Lab 1

University of California, Los Angeles
EE209AS

James (Zach) Harris (UID: 105221897)
Hamza Khan (UID: 205223240)
Joshua Hannan (UID: 805221851)

`jzharris@g.ucla.edu`
`hamzakhan@ucla.edu`
`jhannan@g.ucla.edu`

0 Preliminaries

0(a) Links

Public Github repo: `https://github.com/jzharris/ECE209AS_Project_Submissions`

0(b) Collaborators

Our team was composed of three people: James (Zach) Harris, Hamza Khan, and Joshua Hannan. We only consulted the course notes to complete this project, since they were sufficient to complete the lab.

0(c) Aggregate Contributions

The aggregate contributions percent of each member was basically equal, 33%. During the project, we were always all three working at the same time, alternating between one person coding and the other two looking through notes and using the information to help the current designated coder. We determined our algorithms together and debugged together.

1 MDP system

1(a) State Space $S = \{s\}$

The size of the state space is is $W * L * 12 = 8 * 8 * 12 = 768$.

1(b) Action Space $A = \{a\}$

The size of the action space is 7: we have three rotations each for forward and backward movement, and one extra action for staying put (which doesn't allow any rotation).

1(c) Probability $p_{sa}(s')$

The function for calculating $p_{sa}(s')$ is defined in the code as "statespace.getPsa()". It gives a probability of $(1 - 2p_e)$ for the desired output location, and p_e for one direction to the left and right each (due to prerotation error).

1(d) Next state s'

The function "getNextState()" takes care of finding the next state by sampling from the specified probability distribution given based on p_e. To test this, we took 10000 trials with $p_e = .2$, all starting from the same state and taking the same action. We expected to see the percentage of trials to be 60% correct (the state we expected), and 20% each for the left and right random prerotations.

2 Planning problem

2(a) Reward $R(s)$

For the rewards, we created a Rewards class that holds the reward for each state, and has a "getReward(s)" function which returns back the reward.

3 Policy iteration

3(a) Action Matrix

We created a function called "getInitialPolicyMatrix()" that returns the best action to take given a state. It uses a basic policy that just tries to get to the reward without considering obstacles.

3(b) Robot trajectory 1

We wrote two functions to generate and plot the trajectory of the robot, called "getTrajectory()" and "plotTrajectory()".

3(c) Robot trajectory 2

Given $p_e = 0$, we plotted a trajectory starting from (1, 6, 6), which resulted in Figure 1.

3(d) Policy evaluation

We created a function called "policyEval()" which performed the actual policy evaluation. We did so using:

$V^\pi = T^\pi V^\pi$

V = $\sum_{s'} R + P\gamma V = diag(PR^T) + \gamma PV$

We did this iteratively by re-applying the bellman operator until the output V equaled the input V.

3(e) Trajectory value

We calculated a trajectory value of 4.396 using the following equation: $\sum_0^{max}[\gamma^i * V(i)]$.

3(f) Optimal Policy - one-step lookahead

We wrote a function called "policyRefinement()" that calculated the best policy given value V. We used the equation:

$\pi_{new} = arg_a max T V^\pi$

3(g) Optimal policy - iteration

We combined the policy evaluation and refinement functions to create the function "policyIteration()" that repeats until policy matrix doesn't change.

3(h) Trajectory - Optimal Policy

Using the optimal policy, we obtained the trajectory shown in Figure 2.

We found that the trajectory value in this case, using the same equation as before, was 38.35.

