F}$, while the planner defines a set of relevant *events*, $\mathcal{E}$, and a *cost function*, $\mathcal{C}$, for evaluating those events. Events are things 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 planner to evaluate the situation and select appropriate actions. The learner then creates a mapping from the execution features and the events to the costs:

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

Once the mapping has been created, the information is fed back into the system so that future planning will avoid re-encountering the problem events. These steps are summarized in Table 1. 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.

1. Create plan
2. Execute; record execution trace
3. Identify events $\mathcal{E}$ in execution trace
4. Learn mapping: $\mathcal{F} \times \mathcal{E} \rightarrow \mathcal{C}$
5. Update planner

Table 1: Algorithm for learning situation-dependent costs.

We are using the Xavier robot [Simmons *et al.*, 1997] as our learning platform. Knowledge in the path planner is represented as a topological map of the environment in which the robot navigates. The map is a graph with nodes and edges representing office rooms, corridors, doors and lobbies. Learning appropriate arc-cost functions will allow the path planner to avoid troublesome areas of the environment when appropriate. Therefore events for this planner are arc traversals, while costs

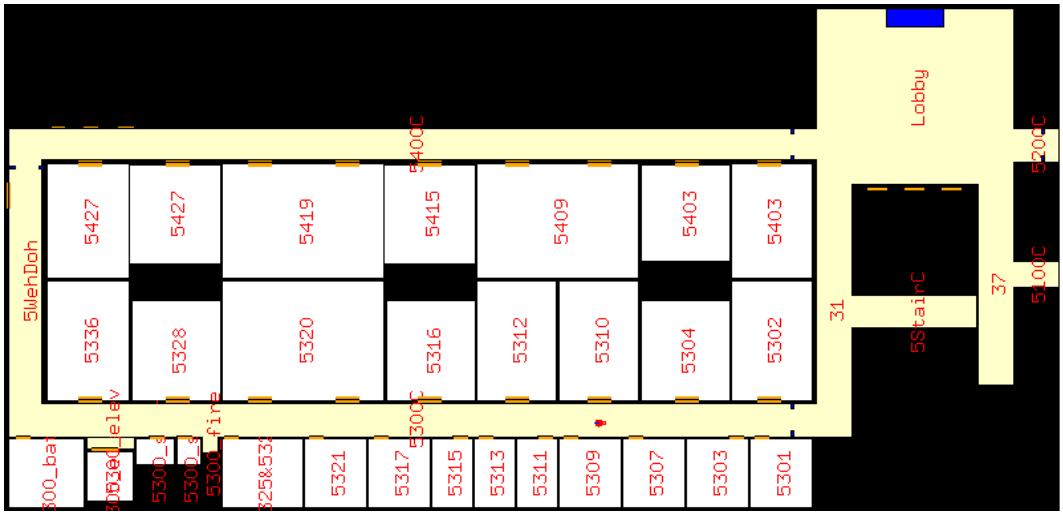


Figure 1: Robot's Map (half of the 5th floor of our building)

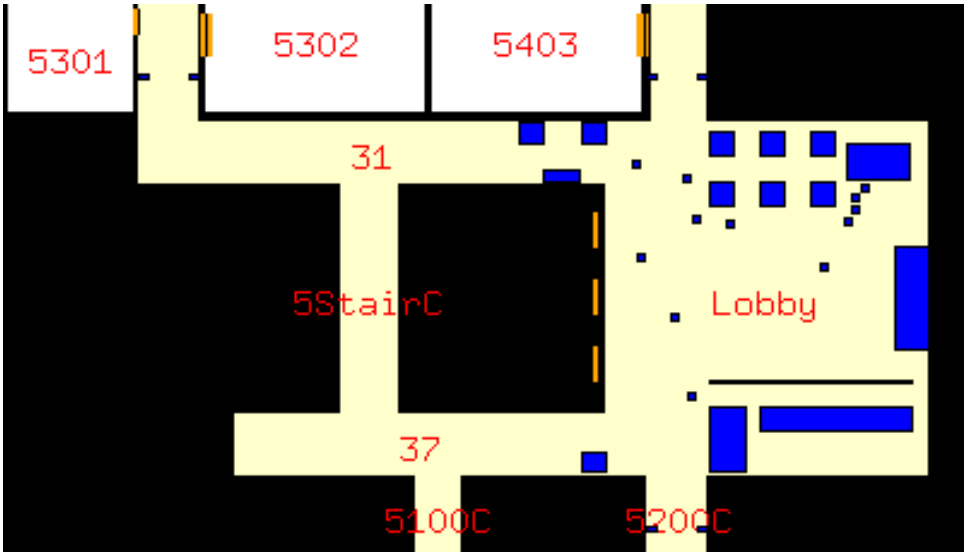


Figure 2: Closeup of map; typical obstacles added for reader

are defined as a function of travel time and position confidence.

Xavier's execution trace is generated by a probabilistic navigation module. Identifying the planners' events from this trace is challenging because the execution traces contain a massive, continuous stream of uncertain data.

To illustrate the goal of our system, consider the following example. For Xavier, the most challenging region of its environment is in the lobby of our building. Figure 1 shows the map of the main floor, and Figure 2 shows a closeup of the lobby area, with typical obstacles added

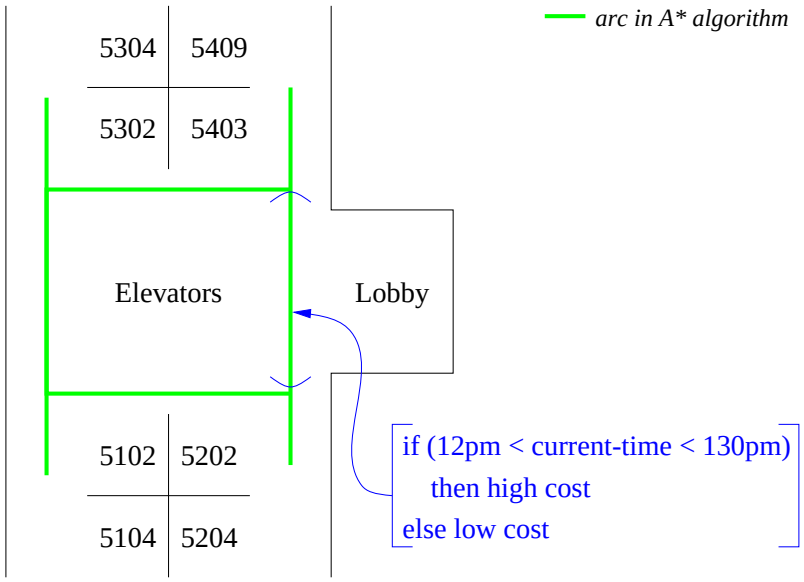


Figure 3: A sample rule

for the reader's benefit (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.

In this example, we would like Xavier learn *when* to avoid the lobby *completely*. A direct path from room 5204 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 3, would force the planner to avoid the lobby during lunch break.

