Patching Approximate Solutions in Reinforcement Learning

Min Sub Kim[1] and William Uther[2]

[1] ARC Centre of Excellence for Autonomous Systems, School of Computer Science and Engineering, University of New South Wales, Sydney NSW 2052, Australia
[2] National ICT Australia, Sydney NSW 2052, Australia
msk@cse.unsw.edu.au, william.uther@nicta.com.au

Abstract. This paper introduces an approach to improving an approximate solution in reinforcement learning by augmenting it with a small overriding patch. Many approximate solutions are smaller and easier to produce than a flat solution, but the best solution within the constraints of the approximation may fall well short of global optimality. We present a technique for efficiently learning a small patch to reduce this gap. Empirical evaluation demonstrates the effectiveness of patching, producing combined solutions that are much closer to global optimality.

1 Introduction

Approximations are widely used in reinforcement learning to cope with large state spaces. The potential advantages offered by approximations include reduced storage requirements and faster learning than a flat solution. The main drawback is that it may be impossible to represent the globally optimal solution, and the best solution within the constraints of the approximation may be arbitrarily worse than global optimality.

In this paper we discuss a technique for learning a small patch, which, when combined with an approximate solution to a reinforcement learning problem, produces performance much closer to the global optimal. This is motivated by the observation that the sub-optimality of many approximate solutions may be attributed to sub-optimal behaviour in small but important regions of the state space. Augmenting the approximate solution with an overriding patch can overcome the sub-optimality in these regions while retaining the benefits of approximation elsewhere.

2 Background

We adopt the usual reinforcement learning setting of finite Markov Decision Problems with discrete time steps [1], with the following notation. A Markov Decision Problem M is a 5-tuple $< S, A, P, R, S_0 >$ where S is a finite set of states, A is a finite set of actions, $P(s, a, s')$ is the probability of reaching state s' after executing action a in state s, $R(s, a)$ is the immediate reward received

J. Fürnkranz, T. Scheffer, and M. Spiliopoulou (Eds.): ECML 2006, LNAI 4212, pp. 258–269, 2006.

for executing action a in state s, and $S_0(s)$ is the probability that M starts in state s.

The objective is to learn a policy $\pi : S \rightarrow A$ that optimises some measure of future reward. In this paper, we will use expected undiscounted reward: the expected sum of reward from following the policy until reaching a terminal state. However, the methods discussed are also applicable with discounting.

We use the action-value or Q function [2] to represent the expected value of a policy. Specifically, for state-action (s, a) and policy π, $Q^\pi(s, a)$ is defined using the Bellman equation:

$$Q^\pi(s, a) = R(s, a) + \sum_{s' \in S} P(s, a, s')Q^\pi(s', \pi(s')) \tag{1}$$

For learning the patch, we adapt prioritised sweeping [3], a model-based reinforcement learning algorithm that efficiently orders backups using a priority queue. At each step, the most recent state-action is promoted to the top of the backup queue. Then, before the next action is taken, a certain number of state-actions are removed from the top of the queue and processed one at a time. Processing a state-action consists of updating its Q value, and adding its predecessors to the backup queue, with priority equal to the expected change in Q value. This has the effect of concentrating computation where the Q function is changing most rapidly.

3 Related Work

Patching starts with an approximate solution and incrementally learns over it, an approach shared by many other methods. One of the earliest on-line algorithms to use this approach was Learning real-time A* [4], a real-time search algorithm. A heuristic cost function serves as the initial approximation, and as the agent searches the problem, it is incrementally overridden by a revised cost function. Real-time dynamic programming [5] generalises Learning real-time A* to Markov Decision Problems. However, both algorithms assume that storage is allocated for all states visited in practice, which becomes intractable over time for large problems.

An alternative approach is to initialise the solution with the approximation and then learn over it directly, instead of incrementally building a partial override. Naively seeding the value function in this manner may cause learning to be slower than starting from scratch [6]. However, careful application has been shown to be capable of accelerating learning [7].

