Yukt Mitash
Dr. Hrolenok
CS 4641: Machine Learning
April 20, 2019

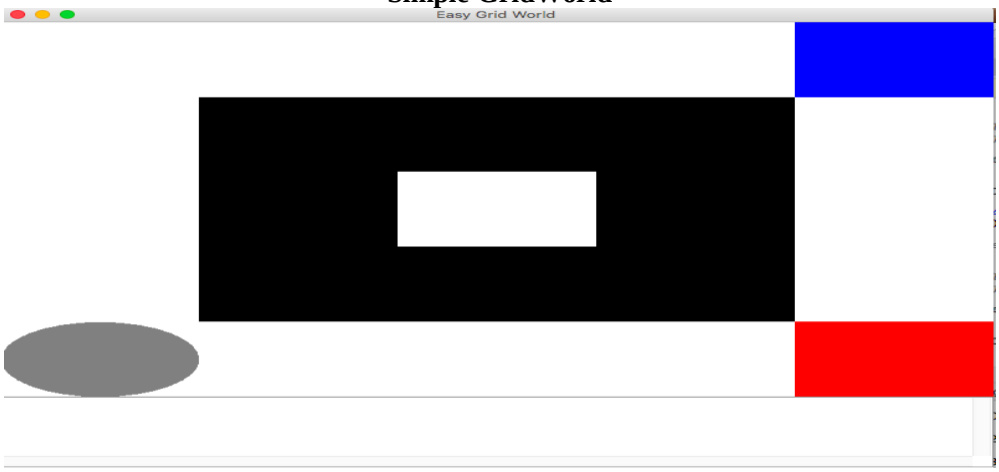
Assignment 4: Reinforcement Learning

Introduction: In this assignment we are going to use three algorithms: Value Iteration, Policy Iteration, and, Q-Learning to solve two unique MDPs. We will examine the efficiency and accuracy of these algorithms in two very different environments.

Background: An MDP, or Markov Decision Process is a model of a real-world environment characterized by stochasticity. In a nutshell, this means that actions that an agent takes in this environment are non-deterministic. The levels of stochasticity vary depending on the problem space, but an example would be that if an agent attempts to move north, there is an 80% chance that it will and a 10% chance that he will move east and a 10% chance that it will move west. An MDP is defined by the following: a set of states: $s \in S$, a set of actions $a \in A$, a start state s, a transition function T(s, a, s') that gives the probability of moving from state s to s' when the agent takes action a, and a reward function R(s) that gives the "reward" (positive or negative number) of moving to state s. Although MDPs do not always have terminal or exit states, the problems we examine will have multiple. This is simply a state at which the agent exits the world. Generally, the reward function will give a large positive reward when the agent enters the Goal state, a large negative reward when the agent enters a Trap state (could be a fire pit), and a small negative reward to the agent simply for existing (this encourages the agent to find the shortest path). Solving an MDP involves creating a policy $\boldsymbol{\pi}^*$ that maps each state to the optimal action such that if the agent acts optimally after taking that action, the sum of rewards the agent collects will be maximized. The algorithms used to solve an MDP are based on the Markov assumption. Essentially, the next state of the agent is only dependent on its current state, not its past states. Now, let's examine the two MDPs we will be solving.

MDPs

Simple GridWorld



The first MDP is represented above. This was borrowed from Juan Jose's code and slightly modified. The grey circle is the agent, the black represents a wall, the blue square is the goal state (reward of 5), and the red square is the terminal state (reward of -5). The reward function returns 5 for the goal state and -5 for the trap state and -0.01 for existing. The action space is {North, South, East, West}. The transition function returns 0.8 for the chosen direction and 0.2/3 for all other directions. The nature of MDP algorithms involve computing the best possible action in an uncertain environment. This MDP is interesting because it allows us to examine, at a very simple level in a small state space, how the algorithms weigh risks and reward to compute optimal policies and how the algorithms deal with walls. Another interesting thing about this MDP is that there are three possible states between the goal state and the trap state that the agent will never be able to access because they are blocked by the walls and the terminal state. Seeing how the different algorithms assign actions to these inaccessible states

will allow us to gain a better understanding of the process these algorithms follow. This MPD has 16 states, so it would be our "smaller" MDP.

Ring of Fire

This is an MDP that I built myself using BURAP to put the decision-making abilities of our algorithms to the ultimate test. The grey circle is our agent. The red represents terminal "trap states" with rewards of -5. The black represents walls and the blue represents our terminal goal state with a reward of 5. The living reward is (-0.01). The action space is {North, South, East, West}. The stochasticity is 0.8 of going in the right direction and 0.2/3 for the remaining directions. At its core, solving an MDP involves balancing risks and potential rewards to assign the optimal action. Basically, in this MDP, the agent has two options: It can go through the states between the fire pits and risk dying for the possibility of reaching the goal state faster. Or it can go around the walls risk-free, but reach the goal state much later. It will be interesting to see how our algorithms balance safety and risk when computing the optimal policy. One real world application of this MDP could be an autonomous vacuum cleaner in a home with stairs. Assuming, the vacuum cleaner has a map of the home and the goal is a dirty state, the vacuum cleaner may take the risky route near the stairs and risk falling down the stairs or take the safe route and get to the goal state much later. This MDP has 67 states, so this will be our "larger" MDP. This MDP is very different from the previous one due to its many terminal states that test the reasoning abilities of the algorithms and the fact that it is larger. Now, let's examine the algorithms we will be using.