In this article, we present the learning algorithm as it applies to Xavier's path planner. Our system demonstrates the ability to *learn situation-dependent costs* for the arcs of the path planner. It extracts relevant training data from the massive, continuous, probabilistic execution traces, and creates appropriate rules that the planner can use to create more efficient paths.

We present the representations of the path planner and the navigation module in Section 2. In Section 3 we describe the execution trace, describe how we use it to identify training data (planner arc traversals) from the probabilistic information. 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 describe how the path planner uses these situation-dependent arc costs to create efficient paths. In Section 6 we present some experimental results. Finally, in Section 7 we summarize the main points of the article.

2 Architecture & Representation

Figure 4: Xavier the Robot

Xavier is a mobile robot being developed at CMU [O'Sullivan *et al.*, 1997, Simmons *et al.*, 1997] (see Figure 4). It is built on an RWI B24 base and includes bump sensors, a laser light striper, 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: Obstacle Avoidance, Navigation, Path Planning, Task Scheduling, and the User Interface. The architecture is described in more detail by Simmons *et al.* (1997) .

In this paper, our concern is to improve the reliability of path planning. The execution traces from which learning can occur are provided by the navigation module. Figure 5 shows how our algorithm fits into the framework of the Xavier architecture. The path planner uses a modified A* algorithm on a topological map that has additional metric information [Goodwin, 1996]. Navigation is done using Partially Observable Markov Decision Process Models (POMDPs) [Simmons & Koenig, 1995]. Our learning algorithm affects the arc costs of the topological map so that the planner will select plans with a higher efficiency.

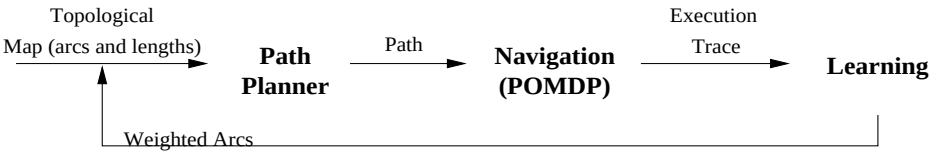


Figure 5: Learning Framework

The main challenge of this work is to process vast amounts of uncertain, continuous navigation data. At no point in the robot's execution is the robot aware of 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.

A second challenge in this research exists as a result of the representation gap between the path planner and the navigation module. The path planner uses a topological map, while the navigation module uses a Markov state representation. There is, therefore, no clear correspondence between the representation used in the navigation module and the representation used in the planning module. More discussion on this point follows in Section 3.2.

We now discuss the available representations, and present how we have dealt with the challenges.

2.1 The Path Planner

The path planner determines how to travel efficiently from one location to another. Actuator and sensor uncertainty complicates path planning since the robot may not be able to accurately follow a path, and the shortest distance path is not necessarily the fastest. Consider, for example, the two paths from A to B shown in Figure 6. Although Path 1 is shorter than Path 2, the robot might

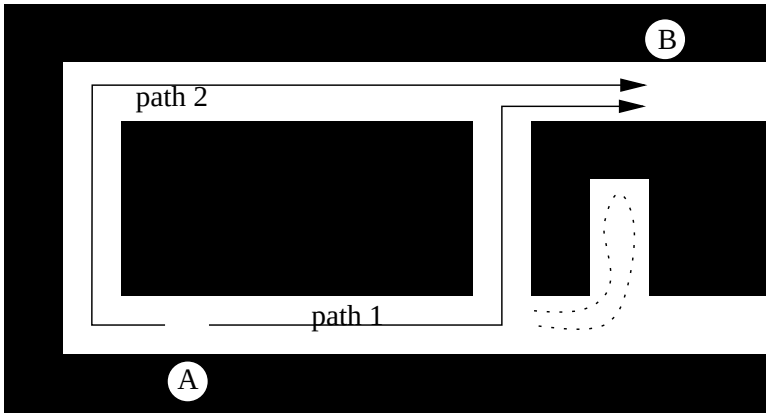


Figure 6: Two paths from A to B. Although Path 1 is shorter, the robot might miss the turn and get trapped in the dead-end.

miss the first turn on Path 1 and have to backtrack. This problem cannot occur on the other path since the end of the corridor prevents the robot from missing the turn. The longer path might take less time on average.

Plans are generated using a decision-theoretic A* search strategy [Goodwin, 1996]. The planner operates on the topological map (augmented with metric information) rather than using the POMDP model directly.[1] The topological map is a standard arc/node representation.

The planner creates a plan with the best expected travel time. The travel time of a complete path is calculated as a function of distance, traversal weight, blockage probability and recovery costs. 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). Finally, recovery costs estimate the difficulty of recovering from a failure, such as missing a turn or discovering a closed door.

2.2 Navigation

Navigation on the robot is done by using Partially Observable Markov Decision Process Models (POMDPs) [Simmons & Koenig, 1995]. The navigation module works by estimating the current location of the robot, determining the direction the robot should be heading at that location to follow the path, and then setting a directional heading.

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

Observations of the world help prune unlikely states from the probability distribution, and regular observations can keep the robot fairly certain of its location. However, if the robot does

[1] It is infeasible to determine optimal POMDP solutions given our real-time constraints and the size of our state spaces (over 3000 states for the map shown in Figure 1) [Cassandra *et al.*, 1994, Lovejoy, 1991].

not receive any observations for a long time (e.g. in a long feature-less corridor), the probability distribution may spread over many states, making it impossible to determine with any precision the exact location of the robot.

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

3 The Execution Trace & Event Identification

The goal of learning in our system is to identify events ($\mathcal{E}$) during execution which do not meet expectations, and to then correlate situational features ($\mathcal{F}$) to those events. Events are then evaluated by a cost function ($\mathcal{C}$). In this section, we describe the execution trace, and how we identify events from it.

Events in any planner can be identified by asking the question: "*What will change the planner's behaviour?*" For Xavier, 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). Problem events are therefore arc traversals that do not meet expectations. Improved cost estimates on these arc traversals will cause the planner to select more appropriate plans.

Arc cost estimates are primarily determined by traversal weight. The traversal weight of an arc can be calculated by examining the travel times of the arcs *actually* travelled. The difference between expected travel time and actual travel time yields an appropriate modifier for the traversal weight.

The kinds of features available in Xavier include speed, time of day, sonar observations (walls, openings), camera images (empty, crowded, cluttered,...), 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.

An *execution trace* from the robot is generated by the navigation module. The data is therefore presented to the learning module using the probability distribution over Markov states. The trace

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 use the Markov state distributions to identify arc traversals. We do this by examining the execution trace, identifying the most likely path that the robot traversed, and then identifying the corresponding planner arcs. We can then map situational features to the arc traversals to create situation-dependent costs.

