Reinforcement Learning (April 2017)

William Z. Ma, *Student, Georgia Institute of Technology*

Abstract – This document presents two Markov decision processes (MDPs), one with a comparatively small number of states, and the other with a comparatively large number of states. It then provides detailed analyses comparing the performances of three reinforcement learning algorithms on each of these processes, namely value iteration, policy iteration, and Q-learning. Q-learning will have its hyperparameters tuned before evaluating its final performance relative to the other two algorithms. These performances will be compared through a variety of metrics, namely runtime, iterations, and how close the algorithms were able to get to the optimal policy.

1 Introduction and Overview

1.1 Overview of Markov Decision Processes (MDPs)

Before we can analyze solutions to Markov decision processes, it is necessary to define what exactly Markov decision processes (MDPs) are. Markov decision processes represent a convenient mathematical framework for modeling real world processes and decision making, particularly in situations where the outcomes of decisions are subject to some degree of randomness, and at the same time partially under the control of a decision maker.

What this amounts to is that we can represent many sequential decision making problems as the following:

- **S**, a finite set of states, from which actions can be performed
- **A**, a finite set of actions, representing the actions that can be performed over the set of states
- **T(*s,a,s'*) = Pr(*s'* | *s, a*)**, a transition model representing the probability of an action a leading to a state *s'* from a given state *s*; this model encapsulates the underlying stochastic process (i.e. randomness)
- **R(*s*)**, an immediate reward function for the immediate value of being in a particular state
- **π(*s*)**, the policy of the decision maker, specifying which action to take in a particular state

Note that the utility of an MDP in solving a decision making problem is dependent on the Markov property, in this case more aptly named the Markov assumption, which states that the conditional probability of future states is dependent only on the present state of the system, and not on any previous states.

In the context of this model, we are looking for π^*, the optimal policy which provides maximum total reward.

Further assumptions about our MDPs include the assumption of stationarity, namely that of infinite horizons; that is to say that the only parameter to our policy function is the given state; we are not limited by other constraints such as time, number of steps, etc, which in real life processes may change our choice of action in a given state.

We also assume the utility of sequences, that the utility of sequences of states being compared with other sequences of states is logical and consistent in some way; this is done through discounting, allowing us to extract finite utilities from potentially infinite sequences of states.

To put it succinctly, we can effectively assume that all our MDPs have stationary preferences.

1.2 Value Iteration

It would be useless to evaluate the performance of various MDP solution algorithms or reinforcement learning algorithms without first taking some time to define exactly what those algorithms are.

The concept behind value iteration is at its most basic a greedy one: if we could assign a value or utility to every state that represented the usefulness of that state to the solution as a whole, then determining the optimal policy for a given process is simple: given a state, simply perform the action that takes us to a state with the maximum utility of our given options. The problem with this idea is that given a typical MDP problem, this overall value or utility for given states is unknown; we only know the immediate reward for given states, R(s). Furthermore, this reward function and a utility function for an MDP may very well give very different results. A state with low reward but high utility is very much possible. That is to say, value iteration aims to transform the actual derivation of the optimal policy into what is effectively a greedy path-finding algorithm. However, the way it does this is anything but greedy, given the potential gaps between immediate reward and utility.

But how do we find this utility? In fact the utility of a state can be defined as the immediate reward for that state, plus the expected discounted reward if the decision maker made only optimal decisions from that state onward. Thus the utility of a particular state is typically dependent on some number of its neighbors. To do this, we perform the following steps, which describe the general algorithm:

- Assign an arbitrary utility to every state in **S**
- For each state *s* in **S**, calculate a new utility for *s* based off the utility of its neighbors
- Update the utility **U** for each state *s* using the below Bellman (1957) equation:

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

- Repeat the previous two steps until convergence

Note that this notion of convergence is really a tolerance, i.e. the algorithm runs until utility changes become smaller than some specified threshold.

1.3 Policy Iteration

Value iteration is rather useful but suffers from two principal weaknesses, namely that it can be incredibly slow to converge for some scenarios, despite the fact that the underlying policy it is computing remains the same, and that it only indirectly finds the optimal policy. That is to say, value iteration is finding the long-term value or utility of each state, and then using that to find the optimal policy. We stated that this may in fact be very slow in certain situations, so why not find the optimal policy directly, instead of doing a value-function based algorithm?

In fact with just a simple modification to value iteration, we can do exactly that. With the below algorithm, we iterate over policies instead of over states, by selecting a random policy, computing the utility of each state, and then selecting an optimal policy from the already computed policies:

- Generate an initial random policy, being a permutation of actions for all states in the MDP
- Loop until no action in our policy changes:
    - Compute the utility for each state in **S** relative to the current policy
    - Update the utilities for each state
    - Select the new optimal actions for each state, changing the current policy in the process

We see that policy iteration is making use of similar Bellman equations as value iteration, though this time around the systems of equations of concern are linear instead of nonlinear as in value iteration.

