To appear in the 1999 AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information

Learning Situation-Dependent Rules: Improving Task Planning for an Incompletely Modelled Domain

Karen Zita Haigh*
khaigh@htc.honeywell.com
http://www.cs.cmu.edu/~khaigh

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

Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

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

Abstract

Most real world environments are hard to model completely and correctly, especially to model the dynamics of the environment. In this paper we present our work to improve a domain model through learning from execution, thereby improving a task planner's performance. Our system collects execution traces from the robot, and automatically extracts relevant information to improve the domain model. We introduce the concept of *situation-dependent rules*, where situational features are used to identify the conditions that affect action achievability. The system then converts this execution knowledge into a symbolic representation that the planner can use to generate plans appropriate for given situations.

Introduction

Most real world environments are hard to model completely and correctly. Regardless of the level of detail and the care taken to model the domain, systems are bound to encounter uncertainty and incomplete information. Advanced search techniques may mitigate some of the problems, but until the system learns to adapt to these situations and to absorb new information about the domain, it will never improve its performance.

In this paper, we present a robotic system, Rogue, that creates and executes plans for multiple, asynchronous, interacting tasks in an incompletely modelled, uncertain, real-world domain. Rogue learns from its execution experience by adding detailed estimates of action costs and probabilities in order to improve the domain model and its overall performance.

We take the novel approach that 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. The system needs 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. Creating a pre-programmed model of these dynamics would be not only time-consuming, but very likely would not capture all relevant information.

The situation-dependent learning approach relies on examining the robot's execution traces to identify situations in which the planner's behaviour needs to change. The planner-independent approach relies on automatically extracting learning opportunities from the execution traces, evaluating them according to a pre-defined cost function. Finally, it correlates the learning opportunities with features of the environment. We call this mapping a *situation-dependent rule*. In the future, when the current situation matches the features of a given rule, the planner will predict the failure and avoid the corresponding action.

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 and features.
3. Identify learning opportunities in the execution trace.
4. Learn mapping between features, learning opportunities and costs.
5. Create rules to update the planner.

Table 1: General approach for learning situation-dependent rules.

Architecture & Domain

Rogue forms the task planning and learning layers for a real mobile robot, Xavier [15]. It is built on an RWI B24 base and includes bump sensors, a laser range finder, sonars, a color camera and a speech board. Rogue

*This paper represents work done while at CMU.

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

Table 2: A high-level view of two sample task planner rules.

operates in an office delivery domain, where tasks include picking up printouts, picking up and returning library books, and delivering mail and packages within the building. User requests are, for example, *"Pickup a package from my office and take it to the mailroom before 4pm today."*

The domain is dynamic, uncertain and incompletely modelled. For example, different people have different working hours, may briefly step out of their offices, or move. If ROGUE has to wait for substantially long times before acquiring or delivering objects, ROGUE's efficiency is severely compromised.

We would like ROGUE to learn to predict when people will be away from their offices. This information can then be used to avoid the task under those conditions. For example, a particular user might work 11am to 8pm, while another works 8am to 5pm. Rules guiding the task planner, like the one shown in Table 2, would help the task planner avoid tasks at appropriate times.

The whole architecture is shown in Figure 1. In this paper, we briefly introduce the *task planner* and describe the learning as applied to it. Haigh & Veloso [8] describe the implementation of the learning approach in the *path planner*. Haigh [6] describes the *complete system* in more detail.

Task Planning

ROGUE's task planner [7] is based on the PRODIGY4.0 planning and learning system [17]. The task planner generates and executes plans for multiple interacting goals, which arrive asynchronously and whose structure is not known *a priori*. ROGUE incorporates the information into PRODIGY4.0's state description, and then PRODIGY4.0 extends the current plan to incorporate the new task.

ROGUE uses *search control rules* to guide the planner in its decisions. These rules reason about task priority and task compatibility, and are used to create good plans. 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.

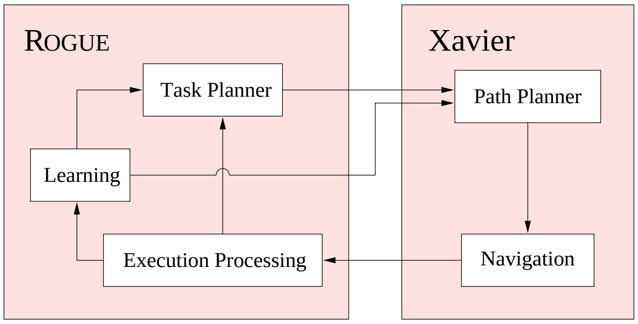


Figure 1: ROGUE architecture.

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

ROGUE controls the execution of the plan, effectively interleaving planning and 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.

The complete interleaved planning and execution cycle is shown in Table 3. 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.

Situation-dependent learning allows ROGUE to improve its domain model and thereby its planning and execution efficiency.