3.1 Identifying the Most Likely Path

Probabilistic navigation systems have a common feature in that, by maintaining probability distributions for all possible positions, they can quickly recover from sensor errors. This feature has a side-effect, however, in that the robot never knows where it *actually* is. At any given moment, it may believe that it is in one of several different locations.

However, the *action sequence* stored in the trace, together with the probability distribution, can be used to calculate the most likely path the robot took through the Markov states.

The algorithm to calculate this path is known as Viterbi's algorithm [Rabiner & Juang, 1986]. By starting at the most likely Markov state at a given time, and then using the transition probabilities between Markov states, it estimates which state the robot was in during the previous time step. Recursing backwards through time, it can calculate the complete path.

Viterbi's algorithm is a slight modification to the standard POMDP algorithm used for navigation. 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.

Note that as each new observation yields a new probability distribution, the most likely path might change significantly. 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 a very poor estimate of its location, some distance from the end of the corridor. Figure 7 demonstrates this change. Figure 7a shows the probability distribution before the robot sees the wall at the end of the corridor.

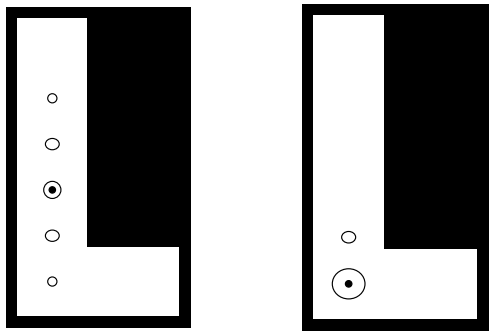


Figure 7: Markov state probability distribution, (a) before and (b) after observing the wall at the end of the corridor. Circles indicate probability distribution, large circles have high probability. At each time step, the most likely state is marked with a dot.

3.2 Identifying the Planner's Arcs

Once the set of most-likely Markov paths has been reconstructed, we need to identify which of the planner's arcs the robot traversed. Although it might appear to be a simple problem, the representation of the planner and of the POMDP are significantly different and the mapping is not direct.

The POMDP represents the world in a set of discrete square blocks. In our environment, 1 meter squares have been found to be empirically reliable and 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. Figure 8 demonstrates the difference for a lobby area. Although these representations clearly make sense for each module, there is no direct correlation between the Markov states and the arcs.

A similar problem exists at junctions in corridors. Our hallways are wider than the Markov

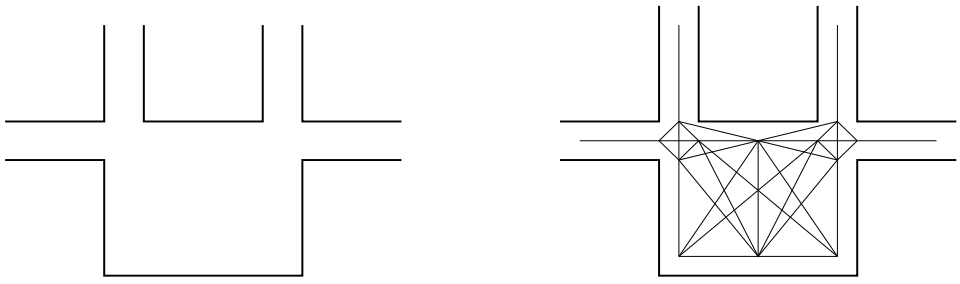
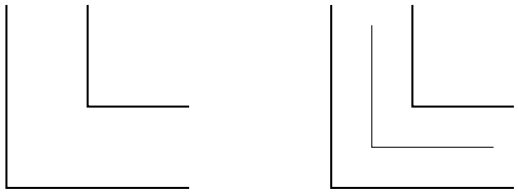


Figure 8: Different representations of a Foyer



a) Markov Representation b) Planner Arc Representation

Figure 9: Different representations of Junctions in Corridors

precision, but along the corridor, we do not represent the full width. This decision increases efficiency while maintaining reliability. At junctions, however, we need finer control of the robot, and therefore finer estimates of location, and therefore we retain the full representation. Figure 9 shows the different representations. It is unclear which arcs correspond precisely to which Markov states.

We have addressed these problems by creating a list of all the arcs that could possibly correlate to the Markov node. For example, the four nodes at junctions in corridors would correspond to all the planner arcs that meet there. These differences mean that there are often Markov nodes associated with multiple arcs. This fact complicates the reconstruction of the arc path because a single Markov sequence may map to multiple arc sequences.

We can reduce the number of possible arc sequences by permitting only those arcs which correspond to the *transition* between adjacent Markov states. However, for a single Viterbi sequence, we are still left with many possible arc sequences. Selecting the correct one is a challenging problem that we address heuristically, based on expectation times.

3.3 Handling Uncertainty in the Data

The probabilistic nature of the navigation module is the primary source of challenges when identifying planner arc traversals.

A single execution trace yields multiple Viterbi sequences (one at each time step), many of which will be similar. For example, if the trace contains several consecutive high-probability locations, Viterbi's algorithm will yield several arc sequences that differ only in their last slot. Arcs appearing earlier in the sequence are effectively the same data point. However, we cannot simply decide to use only a *single* Viterbi sequence from the trace. It is not correct to say that later sequences subsume earlier ones, since each time step yields new information, and it is possible that no single

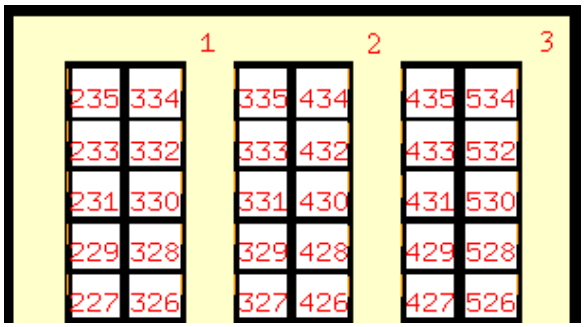


Figure 10: Map used in example of how multiple sequences are used.

Viterbi path is completely correct.

For example, if we use the path generated from the most likely Markov state at the end of the execution trace, we might have a poor prediction of the robot's actual path, since that Markov state might have low probability. We therefore need to use a Markov state with a high probability, since the path from that location is more likely to be correct than sequences generated from low probability locations. However, even this high probability location might be wrong (since it's not 100%), and so our system uses the arc sequences yielded from *many high probability locations*. By using many sequences, the system is able to compensate for the uncertainty in the data.

For example, consider the map shown in Figure 10. Imagine that the robot travels up one of the central corridors, and then turns right towards point 3. The robot initially believes it is heading towards point 1, in the "300" corridor (but because of position uncertainty, it might be in the "400" corridor). 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, and the probability masses around points 1 and 2. Point 1 will have a higher probability, say 0.60, while point 2 is 0.30 and other places with the remaining 0.10. The sequence generated at this moment (from point 1) is then used for learning. Later in the episode, the robot arrives at point 3 with 0.90 probability. The Viterbi sequence generated from here shows that it is more likely that the robot travelled up the "400" corridor. This second sequence is *also* used for learning. Neither of the two sequences is necessarily correct, and so using both allows the system to cover both possibilities.