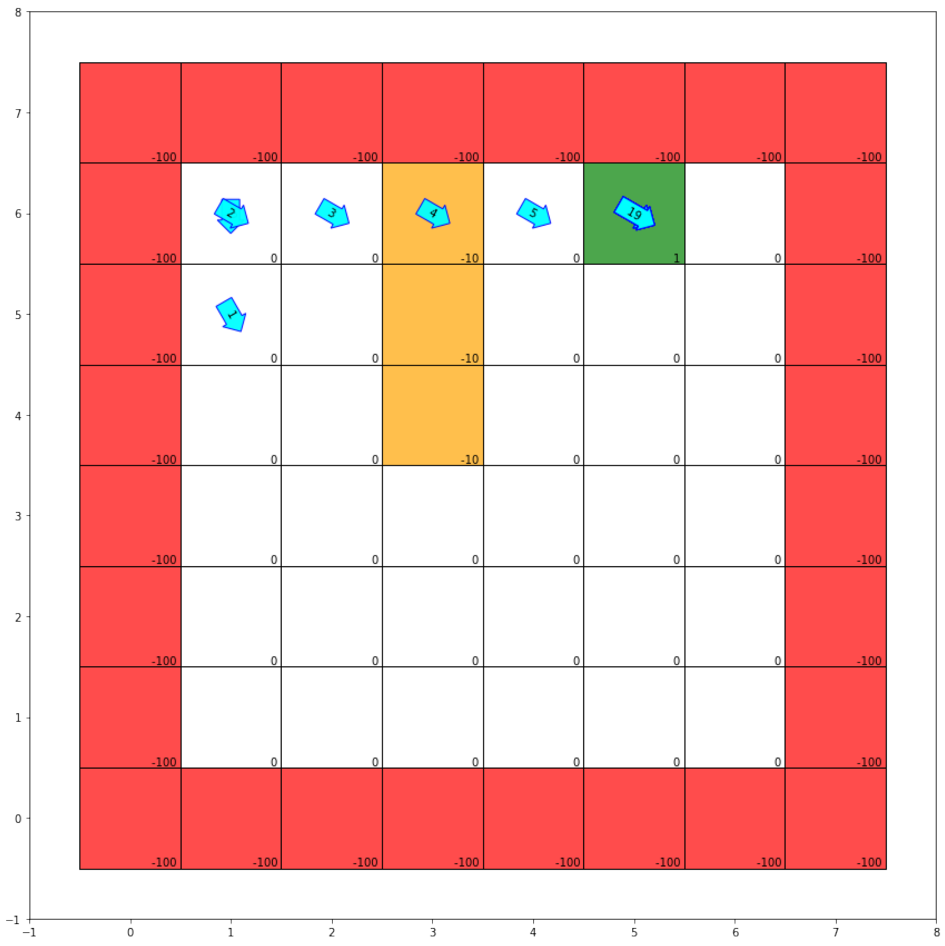

Fig. 1. Trajectory for initial policy.

