CS 7641 Assignment 4 – Markov Decision Process
Jeremy Martinez – jmartinez91@gatech.edu

Introduction
Two separate Markov Decision Processes (MDP) are observed in this project, one with a low number of states and another with a high number of states. The difficulty of each problem is directly correlated to the number of states in the MDP. We solve each MDP using value iteration, policy iteration and Q-learning. We run each of these at several different iterations until they converge. Then we will analyze each in terms of overall reward, number of steps/actions taken to reach final state, and computation time

Our two MDP problems will each be grid worlds, one 11x11 and one 17x19. These will represent our easy and hard MDPs. Each grid will represent a real-world scenario and will tie back in to an application of this problem. For each scenario we will introduce the problem, run all three of our algorithms and analyze how each one performs, then we will compare/contrast each algorithm and try to find an explanation as to how and why they performed the way they did. We will also take a look at how parameter tuning in regard to discount factor plays a role in performance.

Markov Decision Process
It is important to take a step back from the code and revisit MDPs conceptually. An MDP tries to capture a world, like in our grid world representation, by quantifying it in four key units of measurement.

- States: a set of tokens that represent every state one could be in
- Actions: a list of actions performed within the confines of our world (e.g. up, right, down, left)
- Reward: a reflection of usefulness (or reward) of being in a state R(s)
- Model: a function of three variables, T (s, a, s') where s: state, a: action, s' is next state
    - T ~ Pr (s' | s, a) where a T function produces an array of transitions each with a probability

Expanding on this further, this (not so intuitively) introduces the Markovian property. This property suggests that only the current state/action matter and it does not consider any history or prior set of actions. Say for example you wanted to step through a world without retracing your steps. This use case would be outside the scope of an MDP because the Markovian property prohibits it. One way around this is to ensure that each individual state encapsulates history (feels a bit like cheating…).

It is worth noting that reward may be thought of as a function of being in a state as described above R(s). It can also be thought of as a function of entering a state R(s, a), or transitioning from one state to another R(s, a, s'). It is easier to focus on one and stay consistent, so we will define and implement it as R(s).

The MDP is evaluated in the result of a policy: $\prod$(s) -> a. When referring to "solving" our MDP, we mean solving for the optimal policy which maximizes the long term expected reward. In the plots this will be measured by the highest ending reward. This similarly ties back into what we've been doing all semester when we think of a policy as a function of state, action and reward. Our optimal policy ($\prod$*) will be represented as a sequence of units <s, a, r>.

The indicator of winning/losing the game (whether or not the policy was optimal) is determined by the total reward, which the reward of the solution (final) state has a strong influence on. With several in-between steps and a delayed reward providing feedback on success, it is difficult to hone in on which actions led to the policy being optimal. This problem is referred to as the temporal credit assignment problem. To help with this, along with encouraging progress towards a solution state, often times in an MDP a negative reward will incur on all intermediate steps. The lower the [negative] reward, the quicker the game will end. We will discount these rewards at different rates and this is a crucial parameter that will be tuned in our problems. We will refer to it as gamma (γ) where $0 \leq \gamma < 1$.

Bellman Equation

The Bellman Equation is a formula for finding an optimal policy, defined by $\prod^*$. The sequence of actions taken in the optimal policy can also be derived as $U^{\Pi^*}(s)$. It is also worth noting that reward is not the same as utility $R(s) \neq U^{\Pi^*}(s)$. Below is provided an overview function for deriving the optimal policy and utility function.

$$\Pi^* = argmax_{\Pi} E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \Pi\right] \quad U^{\Pi^*}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \Pi, s_0 = s\right]$$

After defining the optimal utility function, we can plug it into our $\prod^*$. This formula is easier to grasp when we unroll the intuition behind the recursive nature of the utility function. We ultimately define Bellman equation in the equation below to the left. The utility function to the right is provided for a better understand of this Bellman equation:

$$\Pi^*(s) = argmax_{\mathrm{a}} E\left[\sum_{s\prime} \mathrm{T}(\mathrm{s}, \mathrm{a}, \mathrm{s}') U(s')\right] \quad U^{\Pi^*}(s) = R(s) + \gamma \, max_a \sum_{s\prime} T(s, a, s') U(s')$$

Bellman equation provides n equations. For each of these equations we can enumerate R and T, and so these values are known. However, we do not know U. When solving Bellman equation, we must solve for U, which there is a quantity of n. It is worth noting that the max operator makes this equation non-linear.

Value Iteration, Policy Iteration, and Q-Learning

One way of solving Bellman equation is to start with arbitrary utilities, updating them based on the neighbors, and repeating until it converges. What does it mean to converge? To converge means to find the optimal policy, which involves the utility set that maximizes our overall reward.

Value Iteration (VI) attempts to solve our MDP by applying the Bellman equation. It starts with an arbitrary utility and optimizes that utility from the solution state outward, updating the utility at neighboring states until all of the states are optimized. This follows intuition as it is just continually building upon an initial utility as a result of subsequent iterations. To recap, VI finds the optimal policy by actually finding the optimal utility.

Policy Iteration (PI) is similar to VI, however, instead of optimizing utility, it optimizes the policy (all while applying the Bellman equation). How this works is the algorithm will choose an arbitrary policy and then enumerate the utility at all the states under that policy. It will then iteratively improve upon this policy. The policy iteration does not have a max operator and so the equation turns out to be linear and therefore solvable in a reasonable amount of time. We will revisit this when comparing computation time.