Another challenge in identifying path planner events is ambiguity. Recall the dead-end example presented in Figure 6. Let the desired arc list for path 1 correspond to arcs $a_1 ... a_n$. When the system reconstructs the traversed arc sequence, it gets: $a_1, ..., a_i, b_1, ..., b_m, a_{i+1}, ..., a_n$, where $b_1, ..., b_m$ correspond to the arcs in the dead-end. At this point, the system needs to decide whether:

To blame it on the past: to consider a_i expensive to traverse,

To blame it on the future: to consider a_{i+1} expensive to traverse,

To blame it on the connection: to consider a_{i+1} as partially blocked from the direction of a_i,

or

To blame it on the divergence: to learn and modify recovery costs (see Section 2.1).

Our system currently arbitrarily decides to blame it on the connection, but there are occasions when this decision is incorrect. Future work includes investigating the impact of these incorrect decisions, and if necessary, building heuristics to choose the correct decision.

3.4 Creating a Uniform Output

Once arc traversal events have been identified from the execution trace, they are listed in a table format along with the cost evaluation and the environmental features observed when the event occurred.

The cost evaluation function, $\mathcal{C}$, yields an updated arc traversal weight. This weight is equal to the average velocity on that arc times the time spent traversing it, divided by the length. Those environmental features which change during the traversal are averaged.

Table 2 shows a sampling from an *events table* generated by the system. The table is presented in a uniform format for use by any learning mechanism.

This table 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 table. The learning algorithm processes the entire body of data, and creates a new set of situation-dependent rules for use when the robot is next run. By using incremental learning, the system is able to more quickly learn a model of the environment, as well as to notice and respond to changes on a continuous basis.

4 Learning Algorithm & Issues

In this section we present the learning mechanism we use to create the mapping from situation features ($\mathcal{F}$) and events ($\mathcal{E}$) to costs ($\mathcal{C}$).

The input to the algorithm is the events table described in Section 3.4.

The desired output is situation-dependent knowledge in a form that can be used by the planner.

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

ArcNo	Weight	CT	Speed	PriorArc	Goal	Year	Month	Date	DayOfWeek
233	0.348354	38108	34.998001	234	90	1997	06	30	1
232	0.264036	38105	34.998001	233	405	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 2: Arc Traversals ($\mathcal{E}$) listed with Environmental Features ($\mathcal{F}$) valid at time of the traversal. *Weight* is arc traversal weight, *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, DayOfWeek* is the date of the traversal.

- 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*.

Standard decision trees do not handle continuous valued output particularly well, Bayesian learning would not successfully handle disjunctive functions, k-Nearest Neighbour algorithms would not handle irrelevant features well and neural networks would not generalize well for sparse data [Mitchell, 1997]. 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 fitted for each arc using *binary recursive partitioning* where the data is successively split until data is too sparse or nodes are pure (below a preset deviance). Splits are selected to maximize the reduction in deviance of the node. 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. Chambers & Hastie (1992) discuss the method in more detail. Figure 11 shows one sample regression tree built with our data.

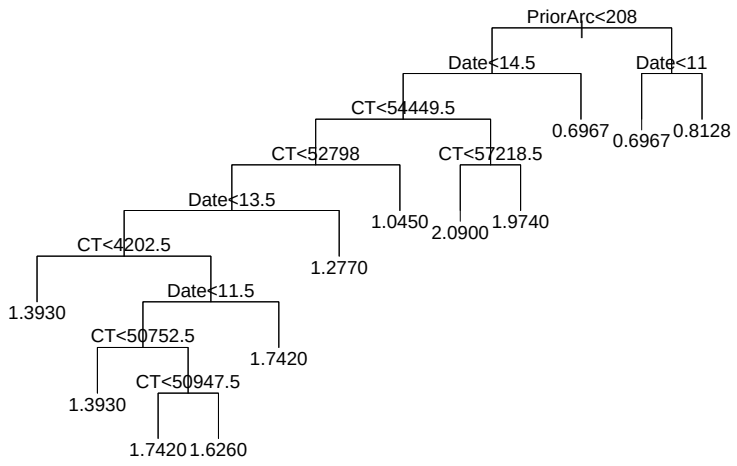


Figure 11: Tree for arc 208

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. 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 overfit the data. The least important splits are then pruned off the tree until it reaches the desired size. Figure 12 shows the tree after pruning.

Incremental learning raises the issue of the tradeoff between ignoring temporary phenomena and learning long-term patterns. Most temporary phenomena are treated as noise, but occasionally, they will last long enough to affect learning. A scheme such as giving lower weights to older data would effectively "forget" temporary phenomena after sufficient time has past. However, weighting

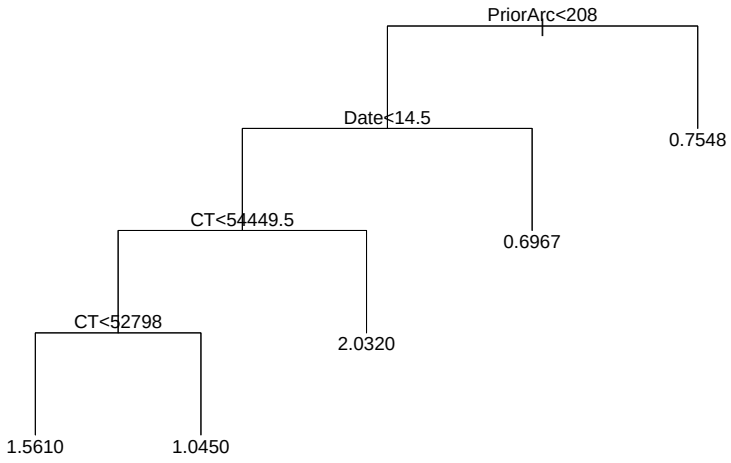


Figure 12: The pruned tree from arc 208.

data in this manner would adversely affect the system's ability to learn long-term patterns, such as those occurring annually. Currently, we weight all data equally, allowing the system to learn long-term patterns.

Section 6 presents the results of using regression trees to learn situation-dependent costs in this learning task. 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 planner, and significantly enhance the system.

5 Updating the Path Planner

Once the set of regression trees have been created (one for each arc), they are ready for use by the planner. Each time the path planner generates a path, it requests the updated arc costs. These costs are generated by matching the current situation against each arc's learned tree.

The planner will then use the updated cost to calculate the best path. If the updated arc cost is high, then the 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.

6 Experimental Results

To date, we have conducted two sets of experiments. The first set involves a simulated world because it is an environment where the experiments can be tightly controlled. The second set was run on the real robot, validating the algorithm and the need for it.