Algorithms

Value Iteration: Value Iteration takes the state space, the action space, the transition function, and the reward function of an MDP and returns a policy, or a map of each state to the optimal action. It does this using a utility function to assign a utility value to each state. A utility value is the sum of the current reward and all following discounted rewards, assuming the agent acts optimally after its current state. Then it determines the optimal action for each state based on which action maximizes the utility value. The utility function can be expressed using the Bellman Equation:

$$U_{t+1}(s) = R(s) + \gamma max_a \sum_{s'} T(s, a, s')U_t(s')$$

Basically, the Bellman equation states that the utility at time step t+1 for state s is the reward at state s plus gamma (discount value) multiplied by the maximum (for each possible action) sum of transition probabilities multiplied by utilities at state s' determined at time step t. It is the reward at the current state plus the average reward weighted by the transition function following the optimal action sequence. The action for the state in the policy is the action that returns the maximum $\sum T(s, a, s')Ut(s')$. The algorithm keeps updating utilities and actions until the utility values converge (differ by smaller than a certain threshold). I have yet to explain what gamma is. Gamma is a discount factor, such that $0 < \gamma < 1$, that serves two purposes: It encodes the real world idea that rewards received earlier take precedence over rewards received later and forces the algorithm to value earlier rewards more. This makes sense because the agent could die and not live to see later rewards. It also makes sure that the equation converges and does not run for an infinite number of time steps. The algorithm is actually very simple if you understand the equation. First, it assigns arbitrary utilities to each state (generally 0.0) at time step 0. Then it reassigns utilities and actions for each state based on the Bellman

equation and utility values from the previous time step. This continues for either the maximum number of iterations or until the utilities converge.

Policy Iteration: Policy Iteration takes the state space, the action space, the transition function, and the reward function of an MDP and returns a policy. Policy iteration seeks to fix one of the big problems with Value Iteration, VI will generally find an optimal policy well before its utility values converge. This makes intuitive sense because the optimal action for a state is the action that maximizes the utility function. So even though the utility value will change from iteration to iteration, the action that maximizes it will stop changing much earlier. As a result, VI does many unnecessary iterations. Policy Iteration fixes this by using modified forms of the Bellman Equation to iterate through many policies rather than utilities. First, the algorithm generates a completely random policy(π^0) by sampling an action for each state. Then, it uses policy evaluation (equation below) and the current policy to calculate a utility for each state.

$$U_{k+1}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') + U'(s'))$$

This is essentially the original Bellman equation except it already has an optimal action so it does not need to determine the max action. After determining a utility for each state it updates the action at each state using policy extraction (equation below) and the utility from the previous step.

$$\pi_{i+1}(s) = \text{argMax}_a (T(s, a, s')[R(s) + \gamma U'(s')))$$

After the algorithm has a utility value for each state, it can plug that utility into the equation above and use the argMax function to calculate the optimal action for each policy. This continues until the policy converges or the algorithm reaches the maximum number of iterations. The two benefits to this algorithm are that the policy evaluation equation is linear, so it can be solved with linear algebra rather than iteration and the algorithm continues until the policy converges not until the utility converges, so it should be less expensive for many MDPs. If it is an MDP with many similarly value utilities, it might not be cheaper.

Q-Learning: Generally, an agent will not have a computational model of the world it is exploring and will have to learn the features of the world as it goes. This is where Reinforcement Learning is used. Reinforcement Learning algorithms compute an optimal policy when given only the state space and the action space (no reward or transition functions) and the start state. Q-Learning involves having an agent explore an environment both randomly and non-randomly and using what it experiences to update the q-value for each state, action pair. The Q-value is generally what follows the max function in the Bellman Equation. The equation is below.

$$Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')[R(s') + \gamma \max_{a'} Q_k(s',a')]$$

Essentially, the q value of a state-action pair is the sum of the probabilities of all resulting states multiplied by their reward plus the discounted maximum Q values of the possible state. This is actually pretty easy to compute without a transition function or reward function. The algorithm starts by setting default values (generally 0) for each state-action pair. At each time step, the agent uses an epsilon value (exploration rate) to decide what action to take. This is similar to algorithms like RHC and SA. The agent will use the epsilon value to decide between randomly picking an action or picking the action corresponding to the max Q in the current state. After the agent takes an action, the algorithm uses sampling to update our Q for the given state and action. Below is the updating equation.

