ECE 209 PS 3

Alexie Pogue

October 25, 2018

0 Preliminaries

0(a) Github Link

`https://github.com/APogue/209AS-Project`

0(b) Collaboration

I spoke with Colin Togashi briefly about some of the concepts.

0(c) Contributions

I wrote the code for everything but the visuals, i.e. trajectory and value plots. To have a better understanding of how to approach the algorithms, I watched Dan Klein's CS 188 Fall 2018 lectures on MDPs (Berkeley). After I wrote the code, I tried to make it more efficient* by referencing "Deep Reinforcement Learning Demystified (Episode 2)- Policy Iteration, Value Iteration, and Q-Learning". Website: `https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-\tolerance9999\emergencystretch3em\hfuzz.5\p@\vfuzz\hfuzziteration-value-iteration-and-q-978f9e89ddaa`

0(d) Aggregate Contributions

I was the only person on this team, but I used some code from Colin Togashi's repository, for that I will deduct 20% for a total aggregate contribution of 80%.

1 Setup

This problem set utilizes a Markov decision process (MDP) in order to plan a path through a known environment. Knowing the environment means the path can be computed offline. A "Markov" process means action outcomes only

*This is the first engineering assignment I've done in Python, please excuse the novice coding techniques.

depend on the current state, and decisions are made regardless of how the agent arrived at that state. The agent behaves based on reward incentives with the goal of maximizing rewards by acting optimally. An MDP is defined by:

- A set of states, $S = \{s\}$
- A set of actions $A = \{A\}$
- A transition function $P_{sa} = T(s, a, s')$
- A reward function $R(s, a, s')$
- A horizon H

1(a) State Space

Based on a 6 x 6 position grid with 12 orientation possibilities for each location, for state space $S = \{s\}$, $N_S = 432$.

1(b) Action Space

The robot can translate forwards or backwards, and may make an orientation change of 30 degrees clockwise or counterclockwise; it also has the choice to do nothing. According to these non-holonimic constraints, the robot can choose from 7 actions every time it moves. Thus for actions space, $A = \{a\}$, $N_A = 7$.

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

Algorithms involving policy and value iteration are probabilistic search problems. This means that actions don't always go as planned. Given an action at a particular state, there is the probability of an undesired outcome, and the agent will make decisions with this in mind. For example, if there is a high probability of swerving left or right when the agent intends to move forward, it may take a different route to its destination in order to avoid passing treacherous states on either side. Non-deterministic problems are beneficial when considering real-world situations where outcomes can't be guaranteed. The role of the transition function is to sample a possible state from its probability distribution given a current state and an action taken.

1(d) Next State Function

A function that provides the next state given error probability p_e, action a and initial state s was created to simulate the robot movement. The pseudocode is provided in Algorithm 1. For a given state, the x value indicates column location, the y value indicates row location, and h indicates heading.

Algorithm 1 Function $f(p_e, s, a) = s'$

if action is 'none' **then**
  return $s' = s$
else
  $s'[h] \in S'[h] = \{s'[h]_0, s'[h]_+, s'[h]_-\}$
  $s'[h] \leftarrow \text{random}(S'[h]|p_{sa}(s'))$
  if action is 'forwards' **then**
    $s'[x, y] \leftarrow \text{forwards}(s[x, y]|s'[h])$
  else
    $s'[x, y] \leftarrow \text{backwards}(s[x, y]|s'[h])$
  end if
  if $s'[x, y]$ not in $S = \{s[x, y]\}$ **then**
    $s'[x, y] \leftarrow s[x, y]$
  end if
  if length(action) $= 2$ **then**
    if action is 'left' **then**
      $s'[h] \leftarrow s'[h]_-$
    else
      $s'[h] \leftarrow s'[h]_+$
    end if
  end if
  return s'
end if

2 Problem

The following algorithms are predeterministic, they assume the agent knows something about the environment. The model layout is shown in Fig. 1. If the agent moves the robot every time step, its goal is:

$$\max \mathbf{E}[\sum_{t=0}^{H} \gamma^t R(s_t)], \tag{1}$$

or to maximize its expected sum of rewards. In this case the time horizon H is equal to infinity. This implies that even after the agent reaches the goal, it may continue to collect reward, for a maximum reward of $\frac{R}{1-\gamma}$. The parameter γ in (1) is the discount factor. Convergence is possible because the series has a limit as time goes to infinity (for $\gamma < 1$).