6.1 Simulated World

Xavier has a simulator whose primary function is 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 is functionally equivalent to 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.

We have built a world in which we tested the algorithm. The simulator allows us to tightly control the experiments to ensure that the learning algorithm is indeed learning appropriate situation-

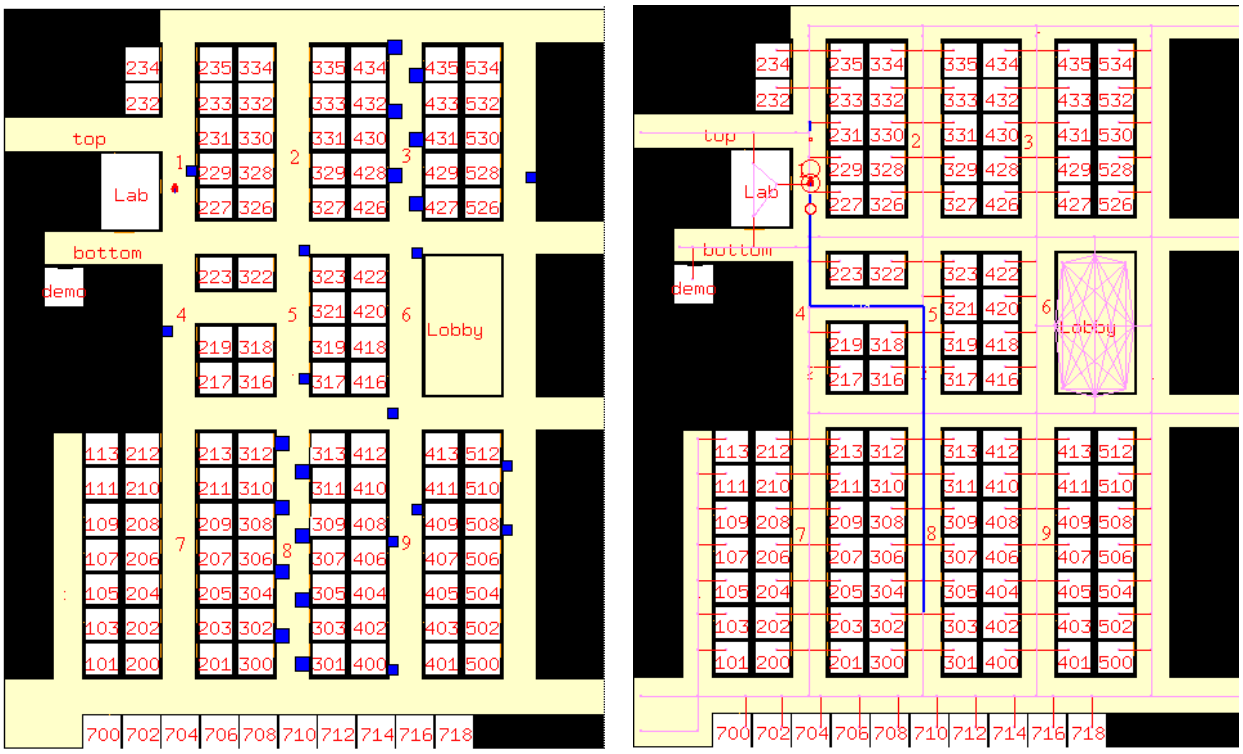


Figure 13: Boston Exposition World. (a) Simulator: operating environment (b) Path Planner: topological map.

dependent costs.

Figure 13 shows the world: an exposition of the variety one might see at a conference. Figure 13a shows the simulated world, complete with obstacles. Figure 13b shows the topological map used by the path planning module; this map displays everything the robot "knows" about its environment.

The simulator of course has limited capabilities for dynamism: currently doors can only be opened and closed at the whim of the user; obstacles are static. For our experimental stage, we needed the robot to be operating in a dynamic world. We add dynamism by running each experiment in a *variation* of the map shown in Figure 13. 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)[2]
  - OR one of the other days between (1 and 2am)
- corridor 8 always with obstacles
- remaining corridors with random obstacles (approximately 10 per map)

[2]We collected data overnight.

In each map, we ran a fixed path through the environment[3]: from corridor 1 to booth 303 to 411 to 327 to 435 to 210, collecting the execution trace.

This set of environments allowed us to test whether the system would successfully identify:

- permanent phenomena (corridors 2 and 4),
- temporary phenomena (random obstacles), and
- patterns in the environment (corridor 3).

The events table was generated as described in Section 3, and then processed as described in Section 4.

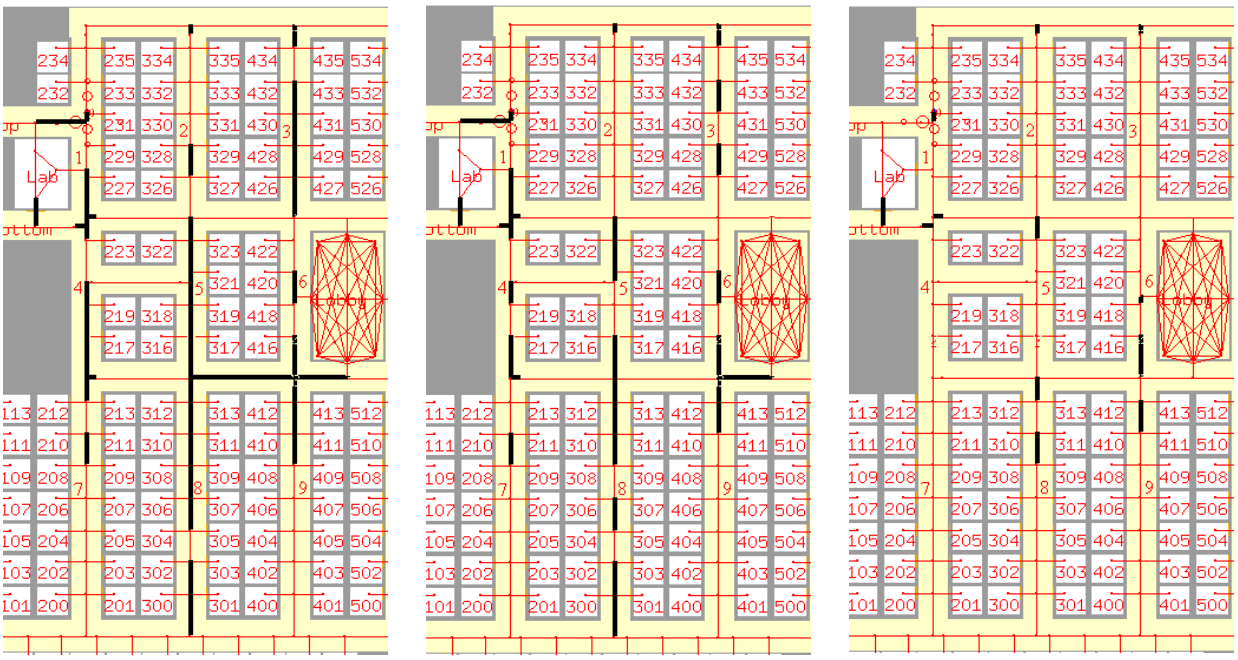
Over a period of 2 weeks, 651 execution traces were collected. Almost 306,500 arc traversals were identified, creating an events table of 15.3 MB. The 17 arcs with fewer than 25 traversal events were discarded as insignificant, leaving 100 arcs for which the system learned trees. (There are a total of 331 arcs in this environment, of which 116 are doors, and 32 are in the lobby.) Trees were generated with as few as 25 events, and as many as 15,340 events, averaging 3060. Generated trees had an average size of 6.9 nodes (3.45 leaf nodes).

