RL-TOPs: An Architecture for Modularity and Re-Use in Reinforcement Learning

Malcolm R. K. Ryan
Department of Artificial Intelligence
School of Computer Science and Engineering
University of New South Wales
Sydney 2052 Australia
malcolmr@cse.unsw.edu.au

Mark D. Pendrith*
Department of Artificial Intelligence
School of Computer Science and Engineering
University of New South Wales
Sydney 2052 Australia
pendrith@cse.unsw.edu.au

Abstract

This paper introduces the **RL-TOPs** architecture for robot learning, a hybrid system combining teleo-reactive planning and reinforcement learning techniques. The aim of this system is to speed up learning by decomposing complex tasks into hierarchies of simple behaviours which can be learnt more easily. Behaviours learnt in this way can subsequently be re-used to solve a variety of problems, reducing the need to learn every new task from scratch. It is even possible to learn multiple behaviours simultaneously, thus making more efficient use of experience. We demonstrate these advantages in a simple simulated environment.

1 INTRODUCTION

Programming robots is difficult (Dorigo, 1996). Often the best way for the robot to solve a problem is unknown, or hard to express. The real world is dynamic, and to be truly autonomous, robots need to be able to cope with a changing environment (Covigaru & Lindsay, 1991). Robot programming would be greatly simplified if robots were able to learn appropriate behaviours of their own accord, and could adapt those behaviours to changes in the world around them. Reinforcement Learning (RL) provides an elegant theoretical framework to achieve these goals but often fails in practice due to the "curse of dimensionality" operating in large state spaces and with complex problems such as those typically found in real robot domains. As the number of states grows, the problem of determining the best action to perform in each state becomes impossibly difficult.

This problem is not peculiar to RL, traditional robot programmers have faced it also. It is generally not feasible to produce a single monolithic control system which handles all possibilities. Instead, the trend has been towards *behaviour-based programming* (Matarić, 1996). A complex task is decomposed into a set of simple modules or *behaviours*, each of which handle a small part of the problem. These are more easily programmed, and can then be combined to solve the full problem.

One such technique, Brook's *subsumption architecture* (Brooks, 1986), has been successfully transferred to the RL domain, to simplify learning. Mahadevan and Connell (Mahadevan & Connell, 1992) showed that a complex learning task (robot box-pushing), which could not be learnt by a simple reinforcement learner, could, however, be learnt by decomposing it into a subsumption-style hierarchy of simple behaviours, and learning each of these behaviours as distinct reinforcement learning tasks. Thus the robot effectively had several separate learning modules, each of which works independently to learn a sub-part of the task, but which can all cooperate together to provide the overall solution to the problem.

Task decomposition of this kind is well recognised as a way to improve learning rates. As each module only has to learn its behaviour on a small subset of possible states, its search-space is reduced, and so it can find the optimal policy more quickly. Other authors to have produced algorithms based on this realisation include Kaelbling's HDG (Kaelbling, 1993), Dayan and Hinton's Feudal Reinforcement (Dayan & Hinton, 1992) and Dietterich's MAXQ (Dietterich, 1997) algorithms. These algorithms differ from Mahadevan and Connell's

* Current address: Daimler-Benz Research and Technology Center, 1510 Page Mill Rd, Palo Alto, CA 94304, USA. e-mail: pendrith@rtna.daimlerbenz.com

in that they are based on more geometrical decompositions of the world, rather than using specific domain knowledge to define the behaviours. Because of this, they appear to be less applicable to problems in robotics, which involve high-dimensional state information, from a variety of sensing apparatus, without a simple uniform geometry.

The advantages of the subsumption-architecture, however, are offset by the rigidity of the representation used. The hierarchy has to be designed by hand by the programmer, which can be a non-trivial task for many problems. What is more, a new task requires a new set of behaviours and a new hierarchy. Is it possible to design a more flexible system that can automatically build behaviour hierarchies to solve particular problems? Can behaviours learnt to solve one task be re-used to accelerate the learning of others? These are the questions that this paper seeks to address.

2 TELEO-REACTIVE PLANNING

This problem of selecting and ordering an appropriate set of predefined behaviours to achieve a certain goal has traditionally been the domain of planning algorithms. Historically, planning systems have been deemed unsuitable for robot control, because they failed to model the complexity of the real world. Plans were based on sequences of instantaneous actions, which were expected to succeed every time; but in the real world actions take time to perform, and are not always reliable. However modern planning algorithms are now able to produce plans which closely resemble the behaviour based architectures of Brooks and others. Plans can now include durative actions, which operate over a period of time. Execution of plans is reactive (i.e. the state of the world is constantly re-evaluated to determine which action to perform), and universal (i.e. contingencies exist for all situations).