VI and PI are an indirect approach to learning as it is not directly a supervised learning approach mapping inputs to outputs. Q-learning is an algorithm that approaches this in a supervised learning way by ingesting <s,a> pairs and directly mapping those to <s', r> pairs. Q uses parts of the Bellman equation and is defined below

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

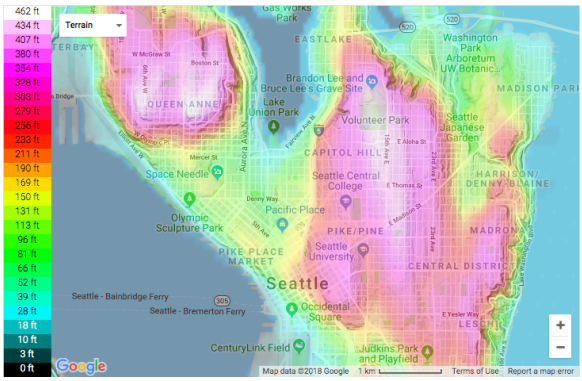
It is worth noting that when evaluating Q, we do not have access to the policy $\prod$ or utility U. We can derive $\prod$ and U from Q without needing the R or T function. Using Q, we define $\prod = \mathrm{argmax}_a Q(s,a)$ and $U = \mathrm{max}_a Q(s,a)$. Q-learning applies a Q-value (hence the name) and iteratively modifies the policy exploiting/exploring immediate/delayed rewards. This form of exploit vs explore sounds familiar looking back on our simulated annealing searching algorithm. We will discuss this further in our analysis of Q's performance on our MDPs.

Overview of Grid World (relevant to both MDP's)

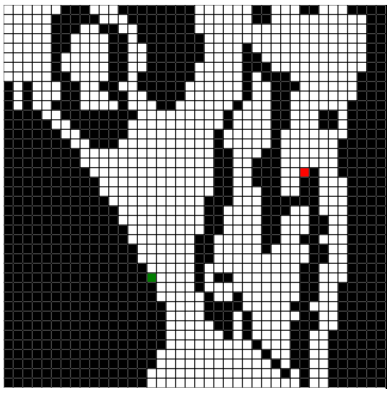
Each grid world is represented as a black and white grid. Black indicates the restricted states and white indicates the actionable states. The initial start state is marked with a **grey** ellipsis and the solution state is marked with a **blue** square. A **reward** of 100 for the solution state was used. A **negative reward** at each iteration (encouraging the agent to finish the game) of -1 was used. A discount factor (γ) of 0.99 was used. Printed beneath each utility function are the number of iterations they pertain to (e.g. i=5 signifies that policy was taken at 5 iterations).

Elevation Gain – Hard MDP

For our first MDP, we will consider a topographical map as our grid world. What we want to solve in our grid world is finding a path of least resistance in terms of elevation gain. This is a particularly interesting problem for runners. This appeals to me personally because at the moment I am training for a marathon. I also happen to live in Seattle, which is a city with WAY too many hills. This makes running long distances extremely difficult because almost no matter where you run, you are bound to encounter daunting hills that will ruin your ability to training in a consistent manner.

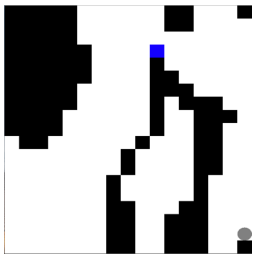


The topographical problem poses an interesting perspective on this problem because it introduces degrees of negative reward. The increase in elevation would be scrutinized in evaluating a specific negative reward in regard to the incline. The transition state would capture the meaning of this change in elevation. Further, if the start state occurs at a high elevation, the game is incentivized to remain at the top of the hill and stepping within the bounds of its elevation. For the sake of brevity, we will reduce the scope of this problem to evaluate steep elevation increase as a binary measure of negative reward. This problem could also be extended to require that the agent stays alive for a certain number of states (to capture the meaning of running distance requirements) as well as favoring new states over visited states to encourage taking new paths. This would be an interesting case to explore further but for now is out of the scope of this assignment.



One has to be strategic in the route that they choose, and there most certainly would be a market for an app that could efficiently generate several *unique* running routes at increments leading up to 26 miles (and perhaps even further). In an attempt to recreate this topographical map, we will use a grid world of binary placeholders for each state. 1 (black) will indicate areas the agent is not allowed to enter while 0 (white) will indicate areas the agent is allowed to enter. The grid world will represent water and steep incline as 1. Reason being, a runner will not want to enter the water or run up a steep hill. Start state is indicated as red, end state green.

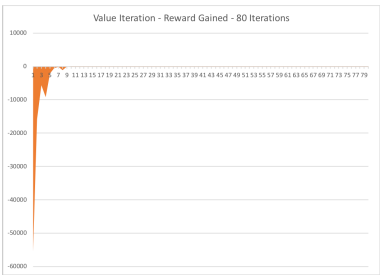
We will run value iteration, policy iteration, and Q-learning on this grid and analyze how each algorithm performs.

