EE183DA LAB3

Authors: Li-Wei Chi, Ayush Rath,
Misheel Bayartsengel, Brendan Tanaka

February 2019

1 Introduction

This experiment was designed to demonstrate the application of a Markov Decision Process (MDP) to control a robot with a finite set of possible actions in a discretized space. An MDP describes a problem where each set of state, action and next state, (s, a, s'), is associated with a transition probability and reward. The goal is to generate a policy $\pi(s)$ that assigns an action to every state such that the expected value of the reward is maximized. This experiment was conducted entirely in simulation and the code created for this experiment can be found at: `https://github.com/CodeInSleep/Cornhuskers/tree/master/lab3`

2 Experimental Setup

The world is set up as a 6-by-6 grid, with the bottom left corner being $(0, 0)$ in the xy plane and the top right corner being $(5, 5)$. The robot can have twelve different headings, corresponding to the directions of numbers on an analog clock minus 1, with 0 indicating the $+y$ direction, 2 indicating $+x$, 5 indicating $-y$, 8 indicating $-x$ and interpolating for the other eight headings. A diagram of the world is reproduced below.

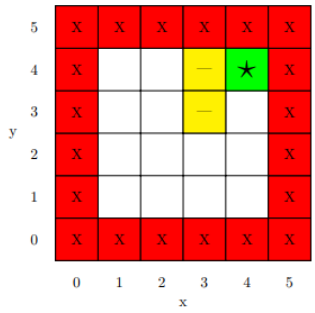


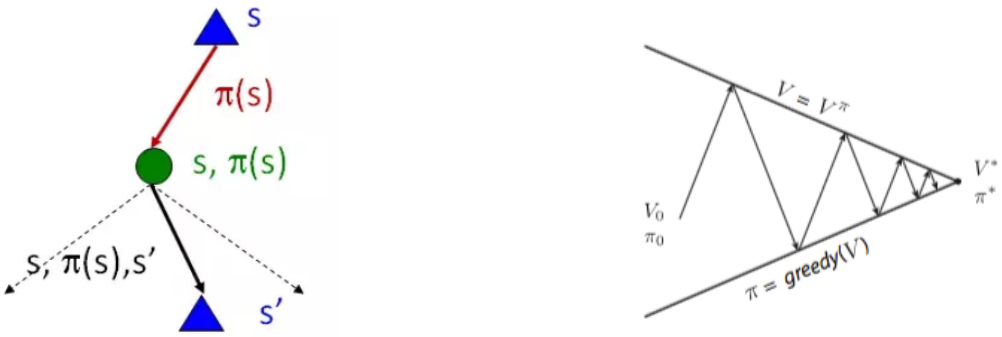
Figure 1: 6-by-6 grid world in experiment.

The locations marked in red with an X correspond to a reward of -100, the yellow '-' to a reward of -10 and the green star to a reward of +1. To solve this problem, we consider the state of the robot to be given by the x and y coordinates of the robot along with its heading. This results in a set of 36 states for the state space. The robot can take 7 possible actions, it can either stay still, move forward and rotate clockwise, move backward and rotate counterclockwise, move backward and rotate clockwise, move backward and rotate counterclockwise, move forward without rotating or move backward without rotating. Additionally, the robot has a p_e chance to 'prerotate' clockwise or counterclockwise before moving. Any rotation results in a change of the heading by 1.

3 Policy Iteration

3.1 Theory

In Policy Iteration, the policy is evaluated and improved alternately. First, using the current policy, the value function at each state is calculated. Thereafter, the policy is improved by taking the action that corresponds to the maximum reward that will be given as a result of that action. Through this iterative process, the expected reward of the policy is guaranteed to increase, thus bringing the policy closer and closer to the optimal one.



(a) Individual Step. For each state, we pick the action that maximizes the expected reward at the next state. The value function of the states are calculated in policy evaluation using the previous policy. [3].

(b) Overall Simplified Graph of Policy Iteration Algorithm. Policy Iteration greedily improves the policy, leading to smaller and smaller improvements on the policy and reaching the optimal policy eventually. [1].

Figure 2: Policy Iteration. In short, the algorithm includes: **policy evaluation + policy improvement**

In the Policy Evaluation Step, we use one step lookahead to evaluate the value function at a particular state in the following way:

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_\pi(s')$$

In the policy Improvement Step, we can improve the policy at every state by taking the action that maximizes the one step lookahead value function:

$$\pi'(s) = argmax_a(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_\pi(s'))$$

In implementation, the Policy Evaluation Step is carried out until the value function approximately converges, that is until the change in the value function of any state is less than some threshold value θ. In our case, we set theta to 0.1. In the Policy Improvement Step, we log the number of policies changed to track the progress on improving the policy. We cease the running of Policy Iteration once the policy is stable and does not change.