One such planner is Nilsson's Teleo-Reactive (TR) planning system (Nilsson, 1994). It is based around the notion of a *teleo-operator* (or *TOP*), which is a means of describing a durative action in terms of its conditions and effects. A TOP consists of an action a, a pre-image π and a post-condition λ. The pre-image and post-condition are conjunctions of predicates from the planner's state description language. The action may be a simple primitive action, or may be a complex behaviour in its own right. The TOP $a : \pi \rightarrow \lambda$ signifies that if a is executed while π is true, then λ will eventually become true. Until such time as λ is achieved, π is maintained[1].

Teleo-reactive plans are represented as structures called *TR-Trees*. Nodes in TR-Trees represent state descriptions, with the root node as the goal. Connections between nodes are labelled with actions, indicating that if the action shown is executed in the lower node, then the condition of the upper node will eventually be achieved.

TR-trees are executed reactively. The nodes in the tree are continually re-evaluated and the action corresponding to the shallowest true node is executed. If at any time there is no true node in the tree, then the planner can be reactivated to grow the plan to cover the new situation; thus TR-trees represent (near-) universal plans.

3 REINFORCEMENT LEARNT BEHAVIOURS AND TELEO-OPERATORS

Like TOPs, behaviours acquired by reinforcement learning are also durative actions with a pre-image (application space) and a post-condition (goal). Given a suitable language to describe these attributes, a set of reinforcement learnt behaviours can easily be represented as a list of TOPs. A TR-planner could then be used to combine these behaviours automatically into a hierarchy to solve a given problem, removing the need for the programmer to do this by hand.

Furthermore, the same TOP descriptions can also be used at the lower level as reinforcement schema for the learning algorithm: The post-condition, if achieved, indicates success, which should be rewarded. Prematurely quitting the pre-image indicates failure, which carries a punishment. Thus the one description has two functions: it is used at the high level to tell the planner how to use the behaviour, and at the low level to tell the learner what it is trying to learn. This duality is the basis of the Reinforcement Learnt TOPs (RL-TOPs) system.

4 THE RL-TOPS ARCHITECTURE

The RL-TOPs architecture is a combination of a simple goal-regression TR-planner, and the discounted-

[1] A TOP may also have side-effects which are not part of its post-condition, but these are not relevant to the current discussion.

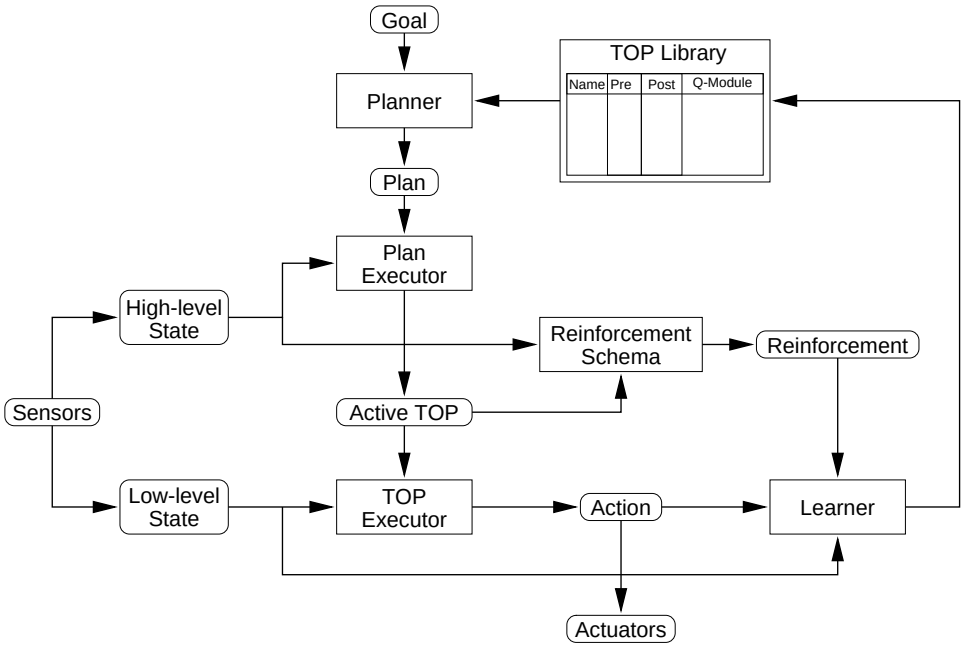


Figure 1: The RL-TOPs architecture.

reward reinforcement learning algorithm C-Trace[2] (Pendrith & Ryan, 1996). An outline is shown in Figure 1.

