

An Instance-based State Representation for Network Repair

Michael L. Littman and Nishkam Ravi

Department of Computer Science
Rutgers University
Piscataway, NJ
{mlittman,nravi}@cs.rutgers.edu

Eitan Fenson and Rich Howard

PnP Networks, Inc.
Los Altos, CA
{eitan,reh}@pnphome.com

Abstract

We describe a formal framework for diagnosis and repair problems that shares elements of the well known partially observable MDP and cost-sensitive classification models. Our cost-sensitive fault remediation model is amenable to implementation as a reinforcement-learning system, and we describe an instance-based state representation that is compatible with learning and planning in this framework. We demonstrate a system that uses these ideas to learn to efficiently restore network connectivity after a failure.

Introduction

The first, and possibly most important, question to answer when creating a reinforcement-learning system (Sutton & Barto 1998) for an application is how the state space for learning and planning should be defined. A good choice of state representation can make the difference between a problem that is trivial and one that is impossible to solve via learning. There is a great need for learning approaches that can evolve toward a state representation that is natural and efficient for the problem at hand based on relatively raw inputs.

Nowhere is the need for automatic development of state representations more evident than in environments that are not Markov in the decision maker's direct observables. In such partially observable environments, decision makers often must exploit their history of interaction and reason explicitly about their uncertainty about the state of the world during decision making, particularly if they need to take actions to gain information to behave effectively. Attempts to learn complex environments represented with hidden states (Chrisman 1992), histories (McCallum 1995a), and as predictions of future observations (Littman, Sutton, & Singh 2002) have met with some limited success, primarily in simulations.

This paper explores an instance-based representation for a special kind of partially observable environment called a CSFR (cost-sensitive fault remediation) model. The CSFR model captures a diagnosis-and-repair situation in which a

system can detect if it has entered a failure mode, take actions to gain information about the failure mode, attempt repair actions, and detect when correct functioning has been restored. Although we believe that the learning approach we describe can be applied in more general environments, we focus here on this specialized subclass.

We demonstrate our approach to learning and planning in the CSFR framework with an implemented network-repair application.

The CSFR Model

The CSFR model was motivated by an exploration of problems that arise in sequential decision making for diagnosis and repair including problems of web-server maintenance, disk-system replacement, and the network-repair application illustrated later in the paper.

In cost-sensitive fault remediation, a decision maker is responsible for repairing a system when it breaks down. To narrow down the source of the fault the decision maker can perform a *test action* at some cost and to repair the fault it can carry out a *repair action*. A repair action incurs a cost and either restores the system to proper functioning or fails. In either case, the system informs the decision maker of the outcome. The decision maker seeks a minimum cost policy for restoring the system to proper functioning.

CSFR bears many similarities to cost-sensitive classification (CSC, see Turney 1995, Greiner, Grove, & Roth 1996). Like standard classification, CSC is concerned with categorizing unknown instances. However, CSC can be viewed as a kind of sequential decision problem, in which the decision maker interacts with a system by requesting attribute values (running test actions). The decision maker's performance is the total of the individual action costs plus a charge for misclassification, if applicable. Diagnosis tasks formulated as CSCs include hidden information—the identity of the fault state—making them a kind of partially observable Markov decision process (POMDP). POMDPs can be very notoriously difficult to solve, although there is reason to believe that CSCs result in relatively benign POMDP instances (Zubek & Dietterich 2002, Guo 2002).

The main difference between CSC and CSFR, and the principle novelty of our framework, is that a decision maker in a CSFR model can use feedback on the success or failure of a repair action to attempt an alternate repair. In CSC, clas-

sification actions end episodes, whereas in CSFR, episodes continue until the fault is repaired. Although this constitutes a relatively small change, we feel it adds significantly to the degree of autonomy supported by the model. Specifically, CSFRs are amenable to reinforcement learning as they have no dependence on supervisory signals to evaluate the attempted classifications—as long as the system is capable of detecting proper and improper function, learning can take place. We demonstrate this capability in an implemented network-repair scenario later in the paper.

