Planning, Execution and Learning in a Robotic Agent

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

This paper presents the complete integrated planning, executing and learning robotic agent Rogue. We describe Rogue's task planner that interleaves high-level task planning with real world robot execution. It supports multiple, asynchronous goals, suspends and interrupts tasks, and monitors and compensates for failure. We present a general approach for learning *situation-dependent rules* from execution, which correlates environmental features with learning opportunities, thereby detecting patterns and allowing planners to predict and avoid failures. We present two implementations of the general learning approach, in the robot's path planner, and in the task planner. We present empirical data to show the effectiveness of Rogue's novel learning approach.

Introduction

In complex, dynamic domains, a robot's knowledge about the environment will rarely be complete and correct. Since Shakey the robot [12], researchers have been trying to build autonomous robots that are capable of planning and executing high-level tasks, as well as learning from the analysis of execution experience.

This paper presents our work extending the high-level reasoning capabilities of a real robot in two ways:

- by adding a high-level task planner that interleaves planning with execution, and
- by adding the ability to learn from real execution to improve planning.

We have developed Rogue [5; 6; 7] which forms the task planning and learning modules for a real mobile robot, Xavier (see Figure 1). One of the goals of the project is to have the robot move autonomously in an office building reliably performing office tasks such as picking up and delivering mail and computer printouts, picking up and returning library books, and carrying recycling cans to the appropriate containers.

Xavier is a mobile robot being developed at Carnegie Mellon University [13; 18]. 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. Much of the software can be classified into five layers: Obstacle Avoidance, Navigation, Path Planning, Task Planning (provided by Rogue), and the User Interface. The underlying architecture is described in more detail by Simmons *et al.* [18].

Figure 1: Xavier the Robot.

Rogue provides a setup where users can post tasks for which the planner generates appropriate plans, delivers them to the robot, monitors their execution, and learns from feedback about execution performance.

Rogue's task planner is built upon the PRODIGY4.0 planning and learning system [6; 23]. The task planner generates and executes plans for multiple interacting goals which arrive asynchronously and whose task structure is not known *a priori*. The task planner interleaves tasks and reasons about task priority and task compatibility. Rogue interleaves planning and execution to detect successes or failures and responds to them. The task planner controls the execution of a real robot to accomplish tasks in the real world. Rogue effectively enables the communication between Xavier, PRODIGY

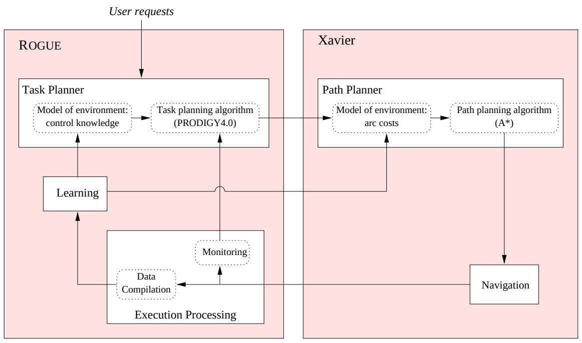


Figure 2: ROGUE architecture.

and the user. The planning and execution capabilities of ROGUE form the foundation for a complete, learning, autonomous agent.

ROGUE's planner-independent learning approach processes the robot's execution data with the goal of improving planning [5; 7]. It applies to two planners: Xavier's path planner, and the task planner. ROGUE learns *situation-dependent rules* that affect the planners' decisions. Our approach relies on direct examination of the robot's execution traces to identify situations in which the planner's behaviour needs to change. ROGUE then correlates features of the domain with the learning opportunities, and creates situation-dependent rules for the planners. The planners then use these rules to select between alternatives to create better plans.

ROGUE's overall architecture is shown in Figure 2. ROGUE exploits Xavier's reliable lower-level behaviours, including path planning, navigation, speech and vision. ROGUE provides Xavier with a high-level task planning component, as well as learning abilities that extract information from low-level execution data to improve higher-level planning. The task planner receives task requests from users, and then determines good interleavings of multiple tasks. The path planner creates paths between locations which the navigation module then executes. ROGUE processes the execution trace of the navigation module to extract learning opportunities, and then feeds that information to the learning algorithm. ROGUE then processes the resulting situation-dependent knowledge into rules for use by the planners.