1.4 Q-learning

Both value iteration and policy iteration are incredibly useful for determining the optimal policy for a given MDP, but they do so by making the strong assumption that the decision maker has a substantial amount of knowledge about the underlying stochastic process, namely that the transition model is known along with the rewards for all possible states. In fact, in many situations, we may not have access to all this information about a process.

In fact, in some sense, neither value iteration nor policy iteration are true reinforcement learning algorithms, because although they compute the solution to a given MDP, all the information necessary to do so is already available: they do not really learn anything over the runtime of the algorithm, since the optimal policy can be effectively computed almost deterministically from the information available. Thus value iteration and policy iteration are more akin to planning algorithms (with some included randomness), similar to algorithms such as Dijkstra's algorithm for shortest path in a graph (although Dijkstra's need not deal with such randomness).

In this sense, we give the following analogy for how many reinforcement learning algorithms would appear to work: assume we have an animal who, after receiving some stimulus, has the option to press a button and receive a reward. Over time, the animal's response to this stimulus would be strengthened, and they would associate the stimulus with the action and the reward.

Q-learning, then, is a family of model-free reinforcement learning algorithms that given knowledge of the possible states and the actions possible at each state, attempts to learn the optimal policy from the incomplete information available. It does this first by assigning to each state an estimated value, known as a Q-value. When a state is visited, we receive a reward for having visited that state, which is then used to update the Q-value for that state. This may happen many times, since rewards may be stochastic (actions have a probability of getting us to a state). We can express our Q-value as follows (another Bellman equation):

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

This Q-value, is in effect, the value for arriving in state s, leaving via the action a, and then proceeding optimally.

Q-learning updates the estimates of Q-values for each state until hopefully reaching some convergence, namely the point when the estimate is equivalent to the actual result from the Bellman equation (or as close as possible).

However, how does Q-learning select which actions to take at each state? We could take the action to the state with the highest known utility as we did with value and policy iteration, but the concern here is that this doesn't imply any sort of learning. How do we actually learn and update our Q-values? We must take actions to states where we may not know the reward, in order to get enough information to hopefully converge to a solution. This leads to the infamous exploration-exploitation dilemma in reinforcement learning, which asks how much time the algorithm should spend learning, and how much it should spend trying to apply what it has learned.

The approach we take to tackle this problem is the same as used in the library we are using for our analysis, namely the method of ϵ-Greedy exploration (assuming very small ϵ). We will also take a look at other exploration strategies, and determine the best exploration strategy for the given MDP.

1.5 Implementation Specifics/Attribution

This paper makes use of Juan J. San Emeterio's extension of the Brown-UMBC Reinforcement Learning and Planning (BURLAP) Java library, in particular its implementation of the GridWorld Markov decision processes, with customized worlds, reward functions, and states. This library is provided for free under the MIT license, with San Emeterio's extension itself being available for free under the Lesser GNU Public License, a copyleft license provided by the Free Software Foundation. All algorithms and visualizations, besides the graphs (generated in Microsoft Excel), are provided through software provided by one of these aforementioned libraries.

1.6 GridWorlds

Now that we have given the necessary descriptions for what MDPs are and the algorithms that we will use to solve them, we must now describe the actual MDPs themselves that we are trying to solve. GridWorlds are simple rectangular spaces divided into squares of uniform size. Each of these squares can be either an agent or the decision maker (a grey circle), walls or obstacles (black squares), or a goal or terminal/absorbing state (blue squares). They are frequently used as MDP examples in reinforcement learning, typically for learning purposes.

The only invariant between the GridWorlds used in our analyses is in the probability of success of taking actions (which is necessary to create a stochastic process for our MDP); i.e. if our decision maker opts to take an action, there is an 80% chance of that action actually occurring, and a 20% chance of moving in any other direction (note that this 20% chance is split uniformly between the other three possible actions for any given state).

Moreover, each GridWorld will have a reward function specifying the cost/reward of performing various actions. This should complete all the requirements for an MDP to be used in our three algorithms.

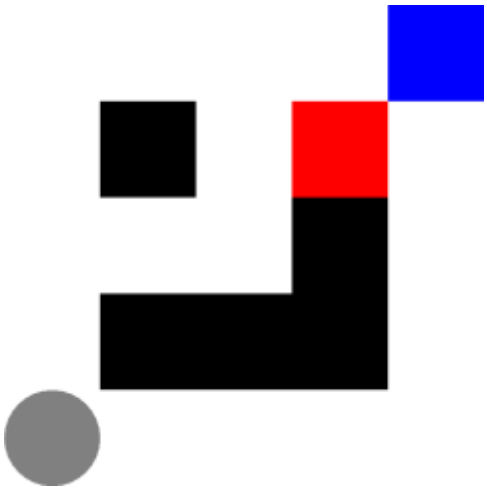
Finally, there may be some concern about the usefulness and applicability of GridWorlds, in particular because they seem oversimplified or perhaps too abstract. However, simple and abstract problems are in fact incredibly useful, because their simplicity and abstract nature allows them to be useful in a broad range of applications. For example, this sort of problem will be highly relevant to almost any sort of robotics planning algorithm; a robot moving autonomously will need to be able to avoid obstacles and reach goals without paths being explicitly laid out by a programmer. Such reasoning is necessary for solving problems such as the kidnapped robot problem, wherein a robot must determine information about its environment while simultaneously moving in that environment. This very unique problem requires a highly robust and flexible model, which MDPs can satisfy.

