Zida Wang
CS 4641

Markov Decision Processes

This paper uses code and data from *Reinforcement Learning (April 2017)*, by William Z. Ma. His paper can be found at https://github.com/willzma/CS4641-Machine-Learning. However, the analyses and conclusions are purely my own.

Abstract

This paper looks at two Markov Decision Process problems, one with a smaller number of states and the other with a comparatively greater number of states. I will look at three different reinforcement learning algorithms: value iteration, policy iteration, and Q-learning. The three algorithms will be run on both problems, and their resulting performances will be compared in terms of runtime, number of iterations, and how optimal the resulting policy is. Since Q-learning has various hyperparameters that could affect its performance, I will find the optimal hyperparameters before comparing its performance to that of value and policy iteration.

1. Introduction

1.1 Markov Decision Processes

Markov Decision Processes are an artificial intelligence problem in which a stochastic agent is placed in a well-known environment. The agent is stochastic (or random) in that, while it can choose an action to take at a specific state, the action that actually occurs can be different from the intended action, complicating the problem such that it can't be solved with simple graph searching algorithms. The environment is a well-known state-space, where states have a "rewards," and there are sink states that terminate the process upon being reached. The agent operates according to the Markov property, that the agent has no memory and its decisions only depend on the present state. Since this leads to the possibility of the agent infinitely wandering around the problem space, we can develop a utility function at each state that determines the policy at that state. The utility function is dependent on its surrounding states, but each states' influence is limited by a discounting function so that a state very far away doesn't have the same effect in determining utility as an adjacent state. The goal of the Markov Decision Process is to find the optimal policy (π^*) over the environment, where each state has the best action for the agent to take to maximize reward.

The following terms will be referred to in this paper and are essential pieces of the MDP:

- **S** is the set of all the possible states that the agent can be in
- **A** is the set of all possible actions that the agent can make
- **T(s,a,s') = Pr(s' | s,a)** is the transition function that describes the probability of reaching state **s'** by taking action **a** from state **s**
- **R(s)** is the reward associated with state **s**
- **π(s)** is the policy, the action to take, at state **s**

1.2 Value Iteration

Value iteration is a reinforcement algorithm that heavily relies on the utility of each state to calculate an optimal policy. Initially, the only information we are given is the reward function at every state. However, we can't use reward as the sole definer of the policy since there could exist intermediate states that don't necessarily have high rewards but lead to rewards. Therefore, we try to come up with a utility function at each state that captures the usefulness of traversing to that state. Ideally, if the usefulness of every state is perfectly captured in the utility value, then coming up with a perfect policy for the problem is straightforward. However, since we don't know the perfect utility value of every state, value iteration tries to find it by iterating through all the states multiple times, recalculating the utility for each state based on the utility immediately surrounding states.

The algorithm is as follows:

1. Start off with random utility values (within a reasonable range depending on the scale of the rewards) for every state
2. Iterate through every state and calculate its utility using the Bellman Equation

$$\widehat{U}(s)_{t+1} = R(s) + \gamma max_a \sum_{s'} T(s,a,s')\widehat{U}_t(s')$$

3. Repeat step 2 until the values converge (the total change in utility across the values falls under a tolerance value).

1.3 Policy Iteration

Value iteration relies on calculating the utility of each state. Since it has to loop through every state until convergence, it can take a long time to finish and takes extra iterations or get stuck in calculating values even though the resulting policy extracted from the utility values is the same. Thus, policy iteration comes in by bypassing much of the tedious calculation in value iteration.

The algorithm is as follows:

1. Assign to each state a random action to take to create a policy for the MDP.
2. Iterate through each state and calculate its utility according to the current policy.
3. Update the utilities of all the states at the same time and create a new policy based on the the new utilities.
4. Repeat steps 2 and 3 until no action in the policy changes between two loops.

The value iteration Bellman equation discounts further states using γ, but the policy iteration update equation doesn't use need to discount further states.