Based on the domain and the problem to be solved, the user provides five things:

- A **Low-level State** representation, based on the robot's sensors,
- A set of primitive **Actions**, based on the available actuators,
- A **High-level State** description language (which includes whatever features of the state space are likely to be relevant to the planner, including the goal),
- A **Goal** description,
- A set of behaviour descriptions of the form *(Name, Pre-image, Post-condition)*, which form the **RL-TOP Library**.

The first thing the system does is to supplement each of these RL-TOPs with its own Q-Module. This contains all the information required by the reinforcement learning algorithm to represent the behaviour. The primary component is the utility (or Q) function, but there may be other components depending on the algorithm. Unless previously saved behaviours are being re-used, the Q-function is initialised to be zero everywhere.

Now, given the goal definition and the library of behaviours available to it, the **Planner** constructs a plan in the form of a TR-Tree. The Planner only constructs as much of the tree as is necessary at any time. Initially the tree consists of just the goal node. As the agent encounters situations which aren't covered by the plan, the Planner will add new nodes to the tree to cover these states, and will add appropriate actions to the plan to link them in to the tree.

The plan is passed to the **Plan Executor**, which also reads the current high-level state description, and chooses which TOP to execute. If the plan does not cover the state, then the Executor re-calls the Planner. Otherwise, the selected TOP is passed to the TOP Executor.

The **TOP Executor** takes the active RL-TOP and the current low-level state, and decides which low-level action to execute. Typically, this will be the policy action provided by the TOP's Q-Module, but an occasional exploratory action may also be performed. For

[2]The actual reinforcement learning algorithm used is not important, except insofar as it must support learning from both successful and unsuccessful trials. This includes most common RL algorithms such as Q-Learning (Watkins, 1989) and SARSA(λ) (Singh & Sutton, 1996).

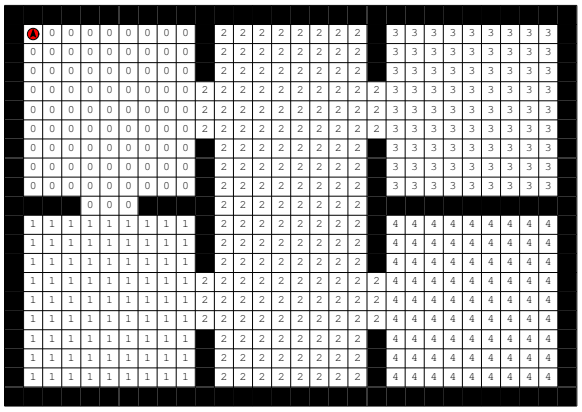


Figure 2: The gridworld domain.

RL-TOP	pre-image	post-condition
go02	room(0)	room(2)
go20	room(2)	room(0)
go12	room(1)	room(2)
go21	room(2)	room(1)
go32	room(3)	room(2)
go23	room(2)	room(3)
go42	room(4)	room(2)
go24	room(2)	room(4)

Table 1: RL-TOPs used for gridworld experiments.

the experiments detailed in this paper, the ϵ-greedy exploration algorithm (Thrun, 1992) was used, with $\epsilon = 0.1$ (i.e. at each step a random exploratory action is chosen with probability 1 in 10.)

The result of the executed action, in terms of changes in the high-level state description, is used by the **Reinforcement Schema** to determine the reinforcement feedback, r, to provide to the Learner. This unit determines whether, in terms of the its pre-image and post-condition, the RL-TOP has succeeded or failed. If the post-condition has become true, then the TOP has succeeded, and a reward of $r = +1$ is returned. Otherwise, if the pre-image is no longer true, then the TOP has failed (by exiting its application space prematurely), and a punishment of $r = -1$ is returned. If neither of these is the case, then $r = 0$.