The argument can then be posed that GridWorlds yet remain too simple for problems such as these. However, GridWorlds in their simplicity are highly extendable by the same token, solving this problem just as handily.

2 GridWorld (easy)

2.1 Introduction

To get a good idea of the potential performance differences between value iteration, policy iteration, and Q-learning, we will first apply them to an easy GridWorld, before eventually applying them to a harder one and seeing how the algorithms behave in that case. Find below the GridWorld in question (with 15 states):



Our easy GridWorld is grounded in a real world example; you are driving along a road, rushing to get to your next meeting. The closer you get to your destination, the more urgent it is that you finish as soon as possible. As a

result, I have included a linearly decreasing reward function that decreases the reward for each state closer to the goal. This should create a similar sense of urgency to the algorithms that would hopefully capture some of the real-life urgency of the situation. There are then three possible routes to take: There are the two routes skirting around the edge of the GridWorld, and then the route between the obstacles in the center. You may note, however, that there seems to be a red state. What does this red state represent? In fact it is effectively a toll booth, going through which would cause you to miss your meeting entirely. Thus this red state, once landed on, gives a massive negative reward, larger than any of the other negative rewards in the GridWorld. In fact, landing on this toll booth gives -10000 reward, while getting to the meeting at the blue states gives 1000. Getting to this meeting is urgent!

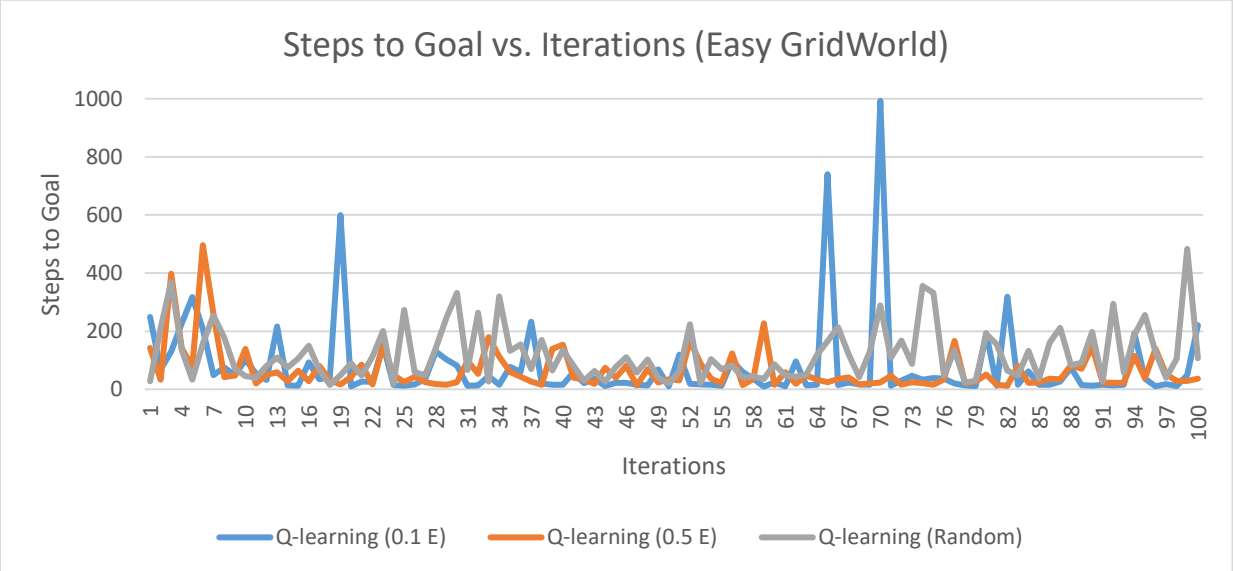
When first creating this GridWorld, I was very much curious about how Q-learning would behave, given the extremely unoptimal state and how close it was to the finishing state.

2.2 Q-learning Exploration Strategy

Before we can evaluate the performance of each of the algorithms, we must first tune the performance of Q-learning, by selecting an exploration strategy and tuning the hyperparameters of that exploration strategy. For this paper, we made use of three exploration strategies made available through BURLAP: epsilon-greedy exploration, Boltzmann exploration, or simply random exploration. These strategies of action selection are vastly different, with epsilon-greedy exploration having decreasing randomness over time (decaying epsilon), Boltzmann exploration making use of the Boltzmann probability distribution, and random exploration making use of Java's built in Random class, which means its choices are made using a pseudorandom linear congruential generator.