1.4 Q-Learning

Q-Learning is an improvement to value and policy iteration in that it can learn the optimal policy over a MDP problem without the requirement of knowing the rewards of every possible state. There are many situations in which the rewards of all possible states are not known and finding all of them is not feasibly possible. The value and policy iteration algorithms are rather deterministic since all the information is known at the beginning. Also, if the MDP has a very large number of states, iterating and updating every single state can be quite costly. Also,

calculating every single possible state is very inefficient if the agent realistically only deals with a small fraction of those states most of the time. Q-Learning aims to solve the problem of not knowing all the information about every reward state. Like value iteration, the algorithm initially assigns a random value, the Q-value, to every state. Then, we start at the starting point and repeated move through the states according to the Q-values until we reach a sink state. As we visit different states, we update the Q-values of those states with the Bellman equation:

$$Q(s,a) = R(s) + \gamma \sum_{s'} T(s,a,s') max_{a'} Q(s',a')$$

But if it was simply just like this, the stochastic nature of the agent would not be accounted for. As it is now, the algorithm could easily get stuck on local maximal optimal paths, as it could just continue to update the current path that it takes over and over again, even though there is truly a better alternative that just isn't discovered due to unfortunate random Q-value assignment. To fix this, we apply the epsilon greedy technique into the Q-value learning process. Instead of always going to the next most optimal value, we give an epsilon (a probability) chance for the algorithm to explore. This way, states that don't lie on the locally optimal path can still be updated to see whether or not they are truly better alternative states to be in. Epsilon greedy is the technique used in our analysis, but we will discuss and compare other exploration techniques.

1.5 GridWorld

GridWorld is found in the Java SDK package, and it is a convenient substrate to run MDP experiments on. It models a grid of equally sized squares, and we can use each square to represent a state. Each square will also have an associated reward function. In our GridWorld models of MDPs, we will use a gray circle to represent the agent, black squares to represent walls and obstacles, and blue squares to represent sink states. Both GridWorlds will assume a stochastic transition function for the agent that has an 80% chance of taking the intended action and a 20% chance of taking any action that is not the intended one. Since the actions are Up, Down, Left, and Right, the 20% chance is split among the other 3 possible actions (unless one of those actions is impossible due to a wall or obstacle).

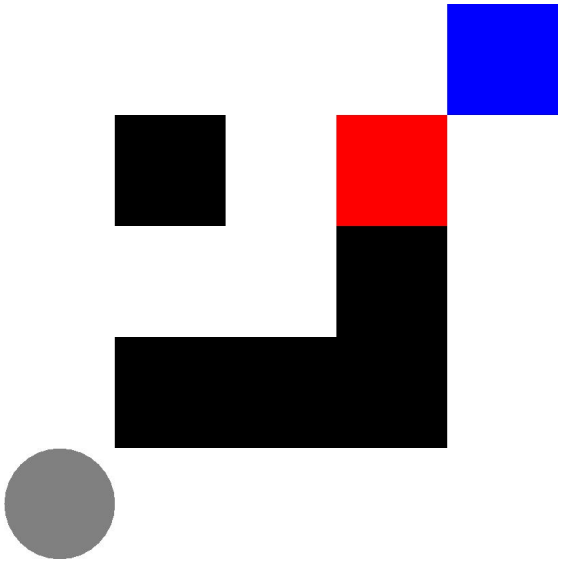
Since GridWorld is very simple and rectangular by design, the MDPs trying to be modeled by them may seem to be oversimplified. However, GridWorld can represent different levels of abstraction. While it will never be able to truly capture the continuous nature of real life MDPs, GridWorld can be changed to be increasingly specific by increasing the size and making each state represent more specific situations. But there is also usefulness in simplifying an environment into a MDP, in that it can rule out many useless details and focus on what is important to the task at hand.

2. Easy GridWorld

2.1 Problem Description

We will compare value iteration, policy iteration, and Q-learning in terms of time and number of iterations before convergence on an easy GridWorld MDP so see how they perform in a smaller,

controlled environment, and then in the next section we will look at their performance on a larger, more complex GridWorld MDP.