3.2 Experimental Results

We setup simulation cases to test our Policy Iteration algorithm. We test both cases where the p_e, the probability of prerotation, is zero and nonzero to check how having probabilistic actions affects the planned route and the actions taken. For both cases, we initialize the robot at position (1, 4) with heading 6 and the goal at position (4, 4).

For the Policy Iteration algorithm we developed, with a p_e of 0, we obtained a route that is five forward movements with counter-clockwise rotation, followed by two movements without rotation. This trajectory resulted in a total value of 10.722. A visualization of this route is shown below.

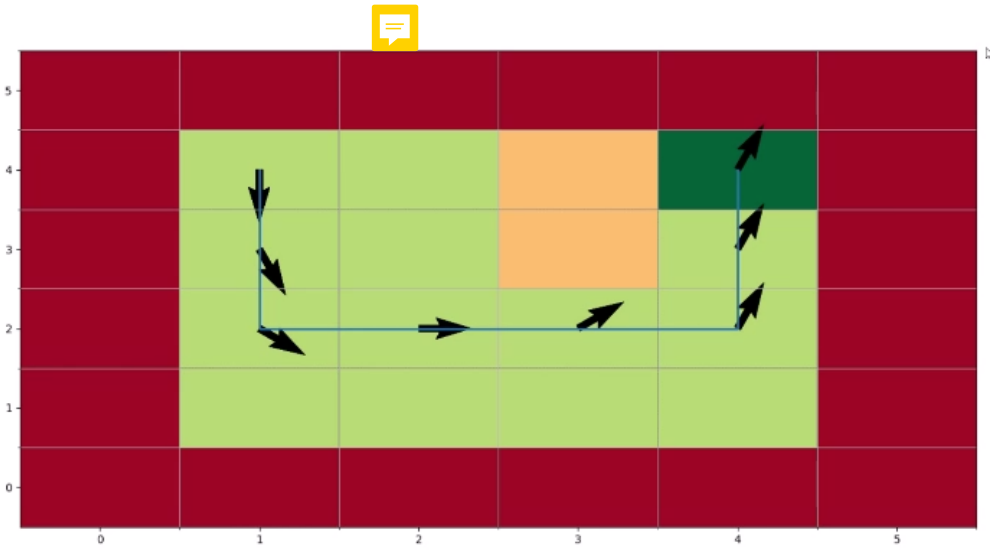


Figure 3: Optimal Trajectory found using Policy Iteration for $p_e = 0$.

Policy iteration was also performed with a prerotation chance of 10%. The result can vary between runs, but we obtained a route of forward counter clockwise rotation twice, forward without rotating twice, forward with clockwise rotation once, backwards with clockwise rotation once, and finally, backwards without rotation. This trajectory had a total value of 18.900. The resulting route is shown below.

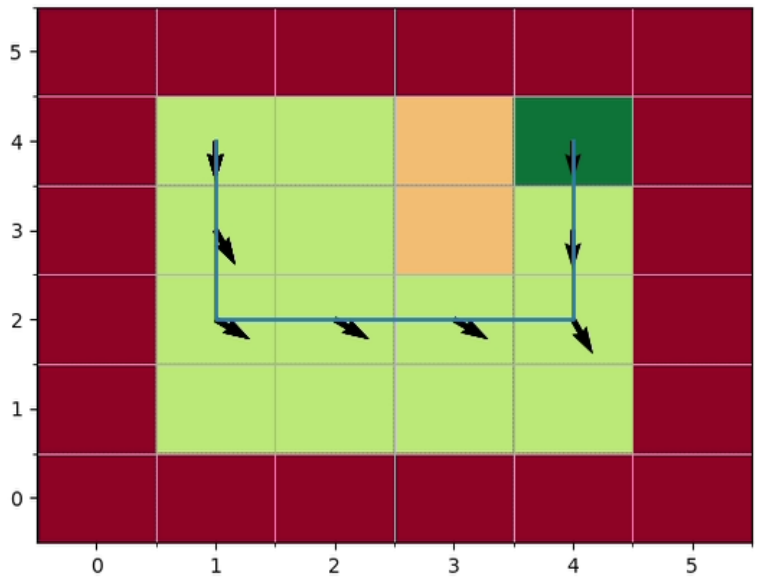


Figure 4: Optimal Trajectory found using Policy Iteration for $p_e = 0.1$.