$$Q(s,a) = (1-\alpha)Q(s,a) + (\alpha)[R + \gamma \max_{a'} Q(s',a')]$$

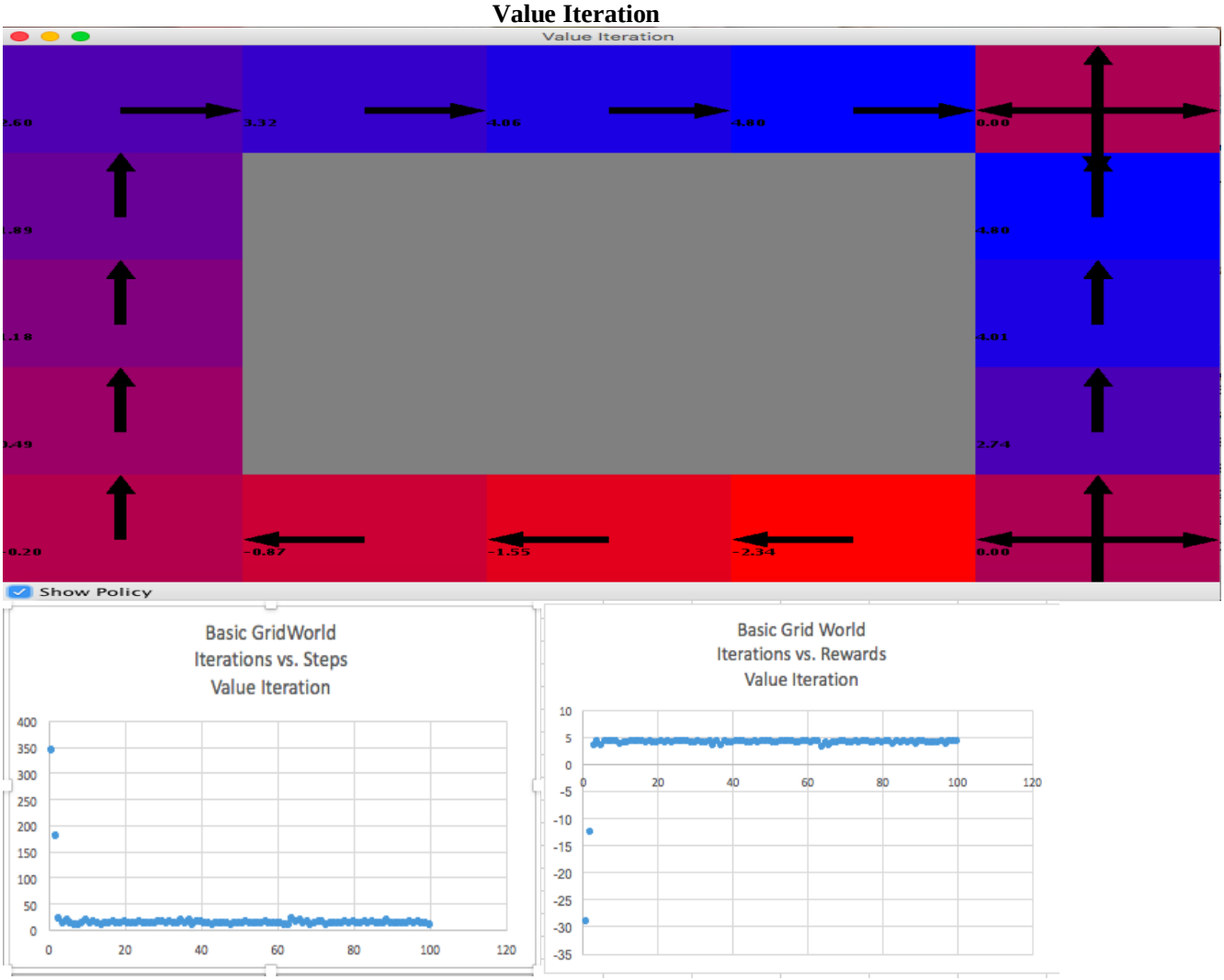
This equation is great because it replaces the transition model with an average of many different outcomes and updates the q value accordingly. R is the reward received from the action, Q(s,a) is the current Q estimate, and alpha (α) is the learning rate. A higher alpha corresponds to a higher weight placed on more newly sampled Q values and thereby a faster rate of learning. The great thing about Q-Learning is that it can learn while acting sub optimally. This is called off-policy learning. Now let's use these algorithms to solve MDPs.

Simple Grid World

Introduction: We are going to run all three algorithms on our Simple Grid World, examine the resulting policies, and examine how long the algorithms take to converge. There are two measures of convergence we rely on: The sum of rewards and the size of the action sequence. We want to be able to run these algorithms for a specific number of iterations so that we can examine graphically when the algorithms converge. For each algorithm, we will set a very high delta (this is the maximum difference between values such that a difference

below this will cause the algorithm to converge) so that the algorithm runs for the specified number of iterations rather than until convergence.