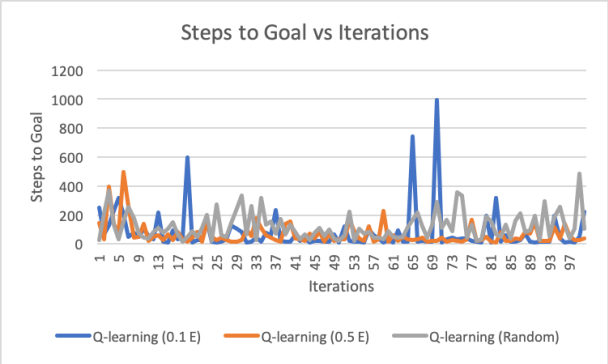


The figure on the right is the "easy" GridWorld MDP that we will be comparing the reinforcement learning algorithms on. In the original paper that it is from, the author likened the problem as getting to a work meeting, where the red state is a toll booth. However, I do not particularly see the stochasticity in that example, since the route that a person takes to get to work is not a very stochastic process since people drive with a predetermined route in mind and rarely randomly deviate from said route. Thus, there is no need for a policy. Instead, the situation can be extrapolated as a robot navigating an environment. Imagine a robot navigating a room within a house. The obstacles (black squares) would be pieces of furniture that obstruct the robot. The extreme negative reward state (red square) would be a liquid spill (if the robot gets into the spill, the liquid would get into it and break the robot), and since the consequences are so severe, the reward is set to -10000. The positive reward goal state (blue square) would be where the robot realizes its entire reason for existence, but since such a realization is so high up in Maslow's hierarchy of needs and not necessary for survival, it only gets a reward of 1000. In all the other squares, there is a linearly decreasing negative reward function such that states closer to the goal will have less reward so that the robot doesn't just wander around forever (since it can maximize rewards by just wandering around if the rewards are positive and will always take the safest path if the rewards are zero). The stochasticity of the problem comes from the conversion of the action from digital to analog and other possible environmental confounding factors, such as a dog running around the room that might knock around the robot. The optimal path of the problem will take 8 steps.

2.2 Optimizing Q-Learning

Before comparing Q-Learning with the other reinforcement learning algorithms, we must find the most optimal hyperparameters so that it can bring its best game. We will analyze three different exploration strategies that are found in the BURLAP library: epsilon-greedy exploration, Boltzmann exploration, and simply random exploration. Epsilon greedy explores randomly based on a decreasing epsilon probability over time, the Boltzmann exploration uses the Boltzmann probability distribution, and simply random exploration just randomly exploring using a random number generator. Epsilon greedy has the hyperparameter of epsilon itself, Boltzmann has the hyperparameter of temperature, and simple random has no hyperparameters (it's just random).

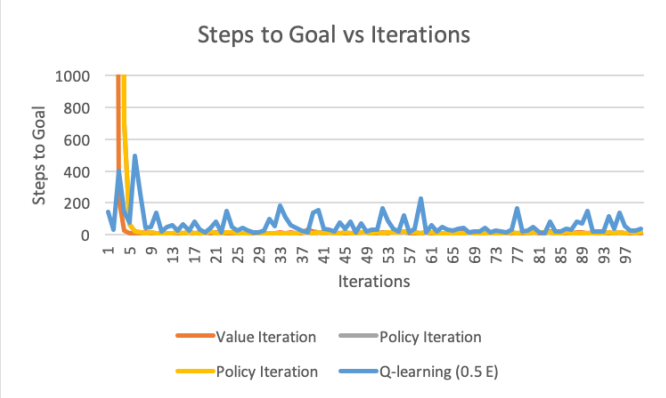
We don't include the Boltzmann exploration technique in the graph comparison since the number of steps that it takes per iteration is extremely inconsistent and that ranges way out of the range of the other techniques.



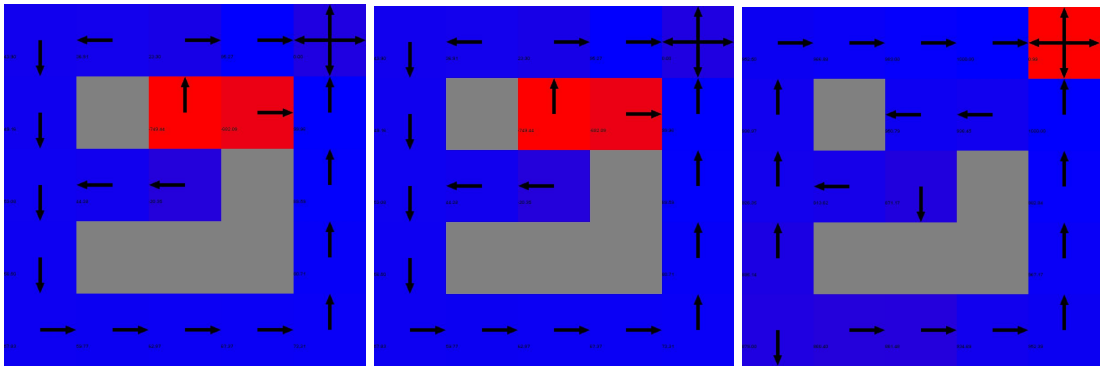
The data shows that Epsilon Greedy with Epsilon = 0.5 converges on a low number of steps rather consistently (but doesn't reach the optimal since it still has a randomness in each iteration).