As for potential hyperparameters, Epsilon-greedy exploration has a hyperparameter epsilon, Boltzmann exploration a hyperparameter temperature, while random exploration has no such hyperparameters, having been almost entirely based on Java's built in Random class. For each exploration strategy, we picked a few values for their hyperparameters and then compared the performance of each hyperparameter and strategy (performance being measured in terms of the number of steps required to reach the goal, since the optimal is 8).

We can immediately discount the Boltzmann exploration strategy, seeing as how the required number of steps is consistently inconsistent, with fluctuations orders of magnitude above the other strategies. We thus compare only the performances of the epsilon-greedy and random strategies:

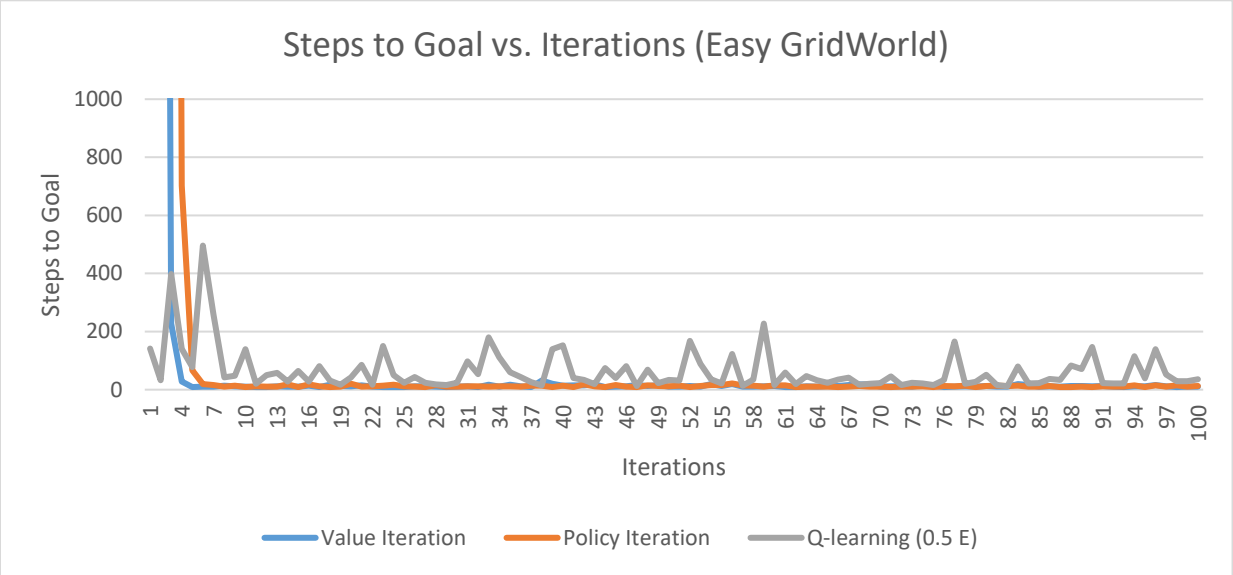


In this instance, we'd like to see which exploration strategy (and hyperparameter) resulted in the best and fastest convergence (fewest steps taken to reach the goal). For this reasoning, we have picked epsilon-greedy exploration

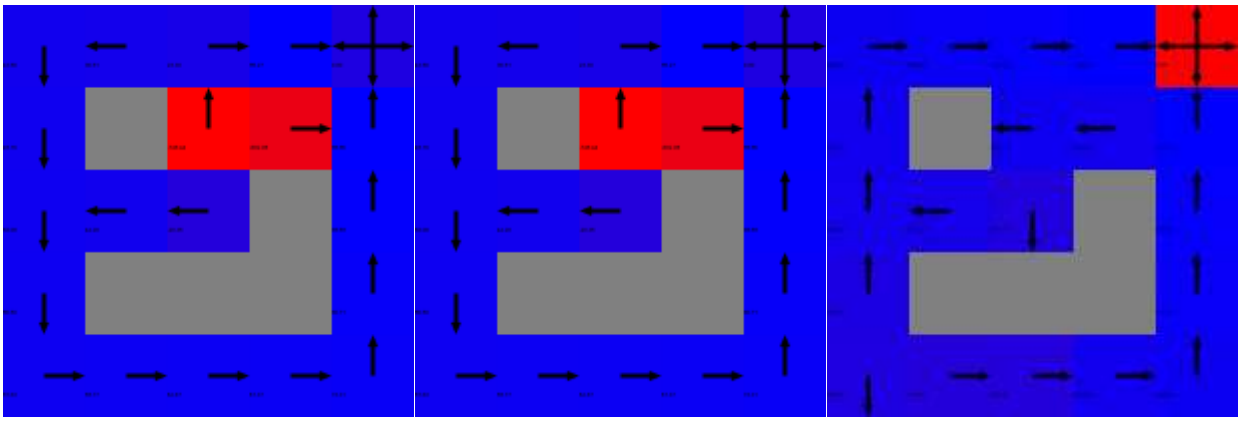
as the best strategy, but with an epsilon value of 0.5. This strategy gives consistently better results with less variation than either the smaller epsilon value or the random exploration strategy.

