Patching Approximate Solutions in Reinforcement Learning
UNSW-CSE-TR-0610

Min Sub Kim
ARC Centre of Excellence for Autonomous Systems
School of Computer Science and Engineering
University of New South Wales
Sydney NSW 2052 Australia
msk@cse.unsw.edu.au

William Uther
National ICT Australia
Sydney NSW 2052 Australia
william.uther@nicta.com.au

May 10, 2006

Abstract

This report 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 an algorithm 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

A well known property of classical reinforcement learning algorithms is that they exhibit poor scalability to large problems. Large state spaces are common as the size of the state space is exponential in the number of state variables used to describe it.

Approximation is often used with reinforcement learning to cope with this exponential state space explosion. The potential advantages offered by approximations include reduced storage requirements, improved generalisation, and faster learning than a flat solution as the solution search space is constrained. The main drawbacks of approximation are that it may be impossible to represent the globally optimal solution, and the approximation method may fail to converge to the optimal solution within its constraints.

In this report, we discuss a technique for learning a small patch, which, combined with the approximate solution, produces performance much closer to global optimality. This is motivated by the observation that the sub-optimality of many approximate solutions may be attributed to poor 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 [Kaelbling et al., 1996, Sutton and Barto, 1998], with the following notation. A Markov Decision Problem M is a 5-tuple $< S, A, P, R, S_0 >$ with:

- S – a finite set of discrete states.
- A – a finite set of discrete actions. In general, the available actions may depend on the current state: $A(s)$ denotes the set of actions that may be executed at state s.
- $P : S \times A \times S \rightarrow [0, 1]$ – the transition probability distribution, where $P(s, a, s')$ is the probability of reaching state s' after executing action a in state s.
- $R : S \times A \rightarrow \Re$ – the reward function, where $R(s, a)$ is the immediate reward received for executing action a in state s. More generally, the reward may be a (possibly stochastic) function of the state, action, and next state; in this case, we take $R(s, a, s')$ as the expected value of the reward received for transitioning to state s' after executing action a in state s, and $R(s, a)$ as the weighted average of $R(s, a, s')$ over s':

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

- $S_0 : S \rightarrow [0, 1]$ – the initial state distribution, where $S_0(s)$ is the probability of starting in state s.

The choice of action to take is specified by a policy $\pi : S \rightarrow A$, where $\pi(s)$ is the action to take in state s. More generally, policies may also be stochastic, in which case the representation is changed to $\pi : S \times A \rightarrow [0, 1]$, where $\pi(s, a)$ is the probability of taking action a in state s.

The objective of reinforcement learning is to learn a policy that optimises some criterion based on expected future reward. A commonly used criterion is the expected sum of discounted reward:

$$E(\sum_{t=1}^{\infty} \gamma^{t-1} r_t) \tag{2}$$

where r_t is a random variable representing the reward received at time step t, and γ is a constant known as the discount factor.

The action-value or Q function [Watkins, 1989, Watkins and Dayan, 1992], $Q^\pi : S \times A \rightarrow \Re$ is used to represent the expected value of the policy π. Specifically, $Q^\pi(s, a)$ is defined as the value of taking action a in state s and following policy π thereafter, using the Bellman equation:

$$Q^\pi(s, a) = R(s, a) + \sum_{s' \in S} P(s, a, s') Q^\pi(s', \pi(s')) \tag{3}$$

The optimal Q function, Q^*, maximises this expected future value:

$$Q^*(s, a) = R(s, a) + \sum_{s' \in S} P(s, a, s') \max_{a' \in A(s)} Q^*(s', a') \tag{4}$$

Given an optimal Q function, the optimal policy π^* may be determined as:

$$\pi^*(s) = \arg \max_{a \in A(s)} Q^*(s, a) \tag{5}$$

For learning the patch, we adapt prioritised sweeping [Moore and Atkeson, 1993] (Algorithm 1). Prioritised sweeping is a model-based reinforcement algorithm that uses a priority queue to efficiently order updates. For each step taken, the most recently experienced state-action is immediately 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 backup 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 change in Q value. If a predecessor is already in the queue, then its priority is overridden if the new priority exceeds the old priority.

3 Related Work

Patching starts with an approximate solution and incrementally learns over it, an approach shared by many other methods. Key features that distinguish these methods include the way in which the initial solution is revised, and the solution representation.

Two of the earliest on-line algorithms to incrementally learn a partial overriding function over an initial heuristic were the real-time search algorithms Real-time A* and Learning Real-time A* [Korf, 1990]. These algorithms use a heuristic function over states as the initial approximation. Then, as the agent searches the problem on-line, it incrementally builds a revised cost function, overriding the heuristic for the states encountered in practice. The key intuition is that because the heuristic guides search initially, if the heuristic is good, then the number of states visited in practice will be small. Real-time dynamic programming [Barto et al., 1995] generalises Learning Real-Time A* to the Markov Decision Problem framework. These algorithms assume that storage is allocated for all states visited in practice, which becomes intractable over time without some form of function approximation.

Alternatively, instead of incrementally building a partial overriding function, another approach is to "seed" the solution with the approximation and then learn over it directly. Classical reinforcement learning algorithms make no assumption about how the Q function is initialised – this approach can be seen as deliberately initialising the Q function with particular values,

Algorithm 1: Prioritised sweeping, as given by Moore and Atkeson [1993].

```
1  begin
2      initialise empty backup queue;
3      repeat
4          s ← initial state chosen according to S_0;
5          while s is not terminal do
6              a ← chosen action according to Q and exploration policy;
7              execute a, observe reward r, and next state s';
8              update model with sample (s, a, r, s');
9              promote (s, a) to top of backup queue;
10             while backup queue not empty and allowed to do backups do
11                 (s, a) ← remove top of backup queue;
12                 Q(s, a) ← R(s, a) + γ Σ_{s'∈S} P(s, a, s') max_{a'∈A(s')} Q(s', a');
13                 foreach predecessor (s_p, a_p) of s do
14                     q ← R(s_p, a_p) + γ Σ_{s'∈S} P(s_p, a_p, s') max_{a'∈A(s')} Q(s', a');
15                     Δ ← | Q(s_p, a_p) − q |;
16                     if Δ > Δ_ε and ((s_p, a_p) not in backup queue or current priority of
                       (s_p, a_p) < Δ) then
17                         promote (s_p, a_p) in backup queue to priority Δ;
18                     end
19                 end
20             end
21             s ← s';
22         end
23     until finished;
24 end
```

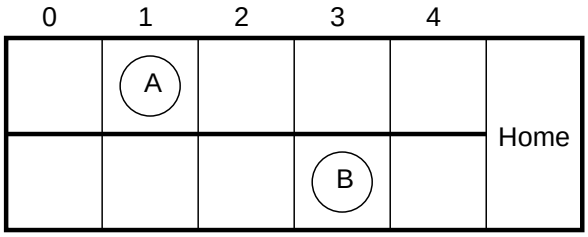


Figure 1: Small coordination example world.

such as Q values from a related task, and then learning over it as per usual. This approach must be implemented carefully – Bowling and Veloso [1998] report that naively seeding the Q function in this manner may cause learning to be slower than starting from scratch. However, as demonstrated by Taylor and Stone [2005] and Taylor et al. [2005], carefully initialising the Q function has been shown to be capable of accelerating learning.

A related method from multi-agent learning is Sparse cooperative Q-learning [Kok and Vlassis, 2004]. 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 in the form of a coordination graph [Guestrin et al., 2001]. Utile coordination [Kok et al., 2005] extends Sparse cooperative Q-learning by detecting these dependencies automatically, via statistical testing based on utile distinction [McCallum, 1995].

4 Patching in Reinforcement Learning

In this section, we describe the key building blocks for patching, illustrated with a small example problem.

4.1 A Small Example

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

States are described by two variables, representing the locations of the two actors separately. A state $s = (l_1, l_2)$ represents the configuration where actor A is at location l_1 and actor B is at location l_2, where l_i can take a value from the set of locations for one actor, $L = \{0, 1, 2, 3, 4, \text{Home}\}$.

Actions are also represented by two variables: at each time step, each actor that has not yet reached Home either Moves one location closer to Home or Waits at their current location; otherwise, if an actor is already at Home, that actor must Wait. An action $a = (a_1, a_2)$ represents actor A executing a_1 and actor B executing a_2, where a_i can be either Move or Wait. The transitions are deterministic and independent between the two actors.

The reward function is also deterministic and decomposed by actors: at each time step, each actor that has not yet reached Home receives a reward of -1, except if they Move to Home on that step, in which case the reward is 10. If an actor is already at Home, then the reward is 0. The total reward at each time step is the sum of rewards for both actors, except if both

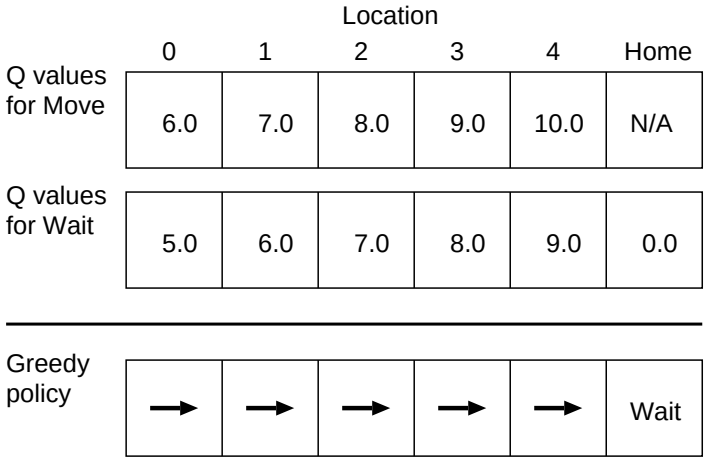


Figure 2: Optimal Q table and policy for a single actor for the small example domain. For the policy, the right-wards pointing arrow corresponds to Move. Note that Move is not a valid action when the location is Home.

actors Move to Home at the same time, in which case a different combined reward value of 100 is earned.

The task is undiscounted and terminates when both actors have reached Home.

A flat Q table for this problem would require 4 Q values for each state where neither actor is at Home, and 2 Q values for each state where one actor is at Home (since the action of the actor at Home is always Wait). This produces a total Q table size of $5 \times 5 \times 4 + 5 \times 2 \times 2 = 120$.

An intuitive decomposition to produce an approximate solution to this problem is to treat the actors individually. This divides the problem into two separate and identical sub-problems of one actor reaching Home individually. Boutilier et al. [1999] refer to this type of decomposition as a *parallel decomposition*, where a problem is divided into sub-problems that "run in parallel". We can construct a Q function for the task of one actor reaching home individually, $Q_s : L \times \{\mathsf{Move}, \mathsf{Wait}\} \rightarrow \Re$, and take the approximate Q function for the entire problem as the sum of these values for each actor. The intuitive justification for this approximation is that the actors are entirely independent except for the combined reward when both actors Move to Home at the same time. If the combined reward was not present, the approximate solution would be optimal.

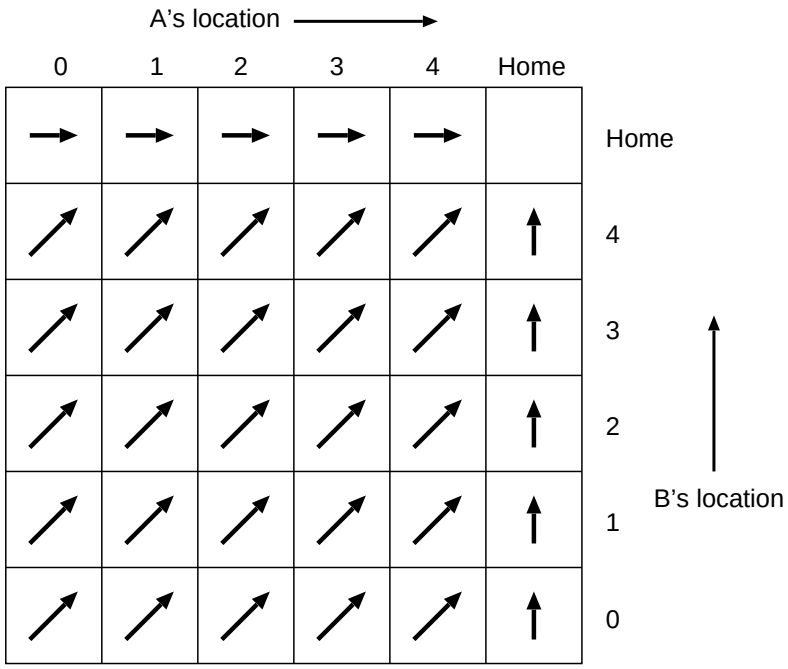
The representation of the approximation is far smaller than the flat Q table, since Q_s is re-used for both actors, and is defined over a much smaller domain. Specifically, Q_s requires 10 Q values [1], and since the approximate Q function is calculated dynamically using Q_s for both actors, its total storage requirement is also 10 Q values.

Figure 2 shows the optimal Q table and policy for a single actor for this domain. Assuming decomposition by actors, we can construct the approximate solution shown in Figure 3(a). This policy is *not* represented in tabular form – each policy action shown in Figure 3(a) is calculated dynamically by referring to the single actor Q table in Figure 2.

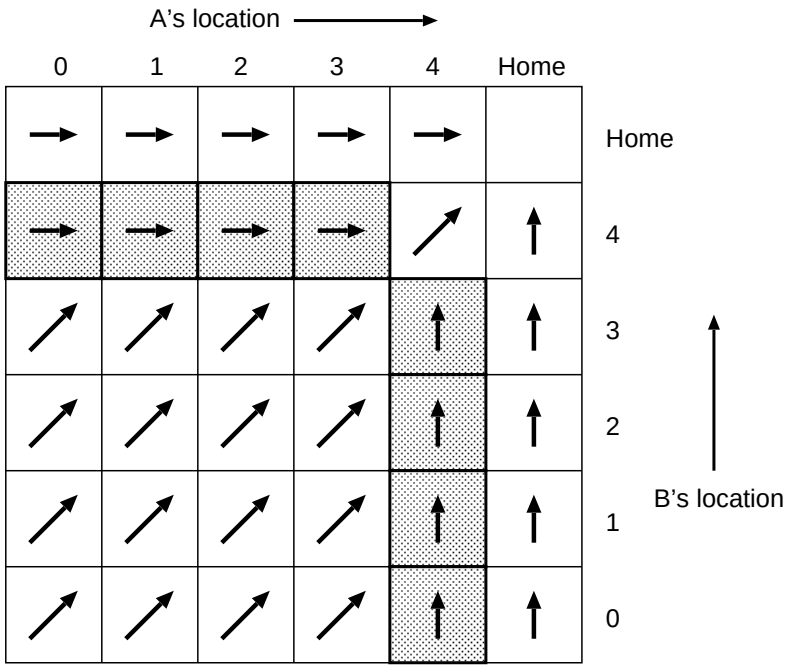
The optimal policy is shown in Figure 3(b), highlighting the states where the optimal policy differs from the approximate policy in Figure 3(a). While there are some differences in the policy, the optimal policy is represented using 120 Q values, as opposed to 10 Q values for the approximate solution.

Depending on the combined reward value at the last step when both actors Move to Home,

[1] Assuming $Q_s(\mathsf{Home}, .) = 0$ implicitly, since Home is terminal for a single actor.



(a) The approximate policy constructed by decomposing the problem by actors.



(b) The optimal policy, with shaded states indicating where the optimal policy differs from the approximate policy.

Figure 3: Comparison of the approximate and optimal policies for the small coordination example domain. Right-wards pointing arrows represent A executing Move and B executing Wait, diagonal arrows represent both actors executing Move, and up-wards pointing arrows represent A executing Wait and B executing Move. Note that the approximate policy is represented abstractly and requires far fewer values than the optimal policy.

the expected reward of the approximate solution may be arbitrarily worse than the optimal. When an actor is at the last step before Home, because the combined reward for reaching Home together is much larger than the rewards for reaching Home individually, it is more profitable to wait for its partner to co-operate to collect the combined reward. Because the approximate solution over-generalises, it is not able to represent this. However, we do not want to revert to solving the problem using a flat representation. The approximation gives us the optimal policy at most states, so we would like to exploit the benefits of an approximate solution for most of the problem, while overriding it with a small patch where it is sub-optimal.

4.2 Specifying the Approximate Solution

We now formalise the discussion up to this point. 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 abstract and compactly represented, *e.g.* $\hat{Q}$ may be a hierarchically decomposed Q function. This means that the approximate solution will have significantly lower storage requirements than a flat solution, as well as being easier to solve. We assume that these functions are pre-computed and fixed throughout patch learning.

For our example problem:

- $\hat{Q}$ is the sum of Q values for each actor individually using Q_s:

$$\hat{Q}((l_1, l_2), (a_1, a_2)) = Q_s(l_1, a_1) + Q_s(l_2, a_2) \tag{6}$$

- $\hat{P}$ is the product of the individual transition functions for each actor. Let $P_s : L \times \{\mathsf{Move}, \mathsf{Wait}\} \times L \to [0, 1]$ be the transition function for one actor. Then:

$$\hat{P}((l_1, l_2), (a_1, a_2), (l_1', l_2')) = P_s(l_1, a_1, l_1') \times P_s(l_2, a_2, l_2') \tag{7}$$

  Since there is no interaction in the individual transition functions for this domain, $\hat{P}$ is exact.

- $\hat{R}$ is the sum of expected individual actor rewards. Let $R_s : L \times \{\mathsf{Move}, \mathsf{Wait}\} \to \Re$ be the reward function for one actor. Then:

$$\hat{R}((l_1, l_2), (a_1, a_2)) = R_s(l_1, a_1) + R_s(l_2, a_2) \tag{8}$$

  $\hat{R}$ is exact for all state-actions except the combined reward when both actors Move to Home simultaneously.

Note that $\hat{Q}$, $\hat{P}$, and $\hat{R}$ for the example problem are all represented abstractly – each function is calculated on demand as required, by referring to corresponding entries in their single actor counterparts.

4.3 Patch Functions

Patching aims to improve on the greedy policy implied by $\hat{Q}$ by overriding some of $\hat{Q}$'s values. Together, they combine 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)*:*

$$Q(s,a) = \begin{cases} Q_{\text{patch}}(s,a) & \text{if } Q_{\text{patch}}(s,a) \text{ defined} \\ \hat{Q}(s,a) & \text{otherwise} \end{cases} \tag{9}$$

There are no strict requirements on the representation of Q_{patch}, so it is possible for Q_{patch} to also be an approximation. However, in order to improve on the greedy policy defined by $\hat{Q}$, Q_{patch} needs to have sufficient representational power to cover sub-optimalities caused by the approximations used in $\hat{Q}$. Conversely, over-generalisation may cause Q_{patch} to extend to cover too much of the state-action space, which may make the solution worse. The "default" choice of representation is to define Q_{patch} over flat state-action pairs with no abstraction.

Since Q_{patch} is a partial function, it only holds as many entries as are added to it. The statement $Q_{\text{patch}}(s,a) \leftarrow q$ has the operating convention:

- if an entry for (s,a) already exists in Q_{patch}, its value is overridden by q;
- otherwise an entry for (s,a) is added to Q_{patch} with the value q.

In practice, our implementation of Q_{patch} uses a dynamically sized hashtable, so it grows according to storage demand. The look-up procedure for the Q function is modified to use the value in Q_{patch} when it is present, and otherwise refer to $\hat{Q}$.

As well as overriding the Q function, it may be found that the approximations of the model, $\hat{P}$ and $\hat{R}$ are also inaccurate. Patches for the transition and reward models are defined analogously to Q_{patch}.

Definition 2. *The* transition function patch, $P_{\text{patch}} : S \times A \times S \to [0,1]$, *is a partial function that overrides some values of* $\hat{P}$, *with the property that if* $P_{\text{patch}}(s,a,s')$ *is defined, then it must be defined for all possible values of* s' *for those values of* s *and* a. *Then, for any state-action-state triple* (s,a,s'), *the learner's internal model of* P *is:*

$$P(s,a,s') = \begin{cases} P_{\text{patch}}(s,a,s') & \text{if } P_{\text{patch}}(s,a,s') \text{ defined} \\ \hat{P}(s,a,s') & \text{otherwise} \end{cases} \tag{10}$$

Definition 3. *The* reward function patch, $R_{\text{patch}} : S \times A \to \Re$, *is a partial function that overrides some values of* $\hat{R}$. *Then, for any state-action pair* (s,a), *the learner's internal model of* R *is:*

$$R(s,a) = \begin{cases} R_{\text{patch}}(s,a) & \text{if } R_{\text{patch}}(s,a) \text{ defined} \\ \hat{R}(s,a) & \text{otherwise} \end{cases} \tag{11}$$

As with Q_{patch}, there are no strict requirements on the representation of P_{patch} and R_{patch}, but they should have sufficient representational power to cover inaccuracies in $\hat{P}$ and $\hat{R}$. As with Q_{patch}, the "default" choice of representation here is over flat states and actions.

In practice, our implementation of P_{patch} uses a hashtable, with state-action pairs as the key and a list of pairs of successor states and observed frequencies as the value. We also adaptively

update the patch for the inverse of P. Its implementation is somewhat complicated so as to minimise storage requirements, and we defer its discussion to Section 5.3.3. For R_{patch}, we use a hashtable with state-action pairs as the key and the moving average observed reward as the value (*i.e.* the average reward value and the number of times the reward has been sampled for that state-action).

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

We leave the definition of the patch seed predicate open to the user. The patch seed predicate allows the user to provide hints as to where they suspect the approximation may be sub-optimal. In practice, the user will not know in advance exactly which state-actions may require patching, but may have some rough idea. For example, if $\hat{Q}$ is constructed using a relaxed version of the problem, the assumptions made in relaxing the problem may serve to suggest where $\hat{Q}$ may be inaccurate. Patch seeding is not intended as an exact listing of sub-optimalities in the approximation, but allows the user to suggest the regions of the problem that deserve attention instead of growing the patch blindly over the entire state-action space.

The patch seed predicate does not need to be constant – it may change over time as irregularities are detected in the problem domain. For our running example, we take the patch seed predicate to be true where the approximate model is inaccurate. Specifically, this inaccuracy will be detected on receiving the combined reward when both actors Move to Home simultaneously. Since the combined reward is different to the independent sum suggested by $\hat{R}$, this state-action will be detected as a patch seed point. These irregularities are detected automatically as the model is patched, making this a convenient patch seed predicate.

Some care needs to be taken when defining the patch seed predicate, because it is desirable to keep Q_{patch} relatively small. The "default" representation of Q_{patch} is over flat state-actions and is not compacted in any way. If the patch seed predicate is true for a large number of state-action pairs, it follows that Q_{patch} should cover most of the state-action space, requiring storage on par with a flat Q table representation.

5 Learning the Patch

We now describe our patch learning algorithms that use $\hat{Q}$, $\hat{P}$, $\hat{R}$, and the patch seed predicate to learn Q_{patch}, P_{patch}, and R_{patch}.

5.1 Unbounded Patching

The *unbounded patching* algorithm (Algorithm 2) directly adapts prioritised sweeping [Moore and Atkeson, 1993] to patching, by incorporating the patch functions and patch seeding. Other model-based reinforcement learning algorithms such as Real-time dynamic programming [Barto et al., 1995], the Dyna class of algorithms [Sutton, 1990, 1991a,b], and Queue-Dyna [Peng and Williams, 1993] may also be similarly adapted to learning Q_{patch}.

In prioritised sweeping, *all* state-actions are considered eligible for update – to adapt this to patching, we need to re-define this condition, so as to learn Q_{patch} around the patch seeds. For the unbounded patching algorithm, an experienced state-action (s, a) is considered eligible for update and promoted to the top of the backup queue (lines 11–13) if either:

Algorithm 2: The unbounded patching algorithm.

```
1  begin
2      initialise empty backup queue;
3      initialise Q_patch, P_patch, R_patch as empty functions;
4      repeat
5          s ← initial state chosen according to S_0;
6          while s is not terminal do
7              a ← chosen action according to Q and exploration policy;
8              execute a, observe reward r, and next state s';
9              update_P_patch(s, a, s');
10             update_R_patch(s, a, r);
11             if seed(s, a) or (s, a) ∈ Dom(Q_patch) or
               ∃s' [ P(s, a, s') > 0 ∧ (s', .) ∈ Dom(Q_patch) ] then
12                 promote (s, a) to top of backup queue;
13             end
14             while backup queue not empty and allowed to do backups do
15                 (s, a) ← remove top of backup queue;
16                 Q_patch(s, a) ← R(s, a) + γ Σ_{s'∈S} P(s, a, s') max_{a'∈A(s')} Q(s', a');
17                 foreach predecessor (s_p, a_p) of s do
18                     q ← R(s_p, a_p) + γ Σ_{s'∈S} P(s_p, a_p, s') max_{a'∈A(s')} Q(s', a');
19                     Δ ← | Q(s_p, a_p) − q |;
20                     if Δ > Δ_ε then
21                         if (s_p, a_p) is in backup queue then
22                             increase priority of (s_p, a_p) in backup queue by Δ;
23                         else
24                             insert (s_p, a_p) into backup queue with priority Δ;
25                         end
26                     end
27                 end
28             end
29             s ← s';
30         end
31     until finished;
32 end
```

- (s, a) satisfies the patch seed predicate, *i.e.* $seed(s, a)$ is true; or
- (s, a) is already patched, *i.e.* $(s, a) \in Dom(Q_{\text{patch}})$; or
- (s, a) is a predecessor of some state s' in the domain of Q_{patch}, or formally, if $\exists s' \ [\ P(s, a, s') > 0 \wedge (s', .) \in Dom(Q_{\text{patch}})\ ]$.

Note that Q_{patch} is initially empty, so the first eligible state-action for update will be a patch seed.

Between each step of execution, a certain number of state-actions are removed from the backup queue and processed one at a time (lines 14–28). Processing a state-action consists of calculating its model-based value, updating Q_{patch} with this value, and then adding its predecessors to the backup queue. Predecessors are added to or have their priority increased in the backup queue as long as their predicted change in Q value is greater than a small threshold, Δ_ϵ [2]. Note that modifications to the Q function are restricted to modification of values already in Q_{patch} or extending Q_{patch} via predecessors of state-actions already in Q_{patch} – this restricts the value updates to propagate from the patch seeds.

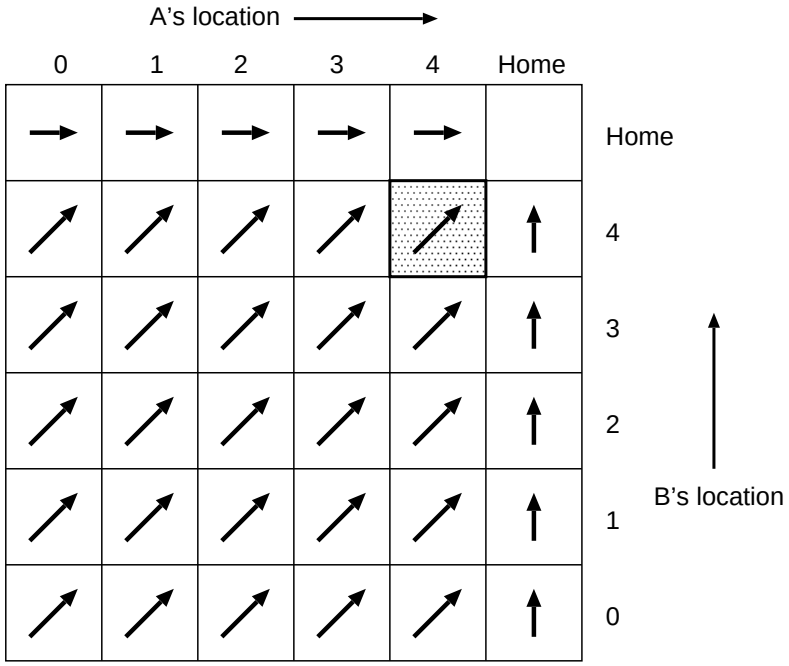
As with the original prioritised sweeping algorithm, the backup queue is kept sorted by priority based on the change in Q value. However, unlike the original algorithm, if a predecessor is already in the backup queue, its priority is increased by the predicted change in Q value. This method of adjusting priorities appears to have been first proposed by Andre et al. [1998], and guarantees that every state-action requiring update will eventually be processed. The prioritised sweeping mechanism of efficiently propagating updates through predecessors is well-suited to patching. Its effect is that the change in Q value at a patch seed is quickly propagated through its predecessors to adjust the policy accordingly.

The algorithm is called unbounded patching because propagation of patch values through predecessors is not restricted. Once a patch seed has been added, all of its predecessors become eligible for patching, followed by the next generation of predecessors, and so on. Therefore, if all state-actions are ancestors of patch seeds, and there are no bounds on patch capacity, unbounded patching will eventually expand the Q function patch to cover the full state-action space and converge to the optimal solution. In short, 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}.

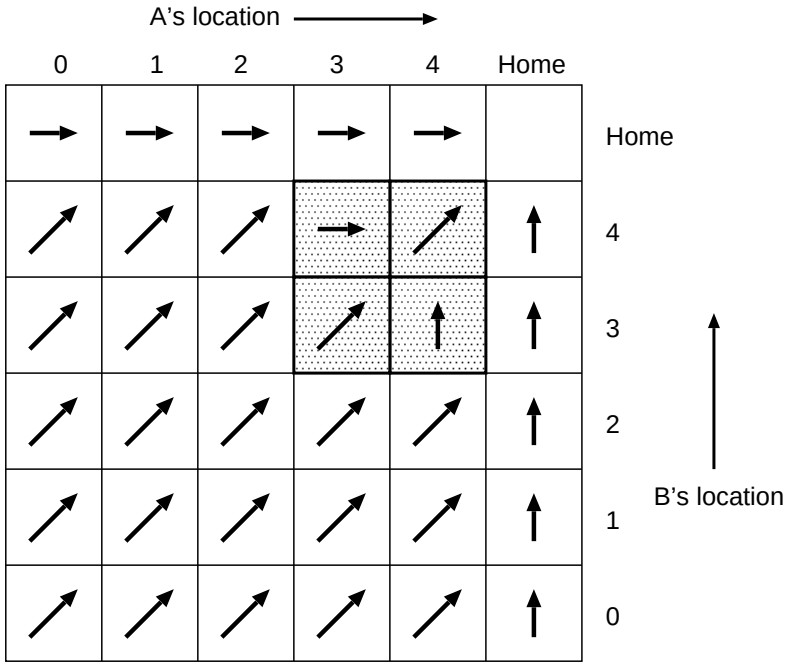
To better illustrate the algorithm, consider its application to our running example. We start with the policy shown in Figure 4(a), where the state-action where both actors Move to Home simultaneously to collect the combined reward has just been detected as a patch seed at line 11 of the algorithm. Because it is a patch seed, ((4,4), (Move,Move)) is placed on the top of the backup queue, which is empty thus far, since no other patch seeds have been detected. Then, we remove ((4,4), (Move,Move)) from the top of the backup queue (line 15), and update its value, *i.e.* $Q_{\text{patch}}((4, 4), (\text{Move}, \text{Move})) \leftarrow 100$ (line 16). The predecessors of the state (4,4) will then be added to the backup queue. They are:

- ((4,4), (Wait,Wait))
- ((3,4), (Move,Wait))
- ((4,3), (Wait,Move))
- ((3,3), (Move,Move))

[2] Moore and Atkeson [1993] used the symbol ϵ to represent this threshold. We use Δ_ϵ instead to distinguish it from ϵ for ϵ-greedy exploration.



(a) The initial step of patching when the unexpectedly large reward is observed at ((4,4), (Move,Move)).



(b) Backing up the change in Q value of the patch seed through its predecessors.

Figure 4: Stages of unbounded patching on the small coordination example domain. Shaded cells indicate states with at least one action patched at that state.

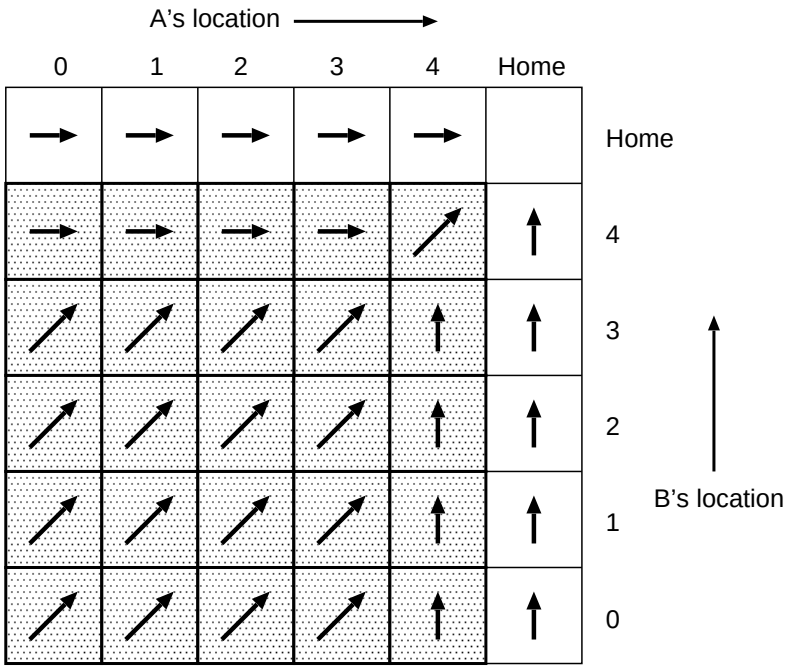


Figure 5: The converged policy from unbounded patching on the example domain. Shaded cells indicate states with at least one action patched at that state.

All of these predecessors will be predicted to have the same large change in Q value of +80, so they will all be added to the backup queue with the same priority. The next state-actions that are backed up will be dependent on the ordering of equal priority predecessors in the backup queue, which is implementation dependent. If the backup queue is a first-in-first-out queue, then this generation of predecessors will be updated before the next generation, producing the policy shown in Figure 4(b). Patching at (3,4) and (4,3) has produced a change in the greedy policy at those states, as the patched Q values now reflect that it is much more profitable for the actor at location 4 to wait for its partner so as to collect the much larger combined reward.

Continuing with the algorithm will propagate the change in Q value throughout all predecessors of the patch seed. This will converge to the policy shown in Figure 5, where the Q function patch has expanded to a total of 82 values. Unbounded patching has converged to the optimal policy. Note also that the states at which one actor is already at Home have not been patched, as they are not predecessors of the patch seed.

5.2 Bounding the Patch

Eventually, unbounded patching will add all ancestors of all patch seeds to Q_{patch}. This is not unexpected, since unbounded patching is a direct adaptation of prioritised sweeping to patching, with no attempt at bounding Q_{patch} growth.

We now discuss two heuristics that aim to restrict patch growth while still improving the policy. These heuristics may be applied to the unbounded patching algorithm separately or together.

5.2.1 Policy Bounding

In the unbounded patching algorithm, patch values are propagated back from the seed points to all of its predecessors, adjusting the policy as it goes. It is in a literal sense the unbounded patching strategy.

A problem with this is that not all added patch values will affect the greedy policy, which is what we are actually interested in improving. The most blatant example of this in our example is the action (Wait,Wait) – this always deterministically results in a self-transition with negative reward, and is not optimal in any state. Nevertheless, unbounded patching will patch it as long as it is an ancestor of any patch seed. Consequently, Q_{patch} grows without any improvements in the resulting behaviour.

In order to prevent this kind of unnecessary patch growth, *Policy bounding* adds the restriction that values are added to Q_{patch} only when they immediately affect the greedy policy. This can be seen as using immediate change in the policy as a heuristic to decide whether growing the patch is still effective. Adding policy bounding to unbounded patching produces the modified algorithm shown in Algorithm 3.

We refer to the condition on propagation of patch values as the *policy bounding condition* (referred at lines 15, 22, 27 of Algorithm 3). The proposed update $Q_{\text{patch}}(s,a) \leftarrow q$ for state-action (s,a) and new Q value q satisfies the policy bounding condition if either:

- (s,a) satisfies the patch seed predicate, *i.e.* $seed(s,a)$ is true; or
- (s,a) is already patched, *i.e.* $(s,a) \in Dom(Q_{\text{patch}})$; or
- the greedy action at s would change as a result of the updated value, *i.e.* $Q_{\text{patch}}(s,a) \leftarrow q$ would change $\arg\max_{a' \in A(s)} Q(s,a')$; or
- a is the greedy action at s and has a non-zero probability of self-transition, *i.e.* $a = \arg\max_{a' \in A(s)} Q(s,a') \wedge P(s,a,s) > 0$.

When computing the greedy action at s to test these conditions, the current set of Q_{patch} values are taken into account – thus, policy bounding will allow values to be added to Q_{patch} so long as the greedy policy is changing. The last condition that checks for the possibility of self-transitioning is required because self-transitioning greedy actions re-use their own value to calculate their updated value.

The policy bounding condition is checked both on entry into the backup queue (line 27) and on exit from the backup queue (line 22). Checking the condition on entry into the backup queue reduces the number of ineffective backups in the queue – this is important, because in practice, we want a backup queue with restricted capacity. Checking the condition on exit ensures that the update is still effective after taking into account the changes to the model and other Q values that occurred while it was in the queue.

