CS 7/4641 Project 4 Analysis – Markov Decision Processes

Eric Wang - 4/14/2019

Introduction

Decision making is a difficult process that can be modeled using Markov Decision Processes, which provide a base framework that can then be solved using various algorithms to find the optimal decisions. Reinforcement learning takes these processes and attempts to solve them without full knowledge of the world. This paper aims to highlight the differences between performance of different algorithms depending on the scale of the decision problem, and the degree of known information.

Defining Markov Decision Processes (MDPs)

An MDP represents a world where outcomes are a combination of input from a decision maker and a predetermined function. It is robust in its possible representations and requires just four components. The first is a (finite) **set of states S** that represents all of the potential states that can exist in a particular world. The second is a (finite) **set of actions A** that represents all the potential actions that can be taken at any particular state. Some actions are restricted to certain states, depending on the representation. The third is a **transition model T** that represents how actions will affect state changes. This model completely changes how much control the decision maker will have over the state changes, and effectively how that particular world will work. It can be completely deterministic, where the decision maker will 100% land in the state that it has chosen to move to, or it can be completely uniformly stochastic, where the decision maker has no say. Usually, in representing a real world process, there will be a specific stochastic transition model that can capture the decision maker's choice most of the time, but provide a different action with some smaller chance. The final component is a **reward function R** that provide the immediate expected reward at each state in the set of states S. With these four components, an MDP is robust enough to represent a wide range of problems.

A strong assumption of an MDP is the Markov assumption, which states that only the most recent state matters in determining the next state. Therefore, a solution to the MDP will not have to keep track and consider all previously visited states, but rather simply just the current state, to determine the most optimal path. A solution to an MDP is a **policy π** which is simply a mapping of states to actions. The optimal policy is the policy that provides the actions that lead to the highest long term expected reward. This "long term expected reward" is called **utility U**.

More assumptions must be made in order to effectively solve MDPs. We assume that the solution paths are effectively unbounded in size, and unless there is a sink state, there is an infinite horizon. We also assume that we are working with stationary transition models, meaning that the way the particular world works mapping actions to next states will not change as more actions are taken. With the possibility of infinite paths, we also need to ensure that utilities are comparable in order to find the optimal policy. Therefore, we also assume that rewards are discounted through timesteps when counted towards utility, with a provided discount rate. This prevents utilities from being infinite.

Ultimately, an MDP as described here represents the world and all that can be done in the world, along with the good and bad outcomes. It provides all the necessary structure to set up algorithms that can find the best decisions to make in every state, leading to an optimal path.

Value Iteration

Given that we know all four components, finding the optimal policy is just a matter of iterating through known values. Value iteration finds the optimal policy by returning the action that brings the highest utility at each state. Since utility is not known but reward is, the algorithm iteratively builds up knowledge of utility in a state, starting with randomized initialization, based on the reward of the neighboring states. The utility is the current immediate reward plus the future potential reward given optimal actions, discounted at a specified rate. This iterative process is repeated until convergence.

The purpose of value iteration is to not take the rewards at face value. Sure, a negative immediate reward seems to be worse than a positive immediate reward, but if the state with the negative reward puts you in a better position to get higher positive reward in the future, then the utility of that state should be higher. Value iteration, in iteratively improving its idea of utility for each state, aims to capture this insight overtime by bleeding information through the states. Although the utilities are initialized randomly, the states will get closer to their true utilities over time as more information is applied from the rewards from neighboring states.

Ultimately, this utility that we find from value iteration is only used to find the optimal policy, which is just a mapping of states to actions. Since this is our final output that we care about, the actual exact value of utility does not actually matter, as long as it is close enough to give us the same optimal action. This is why random initialization of utilities is not a big deal. However, a drawback of value iteration is that it can be slow to converge and has to take that extra step in converting utilities to a policy.

Policy Iteration

Policy iteration skips the middle step of calculating the optimal policy from the max utility, and instead iteratively finds the best policy directly. Utilities are still calculated per iteration to determine which policies are better, but the changes are the actions themselves, which reflect the changes in utilities. Instead of value iteration, which looks at individual states' utilities per iteration, policy iteration takes the overall utility given the policy, which is initialized randomly, and compares that to determine changes in the policy. Ultimately, the policy iteratively gets better until it converges to the optimal policy.

Policy iteration is more easily solvable, since the update equation is not concerned with finding the max, and therefore is a linear equation. Moreover, it will always converge, since with each iteration, the performance is always improving. Therefore, with a finite number of policies (since there are a finite number of states and a finite number of actions), policy iteration will at worst exhaust every single combination and eventually come to an endpoint.