Figure 14 shows learned costs for Wednesday at 01:05am. Figure 14a shows those arcs only *slightly* more expensive than default, demonstrating that the system has successfully identified areas where obstacles may appear: corridors 3 and 8, plus most of the arcs in which random obstacles may appear. Figure 14b shows those arcs *somewhat* more expensive than default... the arcs in corridors 3 and 8, plus the arcs near a turn. Finally, Figure 14c shows those arcs *much* more expensive than default. For comparison, Figure 15 shows learned costs for Tuesday at 09:45am. Note that corridor 3 is not considered expensive.

Figure 16 shows the cost, averaged over all the arcs in the corridor, as it changes throughout the day (a Wednesday). The system has correctly identified that corridor 3 is difficult to traverse between midnight and 3am, and during the rest of the day is close to default cost. Corridor 8, meanwhile, is *always* well above default, while corridor 2 is slightly below default.

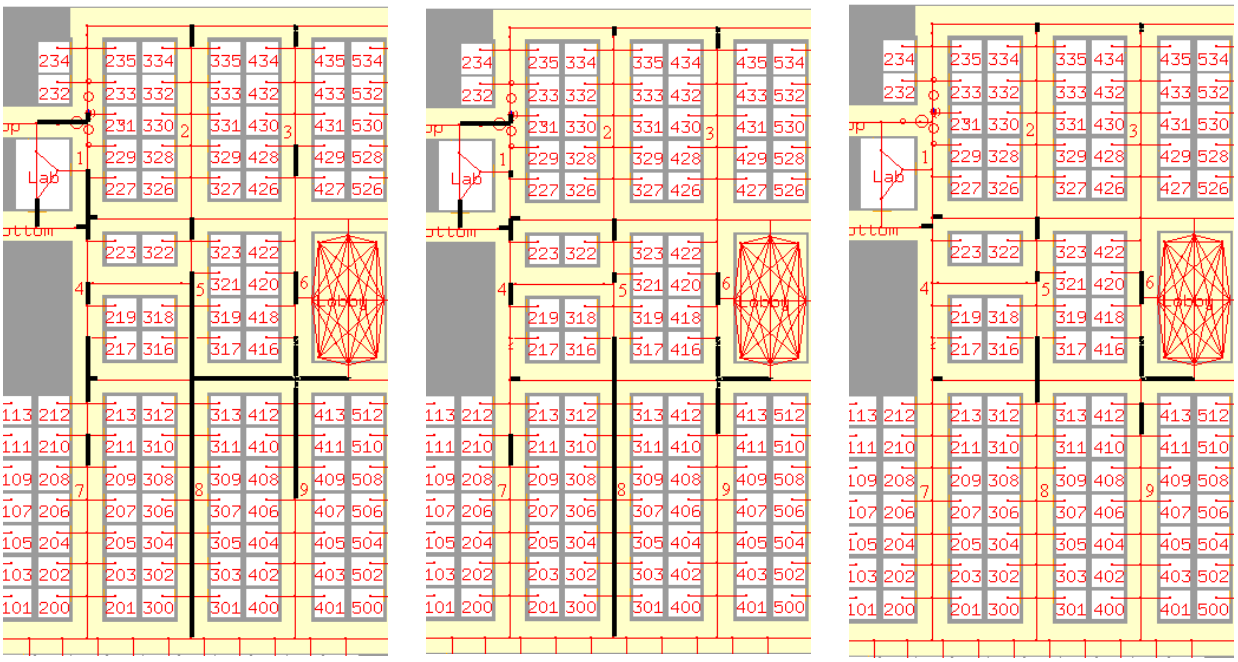
Figure 17 shows the effect of learning on the path planner. The goal is to have the system learn to avoid expensive arcs (those with many obstacles). Figure 17a shows the path normally generated. Figure 17b shows the path generated by the planner after learning; note that the expensive arcs have been avoided.

[3]It would have been trivial to randomly generate locations, but our analysis of requests sent to Xavier show that there is minimal variation in locations and routes.



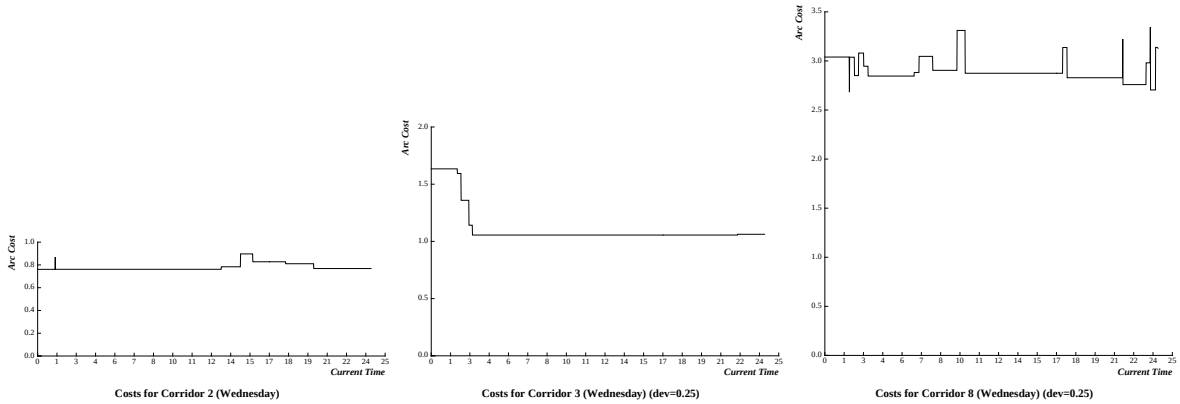
(a) Costs > 1.25 (b) Costs > 2.00 (c) Costs > 5.00

Figure 14: Arcs that the system has learned are more expensive than default, for situation: Wednesday, 01:05am. Note that corridor 3 and 8 are expensive, along with arcs containing random obstacles and difficult turns.



