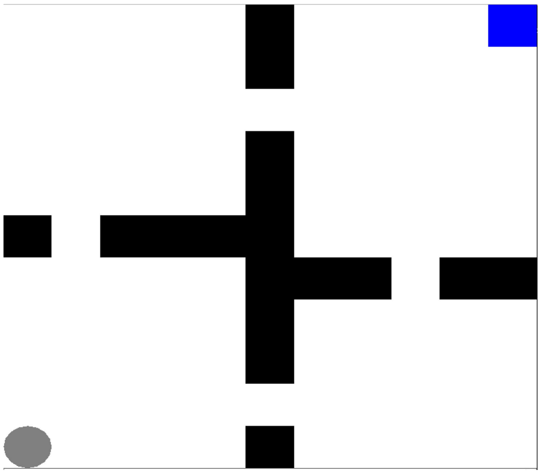
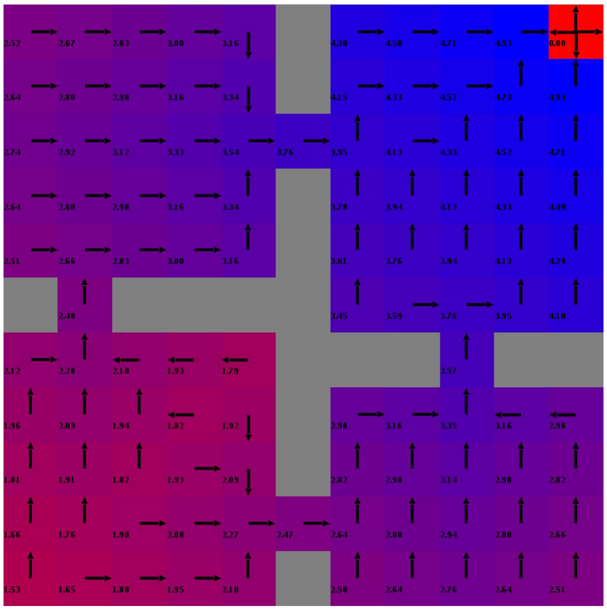
# CS 7641 Assignment 3:MDPs and RL

## Grid World

Grid world is a classic Markov Decision Process (MDP). The problem is to find the solution to a maze from any starting location and maximize the reward collected along the way by an agent to the end of the maze (known as the absorbing state). The reason gridworld is a MDP instead of a breadth/depth/shortest path search is that movement through the maze is modeled stochastically and there are rewards associated with arriving in each grid location. My particular instance of grid world was an 11x11 grid with 4 rooms. A visualization of the world is to the left. The blue square is the absorbing state and the grey circle is a hypothetical starting location of the agent. There are 104 unique states in this MDP; the white squares are states and the black squares are walls which the agent cannot move into. Movement through the space is stochastic; the intended movement occurs with 80% chance (10% for each direction perpendicular to the intended direction of travel). The reward for entering the absorbing state is 5 and every other state is -0.1.