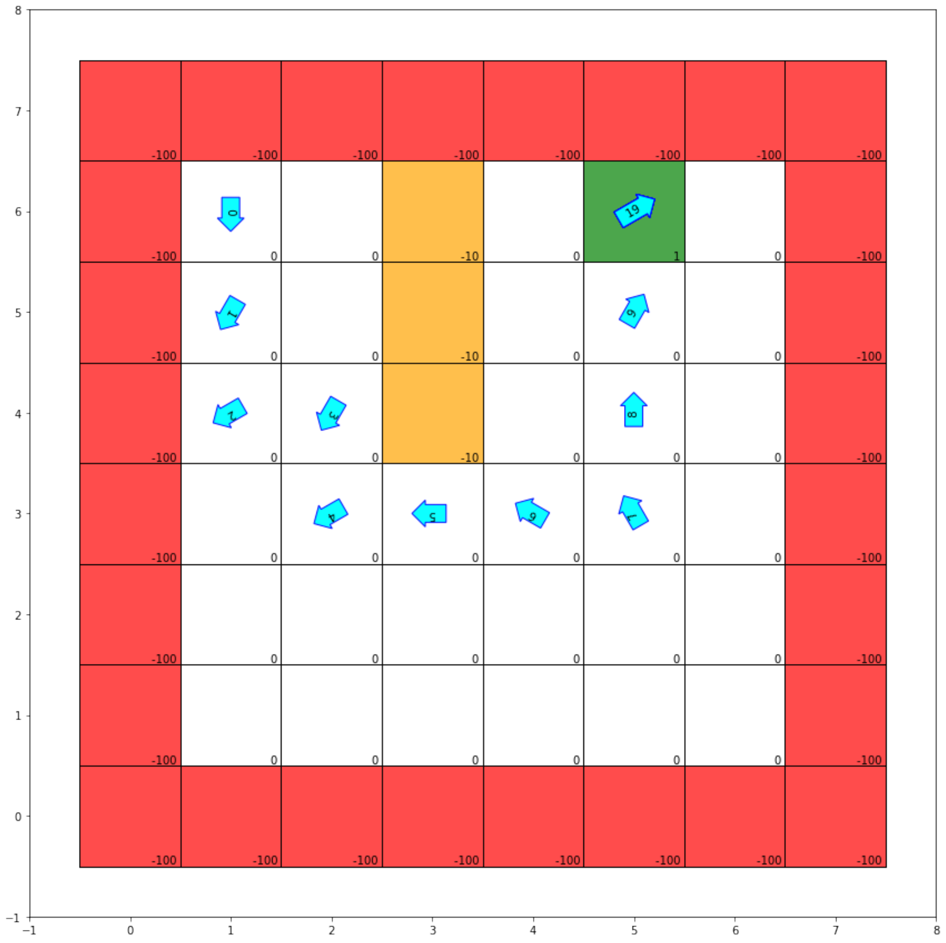

Fig. 2. Trajectory for policy iteration's optimal policy.

3(i) Compute time

Using a simple start and end time comparison, we found that the compute time was about 4.27 seconds.

4 Value iteration

4(a) Value iteration

We wrote a function called ”valueIteration()”, which uses an initial condition of $V(s) = 0\ \forall S$ and, given a policy vector π_0, computes the optimal policy vector via Value Iteration.

4(b) Trajectory

The trajectory is shown in Figure 3.

Policy iteration converged significantly faster (4.27 seconds) than value iteration (63.87 seconds). Additionally, the value of the trajectory that policy iteration calculates for its trajectory is higher (38.35) than the value that value iteration calculates (22.43). However, despite these differences, the actual trajectories calculated by the two methods are the same. The difference in the runtimes of the two algorithms is caused by value iteration taking more steps, each with a smaller gain–compared to policy iteration which takes fewer steps, each with a larger gain. Thus, value iteration can be stopped before convergence and have a fairly accurate trajectory, while policy iteration is likely to be very far off before convergence. Consequently, it can be useful to combine the two iteration methods.

4(c) Compute time

It took 63.87 seconds to complete value iteration in 4(b). This is significantly longer than it took us to complete 3(h) (4.27 seconds).

5 Additional scenarios

5(a) Trajectory - $p_e = 25\%$

Given $p_e = 0.25$, we ran policy and value iteration and got trajectory values of 28.15 and 22.86 respectively. Our trajectories are shown in Figures 4 and 5.

5(b) Trajectory - modified reward

We obtained the trajectories shown in Figure 6 when $p_e = 0\%$. Our value was 22.43.

We obtained the trajectories shown in Figure 7 when $p_e = 0.25\%$. Our value was 7.39.