A related method from multi-agent learning is Sparse cooperative Q-learning [8]. In this approach, the value function is approximated by agent-wise decomposition for some states, but depends on the entire joint state for others. This kind of partially abstract, partially flat value function is similar to that produced by patching, although patching is not limited to this particular type of decomposition. Coordination dependencies are specified by the user in this

algorithm; Utile coordination [9] is an extension of Sparse cooperative Q-learning that detects coordination dependencies automatically.

4 Patching in Reinforcement Learning

4.1 A Small Example

To illustrate the basic concepts and motivation for patching, consider the problem shown in Fig. 1. Two actors, A and B, are initially placed randomly on two separate paths with the goal of reaching Home.

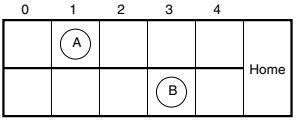

Fig. 1. A small coordination problem.

At each time step, each actor may either Move one cell to the right if it is not at Home, or Wait in its current position. These actions are deterministic. Each actor contributes a reward of 10 if it Moves to Home on that time step, or 0 if it is already at Home, or -1 otherwise. The total reward at each time step is the sum of the individual actor rewards, except if both actors Move to Home simultaneously, in which case the reward is 100. The task is undiscounted and terminates when both actors have reached Home.

An intuitive approximation for this problem is to assume that the actors are entirely independent. This divides the problem into two separate and identical sub-problems of one actor reaching Home individually. This type of decomposition is referred to as a *parallel decomposition* [10], because the problem is divided into sub-problems that "run in parallel". These sub-problems are much smaller and easier to solve separately than the whole problem. In this example, the optimal policy for one actor (defined over its individual state-action space only) is to Move on each step to reach Home and quickly as possible. Therefore, by combining the sub-problem solutions, the approximate solution suggests that both actors Move at every step until reaching Home.

The price paid for the reduction in solution complexity is that the solution is not optimal: the approximation is unable to represent that it is more profitable for the actors to cooperate on the last step before Home to collect the large combined reward. In terms of the policy value, it is easy to see that the approximate policy may have arbitrarily worse expected reward than optimal, depending on the value of the combined reward for reaching Home together. However, from the perspective of the policy, the approximate solution only requires modification at that last step. Ideally, we would like to bridge the gap to optimality, but without reverting to a flat Q table over the entire problem. This is exactly the aim of patching.

4.2 Specifying the Approximate Solution

We assume that the approximate solution is specified by:

- an approximate Q function, $\hat{Q}$;
- an approximate model of the transitions, $\hat{P}$, and rewards $\hat{R}$.

There are no strict requirements on the underlying representation of these functions, but in practice, it is expected that they will be compactly represented, *e.g.* $\hat{Q}$ may be a hierarchically decomposed Q function. We assume that these functions are pre-computed and fixed throughout patch learning.

For our example problem, $\hat{Q}$ is the sum of the corresponding single actor Q values: the expected reward for both actors reaching Home is estimated as the expected reward for the two actors reaching Home separately. This can be calculated on demand by looking up the corresponding entries in the single actor Q table. $\hat{P}$ and $\hat{R}$ are calculated similarly using their single actor counterparts.

4.3 Patching the Q Function

Patching aims to improve on the policy defined by $\hat{Q}$ by overriding some values in $\hat{Q}$. This override combines with $\hat{Q}$ to form the Q function representation used by patching.

Definition 1. *The* Q function patch, $Q_{\text{patch}} : S \times A \to \Re$ *is a partial function that overrides some values of* $\hat{Q}$*. Then, for any state-action pair* (s, a)*:*

$$Q(s,a) = \begin{cases} Q_{\text{patch}}(s,a) & \text{if } Q_{\text{patch}}(s,a) \text{ defined} \\ \hat{Q}(s,a) & \text{otherwise} \end{cases} \tag{2}$$

The "default" choice of representation for Q_{patch} is a dynamically sized hashtable over flat state-action pairs, holding only as many entries as are added to it. Q_{patch} may also employ abstractions, but this requires careful design: it needs to have sufficient representational power to cover sub-optimalities caused by approximations used in $\hat{Q}$, but over-generalising may make the solution worse. In this representation, updates to the Q function are made by adding or updating entries in Q_{patch}, overriding the value in $\hat{Q}$.