2.3 Performance Comparison

Using the newly found optimal Q-learning exploration strategy and hyperparameter, we can now compare the performances of value iteration, policy iteration, and Q-learning on this "easy" GridWorld MDP in terms of time and iterations to convergence.



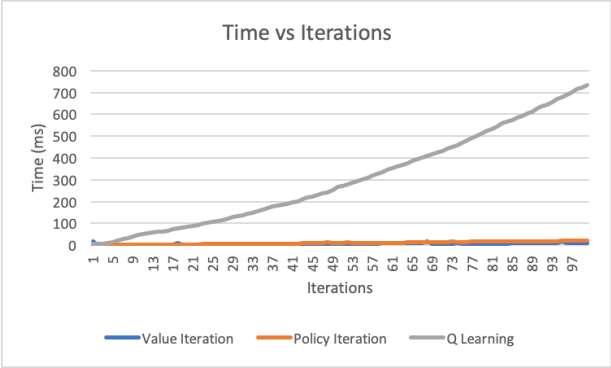
First we will look at the steps to the goal. We see that although value and policy iteration start off extremely high (>30000 steps) due to having randomized utility values, they can quickly converge on a minimum deterministically. Q-Learning, since it has an inherent random choice in every iteration, still has high numbers of steps to the goal after many iterations. This makes sense because value iteration and policy iteration have more information about the environment, knowing the rewards of every state and iterating over every state, whereas Q-learning only is able to update the Q-values that it traverses in its iterations. Let's look at the policies now:

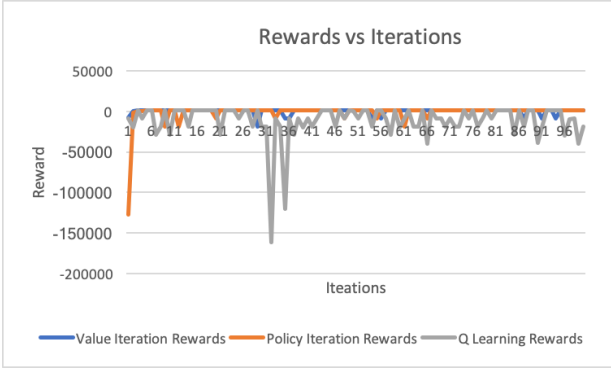


Value Iteration (Left), Policy Iteration (center), Q-Learning (right)

What we see is that value and policy iteration have found the most "logical" routes. It aims to avoid the toll booth until the latest it can, which reduces the risk of ending up in the toll booth due to random chance. By starting to taking the going up then right route, the agent has a chance of going through the middle path, which has more chances to end up at the toll booth. Therefore, value and policy iteration correctly choose to go down on the left side, to take the safer going right then up route. Interestingly, Q-learning doesn't show the same intuition and would continue on that path. Also, it doesn't show as strong recognition of the negative reward state, since the squares are still blue, but it still chooses to move away from it in the case that it goes in the middle route. Q-learning therefore ends up with a riskier policy than value and policy iteration, but it is not necessarily incorrect. This makes sense due to the fact that it has less knowledge of the rewards.

Since value and policy iteration end up updating every single state, we should still check the time efficiency and the rewards of the algorithms in comparison to Q-learning.



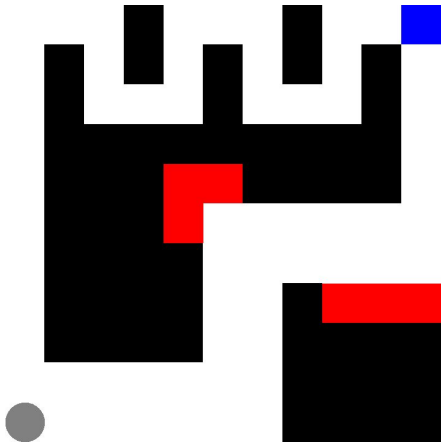


Q-learning cumulatively takes a lot longer than that of value and policy iteration since it is constantly randomly exploring. Value and policy iteration don't need to explore since they ultimately assess every single possible state, so their iterations are very deterministic and result in being stable in runtime per iteration. As for rewards, it shows the same thing. Value and policy iteration reach higher rewards faster and stay consistently higher, but Q-learning is erratic due to the fact that a random exploration can send it through many many more steps. Also, the fact that the non-positive reward goal states have a negative reward value (so that the agent doesn't just wander forever) is shown in the more steps that Q-learning has to take due to random chance.