Reinforcement Learning and Q-Learning

So far, value iteration and policy iteration assume that we know all four components: states, actions, transition model, and reward function. Therefore, the algorithms are essentially just iteratively finding the optimal policy and not technically learning. However, it is not realistic to assume that we will always know everything about the world and problem we are trying to model. It is reasonable to approximate the states and possible actions, but many times we do not know exactly how the world will react to our actions, nor will we know the consequences of our actions and how good and bad they are. Therefore, often times we will not have the transition model or the reward function. In these cases, we need to explore the world to learn more about how it works, and that means testing out actions to change states and gain rewards. Reinforcement learning uses these transition samples to iteratively improve our

value function and “reinforce” our knowledge to the point where we can come up with the optimal policy.

The reinforcement learning family of algorithms used is Q-Learning, which initializes each state with a “Q-value” representing utility of taking an action leaving that state and then proceeding optimally. Then, samples are taken by selecting certain actions at each state and using the immediate reward gained to update the Q-value. This is repeated and discounted by a specified learning rate until the Q-values hopefully converge. This update is similar to the iterative update from before, but the main difference lies in the implementation. The action taken at each state is different from just choosing max utility or something of the like. If that was the case, then the algorithm would never really learn and just trust the randomized initialization and first rewards. Therefore, there are different ways of choosing the actions, whether its randomized or following some distribution.

The Q-learning algorithm used in analysis here follows an epsilon greedy exploration method. A parameter epsilon is specified between 0 and 1. The algorithm will choose the optimal action based on max Q-value by default, but with probability epsilon, will instead choose a random action. This can then be tuned to balance the tradeoff between exploration and exploitation. This parameter can be constant, or similarly to simulated annealing, this parameter can also be decaying over time. The implementation used here is a constant epsilon value of 0.1.

Worlds

Two worlds are modeled by MDPs and explored by the algorithms. Both are GridWorlds, which take the shape of a rectangular grid. The first is a small 5x5 grid, while the second is a larger 11x11 grid. The grey circle is the agent, the white squares are spaces, the black squares are walls, and the blue square is the terminal state and eventual goal.

EasyGridWorld

Figure 1 represents the first problem.

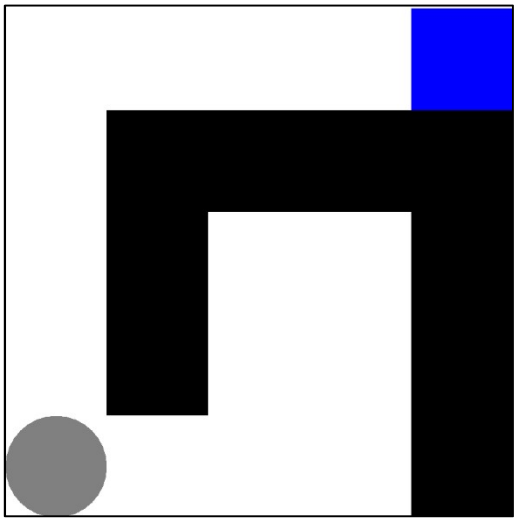

Figure 1

The states are the white spaces, including the one currently occupied by the agent, and the blue terminal state. In total, there are 16 states. The actions are up, down, left, and right. The transition model captures the success rate of the actions. For any action, there is an 80% chance that the agent

will attempt to move in the direction specified. However, there is a 20% chance that the agent will instead choose another direction to move in, with probability spread evenly across the other three directions. For the actual direction attempted, if there is a state there, then the agent will move to that state. If there is a wall, then the agent will stay in the state that it currently is in. The reward function specifies that every normal state gives immediate reward of -1, and the goal gives reward of 100.

This gridworld represents a very simple world but highlights a very important and commonly seen decision process. The initial state is the position shown, and initially there are two paths to choose from. The one on the right is wider and more open but leads to nothing. The one on the left is a narrow path and leads to the goal. This split is representative of a situation where the user has two ways to approach a problem or decision – either take some time to look around or follow a straightforward and narrow path. The problem is set up so that the straightforward path is the better decision leading to the goal, so this MDP represents any situation where that is the case. Perhaps it is a time-sensitive project with low standards, so the best course of action is just to quickly choose a plan and stick with it just to get the project done. Or maybe it's a group of friends trying to decide dinner plans, and the more they spend discussing the hungrier everyone gets and it's better to just make one decision and stick with it.