Data Processing

ROGUE extracts relevant learning data from the robot's execution traces. This data includes situational features and learning opportunities. ROGUE then evaluates the learning opportunities, and then correlates the resulting costs or probabilities with the situational features.

Features

Features of the environment are used to discriminate between different learning events. Features are defined by the robot architecture and the environment. *High-level features* of the environment can be detected well before actions need to be executed. High-level features available in Xavier include speed, time of day, day of week, goals, and the desired route.

Execution-level features are environmental features that can only be detected just before the action is executed. They include current location and sensor observations (e.g. walls, openings, people). For example, images with many people may indicate difficult navigation.

Learning Opportunities

The goal of learning control knowledge for the planner is to have ROGUE learn when tasks can and cannot be easily achieved. Therefore learning opportunities 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. Most opportunities correspond directly to operator applications. Progress towards the goal, such as acquiring a necessary object, is recorded as a successful event. Lack of progress is recorded when the corresponding action fails. Each time the task planner records the event in an execution trace, it requests current situational features from the execution module.

Costs

Once learning opportunities have been identified from the execution trace, updated costs or probabilities need to be calculated. These costs become the situation-dependent value predicted by the learning algorithm.

For ROGUE's task planner, the cost function assigns successful outcomes a cost of zero, and failed outcomes a cost of one, effectively calculating the probability of a successful outcome. Rules are then of the form *if predicted-cost-is-near-one, then avoid-task.*

Events Matrix

The learning opportunity is stored in an *events matrix* along with the cost evaluation and the environmental features observed when the learning opportunity occurred. This collection of feature-value vectors is presented in a uniform format for use by any learning mechanism. Additional features from the execution trace can be trivially added.

Each time the robot is idle, ROGUE processes the execution trace and appends new events incrementally to the events matrix. The learning algorithm then processes the entire body of data, and creates a new set of situation-dependent rules. By using incremental learning, ROGUE can notice changes and respond to them on a continual basis.

Learning

We now present the learning mechanism that creates the mapping from situation features and learning opportunities to costs. The input to the algorithm is the events matrix described above. The desired output is situation-dependent knowledge in a form that can be used by the planner.

We selected *regression trees* [5] as our learning mechanism because it is well-suited for learning search control rules for PRODIGY4.0 in this domain. Through its statistical analysis of the data, it is less sensitive to noise and exogenous events. Moreover, the symbolic representation of the features in the trees leads to an easy translation to search control rules. Other learning mechanisms may be appropriate in different robot architectures with different data representations.

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

We prune the tree using *10-fold random cross validation*, giving us the best tree size so as not to over-fit the data. The least important splits are then pruned off the tree until it reaches the desired size.

Creating Rules

Once the regression trees have been created, they need to be translated into PRODIGY4.0 search control rules. Control rules can be used to focus planning on particular goals and towards desirable plans, and to prune out undesirable actions. *Goal selection* control rules decide which tasks to focus on achieving, and hence aim at *creating a better plan. Applicable operator* rules decide what order to execute actions, and hence aim at *executing the plan more efficiently.*

A control rule is created at each leaf node; it corresponds to the path from the root node to the leaf. We decide whether to make a rule a *select*, *prefer-select*, *prefer-reject* or a *reject* based on the learned value of the leaf node. Recall that the success or failure of a training event is indicated with a value of 0.0 or 1.0 respectively. Hence leaf nodes with values close to 0.0 are considered *select* rules, and those close to 1.0 are considered *reject* rules. Intermediate values become *prefer* rules.

Status	CurrLoc	Date	CT	Who	Task	PickupLoc	DeliverLoc	Sensor0	Sensor1	...	Sensor22	Sensor23
0	5301	29	81982	JAN	DELIVERFAX	5310	5301	596.676	224.820	...	163.860	773.460
1	5301	1	12903	WILL	DELIVERFAX	5302	5301	81.564	84.612	...	157.764	154.716
1	5336	3	1200	JEAN	DELIVERMAIL	5336	5302	78.516	78.516	...	243.108	243.108
0	5321	4	8522	MCOX	DELIVERMAIL	5321	5328	456.468	166.908	...	84.612	84.612
1	5415	5	48733	REIDS	PICKUPCOFFEE	5415	5321	102.900	99.852	...	139.476	133.380
0	5304	6	9482	MMV	DELIVERMAIL	5304	5303	773.460	133.380	...	108.996	108.996
0	5310	6	76701	THRUN	PICKUPFAX	5320	5310	773.460	733.836	...	96.804	96.804
1	5301	7	22746	SKOENIG	DELIVERFAX	5313	5301	81.564	81.564	...	154.716	154.716
1	5409	8	1320	REIDS	PICKUPMAIL	5427	5409	81.564	78.516	...	157.764	157.764
0	5303	9	4141	SKOENIG	DELIVERMAIL	5301	5303	642.396	136.428	...	115.092	112.044
1	5313	10	83303	MMV	DELIVERFEDEX	5313	5313	130.332	136.428	...	105.948	105.948
1	5321	11	78501	JRS	PICKUPMAIL	5310	5321	145.572	148.620	...	90.708	90.708
0	5427	12	19745	KHAIGH	PICKUPMAIL	5427	5311	773.460	773.460	...	169.956	169.956
1	5307	12	29888	MCOX	DELIVERMAIL	5307	5415	148.620	148.620	...	90.708	87.660