Smaller discount factors encourage agents to seek immediate rewards. If a reward is reduced by a factor of 2 every time step for example, an agent will prefer a smaller reward that is close by. Discount factors may impact other decisions as well, if a path is longer but less risky, it may be of more value to take that route. If the discount factor is lowered, however, it may encourage the robot to take more risk. In this case the discount factor is .9, giving the agent a fairly long horizon to make decisions.

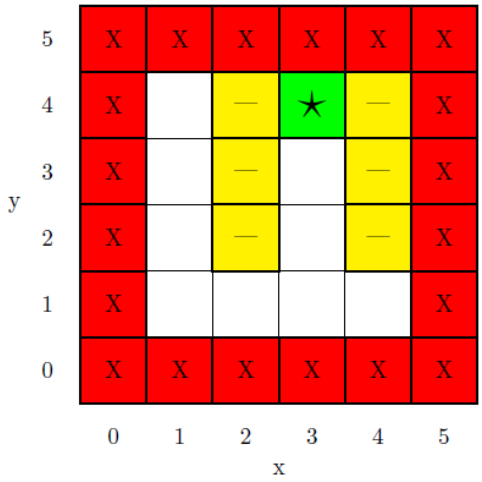

Figure 1: Grid World layout used for planning algorithms.

2(a) Reward

The reward in this case was given for being in a particular state, and did not consider transitions that arise from actions. The red squares in Fig. 1 have a reward of -100, the yellow have a reward of -10, and the green a reward of +1.

3 Policy Iteration

Policy iteration begins with an initial policy, which can be random, then recursively improves the policy until convergence to an optimal policy. The sub-algorithms used in this process are policy evaluation and policy extraction. Policy evaluation determines how good a policy is by calculating each state's expected value using a one step look ahead. One step look ahead means the agent looks at the expected reward given a single action (the agent can move a maximum of one square). Policy evaluation calculates the highest value possible given the policy.

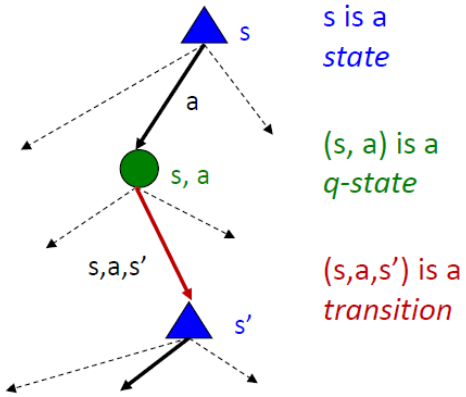

Figure 2: MDP search tree.

All algorithms are based on the search tree shown in Fig. 2*. While convergence conveys looking farther and farther ahead, the tree is built from the bottom up. Values associated with s' are first zero, the corresponding s_k determined at the top of the tree is then brought to the bottom to determine s_{k+1}.

If the "q-state" is the expected value for beginning in a state and acting optimally, then policy extraction takes the q-states generated by the original policy, and compares it to all other q-states resulting from every other possible action. If it finds a q-state higher than the one given by the old policy it replaces the action with the one deemed more optimal. The values for each state used in comparison are taken from the last round of policy evaluation, effectively creating a one step look ahead in policy. A new policy is now ready to be evaluated and so on, until the policy converges to the optimal policy $\pi^*(s)$.

3(a) Initial Policy $\pi_0(s)$

The initial policy moved the robot such that it would get to the goal square in the most efficient way possible, without regard for the lane markers or border states. The function created a vector field on the grid with the goal as the fixed point. The agent always translated towards the goal then rotated towards the vector field vector. If the robot's cardinal direction (h $\in$ group N is (11,0,1) for example) was perpendicular to the field vector for its current state it was necessary for the robot to first move away from the goal, in order to change heading direction. In these cases, given the two directions the robot could move, it would choose to move in the direction with the vector field vector

*Courtesy of Dan Klein `https://inst.eecs.berkeley.edu/~cs188/fa18/`

closest in angle to either its heading or $\pi+$ its heading. This guaranteed the angle between the heading and the vector for the next state was less than or equal to $\pi/3$, which rotates the heading towards the closest cardinal direction change. The exception is if the heading is exactly in a cardinal direction, as rotating in either direction is equally inefficient.