HardGridWorld

Figure 2 below represents the second problem.

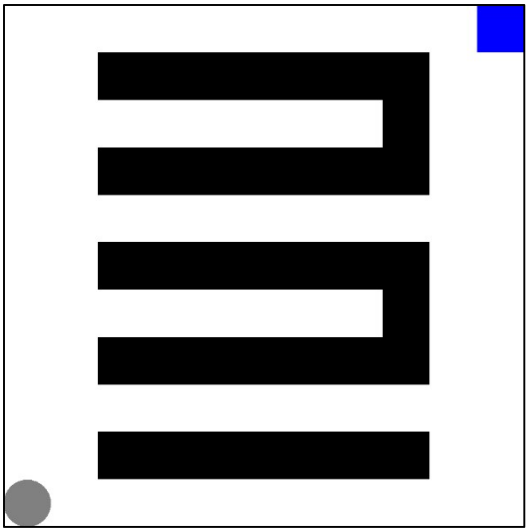

Figure 2

This gridworld represents a more complicated world with 83 states, and conceptually models another commonly seen decision process. The actions, transition model, and rewards function are all fundamentally the same layout as the first. The initial state is the position shown, and the world is split into two sides. The left side is the starting point, and the right side is the side the agent wants to be in. The agent also wants to move up, since the goal is at the top. As the agent moves up along the left side, there are multiple paths to travel right through, but some of them lead to dead ends. This means the agent must learn which paths are dead ends and which paths lead to the right side. The problem is set up so that the agent must take a chance, since from the left side all paths look the same. This is like a situation where you know roughly the general direction to go, but you just don't know the right path. Maybe you are working on a project with vague guidelines, and there are multiple ways to approach it with some leading to failure. Or maybe you are driving to the restaurant with your friends, but you forgot which intersection to turn on and some don't lead to the right place.

Solutions

EasyGridWorld

Figure 3 below shows the optimal policies for the first problem after running Value Iteration, Policy Iteration, and Q-Learning, respectively, on the MDP. The color is a scale from more blue representing ideal states with higher utility/Q-value, and red representing lower utility/Q-value. Purplish blocks are somewhere in between. The goal state at the top right does not count on this scale, since there are no actions which can be taken from it.