2.3 Performance Evaluation

Now that we are aware of the best exploration strategy and hyperparameters for our Q-learning algorithm, we can now compare its relative performance to that of value iteration and policy iteration. We first begin by taking a look at the steps to goal vs. iterations graph again, this time with the other algorithms included:



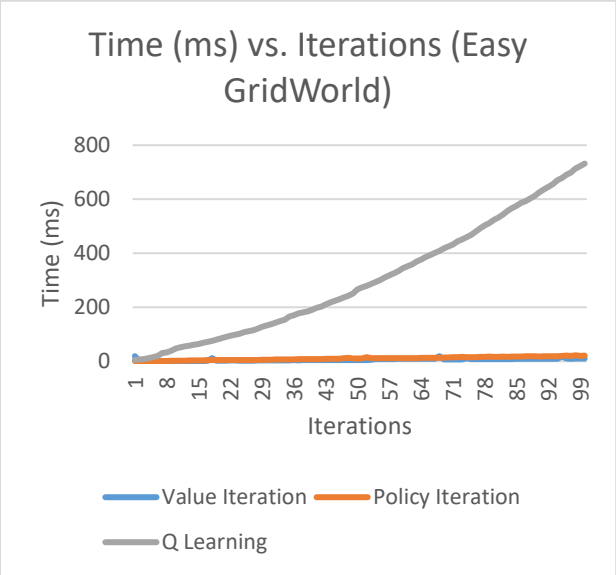
Perhaps unsurprisingly, Q-learning does well enough but is handily outperformed by value iteration and by policy iteration. This is a sensible result given that value iteration and policy iteration both have perfect information about their environment, while Q-learning must make sense of its environment experientially. We can see the effect this experiential handicap had on the policy determination below:

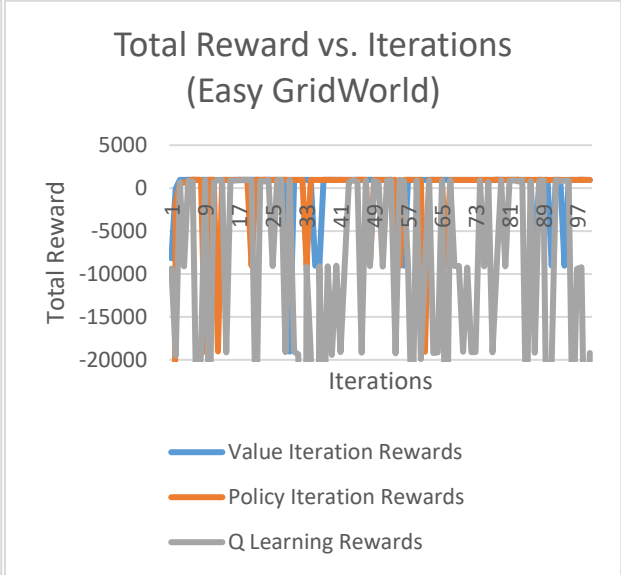


Value iteration (left), policy iteration (center), and Q-learning (right).

We can see immediately that policy iteration and value iteration both resulted in the correct optimal policy, while Q-learning seems to have gotten rather confused; i.e. it seems to have noticed the toll booth, and thus in its optimal policy opted to move away from it when in those states, but doesn't seem to have recognized the actual negative utility that those states have. So in this sense, Q-learning ends up with almost the same optimal policy but misses out on the finer details, which is sensible given it does not have perfect information or domain knowledge.

But unfortunately, we can't select Q-learning as the best algorithm, because it simply doesn't have the optimal policy, although getting somewhat close. Since policy iteration and value iteration resulted in the same optimal policy, we must compare their performance in some other way. What about time? What about total reward?





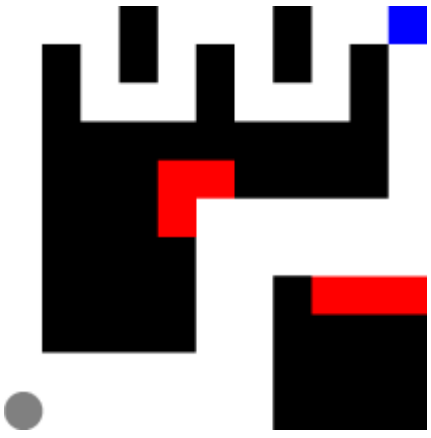
On the left we see the cumulative amount of time it takes for each algorithm to run and complete its 100 iterations. We see that once again Q-learning is worse in this regard, again a result of its handicap, since value iteration and policy iteration do not suffer from the exploration-exploitation dilemma, being entirely exploitation. As a result, Q-learning should naturally take substantially more time. We have already discounted Q-learning, but time comparison doesn't seem to yield any insights as to which algorithm to use for this particular problem.