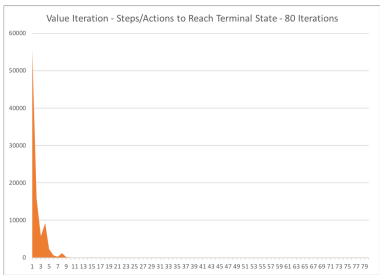


Quick Note: as I began development, this grid proved to be way to hard computationally, so I zoomed in on the map and ran it on a subspace. The new grid can be seen to the right. The grid dimensions are 17x19 with 215 white states and 108 black states. The starting position is at (17, 1) and the final state is at (11, 16). The agent was unable to generate a policy in a finite amount of time when increasing the parameters past 17x19 and tweaking the placement of the solution state. Mostly, policy iteration failed due to time complexity. When expanding the grid past 17x19 or, more frequently, when moving the solution state further away and into more open area, we saw the process timeout. For example, value iteration was able to find a solution with the **solution state** at 7x4, however, policy iteration failed every time. This was due to the large number of available states to explore and the agent was unable to find a valid policy in the given number of iterations. For this reason, it was necessary to zoom in on our original grid and move the solution state closer.

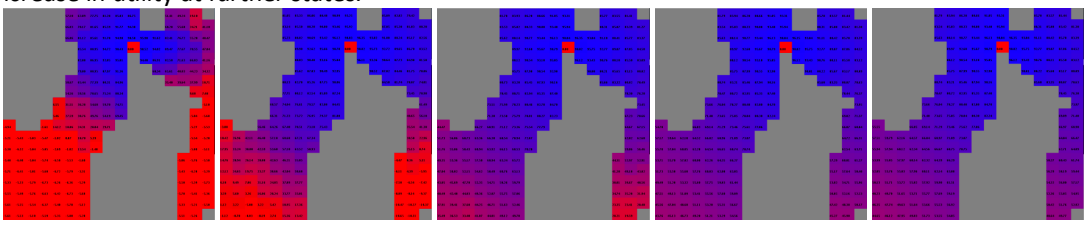
Value Iteration

The reward plot suggests a significant negative overall reward up until about 5 iterations. By 10 iterations the algorithm fully converged at a **reward** of ~73 on average. The number of **actions** mirrored this behavior and converged by 10 with roughly 27 steps on average (fluctuated between 20-30 up until 80 iterations). Final reward of 100, with a -1 **reward** at each interim step, (27 steps is -27) gives us -1 * (27) + 100 = 73 final reward, this follows intuition. The spikes in our data are expected given the probabilistic nature of the problem at hand.





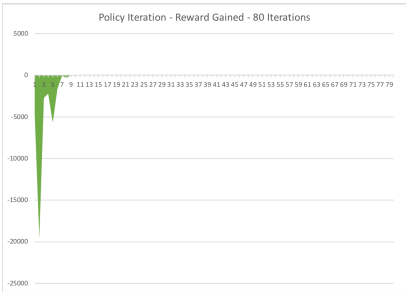
Below we can see our policy at each given iteration. Within each state is printed the utility of that given state. As expected from our definition of the Bellman equation, optimal utility is projected outward from the solution state. The lower the utility the redder the state appears. The **utility** in this policy is interpreted as the reward in that current state plus the expected reward from there forward. It is easy to see the convergence in our policies in the plots below at i=20 and on. Analyzing the differences between i=15 and i=20 closer, around the epicenter the utilities are similar. The further we get away from the solution state the utilities diverge, and we see that summation taking effect and is reflected in larger increase in utility at further states.



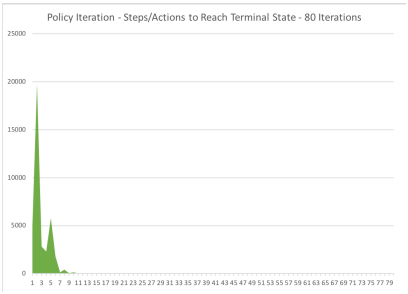
i=5 i=10 i=15 i=20 i=40

Policy Iteration

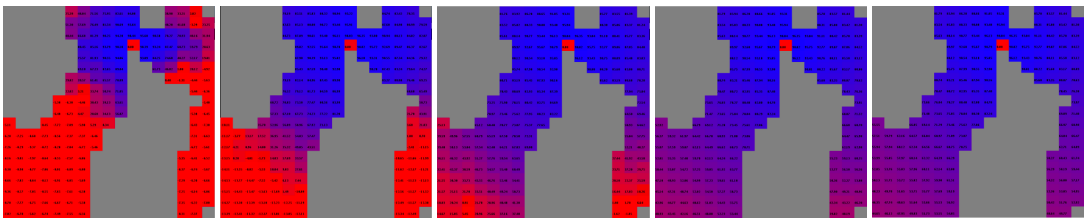
The reward plot suggests, similarly to value iteration, a significant negative reward again until about iteration 5-7. However, the reward here is far greater than that of value iteration. On average by about 10 the algorithm fully converged at a reward of ~72, which was on par with value iteration. The number of actions converged to ~28. This is slightly worse than value iteration. Though, each of these algorithms fluctuated a decent amount and they are extremely close so a definitive winner cannot be drawn from these figures.



Policy iteration requires more computation and we expect the function to converge with fewer iterations, since it is evaluating and processing the information at a given policy. Since it converges at the same or slightly fewer iterations, this is consistent with what we expected. Policy iteration also performed much slower than VI, which we will see in our final analysis.