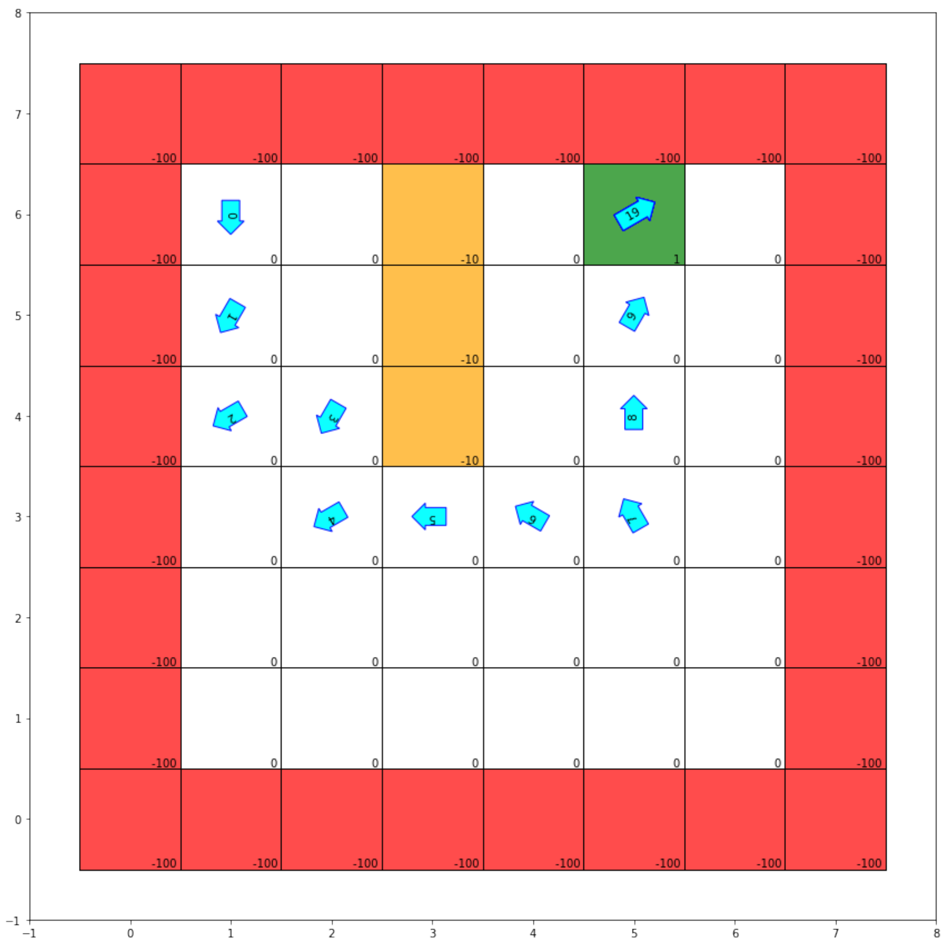

Fig. 3. Trajectory for value iteration's optimal policy.