Table 4: Sampling from the events matrix for the "Should I wait?" task. *Status* indicates whether the door was open; 1 is closed, *CT* is current time, in seconds since midnight. Additional features (not shown) include the operator name, *Year*, *Month*, *DayOfWeek*, *TaskRank*, *WhoRank* and other sensor values.

Experimental Results

We describe below two experiments to test ROGUE's learning ability. The first experiment explores creating applicable operator rules. The second set explores creating goal selection control rules. We used the robot simulator to collect this data because we have more control over the experiment. Haigh [6] describes experiments on the real robot.

Experiment 1: Execution Performance

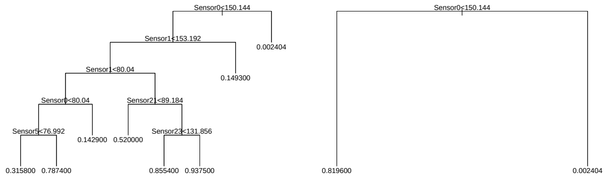
The goal in this experiment was to have ROGUE autonomously determine *"Should I wait?"* by detecting closed doors and inferring that the task can not be completed, for example when the occupant has unexpectedly stepped out of his office. In particular, we wanted ROGUE to learn an applicable operator search control rule that avoided trying to acquire or deliver an item when it sees that the door is closed.

For training data, ROGUE recorded a timeout event every time the average length of the front three sonars was less than 1.5 metres. Table 4 shows a sampling from the events matrix generated for this task. A total of 3987 events were recorded, of which 2419 (61%) were door-open events and 1568 (39%) were door-closed events. Notice that *Status* does not depend on features such as time or location, only on the three front-most sonar values.

The regression tree algorithm created rules that depend primarily on Sensor0 (the front-most sonar), shown in Figure 2. Other sensor values appear only in the *unpruned* trees.

Leaf values correspond to the learned value of the event in the situation described by each split. In the pruned trees, then, when sensor 0 has a reading of less than 150.144cm, the door is closed 81.96% of the time. When sensor 0 has a reading greater than 150.144cm, the door is open 99.76% of the time.

Table 5 shows the control rules generated from the pruned trees for PRODIGY4.0. Notice that each leaf in the pruned tree forms a rule, and that the single split value on the path from the root is now part of the rule. The generated rules were used at operator application time, i.e. when the operator is about to be released for



(a) Raw Tree.　(b) Pruned Tree.

Figure 2: Sample regression tree learned for the "Should I wait?" task.

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

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

Table 5: PRODIGY4.0 control rules generated for the learned pruned trees in the "Should I wait?" task.

execution.

A trace generated by PRODIGY4.0 when using these control rules appears in Haigh [6]. ROGUE discovers a closed door for a particular plan, and so the task planner rejects the action to acquire an item. Later, when ROGUE returns to the same room, the door is open. The task planner now decides to acquire the item, thereby demonstrating the effectiveness of these situation-dependent rules.

This experiment shows that ROGUE can use real-world execution data to learn when to execute actions, hence learning to execute plans more effectively.

Experiment 2: Planning Performance

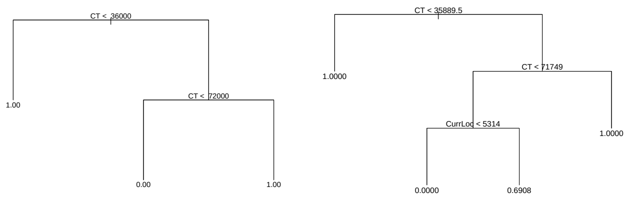
This experiment was designed to test ROGUE's ability to identify and use high-level features to learn to *create better plans.* The goal was to have ROGUE identify times for which tasks could not be completed, and then create goal selection rules of the form "reject task until..."

For training data, we generated two maps for the simulator. Between 10:00 and 19:59, all doors in the map were open. At other times, all doors were closed. When a door was closed, we defined the task as incompletable.

We were expecting the learned tree to resemble the example shown in Figure 3a, in which time was the only feature used to build the tree. Figure 3b shows the actual regression tree learned for this data (the same pruned tree was created for all tested deviance levels).