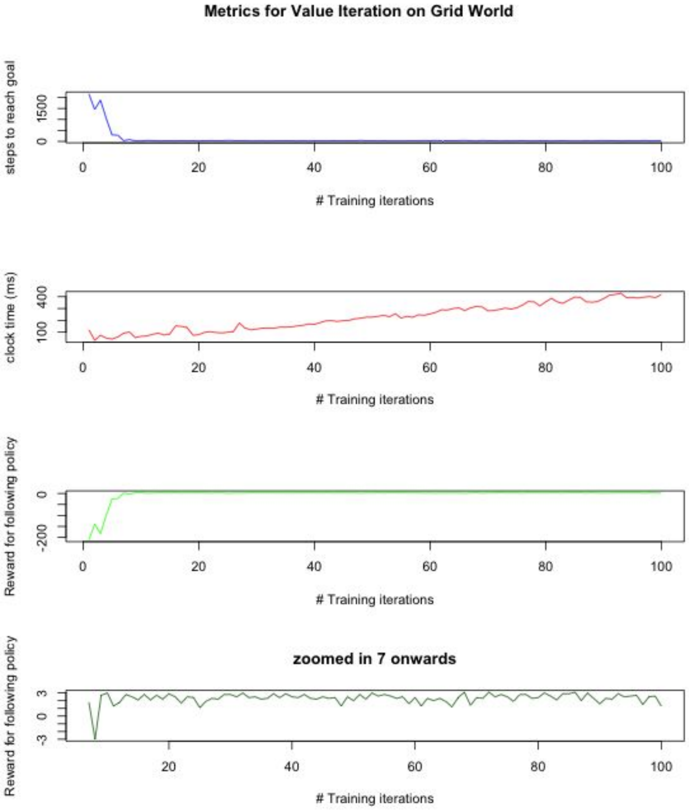
### Value Iteration



I first solved this MDP using value iteration and the algorithm converged in 26 iterations using a max delta of .001. A visualization of the generated policy is to the right. Max delta is the convergence criteria for the value iteration algorithm. When the maximum change in the value function is smaller than the max delta, value iteration stops running and returns a solution. The expected value of the reward from starting in the lower left corner as calculated by the value iteration algorithm is 1.53. Let's relate this value to the potential maximum reward for solving this grid.

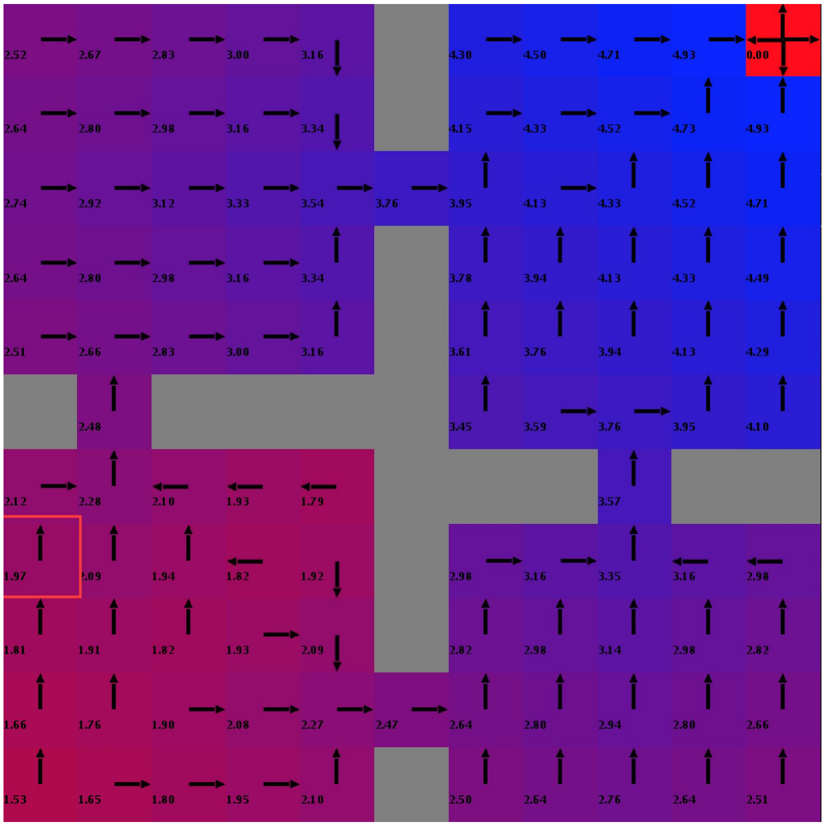
The minimum distance from the lower left corner to the absorbing state is 19 steps and is achieved by multiple paths. Therefore, the largest reward the agent could possibly accumulate in this grid is 3.1; 19*(-.1) + 5 = 3.1.

Manually calculating the "true" expected value of the starting location and comparing it to 1.53 would be quite difficult. For example, we know the probability of following one of the optimal paths perfectly with no unintended perpendicular movement is .8^(19)=.0144. This probability plus the reward would be incorporated into the estimate of the expected value of the lower left corner by multiplying .0144 by 3.1 and adding this value to the other path's total rewards plus those path's conditional probabilities. However, there are an infinite number of paths to the absorbing state, which each have some probability of occurring because of stochastic movement (even if it is infinitesimal). So it is difficult to compare 1.53 to the actual expected value, but it is likely a very very good estimate of it.