In contrast to general POMDPs, faults in a CSFR remain constant until repaired. This simplifies the construction of policies compared to the general case and may enable efficient approximations. Therefore, the CSFR model lies at an intermediate point on the spectrum from the simple CSC model to the general POMDP model.

CSFR Definition

A CSFR problem can be defined formally by the quantities:

- S , a set of fault states;
- $\Pr(s)$, a prior probability distribution over the states $s \in S$;
- A_T , a set of test actions;
- A_R , a set of repair actions;
- $c(s, a)$, a cost function over actions $a \in A_T \cup A_R$ and states $s \in S$;
- $o(s, a)$, an outcome or observation model over actions $a \in A_T \cup A_R$ and states $s \in S$.

In this paper, we assume the range of the outcome model $o(s, a)$ is 0 and 1. For repair actions $a \in A_R$, $o(s, a) = 1$ is interpreted to mean that the repair action a returns the system to proper working order if afflicted with fault s . A *repair episode* is the sequence of actions and outcomes from the time the fault is first detected to the time it is repaired. Note that initial fault detection and the determination that the fault is resolved happens in the environment—outside of the CSFR model. In addition, repair episodes are assumed to be independent.

Finding an Optimal Policy

Note that we are assuming deterministic observations in the CSFR model. As in POMDPs, stochastic observations can be modeled by increasing the size of the state space. Unlike POMDPs, the expansion may require an exponential increase in the number of states, perhaps to one state for each possible combination of stochastic outcomes. In addition, stochastic observations can mean that there is additional information to be gained by repeating a test or repair action more than once. Ignoring this issue is a significant limitation of the current formulation.

Nevertheless, the combination of deterministic observations and the lack of state transitions provides simplifications in the state space for planning that can be exploited. At the beginning of the repair process, the probability of a fault s is simply its prior $\Pr(s)$. After test actions have been taken, some states are inconsistent with the new information,

and their probability becomes zero while the probability of the remaining states is normalized to one. So, during a repair episode, every fault-state probability is either zero or is proportional to its prior probability.

We say that a fault state s is *active* if it is consistent with the outcome of all test actions during the current episode. At the beginning of a repair episode, all fault states are active. As the episode progresses and assuming that the decision maker only selects test actions that have differing outcomes for the currently active fault states, each decision results in a shrinking set of active states. It follows that the number of decision situations that the decision maker will face is bounded by the size of the power set of S . Although this can be a very large number, it is small enough that it has not been a bottleneck in the applications we have examined to date.

We can find an optimal repair policy via dynamic programming. Let B be the power set of S , which is the set of belief states of the system. For each $b \in B$ and $a \in A_T \cup A_R$, define the expected value of action a in belief state b as the expected cost of the action plus the value of the resulting belief state:

$$Q(b, a) = \begin{cases} \Pr(b_0)/\Pr(b)(c(b_0, a) + V(b_0)) \\ \quad + \Pr(b_1)/\Pr(b)(c(b_1, a) + V(b_1)), \\ \quad \text{if } \Pr(b_0) > 0 \text{ and } \Pr(b_1) > 0, \\ \quad \text{or } \Pr(b_1) > 0 \text{ and } a \in A_R; \\ \infty, \text{ otherwise.} \end{cases}$$

Here, $b_i = \{s \in b \mid o(s, a) = i\}$ is the belief state resulting from applying action a from belief state b and obtaining outcome $i \in \{0, 1\}$. If $a \in A_R$ and $i = 1$, we define $V(b_1) = 0$, as there is no additional cost incurred once a repair action is successful. In all other cases, the value of a belief state is the minimum action value taken over all available choices: $V(b) = \min_{a \in A_T \cup A_R} Q(b, a)$. The quantities $\Pr(b)$ and $c(b, a)$ are the probability and cost functions extended to belief states in the natural probability-weighted way. Specifically, let $\Pr(b) = \sum_{s \in b} \Pr(s)$ and $c(b, a) = \sum_{s \in b} \Pr(s)c(s, a)$. The conditions that $\Pr(b_0) > 0$ and $\Pr(b_1) > 0$ (or $\Pr(b_1) > 0$ for a repair action $a \in A_R$) in the recursion ensure that the equations bottom out; no quantity is defined in terms of itself. This restriction rules out only suboptimal policies that include actions that do not alter the belief state and guarantees that the algorithm can be implemented without considering cyclic dependencies.