Task Planning

The office delivery domain involves multiple users and multiple tasks. ROGUE's task planner has the ability

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

The task planner is based on PRODIGY4.0 [23], a domain-independent nonlinear state-space planner that uses means-ends analysis and backward chaining to reason about multiple goals and multiple alternative operators to achieve the goals. It has been extended to support real-world execution of its symbolic actions [20].

Task requests arrive from users at any time. ROGUE incorporates the information into PRODIGY4.0's state description, and then PRODIGY4.0 extends the current plan to incorporate the new task.

The planning cycle involves several decision points, including which goal to select from the set of pending goals, and which applicable action to execute. Dynamic goal selection from the set of pending goals enables the planner to interleave plans, exploiting common subgoals and addressing issues of resource contention.

Search control rules can reduce the number of choices at each decision point by pruning the search space or suggesting a course of action while expanding the plan. ROGUE uses search control rules that reason about task *priority* and task *compatibility*. ROGUE can suspend and reactivate lower-priority tasks, as well as recognize opportunities for parallel execution. ROGUE can thus interleave the execution of multiple compatible tasks to improve overall execution efficiency.

ROGUE interleaves planning with execution. Each time PRODIGY4.0 generates an executable plan step, ROGUE maps the action into a sequence of navigation and other commands which are sent to the Xavier module designed to handle them. ROGUE then monitors the outcome of the action to determine its success or failure. ROGUE can detect execution failures, side-effects (including helpful ones), and opportunities. For example, it can prune alternative outcomes of a non-deterministic action, notice external events (e.g. doors opening/closing), notice limited resources (e.g. battery level), and notice failures.

For example, each time a navigation command is issued, the task planner monitors its outcome. Since the navigation module may occasionally get confused and report a success even in a failure situation, the planner always verifies the location with a secondary test (vision or human interaction). If ROGUE detects that the robot is *not* at the correct goal location, it updates PRODIGY4.0's state knowledge with the correct information, and forces replanning.

ROGUE controls the execution of a real robot to accomplish tasks in the real world. The complete interleaved planning and execution cycle is shown in Table 1. The task planner is described in more detail elsewhere [6].

Learning

Learning has been applied to robotics problems in a variety of manners. Common applications include map learning and localization (e.g. [9; 10; 22]), or learning operational parameters for better actuator control (e.g. [1; 2; 15]). Instead of improving low-level *actuator* con-

In Parallel:

1. Rogue receives a task request from a user, and adds the information to prodigy4.0's state.
2. Rogue requests a plan from prodigy4.0.

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

   (a) Using up-to-date state information, prodigy4.0 generates a plan step, considering task priority, task compatibility and execution efficiency.
   (b) Rogue translates and sends the planning step to Xavier.
   (c) Rogue monitors execution and identifies goal status; in case of failure, it modifies prodigy4.0's state information.

3. Rogue monitors the environment for exogenous events; when they occur, Rogue updates prodigy4.0's state information.

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

trol, 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. [8; 14]), or learning costs and applicability of actions (e.g. [11; 17; 21]). Our work falls into the latter category.

In some situations, it is enough to learn that a particular action has a certain average probability or cost. However, actions may have different costs under different conditions. Instead of learning a global description, we would rather that the system learn the pattern by which these situations can be identified. We introduce an approach to learn the correlation between features of the environment and the situations, thereby creating *situation-dependent rules*.

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.

Our approach relies on examining the execution traces of the robot to identify situations in which the planner's behaviour needs to change. It 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 learning opportunities in the environment

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

Table 2: General process for learning situation-dependent rules.

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. The learner 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 calculates a cost $c \in \mathcal{C}$. We call this mapping a *situation-dependent rule*.