My next bit of analysis is about how value iteration performs by varying the number of iterations it ran for. Instead of letting it run until convergence under a max delta, I let it run for **X** iterations and tracked a number of different metrics including: "steps to reach goal", "clock time", and "reward for following the optimal policy generated after **X** iterations". "Steps to reach goal" is how quickly the agent arrived at the absorbing state by following the policy generated after **X** iterations. A visualization of these metrics is to the left; **X** was increased from 1 to 100 iterations.

The first thing to notice is that wall clock time increases linearly. This occurs because even though the updates to the values of states will be very very small in later iterations, we still perform the operation on all states. Another thing to notice is that steps to reach the goal and reward received converge around the 10th iteration. The minimum number of steps in the first time series is 21 (19 moves that occupied 21 states) and has a corresponding reward value of 3.1. This is the "best" case scenario and as discussed earlier occurs with a probability greater than .0144 (since there are multiple optimal paths). It is not surprising then that since the algorithm converges to the optimal policy at around 10 iterations we see a couple of observations in the time series that achieve 3.1 as their total reward. This is due to the sample size of gridworld episodes that were run with an optimal policy (about 90) and the fact that we can achieve a reward of 3.1 with a probability greater than .0144. The reason we do not always see 3.1 as the

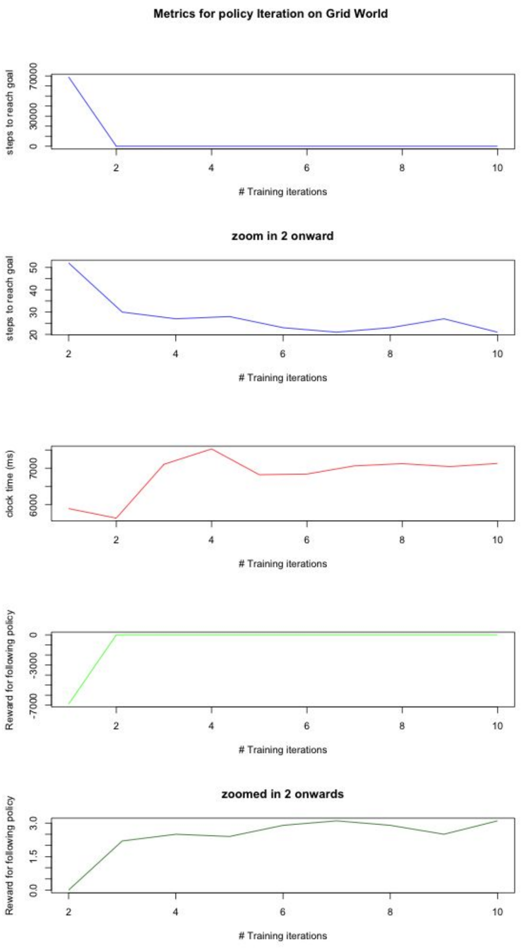


| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 2.52 → | 2.67 → | 2.83 → | 3.00 → | 3.16 ↓ | | 4.30 → | 4.50 → | 4.71 → | 4.93 → | 0.00 ↑↓←→ |
| 2.64 → | 2.80 → | 2.98 → | 3.16 → | 3.34 ↓ | | 4.15 → | 4.33 → | 4.52 → | 4.73 ↑ | 4.93 ↑ |
| 2.74 → | 2.92 → | 3.12 → | 3.33 → | 3.54 → | 3.76 → | 3.95 ↑ | 4.13 → | 4.33 ↑ | 4.52 ↑ | 4.71 ↑ |
| 2.64 → | 2.80 → | 2.98 → | 3.16 → | 3.34 ↑ | | 3.78 ↑ | 3.94 ↑ | 4.13 ↑ | 4.33 ↑ | 4.49 ↑ |
| 2.51 → | 2.66 → | 2.83 → | 3.00 → | 3.16 ↑ | | 3.61 ↑ | 3.76 ↑ | 3.94 ↑ | 4.13 ↑ | 4.29 ↑ |
| | 2.48 ↑ | | | | | 3.45 ↑ | 3.59 → | 3.76 → | 3.95 ↑ | 4.10 ↑ |
| 2.12 → | 2.28 ↑ | 2.10 ← | 1.93 ← | 1.79 ← | | | | 3.57 ↑ | | |
| 1.97 ↑ | 2.09 ↑ | 1.94 ↑ | 1.82 ← | 1.92 ↓ | | 2.98 → | 3.16 → | 3.35 ↑ | 3.16 ← | 2.98 ← |
| 1.81 ↑ | 1.91 ↑ | 1.82 ↑ | 1.93 → | 2.09 ↓ | | 2.82 ↑ | 2.98 ↑ | 3.14 ↑ | 2.98 ↑ | 2.82 ↑ |
| 1.66 ↑ | 1.76 ↑ | 1.90 → | 2.08 → | 2.27 → | 2.47 → | 2.64 ↑ | 2.80 ↑ | 2.94 ↑ | 2.80 ↑ | 2.66 ↑ |
| 1.53 ↑ | 1.65 → | 1.80 → | 1.95 → | 2.10 ↑ | | 2.50 ↑ | 2.64 ↑ | 2.76 ↑ | 2.64 ↑ | 2.51 ↑ |