Combining the low-level state and action information, and the reinforcement signal provided by the Reinforcement Schema, the **Learner** then performs the appropriate update on the RL-TOP's Q-Module, according to whatever reinforcement learning algorithm is used. Then the process repeats, with the Plan Executor deciding which TOP to execute for the next time step, until the goal is achieved.

5 EXPERIMENTAL DOMAIN

Experimental work is currently under way to demonstrate the RL-TOPs architecture on an insectoid robot called Prometheus, aiming to get the robot to learn how to walk towards a beacon. Results from this platform are not yet available, so a simple simulated domain was constructed to demonstrate the system.

The simulation consists of an agent in a 30×21 gridworld, as shown in Figure 2. At the low-level, the agent can sense its position within the world (as an xy-coordinate) and has four actions available to it, to move north, south, east or west. Each action is guaranteed to succeed unless there is a wall in the way.

The world is divided into five rooms, labelled 0 through 4, and the agent's goal is to reach a particular one, from a randomly chosen starting position. The high-level state and action descriptions are all in terms of which room the agent occupies, given by the predicate `room(R)`.

For each of the experiments following, the agent was allowed to run for 400 trials, each starting at a random location in the world and finishing when the goal is achieved. The length of each trial, in terms of the total number of low-level actions performed, was recorded. Twenty such runs were performed for each algorithm presented, and the results are the average trial lengths over these twenty runs.

The measurement we are interested in comparing is the time taken to learn the task, that is, the number of primitive actions performed before the agent converged to an optimal (or near-optimal) policy. To this end, the graphs compare cumulative trial lengths for each experiment. The cumulative trial length is the sum of the lengths all trials up to and including the current one.

5.1 EXPERIMENT 1: MODULAR VS. MONOLITHIC

The first experiment demonstrates the improvement in performance of the modular RL-TOPs architecture over a simple monolithic reinforcement learner. The agent's goal is to reach room 4. The monolithic learner has a single Q-Module which covers the entire state space, whereas the modular learner has been provided with eight RL-TOP descriptions, corresponding

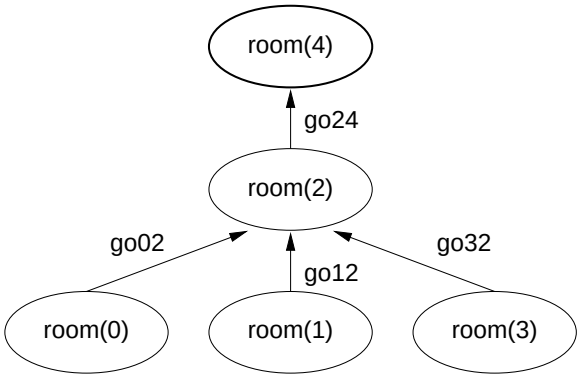
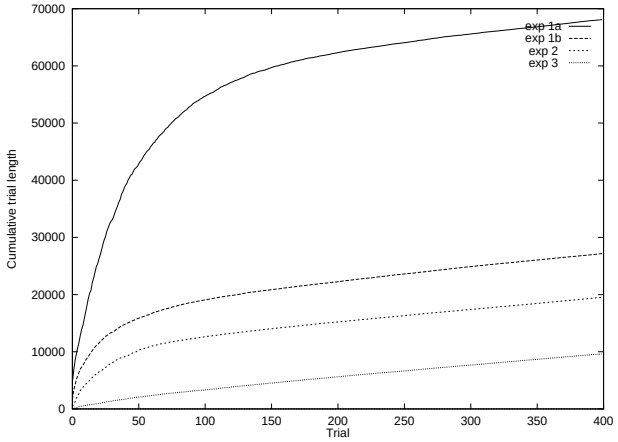


Figure 3: The TR-Tree for going to room 4.



Graph 1: Learning times for gridworld task using (1a) Monolithic learner, (1b) RL-TOPs, (2) RL-TOPs re-using previously learnt behaviours, (3) RL-TOPs using behaviours learnt with concurrent learning.

to movement from one room to an adjoining one, as listed in Table 1. The TR-Tree produced by the planner is shown in Figure 3.

The C-Trace learning algorithm was used in both cases, with the learning rate $\beta = 0.1$ and discount factor $\gamma = 0.9$. The monolithic learner was rewarded on success only, with a reinforcement value of 1. As with the RL-TOPs algorithm, the monolithic learner used the ϵ-greedy exploration algorithm, with $\epsilon = 0.1$.