These equations can be evaluated in time proportional to $2^{|S|}(|A_T| + |A_R|)(|S| + |A_T| + |A_R|)$, at which point the optimal policy can be extracted by choosing the action a in belief state b that minimizes $Q(b, a)$. Such a policy will optimally trade off actions to gain information, actions to repair, and the information gained from a failed attempt at repair, in a way that minimizes the total expected cost of repair.

Example

Table 1 illustrates a small CSFR example with two fault states, A and B .

The planning process for this example begins with the belief state $\{A, B\}$. It considers the test actions **DefaultGateWay** and **PingIP** and the repair actions **FixIP**, **UseCachedIP**,

	A		B	
DnsLookup	0	(2500ms)	0	(2500ms)
DefaultGateway	1	(50ms)	0	(50ms)
PingIp	0	(50ms)	1	(250ms)
RenewLease	1	(2500ms)	0	(1000ms)
UseCachedIP	0	(10000ms)	1	(25000ms)
FixIP	1	(20000ms)	1	(20000ms)

Table 1: A small table of CSFR states. Each row lists the outcome and cost of a test action (DnsLookup, DefaultGateway, and PingIP) or repair action (**FixIP**, **UseCachedIP**, **RenewLease**, in boldface) for each of the states. The priors are $\Pr(A) = .25$ and $\Pr(B) = .75$.

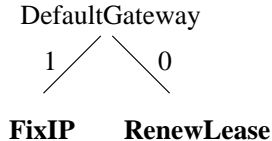


Figure 1: The optimal policy for the small example CSFR.

and **RenewLease**. It does not consider DnsLookup since it neither provides information (always 0), nor has a non-zero chance of repair. In evaluating the action PingIP, the algorithm finds that outcome 0 has a probability of .25 and outcome 1 has a probability of .75. Its expected cost from belief state $\{A, B\}$ is then

$$.25(50 + \text{cost}(\{A\})) + .75(250 + \text{cost}(\{B\})). \quad (1)$$

The expected cost from belief state $\{A\}$ is computed recursively. Since all test actions have an outcome with an estimated probability of 0, only repair actions are considered. Of these, **RenewLease** is chosen as the optimal action, with a cost of 2500. Similarly, the optimal expected cost from belief state $\{B\}$ is 20000 by taking action **FixIP**. Substituting these costs into Equation 1, we find the optimal expected cost of repair from belief state $\{A, B\}$, starting with PingIP, is 15825. The minimum over all actions from belief state $\{A, B\}$ is achieved by DefaultGateway (expected cost 15675), leading to a repair policy shown in Figure 1.

Instance-based CSFR Learning

The previous section described how to create optimal repair policies for problems specified as CSFR models. A CSFR model of an application can be created through carefully supervised experimentation and fault injection (Li *et al.* 2002). That is, human experimenters can identify the set of fault states that they would like their repair system to handle, create the symptoms of these faults, and measure the costs and outcomes for each action for each injected fault. In addition to being labor intensive, this fault-injection methodology is limited to collecting data on a small set of artificially created fault states instead of the problems that occur during actual operation.

We are interested in developing learning methodologies that can create and manipulate models based on unstructured interaction with the environment. We now describe

	V	W	X	Y	Z
DnsLookup	0	0	?	?	?
DefaultGateway	1	0	1	?	0
PingIp	0	?	0	1	1
RenewLease	1	0	?	?	?
UseCachedIP	0	1	0	?	1
FixIP	?	?	1	1	1

Table 2: A small collection of CSFR episodes, E .

an instance-based reinforcement-learning approach we developed for the CSFR setting.