reward from 10 iterations onwards because of the probabilistic movement that occurs *even* when we intended to follow the optimal path.

## Policy Iteration

I solved the same MDP using policy iteration with an inner max delta of .001. A visualization of the generated policy with a max delta of .001 is on the previous page (the algorithm ran for 8 iterations). The policy generated is the exact same as that generated by value iteration. The only noticeable difference is the square with the red border which is different from value iteration's estimate by .01. Again, I varied the number of iterations and tracked the same metrics I did for value iteration.

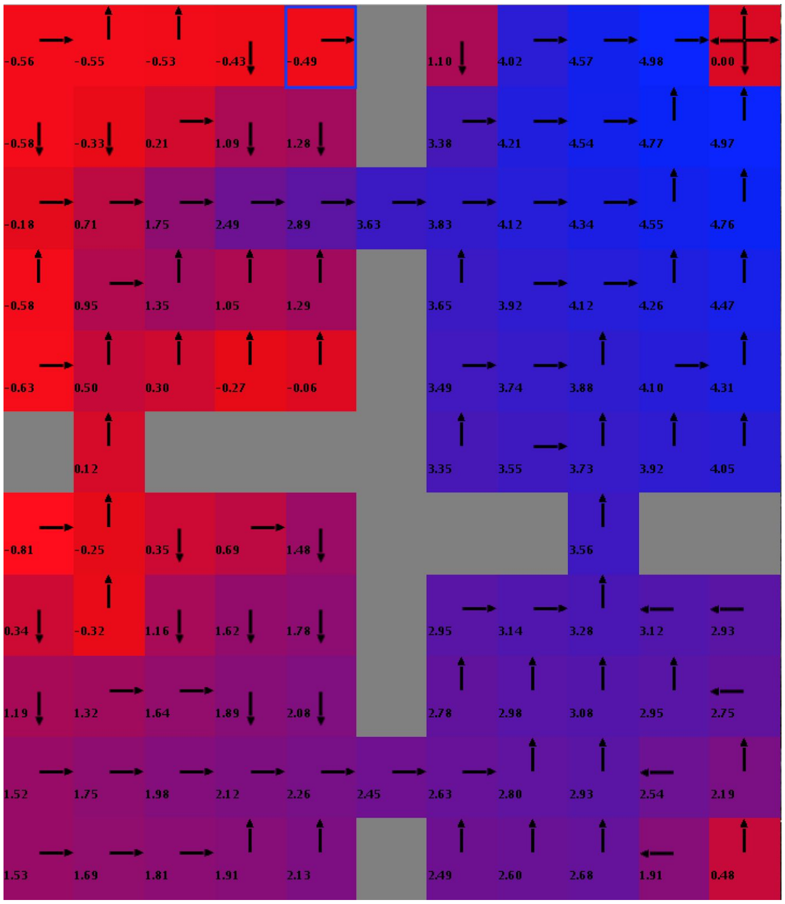


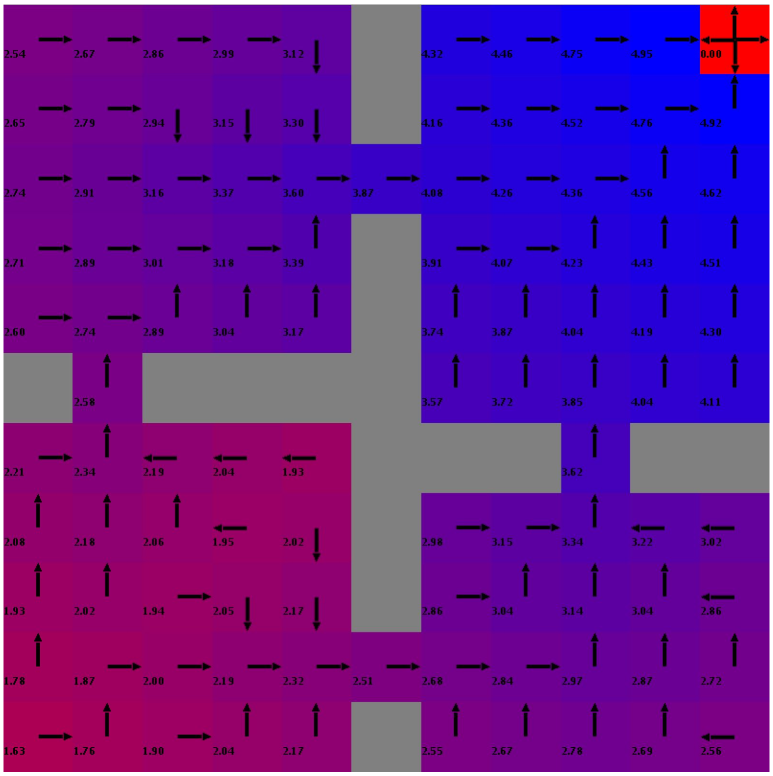
The results, visualized to the left, show the algorithm getting approximately the right solution around 5 or 6 iterations. Since we only run 1 through 10 iterations it is harder to see a linear trend in the wall clock time. It is worth noting that policy iteration converged in 7125 ms versus 107 ms for value iteration. Why is there such a huge difference in speed? For each iteration of policy there is an inner loop that updates all the estimates of the values for each cell. This step involves solving n equations with n unknowns. The implementation in burlap uses an approximation algorithm for this step and I set the convergence criteria to a max delta of .001. It is interesting to note that the expected value of each state likely does not need to be estimated to a degree of .001. Why? Even a poor estimate of the expected value of each state might update the overall policy and move the algorithm forward towards an optimal policy.

In this vein, I re-ran my original policy iteration algorithm again with a inner max delta of .01 (10x larger). It converged in 8 iterations, and generated a policy with paths that could reach the goal with 19 moves. However, its expected value estimates of each state were not the same as those estimated by the value iteration or policy iteration with inner delta of .001. So all and all it got the job done as well as value iteration and policy iteration with a inner delta of .001. Still, it was not as fast as value iteration and took 2349 ms to converge.