Patches to cover inaccuracies in $\hat{P}$ and $\hat{R}$ are defined analogously, and have similar conventions for partial override of $\hat{P}$ and $\hat{R}$. We omit details due to lack of space; full details are presented in an accompanying technical report [11].

4.4 Seeding the Patch

Having decided how the patch is represented, the next problem is to decide which action values should be added to the patch.

Definition 2. *The* patch seed predicate*, defined over state-actions, indicates the starting points for* Q_{patch}*, from which it will grow.*

We require the user to supply the patch seed predicate. A reasonable strategy for seeding the patch is to focus on the parts of the problem where $\hat{Q}$ may lack sufficient representational power, or where structural assumptions and abstractions used in $\hat{Q}$ may over-generalise. In general, patch seeding is not intended to be an exact listing of sub-optimalities in the approximation, but allows the user to suggest regions of the problem that deserve attention, instead of growing the patch blindly over the entire state-action space. An automatic method for seeding the patch that we use in our experiments is to detect inaccuracies in the model, discussed in Sect. 5.4. This is appropriate when $\hat{Q}$, $\hat{P}$, and $\hat{R}$ all depend on the same structural assumptions for approximation.

For our example, the assumption made when constructing the approximation was that the actors are entirely independent. However, when both actors Move to Home simultaneously, they will receive a combined reward of 100 instead of the sum of individual rewards predicted by $\hat{R}$ (+10 for each actor, for a total of 20). Therefore, this state-action is a patch seed, indicating that $\hat{Q}$ may be inaccurate around this state-action, and therefore the policy may be improved by patching around this state-action.

5 Learning the Patch

5.1 Unbounded Patching

With the initial approximation and patch seed predicate set, we now need an algorithm to learn Q_{patch}. Intuitively, we want to improve the policy defined by $\hat{Q}$ by adding override values to Q_{patch}, starting from the areas of interest suggested by the seed predicate.

Unbounded patching directly adapts prioritised sweeping for this purpose as follows. We follow the policy according to the current Q values as per usual. Then, if the most recent state-action is a patch seed according to the seed predicate, it is added to the backup queue. It then proceeds as per prioritised sweeping, by processing entries from the backup queue and making model-based Q function updates, with the difference that updates are made by adding or updating values in Q_{patch}. This has the effect of quickly growing Q_{patch} from the patch seeds through predecessors, adjusting the policy as it proceeds.

Eventually, unbounded patching will add all ancestors of all patch seeds to Q_{patch}, effectively reverting to prioritised sweeping over the flat problem. This is not unexpected, since unbounded patching is just prioritised sweeping with the first entries to the backup queue determined by patch seeds, combined with partial override by Q_{patch}. We need heuristics to bound patch growth while still repairing the policy where required.

5.2 Policy Bounding

Unbounded patching propagates changes in value from the seed points through predecessors. In our example problem, a lot of these changes in value do not affect the policy. An example is both actors Waiting in their current position: this

is equivalent to a null action and is not optimal in any state, but nevertheless, unbounded patching will patch it as long as it is a patch seed ancestor. Consequently, values are added to Q_{patch} without actually improving the resulting behaviour.

Policy bounding adds the restriction that patch values are added only when they immediately affect the current greedy policy (including the current set of Q_{patch} values). This can be seen as using immediate change in the policy as a heuristic to decide whether growing the patch is still effective. Under policy bounding, a proposed update for state-action (s, a) that is not currently in Q_{patch} is accepted if either the greedy action at s would change, or if a is the greedy action at s and has a non-zero probability of self-transition[1]. This condition is checked both when entries are added to and removed from the backup queue.

Policy bounding tends to constrain patching to only local adjustments around patch seeds. This usually keeps patch sizes smaller, but also limits patching to local policy repair only. In general, if the approximate solution requires correction on a more global scale, then policy bounding will either ignore some corrections or be ineffective in reducing Q_{patch} growth.

5.3 Utility Bounding

If there are hard limits on storage, policy bounding alone may not be sufficient. In this case, we would like to obtain the greatest improvement possible from the limited storage. One way to do this is to rank entries in Q_{patch} according to some measure of usefulness, so that the least useful values can be discarded if necessary.