Once the rule has been created, Rogue gives the information back to the planners so that they will avoid re-encountering the problem events. 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.

We demonstrate our learning approach in two planners: Xavier's path planner and Rogue's task planner. While the details of extracting and evaluating events from execution are domain-specific, the general approach is planner- and domain- independent.

Learning for the Path Planner

In this section, we present the learning algorithm as it applies to Xavier's path planner[1], where our concern is to improve the reliability and efficiency of selected paths. The path planner uses a modified A* algorithm on a topological map that has additional metric information [4]. The map is a graph with nodes and arcs representing rooms, corridors, doors and lobbies.

Rogue demonstrates the ability to *learn situation-dependent costs* for the path planner's arcs. Learning appropriate arc-cost functions will allow the path planner to avoid troublesome areas of the environment when appropriate. Therefore we define events, $\mathcal{E}$, for this planner as arc traversals and costs, $\mathcal{C}$, as travel time. Features, $\mathcal{F}$, include both robot data and high-level features. Features are hand-picked by the designers, and are extracted from the robot, the environment, and the task.

Rogue extracts arc traversal events and environmental features from the massive, continuous, probabilistic

[1] More information about learning for the path planner can be found elsewhere [7].

execution traces, and then evaluates the events according to the cost function. The learning algorithm then creates the situation-dependent arc costs.

The execution traces are provided by the robot's navigation module. Navigation is done using Partially Observable Markov Decision Process models [19]. The execution trace includes observed features of the environment as well as the probability distribution over the Markov states at each time step.

Event Identification. Identifying the planner's arc traversal events from this execution trace is challenging because they contain a massive, continuous stream of uncertain 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 execution trace in particular does *not* contain arc traversals. We therefore need to extract the traversed arc sequence from the Markov state distributions of the navigation module. The first step in this process is to calculate likely sequences of Markov states. We use Viterbi's algorithm [16] to calculate the most likely transitions from state to state. Then, from high probability states throughout the trace, we reconstruct sequences that are likely to reflect the robot's actual trajectory. The second step of this process is to reverse-engineer the Markov state sequences to extract the sequence of planner's arcs.

Once the arc sequences have been identified, Rogue calculates cost estimates for the arcs: $\mathcal{C}(\varepsilon \in \mathcal{E}) = vt/l$, where v is the desired velocity for traversing the arc, t is the actual time taken to traverse the arc, and l is the believed length of the arc.

The data is stored in a matrix along with the cost evaluation and the environmental features observed when the arc traversal event occurred (Table 3). Beyond the list of features shown, we also record sonar values and the highest probability Markov states.

The events matrix is grown incrementally; most recent data is appended at the bottom. By using incremental learning, Rogue can notice and respond to changes on a continuous basis.

Learning. We selected *regression trees* [3] as our learning mechanism because they can handle continuous values, and form disjunctive hypotheses. A regression tree is created for each event, in which features are splits and costs are learned values. Splits are selected to maximize the reduction in deviance of the node, and we then prune the tree using 10-fold random cross validation. The regression trees, one for each arc, are then formatted into situation-dependent rules.

Updating the Planner. The rules are then loaded into an update module. Each time the path planner generates a path, it requests the new arc costs from the update module. These costs are generated by matching the current situation against the learned rule for each arc. Using its A* algorithm, the planner selects the path with the best expected travel time, according to the updated situation-dependent costs.

ArcNo	Weight	CT	Speed	PA	Goal	Year	MM	DD	DoW
233	0.348354	38108	34.998	234	90	1997	06	30	1
192	0.777130	37870	33.461	191	90	1997	06	30	1
196	3.762347	37816	34.998	195	284	1997	06	30	1
175	0.336681	37715	34.998	174	405	1997	06	30	1
168	1.002090	60151	34.998	167	31	1997	07	07	1
134	16.549173	61208	34.998	234	262	1997	07	09	3
238	0.640905	54	34.998	130	379	1997	07	10	4
165	1.472222	8805	34.998	164	379	1997	07	17	4
196	5.823351	3983	34.608	126	253	1997	07	18	5
194	1.878457	85430	34.998	193	262	1997	07	18	5