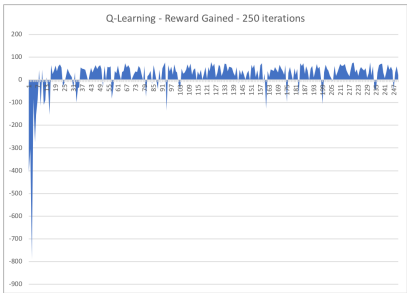


Below are the final policies at their respective iterations. Matching our previously mentioned figures, we can see the policy converging at a slightly slower rate than VI. However, by i=20 the overall utility from our Bellman equation matches that of VI sufficiently.

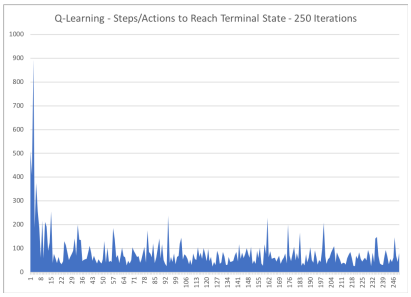


i=5 i=10 i=15 i=20 i=40

Q-learning

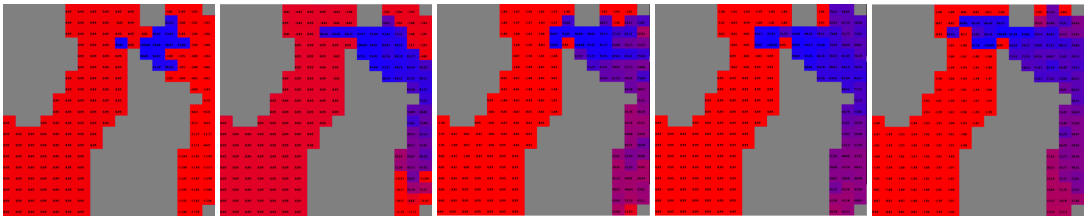


The reward plot in Q-learning suggests that the behavior overall is sporadic. "Convergence" is tricky since it fluctuates significantly starting at iteration 16. From here the algorithm averages a reward of ~30 and number of steps at ~70. This is significantly worse than both VI and PI. It converges at a less optimal reward/step and after more iterations. Though Q-learning converged far slower in terms of iterations, it's computing time was far less than that of PI and VI.



Q-learning, like we mentioned before, trains an algorithm by mapping <s, a> pairs to <s', r> pairs. Our agent wonders through this grid in a probabilistic fashion and so the sporadic fluctuation of Q's reward/step plots are warranted. Since Q-learning is agnostic to any policy ($\prod$) or utility (U), it makes sense that it converges much slower than VI and PI. This is also attributed to the explore/exploit nature of Q-learning that we mentioned earlier. The initial Q-value set in our algorithm was -1 with a learning rate of 0.99. This prefers exploring upfront and overtime transitions to exploiting more.

As we continue to plot our policies into 100, 200, 300 iterations and so on, we see the utility of each state fluctuate as expected. Notice the difference in iterations to VI/PI (10-250). This is interesting because Q isn't learning from subsequent iterations and converging nicely quite like VI/PI. Seen below and in our graphs, Q-learning was iterated far higher than VI/PI to illustrate this inconsistency in utility.



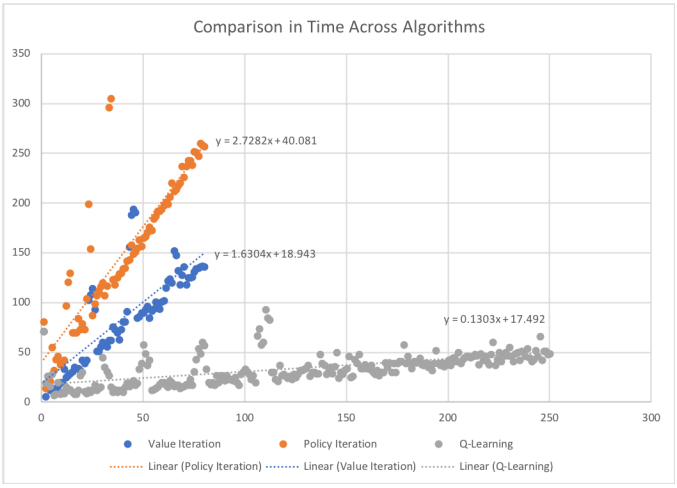
i=10 i=30 i=70 i=140 i=250

Analysis

Overall, number of states had the largest effect (and fatal effect at times) on policy iteration. This was no surprise and follows intuition when we revisit the algorithm of PI. PI first selects a policy, then optimizes the utility. If we are selecting a policy of a much larger array of states, then computation on that policy will grow significantly. As explained before, this limited how large our sample space could be and the reason we zoomed in on our grid world. It is worth nothing that value iteration and Q-learning were able to find policies on grids with much larger numbers of states.

When evaluating computation time across all three of these algorithms, Q-learning is most efficient, followed by VI, then PI. In the figure to the right we see the trends of computation time mapped against iteration. Each trend was linear with some random variability. This variability was consistent across all three algorithms. Recapping just

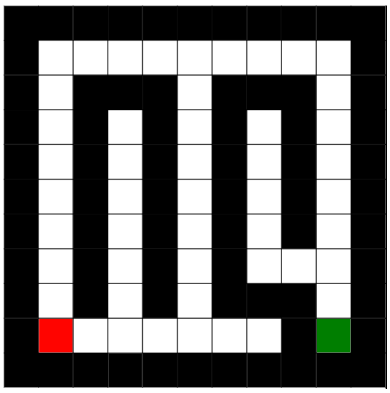
a bit , it is worth noting why PI performed so slowly. The computation required of PI caused it to rapidly perform slower, though we did see some significant accuracy out of it. While Q-learning may have been the quickest, it was the most inaccurate. Value iteration, overall, seems like the best win in each category. VI offered accuracy on par with PI and far better than Q-learning while still performing much quicker than PI. The accuracy gains compared to Q-learning here are a necessary trade-off for computation time.