Instance-based Representations

In instance-based reinforcement-learning, state spaces for learning are constructed from examples of states experienced in interaction with the environment. Instance-based approaches have been primarily explored in continuous-space environments (Moore, Atkeson, & Schaal 1995; Santamaría, Sutton, & Ram 1997; Smart & Kaelbling 2000; Forbes & Andre 2000; Ormonett & Sen 2002), where learning state-action values for each possible state is infeasible. Instead, a value is learned for each observed state. A similarity function is used to relate non-identical observations resulting in a pooling of experience information across different episodes.

Whereas continuous-space environments suffer from an overspecificity of information—we may never see the same state twice—observations in partially observable environments are typically inadequate to sufficiently specify the decision maker’s situation. One way to make up for this lack of information is to view the “state” as the sequence of actions and observations made during an entire episode. As in the continuous-space case, we now run the risk of never returning to any situation a second time, making generalization between experiences a requirement for successful learning.

Motivated by these concerns, McCallum (1995b) described an application of the instance-based approach to partially observable domains. He defined two histories as being similar if they match in their last k actions and observations. The degree of similarity is the length of the match. He demonstrated successful learning on several simulated partially observable environments.

CSFR Learning

In the CSFR setting, the number of possible history sequences is on the order of $(2|A_T| + |A_R|)!$ since an episode consists of an ordering of test actions and their outcomes, along with failed repair actions. Because action outcomes are deterministic, there is no need to repeat an action twice. An example episode, which we’ll call V , is: DefaultGateway=1, DnsLookup=0, **UseCachedIP**=0, PingIp=0, **RenewLease**=1.

Note that the order of previous actions is not relevant for decision making. Thus, a significantly smaller space of possible experiences can be derived by summarizing an episode by the set of actions taken along with their outcomes. The revised representation of episode V , along with four other

episodes, is illustrated in Table 2. Using this reordering observation, the number of distinct episodes is no more than $3^{|A_T|} \cdot 2^{|A_R|-1} \cdot 2^{|A_R|}$, since each test action has an outcome that is 0, 1, or ? (not yet executed), at most one repair action can be successful, and the others have outcomes of 0 or ?.

Our instance-based approach begins with a set of episodes E . For $e_1 \in E$ and $e_2 \in E$, we use the shorthand $e_1 \subseteq e_2$ to mean that all actions that appear in e_1 also appear in e_2 with the same outcome. For example, note that $Y \subseteq Z$ in Table 2.

One method for applying the CSFR planning algorithm from the previous section to a set of episodes E would be to find a small set of fault states that *covers* a set of episodes. That is, we'd want a set of states S such that, for every $e \in E$, there is some $s \in S$ such that $e \subseteq s$. However, explicitly finding such a set of fault states is likely to be hard.

Proposition: *Finding a set of fault states that covers a set of episodes is equivalent to graph coloring. This implies that even approximating the smallest set of fault states for covering a set of episodes is hard (Blum 1994). For example, if the size of the smallest set of fault states that covers the set of episodes is n , no algorithm can be guaranteed to find a set of fault states that covers the episodes of size less than n^ϵ for some $\epsilon > 0$ in polynomial time unless $P=NP$.*

Proof sketch: We present a reduction from graph coloring to the problem of finding a minimum set of fault states. Take a graph G and create a set of episodes as follows. We have a test action for each edge (i, j) that appears in G . We also have an episode for each vertex in G . The outcome of the test action corresponding to edge $(i, j) \in G$ in the episode corresponding to vertex k is 0 if $k = i$, 1 if $k = j$, and ? otherwise. As a result of this construction, two vertices in G have an edge between them (meaning that can't be assigned the same color) if and only if their corresponding episodes conflict—they are consistent with different underlying states. \square

Instead of identifying a minimum set of states to cover the set of episodes E , we change the CSFR planning algorithm from the earlier section to use subsets E as belief states. We say that an episode $e \in E$ is active if the current episode $q \subseteq e$. As an upper bound, the number of belief states cannot be larger than $2^{|E|}$. However, in practice, far fewer belief states are considered during planning (see the results section). We hypothesize, but have not yet proven, that there is a bound of the form $c \cdot 2^{|S|}$ on the number of belief states encountered during planning.