To illustrate the effect of policy bounding, we return to the policy shown in Figure 4(a). The state-action ((4,4), Move,Move)) is detected as a patch seed and thus satisfies the patch seed predicate (line 11), and is placed on the top of the backup queue (line 12). It is then removed from the backup queue (line 20), and has its value backed up (line 23). As with unbounded patching, the predecessors of the state (4,4) are now considered for addition to the backup queue (lines 24 to 34). However, unlike unbounded patching, not all predecessors are enqueued because of the policy bounding condition (line 27) – specifically:

- ((4,4), (Wait,Wait)) is not added because it does not satisfy the policy bounding condition;
- ((3,4), (Move,Wait)) is added because it would change the greedy policy at (3,4);

Algorithm 3: Policy bounded patching.

begin
  initialise empty backup queue;
  initialise Q_{patch}, P_{patch}, R_{patch} as empty functions;
  repeat
    $s \leftarrow$ initial state chosen according to S_0;
    while s is not terminal **do**
      $a \leftarrow$ chosen action according to Q and exploration policy;
      execute a, observe reward r, and next state s';
      update_P_patch(s, a, s');
      update_R_patch(s, a, r);
      if $seed(s, a)$ or $(s, a) \in Dom(Q_{\text{patch}})$ **then**
        promote (s, a) to top of backup queue;
      else if $\exists s'$ [$P(s, a, s') > 0 \wedge (s', .) \in Dom(Q_{\text{patch}})$] **then**
        $q \leftarrow R(s, a) + \gamma \sum_{s' \in S} P(s, a, s') max_{a' \in A(s')} Q(s', a')$;
        if updating (s, a) with q satisfies the policy bounding condition **then**
          promote (s, a) to top of backup queue;
        end
      end
      while backup queue not empty and allowed to do backups **do**
        $(s, a) \leftarrow$ remove top of backup queue;
        $q \leftarrow R(s, a) + \gamma \sum_{s' \in S} P(s, a, s') max_{a' \in A(s')} Q(s', a')$;
        if updating (s, a) with q satisfies the policy bounding condition **then**
          $Q_{\text{patch}}(s, a) \leftarrow q$;
          foreach predecessor (s_p, a_p) of s **do**
            $q \leftarrow R(s_p, a_p) + \gamma \sum_{s' \in S} P(s_p, a_p, s') max_{a' \in A(s')} Q(s', a')$;
            $\Delta \leftarrow \mid Q(s_p, a_p) - q \mid$;
            if $\Delta > \Delta_\epsilon$ and updating (s_p, a_p) with q satisfies the policy bounding condition **then**
              if (s_p, a_p) is in backup queue **then**
                increase priority of (s_p, a_p) in backup queue by Δ;
              else
                insert (s_p, a_p) into backup queue with priority Δ;
              end
            end
          end
        end
      end
      $s \leftarrow s'$;
    end
  until finished;
end

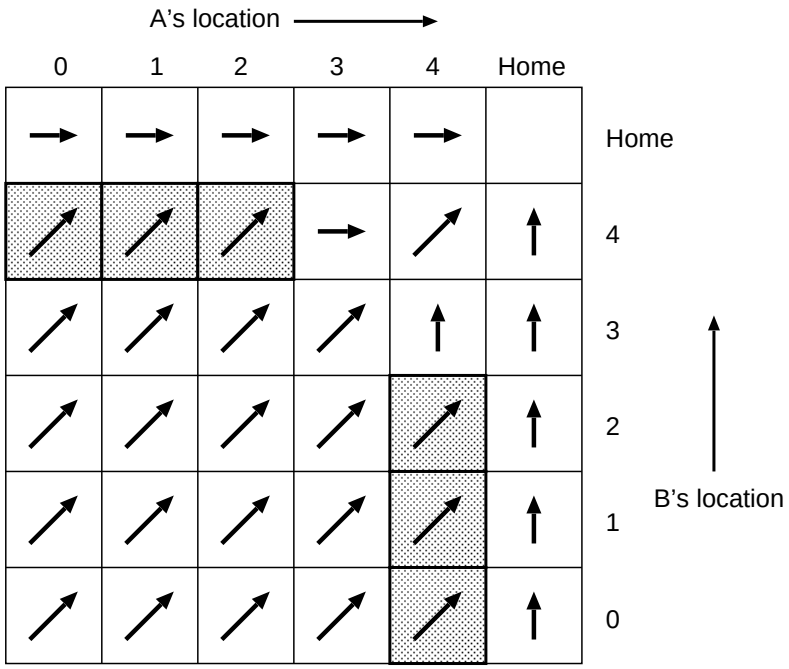


Figure 6: A counter-example for policy bounding. Because the policy at (3,4), (3,3), and (4,3) is already optimal, no action at those states will be patched. Consequently, the change in Q value will not be propagated to the shaded states (l,4) and (4,l) for $0 \leq l \leq 2$, so the sub-optimal policy at those states will not be corrected by policy bounded patching.

- ((4,3), (Wait,Move)) is added because it would change the greedy policy at (4,3);
- ((3,3), (Move,Move)) is not added because it does not satisfy the policy bounding condition.

While ((3,3), (Move,Move)) is not added for backup yet, it will be added later as the policy changes. This is because backing up the value for either (3,4) or (4,3) will make their predecessors look better than the currently underestimated Q value for ((3,3), (Move,Move)). Thus, the policy at (3,3) will be changed to either (Move, Wait) or (Wait,Move) until (4,4) is reached again and the Q value for ((3,3),(Move,Move)) is correctly backed up.

Policy bounded patching tends to learn more cautiously than unbounded patching, and is inclined to be limited to local adjustments around patch seeds. This usually keeps patch sizes smaller than unbounded patching, but may overlook some corrections to the policy. In general, a change in value may potentially affect the policy many steps back from the updated state, even if it does not immediately affect the policy at that state. Unbounded patching will correctly adjust for this as all changes in value are backed up, but policy bounding may ignore it.

Policy bounded patching is also more sensitive to the initial approximation. It is easy to construct examples where policy bounding would cause patching to fail to converge to the global optimum. For example, consider the policy shown in Figure 6. The approximate solution has the correct optimal action at two steps back from Home at states (3,4), (3,3) and (4,3). Therefore, the patch does not propagate beyond the patch seed, leaving the shaded states with the sub-optimal policy. Depending on the value of the combined reward when both actors Move to Home simultaneously, those states may have value that is arbitrarily worse than global optimality.

5.2.2 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.

To this end, *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 approximate value, *i.e.*, priority will be highest where $\hat{Q}$ is perceived to be least accurate[3]. This can be seen as trying to retain the set of Q_{patch} values that will minimise Q function error.

Under utility bounding, the statement $Q_{\text{patch}}(s,a) \leftarrow q$ has the operating convention of inserting an entry for (s,a) with Q value q and priority $\mid q - \hat{Q}(s,a) \mid$, or updating its value and priority if it already exists. Then, if Q_{patch} has exceeded capacity, the lowest priority element is dropped[4]. This modified update procedure is shown in Algorithm 4.

Algorithm 4: Modified update procedure for $Q_{\text{patch}}(s,a) \leftarrow q$.

```
Input: State-action (s, a), and its updated Q value q.
1  begin
2  |  Δ ← | q − Q̂(s, a) |;
3  |  if Q_patch(s, a) is defined then
4  |  |  re-insert (s, a) with Q value q and priority Δ;
5  |  else
6  |  |  insert (s, a) with Q value q and priority Δ;
7  |  |  if Q_patch exceeds capacity then
8  |  |  |  remove the lowest priority element of Q_patch;
9  |  |  end
10 |  end
11 end
```

Note that the priority function used with utility bounding is a heuristic: the set of Q_{patch} values that minimise the error in the Q function is distinct from the optimal set of Q_{patch} values for maximising expected reward when combined with $\hat{Q}$. However, it is not immediately clear how this optimal set of Q_{patch} values may be determined in general without first calculating the entire value function. Alternative heuristics may combine other features to determine the usefulness of a value in Q_{patch}, such as the frequency or probability of visiting its associated state (so as to direct storage to states visited most in practice), or more sophisticated features such as the influence or variance of the state Munos and Moore [2002]. We leave this as an area for future research.

5.3 Patching the Model

The patching algorithms rely on the model to calculate the adjusted values for Q_{patch}. We assumed that the estimates $\hat{P}$ and $\hat{R}$ would be initially provided. As with patching the Q function, we require procedures to maintain P_{patch} and R_{patch}.

[3]Because each entry in Q_{patch} includes the modified Q value, for this particular priority function, the priority value does not need to be explicitly stored since it can be efficiently calculated on demand.

[4]When multiple entries in Q_{patch} tie for the lowest priority and one of them needs to be dropped, deciding which particular value is dropped is implementation-dependent.

5.3.1 Patching Deterministic Transitions and Rewards

In the special case of deterministic transitions and reward functions, patching the model inaccuracies is straight-forward. For each step taken, we have a sample of the experienced state-action (s, a), and the resulting reward r and successor state s'. We can then directly compare the observed outcomes r and s' to their expected outcomes according to the given estimate model, and patch them if they are different. Specifically:

- if $\hat{R}(s, a) \neq r$, then (s, a) is added to R_{patch} with the value r.
- if $\hat{P}(s, a, s') \neq 1$, then (s, a, s') is added to P_{patch} with the value 1 [5].

In the deterministic case, we can determine whether entries should be added to P_{patch} and R_{patch} immediately when an unexpected result is observed. Thus, in our example, when the actors both Move to Home simultaneously for the first time and unexpectedly receive a much larger combined reward, we immediately add that state-action and reward to R_{patch}.

5.3.2 Patching Stochastic Transitions

In the more general case of stochastic transition and reward functions, we use statistical tests to detect inaccuracies. Samples are collected as the agent follows its policy during learning, *i.e.*, the samples are drawn from an on-policy distribution. The collected samples are then compared against the corresponding estimate.

To detect inaccuracies in $\hat{P}$, for a given state-action (s, a), we keep a list of observed successor states and their frequencies. We compare these samples to $\hat{P}(s, a, .)$ using the χ^2 statistical test – the distribution $\hat{P}(s, a, .)$ serves as the expected distribution, while the observed samples are used as the actual distribution. If the χ^2 statistic is below a threshold, (s, a) and the observed samples are added to P_{patch}.

A naive approach to collecting samples for statistical testing is to remember all transitions that are experienced. The problem with this approach is that it requires as much storage as building the transition model entirely. We adopt a more space-efficient strategy, by selectively sampling parts of the transition model at a time. The *transition sampling set*, P_{sample}, is the set of state-actions (s, a) for which successor state statistics are currently being collected. We implement P_{sample} as a priority queue of transitions with a fixed capacity. Entries in P_{sample} are prioritised by the number of experiences of that state-action. When the queue reaches capacity, a portion of the queue is dropped from the bottom. This means that less frequently experienced transitions are dropped to make room for more frequently experienced transitions.

The complete procedure for updating the patched transition function while selectively sampling is shown in Algorithm 5. Transitions may enter P_{sample} only if they are not already in P_{patch}. Transitions may exit P_{sample} by three ways:

- P_{sample} reaches full capacity, in which case a bottom portion of the queue is dropped to make way for the new transition (lines 32–35);
- by sampling an unexpected successor state (*i.e.* a successor with 0 probability according to $\hat{P}$), in which case the state-action is added to P_{patch} immediately (lines 12–15);
- sufficient samples are collected for that state-action and the χ^2 test is performed, in which case the state-action is added to P_{patch} if the χ^2 statistic indicates significant deviation from $\hat{P}$ (lines 17–22).

[5] Implicitly, this then means that $P_{\text{patch}}(s, a, s'') = 0$ for all $s'' \neq s'$.

Algorithm 5: Procedure update_P_patch(s, a, s').

Input: Transition sample with state s, action a, and successor state s'.

```
begin
    if P_patch(s, a, .) is defined then
        update P_patch(s, a, .) with s';
        if (s, a, s') ∈ P_remove^{-1} then
            remove (s, a, s') from P_remove^{-1};
        end
        if (s, a, s') ∉ P_add^{-1} ∧ P̂(s, a, s') = 0 then
            add (s, a, s') to P_add^{-1};
        end
    else if (s, a, .) ∈ P_sample then
        update P_sample(s, a, .) with s';
        if P̂(s, a, s') = 0 then
            initialise P_patch(s, a, .) using P_sample(s, a, .);
            add_P_inverse(s, a, P_sample(s, a, .));
            remove (s, a, .) from P_sample;
        else if sufficient samples collected for (s, a) then
            χ² test on collected samples for (s, a);
            if χ² statistic below threshold then
                initialise P_patch(s, a, .) using P_sample(s, a, .);
                add_P_inverse(s, a, P_sample(s, a, .));
            end
            remove (s, a, .) from P_sample;
        else
            increment priority of (s, a, .) in P_sample;
        end
    else
        initialise transition sample for (s, a, .) with s';
        if P̂(s, a, s') = 0 then
            initialise P_patch(s, a, .) using P_sample(s, a, .);
            add_P_inverse(s, a, [s']);
        else
            if P_sample is full then
                drop portion of P_sample;
            end
            add (s, a, .) to P_sample;
        end
    end
end
```

Once a state-action is in P_{patch}, any successor states experienced from that state-action are used to update the observed distribution in P_{patch} (lines 2–9). This includes updating the patch for the transition function inverse, discussed in Section 5.3.3.

Note that only irregular transitions are remembered by being added to P_{patch}. Transitions that appear to agree with $\hat{P}$ are not stored or remembered. This means that it is possible for a state-action to go through P_{sample} (and be eligible to statistical testing) multiple times.

5.3.3 Patching the Transition Inverse

Our patching algorithms require the computation of predecessors to states, so as to efficiently propagate changes in the Q function patch. Therefore, in addition to maintaining P_{patch} in the forward direction, we need to maintain it in the backwards direction as well. In practice, we require an efficient method for finding all state-actions that lead to a particular state; that is, a method that finds all state-actions that transition to a given state with non-zero probability without exhaustively sweeping through P.

To begin, we assume that the implementation of $\hat{P}$ also provides us with $\hat{P}^{-1}$, with an interface that takes a state s as input, and returns the list of all state-actions (s_p, a_p) such that $\hat{P}(s_p, a_p, s) > 0$. We wish to augment this interface to operate with the transition distribution defined by $\hat{P}$ combined with P_{patch}. We implement this by maintaining two sets, P^{-1}_{add} and P^{-1}_{remove}:

- P^{-1}_{add} is the set of transitions that have 0 probability according to $\hat{P}$, but are observed to actually occur in practice. Intuitively, this set is used to represent the transitions that need to be *added* to the list of predecessors predicted by $\hat{P}^{-1}$. Formally:

$$P^{-1}_{\text{add}} = \{(s, a, s') \mid P(s, a, s') > 0 \wedge \hat{P}(s, a, s') = 0\} \tag{12}$$

  We implement P^{-1}_{add} as a hashtable, mapping successor states to lists of additional predecessor state-actions not included in $\hat{P}^{-1}$.

- P^{-1}_{remove} is the set of transitions that are not observed but have probability greater than 0 according to $\hat{P}$. Intuitively, this set represents the transitions that need to be *removed* from the list of predecessors predicted by $\hat{P}^{-1}$. Formally:

$$P^{-1}_{\text{remove}} = \{(s, a, s') \mid P(s, a, s') = 0 \wedge \hat{P}(s, a, s') > 0\} \tag{13}$$

  We implement P^{-1}_{remove} as a hashed set of state-action-state triples.

When a state-action is already in P_{patch}, for each new transition sample (s, a, s'), P^{-1}_{add} and P^{-1}_{remove} are updated incrementally:

- if the observed transition (s, a, s') is in P^{-1}_{remove}, it is removed to reflect that (s, a, s') does *not* have a 0 probability of occurrence (lines 4–6 of Algorithm 5);
- if the observed transition (s, a, s') is not in P^{-1}_{add} and $\hat{P}(s, a, s') = 0$, it is added to P^{-1}_{add} to reflect that (s, a) is a predecessor of s' that is not predicted by $\hat{P}^{-1}$ (lines 7–9 of Algorithm 5).

Alternatively, if a state-action (s, a) and its observed successor samples have been declared to be inconsistent with $\hat{P}$, the procedure add_P_inverse is invoked (at lines 15, 21, and 31 of Algorithm 5). This effectively does a batch update to P^{-1}_{add} and P^{-1}_{remove} for (s, a):

Algorithm 6: Procedure add_P_inverse(s, a, *successors*).

Input: State s, action a, and *successors*, the list of successor states that have been observed for (s, a).

```
1  begin
2      foreach s' ∈ successors do
3          if (s, a, s') ∉ P⁻¹_add ∧ P̂(s, a, s') = 0 then
4              add (s, a, s') to P⁻¹_add;
5          end
6      end
7      foreach s' ∈ {s' | P̂(s, a, s') > 0} do
8          if (s, a, s') ∉ P⁻¹_remove ∧ s' ∉ successors then
9              add (s, a, s') to P⁻¹_remove;
10         end
11     end
12 end
```

- any observed successor state s' that has 0 probability according to $\hat{P}$ indicates that (s, a) is a predecessor to s' that is not predicted by $\hat{P}^{-1}$, and therefore (s, a, s') is added to P^{-1}_{add} (lines 2–6);
- for all successors that $\hat{P}$ predicts for (s, a), any successor state s' that is not in the observed samples is excluded as a successor for (s, a) in the patched transition function, and therefore (s, a, s') is added to P^{-1}_{remove} (lines 7–11).

Algorithm 7: Modified look-up procedure for P^{-1}.

Input: State s.

Output: The list of predecessors (s_p, a_p) of s, where $P(s_p, a_p, s) > 0$.

```
1  begin
2      predecessors ← empty list;
3      foreach (s_p, a_p) ∈ {(s_p, a_p) | P̂(s_p, a_p, s) > 0} do
4          if (s_p, a_p, s) ∉ P⁻¹_remove then
5              add (s_p, a_p) to predecessors;
6          end
7      end
8      foreach (s_p, a_p) ∈ {(s_p, a_p) | (s_p, a_p, s) ∈ P⁻¹_add(s)} do
9          add (s_p, a_p) to predecessors;
10     end
11     return predecessors
12 end
```

Using P^{-1}_{add} and P^{-1}_{remove}, the modified look-up procedure to find the predecessors of a state s is shown in Algorithm 7. This takes the predecessors according to $\hat{P}^{-1}$ and filters out any transitions in P^{-1}_{remove} (lines 3–7), and then adds all predecessors of s included in P^{-1}_{add} (lines 8–10).

5.3.4 Patching Stochastic Rewards

Our approach to detecting inaccuracies in $\hat{R}$ is similar to our approach for $\hat{P}$. For a given state-action (s, a), we collect samples of the observed reward. These samples are then compared to $\hat{R}(s, a)$ using the Kolmogorov-Smirnov statistical test. Note that this requires knowledge of not just the expected value of $\hat{R}(s, a)$, but its distribution as well. If the Kolmogorov-Smirnov statistic is below a user-defined threshold, (s, a) is added to R_{patch}, with a moving average reward that is initialised with the observed samples.

As with collecting samples for testing $\hat{P}$, R is also selectively sampled, with the *reward sampling set*, R_{sample}. Like P_{sample}, R_{sample} is implemented as a priority queue of state-actions with a fixed capacity, where entries are prioritised by the number of experiences. A portion from the bottom of the queue is dropped when it is full.

A difference between testing $\hat{R}$ and testing $\hat{P}$ is that there is no equivalent of an unexpected successor state for $\hat{R}$. Therefore, while we can immediately determine that $\hat{P}(s, a, .)$ is inaccurate when an unexpected successor state is observed, the accuracy of $\hat{R}(s, a)$ is only gauged when sufficient samples have been collected for $R(s, a)$ [6].

Algorithm 8: Procedure update_R_patch(s, a, r).

```
   Input: Reward sample for state-action (s, a), with observed reward r.
 1 begin
 2   if R_patch(s, a) is defined then
 3     update moving average R_patch(s, a) with r;
 4   else if (s, a, .) ∈ R_sample then
 5     update reward sample for (s, a) with r;
 6     if sufficient samples collected for (s, a) then
 7       Kolmogorov-Smirnov test on collected samples;
 8       if Kolmogorov-Smirnov statistic below threshold then
 9         R_patch(s, a) ← moving average initialised with observed samples;
10       end
11       remove (s, a, .) from R_sample;
12     else
13       increment priority of (s, a, .) in R_sample;
14     end
15   else
16     initialise reward sample for (s, a, .) with r;
17     if R_sample is full then
18       drop portion of R_sample;
19     end
20     add (s, a) to R_sample;
21   end
22 end
```

The procedure for updating the patched reward function with selective sampling is shown in Algorithm 8. State-actions that have already been added to R_{patch} have their moving average value updated (lines 2 and 3). State-actions that are in R_{sample} are updated with the new observed sample (line 6), and statistically tested if sufficient samples have been collected (lines 7–12), or incremented in priority otherwise (line 14). Finally, if (s, a) is a new state-action for

[6] Unless R is known to be deterministic, in which case it may be patched immediately, as discussed in Section 5.3.1

sampling, a portion of R_{sample} is dropped from the bottom if it is full (line 19), then (s, a) is added to R_{sample}.

5.3.5 Patched Model Seed Predicate

In addition to correcting inaccuracies in the supplied approximate model, patching the model serves another useful purpose: the patch seed predicate can be defined as true for the state-actions that appear in P_{patch} and R_{patch}. This is a particularly useful method for detecting patch seeds when $\hat{Q}$, $\hat{P}$, and $\hat{R}$ all depend on the same assumptions for approximation. For our running example, each component was constructed by assuming that the actors are completely independent. Thus, when the model is found to be inaccurate, it is likely that the Q values immediately around that region will also be inaccurate and require patching.

Two key parameters in using this patch seed predicate are the number of samples collected before applying the statistical test, and the threshold at which $\hat{P}$ and $\hat{R}$ are determined to be inaccurate. If the threshold is too lenient, it suggests a large number of patch seeds, which encourages a larger patch. If the threshold is too tight, it may miss true patch seeds. In practice, neither the time nor space required to collect an arbitrarily large number of samples is available, so tuning these parameters is a trade-off between patch storage and performance.

5.4 Model-free Patching

One implication of utility bounded patching is the possibility of model-free patching algorithms. For example, we can directly adapt Q-learning [Watkins, 1989, Watkins and Dayan, 1992] to learning Q_{patch} by treating all experienced state-actions as eligible for patching, and swapping out the usual Q function update step:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left[r + \max_{a' \in A(s')} Q(s',a') \right] \tag{14}$$

with:

$$Q_{\text{patch}}(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left[r + \max_{a' \in A(s')} Q(s',a') \right] \tag{15}$$

We will refer to this modified algorithm as *patch Q-learning*. The effect of this change is that the fixed capacity priority queue used by utility bounding will now adaptively drop values from Q_{patch} when it is full. Storage is therefore allocated to where priority is highest. In our case, as discussed in Section 5.2.2, a value in Q_{patch} has priority equal to its absolute difference from the corresponding value in $\hat{Q}$.

Other model-free reinforcement learning algorithms such as SARSA [Rummery and Niranjan, 1994] may also be adapted similarly.

6 Experiments

In this section, we evaluate patching in several domains. The first is the small coordination example used to illustrate the algorithms through Section 4. We then examine the performance of patching on two variations of Dietterich's taxi problem [Dietterich, 2000], and show how patching may be applied over a MAXQ decomposed value function. The last domain is an extension of the taxi problem to two taxis and two passengers, showing the performance of patching on a larger domain. For the sake of brevity, we present a summary of results in this section, with detailed results deferred to Appendix E.

In all our experiments, we use ϵ-greedy action selection, with $\epsilon = 0.1$. This means that for any state, with probability $1 - \epsilon$, the greedy action is selected, while with probability ϵ, a random action is selected uniformly from all available actions at that state (including the greedy action).

All plots in this section show the averages over 10 runs for each experiment, with errorbars indicating one standard deviation. Learning is suspended at regular intervals to evaluate the greedy policy. The plots of expected reward show the results from the policy evaluation samples, *not* the performance while learning with ϵ-greedy action selection.

Patching was limited to 2 backups per step (line 12 of Algorithm 2 and line 17 of Algorithm 3), except for patch Q-learning which updates only the most recently experienced state-action. For the backup queue threshold Δ_ϵ , all experiments used a value of $\Delta_\epsilon = 0.0001$. For experiments where the transition or reward functions were adaptively patched, the bottom 10% of P_{sample} and R_{sample} were dropped when they were full (lines 32–34 of Algorithm 5, and lines 17–19 of Algorithm 8).

In addition to the patching algorithms, two instances of prioritised sweeping are included in this section as baselines:

- *Prioritised sweeping from scratch* – this is exactly the original prioritised sweeping algorithm (Algorithm 1), starting from scratch with initial Q values of 0 for all state-actions. The model is also constructed from scratch, using only the observed samples from the environment to estimate P and R. This provides a lower bound baseline, indicating whether the initial approximations are helpful.

- *Initialised prioritised sweeping* – this is exactly the original prioritised sweeping algorithm, with the Q table initialised to $Q(s, a) = \hat{Q}(s, a)$ for all state-actions (s, a). In addition, the model is initialised with $\hat{P}$ and $\hat{R}$ and updated using the model patching routines (Algorithms 5 and 8) instead of constructing the model from scratch. Therefore, the only differences between this instance of prioritised sweeping and the patching algorithms are patch seeding and the bounding heuristics. Thus, this provides a measure of the effectiveness of those aspects, indicating whether learning based on patch seeding (as opposed to updating the Q values of all encountered state-actions) and the bounding heuristics are helpful.

Where applicable, for the sake of consistency of comparison, both instances of prioritised sweeping use the same parameters as their model based patching counterparts. This includes ϵ for ϵ-greedy exploration, backup queue capacity, number of updates per step, backup queue threshold Δ_ϵ , and parameters for patching the model for initialised prioritised sweeping.

6.1 Coordination Example

We start with the small example coordination domain. The domain is as described in Section 4.1. In this domain, we evaluate unbounded patching, patching with policy and utility bounding (both separately and together), and patch Q-learning. The total size of the problem is 120 state-actions.

6.1.1 Experiment Setup and Parameters

The initial approximations $\hat{Q}$, $\hat{P}$, and $\hat{R}$ are as described in Section 4.1, as is the patch seed predicate. Because the domain is deterministic, the model is patched immediately when an unexpected outcome is observed, and patch Q-learning uses a learning rate of $\alpha = 1$.

The plots of expected reward were generated by calculating the value function of the policy and taking the weighted average over the initial state distribution.

6.1.2 Results

We start with a comparison of unbounded patching, patching with policy bounding, initialised prioritised sweeping, and prioritised sweeping from scratch. Figure 7(a) shows the expected reward for these algorithms. The three algorithms using the initial approximation rapidly converge to the optimal policy, while prioritised sweeping from scratch manages relatively small improvements in the same short time frame, indicating the usefulness of the initial approximation. The difference between the patching algorithms and initialised prioritised sweeping is caused by the differences in the way that backups are ordered.

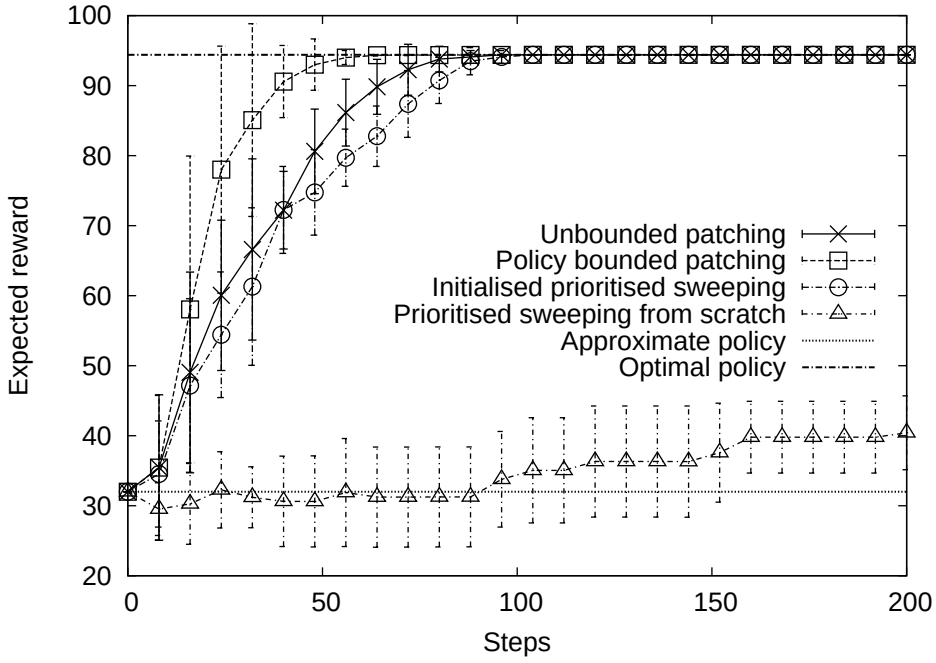
Policy bounded patching propagates backups out from the patch seed once it is discovered, but ignores backups that do not immediately affect the policy. This has the effect of changing the policy faster. At the other end of the spectrum, prioritised sweeping *always* backs up the most recent state-action at each step. This means that it may "waste" one of its updates per step if the correct Q value is already known for the most recent state-action, with no change to both the Q function and (consequently) the resulting behaviour. Unbounded patching lies in the middle: its backups are ordered by propagation outwards from the patch seed, so it may not update the most recent state-action at each step, but backups are not restricted by the policy bounding condition.

The high standard deviation early in learning is caused by the variation in time taken to discover the one patch seed in the problem – neither patching algorithm updates any values until the patch seed is found, so the policy is stationary until it discovers the patch seed. The same is true of initialised prioritised sweeping – $\hat{Q}$ is the exact solution to the relaxed problem described by $\hat{P}$ and $\hat{R}$, so no values are updated until it stumbles on the one state-action where $\hat{P}$ differs from P.

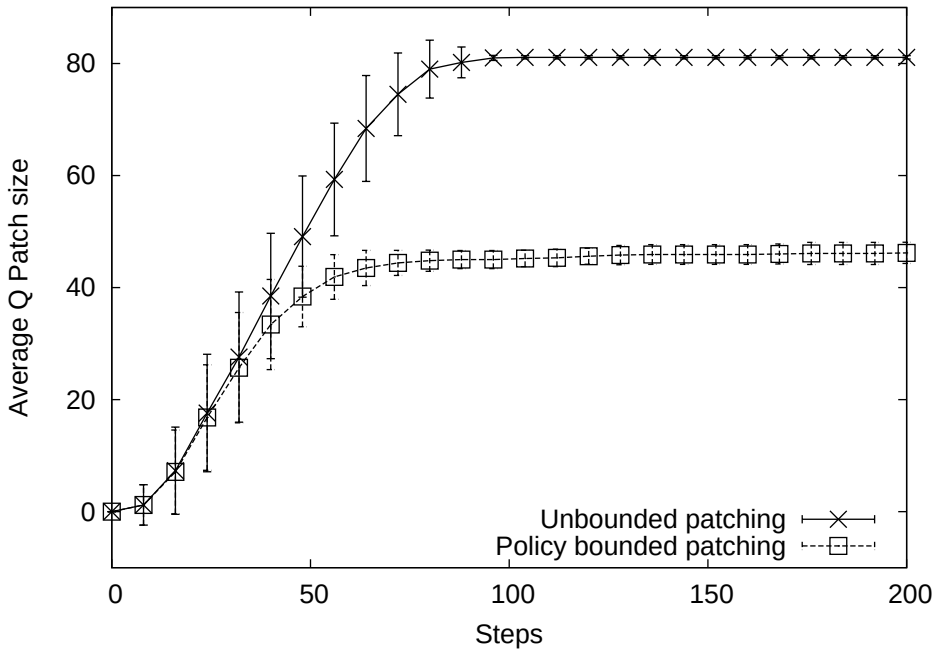
The average Q_{patch} sizes for unbounded patching and policy bounded patching are shown in Figure 7(b). Unbounded patching produces a much larger patch, covering all ancestors of the patch seed. Policy bounding keeps the patch size more manageable. Figure 7(c) plots the average mean squared error of the Q function during learning (*i.e.* $\hat{Q}$ augmented with Q_{patch}) relative to the optimal Q function. Unbounded patching and initialised prioritised sweeping both reduce this error to be very minimal, while policy bounded patching stops well short. Thus, while policy bounded patching produces the optimal policy with a much smaller patch, it does not converge to the optimal Q function.

Next, we examine patching with utility bounding but without policy bounding. Figure 8(a) shows the expected reward for patching with utility bounding, with Q_{patch} capacities of 25, 50, and 75 (20.83%, 41.67%, and 62.5% of the state-action space respectively). These capacities are deliberately undersized, to gauge the deterioration in policy quality as Q_{patch} capacity is reduced. With utility bounding restricting Q_{patch} size to 75 values, patching is still able to reach the optimal policy. As the capacity of Q_{patch} is reduced, the quality of the resulting policy suffers, converging to lower expected reward. However, for all capacities shown, patching produces an improvement on the initial approximation. Figure 8(b) compares the resulting Q functions in this setting to the optimal Q function – the error in the patched Q function steadily drops and converges soon after Q_{patch} reaches maximum capacity.

Next, we switch on both policy and utility bounding. Figure 9(a) shows the expected reward with Q_{patch} capacities of 20 and 30 (16.67% and 25% of the state-action space respectively). In both cases, the expected reward improves dramatically early on, but then settles at a lower expected reward after Q_{patch} reaches its maximum capacity. This occurs because there are

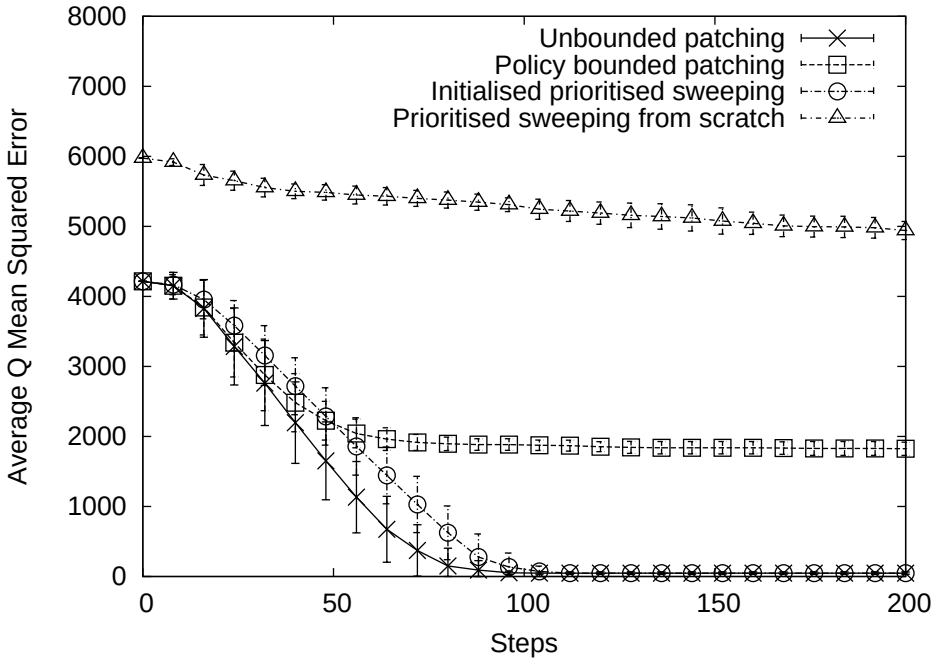


(a) Expected reward, averaged over the initial state distribution, for unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch.



(b) Average Q_{patch} size for unbounded patching and policy bounded patching.

Figure 7: Results for the coordination example domain.



(c) Average mean squared Q function error, compared to the optimal Q function.

Figure 7 (continued): Results for the coordination example domain.

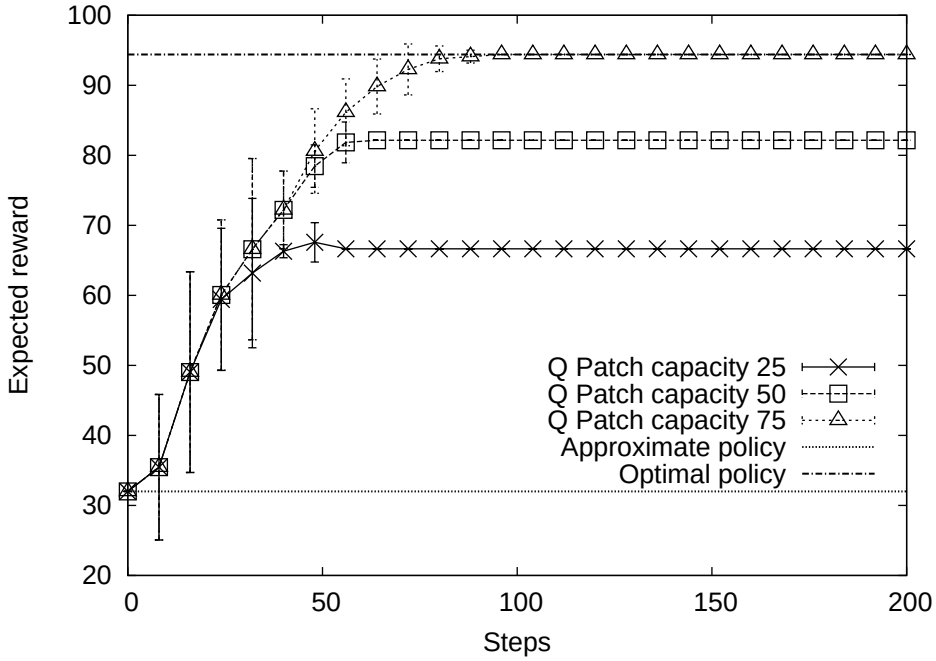
many state-actions that are eligible for patching, but the priority function will cause those with the largest Q function difference from $\hat{Q}$ to be retained. As stated in Section 5.2.2, this is a heuristic, and may not retain the optimal set of Q_{patch} values for maximising policy value. Nevertheless, combining policy and utility bounding makes better use of the limited storage than utility bounding alone in this domain. In terms of the Q function error, as shown in Figure 9(b), the error remains extremely high.

Lastly, we examine results for patch Q-learning. Figure 10(a) shows the expected reward for patch Q-learning, with utility bounding restricting Q_{patch} capacity to 25, 50, and 75. Patch Q-learning converges to the same expected reward as its model-based equivalent with the same Q_{patch} capacities, but without exploiting the benefit of the model, it requires many more steps to do so. Figure 10(b) compares the resulting Q functions to the optimal Q function. As with all other cases described above, the Q function difference drops steadily, and then settles after Q_{patch} capacity is reached.