## QLearning

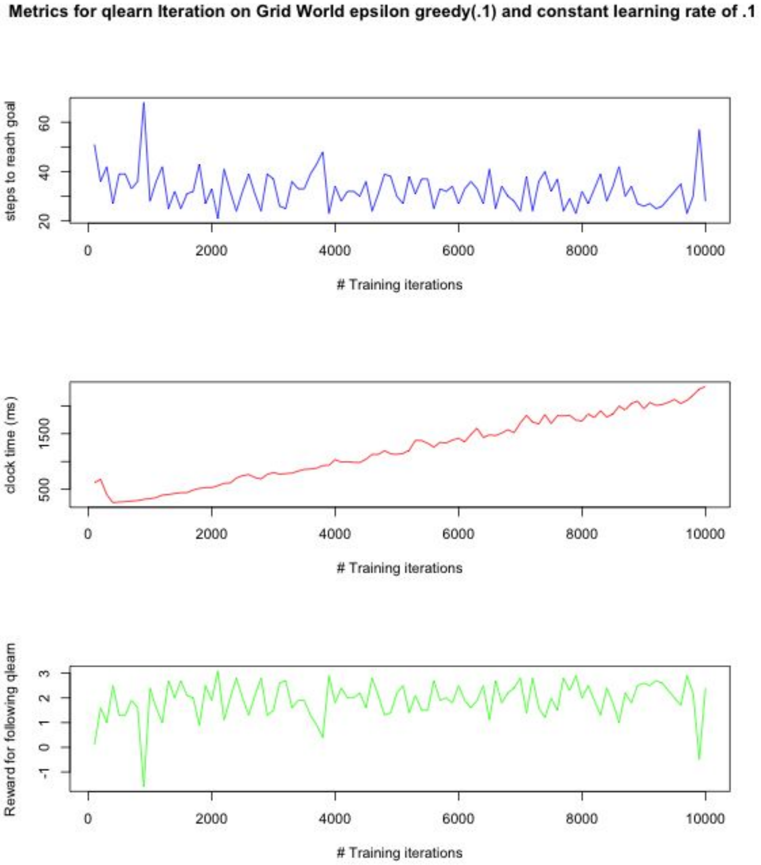
I chose QLearning as my reinforcement learning algorithm and ran it for 10,000 episodes with an epsilon greedy exploration strategy with an epsilon value of 0.1, which means that with probability 0.1 the algorithm takes a random action instead of the calculated optimal action. The generated policy is displayed on the next page in the right column. Of note is that the policy generated contains an optimal path from the lower left corner. However, less visited states such as the one I've highlighted in blue have completely incorrect policies and poor estimates of expected value (comparing the blue squares -0.49 to the estimate of 3.16 in value iteration or policy iteration). Why are these values so off and why is the policy wrong? Well when I ran this instance of QLearning the algorithm always started in the lower left corner. Thus it only really got the chance to explore optimal strategies from this location. A better solution to finding optimal policies for every location would be to restart QLearning from different initial states. However, this could be a waste of time. Let’s say we were using Q-Learning to learn to play the game 2048 (which if you are not familiar with you should check it out). There would be no need to figure out the optimal policy for a board that was already played in a seriously sub-optimal way. Said in the context of this MDP, the blue square is a spot you are not likely to end up in if we are always starting in the lower left corner, so why learn anything about it at all? So this highlights one aspect of QLearning. It all depends on the context of what we want to learn. If we only need to solve mazes from the lower left corner, then QLearning with Epsilon Greedy and starting from the lower left corner works great. If we need a general solution for each square, then this configuration of QLearning does a very poor job. I decided to run QLearning with a completely random policy for 10,000 episodes to show the contrast; a visualization of the policy is to the below.





Since we ran so many episodes of QLearning we see an optimal policy from the lower left corner with an expected value pretty close to our best estimate of 1.53 from value and policy iteration. In addition, we see approximately correct estimate of the value of states (and optimal policies as well) far from the lower left corner that the optimal path is unlikely to visit.

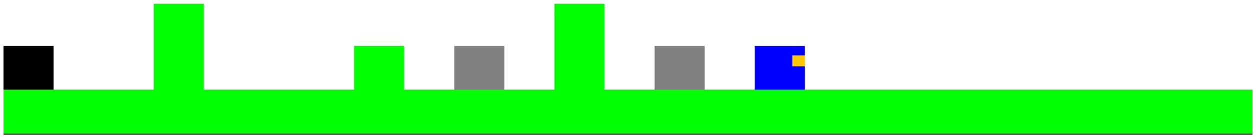
For QLearning I also varied the number of episodes run and tracked the same metrics I did for value iteration and policy iteration. For each epsiode, QLearning was run with a epsilon greedy exploration strategy with an epsilon parameter of 0.1. Since QLearning uses an exploration strategy, steps to reach goal and reward are both much less consistent than the time serieses for both value and policy iteration. Since I ran 100,000 training epsiodes, you see clear linear growth in the runtime of the algorithm.



## Block Dude

This is introduction section is paraphrased from the burlap documentation. Block Dude is a Texas Instruments calculator puzzle game. The goal is for the player to reach an exit. The player has three movement options: west, east, and up if the platform in front of them is only one unit higher. However, because of the landscape, the player will never be able to reach the exit just by moving. Instead, the player must manipulate a set of blocks scattered across the world using a pick up action (which raises the block the player is facing above their head to be carried), and a put down action that places a held block directly in front of the agent.