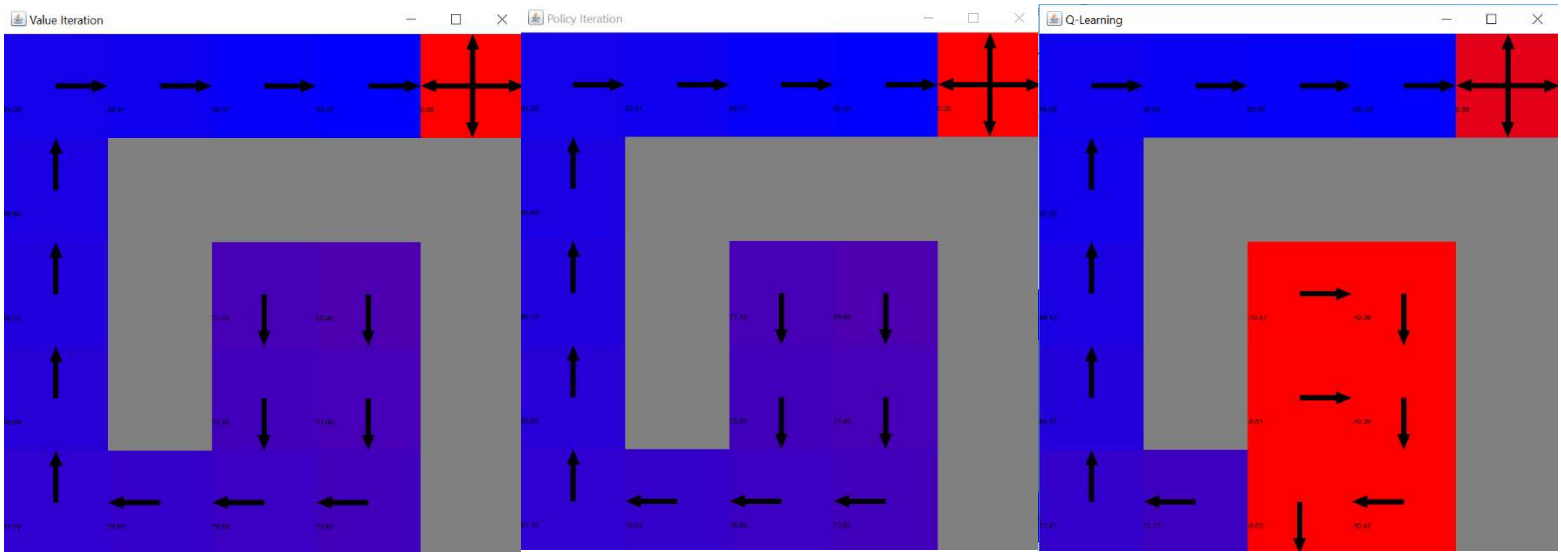

Figure 3

As shown, value iteration and policy iteration both find the clearly optimal policy. The left path takes a straightforward path to the goal, with states getting higher utility the closer they are to the goal. The right path is the wrong way, and therefore the policy takes the quickest way back to the right path. The farther away the state is from the goal, the lower utility it has. This makes a lot of sense, especially considering that both algorithms had full access to the world's information. Knowing the true reward and transition models, both could pick the maximum utility at any point, converging onto the optimal value and picking the optimal policy.

Q-learning, on the other hand, performed a little more poorly, although not terribly so, given that this is an easy problem. The left path looks very similar in utility to the left path in the value and policy iteration solutions and is the same in policy. The right path differs quite a bit. Some states give non-optimal actions, taking more steps than needed. Especially the state that gives down as the optimal action, that is just clearly wrong. The entire right path is also red, meaning it has a very low Q-value. At the very least, Q-learning is also able to recognize the correct path and the wrong path.

Q-learning performing sub-optimally makes sense, since ultimately it does not know the transition model or the reward function. It only knows what it can sense in the moment. Since every state has a reward of -1, Q-learning does not know how well it is doing until it hits the goal state and receives reward of 100. It can only hit the goal state by chance the first time. From there, it will learn that the path towards the goal is better but will still explore other options due to epsilon greedy. Since it explores other options, it learns that the right side is really bad and doesn't lead to anything. However, it

is still not able to determine the optimal policy simply because it may not have that many chances to try out the wrong path. Moreover, the transition model is based on chance, and so it all depends on how many tries went to the right side, which actions Q-learning chose, and how that action was reflected by the underlying transition model. Realistically, more optimal results could have been found with perhaps a higher epsilon rate/different exploration function, or a lower learning rate, or even running Q-learning with more iterations to see if it converges differently. In the interest of time and brevity, these options are not explored here, but would be greatly beneficial in future considerations.

Figure 4 below shows the performance of all three algorithms in more detailed metrics.