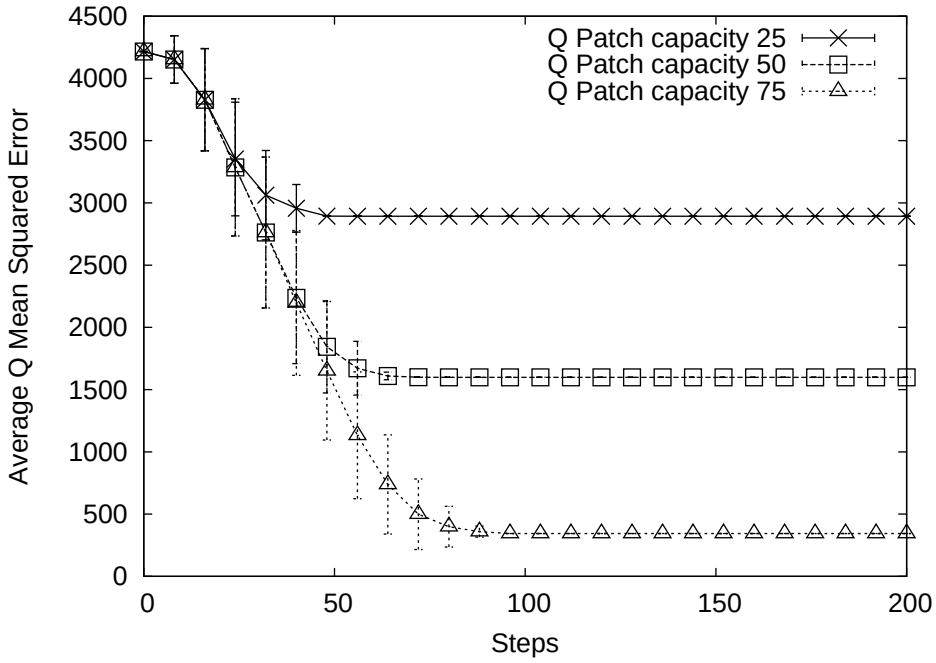
Table 1 summarises the results for the coordination example domain.

6.1.3 Discussion

The coordination example domain is very small, but serves to demonstrate the main characteristics of patching. Policy bounded patching makes much better use of the limited storage than unbounded patching and patch Q-learning in terms of policy improvements. However, unbounded patching and patch Q-learning produce a much more accurate estimate of the optimal Q function. Also, patch Q-learning is much slower to learn than its model-based counterparts, illustrating the usefulness of the model for patching.

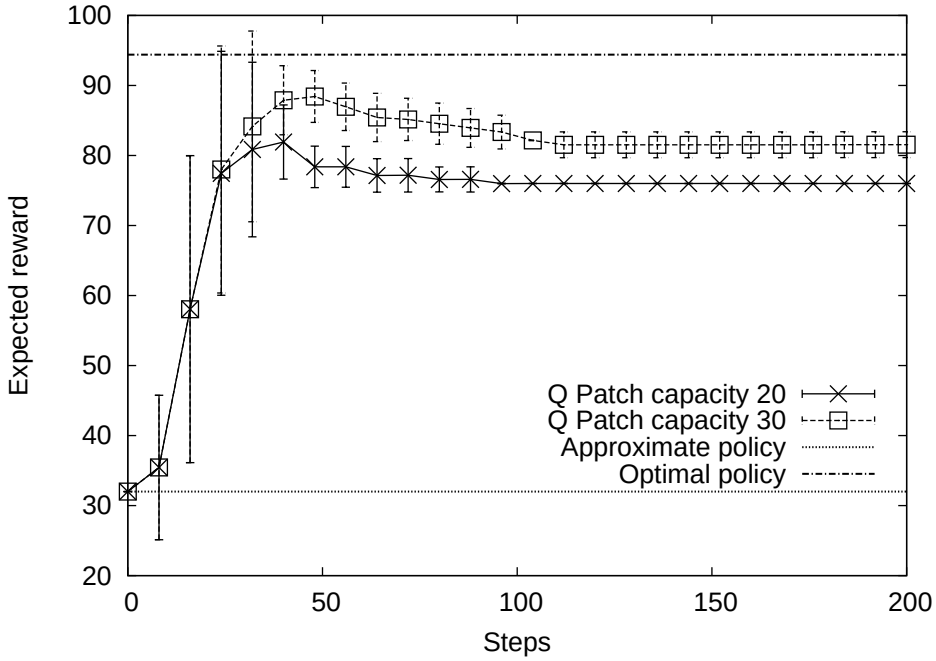


(a) Expected reward, averaged over the initial state distribution.

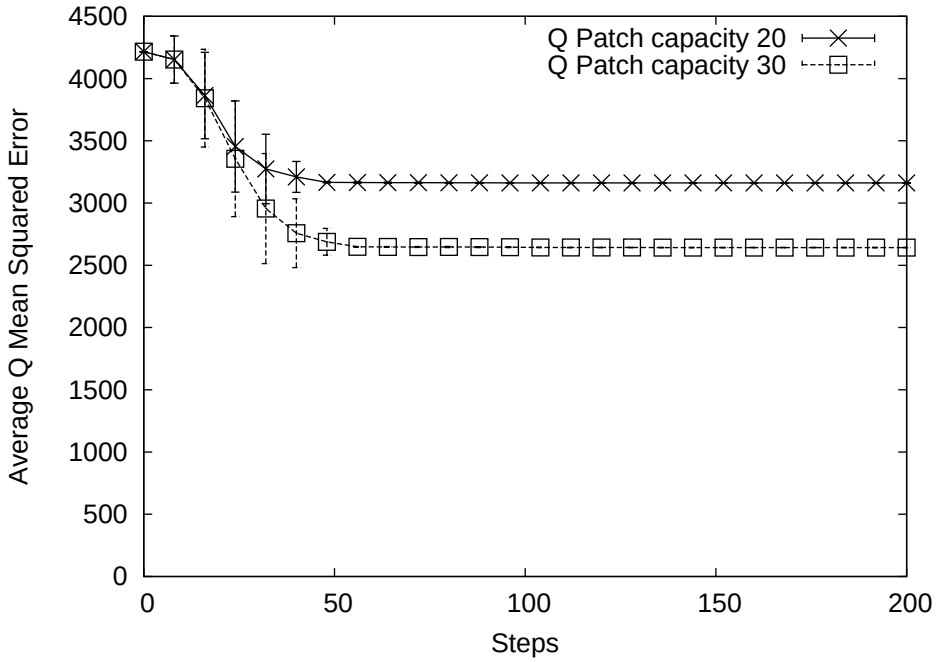


(b) Average mean squared Q function error, compared to the optimal Q function, for unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch.

Figure 8: Results for the coordination example domain for patching with utility bounding, with Q_{patch} capacities of 25, 50, and 75.

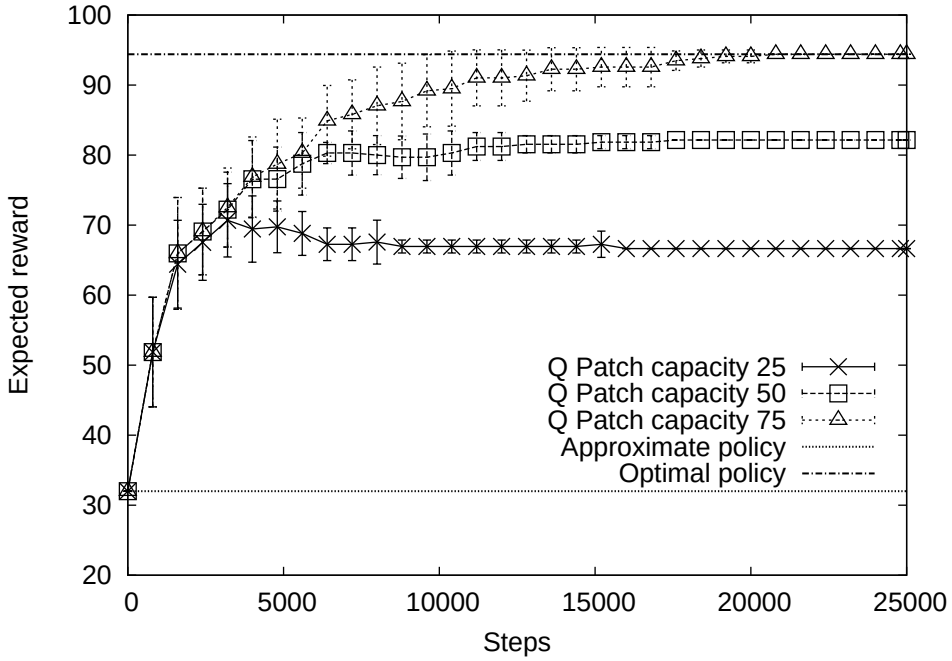


(a) Expected reward, averaged over the initial state distribution.

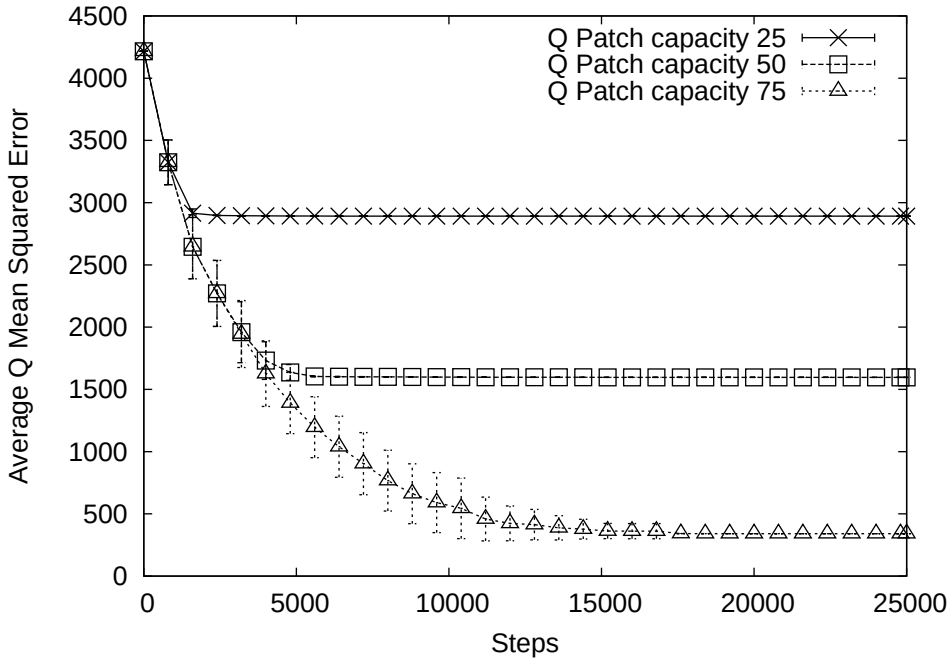


(b) Average mean squared Q function error, compared to the optimal Q function.

Figure 9: Results for the coordination example domain for patching with both policy and utility bounding, with Q_{patch} capacities of 20 and 30.



(a) Expected reward, averaged over the initial state distribution.



(b) Average mean squared Q function error, compared to the optimal Q function.

Figure 10: Results for the coordination domain for patch Q-learning with Q_{patch} capacities of 25, 50, and 75.

Table 1: Summary of results for the coordination example domain. Figures were taken at the end of 200 steps for unbounded patching, patching with policy and/or utility bounding, and both instances of prioritised sweeping. Figures for patch Q-learning were taken at the end of 25,000 steps. 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 ($\hat{Q}$)	32.00	10
Optimal flat	94.40	120
Prioritised sweeping from scratch	40.42 ± 5.27	120
Initialised prioritised sweeping	94.40 ± 0.00	120
Unbounded patching	94.40 ± 0.00	10 + 81.10 ± 0.30
With policy bounding	94.40 ± 0.00	10 + 46.20 ± 1.89
With utility bounding		
– Q_{patch} capacity 25	66.64 ± 0.00	10 + 25
– Q_{patch} capacity 50	82.16 ± 0.00	10 + 50
– Q_{patch} capacity 75	94.40 ± 0.00	10 + 75
With policy and utility bounding		
– Q_{patch} capacity 20	76.00 ± 0.00	10 + 20
– Q_{patch} capacity 30	81.54 ± 1.85	10 + 30
Patch Q-learning		
– Q_{patch} capacity 25	66.64 ± 0.00	10 + 25
– Q_{patch} capacity 50	82.16 ± 0.00	10 + 50
– Q_{patch} capacity 75	94.40 ± 0.00	10 + 75

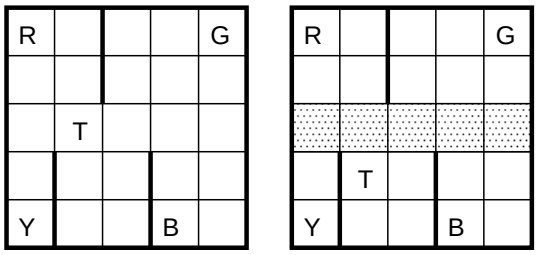


Figure 11: The taxi domain. R, G, B, and Y indicate the taxi stands, T indicates the taxi location. In (b), the shaded cells indicate locations that have had the transition and reward functions slightly changed for the modified taxi domain, for different passenger delivery contexts.

6.2 Modified Taxi

This set of experiments uses modified versions of Dietterich's taxi problem [Dietterich, 2000]. In the taxi problem, a taxi agent in a 5-by-5 grid world has the objective of delivering a passenger from a specially marked taxi stand to a destination taxi stand (see Figure 11(a)). The total size of the taxi problem is 3,000 state-actions.

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 taxi 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. These modifications are deliberately designed such that in the modified problem, the navigation sub-tasks are *not* entirely context independent with respect to the passenger. Consequently, an optimal policy for the original taxi problem falls well short of global optimality when directly transferred to the modified taxi problem. Our aim is to demonstrate patching over a different type of Q function approximation to that used in the previous domain, as well as patching the model for both stochastic transitions and stochastic rewards. We examine the performance of unbounded patching, patching with policy and utility bounding (both separately and together), and patch Q-learning.

6.2.1 Original Taxi Domain Description

We start by describing Dietterich's original taxi domain, then outline the modified domain, followed by the construction of the approximate solution.

The taxi domain is shown in Figure 11(a). States are described by three variables: the taxi location, the passenger location, and the passenger destination. The taxi location may be any of the 25 grid cells in the world. The passenger may be located at one of the taxi stands or In_Taxi, and the passenger's destination may be any of the taxi stands.

At each time step, the taxi may take one of six actions: North, South, East, West, Pickup, and Putdown. The compass point actions move the taxi one cell in the intended direction with probability 0.8, and move one cell to the left or right of the intended direction with probability 0.1 each. The taxi location is unchanged if the taxi hits a wall or barrier (marked by the thicker lines in Figure 11). The Pickup action is legal when the taxi is at the taxi stand where the passenger is located. In this case, the Pickup action deterministically changes the passenger location to be In_Taxi. Otherwise, the Pickup action is illegal and has no effect. The Putdown action is legal when the taxi is carrying the passenger and is at the passenger's destination. In

this case, the passenger is successfully delivered and the problem terminates. In all other cases, the Putdown action is illegal and has no effect.

The reward is -10 for an illegal Pickup or Putdown action, or -1 otherwise. Successfully delivery of the passenger earns an additional $+20$ reward on that time step.

The initial taxi location is selected randomly at the start of each trial from anywhere in the world. The passenger location and destination are both initialised randomly from the four taxi stands.

6.2.2 Modified Taxi Domain

In the modified domain, P and R are slightly changed on the middle row of the grid, shaded in Figure 11(b). For each pair of values for the passenger location and destination, two navigation state-actions on the middle row are modified as follows:

- For state-actions where P is modified, the modified navigation action no longer moves in the intended direction with probability 0.8 and to the left or right of the intended direction with probability 0.1 each. Instead, it now moves in the intended direction with probability 0.1, and to the left or right of the intended direction with probability 0.45 each. Note that all transitions with zero probability according to $\hat{P}$ still have a zero probability in the modified transition distribution. This means that inaccuracies in $\hat{P}$ must be detected by the χ^2 test – they cannot be detected by sampling an unexpected successor state.
- For state-actions where R is modified, the modified navigation action incurs an unexpectedly costly reward of -6 when the effect of the navigation action is to move in the intended direction (with probability 0.8), but the usual -1 if it moves to the left or right of the intended direction (with probability 0.2).

Each of the modified state-actions is optimal in the original taxi problem formulation. The modifications between P and R are balanced to 20 modifications each, with no overlap (*i.e.*, a state-action that is modified is either modified in P or R, but not both).

In addition, uniform noise is added to the reward function, such that a state-action that would normally deterministically receive X reward in the original taxi problem now receives a random value selected uniformly from $[X - 0.5, X + 0.5)$. State-actions for which R has been modified are also subject to this reward signal noise.

6.2.3 Experiment Setup and Parameters

We evaluate patching on three different settings of this problem: undiscounted, discounted terminating, and discounted continuing. The undiscounted setting is the original taxi problem with the modifications to P and R described above. In the discounted terminating setting, the domain specification remains unchanged, but the criteria for optimality is changed by setting the discount factor $\gamma = 0.95$. In the discounted continuing setting, the problem is further modified by no longer terminating on successful delivery of the passenger. Instead, a new passenger randomly appears with location and destination both selected randomly from the taxi stands with equal probability (*i.e.*, a probability of $1/16 = 0.0625$ for each passenger location and destination pair).

The initial MAXQ solution is learned using the MAXQ task hierarchy specified by Dietterich [2000], with minor variations for the different settings. The parameters used for MAXQ learning are similar to those specified by Dietterich. $\hat{Q}$ is calculated on demand from the learned hierarchical value function. The details of the MAXQ hierarchy and the derivation of $\hat{Q}$ are not the focus of this work, and we defer its details to Appendix A and B.

For the initial approximation of the model, $\hat{P}$ and $\hat{R}$ are taken as their counterparts for the original taxi problem without modifications. For statistical testing of the reward function, the uniform noise on the reward signal is also known. The modifications made to P and R are detected using the χ^2 and Kolmogorov-Smirnov statistical tests. For $\hat{P}$, 20 successor state samples were required for a state-action before the χ^2 test was applied. For $\hat{R}$, 10 reward samples were required for a state-action before the Kolmogorov-Smirnov test was applied. In both cases, the approximate model was deemed inaccurate at that state-action if the statistical test returned a null hypothesis probability less than 0.00001. For the results presented in this section, the transition and reward sampling sets, P_{sample} and R_{sample}, both had maximum capacities of 100 state-actions; results for other capacities of P_{sample} and R_{sample} show similar trends to those shown here, and are deferred to Appendix E.1.

For the patch seed predicate, we will use the patched model seed predicate, which deems a state-action to be a patch seed if it has been added to either P_{patch} or R_{patch}.

The backup queue had a maximum capacity of 1,000 state-actions. When the backup queue exceeded capacity, the lowest priority element was dropped. Patch Q-learning used a constant learning rate of $\alpha = 0.1$.

For evaluation of the expected reward, we suspend learning at regular intervals and calculate the value function of the policy. For the undiscounted setting, this may be divergent; for the undiscounted setting only, we calculate the expected finite horizon reward:

$$E(\sum_{t=1}^{H} r_t) \tag{16}$$

where r_t is a random variable representing the reward received at time step t from following the policy, and H is the horizon. For the evaluation in the undiscounted experiments in this domain, we use a horizon of $H = 50$. For the discounted settings, we calculate the value function of the policy, discounted appropriately. In both settings, once the value function is calculated, we take the weighted average over the initial state distribution to produce the plots of expected reward.

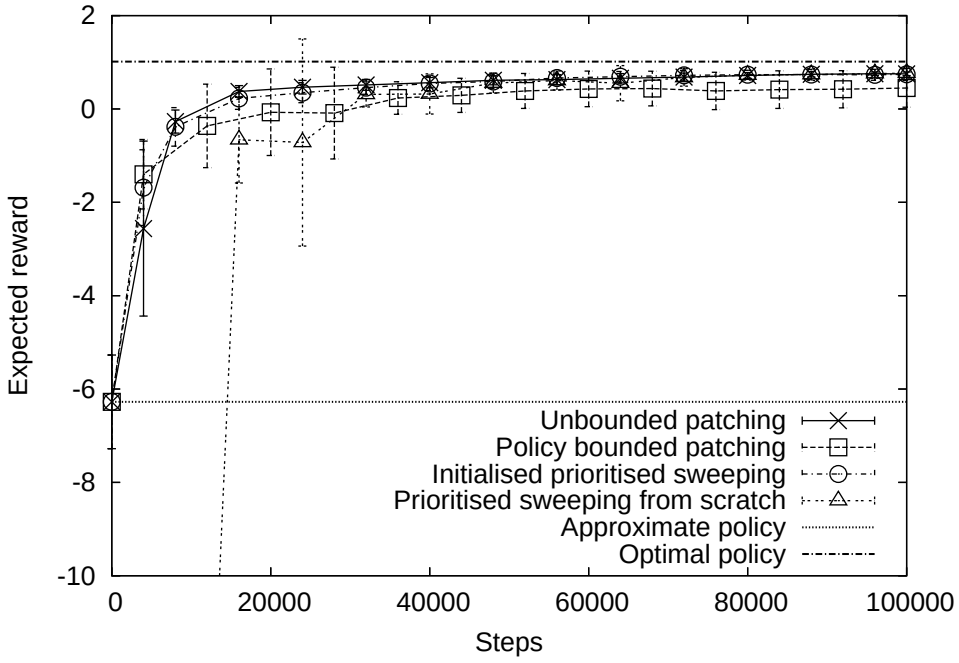
6.2.4 Results for the Undiscounted Setting

We start with the undiscounted setting for this domain. The initial MAXQ solution for this domain used to derive $\hat{Q}$ requires 632 values.

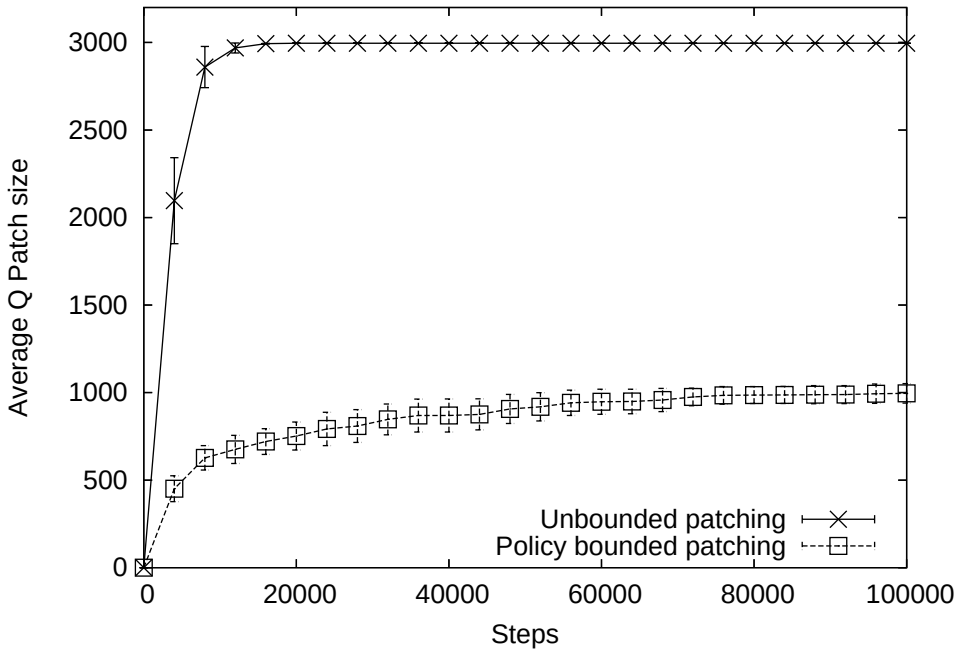
Figure 12(a) compares expected reward for unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch. Initialisation with the approximate solution in this domain appears to be less beneficial than in the coordination example domain, as prioritised sweeping from scratch manages to catch up with the other algorithms reasonably quickly. Unbounded patching and initialised prioritised sweeping produce very similar performance in this domain. Policy bounded patching settles on a less optimal policy than unbounded patching, but learns a much smaller patch – Figure 12(b) shows the average size of Q_{patch} for the two strategies. Unbounded patching consistently patches the entire state-action space[7]; it effectively reverts to prioritised sweeping over the flat problem, as indicated by the similar results to initialised prioritised sweeping. Policy bounded patching produces a slightly weaker policy than unbounded patching, but requires much less space, patching approximately one-third of the state-action space.

Next, we evaluate patching with utility bounding, with Q_{patch} capacities of 1600, 1700, and 1800 (53.33%, 56.67%, and 60.0% of the state-action space respectively). Figure 13 plots the

[7]Except for the 4 terminating state-actions, where the taxi is carrying the passenger and at the passenger's destination, and executes Putdown. These actions deterministically terminate the problem, and since they are not patch seeds, they are never eligible for patching since they cannot be ancestors to patch seeds.



(a) Expected reward, averaged over the initial state distribution, for unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch.



(b) Average Q_{patch} size for unbounded patching and policy bounded patching.

Figure 12: Results for the undiscounted setting of the modified taxi domain.

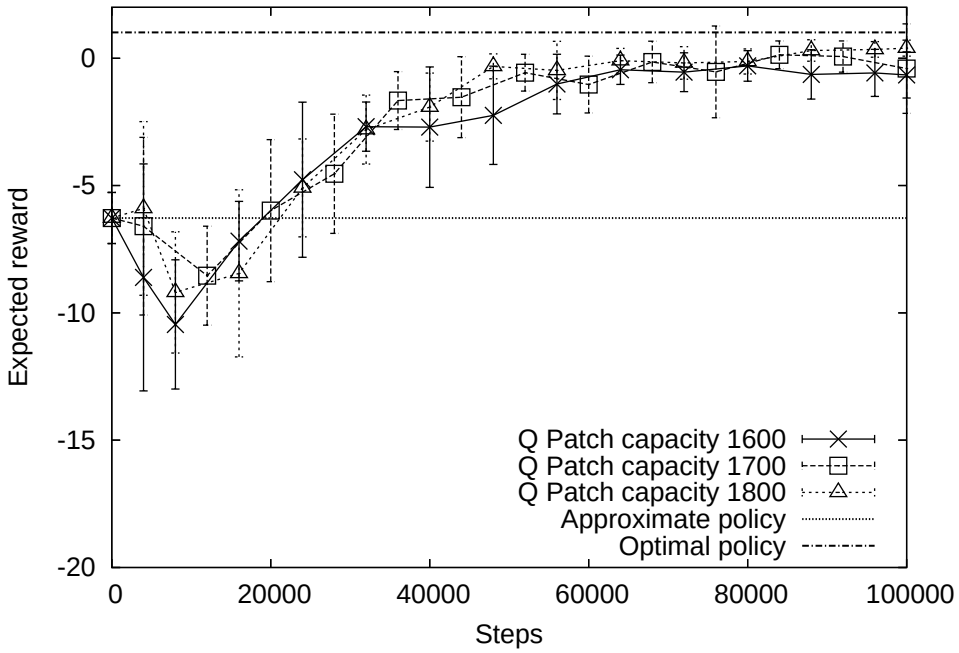


Figure 13: Expected reward, averaged over the initial state distribution, for the undiscounted setting of the modified taxi domain for patching with utility bounding, with Q_{patch} capacities of 1600, 1700, and 1800.

expected reward for patching with utility bounding for these Q_{patch} capacities. In all three cases, there is a drop early on in learning as Q_{patch} capacity is rapidly filled and lower priority Q_{patch} values are forgotten. Expected reward improves afterwards as the set of state-actions in Q_{patch} settles, with higher capacities producing policies closer to global optimality, and with much smaller standard deviation.

Next, we examine patching with both policy and utility bounding enabled. Figure 14 plots the expected reward for Q_{patch} capacities of 800, 900, and 1000 (26.67%, 30.0%, and 33.33% of the state-action space respectively). Patching with policy bounding appears to learn more cautiously than without, as it does not exhibit the drastic drop in expected reward early in learning. One reason for this is that patching with policy bounding is slower to fill its Q_{patch} capacities, as shown by the relatively shallower increase in Q_{patch} size for policy bounded patching in Figure 12(b). Otherwise, similar patterns are observed to patching with utility bounding but without policy bounding – higher Q_{patch} capacities produce more optimal policies with much smaller standard deviation. Enabling policy bounding allows us to use much smaller Q_{patch} capacities than without policy bounding.

Lastly, we evaluate patch Q-learning. Figure 15 plots the expected reward for patch Q-learning with Q_{patch} capacities of 1800, 1900, and 2000 (60.0%, 63.33%, and 66.67% of the state-action space respectively). As with patching with utility bounding but without policy bounding, there is a drop in expected reward early as Q_{patch} reaches capacity and lower priority values are unpatched. As with all other experiments in this setting that used utility bounding, higher Q_{patch} capacities produce policies closer to optimality with lower deviation.

Table 2 summarises the results for this setting of the modified taxi domain.

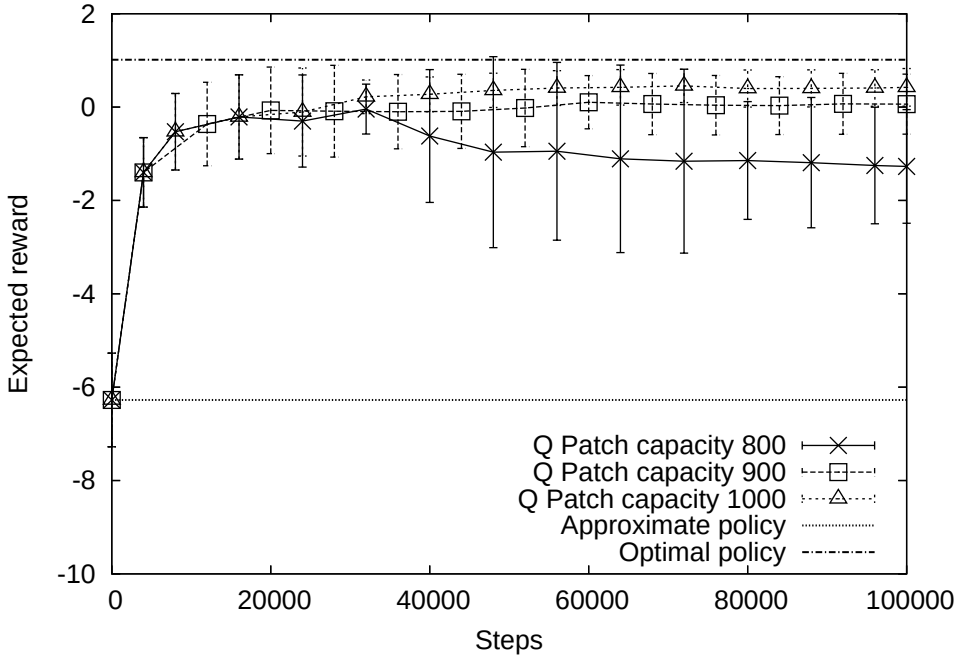


Figure 14: Expected reward, averaged over the initial state distribution, for the undiscounted setting of the modified taxi domain for patching with both policy and utility bounding, with Q_{patch} capacities of 800, 900, and 1000.

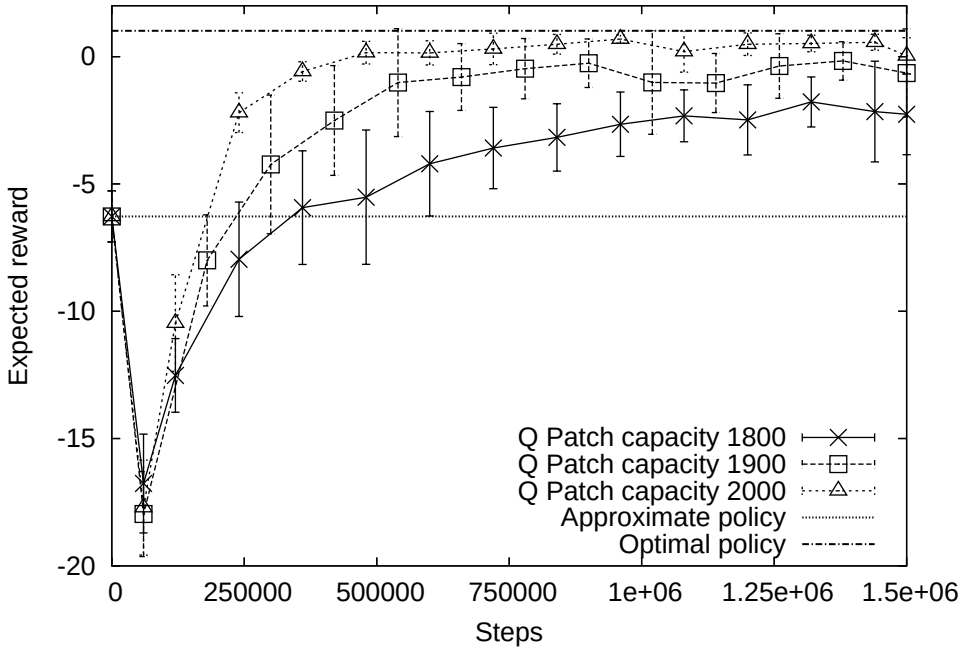


Figure 15: Expected reward, averaged over the initial state distribution, for the undiscounted setting of the modified taxi domain for patch Q-learning, with Q_{patch} capacities of 1800, 1900, and 2000.

Table 2: Summary of results for the undiscounted setting of the modified taxi domain. Figures were taken at the end of 100,000 steps for unbounded patching, patching with policy and/or utility bounding, and both instances of prioritised sweeping. Figures were taken at the end of 1,500,000 steps for patch Q-learning. 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 ($\hat{Q}$)	-6.27 ± 1.00	632
Optimal flat	1.02	3000
Prioritised sweeping from scratch	0.76 ± 0.10	3000
Initialised prioritised sweeping	0.74 ± 0.13	3000
Unbounded patching	0.75 ± 0.04	632 + 2996
With policy bounding	0.45 ± 0.41	632 + 995.70 ± 55.36
With utility bounding		
– Q_{patch} capacity 1600	-0.65 ± 0.91	632 + 1600
– Q_{patch} capacity 1700	-0.41 ± 1.76	632 + 1700
– Q_{patch} capacity 1800	0.40 ± 0.31	632 + 1800
With policy and utility bounding		
– Q_{patch} capacity 800	-1.27 ± 1.22	632 + 800
– Q_{patch} capacity 900	0.06 ± 0.64	632 + 900
– Q_{patch} capacity 1000	0.42 ± 0.40	632 + 974.00 ± 31.88
Patch Q-learning		
– Q_{patch} capacity 1800	-2.26 ± 1.59	632 + 1800
– Q_{patch} capacity 1900	-0.64 ± 1.73	632 + 1900
– Q_{patch} capacity 2000	0.03 ± 0.72	632 + 2000

6.2.5 Results for the Discounted Terminating Setting

The next set of experiments modifies the optimality criteria by incorporating discounting, with discount factor $\gamma = 0.95$. The initial MAXQ solution is adjusted accordingly to accommodate discounting, and now requires 1208 values.

First, we compare unbounded patching, policy bounded patching, and the two instances of prioritised sweeping. Figure 16(a) shows the expected reward for these algorithms. As with the undiscounted setting, prioritised sweeping from scratch draws level with the other algorithms fairly quickly, and all algorithms produce policies of similar quality. The difference between the two patching algorithms is smaller in this setting than in the undiscounted setting. However, in terms of Q_{patch} size, as shown in Figure 16(b), unbounded patching once again consistently patches the entire state-action space[8], while policy bounding covers approximately one third of the state-action space with Q_{patch}.

Next, we examine patching with utility bounding but without policy bounding, for Q_{patch} capacities of 1800, 1900, and 2000 (60.0%, 63.33%, and 66.67% of the state-action space respectively). Figure 17 plots the expected reward for these capacities of Q_{patch}. As with the undiscounted setting of this domain, higher Q_{patch} capacities result in higher expected reward with lower standard deviation. A similar drop in expected reward early in learning as the Q_{patch} capacities are reached is also observed in this setting.

Next, we evaluate patching with both policy and utility bounding. Figure 18 plots the expected reward for patching with both bounding strategies, with Q_{patch} capacities of 600, 800, and 1000 (20.0%, 26.27%, and 33.33% of the state-action space respectively). As with the undiscounted setting of this domain, patching with both policy and utility bounding appears to learn more cautiously than with utility bounding alone, and avoids the drop in policy value early in learning. Also, higher Q_{patch} capacities produce better policies with smaller standard deviation, and policy bounding allows us to use much smaller Q_{patch} capacities than without policy bounding.

Lastly, we try patch Q-learning with Q_{patch} capacities of 1800, 1900, and 2000 (60.0%, 63.33%, and 66.67% of the state-action space respectively). Figure 19 shows the expected reward in this setting, displaying the familiar pattern of higher expected reward and lower standard deviation for higher Q_{patch} capacities.

Table 3 summarises the results for this setting of the modified taxi domain.

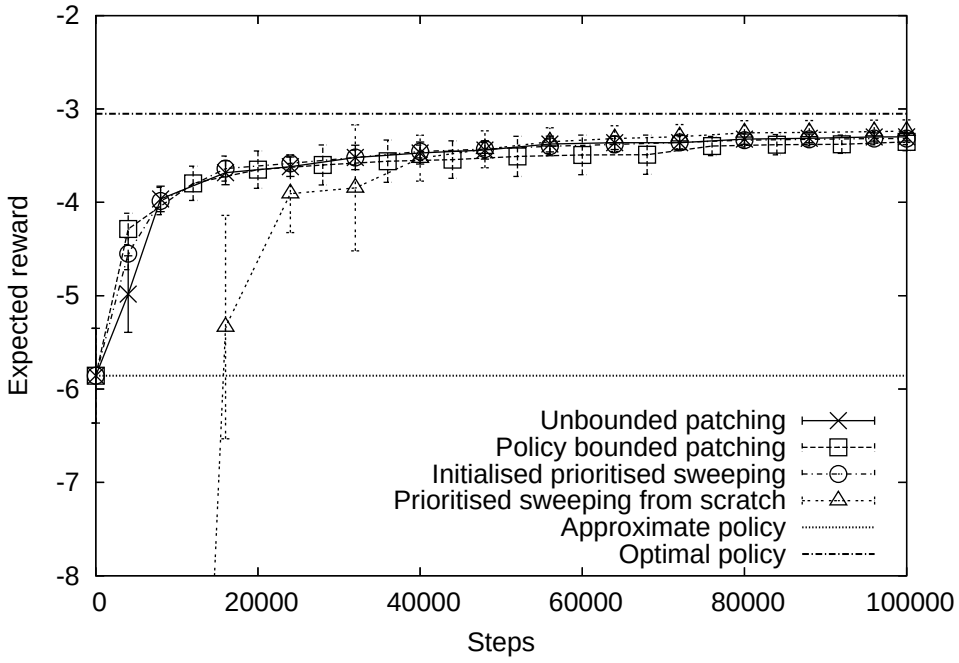
6.2.6 Results for the Discounted Continuing Setting

The last set of experiments in this domain evaluates patching when the task is continuing. In this setting the discount factor is $\gamma = 0.95$, and a new passenger appears at a random taxi stand on successful delivery. The initial MAXQ solution is also adjusted, and now requires 1308 values.