Rationalizing VI's computation time is intuitive as well. VI selects an arbitrary utility and optimizes over iterations until an optimal utility is found. This requires less computation than selecting a policy and enumerating utilities in that policy over each iteration. As we can see, VI's approach scales much better with a large number of states.

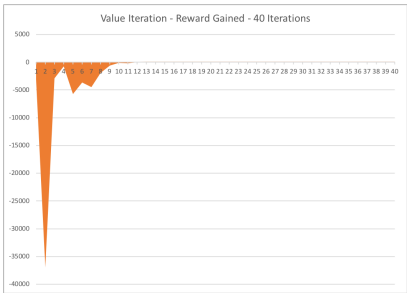
Maze – Easy MDP

For our second MDP, we will consider a maze as the grid world. Maze's are a clever and realistic application of an MDP and can vary in complexity. The complexity of our maze is not too high since we are dealing with a rather small maze, though we could easily scale the complexity by increasing the dimensions of the world. As an explorative exercise one could show how our algorithms perform on more complex mazes. This grid's dimensions are 11x11 with 50 white states, and 71 black states. The **starting position** is at (2, 2) and the **solution state** is at (10, 10).

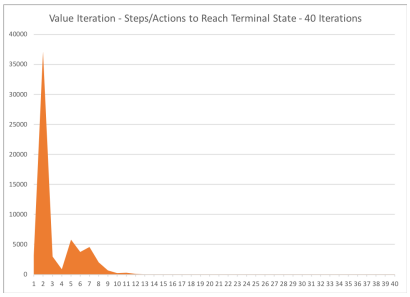


This problem is interesting and differs from our previous example because it provides a more restrictive world for our agent to live in. The walls guide the agent, to a certain extent, towards the **solution state**. Whereas in our last problem, the world was represented as a wide-open field where the agent could roam in any direction and has little indication as to which direction is the correct one.

Value Iteration

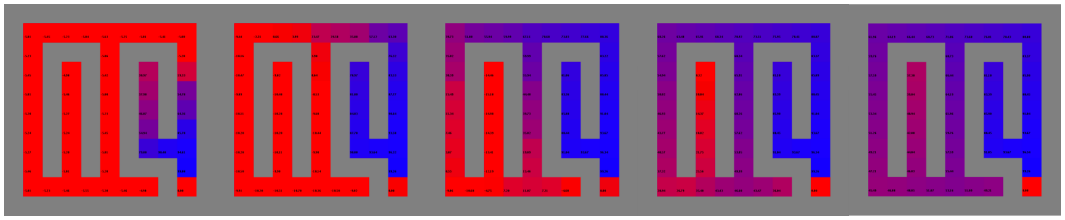


The reward plot suggests a significant negative reward of up until 9 iterations. By 12 iterations the algorithm fully converged at a reward of ~68 on average. The number of actions converged by 14 iterations with about 32 steps on average (-32 intermediate negative rewards, 100 final reward, (-1 * 32) + 100 = 68 (on average). The spikes in our data are more dramatic than with our pervious MDP. This is most likely due to the dramatic false paths in our grid layout. With mazes in particular, there are full paths that lead to nowhere. This makes sense when looking at our policy plots. The states in the first false path have the lowest utility values.



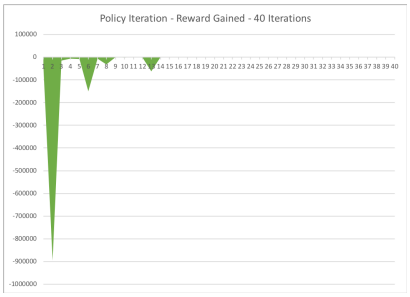
When analyzing our policy maps, the states in the farthest right false path score with a high utility. This makes sense why this algorithm would capture this false path well because it lies so closely to the solution state. Notice the far-left column/path and the middle column/path. They equally have a mild utility value (shade of purple indicates this). This makes sense when looking over the Bellman equation. The utility is evaluated by the reward in that

state plus the discounted sum of state transition and utilities of further states. Neither of these paths seem optimal over the other and so this may be what's affecting the indecisiveness of our utility values in each of these places.