The resulting policy guaranteed that the robot could get to the goal square in the least amount of moves possible. There was an exception of the maximum addition of 2 moves for states whose cardinal direction was perpendicular to its own field vector (row 4 and column 3) or those whose heading direction was perpendicular to its next field vector (rows 3 and 5, columns 2 and 4). These were the least amount of moves possible given the non-holonomic constraints.

3(b) Plot Trajectory Function

This function was written by Colin Togashi, and shows the trajectory over Grid World as a dashed line, and the heading as arrows. Because the agent may traverse a path more than once, to indicate net change the arrows change color with a gradient indicating time evolution.

3(c) Robot Trajectory using $\pi_0(s)$

A trajectory was generated starting in state $x = 1, y = 4, h = 6$ using policy π_0. The trajectory was deterministic. The trajectory states are as follows:

```
[(1, 4, 6), (1, 3, 7), (1, 4, 8), (2, 4, 9), (3, 4, 9)]
```

Fig. 3 shows the trajectory generated on the grid world. As stated before, the heading of the robot is perpendicular to its desired direction. If the most efficient route is 2 moves to get to the goal, then the downward heading should add two moves. Counting the arrows, we see a total of four moves as expected.

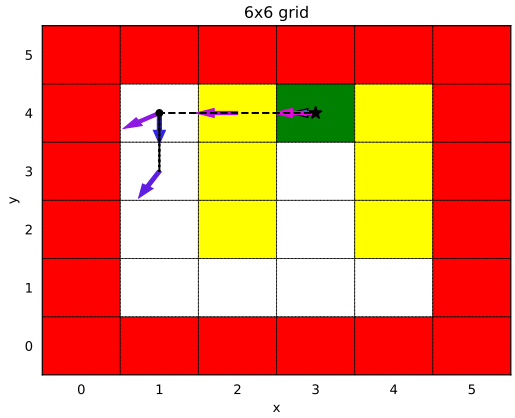

Figure 3: Trajectory using initial policy π_0.

3(d) Policy Evaluation

Policy evaluation computes the value for a single action or policy. The Bellman update for policy evaluation takes the form,

$$V_0^\pi(s) = 0,$$
$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} p_{sa}(s'|s,a)\big[R(s) + \gamma V_k^\pi(s')\big]. \tag{2}$$

Without a "max" in (2) this makes the efficiency $O(S^2)$ (or $O(N_s^2)$), a factor of A faster than value iteration per iteration. The efficiency is for the worst case, where in this case the number of probable outcomes s' does not equal s. The actual complexity per iteration is 1296 computations.

Another benefit of no "max" is for tractable cases it can be solved like a linear equation. In this case dynamic programming was used to find the solution.

3(e) Value of Trajectory generated by $\pi_0(s)$

The value for the initial polciy π_0 is:

```
[array([-0.72900739]), array([-0.81000821]),
array([-0.90000821]), array([-1.00000821]), array([ 9.99999179])]
```

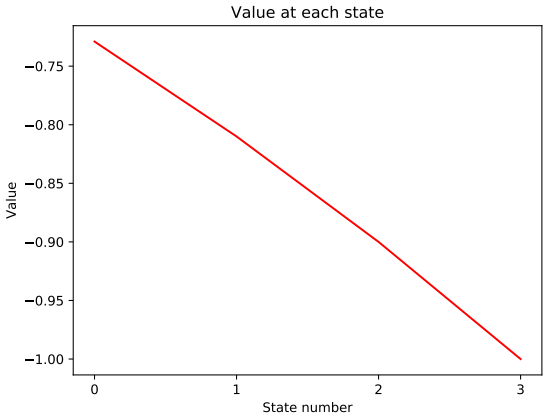

Figure 4: Values using π_0 given $p_e = 0$, $R_{lane} = -10$.

If the reward for the lanes is increased to -1, the opposite behavior is observed:

```
[array([ 5.83199261]), array([ 6.47999179]), array([
7.19999179]), array([ 7.99999179]), array([ 9.99999179])]
```

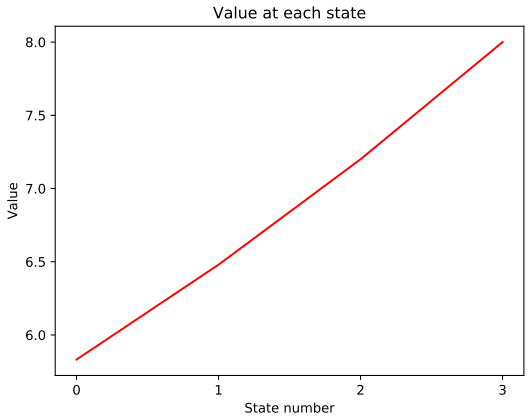

Figure 5: Values using π_0 given $p_e = 0$, $R_{lane} = -1$.