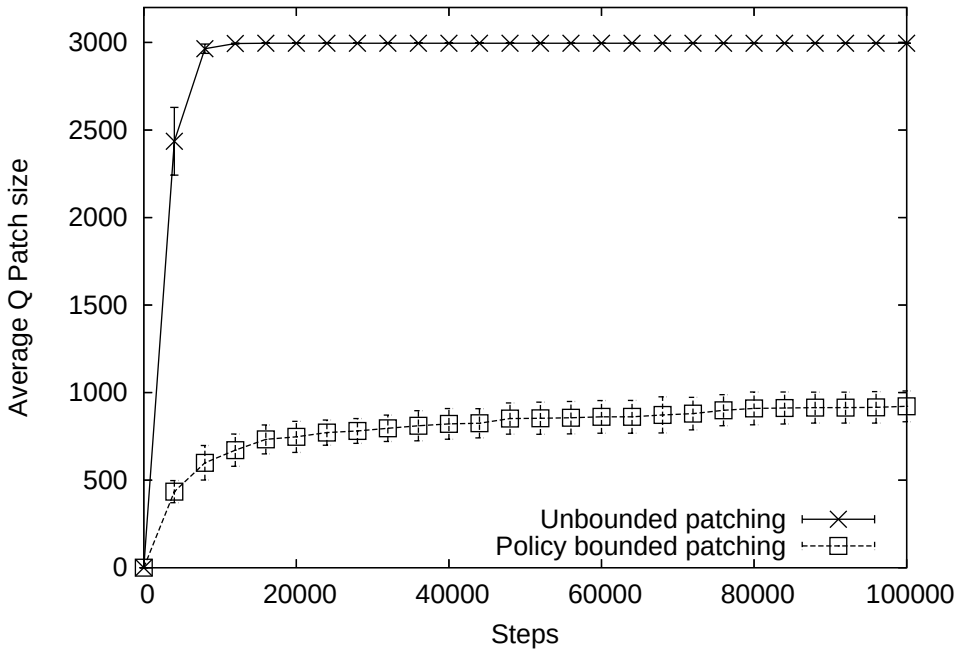
First, we compare unbounded patching, policy bounded patching, and the two instances of prioritised sweeping. Figure 20(a) plots the expected reward for these algorithms. As with the other settings, prioritised sweeping from scratch catches up with the other algorithms fairly quickly, and all algorithms learn policies with similar expected reward. Once again, policy bounded patching produces a slightly weaker policy than unbounded patching, but as shown in Figure 20(b), it does so with a much smaller Q_{patch} size. Unbounded patching consistently patches the entire state-action space[9], while policy bounded patching patches approximately one third of the state-action space.

[8] As in the undiscounted setting, with the exception of the 4 state-actions that terminate the problem.

[9] In this setting, since there are no terminating state-actions, unbounded patching actually patches all 3000 state-actions.



(a) Expected reward, averaged over the initial state distribution, for unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch.



(b) Average Q_{patch} size for unbounded patching and policy bounded patching.

Figure 16: Results for the discounted terminating setting of the modified taxi domain.

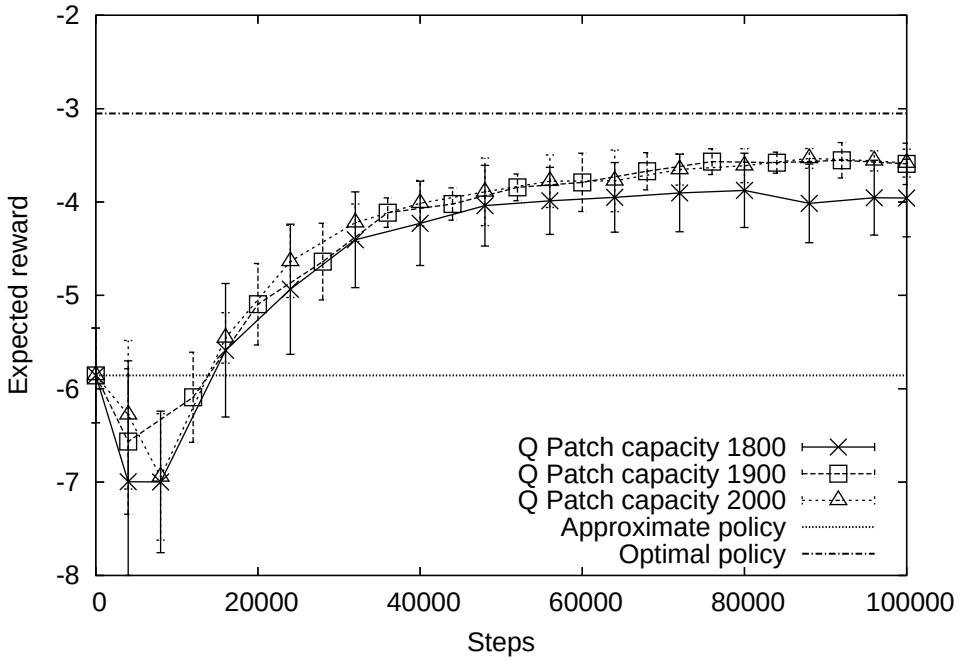


Figure 17: Expected reward, averaged over the initial state distribution, for the discounted terminating setting of the modified taxi domain for patching with utility bounding, with Q_{patch} capacities of 1800, 1900, and 2000.

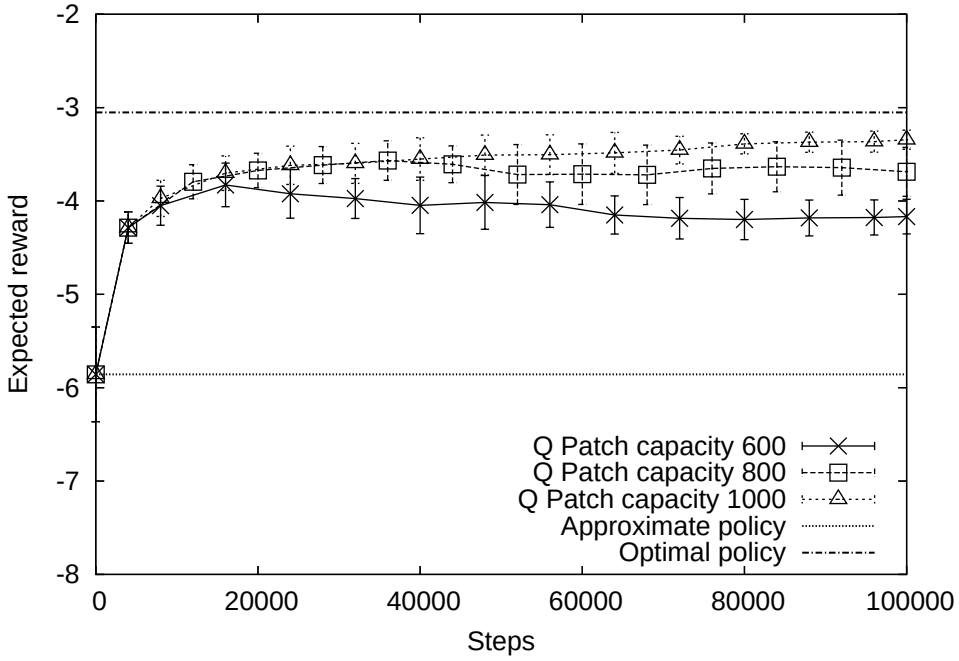


Figure 18: Expected reward, averaged over the initial state distribution, for the discounted terminating setting of the modified taxi domain for patching with policy and utility bounding, with Q_{patch} capacities of 600, 800, and 1000.

Table 3: Summary of results for the discounted terminating setting of the modified taxi domain. Figures were taken at the end of 100,000 steps for unbounded patching, patching with policy and/or utility bounding, and both instances of prioritised sweeping. Figures for patch Q-learning were taken at the end of 1,500,000 steps. 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 ($\hat{Q}$)	-5.86 ± 0.51	1208
Optimal flat	-3.05	3000
Prioritised sweeping from scratch	-3.24 ± 0.12	3000
Initialised prioritised sweeping	-3.32 ± 0.07	3000
Unbounded patching	-3.30 ± 0.04	1208 + 2996
With policy bounding	-3.35 ± 0.09	1208 + 921.60 ± 87.98
With utility bounding		
– Q_{patch} capacity 1800	-3.96 ± 0.42	1208 + 1800
– Q_{patch} capacity 1900	-3.59 ± 0.22	1208 + 1900
– Q_{patch} capacity 2000	-3.58 ± 0.15	1208 + 2000
With policy and utility bounding		
– Q_{patch} capacity 600	-4.17 ± 0.19	1208 + 600
– Q_{patch} capacity 800	-3.69 ± 0.26	1208 + 798.40 ± 3.90
– Q_{patch} capacity 1000	-3.34 ± 0.10	1208 + 917.20 ± 83.24
Patch Q-learning		
– Q_{patch} capacity 1800	-4.92 ± 0.58	1208 + 1800
– Q_{patch} capacity 1900	-3.84 ± 0.27	1208 + 1900
– Q_{patch} capacity 2000	-3.35 ± 0.08	1208 + 2000

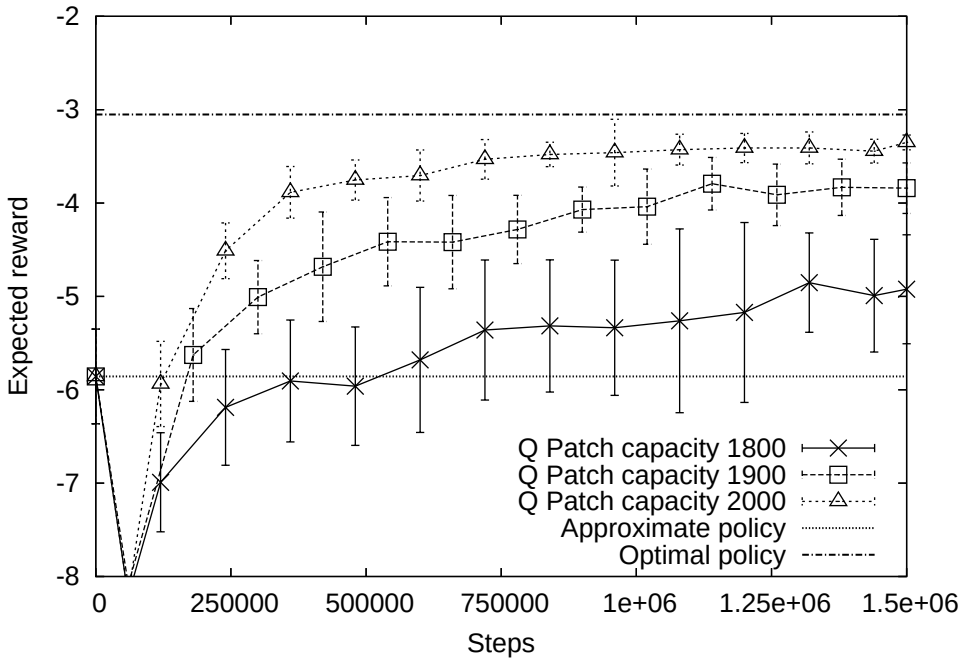


Figure 19: Expected reward, averaged over the initial state distribution, for the discounted terminating setting of the modified taxi domain for patch Q-learning with Q_{patch} capacities of 1800, 1900, and 2000.

Next, we examine the performance of patching with utility bounding, with Q_{patch} capacities of 2100, 2200, and 2300 (70.0%, 73.33%, and 76.67% of the state-action space respectively). Figure 21 plots the expected reward for this setting. As in the other settings, higher Q_{patch} capacities produce higher expected reward. However, this setting appears to require larger Q_{patch} capacities than other settings for utility bounding alone.

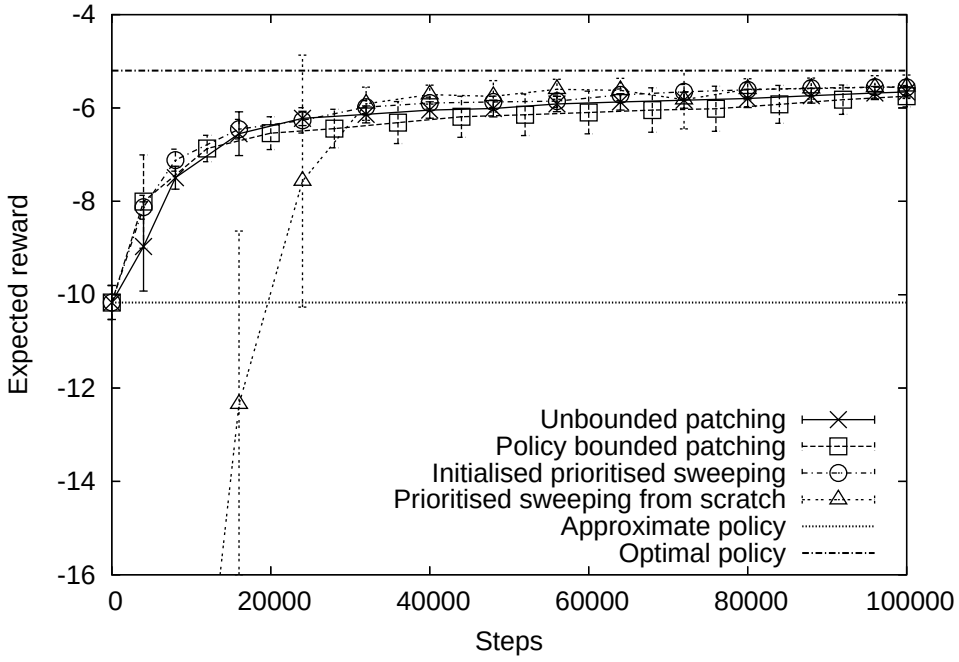
Next, we apply both policy and utility bounding. Figure 22 shows the expected reward for patching with both policy and utility bounding, with Q_{patch} capacities of 600, 800, and 1000 (20.0%, 26.27%, and 33.33% of the state-action space respectively). As in the other settings, policy bounding appears to cause learning to be more cautious than without policy bounding. Also, while patching with utility bounding alone required larger Q_{patch} capacities than in the undiscounted and discounted terminating settings of this domain, combining the two bounding strategies produces reasonably good results without any increase.

Lastly, we try patch Q-learning with Q_{patch} capacities of 2100, 2200, and 2300 (70.0%, 73.33%, and 76.67% of the state-action space respectively). Figure 23 plots the results for patch Q-learning for this setting. Once again, we observe the trend of higher expected reward and lower standard deviation for higher Q_{patch} capacities.

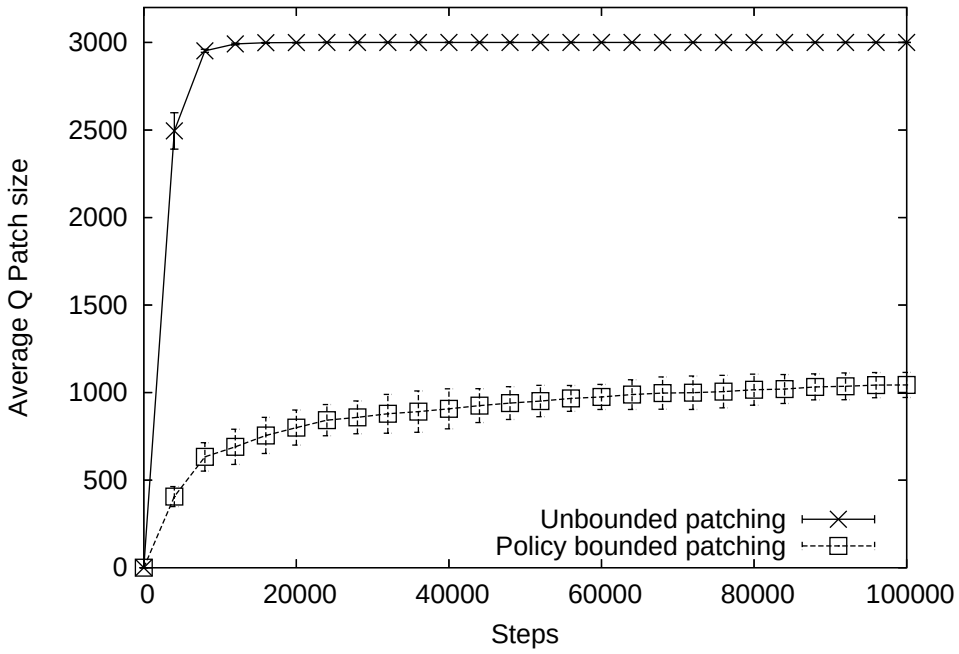
Table 4 summarises the results for this setting of the modified taxi domain.

6.2.7 Discussion

In this set of experiments, we demonstrated that patching is able to handle undiscounted and discounted optimality criteria, as well as terminating and continuing tasks, using a MAXQ hierarchical solution as the base approximation. We also demonstrated patching in a domain with stochastic transitions and rewards, with both components of the model requiring statistical sampling and patching.



(a) Expected reward, averaged over the initial state distribution, for unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch.



(b) Average Q_{patch} size for unbounded patching and policy bounded patching.

Figure 20: Results for the discounted continuing setting of the modified taxi domain.

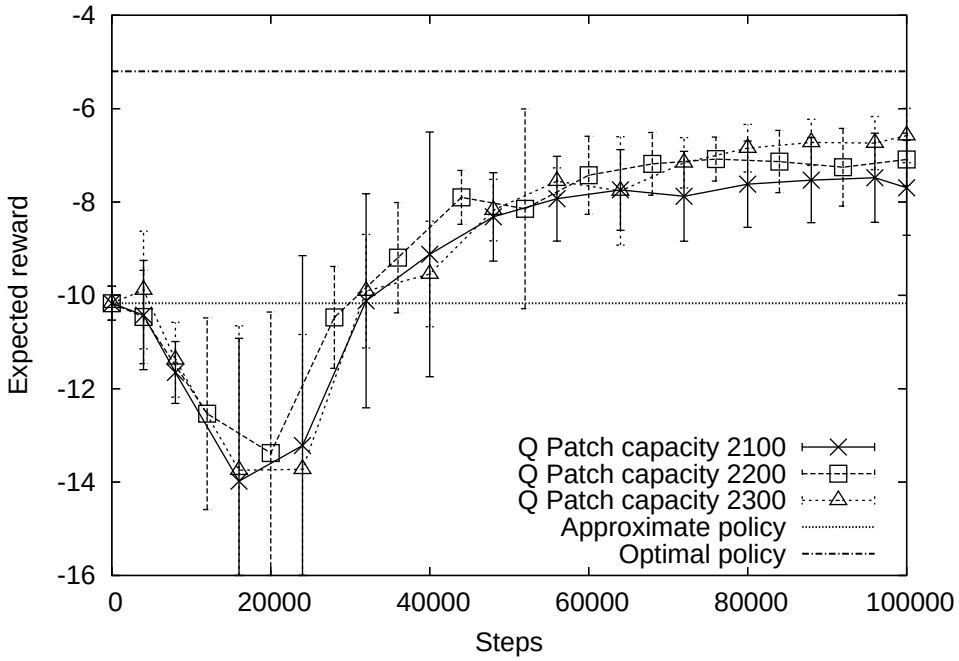


Figure 21: Expected reward, averaged over the initial state distribution, for the discounted continuing setting of the modified taxi domain for patching with utility bounding, with Q_{patch} capacities of 2100, 2200, and 2300.

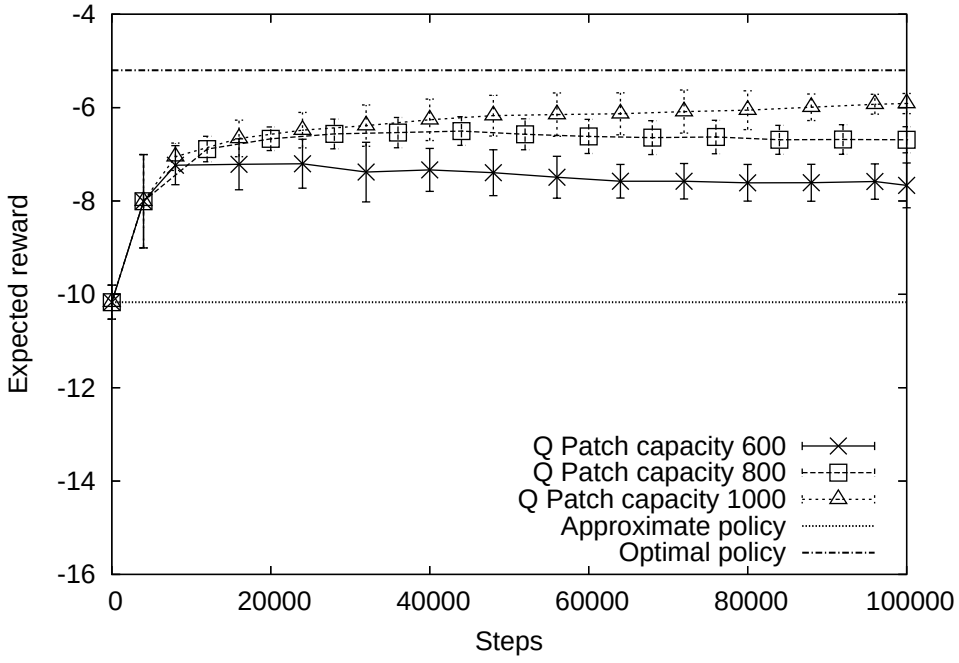


Figure 22: Expected reward, averaged over the initial state distribution, for the discounted continuing setting of the modified taxi domain for patching with both policy and utility bounding, with Q_{patch} capacities of 600, 800, and 1000.

Table 4: Summary of results for the discounted continuing setting of the modified taxi domain. Figures were taken at the end of 100,000 steps for unbounded patching, patching with policy and/or utility bounding, initialised prioritised sweeping, and prioritised sweeping from scratch. Figures for patch Q-learning were taken at the end of 1,500,000 steps. 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 ($\hat{Q}$)	-10.17 ± 0.36	1308
Optimal flat	-5.20	3000
Prioritised sweeping from scratch	-5.55 ± 0.25	3000
Initialised prioritised sweeping	-5.56 ± 0.16	3000
Unbounded patching	-5.66 ± 0.11	1308 + 3000
With policy bounding	-5.75 ± 0.23	1308 + 1043.90 ± 71.47
With utility bounding		
– Q_{patch} capacity 2100	-7.70 ± 1.02	1308 + 2100
– Q_{patch} capacity 2200	-7.09 ± 0.56	1308 + 2200
– Q_{patch} capacity 2300	-6.58 ± 0.58	1308 + 2300
With policy and utility bounding		
– Q_{patch} capacity 600	-7.67 ± 0.48	1308 + 600
– Q_{patch} capacity 800	-6.69 ± 0.28	1308 + 800
– Q_{patch} capacity 1000	-5.91 ± 0.22	1308 + 990.90 ± 15.05
Patch Q-learning		
– Q_{patch} capacity 2100	-8.71 ± 1.07	1308 + 2100
– Q_{patch} capacity 2200	-6.68 ± 0.35	1308 + 2200
– Q_{patch} capacity 2300	-5.99 ± 0.43	1308 + 2300

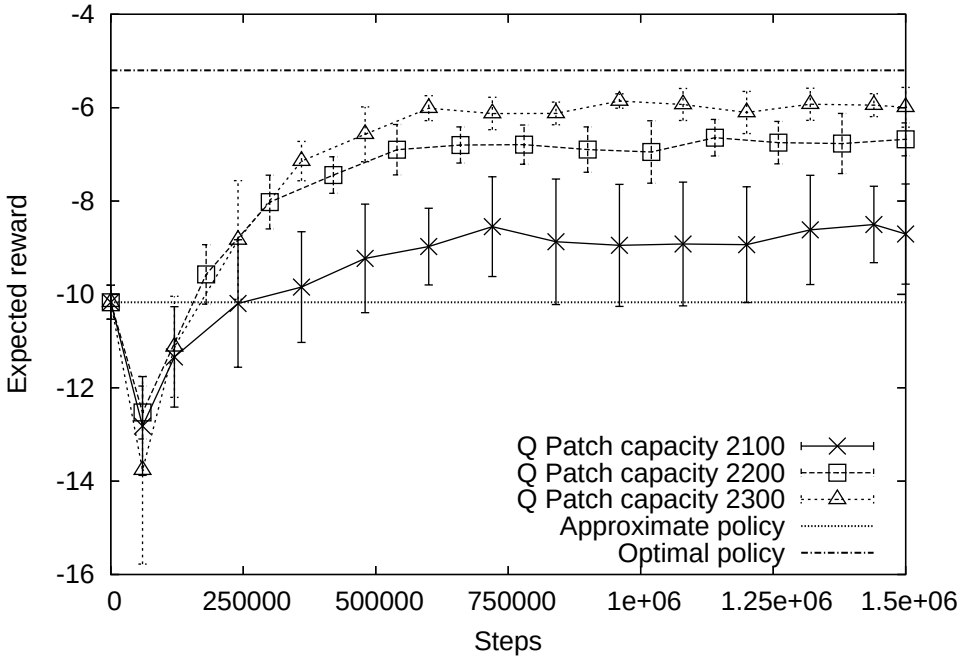


Figure 23: Expected reward, averaged over the initial state distribution, for the discounted continuing setting of the modified taxi domain for patch Q-learning, with Q_{patch} capacities of 2100, 2200, and 2300.

6.3 Taxi with Fuel

The taxi with fuel domain was created by Dietterich [1998] to illustrate the *hierarchical credit assignment problem*. The solution proposed by Dietterich was for the programmer to manually separate the reward to assign credit or blame to the appropriate sub-tasks in the task hierarchy. Our approach will be very different, by starting with a MAXQ hierarchy for the original taxi problem, and then patching it to handle fuel. In this domain, we will evaluate unbounded patching, and patching with policy and utility bounding. The total size of the taxi with fuel problem is 45,500 state-actions.

6.3.1 Domain Description

The taxi with fuel domain, shown in Figure 24, extends the original taxi domain as follows.

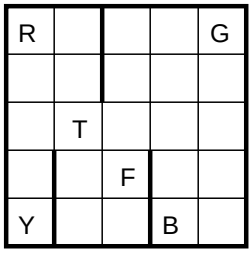


Figure 24: The taxi with fuel domain. R, G, B, and Y indicate the taxi stands, T indicates the taxi location, F indicates the refill station.

States now include a fourth variable that indicates the remaining fuel level. This can take integer values between 0 and 12 inclusive. Each time a navigation action is executed, the fuel level decreases by 1, except if the fuel level is 0, in which case the problem terminates and a reward of -20 is received.

An additional Refill action is available, which, if executed at the refill station marked by F in Figure 24, sets the remaining fuel level to 12, with the usual reward of -1. In all other situations, Refill is illegal and has no effect, and receives a reward of -10.

The task is discounted, with discount factor $\gamma = 0.95$.

6.3.2 Experiment Setup and Parameters

We initialise $\hat{Q}$ as with the modified taxi problem, by learning a MAXQ value function for the original taxi problem with discounting, and then derive $\hat{Q}$. $\hat{Q}$ requires 1208 values. Note that Refill is not included anywhere in this task hierarchy, since Refill is not included in the original taxi problem. Thus, for Refill, $\hat{Q}$ is calculated on demand as its one-step model-based value:

$$\hat{Q}(s, \mathsf{Refill}) = \hat{R}(s, \mathsf{Refill}) + \gamma \sum_{s' \in S} \hat{P}(s, \mathsf{Refill}, s') V^{MAXQ}(s') \tag{17}$$

where $V^{MAXQ}(s')$ is the MAXQ hierarchical value for state s'.

The approximate transition model for fuel is given as follows. For any pair of fuel level f and action a, the next fuel level is predicted to be $(f-1)$ if a is a navigation action, and f otherwise. $\hat{P}$ is calculated as the cross product of P for the original taxi domain and this approximate fuel model. An exception is when Refill is executed, in which case $\hat{P}$ predicts a deterministic self-transition. Finally, when calculating $\hat{P}^{-1}$, it is assumed that there are no predecessors to states where the fuel level is at its maximum value of 12.

$\hat{R}$ is taken as R for the original taxi domain, *i.e.* it is initially assumed that the fuel level has no effect on the reward. An exception is when Refill is executed, in which case $\hat{R}$ predicts a reward of -10 for all states. Combined with $\hat{P}$, this effectively means that Refill is always predicted to cause a deterministic self-transition with large negative reward. This choice of $\hat{P}$ and $\hat{R}$ simplifies $\hat{Q}(s, \mathsf{Refill})$ (Equation 17) to:

$$\hat{Q}(s, \mathsf{Refill}) = -10 + \gamma V^{MAXQ}(s) \tag{18}$$

For statistical testing of the transition model, the χ^2 test required 20 successor state samples before testing $\hat{P}$ for a particular state-action, with a null hypothesis probability cut-off of 0.00001. For patching $\hat{R}$, because rewards in this domain are deterministic, we patched those state-actions deterministically. The transition and reward sampling set capacities both had capacities of 100 state-actions. Other capacities produce very similar results in this domain.

The patch seed predicate is the patched model seed predicate, triggering where inaccuracies are found in $\hat{P}$ and $\hat{R}$. Specifically, $\hat{P}$ and $\hat{R}$ will be found to be inaccurate when:

- the taxi runs out of fuel and the problem terminates, receiving a reward of -20;
- the taxi happens to execute Refill at the refill station and the fuel level is set to maximum (instead of deterministically self-transitioning), and a reward of -1 is received instead of -10.

The backup queue had a maximum capacity of 5000 state-actions. When the backup queue exceeded capacity, the lowest priority element was dropped.

For evaluation in this domain, we suspend learning at regular intervals during learning and calculate the value function of the policy. We then take the weighted average over the initial state distribution to produce the plots of expected reward.

6.3.3 Results

We start with a comparison of unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch. Figure 25(a) plots the expected reward for these algorithms. All algorithms exhibit a significant drop in expected reward in the early stages of learning. This drop occurs because all algorithms need to learn values associated with running out of fuel. Consider a state where the fuel level is 0 and the taxi is far from both its destination and the refill station. All navigation actions from this state will result in termination of the problem with a large penalty. Since the penalty for running out of fuel is much worse than the penalty for an illegal Pickup, Putdown, or Refill, if the navigation actions are sampled first, an illegal action may be temporarily set as the greedy policy until its values is updated. Since an illegal action deterministically self transitions with large negative reward, this causes large drop in expected reward if the policy is evaluated when this is the case. Initialised prioritised sweeping appears to do particularly poorly in this domain – the combination of the initial Q values and patched model appear to hamper learning dramatically [10]. Figure 25(b) plots the average Q_{patch} size – as usual, unbounded patching consistently patches the entire state-action space[11]. Policy bounding reduces Q_{patch} growth, but still covers approximately two thirds of the state-action space.

Next, we evaluate patching with utility bounding. Figure 26 plots the expected reward for patching with utility bounding with Q_{patch} capacities of 25000, 30000, and 35000 (54.95%, 65.93%, and 76.92% of the state-action space respectively). Compared to unbounded patching, the early drop in expected reward is much deeper for the two smaller Q_{patch} capacities, and similar for the largest of the three tested capacities. However, in all cases, it takes much longer than unbounded patching for expected reward to recover. Eventually, the policies improve to perform reasonably well, but still below optimality.

Next, we enforce both policy and utility bounding. Figure 27 shows the expected reward for patching with both bounding strategies, with Q_{patch} capacities of 25000, 27500, and 30000 (54.95%, 60.44%, and 65.93% of the state-action space respectively). With policy bounding enabled, it takes fewer steps for learning to recover from the early drop in policy quality, but also settles well below optimality, and lower than patching with utility bounding alone. This occurs despite using similar Q_{patch} capacities to patching with utility bounding alone – in the modified taxi domain, policy bounding allowed us to use much smaller Q_{patch} capacities with utility bounding, but this is not the case here.

Table 5 summarises the results for this domain.

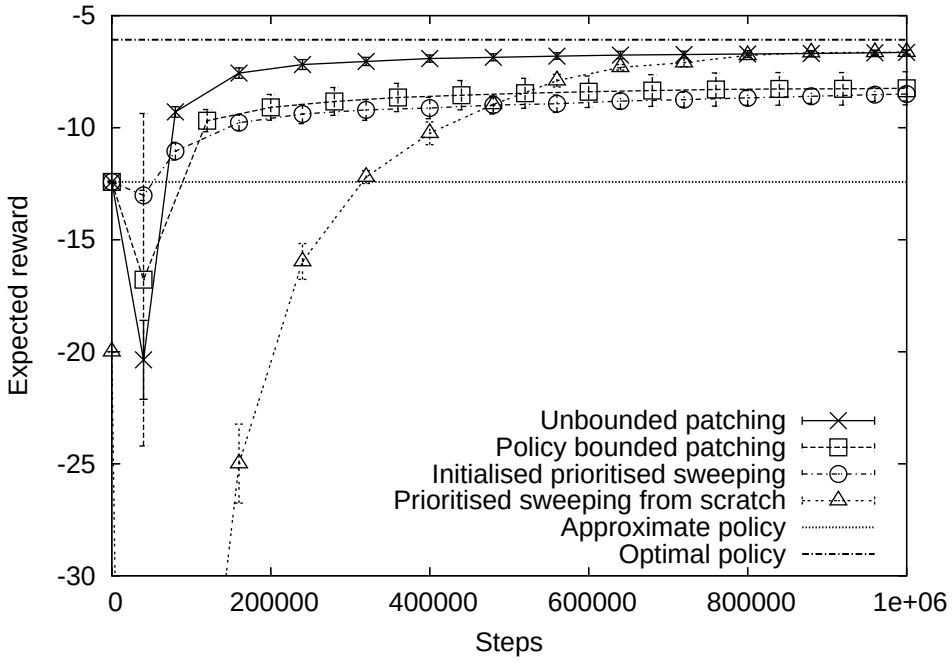
6.3.4 Discussion

The taxi with fuel domain is not solved very well by patching, and illustrates the limitations of patching when the initial approximation is poor. Since the taxi with fuel domain is an extension of the original taxi domain, it is intuitively appealing to assume that a solution to the original domain may be easily adapted to solve the extended domain. This turned out to perform poorly in practice, because the added state variable is actually a critical factor in decision making for most of the state space.

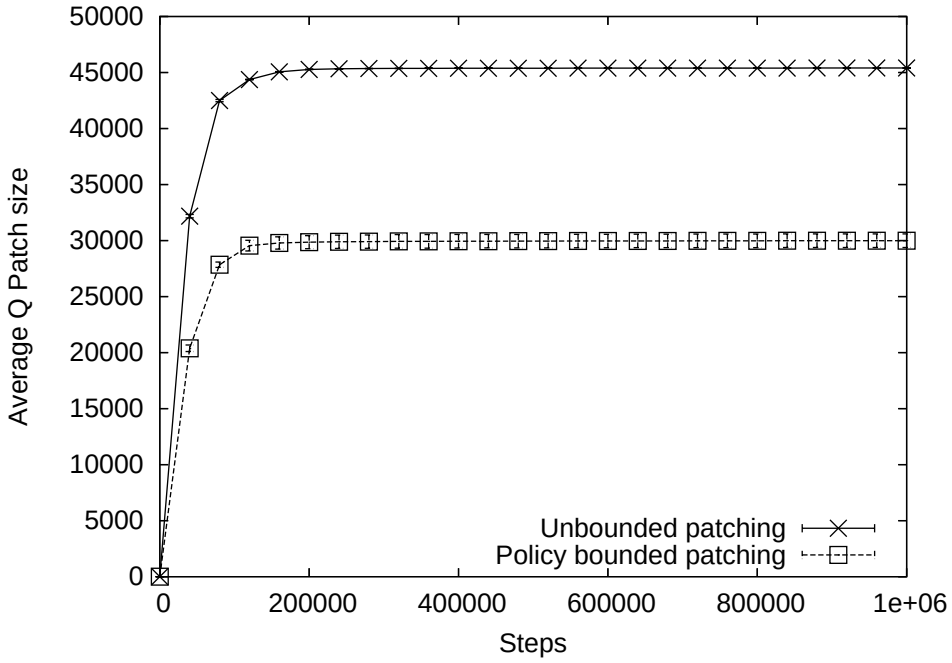
A major problem faced in this domain is that the areas of the problem that require patching

[10] Longer runs of the experiment suggest that it does eventually converge to optimality, but it takes much longer than learning from scratch. It also appears that changing the backup queue updates to add priorities for existing state-actions in the backup queue (as suggested by Andre et al. [1998] and as used by unbounded patching) rather than taking the maximum priority (as specified by Moore and Atkeson [1993]) eliminates this problem.

[11] Except for the terminating state-actions.



(a) Expected reward, averaged over the initial state distribution, for unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch.



(b) Average Q_{patch} size for unbounded patching and policy bounded patching.

Figure 25: Results for the taxi with fuel domain.