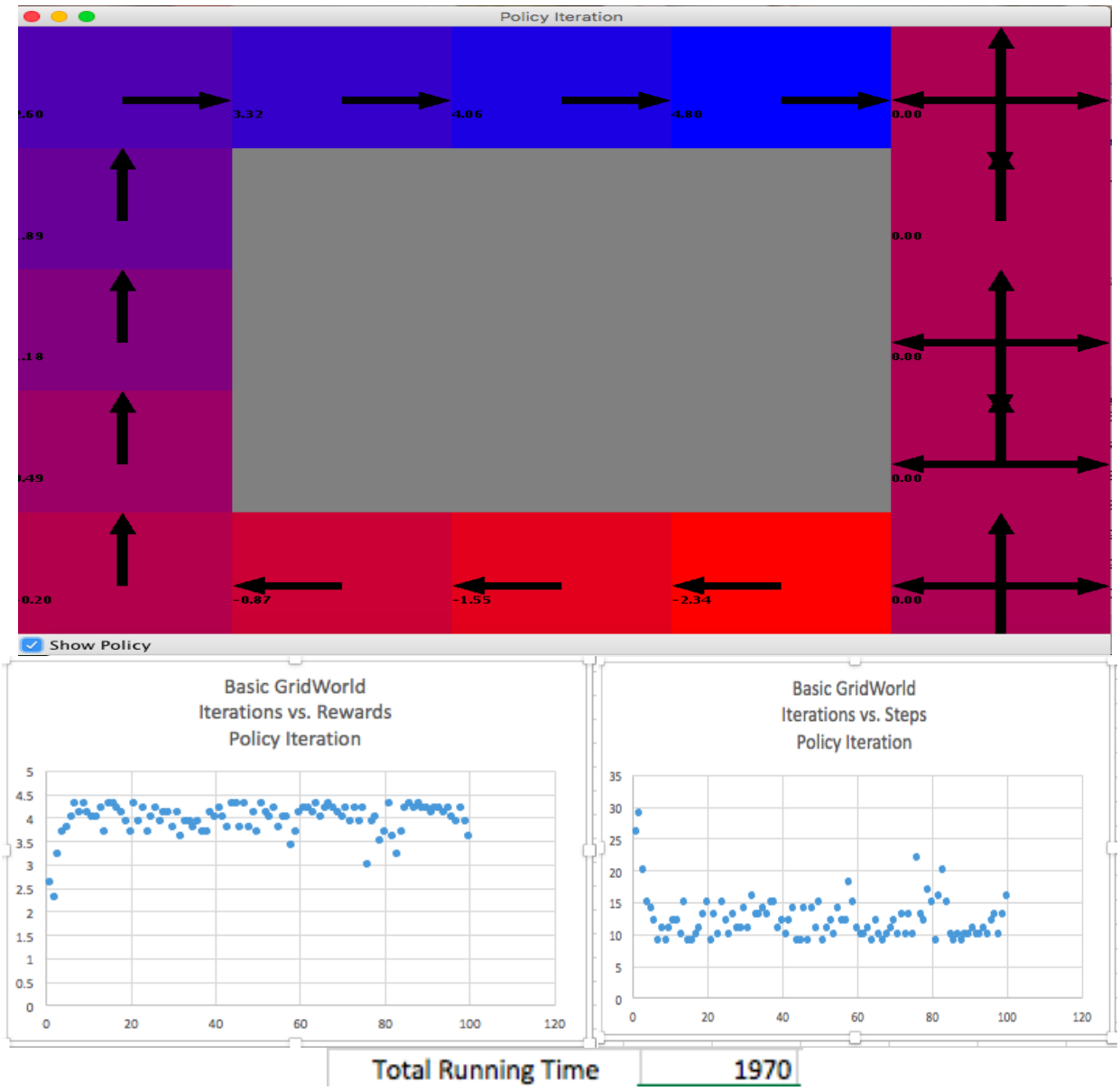
Value Iteration



Total Running Time (ms)	1472

Analysis: After running value iteration for 100 iterations, it is clear from the graphs of the reward sum and the steps (action sequence) that the algorithm will converge very quickly (only about 3 iterations).and very smoothly. Look at the policy above. The goal state is in the top right and the trap state is in the bottom left. The good thing about this small state space is that one can tell the actions make sense simply by looking at them. All of the actions make intuitive sense with regards to the stochasticity of the environment and the reward function. It is interesting that Value Iteration generated utility values for the three states trapped between the goal state and the trap state. The agent, given its start state, will never be in these positions. However, this does not matter because value iteration generates utilities on a state by state basis rather than based on an agent's movement. These values are useful to have in the event that the agent's start state changes. The reason the utility values converged so quickly is because the small state space made determining the utilities computationally simple.

Policy Iteration



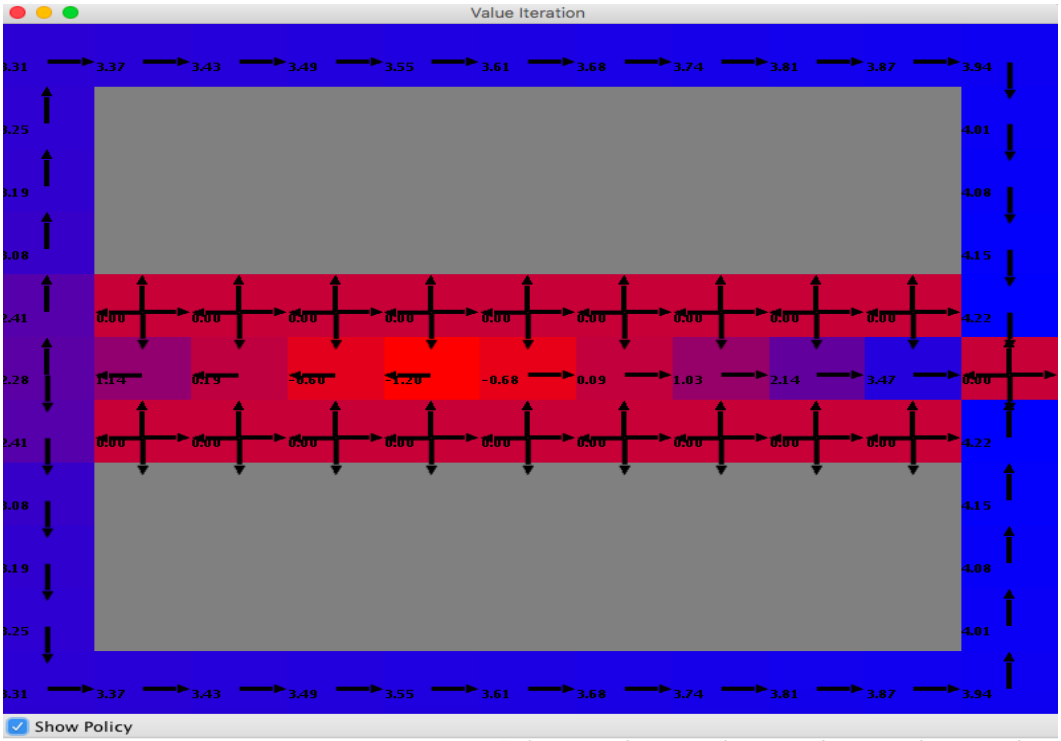
Analysis: The policies and utilities for both algorithms are entirely identical with one exception. Policy Iteration did not generate a utility value or an accurate policy for the three states between the goal state and the traps state (our unreachable states). This is because of the Policy Evaluation step in Policy Iteration. After assigning an action to each state, it uses this equation to update all utilities based on the new policy: $U_{k+1}(s) = R(s) + \gamma \sum s'(T(s, \pi(s), s') + U(s'))$. This equation involves updating utilities for all states based on the current reward and later rewards. This means that policy evaluation can never update values for states it cannot reach based on the state that it starts with. This is why the values for those three states are 0.0 (the starting utility value). Another thing to note is the higher running time of Policy Iteration. This is to be expected because each iteration of Policy Iteration involves two computationally expensive steps. The first step, policy evaluation, involves iteration through each s' for each s, which is approximately $O(s^2)$. The second involves iterating through each s' for each action for each state which is $O(a*s^2)$, while VI involves iterating through each s' for each action for each state, which is $O(a*s^2)$. Although the Policy Evaluation can be reduced computationally with Linear Algebra techniques, it will still involve an extra step causing a larger run time. The tradeoff is generally that although PI takes longer iteration by iteration, it may converge faster. This was not the case here. PI took 7-10 iterations to converge and even then, there was much noise. Although the algorithms computed