Utility bounding implements Q_{patch} as a priority queue with fixed capacity specified by the user. Entries are prioritised by the absolute difference between the patched value and the corresponding value in $\hat{Q}$, *i.e.*, priority will be highest where $\hat{Q}$ is least accurate. This can be seen as using estimated Q function error as a heuristic measure of usefulness of entries in Q_{patch}. If Q_{patch} exceeds capacity, the state-action with lowest priority is dropped, reverting the Q function to $\hat{Q}$ for that state-action.

5.4 Patching the Model

Patching relies on the model to calculate the adjusted values for Q_{patch}. As with patching for the Q function, we augment the provided estimates $\hat{P}$ and $\hat{R}$ with partial patches, and avoid building the entire flat model by patching only the inaccuracies in the estimates. We omit details due to lack of space, but sketch the procedures briefly. Full details are presented in an accompanying technical report [11].

Inaccuracies in the approximate transition and reward models are detected with the χ^2 and Kolmogorov-Smirnov statistical tests. Transition and reward

[1] This is required because greedy actions re-use their own value to calculate their updated value.

samples are collected during learning, and compared to $\hat{P}$ or $\hat{R}$ once enough samples have been observed. If the test shows a significant difference between the distributions, then that state-action is patched with the observed samples, and future references to that state-action refer to the patched distribution rather than $\hat{P}$ or $\hat{R}$. Storage for sampling is kept limited by selective sampling, directing storage to those transitions experienced most frequently in practice.

6 Experiments

We evaluate patching on two domains, to examine the effectiveness of patching in bridging the gap to global optimality. We compare patching against two instances of prioritised sweeping:

- *Prioritised sweeping from scratch*: All Q values are initialised to 0, and the agent's estimates of P and R are built from scratch. This provides a lower baseline to determine whether the initial approximate solution is helpful.
- *Initialised prioritised sweeping*: All Q values are initialised to $\hat{Q}$'s values, and the approximate model is provided and updated by patching, as discussed in Sect. 5.4. The only differences between this algorithm and patching that affect the policy are patch seeding and the bounding heuristics, thus providing a measure of effectiveness of those aspects.

For all experiments, we use ε-greedy exploration, with $\varepsilon = 0.1$. A maximum of 2 state-actions were processed from the backup queue per step. All plots in this section show the average and standard deviation over 10 runs for each experiment.

6.1 Modified Taxi

The first set of experiments uses a modified version of Dietterich's taxi problem [12]. In this problem, a taxi agent in a 5-by-5 grid world (shown in Fig. 2) has the objective of delivering a passenger from a specially marked taxi stand to a destination taxi stand.

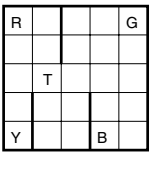

Fig. 2. The taxi problem. R, G, B, Y indicate the taxi stands, T indicates the taxi.

States are described by three variables: the taxi location, the passenger location, and the passenger destination. The taxi has stochastic navigation actions in the four compass point directions that move one cell in the intended direction with probability 0.8 and to the left or right of the intended direction with probability 0.1

each, subject to barriers that block movement (marked by thicker lines in Fig. 2). Two special actions are also available for picking up and putting down the passenger, effective only when the taxi is at the correct stand. The reward is -10 for failed pick-up or put-down actions, +19 on successful delivery, or -1 otherwise. The task is undiscounted and terminates on successful delivery of the passenger.

For the original taxi problem, MAXQ can be used to efficiently learn a compact hierarchical solution. Importantly, the task hierarchy includes navigation sub-tasks for each taxi stand. These sub-tasks are context independent with respect to the passenger location and destination – the optimal policy to navigate to a particular stand is the same regardless of the passenger.

We use the MAXQ solution as the initial approximation on a modified version of the taxi problem, and patch over it to handle the modifications. In the modified problem, 40 navigation state-actions in the middle row of the grid that are optimal in the original problem are modified. These modified actions have an irregular outcome in either P or R, such that the navigation sub-tasks are now *not* entirely context independent with respect to the passenger. Modified state-actions in P move in the intended direction with probability 0.1 and to the left or right of the intended direction with probability 0.45 each. Modified state-actions in R incur an unexpectedly costly reward of -6 with probability 0.8, and the usual -1 otherwise. These changes are deliberately conceived to be costly – an optimal policy for the original problem falls well short of global optimality when directly applied to the modified problem.