However, when looking at the total reward, the answer suddenly becomes much clearer. After 100 iterations, policy iteration shows a much more clear convergence than either of the other solutions. Value iteration still fluctuates notably by 100 iterations, and Q-learning clearly never converges. For this reason, for our easy GridWorld MDP, we pick policy iteration as the best way to solve it, since we do have domain information.

3 GridWorld (hard)

3.1 Introduction

In order to make an MDP that is a little more difficult for our algorithms, we will now test them on a much more complicated GridWorld, with 63 states instead of 15. Find below a picture of the GridWorld in question:



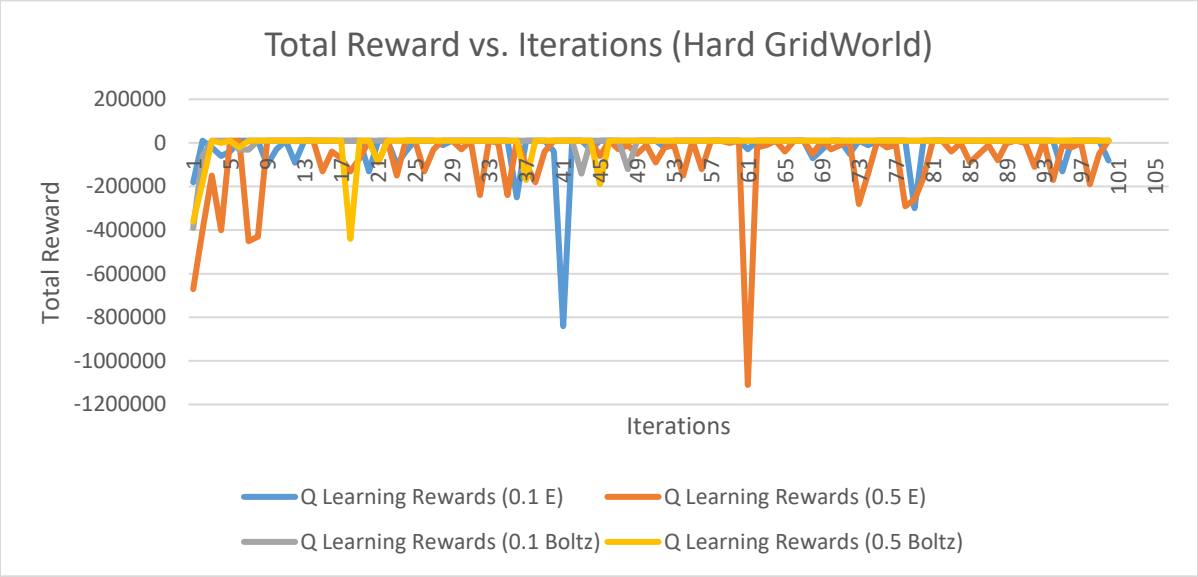
We can see above a much more complicated world than our first example, with much more “toll booths” as well. There are more opportunities in this instance to end up in one of these highly negative states than in the other; there must be more of a balancing of potential cost in taking one path to the goal vs. the longer winding path that wraps around the side of the GridWorld.

To frame this problem in a real-world example, consider the situation where we are attempting to get to a meeting once again. You can take the arguably shorter highway, but unfortunately I-75 has experienced a big accident and now there is a real risk that while traversing you will get caught in traffic. You can take longer winding local roads to get to the meeting, but you might get there slower than you would’ve if you had taken the risk and had it come out in your favor. What is the optimal policy to take when trying to get to this meeting?

This GridWorld again makes use of the same linearly decreasing negative reward function as the previous example, attempting to encode the increasing urgency with which we must get to the meeting. In this example, it will again be interesting to see the effect that this MDP will have on the Q-learning algorithm, as a result of more toll booths.

3.2 Q-learning Exploration Strategy

Once more, we must evaluate which exploration strategy is best for Q-learning with this problem before we can compare the performance of this optimized Q-learning algorithm to policy and value iteration. This time around, instead of considering steps, we will take a look at the total reward of a given exploration strategy vs. the number of iterations (this makes more sense given the longer paths and larger policies overall):