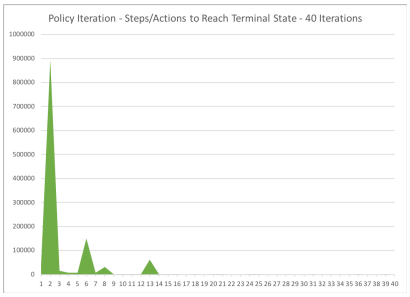


i=5 i=10 i=15 i=20 i=40

Policy Iteration

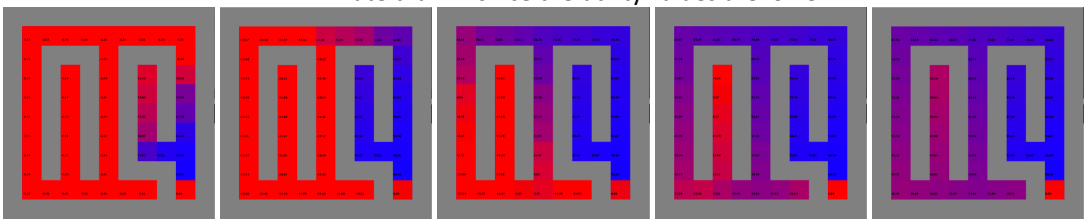


The reward plot suggests a significant negative reward only for the first three iterations, and then for iterations 6, 8 and 14. Otherwise, PI has relatively high reward. After 14 iterations (on average) PI converges at a reward of ~70, which beats VI. Number of actions converges at around 30. This behavior is much more sporadic than that of VI. This sheds some light on how PI and policy optimization (as opposed to utility optimization) performs on smaller maze-like grids such as these. PI retains information of previous iterations and leverages that in computing the next iteration in finding the solution state.



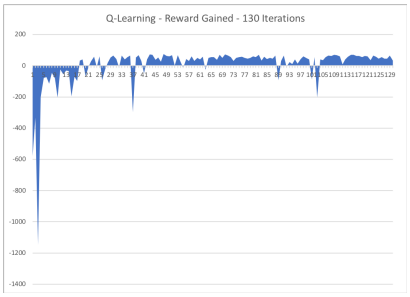
In our previous MDP we were dealing with significantly more states than here. Computation time for PI here did not perform much worse than VI (comparatively speaking). And so here with a smaller MDP we see gains on average in PI with respect to reward/step count with a lower tradeoff of computing cost.

Below we see the policies for PI which look nearly identical to VI's at convergence. This is expected on a smaller scaled grid. If we scrutinize the policy at i=10 we can see an example of how PI's converging at a slower rate than VI since the utility values are lower in PI.

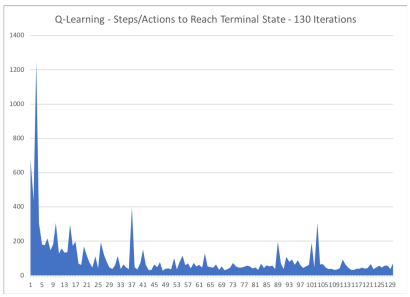


i=5 i=10 i=15 i=20 i=40

Q-learning



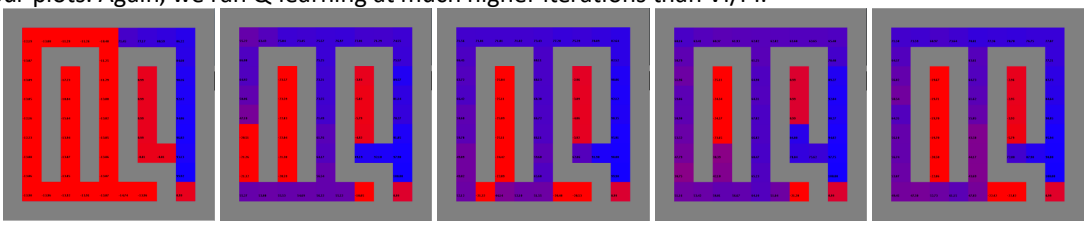
The reward plot in Q-learning suggests, much like for our previous MDP, sporadic behavior. Though it is worth mentioning that our plot in this MDP is tamer, which is most likely due to the size of the grid. We see convergence at iteration ~17 on average, just one iteration less than our last MDP.



However, specifically with Q-learning the runs are sporadic and have a larger standard deviation that VI/PI (so this average can be taken with a grain of sale). Our plots indicate a convergence of 34 for reward and 66 for steps. This

outperforms our previous MDP by about 4 steps. Note: parameters used for our first MDP were repeated here in terms of initial Q-value and learning rate.

As we plot the policies for Q-learning on the maze, there's a dramatic increase in utility scores across all out states. Further, i=100 has significantly better utility scores than i=130, perfectly highlighting that fluctuating behavior we see in our plots. Again, we ran Q-learning at much higher iterations than VI/PI.

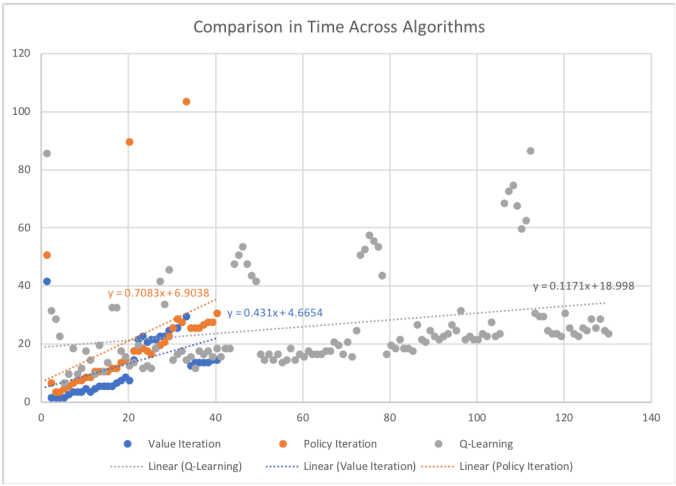


i=10 i=40 i=70 i=100 i=130

Now that we've seen Q-learning for both MDP's, let's circle back to the implementation of Q-learning mathematically and tie it in to our examples. Q is computed recursively, so that any Q is equal to the current reward plus the summation of transitions out from that state multiplied by Q' (defined above). When Q converges, mathematically what is happening is we are finding the true Q function which is only possible under certain conditions. Those conditions are that we have a deterministic MDP (yes), immediate reward value is bounded (yes), the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often[1]. By the third point, it is stating that over time in a given state, our agent will perform a given action in that state with nonzero frequency as our action sequence approaches infinity. This behavior explains why Q fluctuates as our iterations approach infinity, since it is exhausting actions at states.