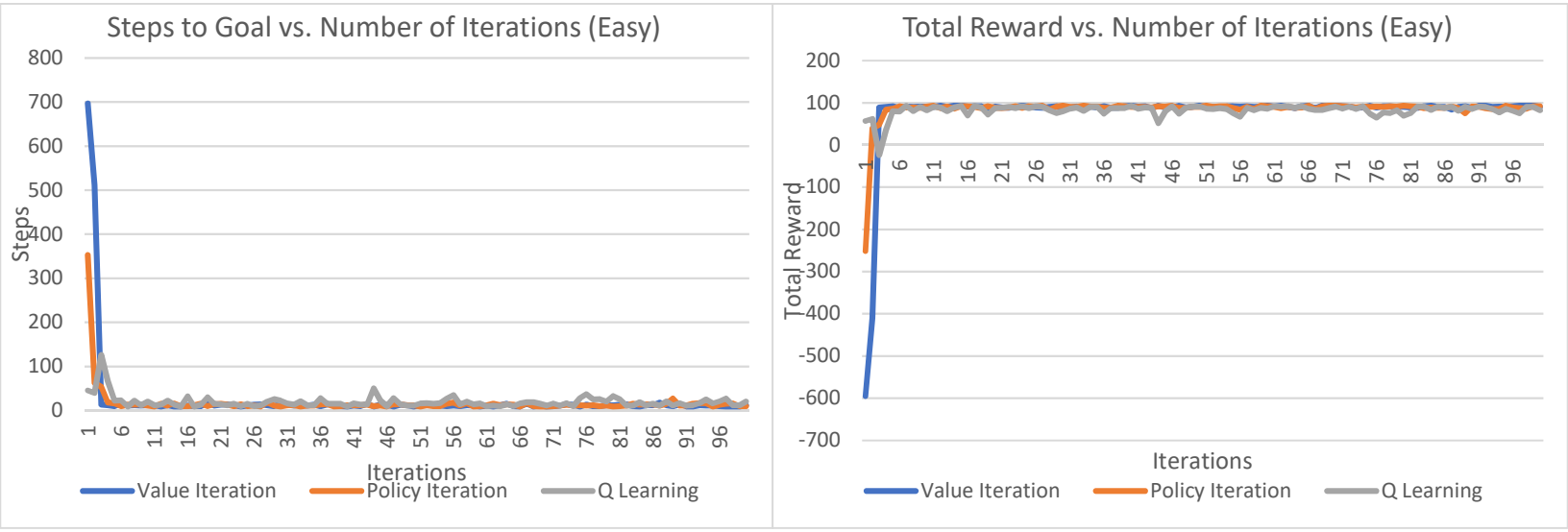

Figure 4

The left graph shows the total number of steps needed to reach the goal state, given the number of iterations run for each algorithm. As seen here, value iteration and policy iteration both took a very large number of steps the first two iterations, which make sense since they are randomly initialized and can only follow whatever the initialized max utility or policy is. Clearly, since it is random, it will take a long time before the agent accidentally falls into the goal simply based off of the 20% chance per action from the transition model. However, as soon as that happens, it only takes 1 or 2 iterations more for the values to start to reflect the true utilities and for the policy to follow along more optimally. Both do consistently well in terms of minimizing the number of steps it takes to get to the goal past the first few iterations.

Q-learning, in contrast, starts off with a lot fewer steps. This is because even though it also starts off randomly, the random aspect is in its actions taken since it knows nothing about the world and is able to explore. This is opposed to the initially fixed actions of value and policy iteration. Therefore, the randomness leads to a higher chance of falling into the goal state, since this world is an easy small one. However, Q-learning does not get much better, as it continuously fluctuates above the number of steps that value and policy iteration lead to. This also reflects the exploration aspect. Even when Q-learning starts to learn the optimal paths, it still has a chance to take an action it knows is not optimal simply for exploration's sake. Thus, on average, each run will take more steps to complete.

The right graph shows total reward, which given that the reward function is simply all -1 except the goal state of 100, it makes sense that the shape is an inverse of the steps taken graph. In fact, the total reward is just the negative number of steps taken plus 102 (to account for the first and last step). The only difference is the scale. This graph gives a better visual representation of a number somewhat comparable to utility and shows the same: value and policy iteration perform similarly optimally, while Q-learning performs just a little less optimally.

Figure 5 below shows the performance time of each algorithm in ms.

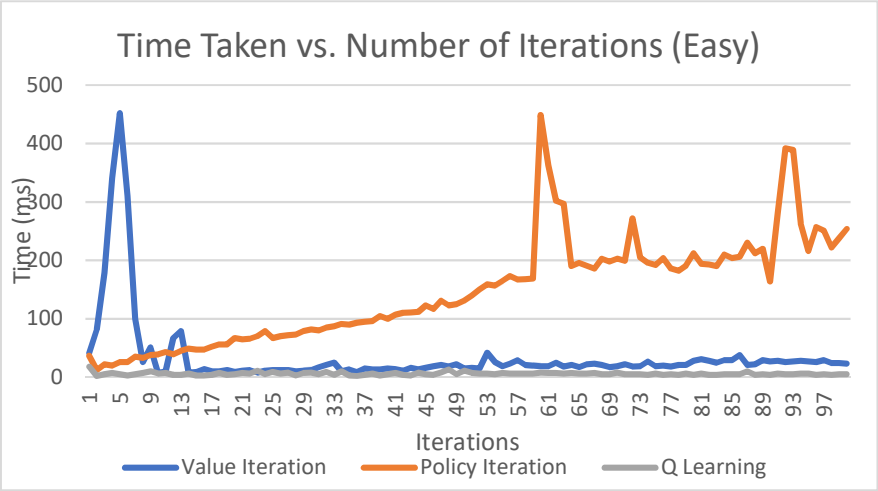

Figure 5

All three algorithms behave differently in terms of runtime. Value iteration's runtime clearly scales with the number of steps taken, starting off very high but then immediately coming back down and being fairly stable if not just slightly slower with more iterations. This makes sense since at each step taken, a max calculation is needed which is more computationally intensive. Policy iteration's runtime starts small, but increases nearly linearly with the number of iterations, save for general variance and a few huge spikes. This makes sense since policy iteration is more computationally involved per iteration in order to determine the policies followed, but the steps don't really matter since the policy is already chosen when the agent moves. Q-Learning performed the best out of the three in terms of runtime, staying very minimal regardless of steps or iterations. Since Q-learning only needs to sample and give a simple value update per iteration, it is computationally more efficient per iteration or per step. However, even though it is not highlighted here, total runtime for Q-Learning has higher potential of being long, since Q-learning tends to take more iterations to converge. With this easy problem, this aspect is less of a concern since Q-learning is able to solve it relatively optimally with a relatively small number of iterations.