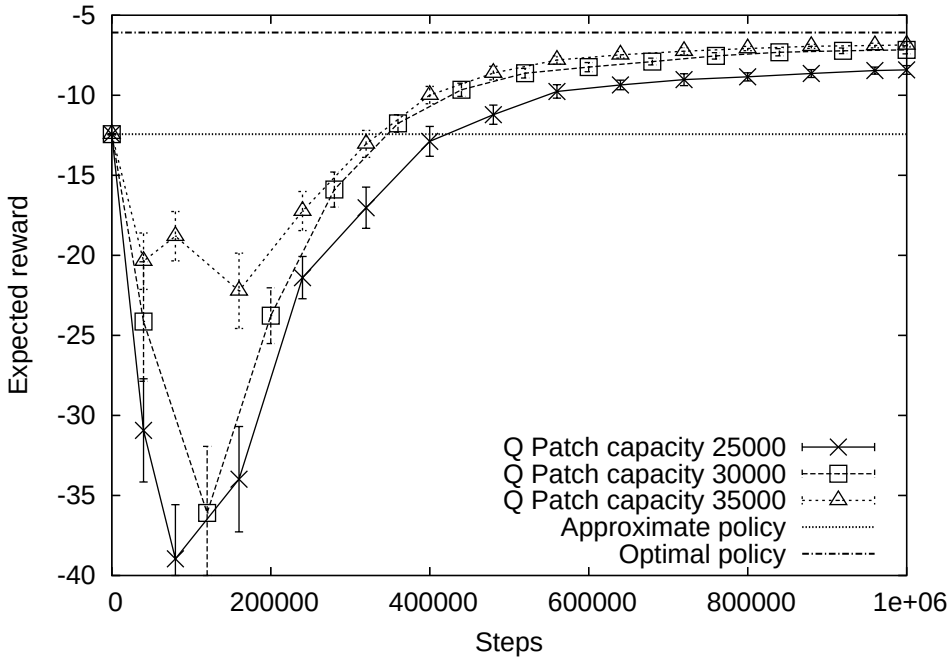


Figure 26: Expected reward, averaged over the initial state distribution, for the taxi with fuel domain for patching with utility bounding, with Q_{patch} capacities of 25000, 30000, and 35000.

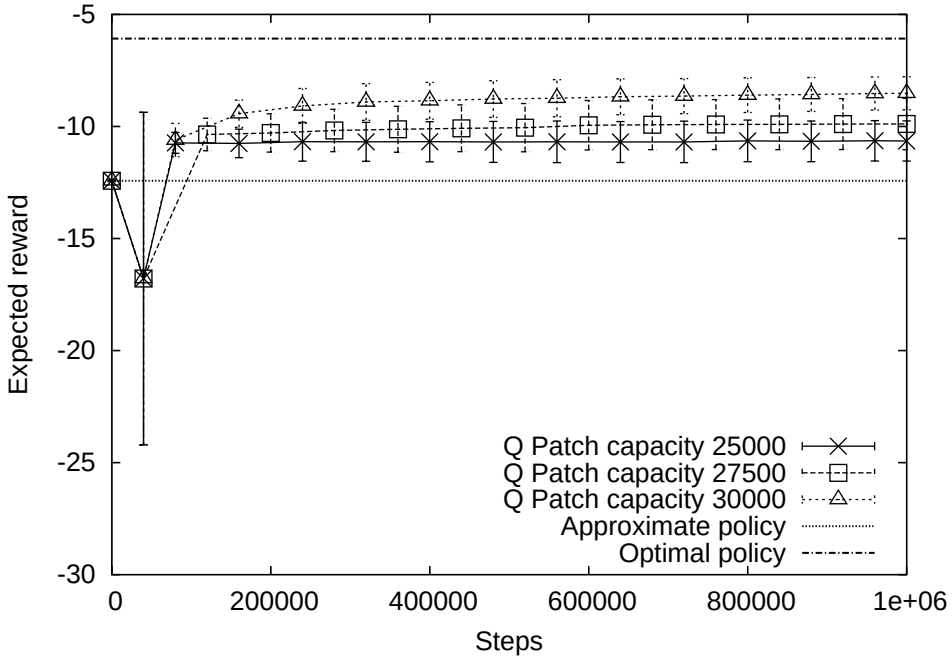


Figure 27: Expected reward, averaged over the initial state distribution, for the taxi with fuel domain for patching with both policy and utility bounding, with Q_{patch} capacities of 25000, 27500, and 30000.

Table 5: Summary of results for the taxi with fuel domain. Figures were taken at the end of 1,000,000 steps for all learning 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 ($\hat{Q}$)	-12.42 ± 0.03	1208
Optimal flat	-6.08	45500
Prioritised sweeping from scratch	-6.63 ± 0.09	45500
Initialised prioritised sweeping	-8.50 ± 0.31	45500
Unbounded patching	-6.65 ± 0.12	1208 + 45400.90 ± 3.14
With policy bounding	-8.24 ± 0.74	1208 + 29989.20 ± 607.87
With utility bounding		
– Q_{patch} capacity 25000	-8.41 ± 0.27	1208 + 25000
– Q_{patch} capacity 30000	-7.16 ± 0.26	1208 + 30000
– Q_{patch} capacity 35000	-6.85 ± 0.13	1208 + 35000
With policy and utility bounding		
– Q_{patch} capacity 25000	-10.65 ± 0.90	1208 + 25000
– Q_{patch} capacity 27500	-9.89 ± 1.15	1208 + 27500
– Q_{patch} capacity 30000	-8.52 ± 0.74	1208 + 29725.20 ± 315.34

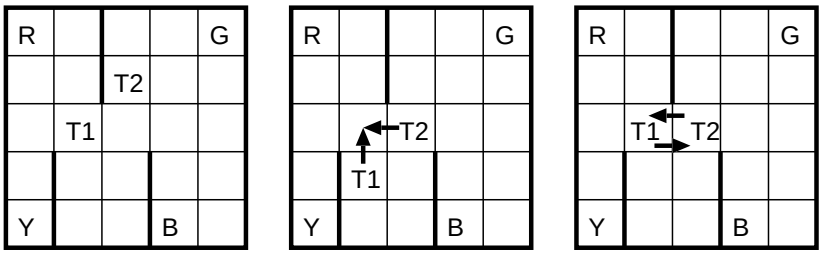


Figure 28: (a) The multi-taxi problem. R, G, B, and Y indicate the taxi stands, T1 and T2 indicate the taxi locations. (b) and (c) show collisions in the multi-taxi problem.

are far away from the patch seeds – when the taxi is about to run out of fuel, in many cases it will be too late to return to the refill station. In general, the patch needs to stretch back from the point of running out of fuel to where it is possible to return to the refill station, which may be many steps back. This relation is hand-engineered into the hierarchy in MAXQ by including all variables at the top level of the hierarchy and manually decomposing the reward function. It is harder to learn by patching without this additional domain knowledge, since it needs to learn this relation step by step from the patch seeds.

This also suggests one reason for the relatively poor performance of policy bounding in this domain – policy bounding is designed to cut off patch growth to prevent it from growing too far, and is suited to domains where local work-arounds near the patch seeds are sufficient. In this domain, local work-arounds are usually inadequate, and adjustments to the policy need to be stretched much further back.

Another problem is that the model is not initialised with domain knowledge to generalise over patch seeds. Without this prior knowledge, patching needs to learn the concept of running out of fuel separately for each possible context in which it may occur, that is, all combinations of taxi location, passenger location, passenger destination, and navigation action when the fuel level is 0. Similarly, the concept of refilling the fuel tank needs to be discovered for all contexts of passenger location, passenger destination, and fuel level when the taxi is at the refill station and executes Refill. This suggests the possible viability of an abstracted or generalising representation for the patches for the model, but this must be applied carefully to avoid over-generalising.

6.4 Multi-taxi

The last set of experiments will examine patching in the multi-taxi problem, an extension of the taxi problem to two taxis and two passengers. This domain is much larger than the previous domains and will use more sophisticated approximations and patching. In this domain, we will evaluate unbounded patching, and patching with policy and utility bounding. The total size of this problem is 17,434,200 state-actions.

6.4.1 Domain Description

The multi-taxi problem, shown in Figure 28(a), extends the original taxi problem to two taxis and two passengers. States are described by six variables: two variables for the taxi locations and four variables to describe the passenger locations and destinations. The range of possible passenger location values is increased by two – the passenger location may be one of the four

stands, In_Taxi_1, In_Taxi_2, or Delivered [12].

Unlike some previous multiple taxi formulations of the taxi problem, the two taxis may not occupy the same location. Both passengers may occupy the same taxi stand, but both passengers may not be in the same taxi at the same time (*i.e.* a taxi may only carry one passenger at a time).

An action for one taxi is now extended by adding a Null action to the six actions from the original taxi domain. As its name suggests, the Null action never has any effect and deterministically self-transitions. An action in the multi-taxi problem is a pair of actions, one for each taxi. The transition model for the multi-taxi problem is generally the product of two transition models for the original taxi problem, with some modifications.

Navigation actions in the multi-taxi domain may cause collisions. If a collision would occur, neither taxi's location is changed. Collisions occur when the effect of the action would cause both taxis to occupy the same location (see Figure 28(b)), or to swap positions when they are next to each other (see Figure 28(c)).

The Pickup action is now legal when the taxi executing it is at a taxi stand occupied by either passenger. If both passengers are at the same taxi stand, then the Pickup action will affect the first passenger (according to the state variable ordering). The affected passenger's location is then accordingly set to either In_Taxi_1 or In_Taxi_2. The resulting change in passenger location and delivery is not affected by collisions, *i.e.* collisions affect the next taxi locations, but not the passenger variables.

At each time step, the reward is -1 for each undelivered passenger, plus an additional reward of -10 for each illegal Pickup or Putdown action. Also, the $+20$ reward on successful delivery in the original taxi problem has been removed.

The initial taxi locations are selected randomly at the start of each trial from all non-overlapping pairs of locations in the world. The passenger locations and destinations are each initialised randomly from the four taxi stands. The task is undiscounted, and terminates when both passengers have been delivered.

6.4.2 Experiment Setup and Parameters

Call the task of delivering one passenger with one taxi the *single taxi problem*. Our approach to this problem domain will be to use the solution to the single taxi problem to construct the approximations. Intuitively, the multi-taxi problem is roughly equal to two single taxi problems running concurrently and independently. Patching will then be applied to correct the approximation where the independence assumption does not hold.

In order to leverage the supplied single taxi solution, we use a hand-crafted allocation function that maps multi-taxi states to pairs of states from the single taxi problem. Intuitively, states in the single taxi problem are interpreted as representative of taxi to passenger allocations, by matching up the relevant taxi and passenger state variables. The allocation function then selects the optimal allocation according to the current Q values. This shift in representation allows us to relate the single taxi solution to this problem. Details of the allocation function are deferred to Appendix C.

With the allocation function in hand, we set up the approximate solution over pairs of allocated single taxi states as follows. $\hat{Q}$ is taken as the sum of the expected reward for each delivery. Referring to the Q function for the single taxi problem as Q_s, $\hat{Q}$ for the pair of single taxi state-actions (s_1, a_1) and (s_2, a_2) is defined as:

[12] When a passenger is successfully delivered, we set their location to Delivered and their destination to a dummy value, since a delivered passenger does not have a destination.

$$\hat{Q}((s_1, s_2), (a_1, a_2)) = Q_s(s_1, a_1) + Q_s(s_2, a_2) \tag{19}$$

That is, the cost of two parallel passenger deliveries is estimated as the sum of the individual passenger delivery costs.

For the model, $\hat{P}$ is taken as the product of two transition functions from the single taxi problem, and $\hat{R}$ is taken as the sum of two single taxi reward functions. Referring to the transition and reward functions for the single taxi problem as P_s and R_s respectively, $\hat{P}$ and $\hat{R}$ are defined as:

$$\hat{P}((s_1, s_2), (a_1, a_2), (s_1', s_2')) = P_s(s_1, a_1, s_1') \times P_s(s_2, a_2, s_2') \tag{20}$$

$$\hat{R}((s_1, s_2), (a_1, a_2)) = R_s(s_1, a_1) + R_s(s_2, a_2) \tag{21}$$

$\hat{R}$ is exact since the reward function decomposes according to passengers, so we will not consider patching $\hat{R}$ for this domain. $\hat{P}$ is not exact since it ignores collisions. Therefore, the patched model seed predicate will trigger when collisions are detected. Note that because $\hat{P}$ assumes the absence of collisions, its inverse will predict predecessors with illegal state configurations where both taxis occupy the same location[13]. Because patching uses the transition function inverse to propagate changes in Q value, some of these illegal states will be added to Q_{patch}. This does not make the policy worse since those states will never be subject to evaluation, but simply wastes storage. We do not do anything particular to prevent this.

For the patch functions, in this domain we will exploit domain knowledge in the form of symmetry. Firstly, both the Q function patch and transition function patch make use of symmetry over pairs of single taxi state-actions:

$$Q_{\text{patch}}((s_1, s_2), (a_1, a_2)) = Q_{\text{patch}}((s_2, s_1), (a_2, a_1)) \tag{22}$$

$$P_{\text{patch}}((s_1, s_2), (a_1, a_2), (s_1', s_2')) = P_{\text{patch}}((s_2, s_1), (a_2, a_1), (s_2', s_1')) \tag{23}$$

This means that Q_{patch} and P_{patch} effectively hold values for twice the number of actual entries. For the transition function patch, we also know that the modifications to the transition model affect the taxi location components of the states only. Therefore, it is defined over pairs of taxi locations and actions rather than over full state-actions. Samples for the χ^2 test are also collected over taxi locations and actions instead of full state-actions. The χ^2 test required 100 successor state samples before testing $\hat{P}$ for a particular state-action, with a null hypothesis probability cut-off of 0.00001. The results shown in this section were produced using a transition sampling set capacity of 1000 state-actions; results using other capacities show similar trends as those shown here, and are deferred to Appendix E.2.

The patch seed predicate is the patched model seed predicate. Since $\hat{R}$ is exact in this domain, the model seed predicate will only trigger for entries in the transition function patch. In this case, since the transition function patch is defined over taxi locations and actions only, the seed predicate is true for all passenger locations and destinations for any taxi location and action pair in the transition function patch.

Both instances of prioritised sweeping also made use of the same assumptions of symmetry in Q_{patch}. Because initialised prioritised sweeping used the same model patching routines as the patching algorithms, it also benefited from symmetry and abstract definition of P_{patch}. Prioritised sweeping from scratch constructed a flat model from scratch with symmetry known.

[13]These illegal configurations are *not* included in the reported problem size.

The backup queue had a maximum capacity of 10,000 state-actions. When the queue exceeded capacity, the lowest priority element was dropped.

In order to calculate the greedy action and value, we try to avoid naively enumerating over the joint action space for the two taxis. As Q_{patch} is learnt over time, it will be common for joint states to have a handful of patched actions, but otherwise be mostly unpatched. This can be seen as analogous to decision making in a multi-agent setting, where some joint actions interact and have value dependent on the joint action choice, but the majority of joint actions do not interact. It would be extremely inefficient to iterate over the joint action space in such a situation since the majority of the joint action choices are independent. Thus, we implement an efficient method for determining the greedy action and value that is linear in the number of patched actions at the given state. Details of this implementation optimisation are deferred to Appendix D.

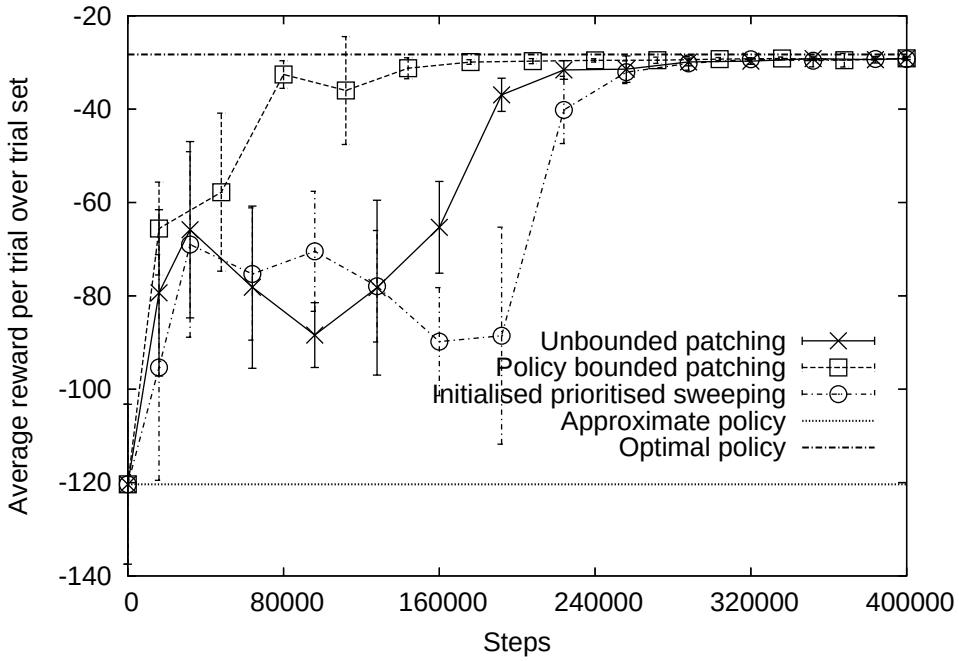
Evaluation of the policy in this domain is different to the previous domains, as it is too large to reasonably calculate the exact value function repeatedly. Learning is suspended at regular intervals to evaluate the policy on a random test set of 1,000 initial states, fixed for each experiment run but different for different runs. When evaluating the policy, a maximum trial length of 1,000 steps was imposed, since the policy may not be guaranteed to eventually terminate (*e.g.* both taxis may get stuck indefinitely in the bottom left corner of the grid if the policy ignores collisions).

6.4.3 Results

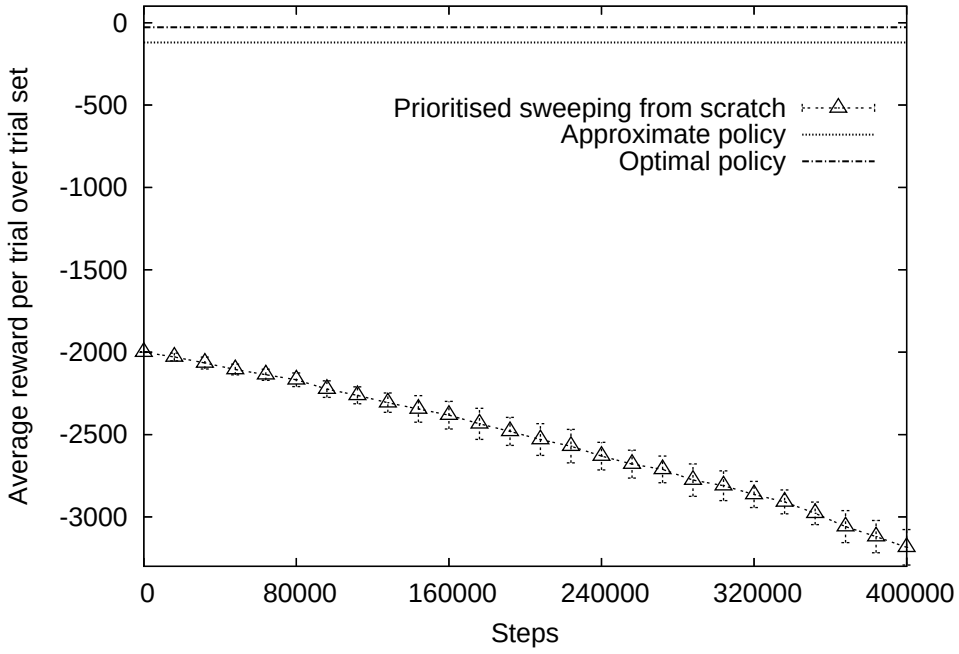
First, we compare unbounded patching, policy bounded patching, initialised prioritised sweeping, and prioritised sweeping from scratch. Figure 29(a) compares the average reward per trial for the three approaches using the initial approximation. All algorithms making use of the initial approximation quickly reduce the gap to optimality, although policy bounded patching is fastest to do so as it focuses its backups on those that immediately affect the policy. Figure 29(b) plots the average reward per trial for prioritised sweeping from scratch. While the initialised algorithms all had learning curves nestled between the approximate and optimal solutions, prioritised sweeping from scratch starts far below the approximate solution (note the compressed vertical axis for average reward). It progressively gets worse as it blindly explores the problem, of which it manages to cover only a small fraction in the experiment duration. Clearly, while the initial approximation is not perfect, it is a much more preferable starting point to nothing.

In terms of Q_{patch} sizes, shown in Figure 29(c), policy bounding does not exhibit the trend shown in other domains where Q_{patch} growth is effectively halted once expected reward as settled. One reason for this is that most of the sub-optimality in the approximate solution can be resolved by patching to navigate around collisions, but further small improvements are possible, such as coordination strategies that make positive use of collisions (*e.g.*, when there is only one passenger to deliver, the "spare" taxi can act as a buttress for the taxi delivering the passenger, so as to reduce losses from stochastic navigation actions). Policy bounding alone does not restrict this patch growth: it will grow the patch as long as the policy is changing, regardless of how small the improvement from those changes may be.

Next, we examine the performance of patching with utility bounding. Figure 30 shows the average reward per trial for patching with utility bounding with Q_{patch} capacities of 200000 and 300000 (1.15% and 1.72% of the state-action space respectively). As with the other domains, we observe the common trend of the higher Q_{patch} capacity producing a better policy with lower deviation. However, neither case appears to settle as steadily as unbounded patching or policy bounded patching. This suggests that the priority function used with utility bounding is

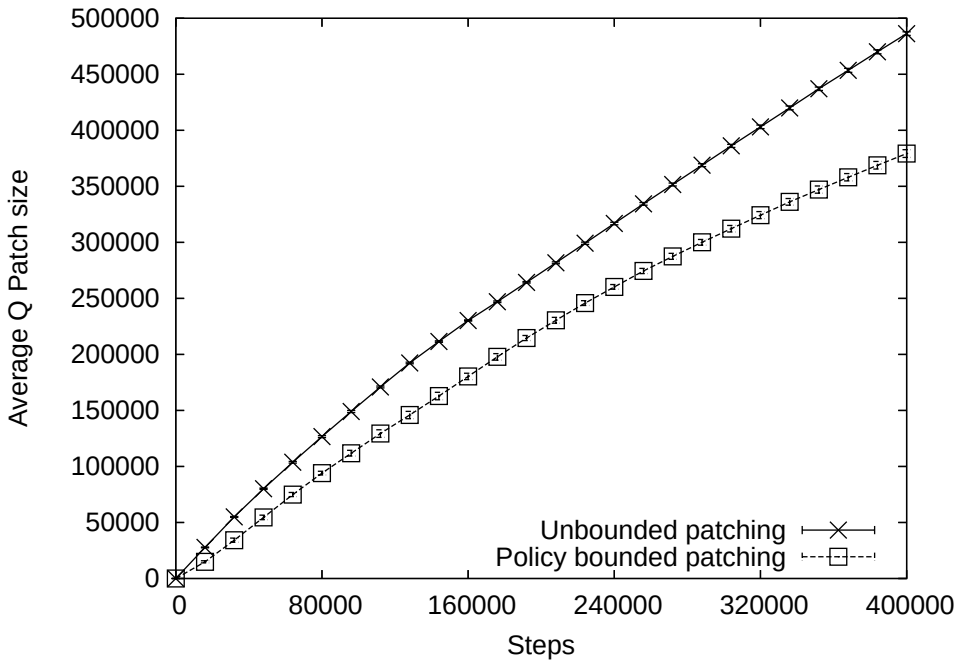


(a) Average reward per trial for unbounded patching, policy bounded patching, and initialised prioritised sweeping.



(b) Average reward per trial for the multi-taxi domain for prioritised sweeping starting from scratch.

Figure 29: Results for the multi-taxi domain.



(c) Average Q_{patch} size for unbounded patching and policy bounded patching.

Figure 29 (continued): Results for the multi-taxi domain.

assigning similar priority to many state-actions in Q_{patch}, such that the set of state-actions in Q_{patch} is somewhat unstable.

Figure 31 plots the average reward per trial for patching with both policy and utility bounding, with Q_{patch} capacities of 100000 and 200000 (0.57% and 1.15% of the state-action space respectively). Adding policy bounding appears to reduce the unsteadiness of the policy quality, and makes much better use of the limited storage. However, performance does drop noticeably for the lower Q_{patch} capacity towards the end of learning. This appears to be the result of the heuristic priority function for utility bounding.

Table 6 summarises the results for the multi-taxi domain.

6.4.4 Discussion

The multi-taxi domain demonstrates the applicability of patching with more sophisticated patch functions, and with a problem that is much larger than the previous examples.

Policy bounding is very effective in this domain. The results indicate that policy bounded patching learns more steadily and attains performance closer to global optimality than patching without policy bounding in this domain. Also, patching fared much better when combining policy and utility bounding than applying utility bounding alone.

An interesting characteristic of this domain is the density of the transition function inverse. Consider two taxis in open space on the grid. Taxi 1 could have reached its current position from any of the four neighbouring cells by executing a navigation action that would have caused it to move there, *e.g.* from one cell north, it could have executed South, East, or West and had the stochastic effect of moving south. In addition, it might have been at that position already and executed Pickup, Putdown, or Null, making a total of 15 predecessors. Predecessors to Taxi 2's state may be enumerated similarly, of which there are also 15. The predecessors of the

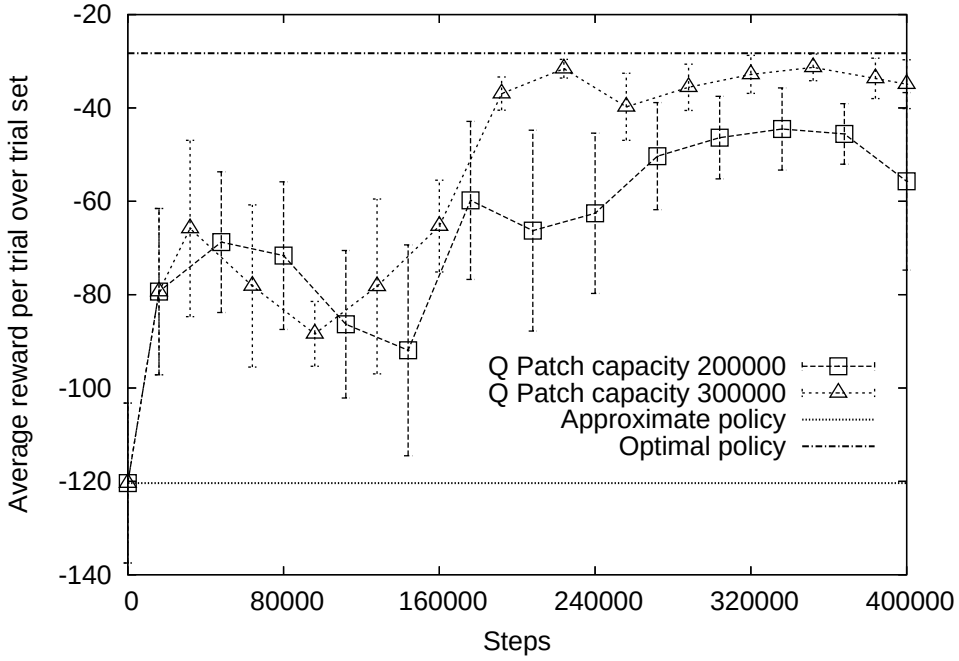


Figure 30: Average reward per trial for the multi-taxi domain for patching with utility bounding, with Q_{patch} capacities of 200000 and 300000.

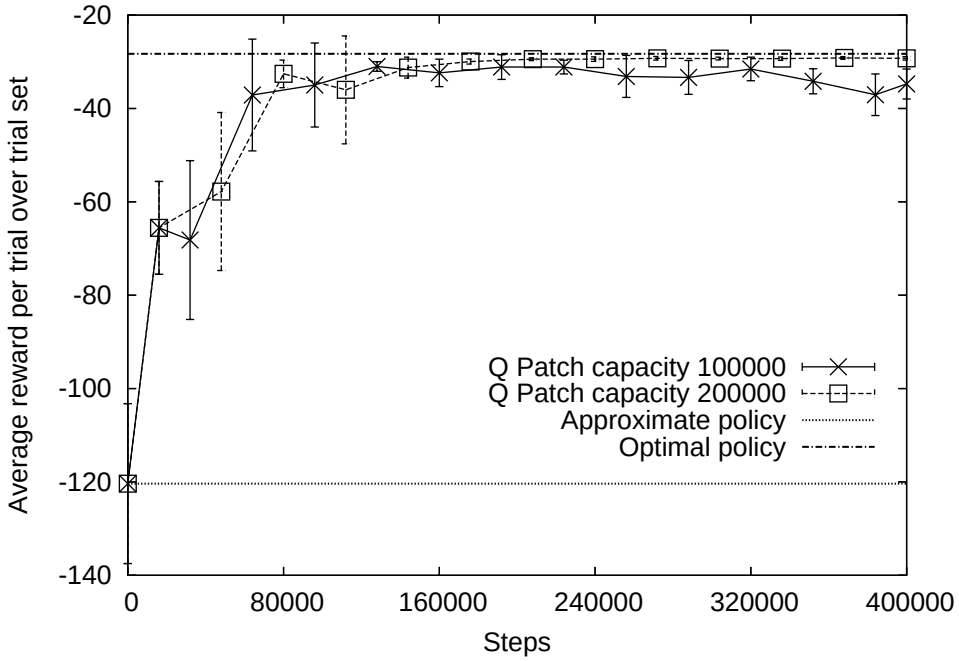


Figure 31: Average reward per trial for the multi-taxi domain for patching with policy and utility bounding, with Q_{patch} capacities of 100000 and 200000.

Table 6: Summary of results for the multi-taxi domain. Figures were taken at the end of 400,000 steps. The reward statistics listed in this table are the reward per trial for the greedy policy over a test set of 1,000 random initial states, *not* the reward per trial while learning with ϵ-greedy action selection. For patched solutions, the total size is listed as the size of $\hat{Q}$ plus the size of Q_{patch}.

Solution	Average reward per trial	# Q values
Initial approximation ($\hat{Q}$)	-120.37 ± 17.12	4200
Optimal flat	-28.29 ± 0.28	17434200
Prioritised sweeping from scratch	-3184.23 ± 107.44	17434200
Initialised prioritised sweeping	-29.26 ± 0.32	17434200
Unbounded patching	-29.11 ± 0.27	4200 + 486358.40 ± 1580.45
With policy bounding	-29.10 ± 0.33	4200 + 379263.10 ± 3444.89
With utility bounding		
– Q_{patch} capacity 200000	-55.73 ± 19.01	4200 + 200000
– Q_{patch} capacity 300000	-34.91 ± 5.23	4200 + 300000
With policy and utility bounding		
– Q_{patch} capacity 100000	-34.75 ± 3.21	4200 + 100000
– Q_{patch} capacity 200000	-29.26 ± 0.37	4200 + 200000

combined state, then, are any pair chosen from those two lists, for a total of 225 predecessors. Adaptively patching the inverse transition function by only storing the changes from $\hat{P}^{-1}$ helps to reduce the storage requirements of the predecessor model.

7 Conclusions and Future Work

In this report, 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.

The unbounded patching algorithm was outlined as a base patching algorithm that quickly improves solution quality by learning around the patch seeds. We proposed policy bounding and utility bounding as two modifications for bounding patch growth. Policy bounding uses immediate change in the policy as a heuristic to reduce patch growth. Utility bounding aims to make the best use of strictly bounded storage by prioritising patch entries, and dropping them when necessary. Patch Q-learning was presented as a model-free patching algorithm that makes use of utility bounding.

We discussed methods for adaptively updating the approximate model to correct inaccuracies. Inaccuracies were detected using statistical tests, with selective sampling to bound storage requirements for collecting samples.

Empirical results demonstrated the effectiveness of patching in several domains. The experiments demonstrated the applicability of patching under several conditions, with different types of underlying approximations and patch representations.

Immediate future work will aim to apply patching to a larger, more complex domain. Another possible avenue for future research is investigation of more sophisticated prioritised patching techniques. We introduced the utility bounding as a method for putting a firm limit on storage, but many areas remain open to further study, such as alternative priority functions, adaptively adjusting patch size using a heuristic rather than using a fixed capacity, and dynamic allocation of storage according to patch seeds.

Acknowledgements

We thank Claude Sammut, Bernhard Hengst, Robert Fitch, and Malcolm Ryan for helpful feedback and discussion that assisted in developing the ideas in this report.

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

D. Andre, N. Friedman, and R. Parr. Generalized prioritized sweeping. In M. I. Jordan, M. S. Kearns, and S. A. Solla, editors, *Advances in Neural Information Processing Systems 10*. MIT Press, 1998.

A. G. Barto, S. J. Bradtke, and S. P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

M. Bowling and M. Veloso. Reusing learned policies between similar problems. In *Proceedings of the AI*IA-98 Workshop on New Trends in Robotics*, Padua, Italy, October 1998.

T. Dietterich. The MAXQ method for hierarchical reinforcement learning. In J. W. Shavlik, editor, *Proceedings of the 15th International Conference on Machine Learning*, pages 118–126. Morgan Kaufmann, 1998.

T. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

C. Guestrin, D. Koller, and R. Parr. Multiagent planning with factored MDPs. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 1523–1530. MIT Press, 2001.

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

J. R. Kok, P. Jan't Hoen, B. Bakker, and N. Vlassis. Utile coordination: Learning interdependencies among cooperative agents. In *IEEE Symposium on Computational Intelligence and Games*, 2005.

J. R. Kok and N. Vlassis. Sparse cooperative Q-learning. In R. Greiner and D. Schuurmans, editors, *Proceedings of the 21st International Conference on Machine Learning*, pages 481–488. ACM, 2004.

R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:189–211, 1990.

A. K. McCallum. *Reinforcement learning with selective perception and hidden state*. PhD thesis, Department of Computer Science, University of Rochester, 1995.

A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

R. Munos and A. Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49:291–323, 2002.

J. Peng and R. J. Williams. Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4):437–454, 1993.

G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.

R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In B. W. Porter and R. J. Mooney, editors, *Proceedings of the 7th International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

R. S. Sutton. Integrated modelling and control based on reinforcement learning and dynamic programming. In R. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 471–478. Morgan Kaufmann, 1991a.

R. S. Sutton. Planning by incremental dynamic programming. In L. Birnbaum and G. Collins, editors, *Proceedings of the 8th International Workshop on Machine Learning*, pages 353–357. Morgan Kaufmann, 1991b.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, 1998.

M. E. Taylor and P. Stone. Behavior transfer for value-function-based reinforcement learning. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Woolridge, editors, *The 4th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 53–59. ACM Press, 2005.

M. E. Taylor, P. Stone, and Y. Liu. Value functions for RL-based behavior transfer: a comparative study. In M. M. Veloso and S. Kambhampati, editors, *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference*, pages 880–885. AAAI Press, 2005.

C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Oxford, 1989.

C. J. C. H. Watkins and P. Dayan. Technical note: Q-Learning. *Machine Learning*, 8:279–292, 1992.

Appendix

A Using a MAXQ Value Function in Patching

We briefly review the MAXQ value function decomposition, and describe how we calculate Q values using this representation. This review of the MAXQ representation is not intended to be exhaustive; a complete description of the MAXQ method for hierarchical reinforcement learning is provided by Dietterich [2000].

A.1 Review of MAXQ Value Function Decomposition

In MAXQ, the task hierarchy is supplied by the user in the form of a directed acyclic graph known as a *MAXQ graph*. Nodes in a MAXQ graph are either Max nodes, corresponding to sub-tasks, or Q nodes, corresponding to actions available to sub-tasks. A Max node may have several child Q nodes (reflecting that a sub-task may have several actions at its disposal), but a Q node will always have exactly one child Max node (reflecting that an action corresponds to exactly one lower level sub-task).

Internal Max nodes in a MAXQ graph are *composite* Max nodes, while leaf nodes are *primitive* Max nodes. Composite Max nodes correspond to temporally extended sub-tasks, while each primitive Max node is associated with a single primitive action, and always terminates after one step. The edges in a MAXQ graph define the sub-tasks available to each task, with each composite Max node only able to invoke the sub-tasks corresponding to its immediate children nodes.

MAXQ decomposes the value function into two parts. The *projected value function* $V(i, s)$ for Max node i and state s is defined as the expected reward from executing sub-task i until it terminates, starting from state s. For primitive Max nodes, the projected value function is the expected one step reward for the associated primitive action, *i.e.* for a primitive Max node i associated with primitive action a, $V(i, s) = R(s, a)$. Each primitive Max node maintains this expected one-step reward value for its relevant action.

The *completion function* $C(i, s, a)$ for Max node i, state s, and child Max node a of i, is the expected reward of completing the sub-task for i after invoking the child sub-task a from state s. Q nodes are responsible for maintaining the completion function for their associated sub-task, *i.e.* a Q node that represents the action a for parent sub-task i will store the values for $C(i, s, a)$.

A key advantage of the MAXQ value function decomposition is *state abstraction*. The values stored in each primitive Max node and Q node are functions of the state, but may not depend on all state variables. Dietterich outlines five conditions for state abstraction that may potentially result in an exponential reduction in storage requirements.

The value function is decomposed recursively as follows. The projected value function for a primitive Max node is its expected one-step reward. The projected value function for a composite Max node defined as the maximum value of the sum of the sub-task value plus the completion function, over valid sub-tasks:

$$V(i, s) = \begin{cases} max_a Q(i, s, a) & \text{if } i \text{ is composite, where } a \text{ is a valid sub-task of } i \text{ at } s \\ R(s, a) & \text{if } i \text{ is primitive, where } a \text{ is the primitive action for } i \end{cases} \tag{24}$$

$$Q(i, s, a) = V(a, s) + C(i, s, a) \tag{25}$$

Whether or not a sub-task is valid at a particular state is defined by its *termination predicate*. This restricts the states in which a sub-task may be executed, and therefore reduces its storage requirements further, since it only needs to store values over valid states.

The overall value of a state is defined as the value of $V(root, s)$, where *root* refers to the root task of the MAXQ graph.

A.2 Calculating Q Values from a MAXQ Value Function

In order to use a MAXQ decomposed value function as the base for patching, we need to calculate a Q function over primitive actions. We do this by traversing the task hierarchy, and for each primitive action, we find the highest value path through the hierarchy that ends in that primitive action, and is valid at all sub-tasks in the path. The procedure is shown in Algorithm 9.

Algorithm 9: Function MAXQ_values(s, i).