Table 3: Events Matrix; each feature-value vector (row of table) corresponds to an arc traversal event $\varepsilon \in \mathcal{E}$. *Weight* is arc traversal weight, $\mathcal{C}(\varepsilon)$. The remaining columns contain environmental features, $\mathcal{F}$, valid at time of the traversal: *CT* is CurrentTime (seconds since midnight), *Speed* is velocity, in cm/sec, *PA* is the previous arc traversed, *Goal* is the Markov state at the goal location, *Year, MM, DD, DoW* is the date and day-of-week of the traversal.

Path Planner Experiments (Simulation) We built a simulated world for testing the system in a controlled environment. Figure 3 shows the *Exposition World*: an exposition of the variety one might see at a conference. Rooms are numbered; corridors are labelled for discussion purposes only. Figure 3 shows the simulated world, complete with a set of obstacles. The path planner does not know about the obstacles; every arc in the topological description has the same default

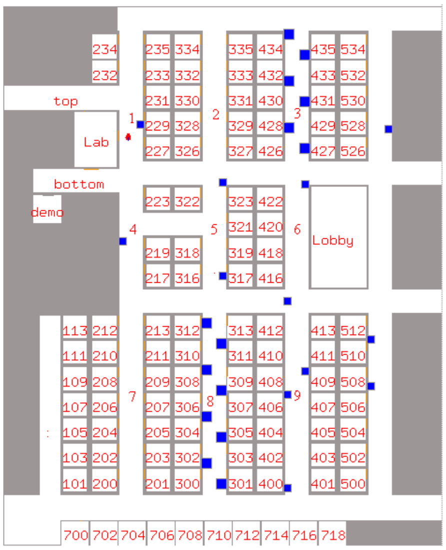
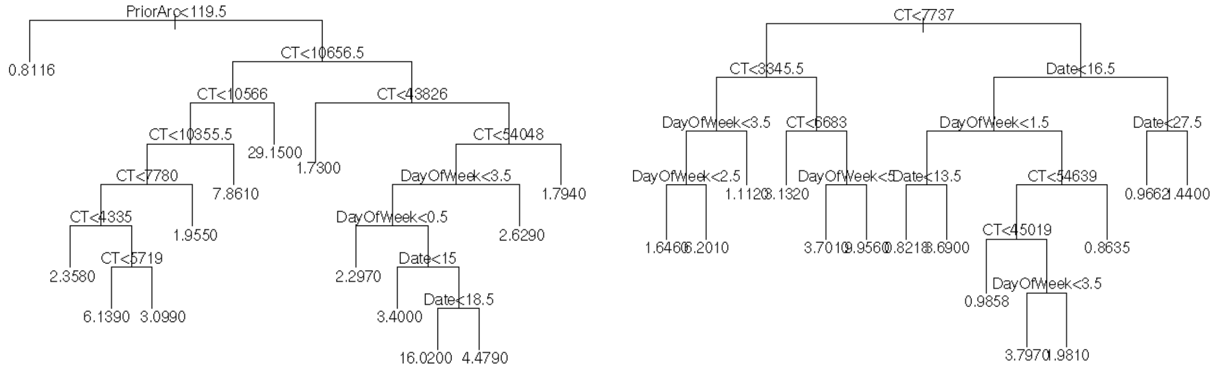


Figure 3: Exposition World. Simulator: operating environment. Obstacles marked with dark boxes. The path planner does not know about the obstacles.



(a) Arc 240 (b) Arc 244

Figure 4: Learned trees for two of the arcs in corridor 3. Leaves show the cost of traversing the arc.

cost of 1.0.

The position of the obstacles in the simulated world changes according to the following schedule:

- corridor 2 is always clear,
- corridor 3 has obstacles on:
  - 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 has obstacles,
- remaining corridors have random obstacles (approximately 10 per map).