A video of these results can be found at `https://github.com/CodeInSleep/Cornhuskers/tree/master/lab3/Demos`

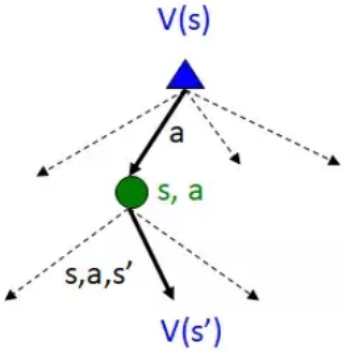
3.3 Policy Iteration Conclusion

Theoretically, Policy Iteration is guaranteed to converge due to the fact that the value function of a the new policy is guaranteed to be greater than the value function of the old policy. The run time of Policy Iteration, however, is $O(|A||S|^2 + |S|^3)$, which makes its run time longer than Value Iteration. We checked the route that we obtained from Policy Iteration with the one that we obtained from Value Iteration to verify that both algorithms yield the same route and both are optimal. In this case, the policy iteration algorithm converged in roughly 400 seconds with no prerotation error and 910 seconds with prerotation error.

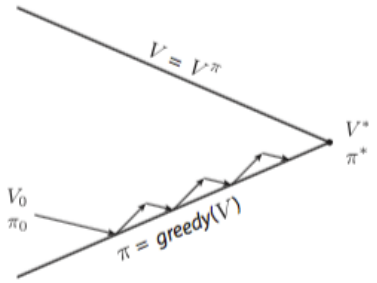
4 Value Iteration

4.1 Theory

Another variation of solving the MDP is a value iteration algorithm. Value iteration algorithm extracts the best policy only when it has found the optimal value function. It means to directly apply the *optimal* Bellman operator 1 in a recursive manner (i.e. picking the actions that maximizes the value of interested state) so that it converges to an optimal value function, and extract the policy from it at the end. The difference between the policy iteration and value iteration can be understood from a simplified graph fig:5a, 5b:



(a) Individual Step. Out of all the possible actions, we pick the action that maximizes the value of the next state using Bellman Eq:1. Difference from policy iteration, in this step, is that it is not using specific fixed policy, see fig:2a [3].



(b) Overall Simplified Graph of Value Iteration Algorithm. We only extract the policy from the optimal value function in the end using one step look-ahead Eq: 3. Compare with fig: 2b [1].

Figure 5: Value Iteration. In short, the algorithm includes: **finding optimal value function + one policy extraction**

So we start with a random value function (initialized to zero) then find a new (improved) value function in an iterative process, until the change in value is small enough, eq 1:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P(s,a,s')[\, R(s,a,s') + \gamma V_k(s')\,] \tag{1}$$

where k is the iteration number, s is current state, s' is next state (notice the algorithm is propagating backwards), $P(s,a,s')$ is probability of state transition given action a, and γ is discount factor (which is smaller than 1 and guarantees the convergence). V_0 starts as a guess (initialized to zero for all s), and we iterate for all s until the value function converges:

$$\max(V_{k+1}(s) - V_k(s)) < \theta \tag{2}$$

where θ is a threshold value we choose. From this "optimal" value function, we extract the policy, which implicitly means the "optimal" policy through one-step look ahead eq:3:

$$\pi^*(s) \leftarrow \underset{a}{\text{argmax}} \sum_{s'} P(s,a,s')[\, R(s,a,s') + \gamma V^*(s')\,] \tag{3}$$

where $\pi^*(s)$ and $V^*(s)$ are optimal policy and value functions respectively.

4.2 Experimental Results

The demo video can be found here or in the link at the bottom.

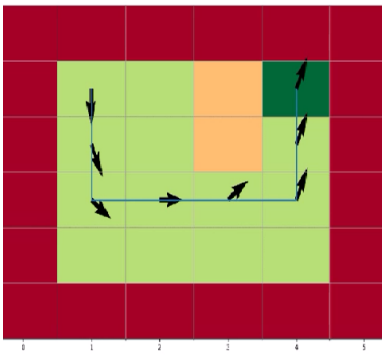


Figure 6: Optimal Trajectory for $p_e = 0$.

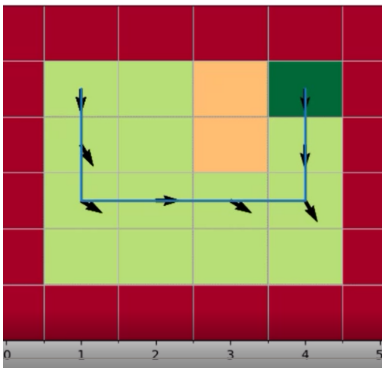


Figure 7: Optimal Trajectory for $p_e = 0.1$. Notice how the heading is changed.