Input: State s and Max node i.
Output: A list of Q values for all primitive actions from s, where each value reflects the highest value path from i to the primitive Max node for that action.

```
1  begin
2      Q_i(s,a) ← −∞ for all primitive actions a;
3      if i is a primitive Max node then
4          Q_i(s,a') ← V(i,s) for the primitive action a' associated with i;
5      else
6          foreach sub-task j of i do
7              if j is valid at s then
8                  Q_j(s,.) ← MAXQ_values(s,j);
9                  foreach primitive action a' do
10                     Q_i(s,a') ← max(Q_i(s,a'), Q_j(s,a') + C(i,s,j));
11                 end
12             end
13         end
14     end
15     return Q_i(s,.);
16 end
```

Line 7 refers to the termination predicate. Line 8 is a recursive call, and gathers the Q values for a valid child sub-task. The following loop (lines 9–11) then takes the maximum between the current value and the child sub-task value plus completion value for all primitive actions. $\hat{Q}(s, .)$ is determined by invoking MAXQ_values$(s, root)$, where *root* is the root of the task hierarchy.

Note that MAXQ_values returns $-\infty$ for unreachable primitive actions – that is, MAXQ_values suggests that $\hat{Q}(s, a) = -\infty$ when no valid path through the task hierarchy exists to the primitive Max node associated with a for state s. Where this is the case, it reflects that a is implicitly forbidden at state s in the user-defined hierarchy. The appropriate way to handle these cases depends on whether the constraints of the hierarchy are appropriate for the task. A serious implication for utility bounding is that the priority of such actions in Q_{patch} would be ∞, because the heuristic priority function for a proposed update $Q_{\text{patch}}(s, a) \leftarrow q$ is the absolute difference between $\hat{Q}(s, a)$ and q. When $\hat{Q}(s, a)$ is $-\infty$, this priority value is ∞ regardless of the value of q.

For experiments in the modified taxi domain (Section 6.2), the MAXQ hierarchy excludes:

- Putdown for states where the passenger location is not In_Taxi.
- Pickup for states where the passenger location is In_Taxi.
- North, South, East, and West for states where the taxi is at the target taxi stand to execute either Pickup or Putdown.

These actions can be safely excluded in the modified taxi problem, since the modifications to P and R affect navigation between the taxi stands, but not the optimal behaviour at the taxi stands – that is, the optimal policy still follows the pattern of navigating to the passenger location, picking them up, navigating to the passenger destination, then putting them down. Thus, in the modified taxi experiments, we modify the priority function for utility bounding to return 0.0 when $\hat{Q}(s,a) = -\infty$.

For experiments in the taxi with fuel domain (Section 6.3), the action set is extended to include the Refill action. Refill is not included in the MAXQ hierarchy used to calculate $\hat{Q}$ in these experiments, so $\hat{Q}(s,\text{Refill}) = -\infty$ for all states s. In this case, because the MAXQ hierarchy tells us nothing about the Refill action, it is useful to approximate it otherwise: for the taxi with fuel experiments, $\hat{Q}(s,a)$ is estimated using MAXQ_values for $a \neq$ Refill, but using the one-step model-based value for $a =$ Refill:

$$\hat{Q}(s,\text{Refill}) = \hat{R}(s,\text{Refill}) + \gamma \sum_{s' \in S} \hat{P}(s,\text{Refill},s') V^{MAXQ}(s') \tag{26}$$

where $V^{MAXQ}(s')$ is the MAXQ hierarchical value for state s'.

B MAXQ Hierarchy and Parameters used in Experiments

The MAXQ task hierarchy for the taxi problem used to initialise $\hat{Q}$ for the modified taxi and taxi with fuel domains is as specified by Dietterich [2000]. For the undiscounted terminating setting, we use the state abstractions specified by Dietterich. The total storage requirements for the MAXQ hierarchy, and therefore also for $\hat{Q}$, is 632 values.

For the discounted terminating setting:

- QGet's completion value now depends on the taxi location, increasing its storage requirements from 16 values to $25 \times 16 = 400$ values;
- QNavigateForGet's completion value now depends on the taxi location, increasing its storage requirements from 4 values to 100 values;
- QNavigateForPut's completion value now depends on the taxi location, increasing its storage requirements from 4 values to 100 values.

The total storage requirements for the MAXQ hierarchy, and therefore for $\hat{Q}$, in this setting is 1208 values.

For the discounted continuing setting, in addition to the changes made in the discounted terminating setting, QPut now needs completion values, since the task does not terminate on delivery. Specifically, it depends on the taxi location and the passenger destination. This raises the total storage requirements in this setting to 1308 values.

The same learning parameters were used for MAXQ to initialise $\hat{Q}$ for each setting, which were as follows. We used the MAXQ-0 algorithm specified by Dietterich [2000], run in purely hierarchical execution mode for 100,000 steps with a constant step size of $\alpha = 0.1$ at all nodes in the hierarchy. For the exploration policy, we used softmax exploration with an initial temperature of 50.0 and a cooling rate of 0.9995 at all Max nodes.

C Allocations in the Multi-taxi Domain

Our experiments for the multi-taxi problem made use of a hand-crafted allocation function that allowed definition of the solution over pairs of states from the single taxi problem. In this section, we describe the procedure for determining taxi to passenger allocations.

We interpret a state in the single taxi problem, $s = (t, l, d)$, as representative of an allocation of the taxi at location t to the delivery of the passenger currently at location l, wishing to go to destination d. In this interpretation, the Q function for the single taxi problem, Q_s, may be seen as the expected reward to completion of the task of this particular passenger delivery.

When two passenger deliveries are executed in parallel, we can estimate the expected reward to completion of both deliveries by taking the sum of the expected individual rewards. Specifically, for single taxi states s_1 and s_2, and single taxi actions a_1 and a_2:

$$\hat{Q}((s_1, s_2), (a_1, a_2)) = Q_s(s_1, a_1) + Q_s(s_2, a_2)$$

This can be calculated on demand using Q_s, requiring exponentially less storage than a flat Q table representation. Because the reward function decomposes according to passengers (or tasks), $\hat{Q}$ exactly reflects the expected reward of completing the two tasks in parallel assuming concurrent independence.

The remaining detail of the approximate solution is deciding how to split a multi-taxi state into two single taxi states, or in other words, how to decide which taxi delivers which passenger. For each taxi, we consider up to three possible allocations at each time step: the taxi may be assigned to deliver passenger 1, or to deliver passenger 2, or the taxi may not be assigned either passenger. These allocations produce single taxi states that may then be used to calculate the expected reward to completion according to $\hat{Q}$.

We will write $T_i \rightarrow P_j$ as meaning that taxi i has been assigned to deliver passenger j, and $T_i \rightarrow \mathsf{null}$ meaning that taxi i has been assigned neither passenger. Then, for multi-taxi state $s = (t_1, t_2, l_1, l_2, d_1, d_2)$, where the t_i are taxi locations, the l_i are passenger locations, and the d_i are passenger destinations, the possible allocations for taxi i and corresponding single taxi states are: $T_i \rightarrow P_1$ and $s_i = (t_i, l_1, d_1)$; $T_i \rightarrow P_2$ and $s_i = (t_i, l_2, d_2)$; and $T_i \rightarrow \mathsf{null}$ and $s_i = (t_i, \mathsf{Delivered})$[14]. A restriction on these allocations is that if a taxi is already carrying a passenger, then that passenger must be allocated to that taxi.

Sequences of these allocations effectively define plans for deciding which taxi delivers which passenger and in which order. The plans considered for each state are shown in Table 7, along with the corresponding local states and estimated rewards to completion.

The first two plans in Table 7 attempt to deliver both passengers in parallel, either with taxi 1 delivering passenger 1 and taxi 2 delivering passenger 2 or with the allocations reversed. The expected reward to completion is the corresponding value in $\hat{Q}$, since completing the two deliveries completes the entire task.

The other four plans in Table 7 attempt to deliver the passengers sequentially, dropping off one passenger before starting on the delivery of the other. It is assumed that this sequential manner of deliveries is required only when it is more efficient for one taxi to deliver both passengers.

Calculating the expected reward to completion for sequential deliveries is more complicated than for parallel deliveries. For sequential deliveries, we cannot take the value from $\hat{Q}$ with the first set of allocated local states, since the first set of allocated local states reflects delivery of the first passenger only. Since delivery of the first passenger does not complete the entire task, this does not reflect the expected reward to completion. Furthermore, the local states passed to

[14] A passenger who is Delivered does not have a destination.

Table 7: Possible plans evaluated for a given multi-taxi state $s = (t_1, t_2, l_1, l_2, d_1, d_2)$.

Plan	**Local states**	**Estimated reward to completion**
$T_1 \to P_1, T_2 \to P_2$	$s_1 = (t_1, l_1, d_1), s_2 = (t_2, l_2, d_2)$	$\max_{a_1,a_2} \hat{Q}((s_1, s_2), (a_1, a_2))$
$T_1 \to P_2, T_2 \to P_1$	$s_1 = (t_1, l_2, d_2), s_2 = (t_2, l_1, d_1)$	$\max_{a_1,a_2} \hat{Q}((s_1, s_2), (a_1, a_2))$
$T_1 \to P_1, T_2 \to \mathsf{null}$; $T_1 \to P_2, T_2 \to \mathsf{null}$	$s_{1a} = (t_1, l_1, d_1), s_{2a} = (t_2, \mathsf{Delivered})$; $s_{1b} = (d_1, l_2, d_2), s_{2b} = (t_2, \mathsf{Delivered})$	$2 \times \max_{a_1,a_2} \hat{Q}((s_{1a}, s_{2a}), (a_1, a_2))$ $+ \max_{a_1,a_2} \hat{Q}((s_{1b}, s_{2b}), (a_1, a_2))$
$T_1 \to P_2, T_2 \to \mathsf{null}$; $T_1 \to P_1, T_2 \to \mathsf{null}$	$s_{1a} = (t_1, l_2, d_2), s_{2a} = (t_2, \mathsf{Delivered})$; $s_{1b} = (d_2, l_1, d_1), s_{2b} = (t_2, \mathsf{Delivered})$	$2 \times \max_{a_1,a_2} \hat{Q}((s_{1a}, s_{2a}), (a_1, a_2))$ $+ \max_{a_1,a_2} \hat{Q}((s_{1b}, s_{2b}), (a_1, a_2))$
$T_1 \to \mathsf{null}, T_2 \to P_1$; $T_1 \to \mathsf{null}, T_2 \to P_2$	$s_{1a} = (t_1, \mathsf{Delivered}), s_{2a} = (t_2, l_1, d_1)$; $s_{1b} = (t_1, \mathsf{Delivered}), s_{2b} = (d_1, l_2, d_2)$	$2 \times \max_{a_1,a_2} \hat{Q}((s_{1a}, s_{2a}), (a_1, a_2))$ $+ \max_{a_1,a_2} \hat{Q}((s_{1b}, s_{2b}), (a_1, a_2))$
$T_1 \to \mathsf{null}, T_2 \to P_2$; $T_1 \to \mathsf{null}, T_2 \to P_1$	$s_{1a} = (t_1, \mathsf{Delivered}), s_{2a} = (t_2, l_2, d_2)$; $s_{1b} = (t_1, \mathsf{Delivered}), s_{2b} = (d_2, l_1, d_1)$	$2 \times \max_{a_1,a_2} \hat{Q}((s_{1a}, s_{2a}), (a_1, a_2))$ $+ \max_{a_1,a_2} \hat{Q}((s_{1b}, s_{2b}), (a_1, a_2))$

Table 8: Evaluation of the approximate solution on the multi-taxi problem *without* collisions.

Solution	Reward per trial
$\hat{Q}$	-27.7138 ± 0.2416
Optimal	-27.6099 ± 0.3173

$\hat{Q}$ only include one passenger, so this value ignores the waiting cost incurred by the unattended passenger. In order to account for the waiting cost we note that a reward of -1 is incurred per step both by the passenger being delivered and the unattended passenger (barring illegal Pickup and Putdown actions). Thus, since the $\hat{Q}$ value for the first set of allocated states reflects the expected reward from one passenger until they are dropped off, we can double this value to get the expected reward of both passengers. The expected reward of the second delivery can then be added, taking the starting taxi location for the second delivery to be the destination taxi stand of the first passenger. This system of calculating Q values relies on the cost function decomposing by passenger: -1 per undelivered passenger per time step.

Finally, choosing between plans is deterministic and greedy on expected reward to completion. When $\hat{Q}$ is augmented with Q_{patch} by patch learning, the allocations in Table 7 use the augmented Q function instead of just $\hat{Q}$.

It is worth noting that this approximation performs very close to optimal on the multi-taxi problem without collisions. Table 8 compares the averages and standard deviation of average reward per trial for $\hat{Q}$ and the optimal policy for the multi-taxi problem without collisions, sampled over the same initial states used to evaluate the policy in the multi-taxi experiments in Section 6.4.

D Optimisations for the Multi-taxi Domain

In this section, we outline the optimisations used in the multi-taxi domain for efficient calculation of greedy actions and values. These optimisations are specific to the type of approximation used for $\hat{Q}$ in this domain.

When no actions have been patched at a state, the greedy value and action can be computed by referring to the Q tables from the single taxi problem as described in Section 6.4.2. When all actions have been patched, we need to iterate through all actions. However, in the intermediate case where only a handful of actions have been patched, we would like to compute the greedy action without enumerating the joint action space. This situation will be very common as Q_{patch} is learnt over time, and especially when policy bounding is applied. It would be extremely wasteful to iterate through the entire joint action space, since, intuitively, the majority of joint actions are independent.

The first optimisation is to re-order the Q table for the single taxi problem for each state by Q value. This reduces the look-up cost for $\hat{Q}$ to constant time instead of linear in the size of the action space for one taxi. This can be pre-computed and is relatively inexpensive, since the single taxi Q table is quite small.

In order to efficiently compute the greedy value and action for a state, we divide the problem into finding the greedy value and action amongst the unpatched actions, then finding the greedy value and action amongst the patched actions.

In order to efficiently iterate through the patched actions, we maintain a list of actions that have been patched at each state. This is implemented as a hashtable with the Q function patch, such that only states that have had actions patched will have a patched action list. The greedy value and action amongst the patched actions can then be found by iterating through this list, requiring time linear in the number of actions patched at the given state.

In order to efficiently iterate through the unpatched actions, we exploit the fact that the single taxi Q tables have already been sorted. We do a greedy search through the sorted table generated by combining single taxi Q values and stop once an unpatched action is found. This search also requires time linear in the number of patched actions at the given state, since, in the worst case, we will look-up every patched action before finding the greedy unpatched action.

Once we have the greedy value and action from the patched and unpatched sets, the greedy value and action is the maximum of the two. Since both sub-procedures require time linear in the number of patched actions at the given state, this entire procedure is also linear time. Computation of the potential effect on the policy for the policy bounding condition also requires time linear in the number of patched actions at the given state, by using a similar algorithm.

E Detailed Result Listing

The results presented in Section 6 used a fixed capacity for the transition and reward sampling sets, P_{sample} and R_{sample}. Results for other sampling set capacities exhibit similar trends, and are presented in this section for the modified taxi and multi-taxi domains. Results for the taxi with fuel domain showed minimal variation for different model sampling set capacities, since its model irregularities are deterministic. The results that were presented in Section 6 are also re-produced here, for easier comparison of performance as P_{sample} and R_{sample} capacities are varied (except for patch Q-learning, because it is model-free).

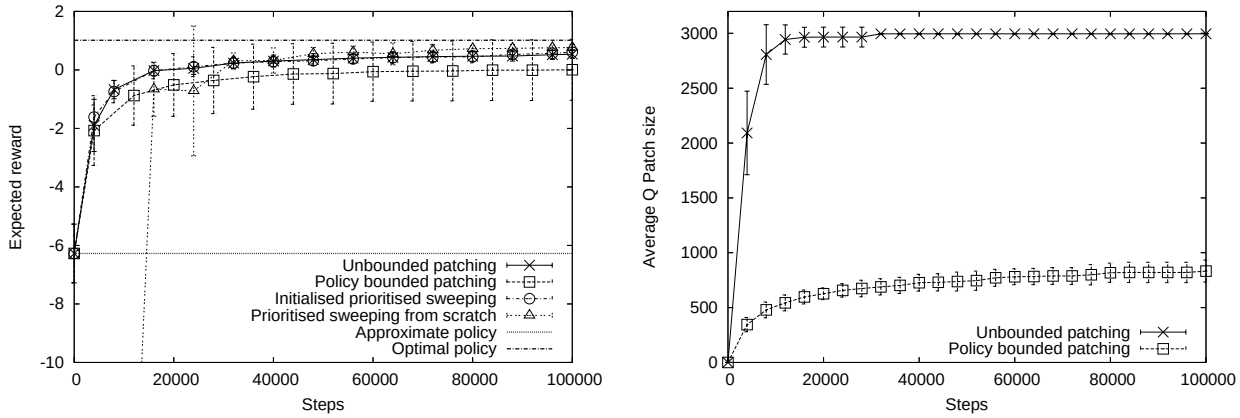
A general trend throughout the results is that higher capacities for P_{sample} and R_{sample} result in closer convergence to optimality in the long run for larger capacities of Q_{patch}, but seem to hinder performance for smaller Q_{patch} capacities. One reason for this is that, on average, it takes longer to discover patch seeds when P_{sample} and R_{sample} have smaller capacities. Consequently,

a smaller Q_{patch} capacity appears to cope better when fewer patch seeds have been found, but performance degrades as more patch seeds are discovered, since limited storage in Q_{patch} needs to be spread more thinly over more seeds.

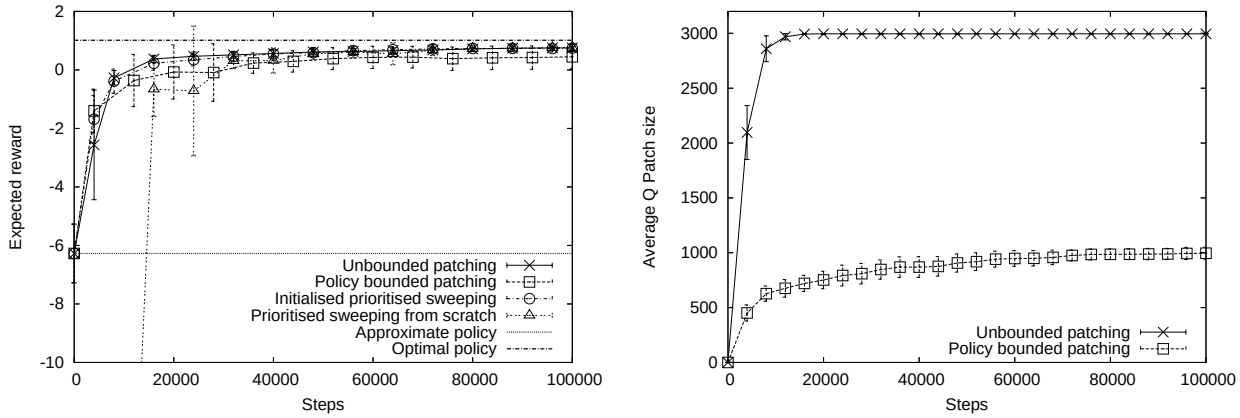
E.1 Modified Taxi

In the modified taxi domain, for each setting, we tried P_{sample} and R_{sample} capacities of 50, 100, and 500, and also with the exact P and R known.

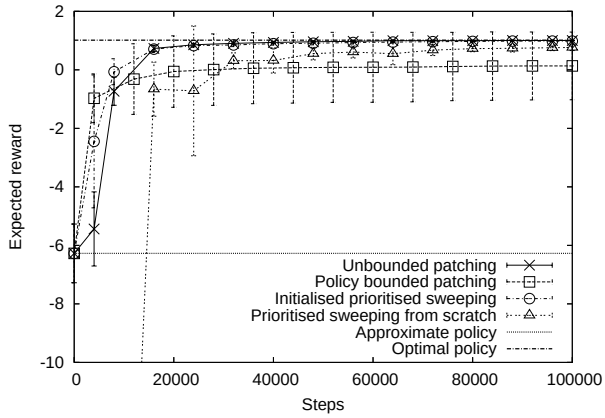
E.1.1 Modified Taxi, Undiscounted

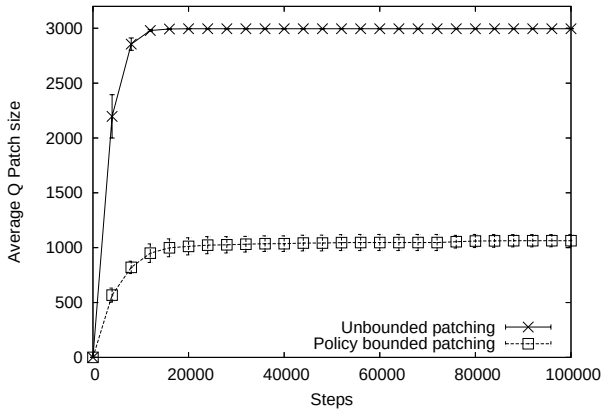


Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 50 each, and prioritised sweeping from scratch.

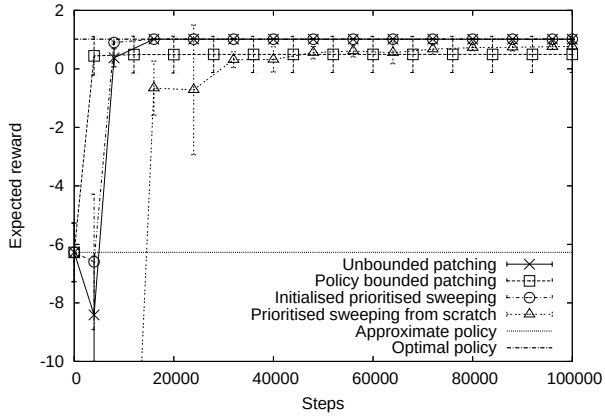


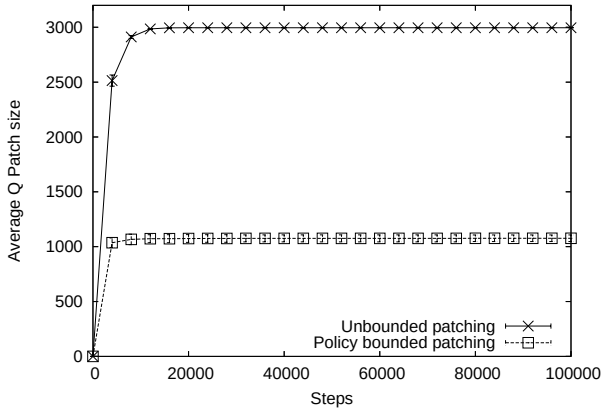
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 100 each, and prioritised sweeping from scratch.



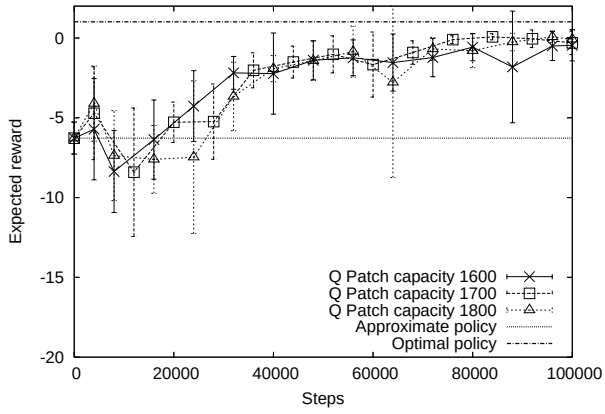


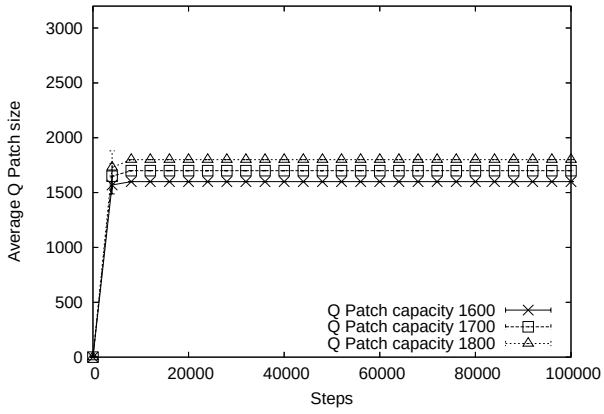
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 500 each, and prioritised sweeping from scratch.



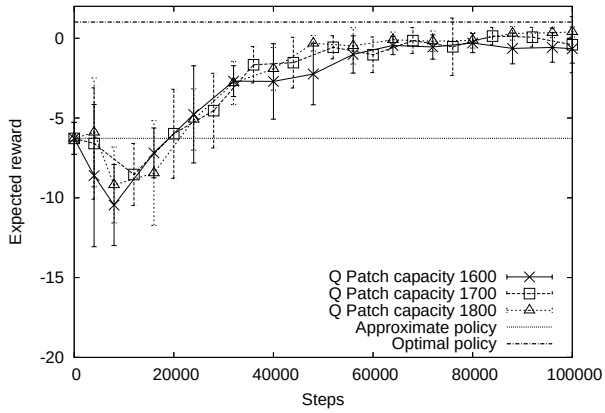


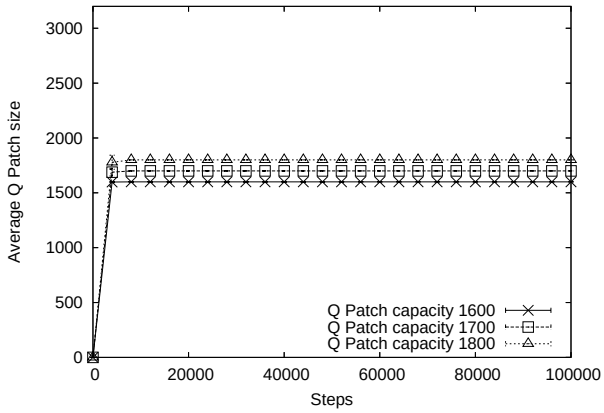
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with exact P and R known, and prioritised sweeping from scratch.



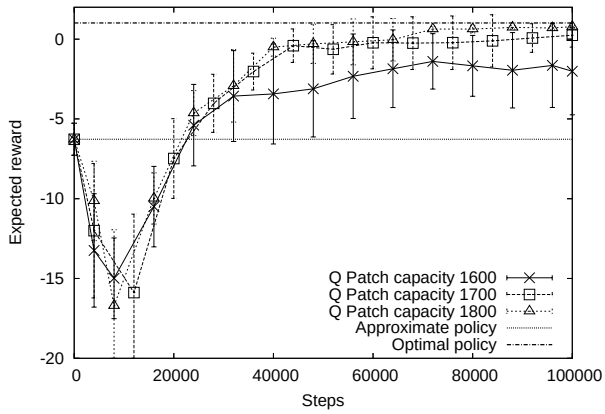


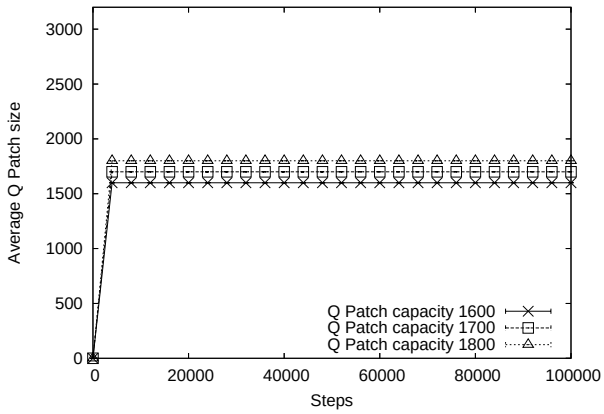
Patching with utility bounding, P_{sample} and R_{sample} capacities of 50 each.



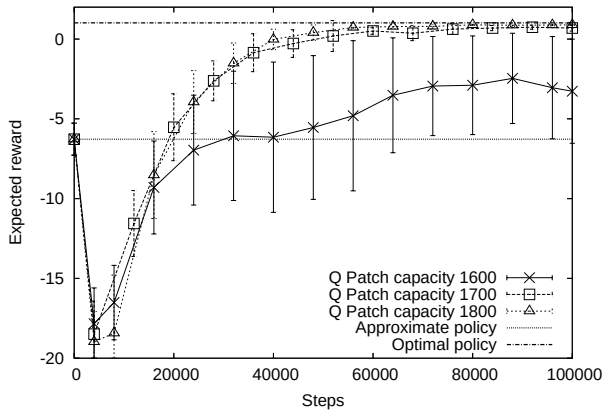


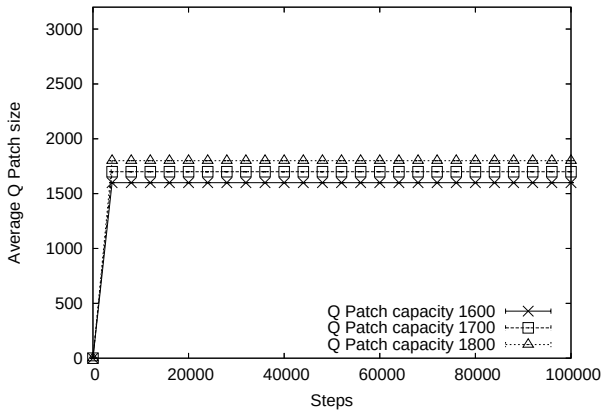
Patching with utility bounding, P_{sample} and R_{sample} capacities of 100 each.



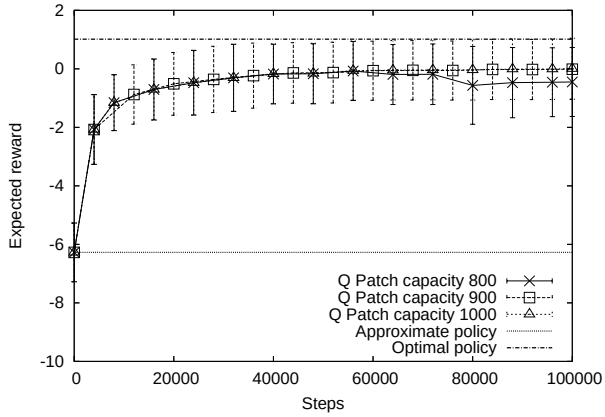


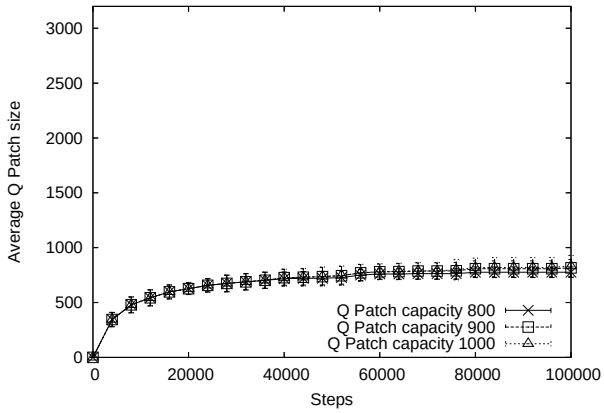
Patching with utility bounding, P_{sample} and R_{sample} capacities of 500 each.



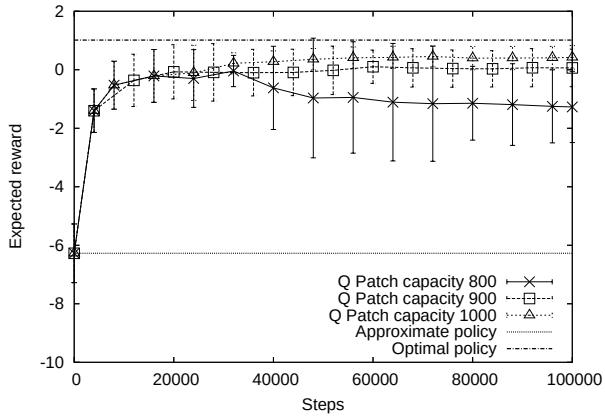


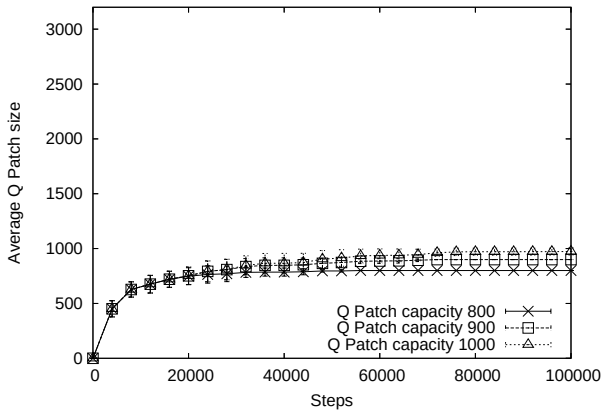
Patching with utility bounding, with exact P and R known.



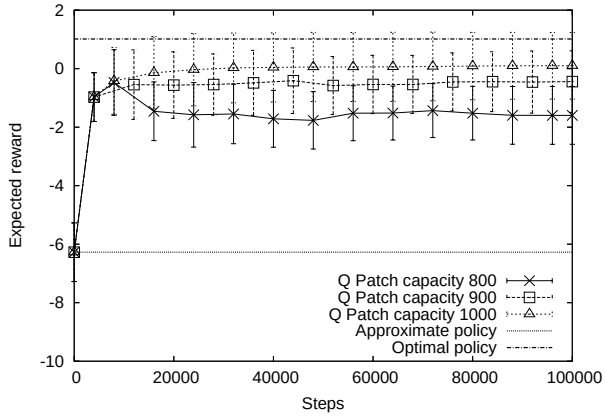


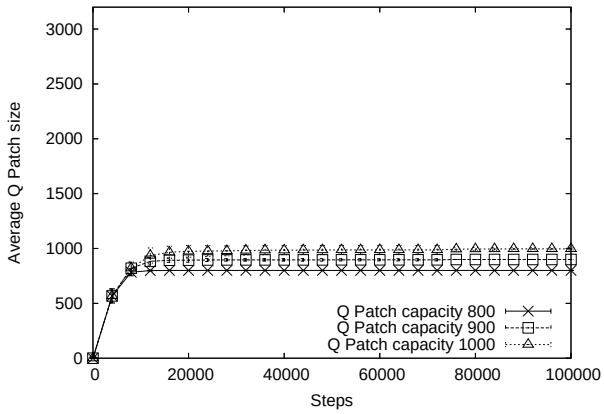
Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 50 each.



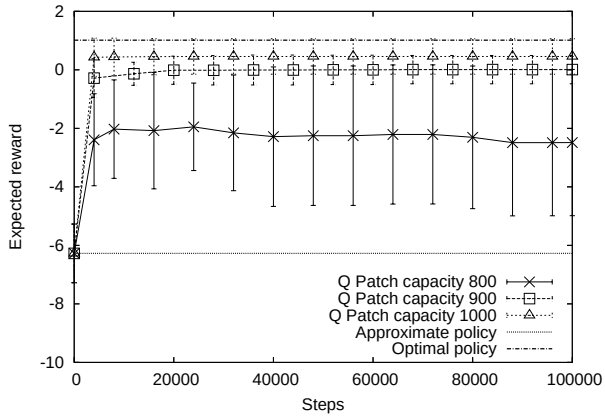


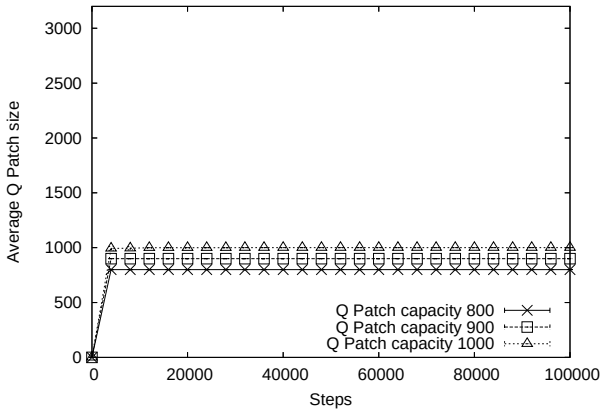
Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 100 each.





Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 500 each.





Patching with policy and utility bounding, with exact P and R known.

Table 9: Summary of results for the undiscounted setting of the modified taxi domain. Figures were taken at the end of 100,000 steps. For patched solutions, the total size is listed as the size of $\hat{Q}$ plus the size of Q_{patch}.

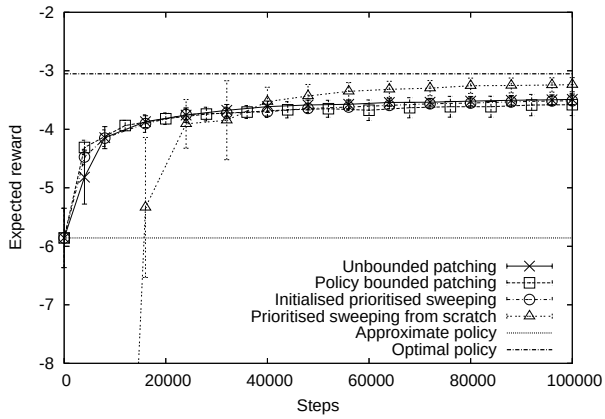
Solution	P_{sample} and R_{sample} capacity	Expected reward	# Q values
Initial approximation ($\hat{Q}$)		-6.27 ± 1.00	632
Optimal flat		1.02	3000
Prioritised sweeping from scratch		0.76 ± 0.10	3000
Initialised prioritised sweeping			
	50	0.61 ± 0.10	3000
	100	0.74 ± 0.13	3000
	500	0.98 ± 0.02	3000
	P and R known	1.02 ± 0.00	3000
Unbounded patching			
	50	0.54 ± 0.15	632 + 2996
	100	0.75 ± 0.04	632 + 2996
	500	1.00 ± 0.01	632 + 2996
	P and R known	1.02 ± 0.00	632 + 2996
With policy bounding			
	50	0.01 ± 1.04	632 + 833.00 ± 100.34
	100	0.45 ± 0.41	632 + 995.70 ± 55.36
	500	0.13 ± 1.16	632 + 1063.50 ± 55.86
	P and R known	0.49 ± 0.62	632 + 1076.90 ± 30.03
With utility bounding – Q_{patch} capacity 1600			
	50	-0.46 ± 0.98	632 + 1600
	100	-0.65 ± 0.91	632 + 1600
	500	-2.01 ± 2.72	632 + 1600
	P and R known	-3.27 ± 3.26	632 + 1600

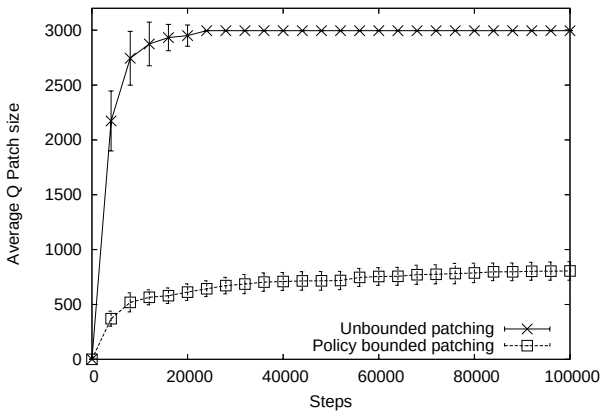
Continued on next page.