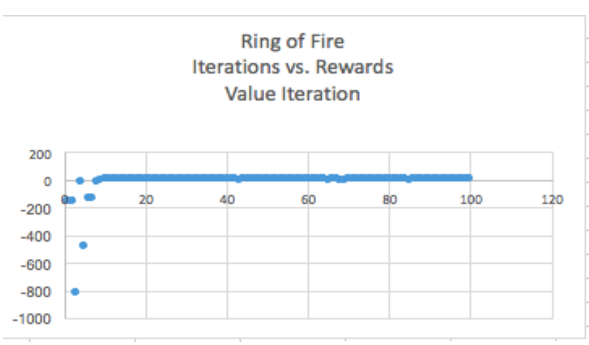
nearly identical policies, PI took almost twice as many iterations. This is because we have a very simple environment with a high stochasticity of 0.8 and very little variability when it comes to decision-making. For environments like these, utilities can be calculated very easily and will converge very quickly. PI is almost too complex for a simple environment and should be saved for cases where VI takes too long to converge.

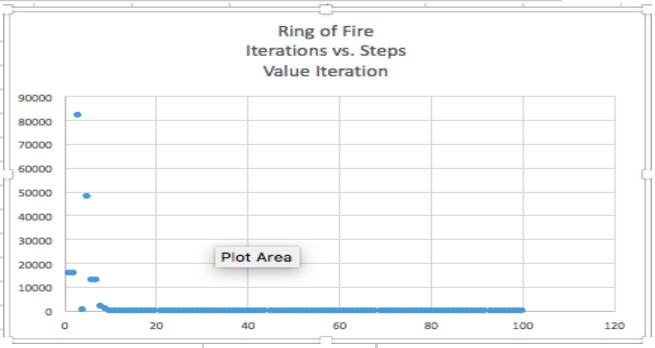
Ring of Fire

Introduction: We will follow the same steps of running the algorithms for a specified number of iterations and utilize the same measure of convergence. This MDP will likely take longer to converge so we may use more iterations.