Graph 1 shows the results of the two experiments. Both approaches converged to a nearly optimal policy within about 200 trials, but the monolithic learner took about 40,000 more steps to reach this point. A large part of this difference is established in the first 20 trials, which took the monolithic and modular systems, 25,372 and 11,253 steps respectively. This demonstrates the important difference between the two. In the early stages of learning, when the Q-function is still mostly zero, the only actions that provide any information are those that provide non-zero feedback. Since, in the monolithic case, rewards are few, the agent has nothing to direct it, and a large amount of time is spent aimlessly exploring the world, without learning anything.

In the modular system, however, the application spaces for individual behaviours are smaller, so the rewards (and penalties) are closer at hand. Thus random exploration is more likely to result in useful information more quickly, and learning is significantly faster.

5.2 EXPERIMENT 2: RE-USING BEHAVIOURS

Another advantage of the modular system over the monolithic is that the individual behaviours learnt in the modular trials can be re-used in a way that the monolithic policy cannot. In the next experiment, the same RL-TOPs from the previous experiment were used, with the Q-Modules saved from each run, in order to solve a new problem.

The goal is now to reach room 3. The new plan is shown in Figure 4. Notice that it includes two of the behaviours learnt in the previous experiment `go02` and `go12`. The other two behaviours, `go42` and `go23`, haven't been used before and still need to be learnt.

From the graph, we can see that a significant amount of time is saved in learning to perform this new task, compared to the previous one, which did not have the benefit of pre-existing behaviours. The reason for this is obvious: the agent does not need to waste time re-learning the `go02` and `go12` behaviours.

Still, a significant amount of time was taken up with learning the `go23` behaviour which would appear to be redundant. Although the agent has never performed this behaviour before, it has nevertheless spent a lot of time in room 2 in the previous experiment, albeit while executing a different behaviour. Common sense suggests that this prior experience should be of some use in learning the new behaviour more quickly. Is it possible to make use of information gathered while executing one behaviour in order to learn another? We

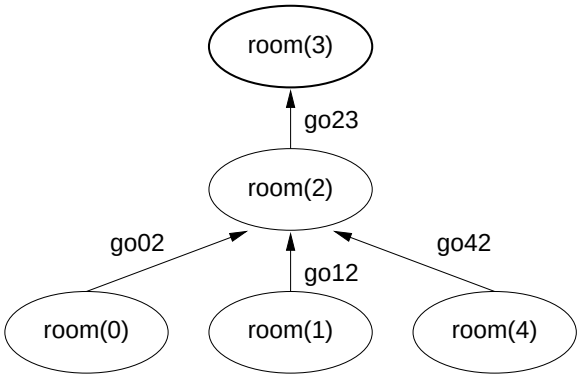


Figure 4: The TR-Tree for going to room 3.

address this question in the next section of this paper.

6 CONCURRENT LEARNING: MAKING BETTER USE OF EXPERIENCE

At this point, one under-appreciated feature of certain RL algorithms comes to our aid. Algorithms such as Q-Learning and C-Trace (but not SARSA) are *off-policy* learners, which means that they sequence of actions presented to the learner do no have to correspond to an actual execution of the policy (Sutton & Barto, 1998). It is even possible to learn one behaviour while executing a quite different one, so long as their application spaces overlap.

This technique, called *concurrent learning* can be added to the RL-TOPs architecture by a simple modification to the Learner module. Rather than just updating the Q-Module of the currently active TOP, the Learner examines the RL-TOP Library and selects all the behaviours which are eligible to be updated. This includes any behaviour the pre-image of which was satisfied before the most recent action was performed. Thus, to use the simulation above as an example, if the agent executes some action in room 2, then, regardless of the result of the action, all those behaviours which have `room(2)` as their pre-image, will be eligible to be updated.

The Learner then consults the Reinforcement Schema for each behaviour separately, to find out the reinforcement value for that particular TOP. For some, the action just executed may comprise success, for others failure, and for others neither of the two. The Learner uses the reinforcement value for each TOP, to update that TOP's Q-Module. Then execution proceeds as usual.

This technique should significantly speed up learning more than one task, because it makes more effective use of experience gained.