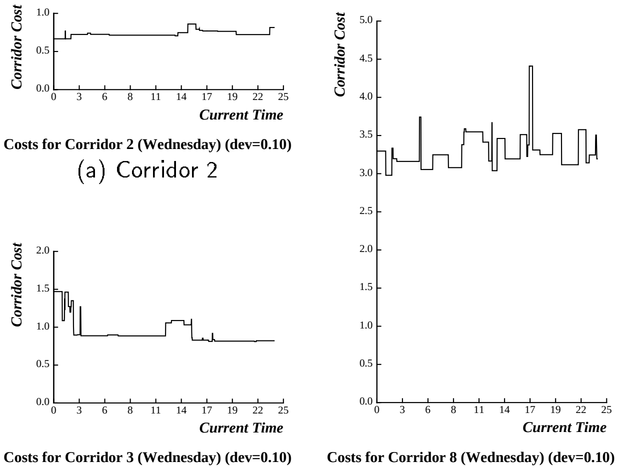
This set of environments allowed us to test whether Rogue would successfully learn:

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

Over a period of 2 weeks, 651 execution traces were collected. Almost 306,500 arc traversals were identified, creating an events matrix 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.) Rules were generated with as few as 25 events, and as many as 15,340 events, averaging 3060.

Figure 4 shows two of the learned trees. Both arcs shown are from corridor 3. Both *DayOfWeek* and *CT* are prevalent in all the trees for that corridor. (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 5 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



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

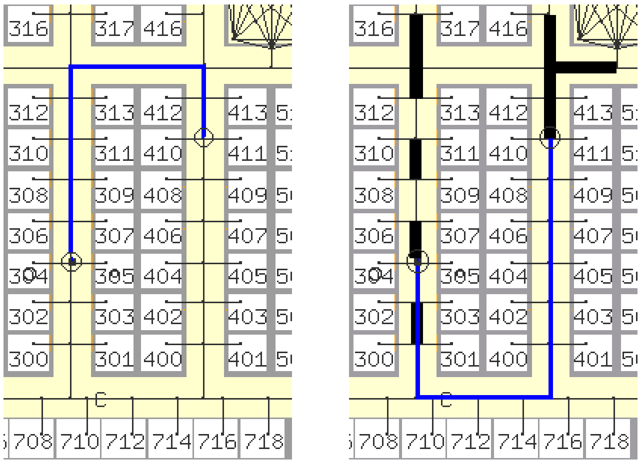
Figure 5: Corridor cost (average over all arcs in that corridor) for Wednesdays.

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.

Effect on Planner. Figure 6 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 6a shows the default path generated by the planner. Figure 6b shows the path generated after learning; note that the expensive arcs, marked with thick segments, have been avoided.

In general, paths generated using the learned costs are 20% faster than default paths evaluated with the learned costs.

The data we have illustrated 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 fea-



(a) Default Path (b) New Path

Figure 6: (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).

tures and events to arc traversal weights. The planner then correctly predicts the expensive arcs and creates plans that avoid difficult areas of the environment.

Path Planner Experiments (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`. These data have 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 five months, we collected 59 robot execution traces. These traces were run between 9:30 and 19:00 and varied from 10 minutes to 82 minutes in length. The majority of the traces were collected between noon and 4pm.

ROGUE recorded 72,516 arc traversal events. Trees were learned for 115 arcs from an average of 631 traversal events per arc (min 38, max 1229). Data from nine arcs was discarded because they had fewer than 25 traversal events.

Figure 7 shows the average learned costs for all the arcs in the lobby on Wednesdays. Values differentiated by other features were averaged[2]. Below the average cost graph, a histogram shows how much training data was collected for each time step during the day.

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

[2]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.