After running the value iteration with varying θ, p_e, and γ, we acquired following results:

p_e	param	$\gamma = 1$	$\gamma = 0.9$	$\gamma = 0.8$	$\gamma = 0.7$
0	t, sec	139.5	226	190.6	201.7
0	θ	3	1.25	0.74	0.49
0	*Goal*?	N	Y	Y	Y
0.1	t, sec	233	242.82	253	267.7
0.1	θ	7.29	5.45	3.7	2.4
0.1	*Goal*?	N	Y	Y	Y

Table 1: Value Iteration Performance. With varying γ, p_e we simulated when does the iteration stop. We stop the iteration when $\theta < 10$, and the θ shown here is the error it actually had when it stopped. From here, we can see that with uncertainty introduced, it takes longer to evaluate, and $\gamma = 1$ does not converge in practice. And also we expect see longer time to converge, with decreasing discount factor. Since value iteration takes shorter time than policy iteration, we analyzed the effects of p_e and γ more closely.

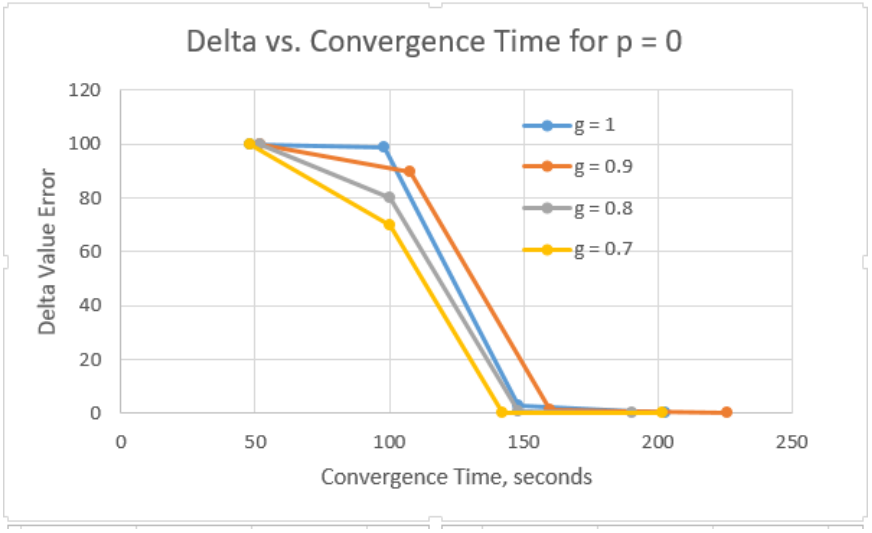


Figure 8: Varying γ Convergence Speed for $p_e = 0$. For deterministic states, we did not see much difference with γ with total convergence time, but errors decrease faster for lower γ, yet takes longer to extract the policy, ending up similar total time. Total time is shorter than $p_e = 0.1$

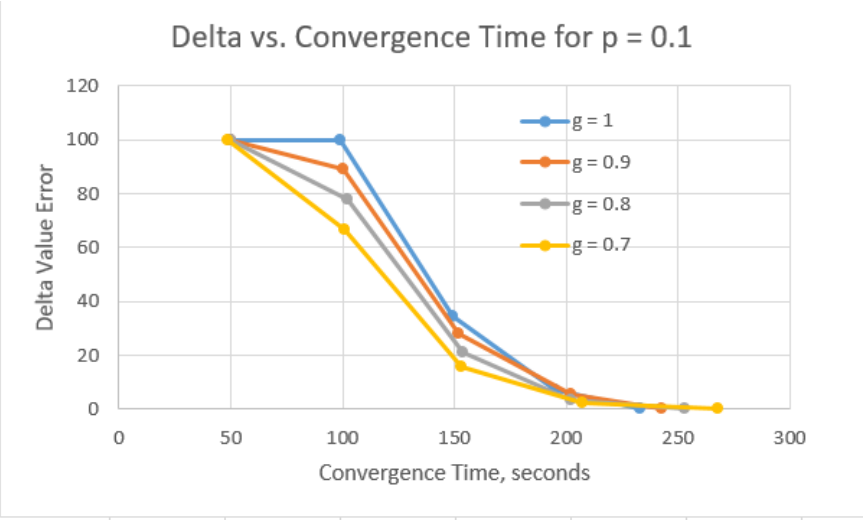


Figure 9: Varying γ Convergence Speed for $p_e = 0.1$. For uncertain states, we see longer convergence time for decreasing γ. Total time is longer than $p_e = 0.0$

4.3 Value Iteration Conclusion

Value iteration ran much faster in our testing, this is likely due to the algorithm having a runtime of $O(|A||S|)$ as opposed to $O(|S|^3)$ for policy iteration. The case with no prerotation error converged in roughly 120 seconds with a total value of 12.05, while the case with 10% error converged in roughly 150 seconds. The same route was found as with policy iteration for the case with no error chance, while a similar route was found when the prerotation chance was present. These are overall much smaller than the times taken to complete policy iteration, which makes sense since the set of states is significantly larger than the set of actions.