6.1 EXPERIMENT 3: CONCURRENT LEARNING

To demonstrate the benefit of concurrent learning the two previous experiments were repeated, but this time with all eligible behaviours being learnt concurrently. First the agent did 400 trials with room 4 as its goal. Then, using the same learnt behaviours, the goal was changed to room 3. Graph 1 shows the results of this run. Compare these to the results of experiment 2, which had the same goal, but did not use concurrent learning. The concurrent system converged in very little time at all. The behaviour `go23` was almost completely optimised before it was even run. The only behaviour to be learnt was `go42`, because the agent had had no prior experience with performing any actions in room 4.

7 RELATED WORK

In addition to those already mentioned, other hierarchical learning/planning systems of note include Singh's Compositional Q-Learning system (Singh, 1992), which learns a Q-function for a complex problem by constructing a *gating module* which selects an appropriate lower-level behaviour at each step; and the work of Precup et al. (Precup, Sutton, & Singh, 1997), which extends standard dynamic programming techniques to be able to use macro actions (behaviours) as well as primitive actions in their policies. Both of these systems assume that the behaviours that are used are already fully specified, perhaps by earlier learning runs.

Benson has produced a system that is complementary to that presented here. His TRAIL (Benson, 1996) architecture takes an existing set of actions or behaviours and, by guided experiments, learns appropriate TOP descriptions. It may be possible to combine that work and this, to produce a system in which learnt information goes in both directions, refining both the behaviours and the model.

8 CONCLUSION

As has been demonstrated, modular decomposition is an effective way to improve the speed of reinforce-

ment learning algorithms. The Reinforcement Learnt Teleo-operators (RL-TOPs) architecture, combining low-level reinforcement learning with high-level symbolic planning, is an elegant and effective way of expressing this decomposition. The system allows the automatic construction of appropriate hierarchies of learnt behaviours to solve a give problem, and provides a means of re-using behaviours learnt in one task, for solving another. With the addition of concurrent learning of multiple behaviours, this can greatly improve learning times over a variety of problems.

A limitation of this system is that the policy learnt is sub-optimal because the agent cannot "cut corners" between behaviours. Work is in progress to find a way to allow the agent to benefit from the domain information given by the task decomposition, while still being able to converge eventually to an optimal policy.

Another avenue for future research would be to investigate the question of what to do when the programmer-specified TOPs are insufficient to find a path to the goal. Possibly the system could be extended so as to postulate its own new behaviours in this state. However, this is likely to be a very difficult problem.

Acknowledgements

We would like to gratefully acknowledge the assistance of Christina Cook in developing the ideas contained in this paper.

References

Benson, S. (1996). *Learning Action Models for Reactive Autonomous Agents*. Ph.D. thesis, Department of Computer Science, Stanford University.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, *RA-2*(1), 14–23.

Covigaru, A. A., & Lindsay, R. K. (1991). Deterministic autonomous systems. *AI Magazine*, *12*(3), 110–117.

Dayan, P., & Hinton, G. E. (1992). Feudal reinforcement learning. *Advances in Neural Information Processing Systems*, *5*, 271–278.

Dietterich, T. G. (1997). Hierarchical reinforcement learning with the MAXQ value function decomposition. Tech. rep., Computer Science Department, Oregon State University.

Dorigo, M. (1996). Editorial: Introduction to the special issue of learning autonomous robots. *IEEE Transactions on Systems, Man and Cybernetics*, *26*(6).

Kaelbling, L. P. (1993). Hierarchical learning in stochastic domains: Preliminary results. In *Proceedings of the 10th International Conference on Machine Learning*. Morgan Kaufmann.

Mahadevan, S., & Connell, J. (1992). Automatic programming of behaviour-based robots using reinforcement learning. *Artificial Intelligence*, *55*(2-3).

Matarić, M. J. (1996). Behaviour based control: Examples from navigation, learning and group behaviour. *Journal of Experimental and Theoretical Artificial Intelligence*.

Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, *1*, 139–158.

Pendrith, M. D., & Ryan, M. R. K. (1996). Actual return reinforcement learning versus temporal differences: Some theoretical and experimental results. In *Proceedings of the 13th International Conference on Machine Learning*. Morgan Kaufmann.

Precup, D., Sutton, R. S., & Singh, S. (1997). Planning with closed-loop macro actions. In *Proceedings of the AAAI Fall Symposium on Model-directed Autonomous Systems*.

Singh, S. P. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, *8*(3), 323–340.

Singh, S. P., & Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Thrun, S. B. (1992). The role of exploration in learning control. In White, D., & Sofge, D. (Eds.), *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*. Van Nostrand Reinhold.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Cambridge.