Value Iteration





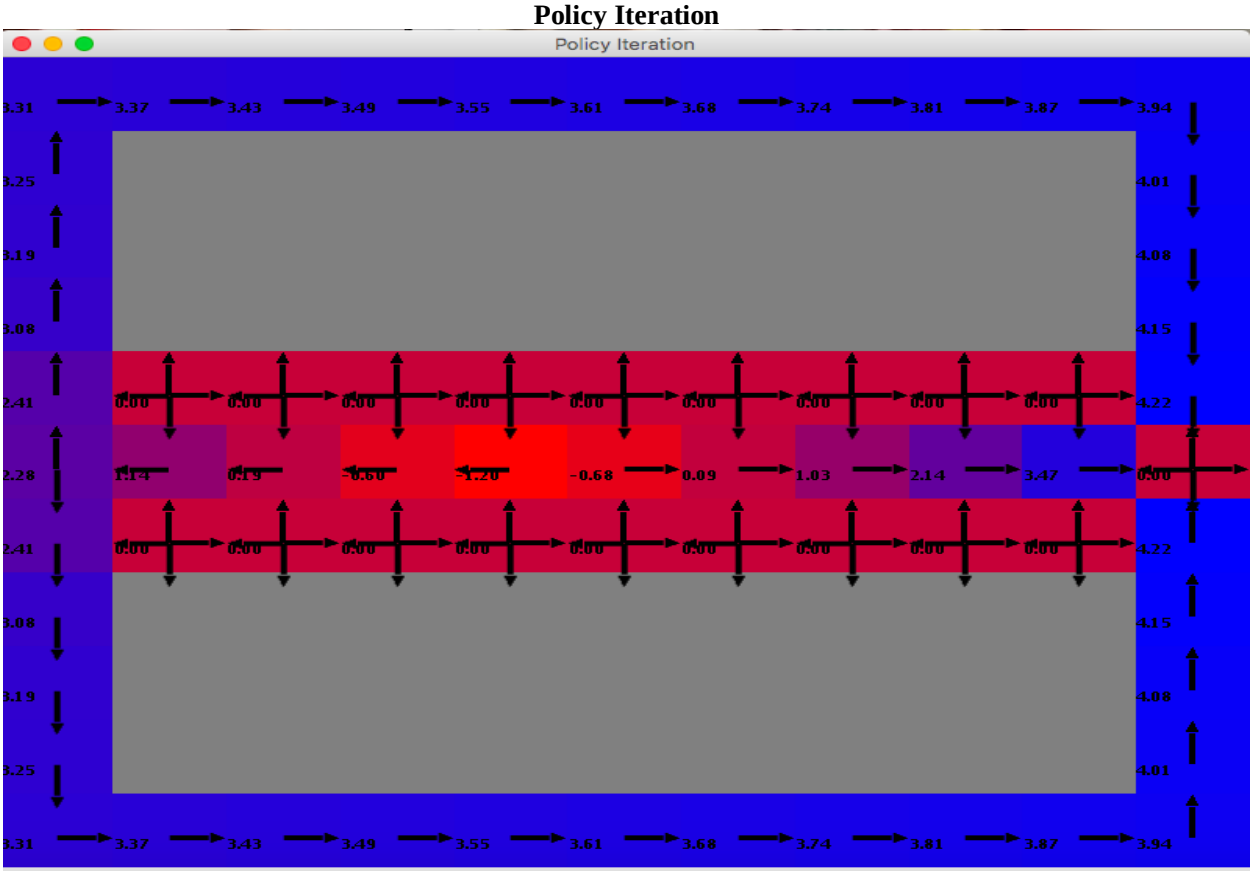


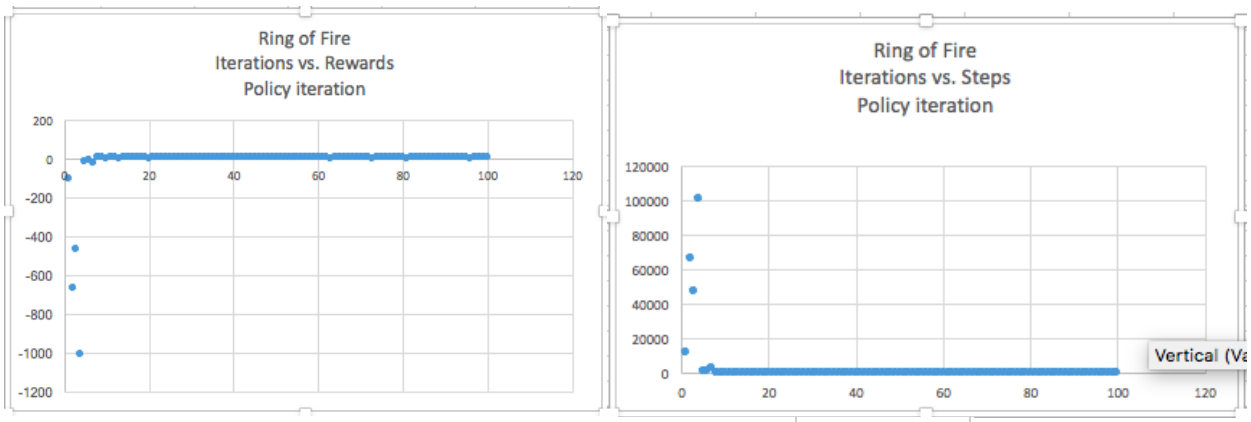
Total Runtime (ms)	2974

Analysis: Unsurprisingly, when VI was run here compared to the first MDP it had a higher runtime and took more iterations (10) to converge. This is because the state space is significantly larger and complex, so the number of iterations to calculate accurate utilities at each state will be higher. More states means more iterations of $U_{t+1}(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_t(s')$ at each iteration of VI and therefore a higher run time. So even though the algorithm was run for the same number of iterations for each MDP, each iteration is computationally more complex for this MDP. 10 iterations are pretty decent for convergence. This means that VI works relatively well for this MDP. This means that the utility values take fewer iterations to converge. Basically, this MDP has a simple enough state space, stochasticity, and action space that computing the equation above is rather simple. Our MDP has a relatively high stochasticity of 0.8, so there is not much

uncertainty in the T(s,a,s') function. Also, given the many walls in our MDP, the actions are not very difficult to compute. The algorithm simply has to weigh the possible future reward of hitting a wall and remaining in the same spot or moving towards the goal state. It will always choose the latter. Essentially, the relatively low room for decision making caused this algorithm to converge quickly. Let's examine the policy above. The boxes adjacent to the walls are the "trap" states and the box outside the wall is our goal state. All boxes with four arrows are terminal states. It is interesting how the policy generated behaves in our "tunnel" between the two fire pits. The living reward is very high here (-0.01). This means that our algorithm is not inclined to create a risky policy because there is basically no "rush" to reach a terminal state. The policy avoids the tunnel at all costs (look at the square before the tunnel's entrance) and encourages the agent to leave the tunnel if it ends up there. The only time the policy encourages the agent to go deeper in the tunnel is if it is in the middle or closer to the living reward because the reward outweighs the risk. When this algorithm is run with a lower living reward (-0.2) it takes much more risks and often encourages the agent to go deeper in the tunnel. If we had more space, I would include those graphics.