This is due to the continuation of collection of rewards that increases the goal state's value. Under these conditions, the initial policy is the optimal policy. This especially clear in Figs. 4 and 5, with the value of the last state omitted.

3(f) Policy Extraction

Policy extraction generates a policy by choosing the action corresponding to the highest q-state. It uses the Bellman equation in the form,

$$\pi_{k+1}(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} p_{sa}(s'|s,a)\big[R(s) + \gamma V_k^{\pi}(s')\big]. \tag{3}$$

Policy evaluation has an efficiency of $O(S^2A)$, where for each state all actions are taken, and for each action this leads you to more possible states s'. Again for this case the probable states arising from an action are far fewer than N_s. This efficiency is equal to that of value iteration per iteration. This equation, however, only needs a single evaluation to generate a policy. The actual complexity per iteration is 9072 computations.

3(g) Policy Iteration

Policy iteration is the combination of policy evaluation and extraction.

3(h) Robot Trajectory using $\pi^*(s)$

A trajectory was generated starting in state $x = 1, y = 4, h = 6$ and $R_{lane} = -10$ was generated using policy π^*. The trajectory states are:

```
[(1, 4, 6), (1, 3, 6), (1, 2, 7), (1, 1, 8), (2, 1, 8), (3, 1,
7), (3, 2, 7), (3, 3, 7), (3, 4, 7)]
```

Their respective values are:

```
[array([ 4.30466612]), array([ 4.78296235]), array([
5.31440261]), array([ 5.90489179]), array([ 6.56099179]), array([
7.28999179]), array([ 8.09999179]), array([ 8.99999179]), array([
9.99999179])]
```

Optimal behavior is shown in Figs. 12 and 13. It is clear when comparing the plots to those generated by the initial policy (Figs. 3 and 4) that this policy is better. It gets to the goal in as efficient manner as possible, the trajectory qualitatively looks optimal.

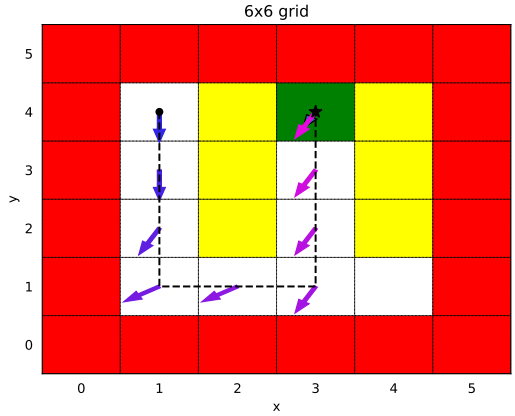

Figure 6: Trajectory using π^* given $p_e = 0$.

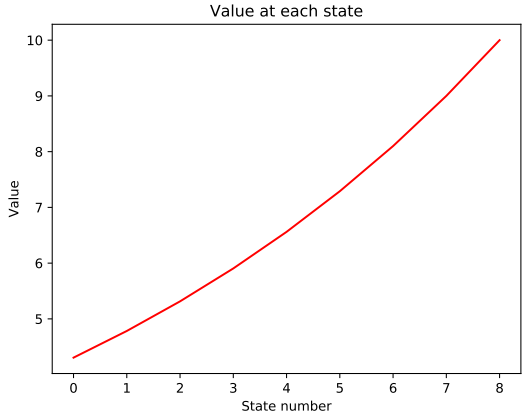

Figure 7: Values using π^* given $p_e = 0$.