One interesting difference between our instance-based reinforcement-learning algorithm and that of McCallum (1995b) is that the state space for planning is that of belief states formed by subsets of instances instead of individual instances. This provides a more direct approach to the problem of taking actions to gain information. We hypothesize that our representation trades additional computation in the planning stage for more efficient use of available data.

Experimental Results

In this section, we demonstrate the ideas behind our instance-based learning algorithm for the CSFR model in a network-repair scenario.

Network Repair

As our dependence on the Internet grows, continuous availability of network services is important. Recoverability from events that impair functionality is therefore becoming a focus of software-systems research. Work has been done to systematically study and classify the kinds of faults that can occur in a system, ranging from hardware and configuration faults (Li *et al.* 2002) to faults generated as a result of executing corrupt processes/applications (Bohra *et al.* 2004). Our initial learning-based approach to diagnosis and repair complements existing systems research by providing a framework for flexibly introducing information gathering and repair actions to a self-healing system without the need for detailed hand tuning; the system discovers the relationship between observed faults and optimal repair actions.

In our first experiments, we focus on recovering from faults that result from a corrupted network-interface configuration. We model the computer system as a CSFR and have implemented a set of triggers to detect faults. A repair episode is initiated if any change is detected in the network configuration. Elapsed time is the cost measure the planner seeks to minimize. A repair episode terminates when the system can reach the wide-area network.

Our test and repair actions were implemented in Java in a Windows XP environment. The test actions were:

- **PluggedIn:** Runs ipconfig to see whether or not the system is connected to the network.
- **IsWireless:** Returns 1 if the network interface is wireless, 0 otherwise.
- **PingIp:** Pings the current IP address of the local machine and indicates whether or not it is reachable.
- **PingLhost:** Pings 127.0.0.1, returning 1 if it is reachable, 0 otherwise.
- **PingGateway:** Pings the network gateway, returning 1 if it is reachable, 0 otherwise.
- **DnsLookup:** Checks to see if DNS is working properly by executing nslookup on an external server. Returns 1 if the attempt is successful, 0 otherwise.
- **DefaultIpAddr:** Checks to see if the IP address is a valid local address by performing a logical “or” with the netmask.
- **DefaultNetmask:** Checks to see if the netmask is valid by comparing it with previously stored valid netmask values.
- **DefaultNameServer:** Returns 1 if the DNS is valid and if an external server can be looked up.
- **DefaultGateway:** Checks to see if the gateway setting is valid by comparing it with previously stored valid gateway values.
- **DHCPEnabled:** Returns 1 if the network interface is DHCP enabled, otherwise 0.

- **PnPReachDns**: Tries to establish an http connection with a remote server. Returns 1 if the attempt is successful, 0 otherwise.

The repair actions were:

- **RenewLease**: Renews DHCP lease.
- **UseCachedIP**: Restores settings for the IP parameters to cached values in an attempt to fix the network configuration.
- **FixIP**: Pops up a dialog box, asking the user to input correct network parameters. For the purpose of our automated experiments, we had the system read correct network parameters from a file.

Results

The network-repair system runs in the background on a workstation, initiating repair episodes when problems are detected. Although we have anecdotal evidence that the system is able to repair naturally occurring faults, including a loss of authentication on our local wireless network after a timeout, we needed a methodology for collecting fault data more quickly.

We constructed a separately running piece of software we called the “breaker”, programmed to introduce any of the following faults:

- **Set bad static netmask**: Invokes netsh, a windows configuration program, with a bad value for the netmask.
- **Set bad static gateway**: Invokes netsh with a bad value for gateway.
- **Set bad static IP address**: Invokes netsh with a bad IP address.
- **Set bad static value for DNS server**: Invokes netsh with a bad value for DNS.
- **Set DNS to DHCP with no lease**: Invokes netsh with DNS set to use DHCP and then cancels the DHCP lease.
- **Set IP address to DHCP with no lease**: Invokes netsh with the IP address set to use DHCP and then cancels the DHCP lease.
- **Lose existing DHCP lease**: Invokes ipconfig to release the DHCP lease.