We apply patching in this domain as follows. $\hat{Q}(s, a)$ is calculated on demand from the MAXQ value function by finding the highest value path in the task hierarchy from the root node to the leaf node for a [2]. $\hat{Q}$ requires 632 values. $\hat{P}$ and $\hat{R}$ are initialised to the transition and reward models of the original taxi domain. Since $\hat{Q}$, $\hat{P}$, and $\hat{R}$ are all based on the original domain, it is reasonable to seed the patch at the transitions where $\hat{P}$ and $\hat{R}$ are found to be inconsistent with the modified domain. These inconsistencies in the model are detected using the procedures discussed in Sect. 5.4.

Figure 3(a) compares the expected reward for policy bounded patching and the two instances of prioritised sweeping. The solutions using the initial approximation have a clear head-start on prioritised sweeping from scratch, but the difference is reduced fairly quickly, and all algorithms reach policies of similar quality. In terms of storage, policy bounded patching settles with Q_{patch} coverage of approximately one third of the state-action space, requiring less total storage than the other algorithms.

If capacity for Q_{patch} is undersized, we can expect that the policy will consequently be worse. Figure 3(b) shows the expected reward for patching with policy and utility bounding, with Q_{patch} capacities of 800, 900, and 1,000 (26.7%, 30%, and 33.3% of the state-action space). As Q_{patch} capacity is reduced, the policy deteriorates, both in terms of expected reward and consistency.

Table 1 summarises the results for this domain.

[2] Subject to the termination predicates in the hierarchy – each sub-task in the path must be valid at s.

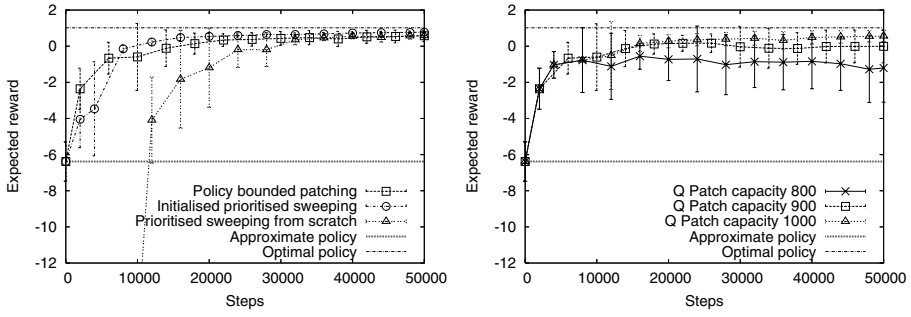

Fig. 3. Results for the modified taxi domain. Expected reward was determined by calculating the policy value, and averaging over the initial state distribution.

Table 1. Summary of results for the modified taxi domain. Statistics were taken at the end of 50,000 steps for all algorithms. For patched solutions, the total size is listed as the size of $\hat{Q}$ plus the size of Q_{patch}.

Solution	Expected reward	# Q values
Initial approximation ($\hat{Q}$)	-6.38 ± 1.09	632
Optimal flat	1.02	3000
Initialised prioritised sweeping	0.78 ± 0.07	3000
Prioritised sweeping from scratch	0.58 ± 0.19	3000
Policy bounded patching	0.58 ± 0.32	632 + 1035.60 ± 54.62
Policy and utility bounded patching		
– with Q_{patch} capacity 800	-1.21 ± 1.90	632 + 800
– with Q_{patch} capacity 900	-0.01 ± 0.91	632 + 900
– with Q_{patch} capacity 1000	0.56 ± 0.31	632 + 989.70 ± 16.64

6.2 Multi-taxi

The second set of experiments will examine patching on the multi-taxi problem: the grid remains the same as in the original taxi problem, but there are now two taxis and two passengers.