Overall, value iteration worked the best for the first EasyGridWorld problem. It gave an optimal policy in a short amount of time and a short number of steps after just 2 iterations, leading to a high total reward. Despite both providing the optimal policy, value iteration marginally outperforms policy iteration, mainly due to being much faster to run. This makes sense, since the easy problem allows it to converge very quickly. It also outperforms Q-learning which was not able to provide the most optimal policy. This makes sense as Q-learning is at a disadvantage of not knowing the transition model or reward function. However, as mentioned before, Q-learning is helpful when we are in a world where we really don't know the transition or reward. Going back to the dinner example, we would have no idea

how a decision for a certain restaurant or a certain input to the group would affect general morale and group dynamic. We could have taken an action with the intent of coming to a decision, but what actually happens could be the exact opposite. Same goes for the reward. We wouldn't know if tossing out a suggestion would provoke a positive or negative reaction. The only thing we could do is just try an action, see what happens, learn and reinforce our position, and try to proceed optimally. Therefore, given this context, Q-learning shapes up pretty well compared to the other two, giving us a nearly optimal solution despite the ambiguity of the world.

HardGridWorld

Figure 6 below shows the optimal policies for the second problem after running Value Iteration, Policy Iteration, and Q-Learning, respectively, on the MDP. The color scale is the same as before.

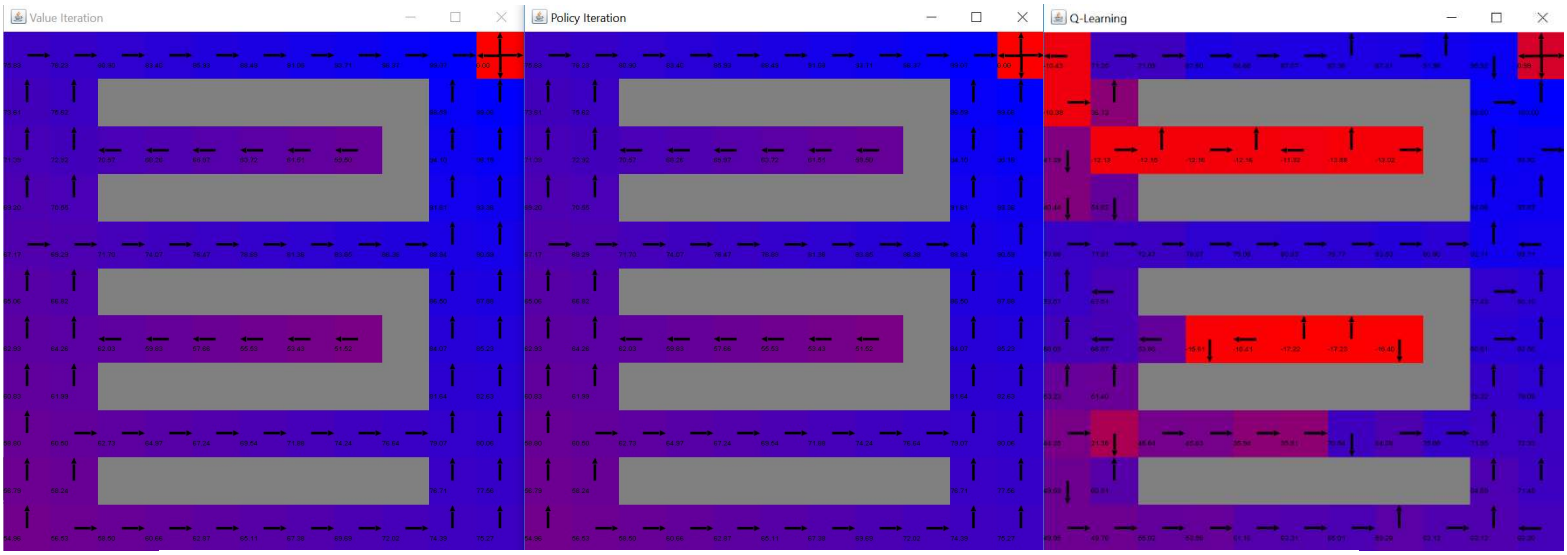

Figure 6

As shown, value iteration and policy iteration both find the same, clearly optimal policy. The initial corner is lower utility since it is the farthest away from the goal. The utility goes a little higher as the agent moves up on the left side, then higher as the agent moves right in open paths, and lower as the agent moves right in closed paths. The entire right side is pretty high utility, given how easy it is to get to the goal from there. This makes a lot of sense, especially considering that both algorithms had full access to the world's information. Knowing the true reward and transition models, both could pick the maximum utility at any point, converging onto the optimal value and picking the optimal policy. The fact that this world was bigger did not cause the algorithms to perform more poorly.