The unexpected appearance of the feature *CurrLoc* caused us to re-examine the data. We found that there was indeed a difference between room 5312 and 5316. The navigation module stopped the robot several feet away from the door of 5316 approximately 69% of the attempts, while centering the robot perfectly in front of 5312. Standing in front of a wall rather than at an open door caused the system to record a closed door and hence a failed task. In the real robot, this failure rate is considerably reduced because the vision module is used to help centre the robot correctly on the door.

ROGUE created four control rules for PRODIGY4.0, one for each leaf node, shown in Table 6. Notice that each leaf in the pruned tree forms a rule, and that each split value on the path from the root is used in the rule.



(a) Expected Tree. (b) Learned Tree.

Figure 3: Expected and Actual trees for door-open times. (Note: CT=36000 is 10:00, and CT=72000 is 20:00.)

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

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

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

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

Table 6: Learned PRODIGY4.0 control rules for the tree in Figure 3b.

These rules were used at goal selection time, allowing PRODIGY4.0 to reject tasks before 10am and after 8pm. (Note that goal selection rules do not contain execution-level features since these features can only be detected when the action is about to be executed.)

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

Related Work

Although there is extensive machine learning research in the artificial intelligence community, very little of it has been applied to real-world domains.

AI research for improving domain models has focussed on correcting action models and learning control rules [12; 14; 18]. Unfortunately, most of these systems rely on complete and correct sensing, in simulated environments with no noise or exogenous events.

In robotics, the difficulties posed by real-world domains have generally been limited to learning action parameters, such as manipulator widths, joint angles or steering direction [2; 3; 13]. Clementine [10] and CSL [16] both learn sensor utilities, including which sensor to use for what information.

Our work combines the AI approach of learning and correcting action models with the robotics approach of learning action costs. ROGUE learns from the execution experience of a real robot to identify the conditions under which actions have different probabilities.

Our work is closely related to Reinforcement Learning (overviewed by Kaelbling *et al.* [9]), which learns the value of being in a particular state, which is then used to select the optimal action. This approach can be viewed as learning the integral of action costs. However, most Reinforcement Learning techniques are unable to generalize learned information, and as a result, they have only been used in small domains[1]. Moreover, Reinforcement Learning techniques typically learn a universal action model for a *single* goal. Our situation-dependent learning approach learns knowledge that will be transferrable to other similar tasks.

Conclusion

ROGUE's learning capabilities are crucial for a planner operating in an incompletely modelled domain. ROGUE learns details of the domain that have not been modelled, and also responds to changes in the environment.

Our experiments show that situation-dependent rules are a useful extension to the task planner. ROGUE improves planning through examination of real-world execution data. ROGUE creates better plans because situation-dependent rules guide the planner away from hard-to-achieve tasks, and towards easy-to-achieve tasks. They also reduce execution effort by learning when to execute the action.

We view the approach as being relevant in many real-world planning domains. Situation-dependent rules are useful in any domain where actions have specific *costs*, *probabilities*, or *achievability criteria* that depend on a complex definition of the state. Planners 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. Haigh [6] and Haigh & Veloso [8] describe an implementation of the approach for the robot's path planner, along with extensive experimental results, demonstrating both the effectiveness and utility of the approach. Situation-dependent features are an effective way to capture the changing nature of a real-world environment.

[1] Generalization in RL is rare and may be very computationally intense [1; 4; 11].

References

[1] L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *ICML95*, pages 30–37. Morgan Kaufmann, 1995.

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

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

[4] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems*, volume 7, pages 369–76. MIT Press, 1995.

[5] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth & Brooks/Cole, 1984.

[6] K. Z. Haigh. *Situation-Dependent Learning for Interleaved Planning and Robot Execution.* PhD thesis, Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1998.

[7] K. Z. Haigh and M. M. Veloso. Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, 5(1):79–95, 1998.

[8] K. Z. Haigh and M. M. Veloso. Learning situation-dependent costs: Improving planning from probabilistic robot execution. In *Autonomous Agents*, pages 231–238. AAAI Press, 1998.

[9] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[10] J. Lindner, R. R. Murphy, and E. Nitz. Learning the expected utility of sensors and algorithms. In *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems*, pages 583–590. IEEE Press, 1994.

[11] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, Computer Science, University of Rochester, Rochester, NY, 1995.

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

[13] D. A. Pomerleau. *Neural network perception for mobile robot guidance.* Kluwer, 1993.

[14] W.-M. Shen. *Autonomous Learning from the Environment.* Computer Science Press, 1994.

[15] R. Simmons, R. Goodwin, K. Z. Haigh, S. Koenig, and J. O'Sullivan. A layered architecture for office delivery robots. In *Autonomous Agents*, pages 245–252. ACM Press, 1997.

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

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

[18] X.-M. Wang. *Leaning Planning Operators by Observation and Practice.* PhD thesis, Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996.