The multi-taxi problem is approximately equal to two instances of the original taxi problem running in parallel, but with some differences. Most importantly, the taxis are subject to collisions with each other, in which case neither taxi location is changed. A taxi may pick-up either passenger, but only one at a time. The reward is decomposed by passengers: at each time step, a reward of -1 is received for each undelivered passenger, plus -10 for each failed pick-up or put-down action. In addition to the action set from the original taxi problem, each taxi also has a null action for staying in place. The task is undiscounted and terminates when both passengers have been successfully delivered.

We apply patching in this domain as follows. $\hat{Q}$ is calculated by using a solution for the task of delivering one passenger with one taxi, requiring 4,200 values.

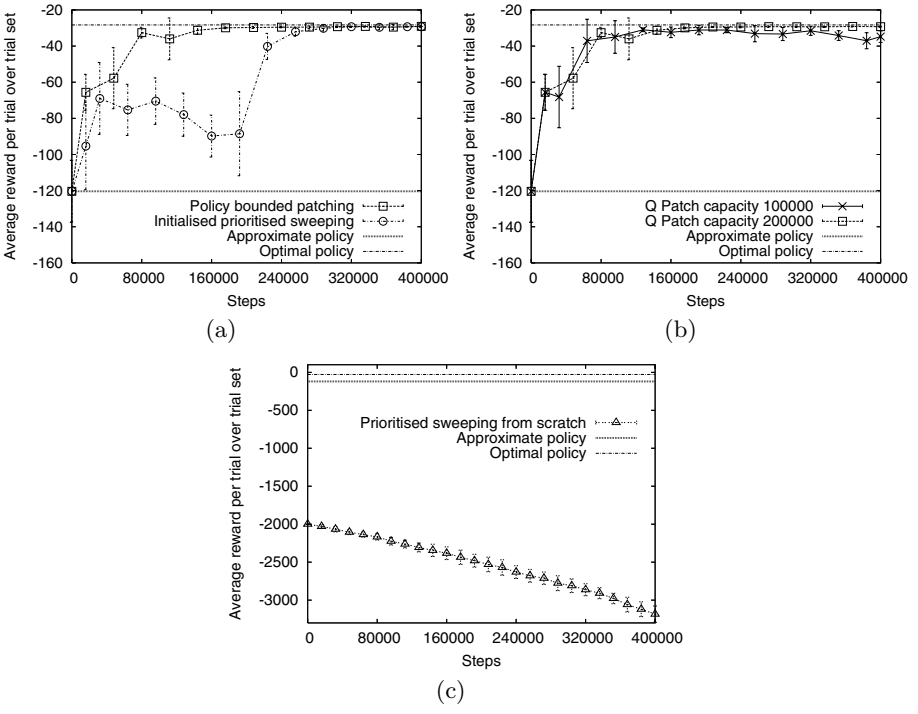

Fig. 4. Results for the multi-taxi domain. Average reward per trial was determined by evaluating the policy on a random test set of 1,000 initial states, fixed for each experiment run but different for separate runs. A maximum trial length of 1,000 steps was imposed for evaluation.

A hand-crafted allocation function determines allocation of taxis to passengers. Given an allocation, the expected reward for the two deliveries is estimated as the sum of the expected reward for the individual deliveries, *i.e.* assuming that the two taxis are entirely independent. $\hat{P}$ and $\hat{R}$ calculated similarly under the same assumption. Patch seeds are found by detecting inaccuracies in $\hat{P}$, which occur when the taxis collide. Both patching and prioritised sweeping make use of symmetry between the taxis to accelerate learning.

Figure 4(a) compares policy bounded patching and initialised prioritised sweeping. In this domain, patch seeding and bounding results in a noticeable difference in early performance – policy bounded patching reduces the gap to global optimality much faster than initialised prioritised sweeping by focusing updates to where the policy immediately requires correction.

In terms of storage, policy bounding alone does not appear sufficient in this domain to limit Q_{patch} growth. One reason for this is that most of the suboptimality in $\hat{Q}$ can be resolved by handling collisions, but further small improvements to the policy are possible, *e.g.* cooperative strategies that make positive use of collisions. Figure 4(b) plots the expected reward with both policy bounding and utility bounding, for Q_{patch} capacities of 100,000 and 200,000 (0.6%

and 1.1% of the state-action space). Combining both bounding heuristics makes efficient use of the limited storage, with little loss in policy value compared to patching without utility bounding.