For our data collection, we ran the network-repair software concurrently with the breaker to inject faults. Faults were selected uniformly at random from the list above and executed at regular intervals of approximately two minutes, providing ample time for recovery.

During learning, we had the system choose actions greedily according to the best policy found for the previous episodes. If the system reached a current episode q such that $q \not\subseteq e$ for all episodes $e \in E$, the system executed all test actions and all repair actions (until one was successful) approximately in order of increasing cost. Each episode q resulting from executing all actions was clearly a distinct fault state.

The learned policy after 95 repair episodes is illustrated in Figure 2. In this policy, **RenewLease** is attempted only when PingGateway, PingIP and DNSLookup all fail, with

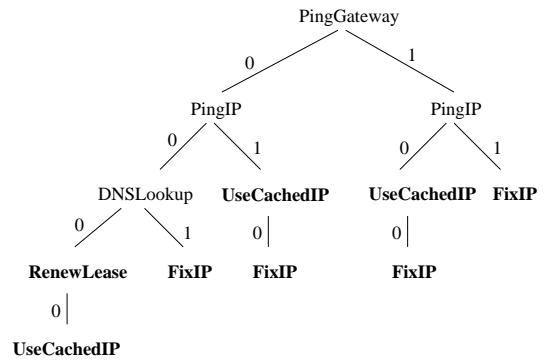


Figure 2: A learned policy for the network-repair domain.

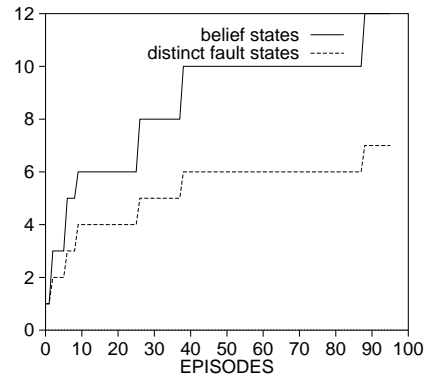


Figure 3: Growth of fault states and belief states with increasing number of episodes.

UseCachedIP as the backup repair action. **RenewLease** works differently from **UseCachedIP** and **FixIP** in that it obtains the IP parameters dynamically. In almost all the other cases, **UseCachedIP**, being cheaper than **FixIP**, is tried first with **FixIP** used as a backup action. The exceptions are the two cases where **FixIP** is tried without trying **UseCachedIP**, which stems from the lack of adequate exploration. Aside from these imperfections, the policy appears perfectly suited to our test domain.

We measured the number of belief states encountered during the planning process as a measure of planning complexity, along with the number of distinct fault states encountered. Although our best upper bound on belief states is exponential in the number of episodes, it is evident from Figure 3 that the number of belief states does not grow nearly this fast. It is roughly twice as large as the number of distinct fault states, which itself grows quite slowly with the number of episodes in the database, increasing only when new fault states are encountered.

Five test actions were used as triggers to detect failures and initiate repair episodes. They were DefaultIpAddress, DefaultNetmask, DefaultNameServer and DefaultGateway. A repair episode began if the value of any of the test actions changed. These test actions were run every few seconds. Note that since the outcomes of these tests were known at the beginning of the episode, they can be included in the

initial episode used in planning “for free”. Our runs that included triggers generally resulted in more efficient policies, although were less interesting from a learning standpoint as they generally needed no additional tests to select an appropriate repair action.

Beyond the network settings discussed above, we also created a preliminary implementation of a test action that monitors processes to identify those consuming more than 90% of CPU time and a corresponding repair action to terminate such processes. We feel a more complete system would include checks to see that other resources such as network bandwidth and memory usage also remain within appropriate limits.