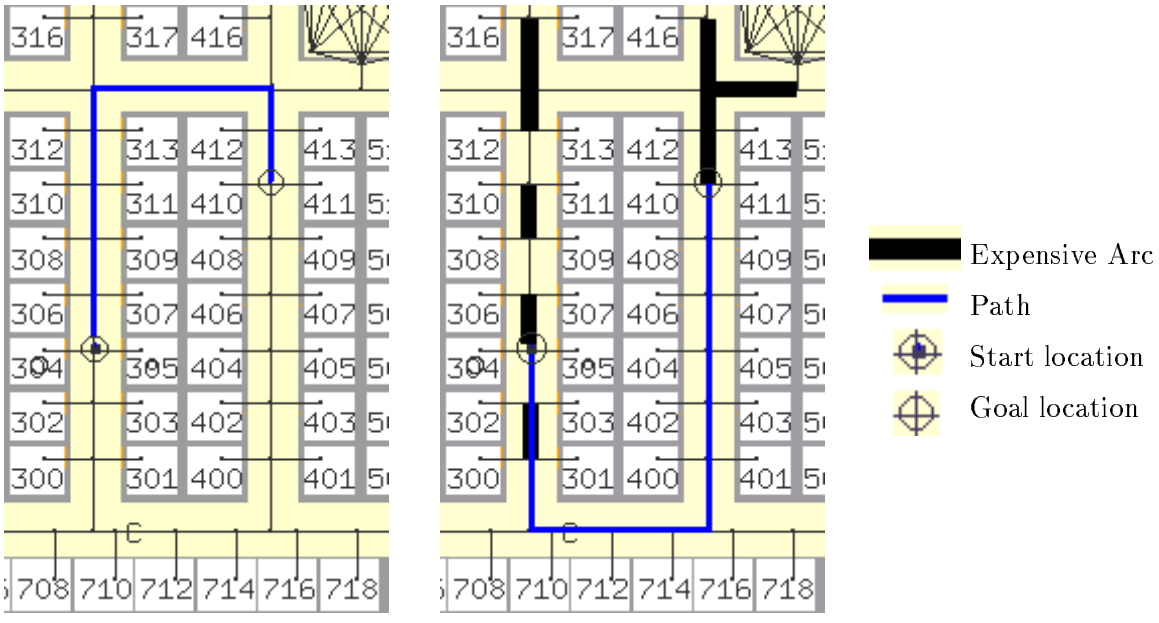
(a) Costs > 1.25 (b) Costs > 2.00 (c) Costs > 5.00

Figure 15: Arcs that the system has learned are more expensive than default, for situation: Tuesday, 09:45am. Note that corridor 3 is not considered expensive.



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

Figure 16: Corridor cost (average over all arcs in that corridor) for Wednesdays.



(a) (b)

Figure 17: (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).

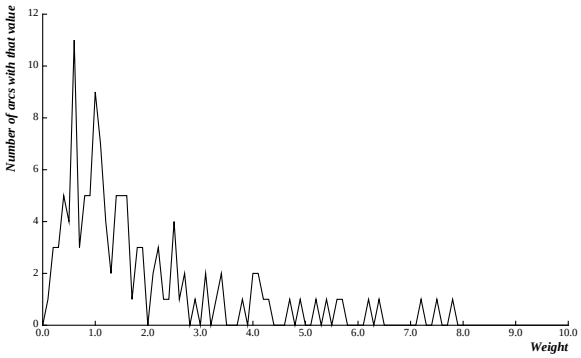


Figure 18: Arc cost frequency: most arcs are close to 1.0... the default value.

Figure 18 shows the training arc costs. This graph shows that the majority of arcs are fairly close to the default value of 1.0.

The data we have presented here demonstrates that our system 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 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. 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.

Over a period of 3 months, 17 robot execution traces were collected on the 5^{th} floor of our building (part of which was shown previously in Figure 1). These traces were run between 9:30 am and 3:40pm and varied from 10 minutes (0.35 MB) to 82 minutes (14 MB). Figure 19 shows the distribution and length of running times.

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

Figure 20 shows the average learned costs for all the arcs in the lobby. Note that the graph is shown for a particular Wednesday; date might be a relevant feature. Values differentiated by other features were averaged. The system correctly identified lunch-time as a more expensive time to

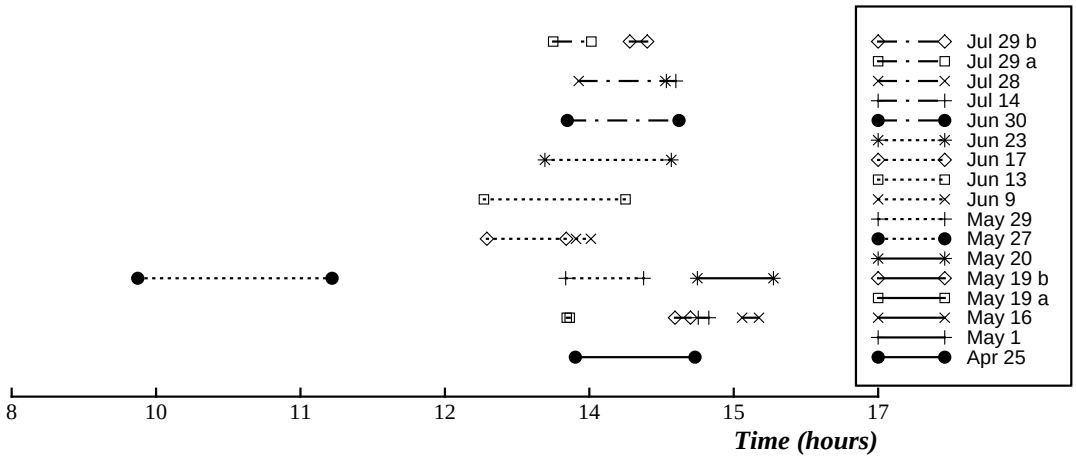


Figure 19: Distribution of robot running times.

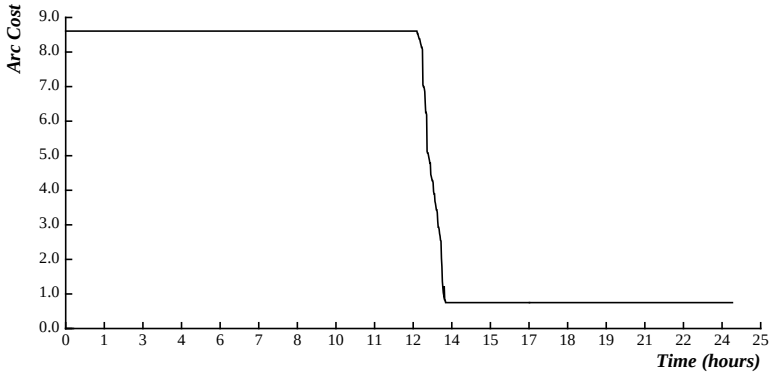


Figure 20: Costs for Wean Hall Lobby on a particular Wednesday.

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. We are currently collecting additional morning data so that the system can improve its model incrementally.