The fact that value and policy iteration perform much better than Q-learning makes sense since the number of states is relatively small. Out of both value and policy iteration, policy iteration seems to perform better since value iteration still seems to have some bumps at later iterations. This is probably due to the overfitting nature of value iteration where the utility values get too specific, and policy iteration avoids this due to only looking at the policy.
Overall, we see that value and policy iteration work (in this case value ite very well in situations in which the number of possible states is small and in situations where the reward function is known.

3. Hard GridWorld

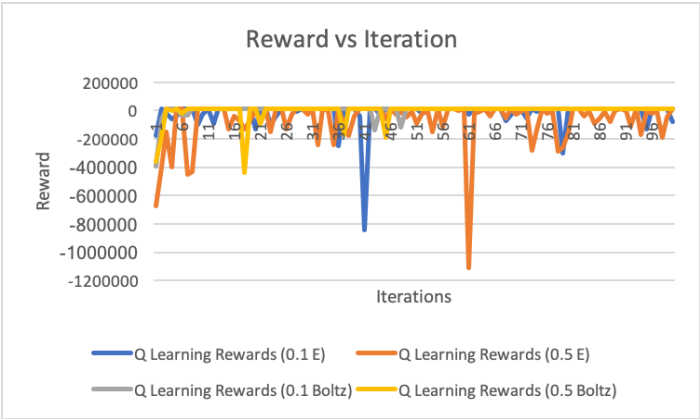
3.1 Problem Description



To make this problem more complex, we expanded the number of states to 63, added more nooks and crannies in the obstacles, and increased the number of negative reward states.
This problem has a longer winding path on the top path and many more opportunities to end up in a damaging liquid on the right path, which makes it harder for even a rational human to determine which path is the most optimal to take.
To reframe the context of this question, the robot is placed in a larger room this time. One route through the room to the goal is through a maze of furniture, and the other route is through the kitchen, where the homeowner has been too lazy to clean up all the mess he or she has made after cooking dinner. Once again, there is a chance of discrepancy between the digital intended action and the actual analog action and a pet that may knock the robot around.

3.2 Optimizing Q-Learning

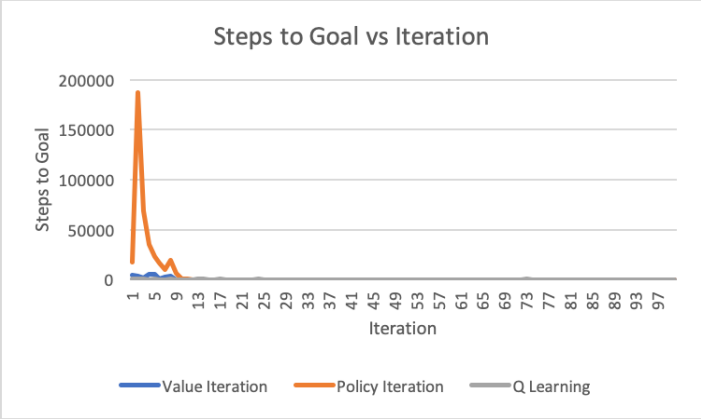
With a new MDP comes the need to find new optimal hyperparameters for Q-learning. We will compare the same three exploration techniques as before, but this time on the basis of reward per iteration since the paths and policies are bigger due to increasing the size of the MDP state space.



This time, we didn't include random exploration in the figure due to it taking many more steps and thus having extremely less reward compared to the other techniques. This makes sense due to the fact that the problem space is so much larger than the "easy" GridWorld problem, making it increasingly improbable for an iteration to randomly stumble into the goal state. This time, we also find that the Boltzmann strategy with a 0.1 temperature hyperparameter is consistently higher in reward than the other strategies as well as converging to a higher reward state much faster than the other strategies. Thus, we will choose that as our Q-learning exploration strategy.