Policy Iteration





Total Runtime (ms)	7600

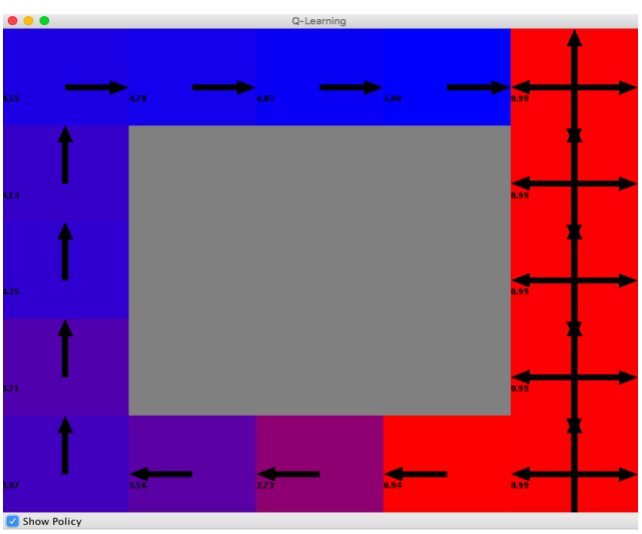
Analysis: If you take a look at the graph, you will see that the policy and utilities generated by Policy Iteration and Value Iteration are identical. This makes sense because both algorithms use variations of the Bellman Equation so after enough iterations they should converge to similar values. What we are more concerned with here is the time it takes to converge. Both graphs show convergence occurs around 5 iterations. This is a significant improvement on the 10 iterations of Value Iteration. The runtime for PI over 100 iterations is over twice that of VI. However, PI makes up for this by converging earlier. Even though the runtime for each iteration of PI is higher than that of VI, PI converges earlier, so if we had not set a very high delta for this problem, PI would have likely had a smaller runtime. Also, PI's runtime can be reduced significantly by using linear algebra techniques rather than iteration for the Policy Evaluation step. The reason PI converged faster is pretty intuitive. This MDP has a large state space, several terminal states and a goal state. This makes the utility equation (U t + 1 (s) = $R(s)$+ γmaxa $\sum$s'$T(s, a, s')U$t (s')) more complicated. This means that starting with utility values of 0 (or something arbitrary) and recalculating utility values until they converge will take a much larger number of iterations than doing so for the previous MDP. As we know, one does not need exact utility values to generate a policy. Because the utility equation is so complex, the optimal actions will converge much faster than the optimal utilities. This makes the Policy Iteration approach of evaluating and updating policies until the policy converges much more efficient.

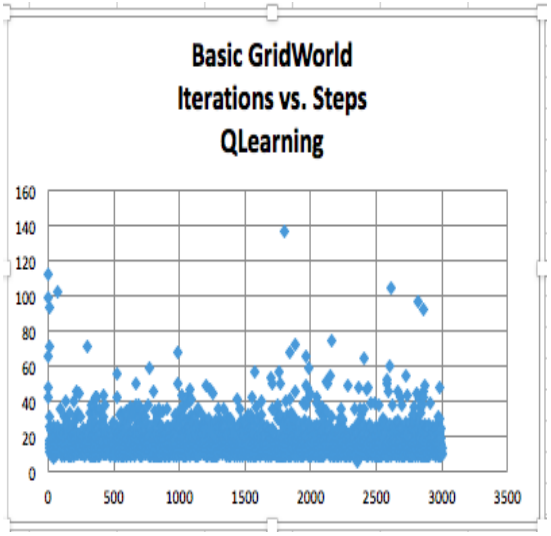
Conclusion: The big take away seems to be that PI converges faster than VI for larger state spaces and VI converges faster than PI for smaller state spaces. If we had used a third MDP with an even larger state space this would have been even more obvious. This is because with large state spaces, it takes many more iterations for the reward values to propagate out to each state as utility values. As a result, the utility values will converge much after the actions, making PI more efficient. With small states (see the first MDP) reward values can propagate out to each state almost instantly (3 iterations). Because Policy Iteration starts out by evaluating a random policy and changing that until convergence, it could never converge as fast as Value iteration in this case. In fact, Policy Iteration seems to act sub optimally with small state spaces. This is because Policy Iteration starts out with a policy, updates utility values based on the policy, and uses the state's information to update the policy. MDPs with a smaller state space have less information and fewer utility values that the algorithm can use to update its policy, causing poor performance and more iterations for convergence. It is interesting that PI and VI return identical policies and utilities regardless of the MDP. This is due to the fact that both algorithms rely on a similar equation and after enough iterations they will produce identical results. Obviously, larger state spaces resulted in larger running times due to more iterations of the Bellman Equation.

Reinforcement Learning

Introduction: Now, we will study the policy and convergence of Q-Learning, a reinforcement learning algorithm, when run on both MDPs.