Block Dude has a very large state action space that makes planning problems in it difficult. States are encoded by the player's position, facing direction, whether they are holding a block, the place of every block in the world and an int array specifying the map of the "bricks" that define the landscape. A visualization of level 1 of block dude is below. The agent is a blue square with a gold eye indicating the direction it's facing; bricks are rendered in green; the exit in black, and movable blocks in grey.



In contrast to GridWorld, Block Dude does not involve any stochastic movement. Therefore, while we can model it as a MDP, it could also be solved with dynamic programming solvers. Block dude level 1 has 804 states. I used level 1 for all of my analysis. Block dude level 2 has 230,995 states, and would have been useful for comparing algorithm performance on MDPs with many many states (as compared to GridWorld), however, the runtime on my macbook proved too infeasible. Therefore, while level 1 is not a huge MDP it is definitely larger than my GridWorld MDP.

Block dude's reward function is -1 for every move. Therefore, we are essentially encouraging the algorithms to learn how to win (reach the black square which is the absorbing state) as fast as possible.

## Value and Policy Iteration

I solved Block Dude Level 1 with value iteration, policy iteration, and QLearning. I varied the convergence criteria for policy and value iteration. The table below shows the results of my experiment. All the algorithms generated a policy which solved block dude level 1 in 19 steps. The solution was: [west, pickup, west, putdown, up, up, west, pickup, west, west, up, west, west, putdown, up, up, west, west, west].

| **Algorithm** | **Total Steps to exit** | **Number of Iterations or Learning Episodes** | **Clock Time (ms)** |
|---|---|---|---|
| Value Iteration (max Delta = .001) | 19 | 688 | 10987 |
| Value Iteration (max delta = .01) | 19 | 459 | 7429 |
| Value Iteration (max delta = .5) | 19 | 69 | 1890 |
| Policy Iteration (inner max delta = .01) | 19 | 19 | 20968 |
| Policy Iteration (inner max delta = .5) | 19 | 25 | 11193 |
| QLearning | 19 | 181 | 151432 |

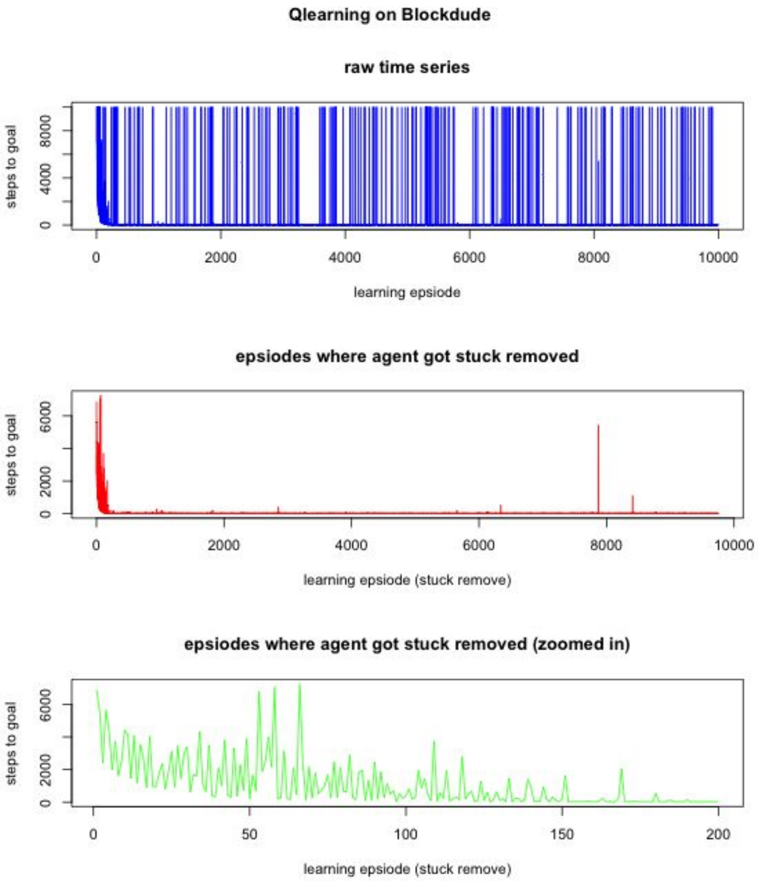
What's interesting in the table above?