Lastly, Fig. 4(c) shows the expected reward for prioritised sweeping from scratch. While all algorithms initialised with $\hat{Q}$ had learning curves between $\hat{Q}$ and the optimal solution, prioritised sweeping from scratch starts far below $\hat{Q}$, and proceeds to blindly explore the problem. Clearly, while the initial approximation is not perfect, it is a much more preferable starting point to nothing.

Table 2 summarises the results for this domain.

Table 2. Summary of results for the multi-taxi domain. Statistics were taken at the end of 400,000 steps for all algorithms. For patched solutions, the total size is listed as the size of $\hat{Q}$ plus the size of Q_{patch}.

Solution	Reward per trial	# Q values
Initial approximation ($\hat{Q}$)	-120.37 ± 17.12	4200
Optimal flat	-28.29 ± 0.28	17434200
Initialised prioritised sweeping	-29.26 ± 0.32	17434200
Prioritised sweeping from scratch	-3184.23 ± 107.44	17434200
Policy bounded patching	-29.10 ± 0.33	4200 + 379263.10 ± 3444.89
Policy and utility bounded patching		
– with Q_{patch} capacity 100000	-34.75 ± 3.21	4200 + 100000
– with Q_{patch} capacity 200000	-29.26 ± 0.37	4200 + 200000

7 Conclusions and Future Work

In this paper, we introduced an approach to reinforcement learning in which an approximate solution is taken as the starting point, and patched to improve performance beyond the constraints imposed by the approximation. We started with unbounded patching as a direct adaptation of prioritised sweeping to patching, and proposed policy bounding and utility bounding as two heuristics for bounding patch growth. Empirical results demonstrated the effectiveness of patching, producing near optimal solutions with limited storage, using two different types of underlying Q function approximations.

Future work will aim to apply patching to larger problems, with more sophisticated approximations and patch functions. We used patching in the scope of entire tasks, but it may be possible to apply patching in separate components of a decomposed solution, such as at various levels of a task hierarchy.

Acknowledgements

We thank Claude Sammut, Bernhard Hengst, Robert Fitch, and Malcolm Ryan for helpful feedback that assisted in developing the ideas in this paper. We also thank the anonymous reviewers for their helpful comments.

This research is supported by the Australian Research Council Centre of Excellence for Autonomous Systems. National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

1. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT Press (1998)
2. Watkins, C.J.C.H.: Learning from delayed rewards. PhD thesis, King's College, Oxford (1989)
3. Moore, A.W., Atkeson, C.G.: Prioritized sweeping: Reinforcement learning with less data and less real time. Machine Learning **13** (1993) 103–130
4. Korf, R.E.: Real-time heuristic search. Artificial Intelligence **42** (1990) 189–211
5. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artificial Intelligence **72** (1995) 81–138
6. Bowling, M., Veloso, M.: Reusing learned policies between similar problems. In: Proceedings of the AI*IA-98 Workshop on New Trends in Robotics, Padua, Italy (1998)
7. Taylor, M.E., Stone, P.: Behavior transfer for value-function-based reinforcement learning. In: The 4th International Joint Conference on Autonomous Agents and Multiagent Systems, ACM Press (2005) 53–59
8. Kok, J.R., Vlassis, N.: Sparse cooperative Q-learning. In: Proceedings of the 21st International Conference on Machine Learning, ACM (2004) 481–488
9. Kok, J.R., Hoen, P.J., Bakker, B., Vlassis, N.: Utile coordination: Learning interdependencies among cooperative agents. In: IEEE Symposium on Computational Intelligence and Games. (2005)
10. Boutilier, C., Dean, T., Hanks, S.: Decision-theoretic planning: Structural assumptions and computational leverage. Journal of Artificial Intelligence Research **11** (1999) 1–94
11. Kim, M.S., Uther, W.: Patching approximate solutions in reinforcement learning. Technical Report 0610, School of Computer Science and Engineering, University of New South Wales (2006)
12. Dietterich, T.: Hierarchical reinforcement learning with the MAXQ value function decomposition. Journal of Artificial Intelligence Research **13** (2000) 227–303