3(i) Compute Time

Policy iteration should be faster in most cases. The slower policy extraction computation is not iterative. In addition, if values change but the max q-state corresponding to the optimal action does not, then policy iteration will converge faster than value iteration. The timed output read was:

```
policy iteration timer 119.2192362
```

The algorithm runs much faster if you change the reward for the goal state to 10. This is the value that policy iteration converges to anyways and the results are very similar to those in the preceeding section. The timed output read was:

```
policy iteration timer 12.4518273
```

The trajectory states are:

```
[(1, 4, 6), (1, 3, 6), (1, 2, 7), (1, 1, 8), (2, 1, 8), (3, 1,
7), (3, 2, 7), (3, 3, 7), (3, 4, 7)]
```

Their respective values are:

```
[array([ 4.3046721]), array([ 4.782969]), array([ 5.31441]),
array([ 5.9049]), array([ 6.561]), array([ 7.29]), array([ 8.1]),
array([ 9.]), array([ 10.])]
```

Value Iteration

Value iteration is a mixture of policy evaluation and policy extraction in a single recursive form. Like policy evaluation, the algorithm iterates until one step look ahead values converge. Rather than calculating the q-state for a single action, however, all actions are considered, and the highest q-state is chosen. In contrast to policy extraction the algorithm does not bother to find the policy at each step. Instead it directly returns values for the next one step look ahead calculation on the states.

4(a) Value Iteration

Value iteration uses the Bellman update in the form (4). The "max" in (4) indicates choosing the maximum expected value over actions available to the agent:

$$\begin{aligned} &V_0(s) = 0, \\ &V_{k+1}(s) \leftarrow \max_a \sum_{s'} p_{sa}(s'|s,a)\big[R(s) + \gamma V_k(s')\big], \\ &\pi^*(s) = \operatorname*{argmax}_a \sum_{s'} p_{sa}(s'|s,a)\big[R(s) + \gamma V^*(s')\big]. \end{aligned} \tag{4}$$

This approach is analogous to a fixed point solution method. The algorithm should have values increase until convergence. Once values have converged to $V^*(s)$, policy extraction can be used to generate optimal policy, $\pi^*(s)$.

The computational complexity of each iteration is $O(S^2A)$, equal to that of policy extraction.

4(b) Robot Trajectory generated under $\pi^*(s)$

A trajectory was generated starting in state $x = 1, y = 4, h = 6$ and $R_{lane} = -10$ was generated using policy π^*. The trajectory states are:

```
[(1, 4, 6), (1, 3, 6), (1, 2, 7), (1, 1, 8), (2, 1, 8), (3, 1,
7), (3, 2, 7), (3, 3, 7), (3, 4, 7)]
```

Their respective values are:

```
[array([ 4.30466612]), array([ 4.78296235]), array([
5.31440261]), array([ 5.90489179]), array([ 6.56099179]), array([
7.28999179]), array([ 8.09999179]), array([ 8.99999179]), array([
9.99999179])]
```

Optimal behavior is shown in Figs. 14 and 15. It is clear when comparing the plots to those generated by policy iteration (Figs. 12 and 13) that the resulting policies are the same. Both the trajectory and the corresponding values are the same.

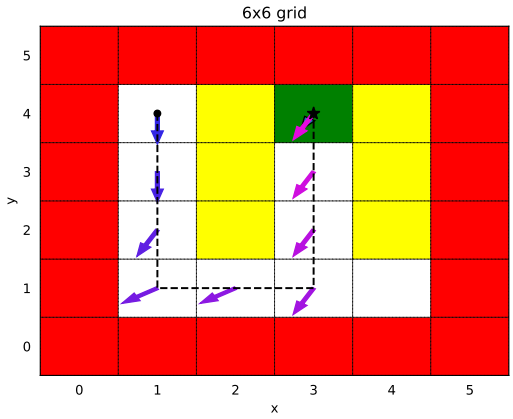

Figure 8: Trajectory using π^* given $p_e = 0$.

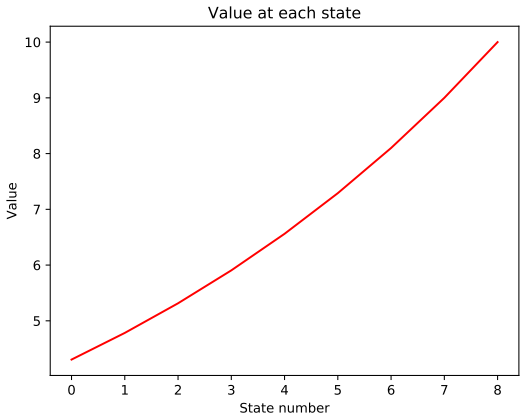

Figure 9: Values using π^* given $p_e = 0$.