Q-learning, on the other hand, performed fairly poorly, giving what sometimes seems like random policies. The utilities are also not smoothly spread out, but overall make sense, with left states being worse than right states, and dead-end states being red with very low utility. The top left corner is also red, suggesting that the algorithm had trouble maneuvering out of that situation. In terms of policy, the general trend is still there. The algorithm is able to learn the general direction of the goal up and to the right, with some stray arrows here and there. It is also able to recognize how bad the dead-end areas are. However, it gives non-optimal policies for many states, especially within the dead-end paths and right next to the goal.

Just as in the last problem, Q-learning performing sub-optimally makes sense, since ultimately it does not know the transition model or the reward function. For the same reasons as previously mentioned, the algorithm can therefore only sample and reinforce accordingly. Since this world has the same reward function, the same assumptions apply. However, since this world has more states, there are more differences in performance. It is noticeably harder to solve this world using Q-learning. With an increased number of states, as well as multiple options for paths towards the goal, the algorithm must go through more uncertainty, compounding the randomized effect. This makes it a lot harder to converge to a specific policy, since the sample paths are more likely to be convoluted and not reflect the optimal path. Realistically, more optimal results could have by running Q-learning with more iterations to see if it converges differently. Giving the algorithm more samples and more opportunities to run, and potentially increasing the exploration rate to allow exploration of more paths, could potentially lead to a more optimal solution. In the interest of time and brevity, these options are not explored here, but would be greatly beneficial in future considerations.

Figure 7 below shows the performance of all three algorithms in more detailed steps to goal. Total reward is omitted, since it effectively graphs the same exact trend, just flipped and on a different scale (as explained in the last problem).

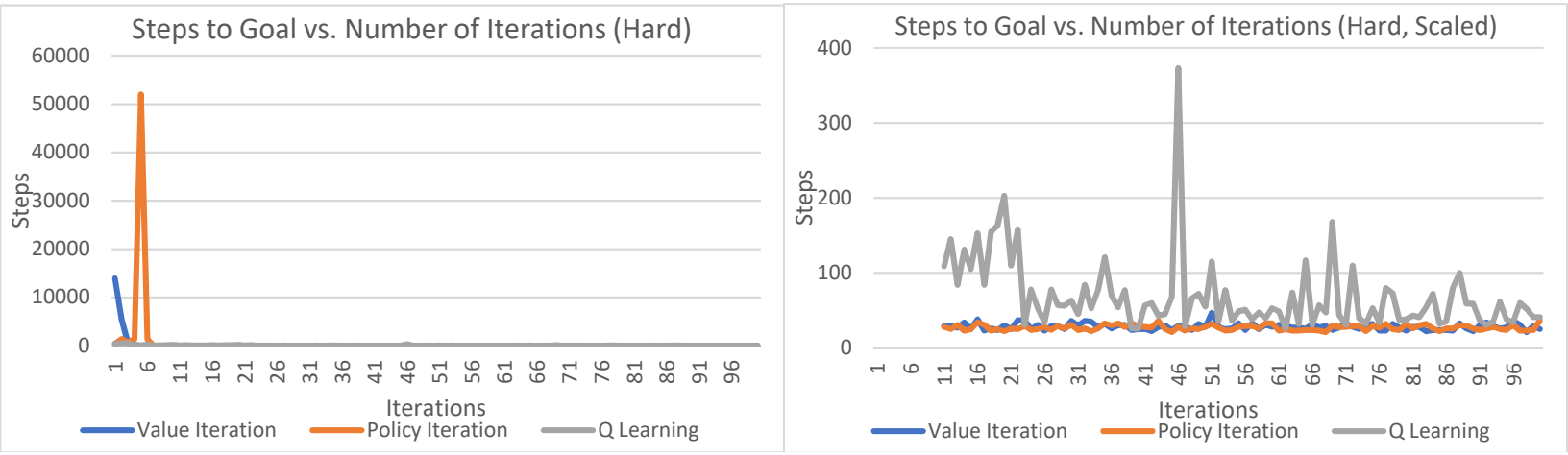

Figure 7

Two views are shown for the steps to goal. The left shows the original while the right shows the rescaled y-axis graph without the first 10 data points. This allows finer comparison without the initial spike obstructing the scale.