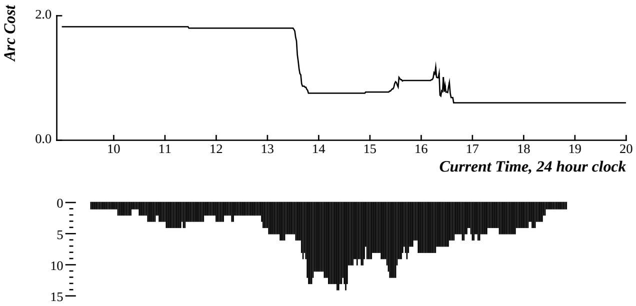


Figure 7: Costs for Wean Hall Lobby on Wednesdays. Graph generated 31 October 1997. The histogram below the graph indicates volume of training data, in terms of number of execution traces; most data was collected between 1pm and 6pm.

ROGUE 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. To our surprise, the graph shows a slightly higher cost during the late afternoon; 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.

These data show that ROGUE learns useful and effective information, even in an environment where many of the default costs were tediously hand tuned by the programmers. The added flexibility of situation-dependent rules to determine arc costs increases the overall reliability and efficiency of the robot.

Learning for the Task Planner

In addition to improving routes, situation-dependent learning can also apply to task planning. In the task planner, ROGUE creates situation-dependent search control rules that guide the planner towards better decisions. It collects execution data to record the success or failure of events for which it needs more information. The learner then correlates situational features to events to create PRODIGY4.0 search control rules.

Learning rules that govern the applicability of actions and tasks will allow the task planner to select, reject or delay tasks in the appropriate situation. Events, $\mathcal{E}$, useful for learning include missed deadlines and timeouts (e.g. waiting at doors), while costs, $\mathcal{C}$, can be defined by task importance, effort expended (travel plus wait time), and how much a deadline was missed by. Features, $\mathcal{F}$, remain the same as for the path planner.

Event Identification. The goal of learning control knowledge for the planner is to have the system learn when tasks can and cannot be easily achieved. Events, $\mathcal{E}$, for this planner are successes and failures related to task achievement. For example, missing or meeting a deadline, or acquiring or not acquiring an object. Careful analysis of the domain model yields these learning opportunities.

Although we could use a complex cost function $\mathcal{C}$ to evaluate task events, we instead simplify the learning task by assigning successes a cost of zero and failures a cost of one.

The event is stored in an events matrix along with the cost evaluation and the environmental features observed when the event occurred. We include task-specific information, sonar values, high probability Markov states along with the features listed in Table 3.

Learning. ROGUE uses the same regression tree analysis for the task planning data as it does for the path planning data. The common learning framework shared for different planners is one of the contributions of this research.

Updating the Planner. Once the set of regression trees have been created (one for each type of event), they to be translated into PRODIGY4.0 search control rules. ROGUE assigns *select* rules to situations with a cost near zero, and *reject* rules to situations with a cost near one. *Prefer* rules are used for more ambiguous situations. PRODIGY4.0 will then use these rules to guide its decisions, selecting, rejecting and preferring goals and actions as required.

Task Planner Experiments This experiment was designed to test ROGUE's ability to identify and use high-level features to create situation-dependent control rules. The goal was to have the system identify times for which tasks could not be completed, and then create goal selection rules of the form "reject task until..."

For training data, we generated two maps for the simulator. Between 10:00 and 19:59, all doors in the map were open. At other times, all doors were closed. (When a door is closed the task is not completable because the human is not available.) We used a single route: from the starting location of 5310, go to room 5312 then to room 5316. The user remained constant and tasks were selected randomly from a uniform distribution.