Conclusions

This work demonstrates an approach to learning for autonomous network repair. Although we have explored several other scenarios to which our cost-sensitive fault remediation model can be applied in simulation, our experience implementing the learning algorithm on a live network helped illustrate the robustness of the basic idea, and also indicated that we should revisit some of the underlying assumptions in our model.

The CSFR model assumes that episodes are independent. While this is true to a first approximation in the network-repair setting, we found many cases in which the effect of a repair action or an injected fault lingered from one episode to the next. For example, several repairs have the side effect of enabling or disabling DHCP. Enabling DHCP in one episode means that the system will not encounter static IP problems in the next, leading to a non-stationary failure distribution.

In our study, we used action-execution time to define costs. In practice, an additional significant action cost is its impact of the user, either due to the need for the user’s involvement (like “Check if the network cable is plugged in”) or because of the action’s effect on other system functions (like “Reboot”). These costs are easily taken into consideration in the model; the difficulty is how to measure them during learning.

Our model also assumes that test and repair actions have no side effects apart from the possibility of fixing a failure. Not only is this assumption violated in the case of repair actions that alter, and possibly worsen, the network configuration, but it limits the types of actions that can be included and how they are used.

Two other issues worth mentioning are the assumption of stationarity in the fault distribution and the lack of explicit exploration to learn more effective repair strategies. These issues are ubiquitous concerns in reinforcement learning and we plan to study them in future work in the context of autonomous diagnosis and repair.

Acknowledgments

We gratefully acknowledge support from DARPA IPTO and the National Science Foundation.

References

- Blum, A. 1994. New approximation algorithms for graph coloring. *Journal of ACM* 41:470–516.
- Bohra, A.; Neamtiu, I.; Gallard, P.; Sultan, F.; and Iftode, L. 2004. Remote repair of OS state using backdoors. Technical Report DCS-TR-543, Department of Computer Science, Rutgers University.
- Chrisman, L. 1992. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 183–188. San Jose, California: AAAI Press.
- Forbes, J., and Andre, D. 2000. Real-time reinforcement learning in continuous domains. Appeared in AAAI Spring Symposium on Real-Time Autonomous Systems.
- Greiner, R.; Grove, A. J.; and Roth, D. 1996. Learning active classifiers. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML-96)*, 207–215.
- Guo, A. 2002. Active classification with bounded resources. At <http://cs.umass.edu/anyuan/publications/SSS402AGuo.ps>.
- Li, X.; Martin, R.; Nagaraja, K.; Nguyen, T. D.; and Zhang, B. 2002. Mendosus: A SAN-based fault-injection test-bed for the construction of highly available network services. In *First Workshop on Novel Uses of System Area Networks (SAN-1)*.
- Littman, M. L.; Sutton, R. S.; and Singh, S. 2002. Predictive representations of state. In *Advances in Neural Information Processing Systems 14*, 1555–1561.
- McCallum, A. K. 1995a. *Reinforcement Learning with Selective Perception and Hidden State*. Ph.D. Dissertation, Department of Computer Science, University of Rochester.
- McCallum, R. A. 1995b. Instance-based utile distinctions for reinforcement learning with hidden state. In *Proceedings of the Twelfth International Conference on Machine Learning*, 387–395. San Francisco, CA: Morgan Kaufmann.
- Moore, A. W.; Atkeson, C. G.; and Schaal, S. 1995. Memory-based learning for control. Technical Report CMU-RI-TR-95-18, CMU Robotics Institute.
- Ormoneit, D., and Sen, S. 2002. Kernel-based reinforcement learning. *Machine Learning* 49:161–178.
- Santamaría, J. C.; Sutton, R. S.; and Ram, A. 1997. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior* 6(2):163–217.
- Smart, W. D., and Kaelbling, L. P. 2000. Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, 903–910.
- Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. The MIT Press.
- Turney, P. D. 1995. Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of Artificial Intelligence Research* 2:369–409.
- Zubek, V. B., and Dietterich, T. G. 2002. Pruning improves heuristic search for cost-sensitive learning. In *Proceedings of the International Conference on Machine Learning*, 27–34.