Analysis

In our analysis of our previous MDP, we pointed out how the grid size affected policy iteration. When plotting the algorithms computation time in the maze experiment, we prove this theory correct. PI does significantly better and on par with VI. However, Q-learning is still the clear front runner in this category, though sacrificing the optimal final policy.



Due to PI's ability to compute in similar time to VI, we can conclude that PI's computation time is linearly related to the number of states in the grid. Do note, however, that VI did still slightly outperform PI. These two are close enough here to not conclude VI or PI to be the better solution when working with a smaller grid size, since they both performed similarly in generating an optimal policy as well.

Comparing back to our first MDP, the slopes of these lines of best fit for VI and PI are significantly faster (slopes: VI – 1.6x, PI – 2.7x, Q – 0.13x). Q-learning actually does somewhat on par and the standard deviation here is worse. This suggests that computation in Q-learning isn't affected by grid size (not as dramatically at least). This is something to consider when dealing only with larger grids.

Comparative Analysis

For the first MDP, the path from the start state to the solution state is 19 steps, which would be considered a perfect score. For the second MDP, the path from the start state to the solution state is 23 steps, which would be considered a perfect score. The first MDP is considered "harder" because it has more states overall, requiring more computation. This leaves room for error in that the agent can get mislead much easier. Comparing these two side

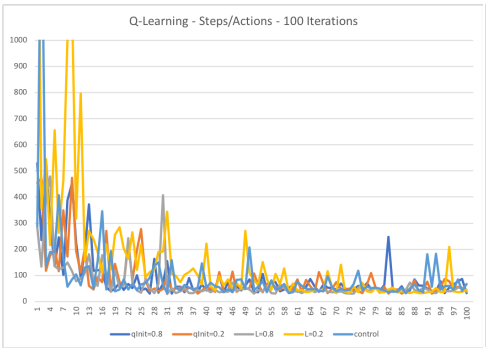
by side we see that it is important to be selective when choosing your algorithm. If your grid space is too large, one may have to compromise accuracy for the ability to actually compute a policy. Having said that, it would be nice if we didn't have to compromise too heavily on our accuracy. So now we will look into tuning parameter in Q to yield a higher accuracy.

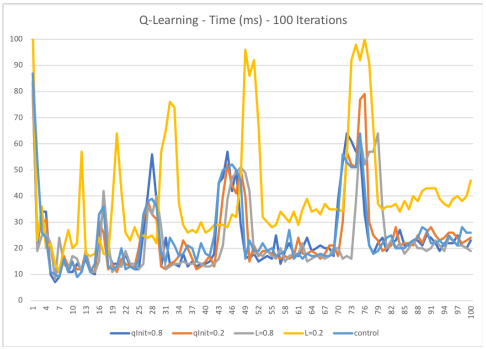
A Deep Dive in Q-Learning Application

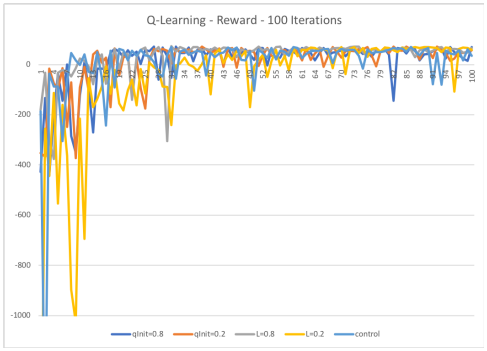
Now we will tune parameter in Q to see if we can't improve our accuracy. We will consider two parameters in this study, the initial Q-value and the learning rate. We will apply these to the maze grid to compare results. Once we land on an optimal set of parameters for this, we will apply them to a much larger complex maze. This way we accomplish Q-learning parameter tuning, test it on both easy and hard grids, and see how we do on a larger maze (just because I'm curious of that over our topographical experiment).

The learning rate (L) controls Q's decision over exploration and exploitation. In the first two examples, this value was set to 0.99. For clarity, L=1 only explores and L=0 only exploits. Exploring means choosing an action at random to learn something new about the grid. Exploiting means choosing what the agent believes is the optimal action at a given state. It is worth noting that L will decrease over time, meaning as it approaches infinity it will prefer exploitation to exploration. This follows intuition since by that point it will have seen all states and determined which actions follow the optimal policy. Values explored for L include [0.8, 0.2].

The initial Q-value is related to the learning rate in that is initializes the exploration/exploitation. Then the learning rate will demonstrate how it changes over time. To illustrate this, a low initial Q-value with a high learning rate will set exploitation high and will remain high. A high initial Q-value with a low learning rate will set exploration high and will remain high. Values explored for initial Q-value include [08, 0.2]. We have an idea for how initial Q=0.99 and L=0.99 performs based on our previous examples, though we will use them while running the other as a control variable. When testing values for L, initial Q = 0.99 and vice versa. Also, for consistency, we reran the 0.99, 0.99 again for our controlled variable.