4(c) Compute Time

Value iteration is slightly slower than policy iteration:

```
value iteration timer 126.1830177
```

If the reward is changed to 10, the states are:

```
[(1, 4, 6), (1, 3, 6), (1, 2, 7), (1, 1, 8), (2, 1, 8), (3, 1,
7), (3, 2, 7), (3, 3, 7), (3, 4, 7)]
```

The values are:

```
[array([4.3046721]), array([4.782969]), array([5.31441]),
array([5.9049]), array([6.561]), array([7.29]), array([8.1]),
array([9.]), array([10.])]
```

The timed output is:

```
value iteration timer 9.3611245
```

The time is faster than that of policy iteration. This may not be the case for the values on average, and more computations would be necessary to draw a conclusion.

5 Additional Scenarios

5(a) Trajectory Generated in Non-Deterministic Conditions

The initial policy was tested under non-deterministic conditions. A trajectory was generated again starting in state $x = 1, y = 4, h = 6$, with $p_e = .25$ and $R_{lane} = -10$. The trajectory states are as follows:

```
[(1, 4, 6), (1, 3, 8), (2, 3, 8), (2, 4, 6), (2, 3, 7), (3, 3,
9), (4, 3, 11), (4, 4, 11), (4, 3, 10), (4, 4, 0), (4, 5, 1), (4,
4, 1), (4, 5, 1), (3, 5, 3), (2, 5, 3), (3, 5, 4), (2, 5, 4), (3,
5, 5), (3, 4, 6)]
```

For the sake of brevity the values are shown in Fig 10 only. It is clear that while the controller is making a good effort given the numerous prerotations, it is not factoring in the reward scheme at all as it chooses to move forward into the boarder states.

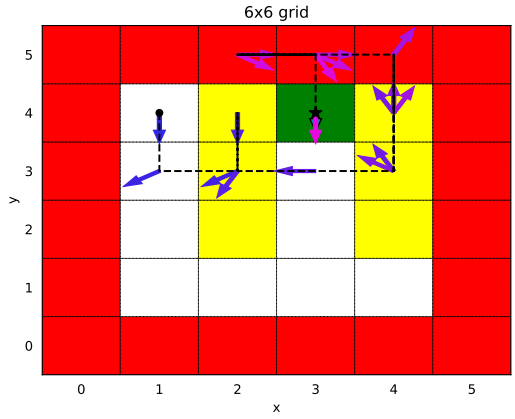

Figure 10: Trajectory using π_0 given $p_e = .25$.

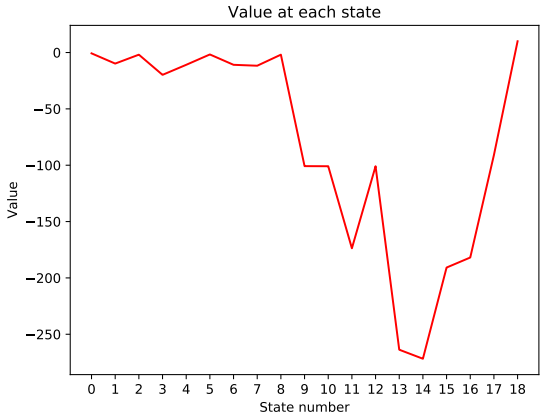

Figure 11: Values using π_0 given $p_e = .25$.

The optimal policy was tested under non-deterministic conditions. A trajectory was generated again starting in state $x = 1, y = 4, h = 6$, with $p_e = .25$

and $R_{lane} = -10$. The trajectory states are as follows:

```
[(1, 4, 6), (1, 3, 6), (1, 2, 7), (1, 1, 7), (1, 2, 6), (1, 3,
5), (1, 2, 6), (1, 3, 5), (1, 2, 5), (1, 1, 4), (2, 1, 4), (3, 1,
5), (3, 2, 6), (3, 3, 7), (3, 4, 7)]
```

Their respective values are:

```
[array([ 4.30466612]), array([ 4.78296235]), array([
5.31440261]), array([ 4.78296161]), array([ 4.30466545]), array([
4.78296235]), array([ 4.30466545]), array([ 4.78296235]), array([
5.31440261]), array([ 5.90489179]), array([ 6.56099179]), array([
7.28999179]), array([ 8.09999179]), array([ 8.99999179]), array([
9.99999179])]
```