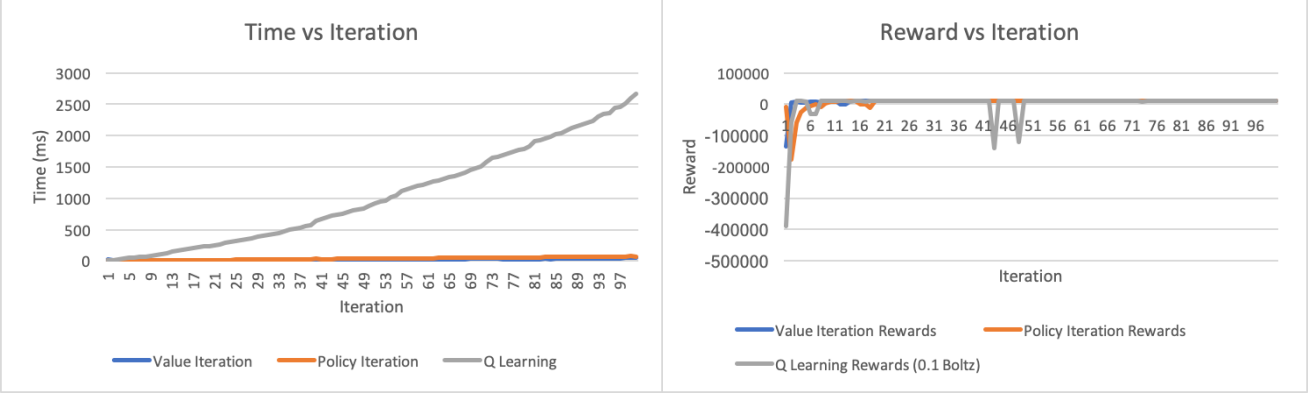
3.3 Performance Comparison

Now we will once again compare the reinforcement learning algorithms with the optimal Q-learning hyperparameters:



This time, we see that Q-learning performs significantly better than value iteration and overall better than value iteration. It converges on a minimal range much faster. This is most likely due to the randomness that value and policy iteration begin with. Since they begin with random values that don't make sense, the initial iterations will have an extremely high number of steps to goal. But there is also a chance that the initial random values could be close to optimal, and value and policy iteration would therefore wouldn't have as costly of a start. And since these first steps are magnitudes of scale more than the future values, they make up a large bulk of the runtime of the algorithm. Thus, the performance of value and policy iteration is dependent on the initially randomly assigned utility values or policy.

Now, we look at time and reward:



We see that Q learning takes more time than value and policy iteration. This is probably due to the fact that it is actually learning through stochastic processes in handling whether it should explore or exploit at each step. The total reward for Q-learning starts off much lower, but ultimately converges. It shows some fluctuation that value and policy iteration don't at some later iterations, but it is to be expected due to the stochastic nature at which Q-learning iterates. Now, we look at the policies:

Q-learning this time really accentuates the severity of the negative reward states, since the first half of the lower right path is purple and have negative Q-values, and it seems to just have a policy of getting out of that situation. On the other hand, value and policy iteration seem less cognizant this time of the negative reward states. Even though the policy matches that of Q-learning, it has much higher utility values and the states are still blue. However, since the policies are the same, all of them seem to find an optimal solution policy of taking the longer, winding path to the goal.

4. Conclusion

What we see from the data that we have gathered is that all three reinforcement learning algorithms perform well, but have their strengths and weaknesses. As we predicted earlier, value iteration seems to get too bogged down in the details, and thus policy iteration performs better after more iterations. Other than however, those two algorithms are essentially the same. The difference between them and Q-learning is the amount of information that is required to be fed into the algorithm. All the rewards need to be known beforehand for value and policy iteration, while Q-learning needs to only know the reward values of the states that it happens to traverse. Thus, in situations in which there is less information available, such as an unknown domain that is being explored by the algorithm itself, Q-learning is much more effective and can adapt. If the new domain is small enough, once the Q-learner has explored all the states and a transition function is extrapolated, we can use that information to use value or policy iterations. However, in the case that the new domain is very large or essentially infinite (continuous problems that cannot be abstracted or simplified), Q-learning is much more useful. In continuation of the experiments in this paper, it would be interesting to look at a GridWorld MDP of similar size to the hard MDP, but have the goal state much closer to the start state. That example would exemplify the usefulness of Q-learning in a MDP where there are many "useless" states. However, if we have domain knowledge over the environment, we could just simply not include those "useless" states just as easily.