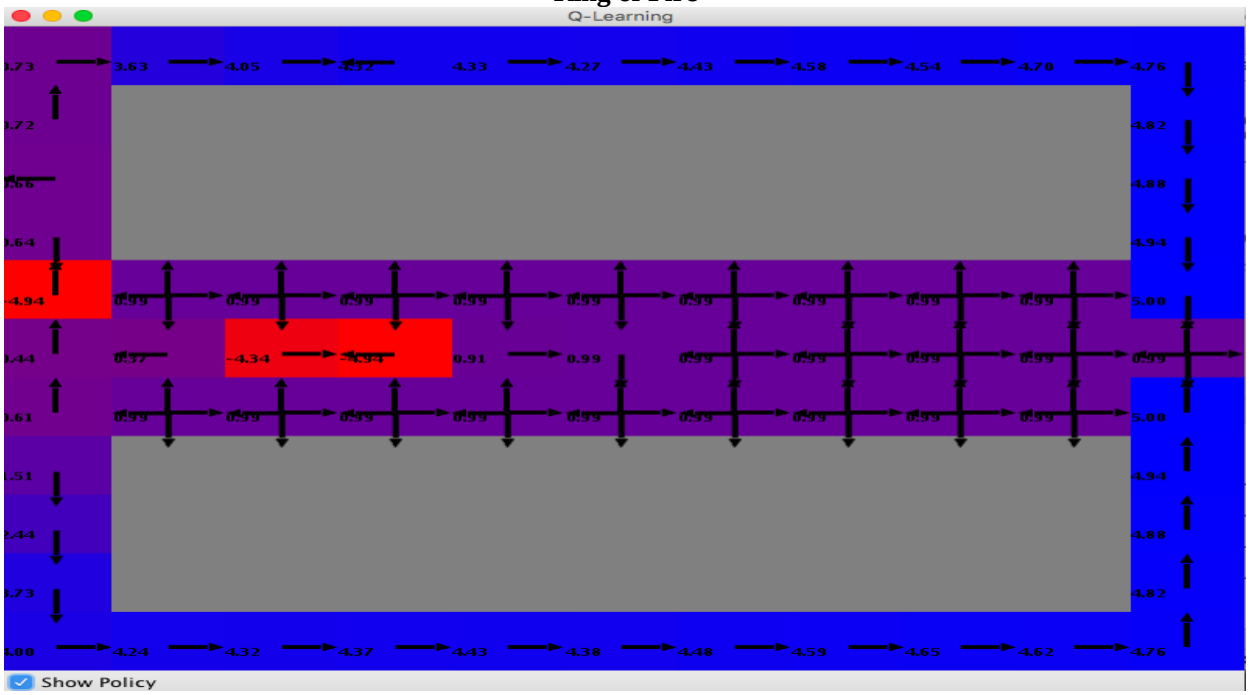
Simple GridWorld





Analysis: Above is the policy and graph of iterations vs. number of expected actions to goal when Q-Learning is run on Simple GridWorld. Unlike with VI and PI, the convergence of Q-Learning cannot be determined based on a graph of Iterations vs. Steps. This is because at each iteration of the algorithm, Q-Learning, unlike our other algorithms, uses some probability epsilon to either act according to the current model or to act randomly. Whenever the agent acts randomly, it will simply pick a random action to explore a new transition and a new reward. This action will likely be suboptimal, which is what causes all the noise in the data. As a result, it is hard to tell where exactly the algorithm converges based on a graph. Instead, I determined when the algorithm converges based on when the policy converges (when the actions stop changing). This occurred at around 3000 iterations compared to 3 and 7-10 for VI and PI. This demonstrates that acting in a world without having a model drastically increases the time an algorithm takes to extract a policy. The policy here is nearly identical to the one computed by the other algorithms. One difference between this and VI is that there is no value for the three unreachable states. This is because Q-Learning is a type of online learning in which an agent interacts with the environments and updates its Q-value for a given state-action pair based on the action it takes, the state it leaves, and the state it enters. Since it cannot enter those three states, it is unable to determine Q values other than the default ones for that state. Because those states have Q values of 0 for each state-action pair, the arrows point in every direction. Nevertheless, this shows that for a simple state, the optimal policy can be determined without a reward or transition model. One more thing to note is that the values displayed in each state are the Q-values associated with optimal action from that state. Given the Bellman Equation for a utility: $\hat{U}_{t+1}(s) = R(s) + \gamma max_a \sum_{s'} T(s, a, s')\hat{U}_t(s')$. Q(s,a) is simply the elements following the max (a) function. The VI and PI algorithms have the optimal utility of each state. The corresponding Q value for those utility values is simply the utility value minus the reward of the state (see Bellman Equation). Since the living reward is -0.01, the Q values here should be more than the corresponding utility values in the PI and VI maps by 0.01. However, these Q values are much higher than the corresponding VI and PI values, leading me to believe that the algorithm has not yet converged after 3000 iterations! This means that Q-Learning requires a massive amount of run time before it can calculate the right Q values. This makes sense because of the way the algorithm learns. It starts off by mapping all possible Q(s,a) to 0 (or arbitrary value). At each state the agent is in, it uses epsilon (the exploration probability) to decide whether to act randomly or act using the model (similar to RHC and SA). If it acts using the model it chooses the action with the greatest Q(s,a) value. Otherwise, it picks a random action. Each time the model/agent chooses an action from a state it updates the Q(s,a) value as follows: $Q(s,a) = (1-\alpha)Q(s,a) + (\alpha)[R + \gamma \max_{a'} Q(s',a')]$. Here alpha is the learning rate, corresponding to how much the algorithm trusts newer estimates compared to older and R is the reward it experiences. The use of averaging makes up for the lack of a transition function because this way the algorithm ends up averaging the outcome of all state-action pairs. One can only imagine how many iterations of this are necessary to create a good policy. The agent must take every possible action from every existing state and experience every resulting state for that action and this must be done several times for the values to converge. Even for a small MDP like this, at least 3000 iterations are needed to create an optimal policy.

Ring of Fire



Analysis: I did not include any of the graphs generally used to determine convergence because of the reasons mentioned above. This is the result of 7000 iterations of Q-Learning and one can tell merely by looking at the policy that it is suboptimal. The actions at (1,4), (3,1), (4,1),(5,1),(6,1),(6,3),(6,5-9) are clearly suboptimal given the results of PI and VI and common sense. However, most of the other results seem to be optimal. This demonstrates that acting in a large, complex MDP makes model-free learning incredibly difficult. The algorithm would have likely computed the optimal policy after 10,000 or so iterations, but time would not permit. Q-Learning converges at >7000 iterations here compared to 10 and 5 iterations for VI and PI respectively. This gives some perspective of the immense complexity of determining a policy in a non-deterministic world without a transition or reward model. Again, the Q-Values here are greater than the utilities for PI and VI by a lot evidencing a lack of convergence (see above). Why does this take so many iterations? To sum up what is mentioned above, the answer lies in the learning techniques. Q-Learning uses an exploration probability to decide between following the policy for the next action or randomly selecting the next action. It then updates its Q(s,a) value based on the state it lands in, the action it takes, and the reward it experiences. To create an optimal mapping for Q(s,a), the agent must visit every state, take every action, and experience every resulting state several times. For a large MDP, this requires an absurd number of iterations.

Conclusion: What we have learned from these tests is how our algorithms behave in varying environments. Policy Iteration works excellently with large MDPs and not so well with small MDPs. Value Iteration is good for small MDPs and not as good for larger MDPs. Finally, for model-free learning, Q-Learning is an excellent tool, but it takes many iterations to converge, especially with large MDPs.