Table 9 (continued). Summary of results for the undiscounted setting of the modified taxi domain.

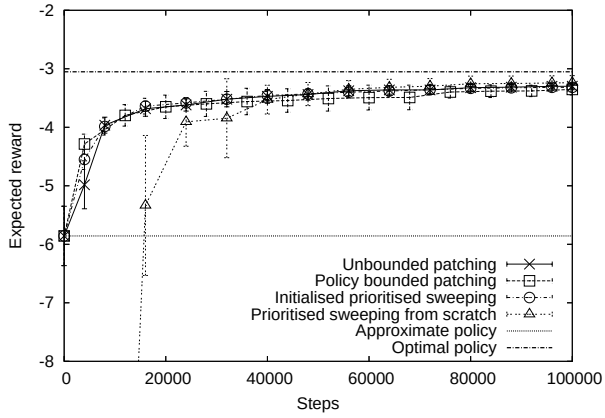
Solution	P_{sample} and R_{sample} capacity	Expected reward	# Q values
With utility bounding			
– Q_{patch} capacity 1700			
	50	-0.30 ± 0.73	632 + 1700
	100	-0.41 ± 1.76	632 + 1700
	500	0.24 ± 0.74	632 + 1700
	P and R known	0.69 ± 0.19	632 + 1700
– Q_{patch} capacity 1800			
	50	-0.08 ± 0.58	632 + 1800
	100	0.40 ± 0.31	632 + 1800
	500	0.78 ± 0.06	632 + 1800
	P and R known	0.89 ± 0.04	632 + 1800
With policy and utility bounding			
– Q_{patch} capacity 800			
	50	-0.45 ± 1.18	632 + 776.30 ± 48.61
	100	-1.27 ± 1.22	632 + 800
	500	-1.60 ± 0.99	632 + 800
	P and R known	-2.49 ± 2.50	632 + 800
– Q_{patch} capacity 900			
	50	-0.01 ± 1.03	632 + 815.70 ± 77.05
	100	0.06 ± 0.64	632 + 900
	500	-0.44 ± 1.05	632 + 900
	P and R known	0.01 ± 0.48	632 + 900
– Q_{patch} capacity 1000			
	50	0.01 ± 1.04	632 + 831.70 ± 98.06
	100	0.42 ± 0.40	632 + 974.00 ± 31.88
	500	0.10 ± 1.14	632 + 996.80 ± 9.60
	P and R known	0.46 ± 0.61	632 + 1000

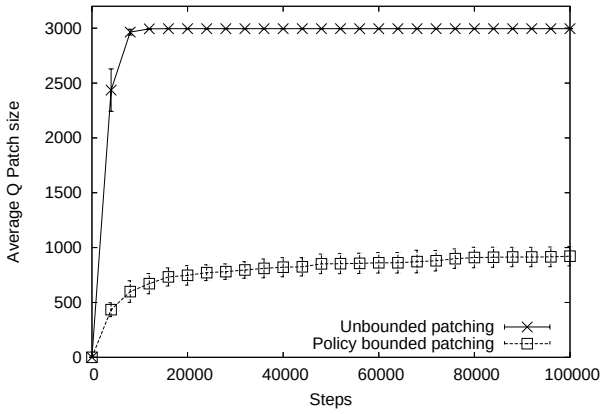
E.1.2 Modified Taxi, Discounted Terminating



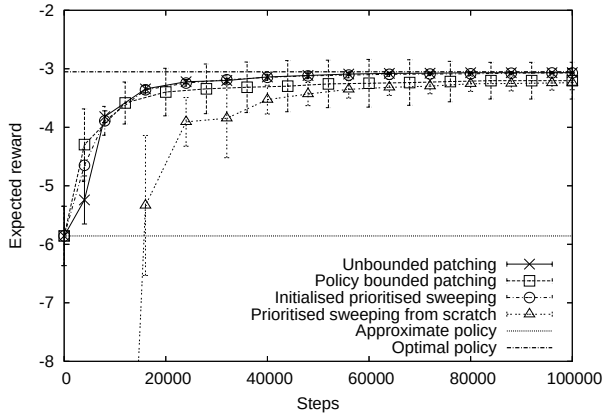


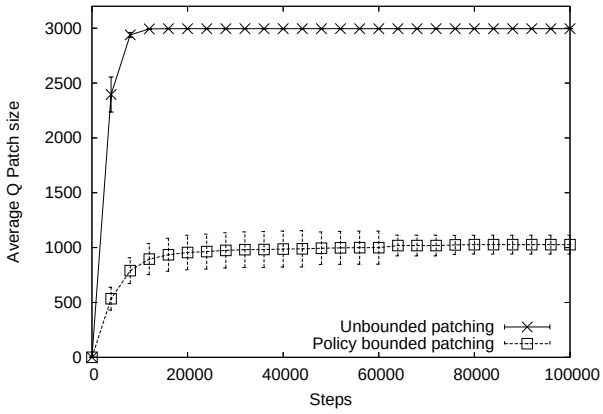
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 50 each, and prioritised sweeping from scratch.



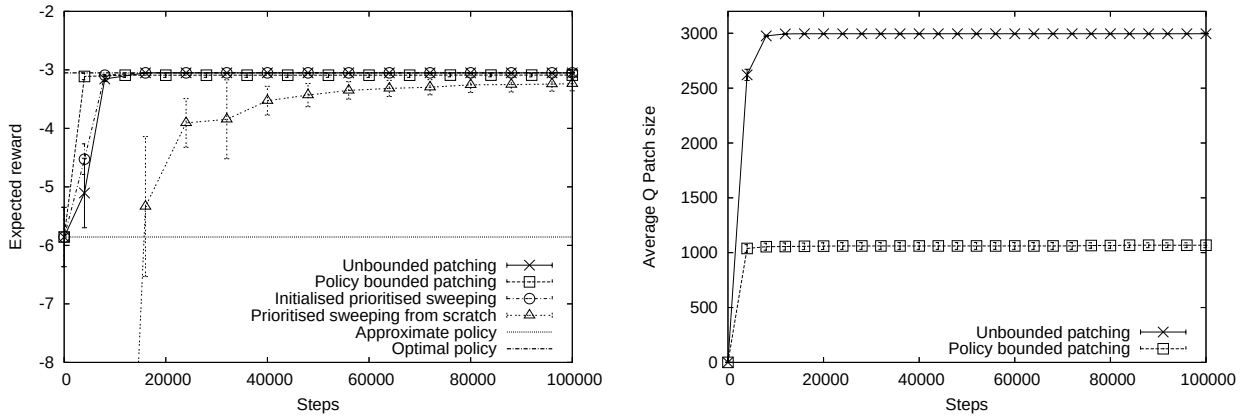


Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 100 each, and prioritised sweeping from scratch.

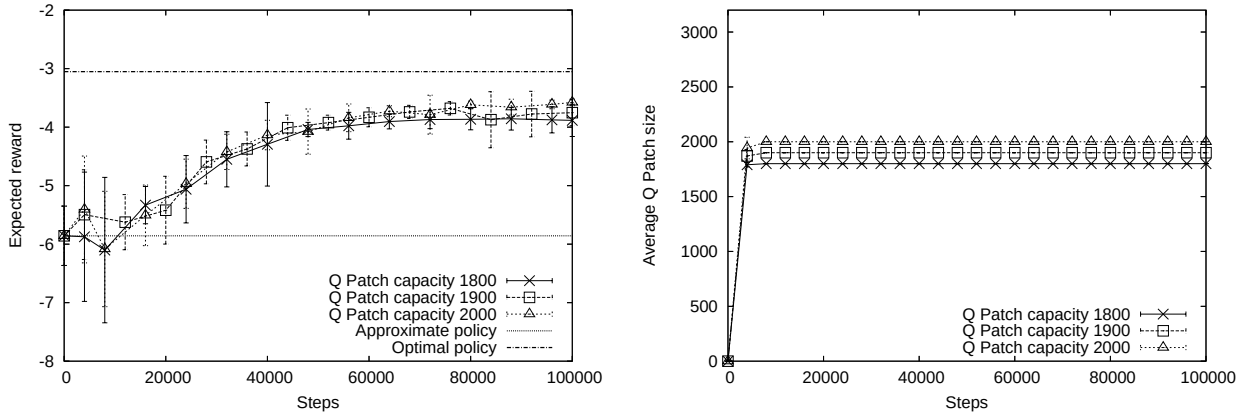




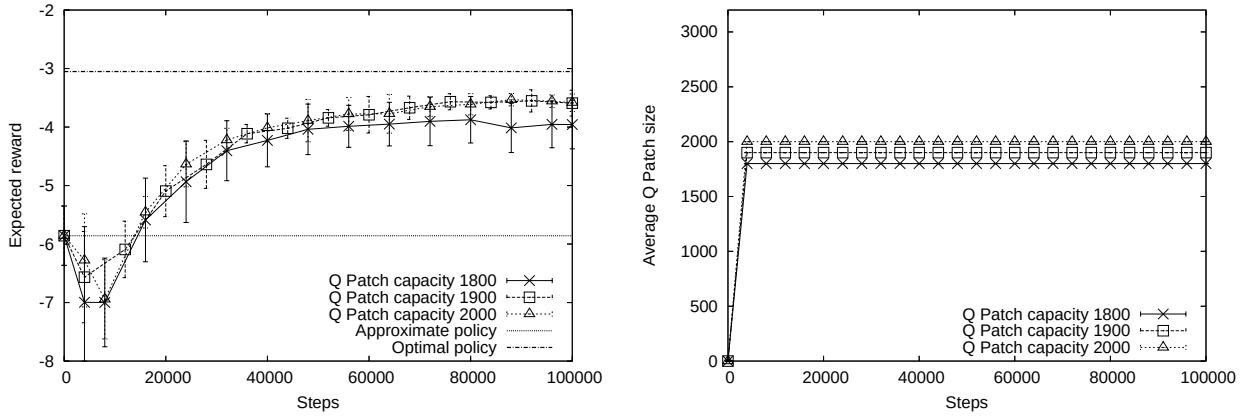
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 500 each, and prioritised sweeping from scratch.



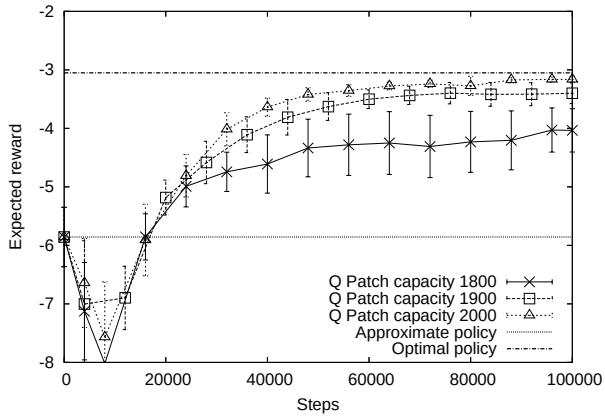
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with exact P and R known, and prioritised sweeping from scratch.

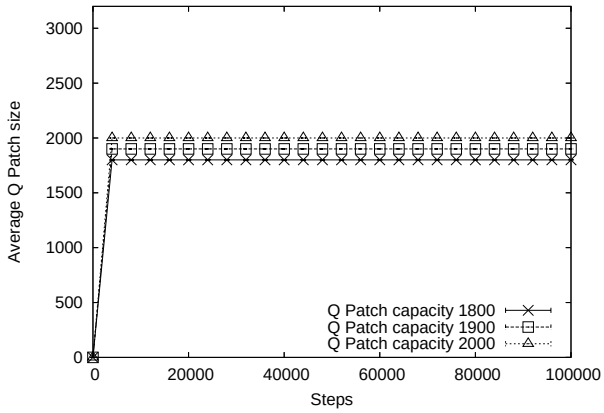


Patching with utility bounding, P_{sample} and R_{sample} capacities of 50 each.

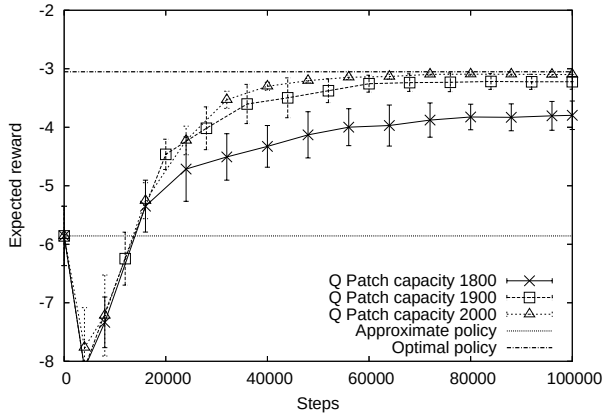


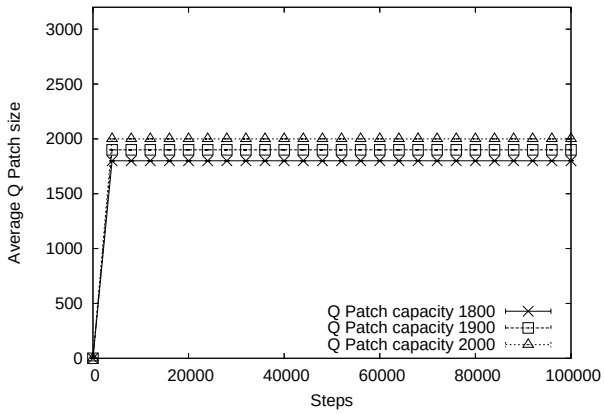
Patching with utility bounding, P_{sample} and R_{sample} capacities of 100 each.



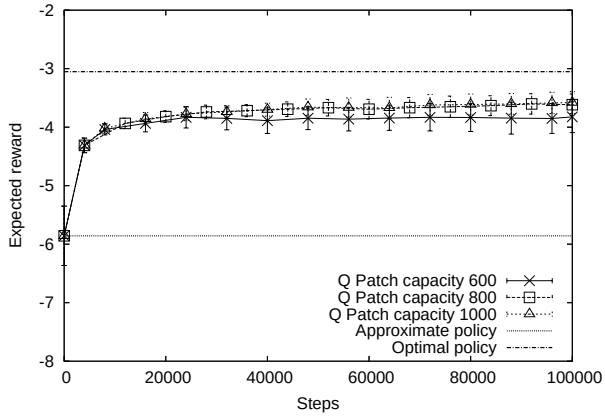


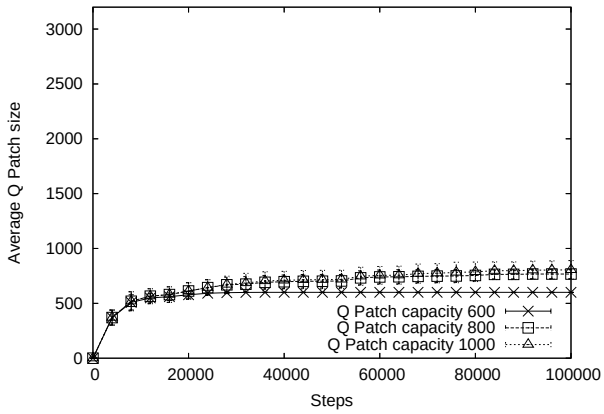
Patching with utility bounding, P_{sample} and R_{sample} capacities of 500 each.



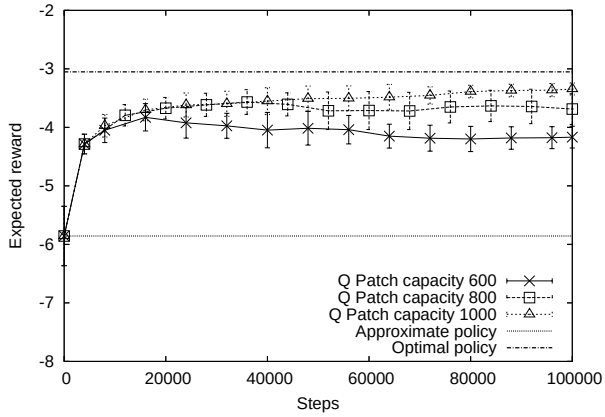


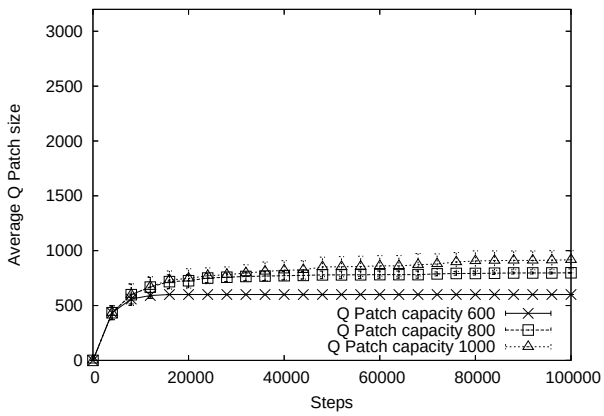
Patching with utility bounding, with exact P and R known.



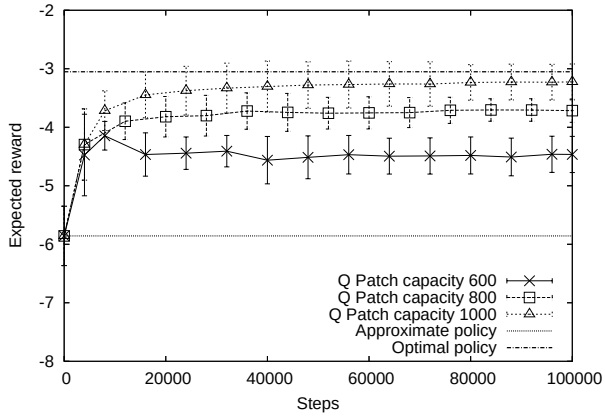


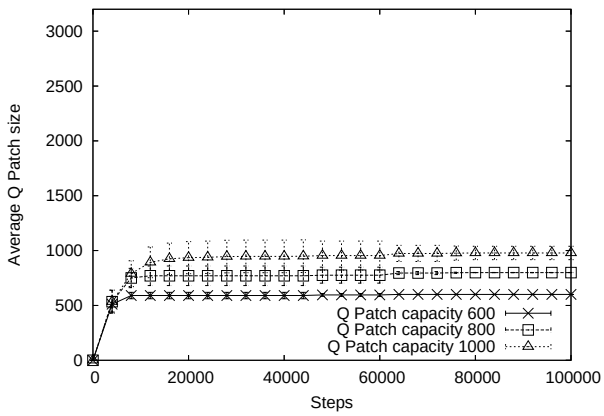
Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 50 each.



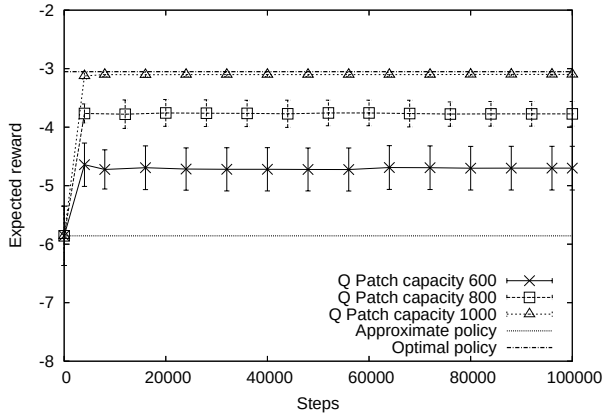


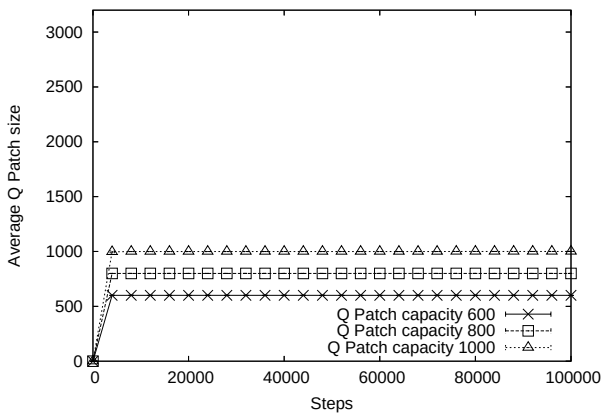
Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 100 each.





Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 500 each.





Patching with policy and utility bounding, with exact P and R known.

Table 10: Summary of results for the discounted terminating setting of the modified taxi domain. Figures were taken at the end of 100,000 steps. For patched solutions, the total size is listed as the size of $\hat{Q}$ plus the size of Q_{patch}.

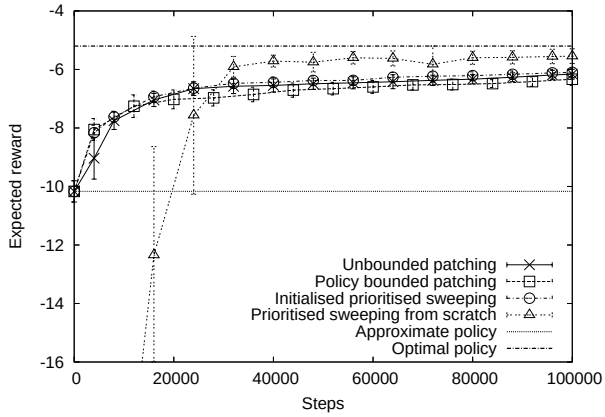
Solution	P_{sample} and R_{sample} capacity	Expected reward	# Q values
Initial approximation ($\hat{Q}$)		-5.86 ± 0.51	1208
Optimal flat		-3.05	3000
Prioritised sweeping from scratch		-3.24 ± 0.12	3000
Initialised prioritised sweeping			
	50	-3.51 ± 0.08	3000
	100	-3.32 ± 0.07	3000
	500	-3.07 ± 0.02	3000
	P and R known	-3.05 ± 0.00	3000
Unbounded patching			
	50	-3.49 ± 0.09	1208 + 2996
	100	-3.30 ± 0.04	1208 + 2996
	500	-3.07 ± 0.01	1208 + 2996
	P and R known	-3.05 ± 0.00	1208 + 2996
With policy bounding			
	50	-3.58 ± 0.19	1208 + 805.40 ± 84.57
	100	-3.35 ± 0.09	1208 + 921.60 ± 87.98
	500	-3.20 ± 0.31	1208 + 1027.40 ± 85.50
	P and R known	-3.09 ± 0.03	1208 + 1067.90 ± 22.45
With utility bounding			
– Q_{patch} capacity 1800			
	50	-3.88 ± 0.28	1208 + 1800
	100	-3.96 ± 0.42	1208 + 1800
	500	-4.03 ± 0.37	1208 + 1800
	P and R known	-3.80 ± 0.24	1208 + 1800
– Q_{patch} capacity 1900			
	50	-3.75 ± 0.22	1208 + 1900
	100	-3.59 ± 0.22	1208 + 1900
	500	-3.40 ± 0.18	1208 + 1900
	P and R known	-3.22 ± 0.13	1208 + 1900
– Q_{patch} capacity 2000			
	50	-3.58 ± 0.06	1208 + 2000
	100	-3.58 ± 0.15	1208 + 2000
	500	-3.16 ± 0.05	1208 + 2000
	P and R known	-3.10 ± 0.03	1208 + 2000
With policy and utility bounding			
– Q_{patch} capacity 600			
	50	-3.82 ± 0.27	1208 + 600
	100	-4.17 ± 0.19	1208 + 600
	500	-4.46 ± 0.31	1208 + 600
	P and R known	-4.70 ± 0.37	1208 + 600

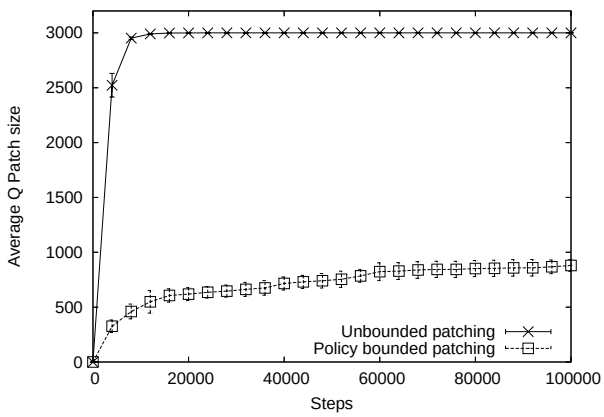
Continued on next page.

Table 10 (continued). Summary of results for the discounted terminating setting of the modified taxi domain.

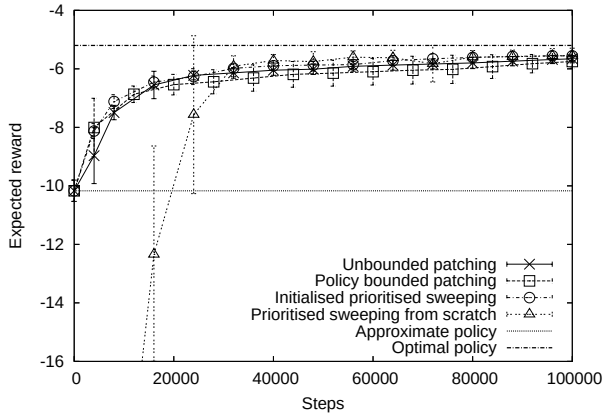
Solution	P_{sample} and R_{sample} capacity	Expected reward	# Q values
With policy and utility bounding			
– Q_{patch} capacity 800			
	50	-3.62 ± 0.18	1208 + 767.00 ± 42.28
	100	-3.69 ± 0.26	1208 + 798.40 ± 3.90
	500	-3.72 ± 0.20	1208 + 799.60 ± 1.20
	P and R known	-3.77 ± 0.21	1208 + 800
– Q_{patch} capacity 1000			
	50	-3.58 ± 0.19	1208 + 805.40 ± 84.57
	100	-3.34 ± 0.10	1208 + 917.20 ± 83.24
	500	-3.22 ± 0.31	1208 + 978.30 ± 60.89
	P and R known	-3.10 ± 0.03	1208 + 1000

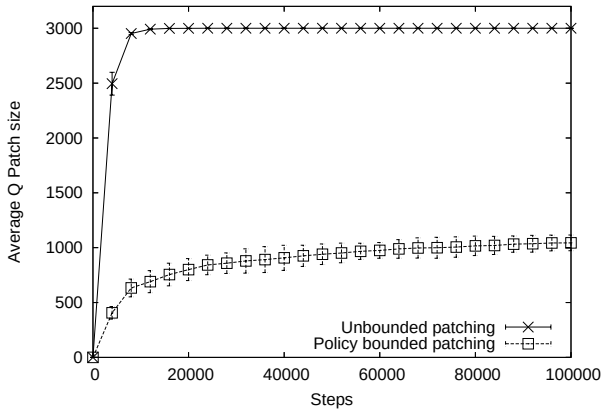
E.1.3 Modified Taxi, Discounted Continuing



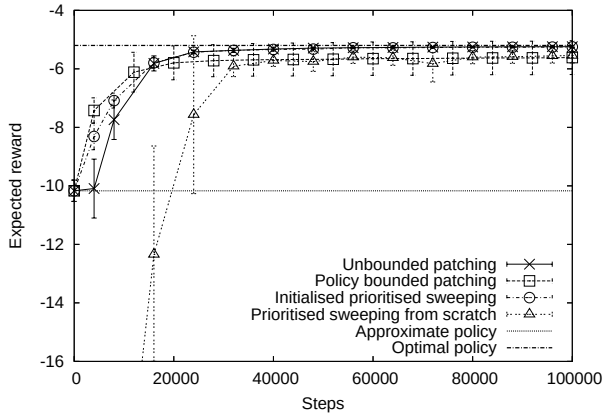


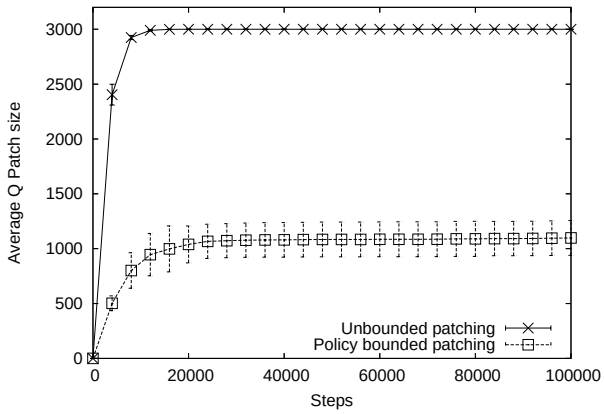
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 50 each, and prioritised sweeping from scratch.



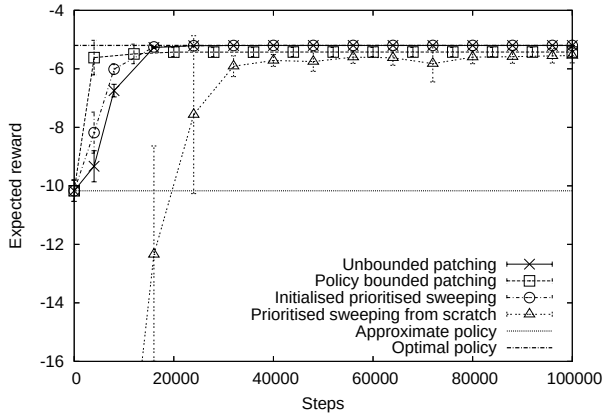


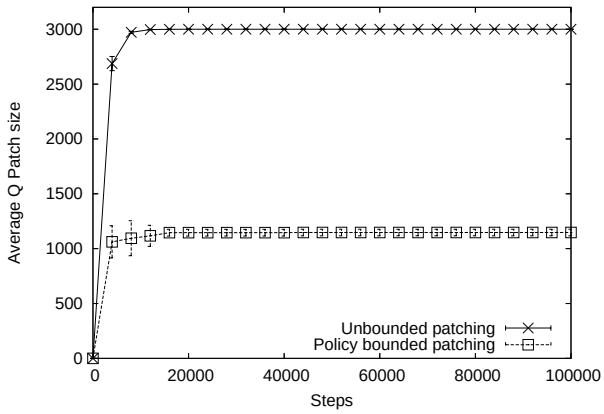
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 100 each, and prioritised sweeping from scratch.



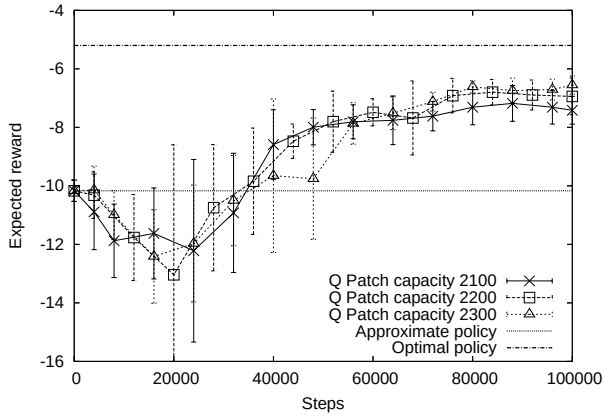


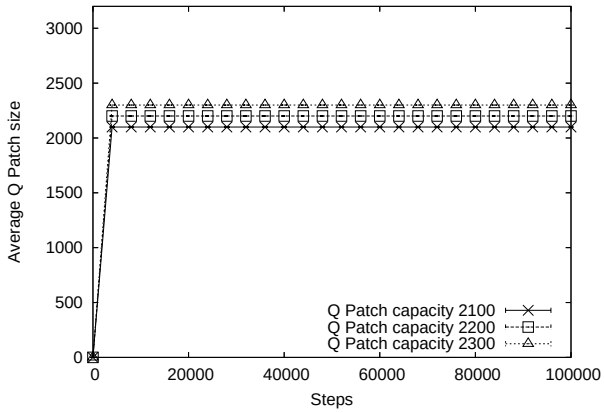
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with P_{sample} and R_{sample} capacities of 500 each, and prioritised sweeping from scratch.



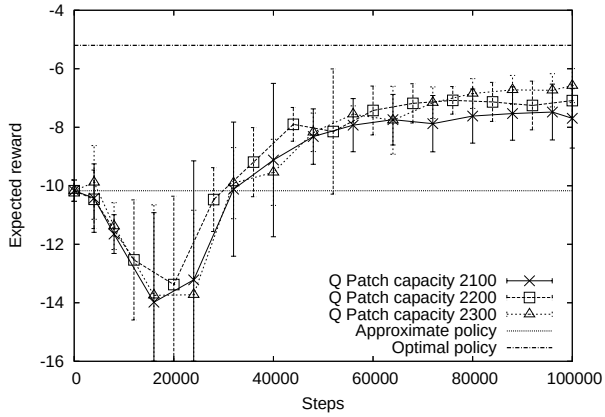


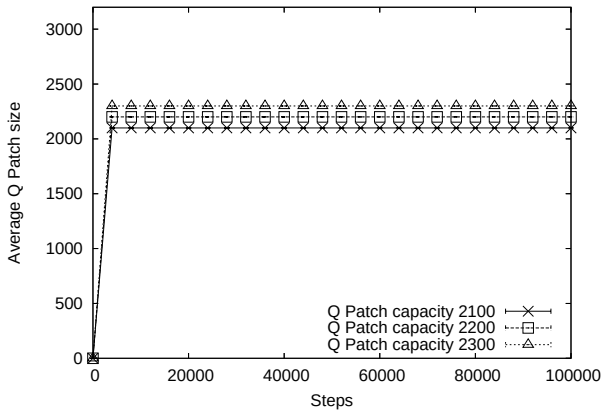
Unbounded patching, policy bounded patching, and initialised prioritised sweeping with exact P and R known, and prioritised sweeping from scratch.



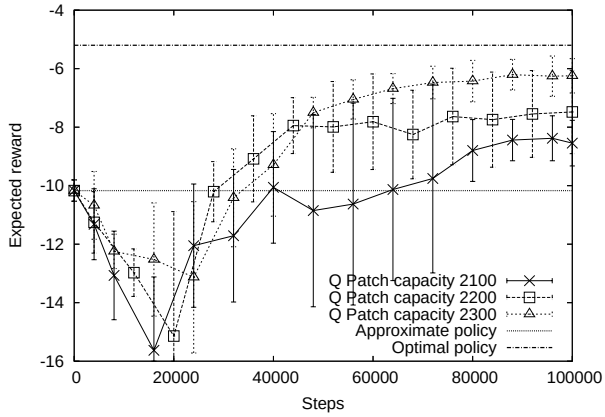


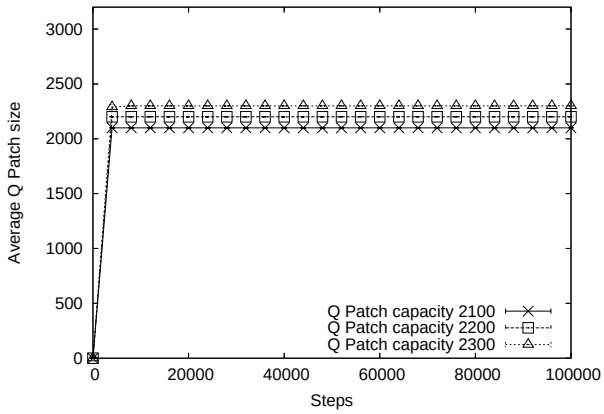
Patching with utility bounding, P_{sample} and R_{sample} capacities of 50 each.



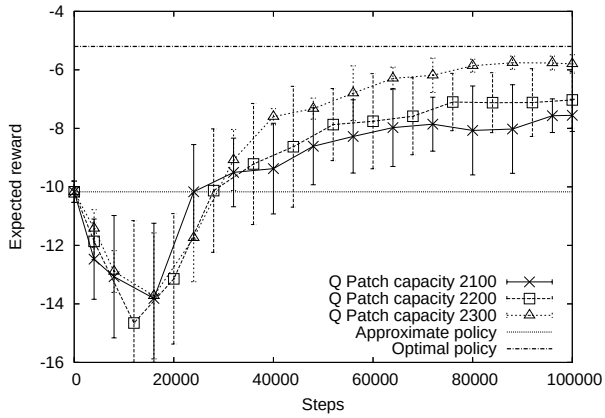


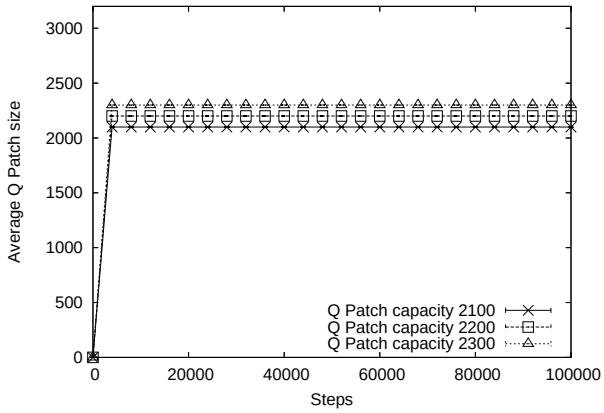
Patching with utility bounding, P_{sample} and R_{sample} capacities of 100 each.