The actions graph is not so easy to read as there is a lot of traffic. Comparing only two lines at a time makes it much easier to see what is going on. Though we can make out that for L=0.2 takes several more actions to complete than others. We can see a table of convergence to the right. Each of these were run five times and the averages we taken. We can see the average of actions and rewards for all the iterations past their point of convergence. The graphs also depict the same information but it is easier to differentiate with the table.

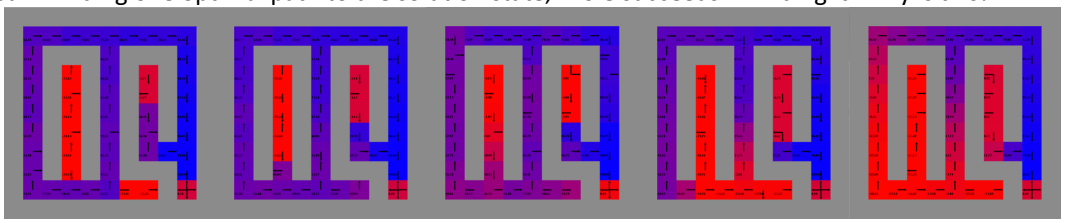
	iterations	actions	rewards
qInit=0.2	17	59	41
qInit=0.8	17	65	35
Control	19	62	38
L=0.8	31	52	48
L=0.2	32	72	28

Interestingly, a decrease in the learning rate had a much stronger effect than decreasing the initial Q-value. Decreasing the learning rate caused the graph to converge at a significantly slower rate. This is easily visible in the graphs depicted by the **yellow** line. The lowest learning rate (L=0.2) converged much slower. Recapping, a lower learning rate means the algorithm explores less and exploits more. This had a negative impact and shows the importance of exploration vs exploitation. A slight decrease in exploration, L=0.8, depicted by the **grey** line

performed quite well. Even though it required far more iterations to converge, in the long run it completed the maze in far fewer actions. This is a very interesting discovery in our data and implies that this parameter tuning improved on our original policy.

A conclusion that can be drawn from this is that there isn't anything to gain from starting at a lower initial Q-value. This should follow intuition as well. We would want our algorithm to being with the maximum exploration since it starts out knowing very little about the grid. To exploit immediately would result in a policy of complete indecision. To prove this, I ran the program with L=0.00 and it completed with a policy full of red states.

Below we can see the respective policies of each parameter set. In each square we can see the utility score along with the optimal action for that state. The policy at L=0.8 is extremely interesting when we analyze what exactly it is doing. At first glance, it has more red spaces and seems like the less optimal policy in comparison to qInit=0.8 and our control. And when evaluating the utility of the entire policy, this is true. Though when we are only interested in finding one optimal path to the solution state, L=0.8 succeeds in finding it. Why is this?



qInit=0.2 qInit=0.8 control L=0.8 L=0.2

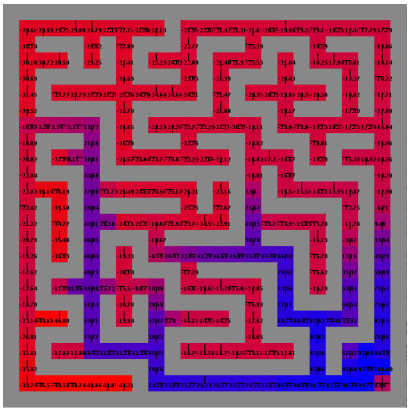
Let's split this maze into two paths. The following bullet points should be visualized as a decision tree of states and actions. Each action has equal probability. For simplicity, every action below has binary probability P(0.5)

- If we begin by going up
  - at state (5, 10) we have the decision of going (incorrectly) down or (correctly) right.
- If we begin by going right
  - At state (3, 1) we have the decision of going (incorrectly) up or (correctly) right.
  - At state (5, 1) we have the decision of going (incorrectly) right or (correctly) going up.
  - At state (5, 10) we have the decision of going (incorrectly) down or (correctly) right.

The first path has a 50% probability of getting on the correct path. The second path has a 12.5% (1/8) probability of getting on the correct path. The correct path is from (6, 10) onward. When choosing to lower exploration, our Q-learning algorithm will prefer optimizing the utility values in the path of higher probability. If we allow Q more wiggle room and push it to explore more, then naturally it will wonder into the second path and optimize utility in those states. This should follow intuition and is incredible to see actually functioning in the way we intend.

25x25 Maze

To the right is an example of running Q-learning on a larger 25x25 maze. Q tries to find only a select few optimal paths with the least amount of actions required. Policy iteration and value iteration, we unable to complete computation when finding a policy. The initial state was placed at (1, 1) and solution state at (24, 1). This policy was generated using qInit=0.99 and L=0.8. This policy was far better than that found with the control parameters.



Tying Q-Learning together

When wanting to find a single path to maximize reward in a large grid (that may be un-computational for VI/PI), we can run Q-learning and tune parameters (perhaps to slightly discourage exploration) to find this path. This has been proven through our experiments in parameter tuning on our 11x11 maze and well as our harder maze of 25x25.

[1] Machine Learning – Tom M. Mitchell