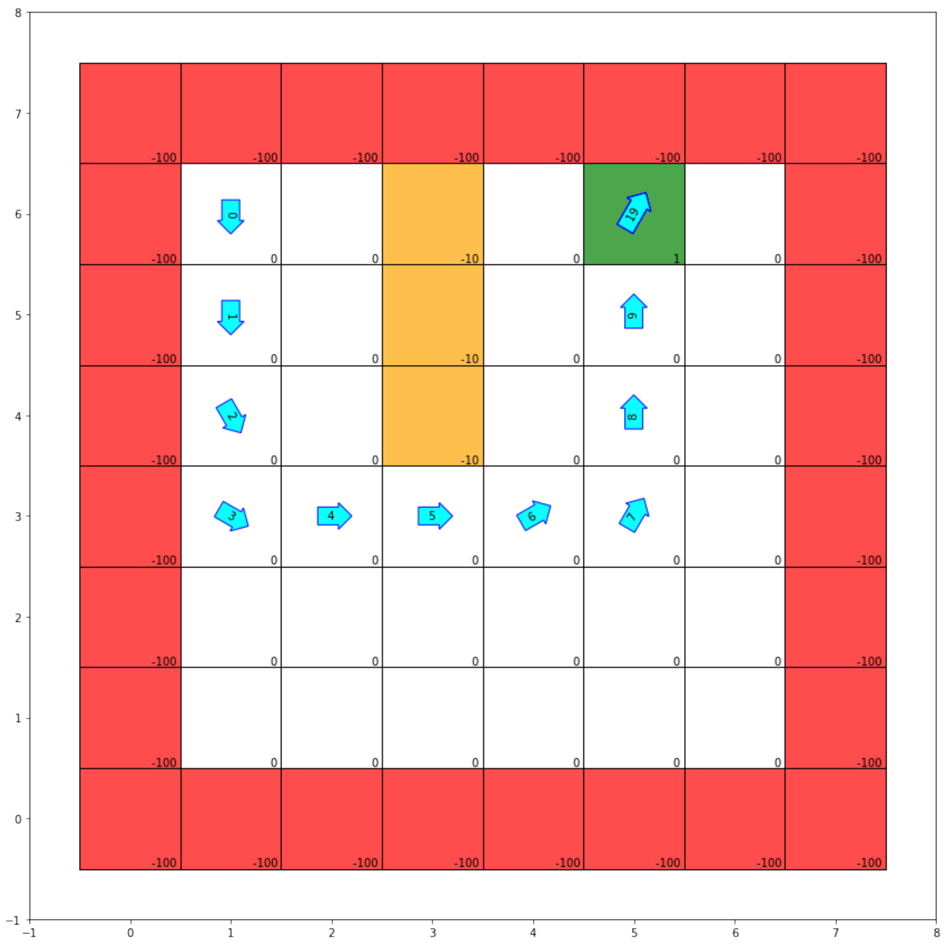

Fig. 4. Trajectory for policy iteration's optimal policy ($p_e = 0.25$).

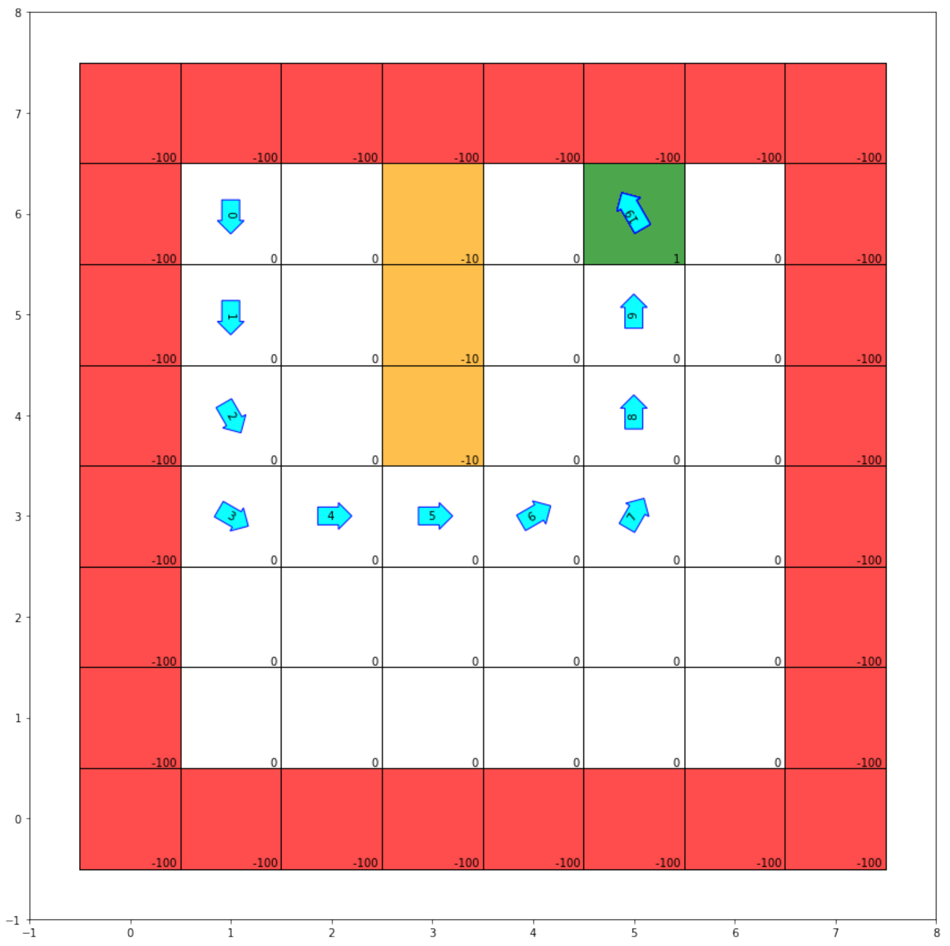

Fig. 5. Trajectory for value iteration's optimal policy ($p_e = 0.25$).