5 Additional Scenario

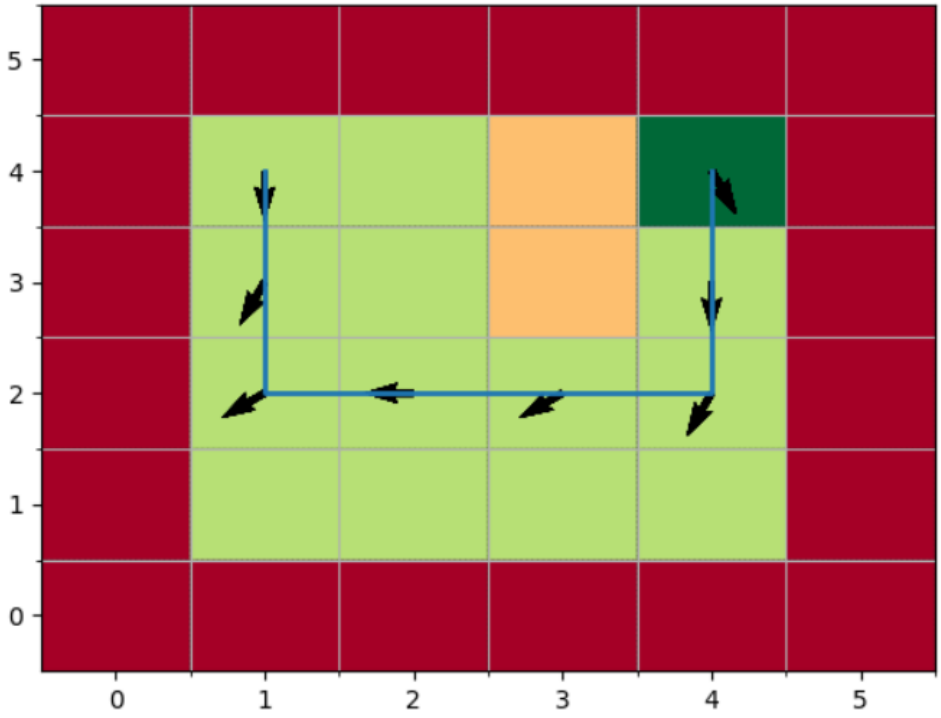


Figure 10: Reward state with Heading 5. Policy Iteration example with $p_e = 0$. See more in the demo folder.

We accidentally chose our heading as 5, not 6 as the spec said, but the the algorithm works nonetheless. The code for this is included in the 3d branch on Github. Demos can also be found in the Github/Demo folder. Results are:

- ValueIter, $p_e = 0$ $t =$ 2min, value = 12.05
- ValueITer, $p_e = 0.1$ $t =$ 3min, value = 14.08
- PolicyIter, $p_e = 0$ $t = 21$ min, value = 14.72
- PolicyITer, $p_e = 0.1$ $t =$ Over 30min, Did not Converge.

This shows that policy iteration does not scale well as was concluded earlier.

6 Future Work

We have several ideas to improve the results of this lab. To improve policy iteration, we can create a flag to determine when our algorithm is switching back and forth between two equally good policies (for example, our algorithm stayed at 10 policies with $\theta = 10$ for an extended period of time). This was was fixed by forcing it to converge more (by modifying theta to be 0.5), but the flag can also be implemented so that it uses less time.
Assuming we run the evaluation on PC not the robot's MCU, we can utilize the multi-core of the machine and parallelize. This can be done during, for example, each of the algorithm's value function calculation.
Alternatively, we can also speed up the algorithm by considering only 9x12 next possible states (9 surrounding grid, 12 headings) except evaluating the whole 6x6x12 states because we know most of them have zero probability (i.e. utilize sparsity)

7 Link to Answers

Click here for direct answers to the questions posed in the lab handout.

References

[1] Difference Between Policy Iteration and Value Iteration. StackOverflow.

[2] Denny Britz. Exercise and Solution Complement for Silver and Sutton. Github.

[3] Dan Klein and Pieter Abbeel. Markov Decision Process II. Berkeley CS188 SP14 Slides.

[2]