- Value iteration with a relatively high max delta of 0.5 gives an optimal policy for the MDP. Why? In this case, since there is nothing stochastic about movement and since we are always starting in the same place, we really only need to solve a shortest path problem. Therefore, getting the actual expected value of the different states perfect does not matter.
- When I increased the inner max delta for policy iteration (meaning the inner value iteration step would converge faster), the algorithm still generated an optimal solution, did so faster, but took more total iterations to do so. Why? Policy Iteration makes rough approximations of the expected utilities to updated the policy. By approximating the values less, an approximate policy was generated quicker which then influenced the next value iteration step. Thus each approximate policy took a shorter time to estimate, but more needed to be estimated.
- The ratio of clock time for value iteration vs. policy iteration for my Grid World MDP when the max delta (.001) was set to the same value for each algorithm was 107:7125=.015017. In contrast as shown in the table above for max delta =.5, the ratio on block dude level 1 is 1890:11193=.1688. This is a huge increase in the ratio. While there are a number of factors that come into play here including the choice of max delta, it leads me to conjecture that when the number of states increase, policy iteration may overcome value iteration in terms of speed. More experimentation would be required to show this trend. As I discussed in the introduction to block dude, I wanted to run block dude level 2, but it proved to take too long to run on my mac.

## QLearning

I also ran QLearning on block dude level 1. While it was included in the comparison chart in the last section, it is worth discussing separately. I ran QLearning on block dude level 1 with an epsilon greedy exploration with an epsilon value of 0.1. In order for QLearning to work at all, I need to limit the episode length, so I set a max episode length to 10,000. This was needed because the agent can create situations where it gets stuck or does something so stupid that it would take an extremely probabilistic set of moves to recover from. For example, if the agent picked up the first block and moved it as far east as possible and dropped it, it would be very unlikely that the agent would produce the opposite situation and pick the block back up and move it back to the original location.

By limiting the length of episodes, QLearning avoids this situations where it would need huge amounts of steps to recovery



from and instead just starts a new episode. The visualization on the previous pages shows "steps to goal" vs. training episode. The first episode in which the agent reaches the goal took 6,888 steps which is a huge number of steps. What's interesting is that QLearning improves this policy iteratively. Over approximately the first 200 episodes (shown in green on the visualization on the previous page), QLearning learns a policy to reach the goal in 19 steps. How? By using the epsilon greedy exploration strategy. After learning a random path in 6,888 steps that reaches the goal, the algorithm probabilistically follows this path. By trying new strategies the algorithm improves. For example, pretend that the policy that took 6,888 steps started by going east over and over again before hitting the wall and then turned around went west and picked up the block. The next iteration of QLearning might forgo going east on the first move (because of the 0.1 probability of not going east) and go west and arrive next to the block. The old policy had that in this state the agent should pick up the block and then the agent follows the old policy's path for the rest of the episode. So QLearning slightly shortened the length of the path by skipping a stupid opening move, but followed the optimal policy by picking up the block. It then iterates on this over and over again until it learns the optimal policy.

The blue line in the visualization on the previous page also shows one of the short comings up the greedy epsilon policy. The blue visualization looks like a bar code since there are blimps of 10,000 steps in late learning episodes. Essentially, even after QLearning has learned an optimal policy, the agent will only probabilistically follow it and a single wrong move can cause a stuck situation or one that it does not have a plan for getting out of.

## Conclusions

By looking at two different sized MDPs, we saw that in general value iteration was faster than policy iteration and provided the same solution. However, as stated earlier, this fact likely depends on the number of states. In regards to QLearning, exploration strategy and starting location matter. We saw that in the GridWorld example if the algorithm was not "random" enough it would never visit certain states and learn their optimal policy. Overall, it was also shown that QLearning was slower than value iteration or policy iteration. QLearning demonstrated one interesting caveat that was not discussed in length. This caveat relates to the point I already made in this conclusion about exploration strategy. QLearning can avoid looking at all states and instead focus on the important ones while value iteration and policy iteration find an optimal policy for every state. I imagine that there are heuristic versions of value iteration and policy iteration similar to how A* is a heuristic on top dijkstra's algorithm which prevents searching the entire graph.