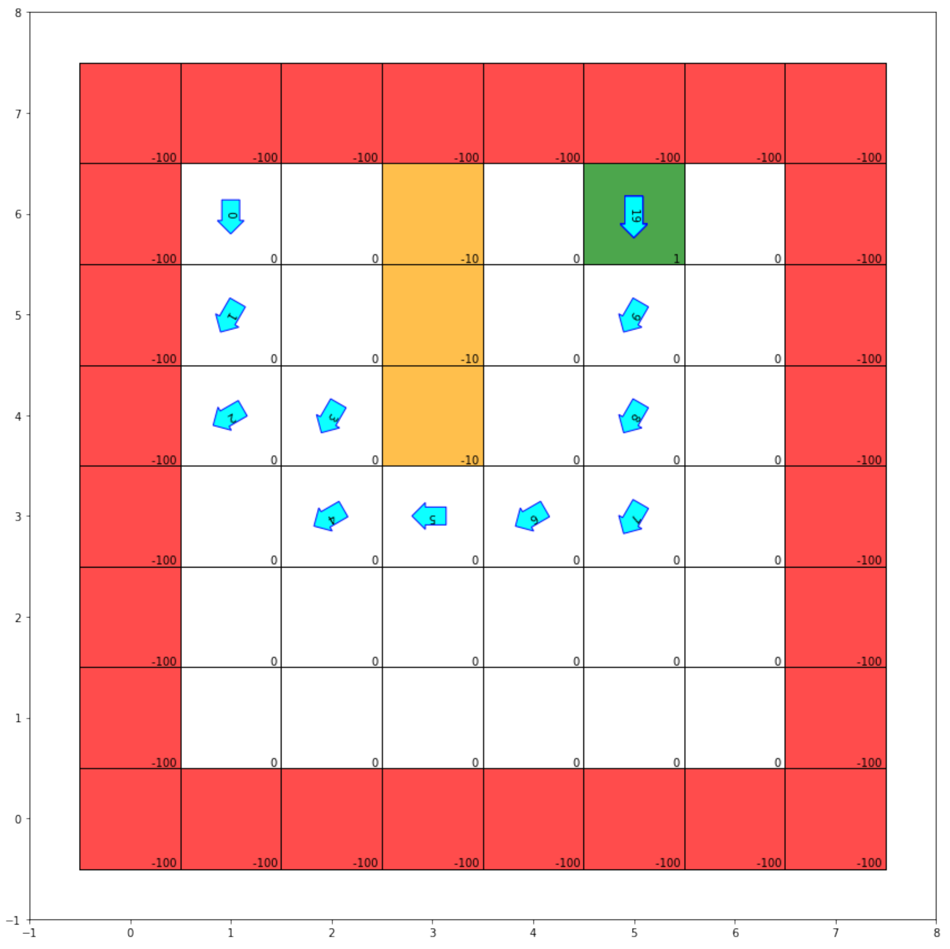

Fig. 6. Trajectory for value iteration's optimal policy (modified reward, $p_e = 0\%$).