Table 4 shows a sample tree learned for this domain. The tree indicates that between 10:00 and 20:00, tasks are more likely to succeed than at night (recall that CT is *CurrentTime*, in seconds since midnight). A control rule is created at each leaf node; it corresponds to the path from the root node to the leaf. Table 5 shows the four control rules created for it.

```
node), split, number of examples, deviance, value

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

Table 4: A sample tree.

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

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

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

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

Table 5: Learned PRODIGY4.0 control rules for the tree in Table 4.

PRODIGY4.0 uses the reject control rules (0 and 3) to reject tasks before 09:58:09 and after 19:55:59. Rule 1 is used to select tasks between those times involving rooms "less than" 5314...namely room 5312. The prefer-reject control rule (rule 2) is used to prefer tasks *other* than those involving room 5316.

Additional experiments are presented elsewhere [5]. They demonstrate that, by learning search control rules from execution experience, ROGUE helps the task planner predict and avoid failures when executing. In this way, the overall system becomes more efficient and effective at accomplishing tasks.

The primary purpose of these experiments was to validate the hypothesis that our general approach for learning situation-dependent rules was planner-independent. We have successfully demonstrated that the basic mechanisms can be transferred, and are both applicable and effective for two planners with very different data representations and task requirements.

Summary

In this paper, we have briefly outlined ROGUE, an integrated planning, executing and learning robot agent.

We described the task planner, which can handle multiple, asynchronous requests from users, and creates plans that require reasoning about task priority and task compatibility.

We presented a robot with the ability to learn from its own execution experience. We outlined our learning approach, which extracts *events*, $\mathcal{E}$, from the robot's execution traces, and evaluates them with a *cost function*, $\mathcal{C}$. It uses regression trees to correlate the events to environmental *features*, $\mathcal{F}$, in a mapping $\mathcal{F} \times \mathcal{E} \to \mathcal{C}$.

We demonstrated our planner-independent learning framework in two planners, a path planner and a task planner. ROGUE demonstrates the ability to learn *situation-dependent rules* that allow the planners to predict and avoid failures at execution time. ROGUE provides the path planner information about which areas of the world to avoid (or exploit), and the planner can then find the most efficient path for each particular situation. ROGUE provides the task planner information about when tasks can or cannot be successfully achieved, and the planner can then create plans with greater likelihood of success.

Through our extensive experiments (simulated and on the real robot), briefly outlined here, we have demonstrated the effectiveness and utility of our learning approach.

References

[1] Baroglio, C., Giordana, A., Kaiser, M., Nuttin, M., and Piola, R. Learning controllers for industrial robots. *Machine Learning*, 23:221–249, 1996.

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

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

[4] Goodwin, R. *Meta-Level Control for Decision-Theoretic Planners.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996.

[5] Haigh, K. Z. *Learning Situation-Dependent Planning Knowledge from Uncertain Robot Execution Data.* PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1998.

[6] Haigh, K. Z. and Veloso, M. M. Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, 1997. In press.

[7] Haigh, K. Z. and Veloso, M. M. Learning situation-dependent costs: Improving planning from probabilistic robot execution. In *Proceedings of the Second International Conference on Autonomous Agents*, 1998. (Menlo Park, CA: AAAI Press). In Press.

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

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

[10] Kortenkamp, D. and Weymouth, T. 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, 1994. (Menlo Park, CA: AAAI Press).

[11] Lindner, J., Murphy, R. R., and Nitz, E. 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), 1994.

[12] Nilsson, N. J. Shakey the robot. Technical Report 323, AI Center, SRI International, Menlo Park, CA, 1984.

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

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

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

[16] Rabiner, L. R. and Juang, B. H. An introduction to hidden Markov models. *IEEE ASSP Magazine*, 6(3):4–16, January 1986.

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

[18] Simmons, R., Goodwin, R., Haigh, K. Z., Koenig, S., and O'Sullivan, J. A layered architecture for office delivery robots. In *Proceedings of the First International Conference on Autonomous Agents*, pages 245–252, 1997. (New York, NY: ACM Press).

[19] Simmons, R. and Koenig, S. Probabilistic robot navigation in partially observable environments. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1080–1087, 1995. (San Mateo, CA: Morgan Kaufmann).

[20] Stone, P. and Veloso, M. M. User-guided interleaving of planning and execution. In *New Directions in AI Planning*, pages 103–112. (Amsterdam, Netherlands: IOS Press), 1996.

[21] Tan, M. *Cost-sensitive robot learning.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[22] Thrun, S. 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, 1996.

[23] Veloso, M. M., Carbonell, J., Pérez, M. A., Borrajo, D., Fink, E., and Blythe, J. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, January 1995.