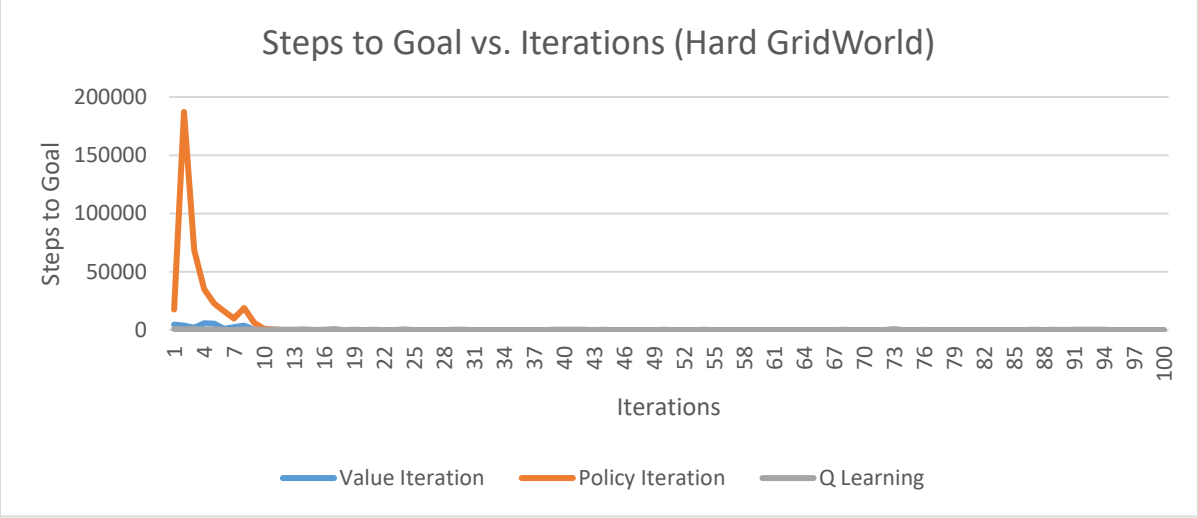
One will immediately note that once again we seem to have excluded an exploration strategy. In this instance, instead of excluding the Boltzmann exploration strategy, we have excluded the random exploration strategy. The reason for this is that random exploration produced an order of magnitude lower reward than any other method. This is very sensible given how many states there are. Whereas in our easy GridWorld, there were relatively few states and thus random exploration could still perform well, in this instance the number of states meant that there was substantially more room for error and bad decision making by that exploration strategy.

Now how do we select our exploration strategy? We see that both of our epsilon-greedy exploration strategies resulted in much more unsteady convergence and fluctuation. For this reason, we pick a more consistently performant solution, seemingly one of the Boltzmann exploration strategies. However, we note that Boltzmann with a temperature hyperparameter of 0.1 seems to have much smaller fluctuations and faster convergence than

Boltzmann exploration with a temperature value of 0.5. For this reason, we therefore pick Boltzmann exploration with a temperature hyperparameter value of 0.1.

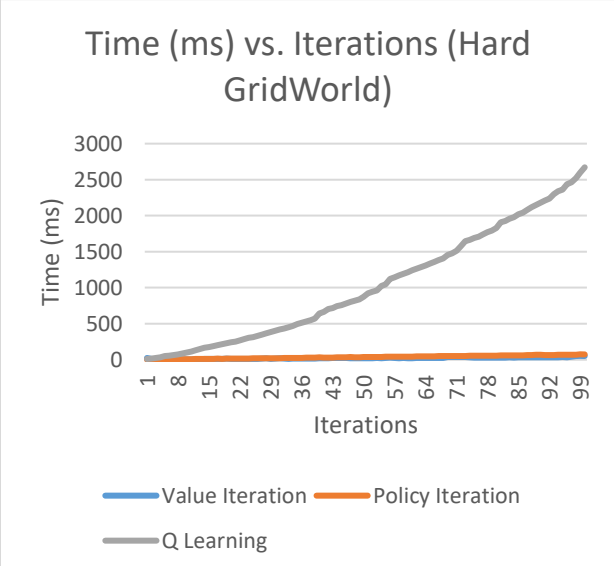
3.3 Performance Evaluation

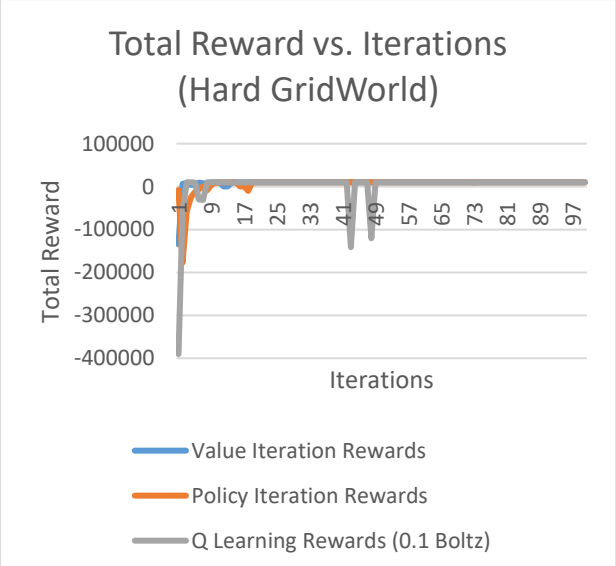
We can now compare the relative performance of the algorithms using Q-learning with our optimized exploration strategy. We begin first by looking at the number of steps required to reach the goal state per iteration:



We see here that policy iteration seems to perform bizarrely badly by this metric, while value iteration and Q-learning do surprisingly well, relatively, several orders of magnitude better, in fact. Why might this be the case? It may be a result of the fact that policy iteration begins with a random permutation of actions. The large number of highly negative states and their massive impact may heavily affect early iterations of these random permutations which may have actions to these states. As a result it may lead to mass confusion policy-wise for a policy iteration algorithm. For this reason we are not discounting it immediately but looking at its performance in other metrics, since we care only about the optimal policy, not necessarily how it got there.