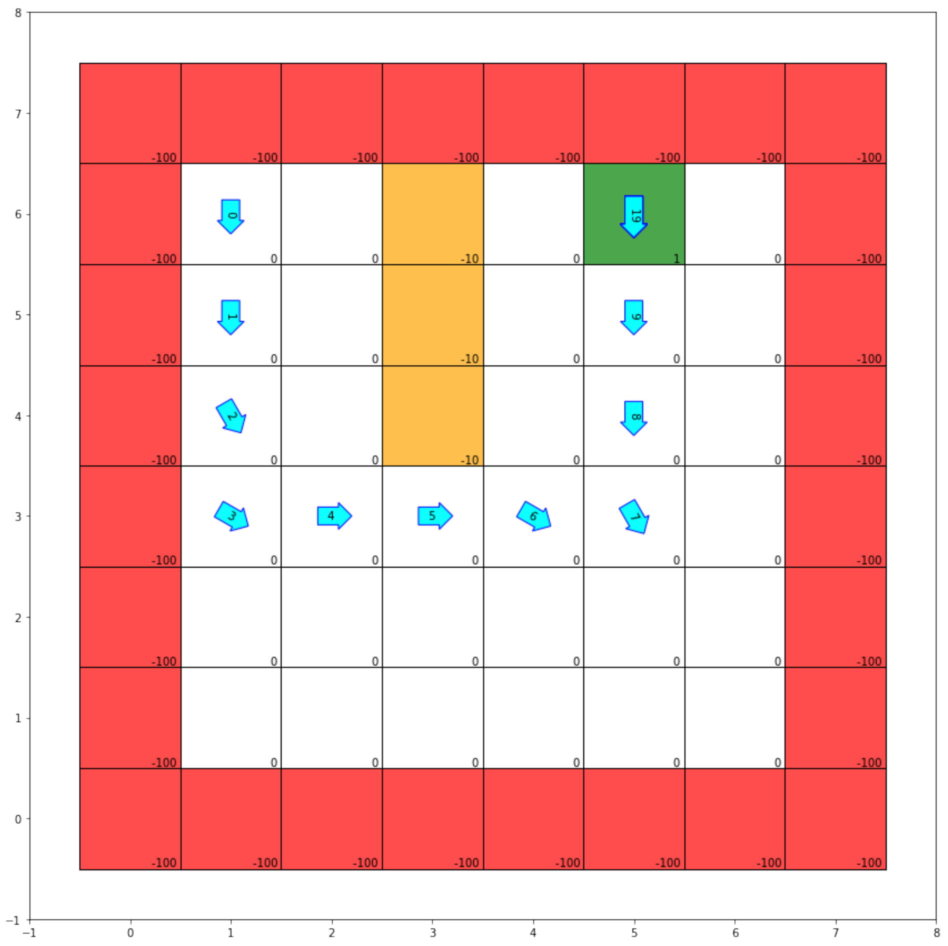

Fig. 7. Trajectory for value iteration's optimal policy (modified reward, $p_e = 0.25\%$).

5(c) Conclusions

Policy iteration converged significantly faster (4.27 seconds) than value iteration (63.87 seconds). Additionally, the value of the trajectory that policy iteration calculates for its trajectory is higher (38.35) than the value that value iteration calculates (22.43). However, despite these differences, the actual trajectories calculated by the two methods are the same.

The difference in the runtimes of the two algorithms is caused by value iteration taking more steps, each with a smaller gain–compared to policy iteration which takes fewer steps, each with a larger gain. Thus, value iteration can be stopped before convergence and have a fairly accurate trajectory, while policy iteration is likely to be very far off before convergence. Consequently, it can be useful to combine the two iteration methods.

For 5a, we saw that when we increased our error probability to 0.25 in each direction, we saw that it took a similar path compared to problem 4, except that we saw a much cleaner path that was much more focused than before. This makes sense because, due to the error probability, our robot wants to be in a state that will still get closer to the goal despite the likely errors. For example, facing directly up is the safest way to go up because even if we have an error to the right or left directions, we will still end up going up. In comparison, if we were at direction=1 and the error moved us clockwise, we would end up going right instead of going up (taking us further from the goal).

For 5b, we saw that when we gave a different reward criteria, we were able to choose a different path that still got to the optimal point in the same amount of time. This shows how by adjusting our hyperparameters and rewards, we are able to fine tune the specific path the robot takes to get to the goal (just like we were able to do by modifying the discount factor and error probability).

Bibliography

[1] A. Metha, “209 computational robotics,” 2019.