This data shows that our system is useful and effective, even in an environment where many of the default costs were hand tuned by the programmers. The added flexibility of situation-dependent arc costs increases the reliability and efficiency of the overall robot system.

7 Conclusion

We have presented a learning robotic system with the ability to learn from its own execution experience. Our system uses predictive features of the environment to create *situation-dependent costs* for the arcs in the topological map used by the path planner to create routes for the robot. These costs will reflect the patterns detected in the environment, and the planner will then know which areas of the world should be avoided (or exploited), and therefore find the most efficient path for each particular situation.

Our system processes the execution trace generated by the navigation module to extract events relevant for replanning. The execution trace contains a a massive, continuous stream of probabilistic, low-level data. Our system abstracts that information to reconstruct the route, and the arcs that the robot traversed through the environment. 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 planner uses the updated costs to create efficient, situation-dependent routes for the robot. The algorithm works incrementally, improving the situation-dependent rules after each run of the robot. The algorithm as used in the Xavier architecture is summarized in Table 3 (compare to Table 1, which outlines the algorithm for a general planner).

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

Our current work includes incorporating this algorithm into ROGUE, Xavier's task planner [Haigh & Veloso, 1997]. Knowledge at the task planner level is represented and manipulated as symbolic information. User requests are for example, *"deliver mail to the main office."* Learning rules that govern the applicability of actions and tasks will allow the task planner to select, reject

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

Table 3: Algorithm in Xavier Framework

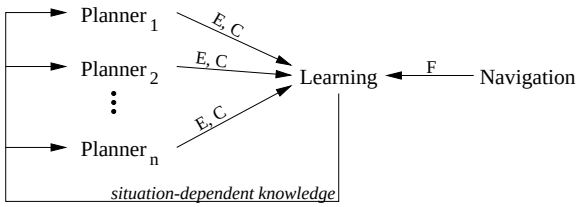


Figure 21: Pictoral representation of learning algorithm.

or delay tasks in the appropriate situation. Events useful for learning these rules include missed deadlines and time-outs (e.g. waiting at doors), while costs are defined by task importance, effort expended (travel plus wait time), and how much a deadline was missed by.

We view the algorithm as being relevant at all levels of a robot hierarchy, as shown in Figure 21. Every planner in the hierarchy can benefit from understanding the patterns of the environment that affect task achievability. This situation-dependent knowledge can be incorporated into the planning effort so that tasks can be achieved with greater reliability and efficiency. Situation-dependent features are an effective way to capture the changing nature of a real-world environment.

Acknowledgements

The authors would like to thank Sven Koenig, Reid Simmons and William Uther for feedback on this article. We would also like to thank the members of the Xavier and PRODIGY groups for feedback, comments and criticism on our research.

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

References

[Baroglio *et al.*, 1996] Baroglio, C., Giordana, A., Kaiser, M., Nuttin, M., & Piola, R. (1996). Learning controllers for industrial robots. *Machine Learning*, 23:221–249.

[Becker *et al.*, 1988] Becker, R. A., Chambers, J. M., & Wilks, A. R. (1988). *The New S Language.* (Pacific Grove, CA: Wadsworth & Brooks/Cole). Code available from `http://www.mathsoft.com/splus/`.

[Bennett & DeJong, 1996] Bennett, S. W. & DeJong, G. F. (1996). Real-world robotics: Learning to plan for robust execution. *Machine Learning*, 23:121–161.

[Breiman *et al.*, 1984] Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and Regression Trees.* (Pacific Grove, CA: Wadsworth & Brooks/Cole).

[Cassandra *et al.*, 1994] Cassandra, A., Kaelbling, L., & Littman, M. (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] Chambers, J. M. & Hastie, T. (1992). *Statistical models in S.* (Pacific Grove, CA: Wadsworth & Brooks/Cole).

[Goodwin, 1996] Goodwin, R. (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.

[Haigh *et al.*, 1997] Haigh, K. Z., Shewchuk, J. R., & Veloso, M. M. (1997). Exploiting domain geometry in analogical route planning. *Journal of Experimental and Theoretical Artificial Intelligence.* In Press. Available at `http://www.cs.cmu.edu/~khaigh/papers.html`.

[Haigh & Veloso, 1997] Haigh, K. Z. & Veloso, M. M. (1997). High-level planning and low-level execution: Towards a complete robotic agent. In Johnson, W. L., editor, *Proceedings of the First International Conference on Autonomous Agents*, pages 363–370, Marina del Rey, CA. (New York, NY: ACM Press).

[Klingspor *et al.*, 1996] Klingspor, V., Morik, K. J., & Rieger, A. D. (1996). Learning concepts from sensor data of a mobile robot. *Machine Learning*, 23:305–332.

[Koenig & Simmons, 1996] Koenig, S. & Simmons, R. G. (1996). Passive distance learning for robot navigation. In Saitta, L., editor, *Machine Learning: Proceedings of the Thirteenth International Conference (ICML96)*, pages 266–274, Bari, Italy. (San Mateo, CA: Morgan Kaufmann).

[Kortenkamp & Weymouth, 1994] Kortenkamp, D. & Weymouth, T. (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] Lindner, J., Murphy, R. R., & Nitz, E. (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] Lovejoy, W. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, 28(1):47–65.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning.* (New York, NY: McGraw Hill).

[Mitchell *et al.*, 1994] Mitchell, T. M., Caruana, R., Freitag, D., McDermott, J. P., & Zabowski, D. (1994). Experience with a learning personal assistant. *CACM*, 37(7):80–91.

[O'Sullivan *et al.*, 1997] O'Sullivan, J., Haigh, K. Z., & Armstrong, G. D. (1997). *Xavier.* Carnegie Mellon University, Pittsburgh, PA. Manual, Version 0.3, unpublished internal report. Available via `http://www.cs.cmu.edu/~Xavier/`.

[Pearson, 1996] Pearson, D. J. (1996). *Learning Procedural Planning Knowledge in Complex Environments.* PhD thesis, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan.

[Pomerleau, 1993] Pomerleau, D. A. (1993). *Neural network perception for mobile robot guidance.* (Dordrecht, Netherlands: Kluwer Academic).

[Rabiner & Juang, 1986] Rabiner, L. R. & Juang, B. H. (1986). An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 4–16.

[Shen, 1994] Shen, W.-M. (1994). *Autonomous Learning from the Environment.* (New York, NY: Computer Science Press).

[Simmons, 1994] Simmons, R. (1994). Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43.

[Simmons *et al.*, 1997] Simmons, R., Goodwin, R., Haigh, K. Z., Koenig, S., & O'Sullivan, J. (1997). A layered architecture for office delivery robots. In Johnson, W. L., 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] Simmons, R. & Koenig, S. (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] Tan, M. (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] Thrun, S. (1996). A Bayesian approach to landmark discovery and active perception for mobile robot navigation. Technical Report CMU-CS-96-122, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.