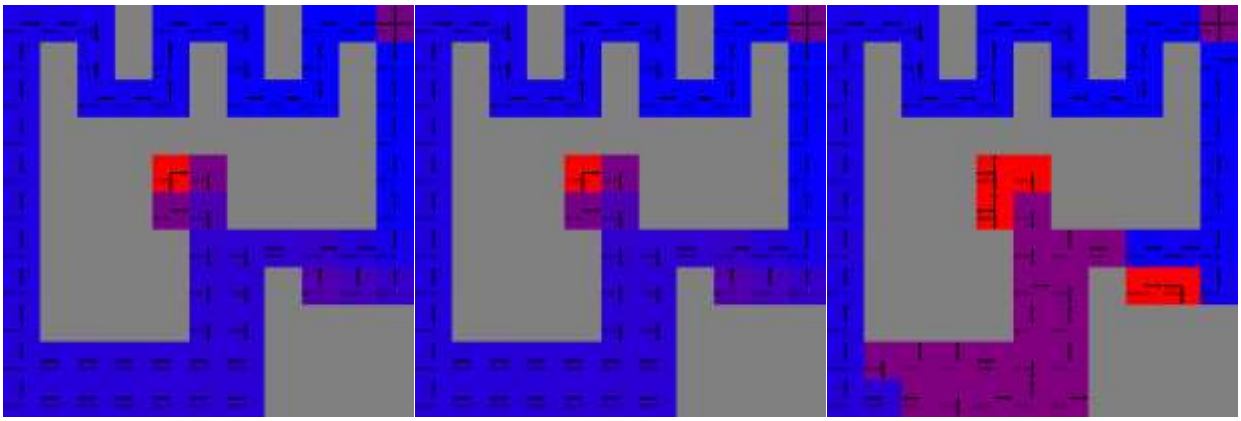
We then take a look at the time required for each algorithm as well as the total reward achieved by each algorithm. This will allow a more diametric view into what makes one better than another:





We see again, unsurprisingly, that Q-learning seems to take much longer than the other algorithms, again a result of being required to handle the exploration-exploitation dilemma, and having to learn instead of simply computing utilities. The total reward given by Q-learning though seems to converge much quicker, and in fact, Q-learning seems to get significantly closer to the optimal policy with this harder GridWorld than with our easier one. This seems to be a result of the actual paths available to the goal with this harder GridWorld. In our easier GridWorld, no matter which path you take, there is some risk of falling into the toll booth or negative zone, and thereby Q-learning may show highly unpredictable results there, but all policies will attempt to minimize the risk. However, for our harder GridWorld, our long winding path is actually able to entirely avoid this toll booth, and thus Q-learning performs much better overall, at near parity with value iteration or policy iteration.

This parity is very well demonstrated in the actual generated policies, as found below:



Value iteration (left), policy iteration (center), and Q-learning (right).

We can see that while Q-learning retains some of the confusion that happened in our easy GridWorld, it encapsulated much of the directional information of the optimal policy given by value iteration and policy iteration, trying to avoid the very risky toll booth areas. Furthermore, neither these pictures of the optimal policy computed nor the comparisons between the algorithms seem to have shown much in the way of differences between value and policy iteration, so given the domain information of the GridWorld, either is acceptable.

4 Conclusion

So we know that value iteration and policy iteration both seem to perform admirably when given domain information, and that Q-learning takes a long time, because it must both explore and exploit what it learns, a dilemma not faced in equal measure by either value iteration or policy iteration. So what is there to glean from our analysis? Q-learning takes exponential time compared to either value iteration and policy iteration, so in any situation where we can be reasonably certain about domain information, we should pick either of those methods.

It very much is the case that these methods cannot be used in the same situations, even though they use much of the same mathematics. Value iteration and policy iteration are good for their respective domains, but Q-learning is very much an actual learning algorithm, good because we can deploy it in situations with much less knowledge than would be suitable for value or policy iteration.

Therefore, in real world situations, Q-learning would be excellent because it is that much more adaptable than either value iteration or policy iteration. However, samples of the domain information captured from multiple instances of Q-learners in these real-world situations may allow us to create the transition model and reward functions for which an optimal policy may be easily computed using value or policy iteration.

Therefore, we can use Q-learning's adaptability to lead in to the use of value or policy iteration in MDPs.

References

Bellman, R (1957). A Markovian Decision Process. Indiana Univ. Math. J. 6 No. 4. 679-684.

Kaelbling, L.P., Littman, M.L., & Moore, A.W. (1996). Reinforcement Learning: A Survey. Journal of Artificial Intelligence Research, (4), 1-49. Retrieved April 21, 2017, from http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a.pdf

Russel, I., & Markov, Z (2009). Value Iteration, Policy Iteration, and Q-Learning. MLeXAI May 2009 Workshop. Retrieved April 21, 2017, from http://uhaweb.hartford.edu/compsci/ccli/projects/QLearning.pdf