As shown, the results are quite similar to the last problem. Both value and policy iteration start off with a huge spike in the number of steps taken (and thus the total reward). The number is on a scale two orders of magnitude larger than before, even though the number of states has only increased by about 5 times. This shows that worlds with larger numbers of states can take a more-than-linear increase in the amount of difficulty is takes to solve them, simply because there are so many more combinations of paths that can be taken before finding the goal. Just like the first problem, both value and policy iteration are able to bring the required number of steps way down within just the first few iterations, especially highlighted in the graphs on the right. They remain relatively stable save for the variance as the number of iterations increase.

The main difference with this problem comes from the fact that Q-learning does even worse than before, which is consistent with what we have seen so far. The same assumptions apply for Q-learning as before, as epsilon is the same value. It initially takes less steps to fall into the goal due to exploration, but that doesn't really get better as the number of iterations increases. The steps (and as a result total reward) fluctuate greatly, and on average stay well above the numbers for value and policy iteration past 10 iterations as shown on the right. This can again be attributed to exploration and the fact that we don't know the transition model and reward function. As mentioned with the optimal policy discussion, the larger fluctuations can also be attributed to the larger number of states, which causes a more compounded effect on the randomness of the algorithm.

Looking at runtime differences in *Figure 8* below, the performances look fairly similar to before. One major difference is the initial spike in the value iteration runtime that was present before is no longer present here, which may be an indication of good initialization in this particular instance. Otherwise, value iteration stays fairly constant, policy iteration increases fairly linearly with a few spikes and variances, and q-learning stays at a minimal runtime below both, all consistent with the reasons mentioned before. An interesting note is that despite this world being larger, the runtime is the same as before if not even shorter. It seems that in this case, the number of states did not affect the runtime, especially considering multiple iterations.

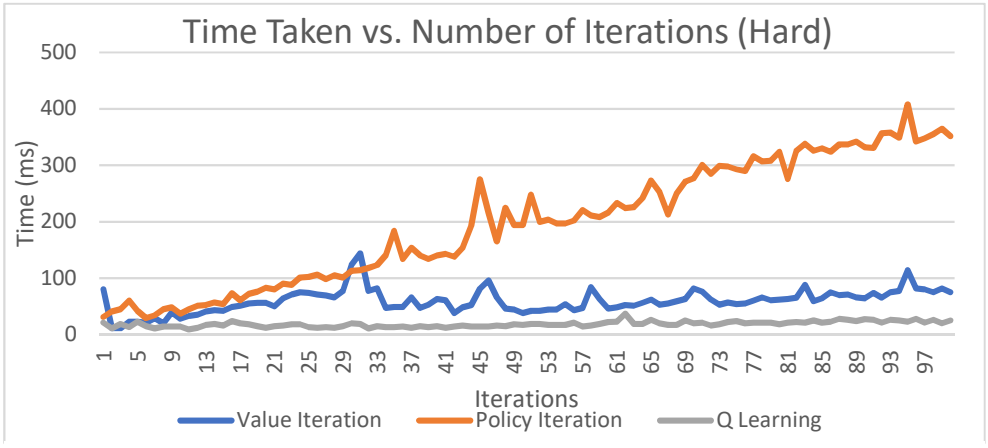

Figure 8

Ultimately, value iteration pulls ahead in the large world as well for the same reasons as the small world, providing an optimal policy solution with faster runtime and lower total steps/reward. Again though, just like before, we most likely will not know the way this world works, i.e. the transition and reward. If we are driving to the restaurant that we vaguely remember without a GPS, we will not know if turning on a particular intersection will lead us to the right path. We won't even know if it will put us on a collision course with another vehicle. Therefore, Q-learning is necessary in cases like this, even though it doesn't perform nearly as well. It can give us a general guideline to directions, but it's not optimal.

Conclusion

MDPs can model specific situations and worlds that can be solved with Value and Policy Iteration, granted that we know the world. Both give optimal policies with value iteration pulling ahead. However, we frequently don't know the model to the world we want to solve, and therefore we must rely on RL and Q-learning. As shown, Q-learning is susceptible to the size of the world and tends to do poorly or require more time in large worlds with many states. With small worlds though, it provides nearly optimal solutions that, all thing considered, work well for the tasks at hand.