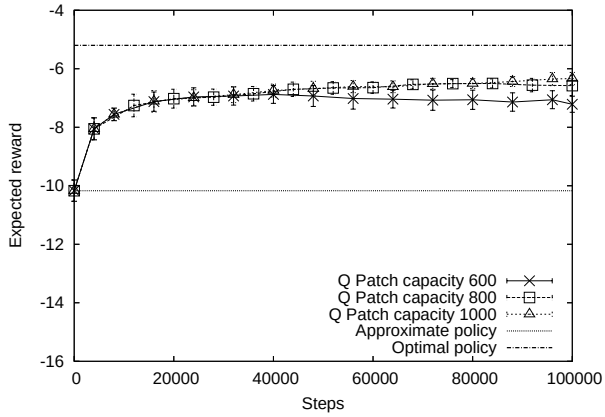


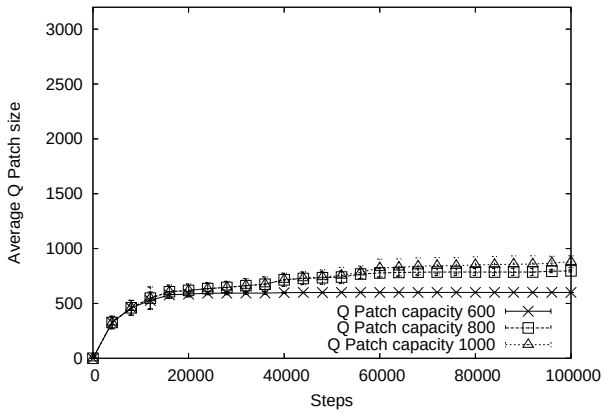
Patching with utility bounding, P_{sample} and R_{sample} capacities of 500 each.



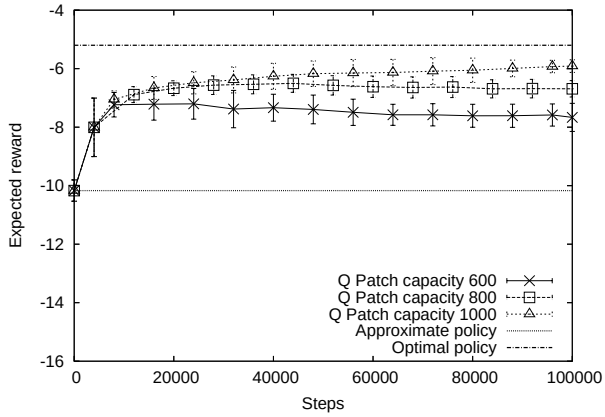


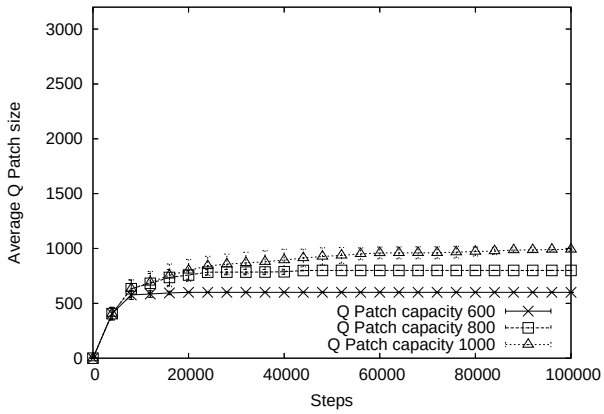
Patching with utility bounding, with exact P and R known.



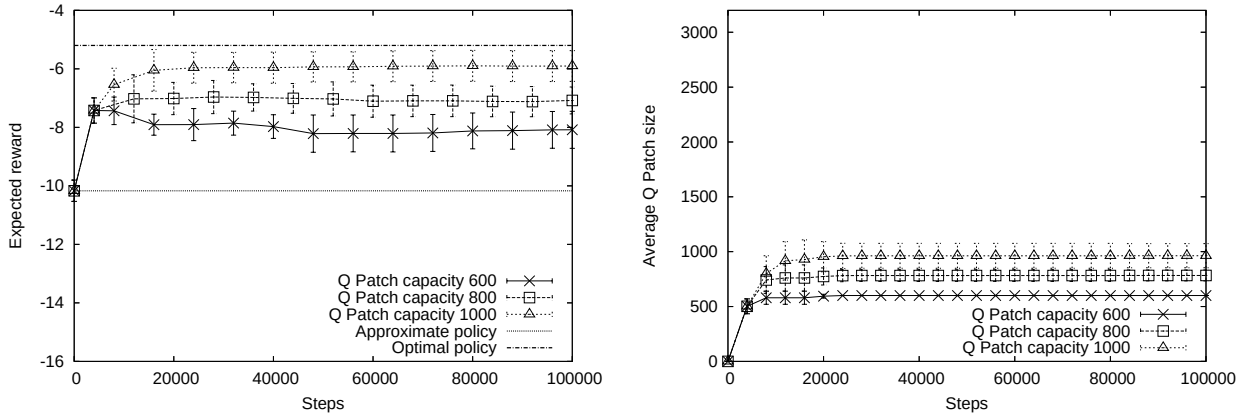


Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 50 each.

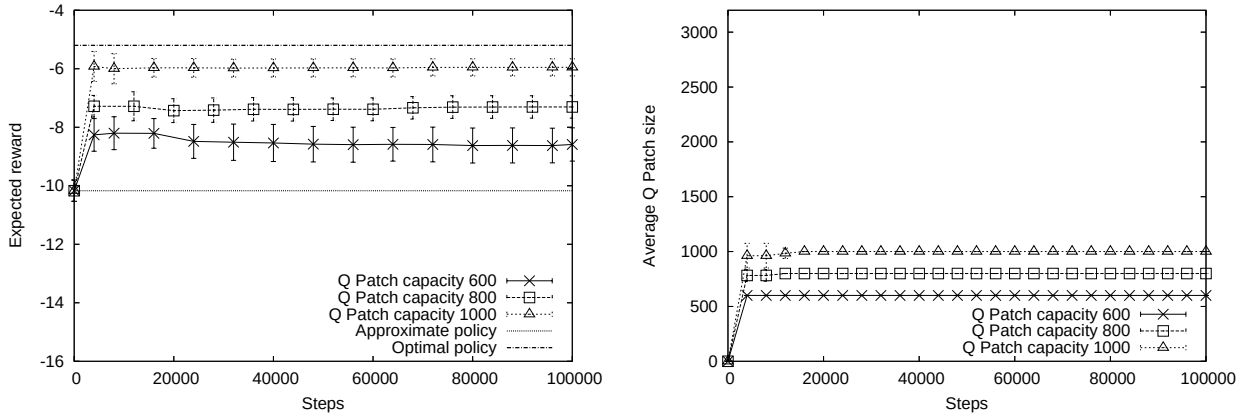




Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 100 each.



Patching with policy and utility bounding, P_{sample} and R_{sample} capacities of 500 each.



Patching with policy and utility bounding, with exact P and R known.

Table 11: Summary of results for the discounted continuing setting of the modified taxi domain. Figures were taken at the end of 100,000 steps. For patched solutions, the total size is listed as the size of $\hat{Q}$ plus the size of Q_{patch}.

Solution	P_{sample} and R_{sample} capacity	Expected reward	# Q values
Initial approximation ($\hat{Q}$)		-10.17 ± 0.36	1308
Optimal flat		-5.20	3000
Prioritised sweeping from scratch		-5.55 ± 0.25	3000
Initialised prioritised sweeping			
	50	-6.10 ± 0.14	3000
	100	-5.56 ± 0.16	3000
	500	-5.25 ± 0.03	3000
	P and R known	-5.20 ± 0.00	3000

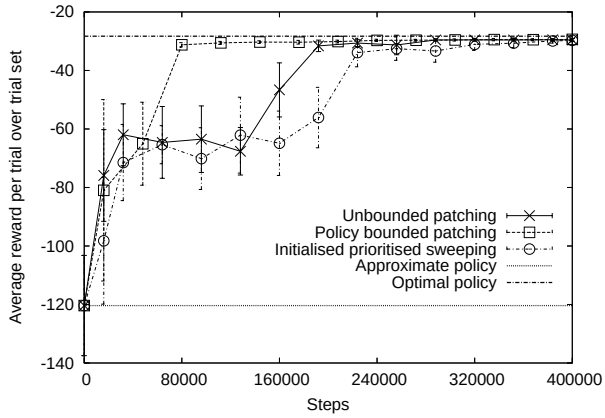
Continued on next page.

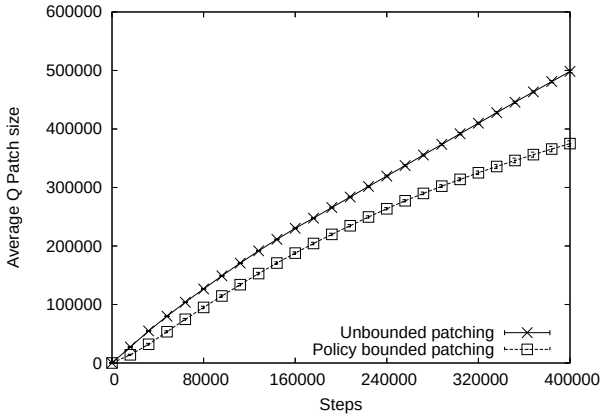
Table 11 (continued). Summary of results for the discounted continuing setting of the modified taxi domain.

Solution	P_{sample} and R_{sample} capacity	Expected reward	# Q values
Unbounded patching			
	50	-6.16 ± 0.17	1308 + 3000
	100	-5.66 ± 0.11	1308 + 3000
	500	-5.24 ± 0.03	1308 + 3000
	P and R known	-5.20 ± 0.00	1308 + 3000
With policy bounding			
	50	-6.33 ± 0.21	1308 + 880.00 ± 54.25
	100	-5.75 ± 0.23	1308 + 1043.90 ± 71.47
	500	-5.62 ± 0.58	1308 + 1098.00 ± 158.58
	P and R known	-5.43 ± 0.15	1308 + 1147.10 ± 30.24
With utility bounding			
– Q_{patch} capacity 2100			
	50	-7.42 ± 0.48	1308 + 2100
	100	-7.70 ± 1.02	1308 + 2100
	500	-8.55 ± 0.78	1308 + 2100
	P and R known	-7.55 ± 0.55	1308 + 2100
– Q_{patch} capacity 2200			
	50	-6.94 ± 0.63	1308 + 2200
	100	-7.09 ± 0.56	1308 + 2200
	500	-7.48 ± 1.42	1308 + 2200
	P and R known	-7.03 ± 0.97	1308 + 2200
– Q_{patch} capacity 2300			
	50	-6.55 ± 0.31	1308 + 2300
	100	-6.58 ± 0.58	1308 + 2300
	500	-6.24 ± 0.59	1308 + 2300
	P and R known	-5.80 ± 0.31	1308 + 2300
With policy and utility bounding			
– Q_{patch} capacity 600			
	50	-7.22 ± 0.27	1308 + 600
	100	-7.67 ± 0.48	1308 + 600
	500	-8.08 ± 0.63	1308 + 600
	P and R known	-8.59 ± 0.57	1308 + 600
– Q_{patch} capacity 800			
	50	-6.58 ± 0.34	1308 + 796.60 ± 10.20
	100	-6.69 ± 0.28	1308 + 800
	500	-7.08 ± 0.46	1308 + 783.40 ± 49.80
	P and R known	-7.31 ± 0.38	1308 + 800
– Q_{patch} capacity 1000			
	50	-6.33 ± 0.21	1308 + 880.00 ± 54.25
	100	-5.91 ± 0.22	1308 + 990.90 ± 15.05
	500	-5.90 ± 0.53	1308 + 963.40 ± 109.80
	P and R known	-5.95 ± 0.29	1308 + 1000

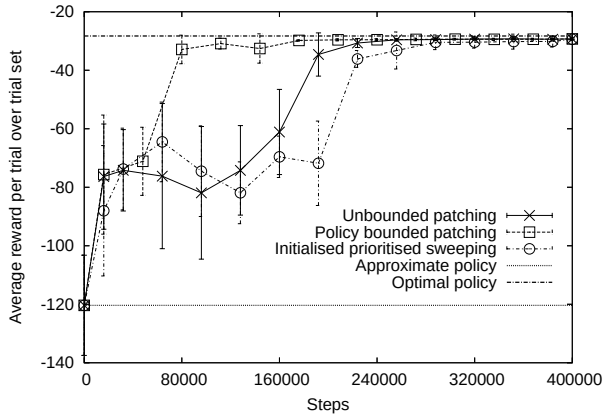
E.2 Multi-taxi

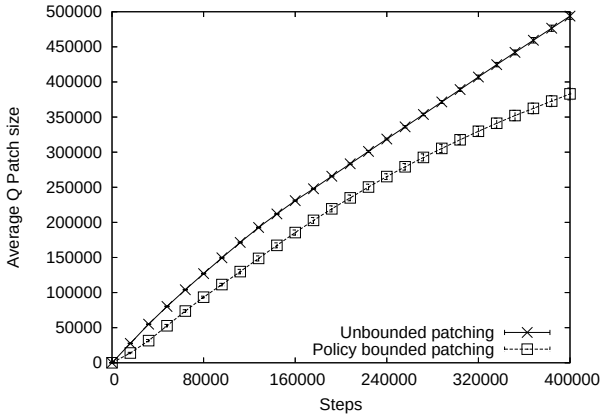
In the multi-taxi domain, we tried P_{sample} capacities of 10, 100, 1000, and 10000, and also with the exact P known. $\hat{R}$ is exact in this domain and was neither sampled nor patched. P_{sample} and P_{patch} were sampled and stored over taxi locations and actions only. For this domain, we do not include further results for prioritised sweeping from scratch; as discussed in Section 6.4, attempting to solve this problem from scratch falls far below methods initialised with the approximate solution.



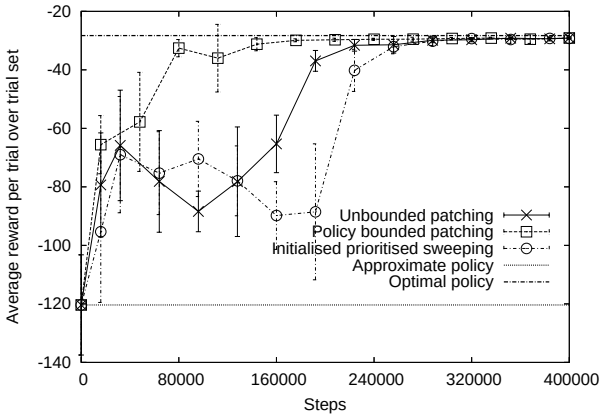


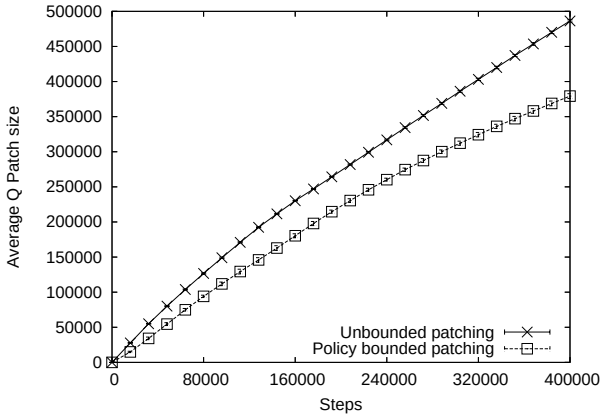
Unbounded patching, policy bounded patching, and initialised prioritised sweeping, P_{sample} capacity of 10.



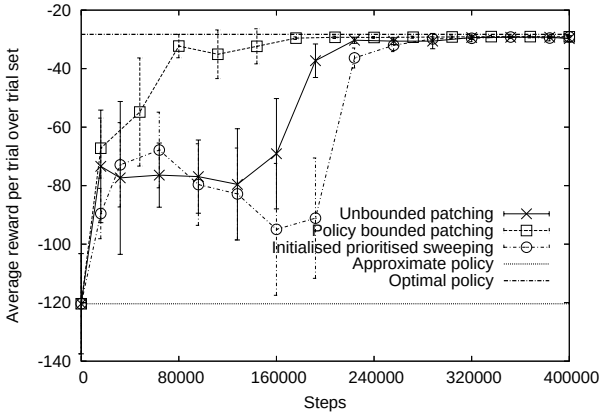


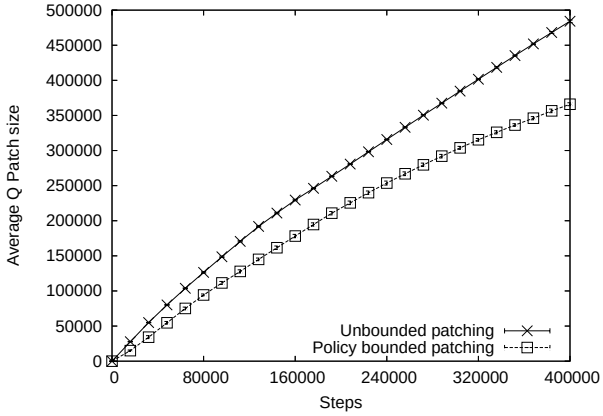
Unbounded patching, policy bounded patching, and initialised prioritised sweeping, P_{sample} capacity of 100.



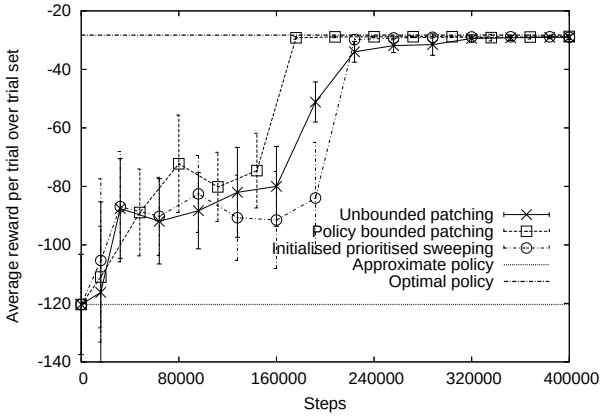


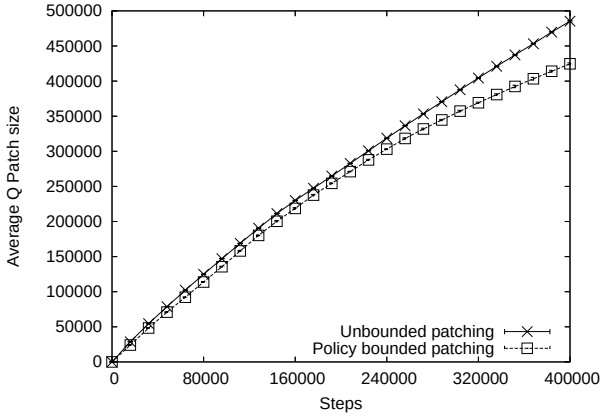
Unbounded patching, policy bounded patching, and initialised prioritised sweeping, P_{sample} capacity of 1000.



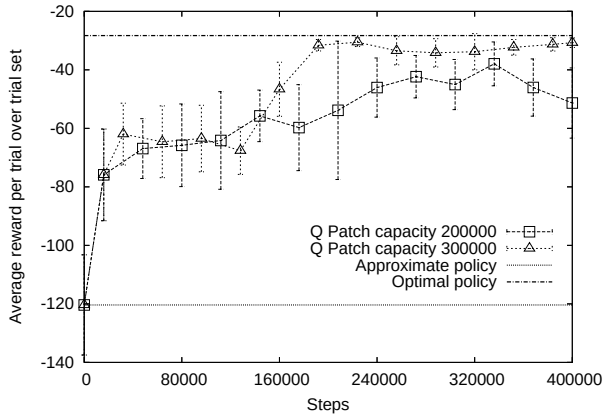


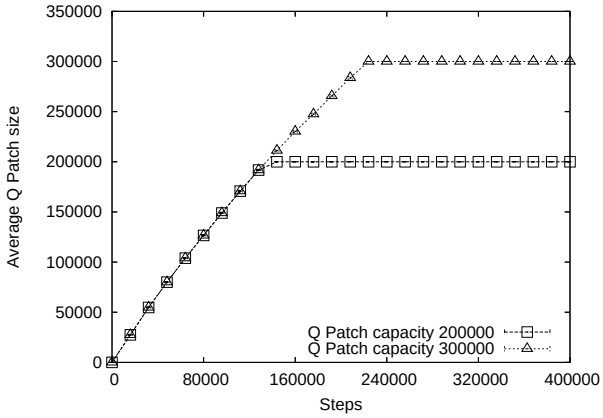
Unbounded patching, policy bounded patching, and initialised prioritised sweeping, P_{sample} capacity of 10000.



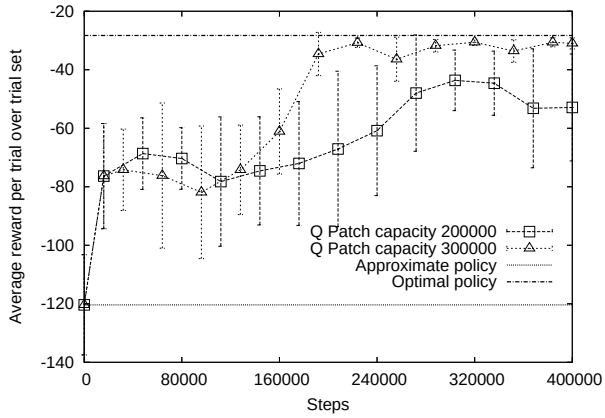


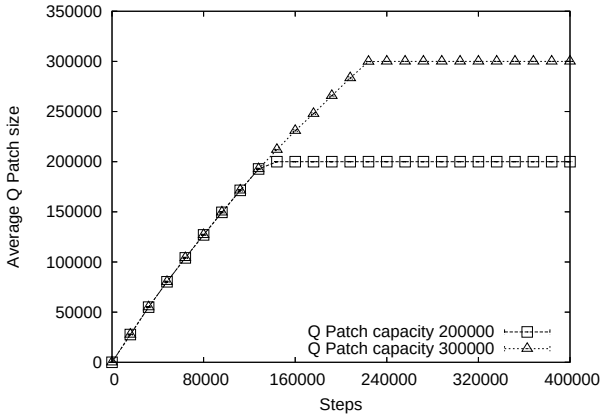
Unbounded patching, policy bounded patching, and initialised prioritised sweeping, with exact P known.



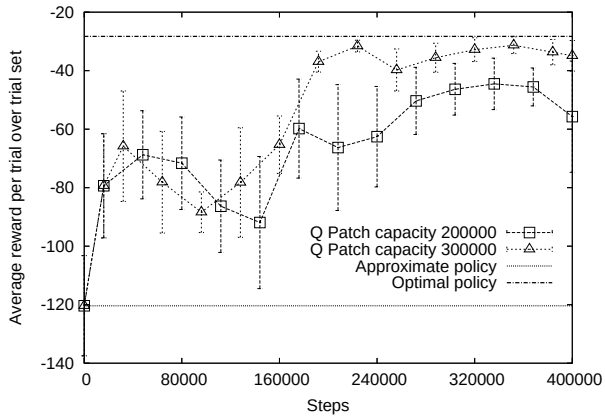


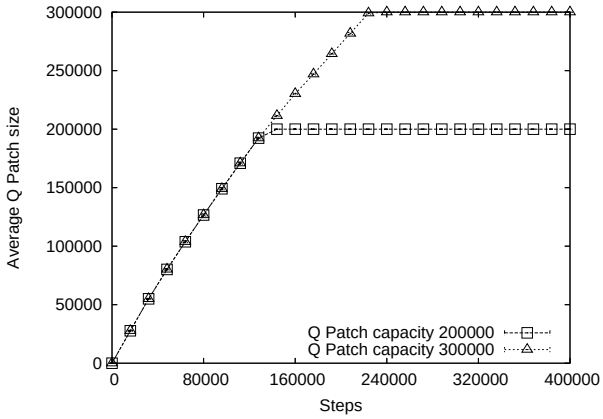
Patching with utility bounding, P_{sample} capacity of 10.



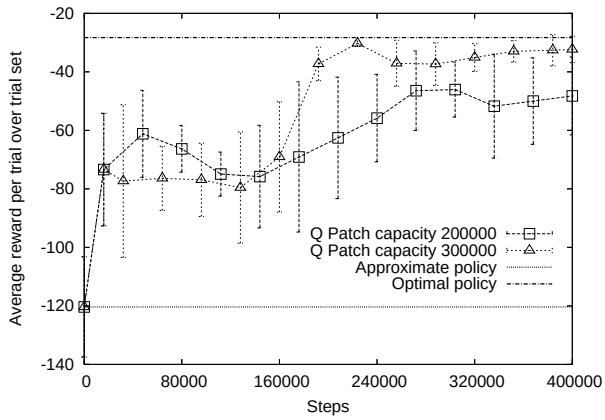


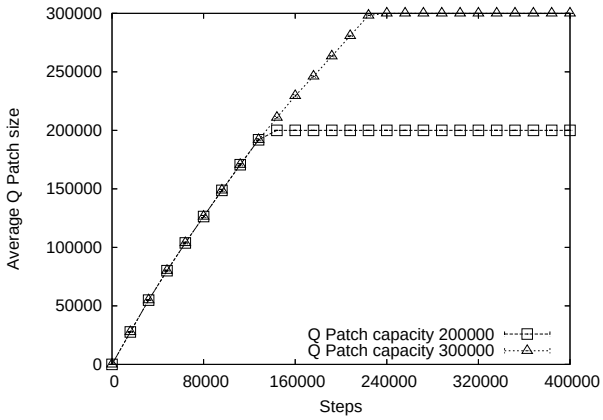
Patching with utility bounding, P_{sample} capacity of 100.



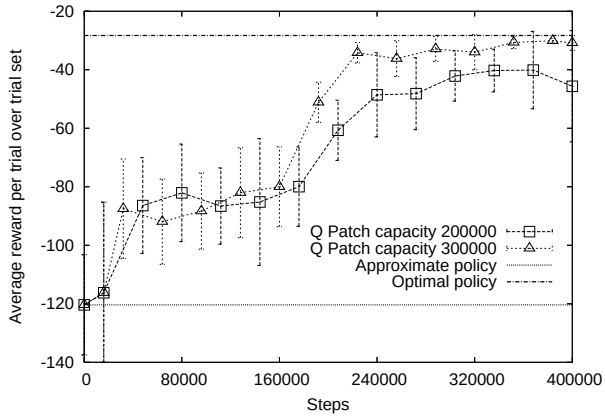


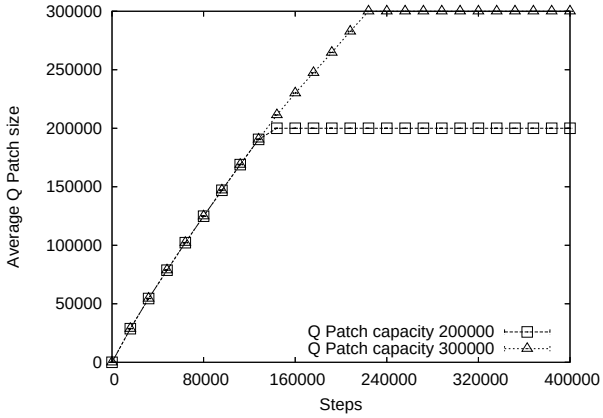
Patching with utility bounding, P_{sample} capacity of 1000.



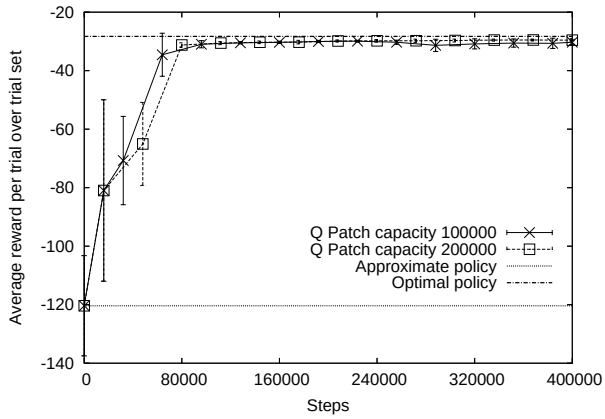


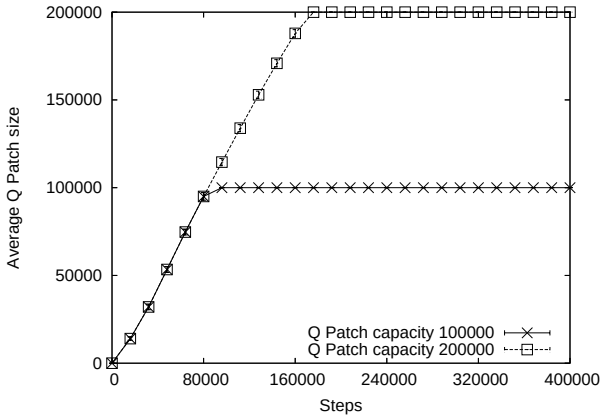
Patching with utility bounding, P_{sample} capacity of 10000.



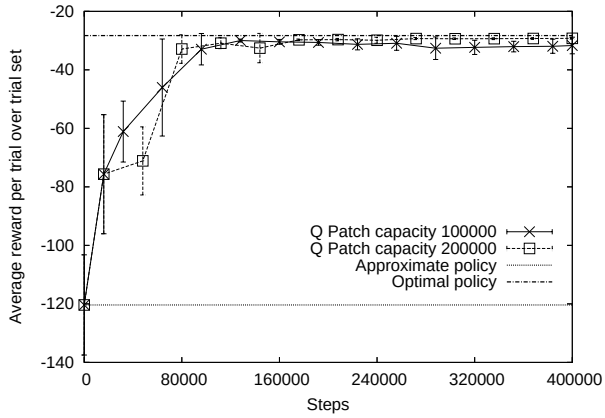


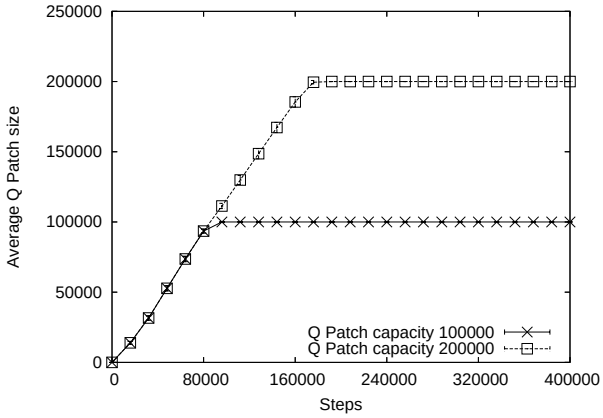
Patching with utility bounding, with exact P known.



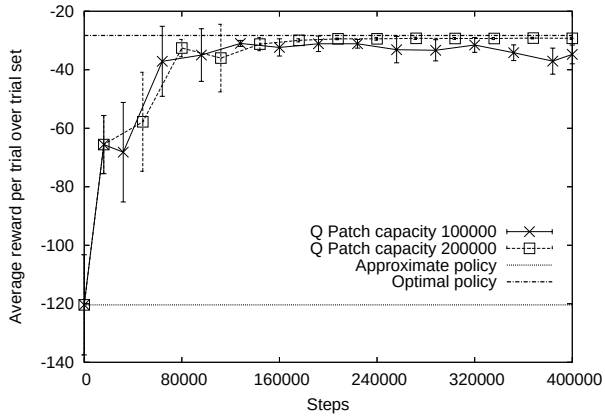


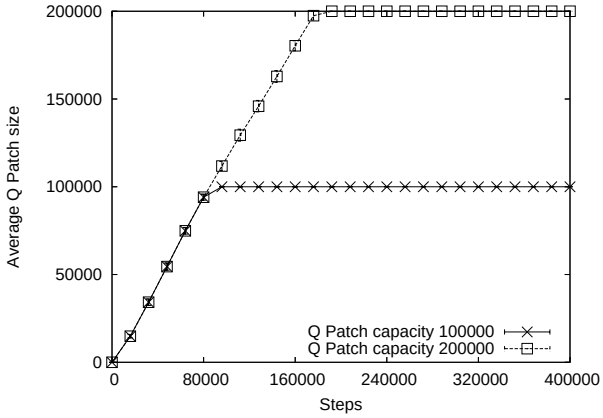
Patching with policy and utility bounding, P_{sample} capacity of 10.



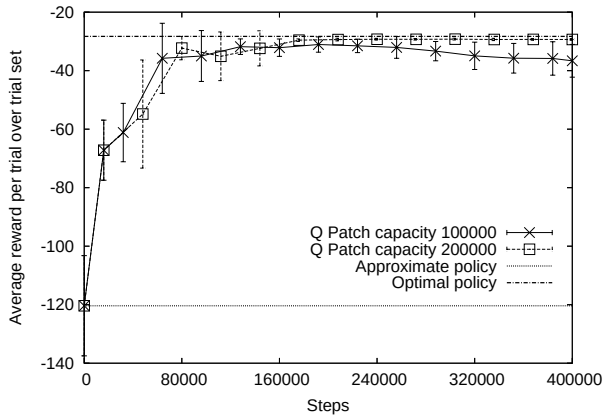


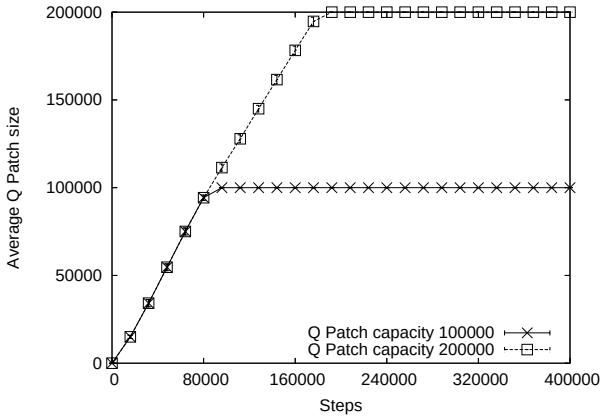
Patching with policy and utility bounding, P_{sample} capacity of 100.



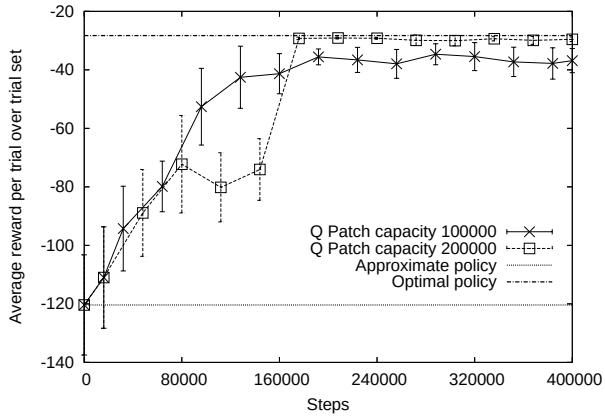


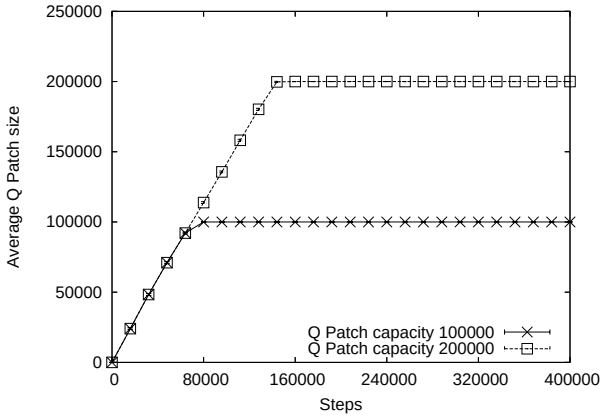
Patching with policy and utility bounding, P_{sample} capacity of 1000.





Patching with policy and utility bounding, P_{sample} capacity of 10000.





Patching with policy and utility bounding, with exact P known.

Table 12: Summary of results for the multi-taxi domain. Figures were taken at the end of 400,000 steps. For patched solutions, the total size is listed as the size of $\hat{Q}$ plus the size of Q_{patch}.

Solution	P_{sample} capacity	Average reward per trial	# Q values
Initial approximation ($\hat{Q}$)		-120.37 ± 17.12	4200
Optimal flat		-28.29 ± 0.28	17434200
Prioritised sweeping from scratch		-3184.23 ± 107.44	17434200
Initialised prioritised sweeping			
	10	-29.73 ± 0.51	17434200
	100	-29.42 ± 0.44	17434200
	1000	-29.26 ± 0.32	17434200
	10000	-29.14 ± 0.41	17434200
	P known	-29.02 ± 0.82	17434200
Unbounded patching			
	10	-29.55 ± 0.49	4200 + 498444.10 ± 2132.01
	100	-29.26 ± 0.32	4200 + 493807.40 ± 4598.51
	1000	-29.11 ± 0.27	4200 + 486358.40 ± 1580.45
	10000	-29.74 ± 1.02	4200 + 484161.50 ± 1740.74
	P known	-29.01 ± 0.79	4200 + 485341.60 ± 1021.22
With policy bounding			
	10	-29.36 ± 0.39	4200 + 374942.20 ± 4859.58
	100	-29.28 ± 0.28	4200 + 382924.80 ± 8548.53
	1000	-29.10 ± 0.33	4200 + 379263.10 ± 3444.89
	10000	-29.11 ± 0.39	4200 + 366005.10 ± 2097.67
	P known	-28.75 ± 0.31	4200 + 424674.80 ± 1300.28

Continued on next page

Table 12 (continued). Summary of results for the multi-taxi domain.

Solution	P_{sample} capacity	Average reward per trial	# Q values
With utility bounding			
– Q_{patch} capacity 200000			
	10	-51.34 ± 11.99	4200 + 200000
	100	-52.88 ± 18.29	4200 + 200000
	1000	-55.73 ± 19.01	4200 + 200000
	10000	-48.23 ± 13.31	4200 + 200000
	P known	-45.66 ± 19.04	4200 + 200000
– Q_{patch} capacity 300000			
	10	-30.72 ± 1.62	4200 + 300000
	100	-30.98 ± 1.91	4200 + 300000
	1000	-34.91 ± 5.23	4200 + 300000
	10000	-32.33 ± 4.52	4200 + 300000
	P known	-30.84 ± 2.51	4200 + 300000
With policy and utility bounding			
– Q_{patch} capacity 100000			
	10	-30.28 ± 0.83	4200 + 100000
	100	-31.67 ± 2.84	4200 + 100000
	1000	-34.75 ± 3.21	4200 + 100000
	10000	-36.63 ± 5.61	4200 + 100000
	P known	-36.85 ± 4.13	4200 + 100000
With policy and utility bounding			
– Q_{patch} capacity 200000			
	10	-29.56 ± 0.49	4200 + 200000
	100	-29.22 ± 0.30	4200 + 200000
	1000	-29.26 ± 0.37	4200 + 200000
	10000	-29.34 ± 0.30	4200 + 200000
	P known	-29.56 ± 0.79	4200 + 200000