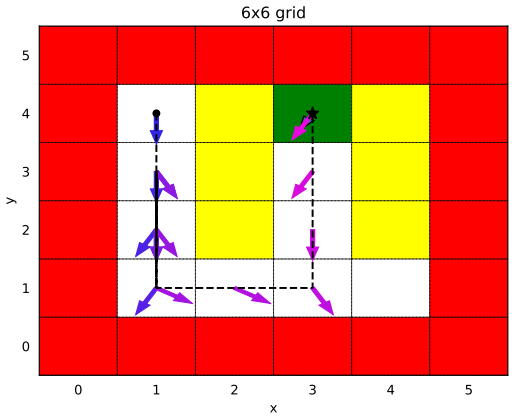

Figure 12: Trajectory using π^* given $p_e = .25$.

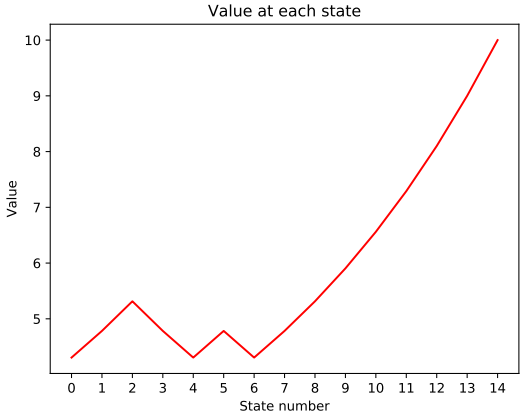

Figure 13: Values using π^* given $p_e = .25$.

In Fig. 12 the policy traverses the same path many times over to avoid the lane blocks. Using value iteration in the same scenario, the policy should be the same. In this case (Figs. 14 and 15), the agent stopped moving before going into a lane block.

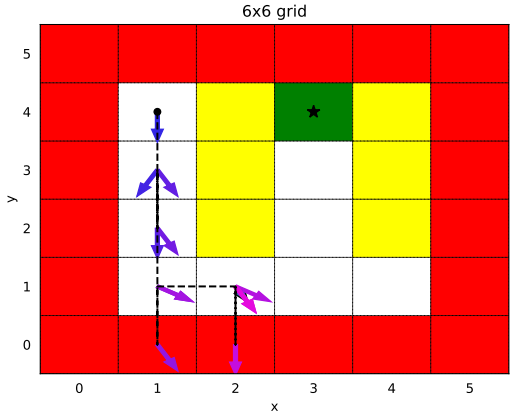

Figure 14: Trajectory using π^* given $p_e = .25$.

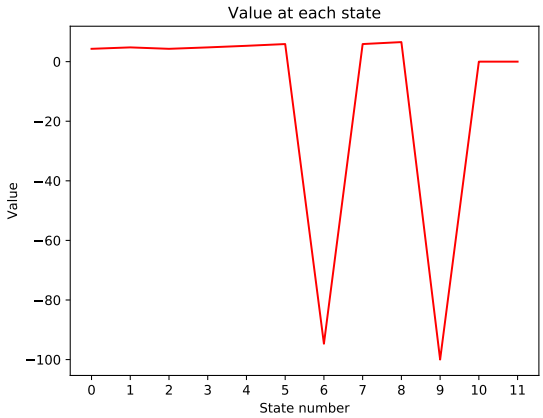

Figure 15: Values using π^* given $p_e = .25$.

5(b) Trajectory Generated for Heading-Dependent Reward Problem

The trajectory generated for the situation when the goal states lie only in the headings = 5, 6 or 7 are shown below. This process was generally slower and more accident prone as the agent tried to rotate its heading.

```
[(1, 4, 6), (1, 3, 6), (1, 2, 7), (1, 1, 7), (2, 1, 8), (3, 1,
7), (3, 2, 7), (4, 2, 8), (3, 2, 7), (3, 3, 6), (3, 4, 7)]
```

The corresponding values are:

```
[array([4.3046721]), array([4.782969]), array([5.31441]),
array([4.782969]), array([6.561]), array([7.29]), array([8.1]),
array([-2.71]), array([8.1]), array([9.]), array([10.])]
```

5(c) Conclusions

There may have been some bugs with the code, which were uncovered when I attempted to plot the heading-dependent trajectory. In some cases if I set the reward to 10, the agent would go out of its way to rotate the heading north, and the corresponding reward would be close to 100. I'm not sure if this means changing the reward to its convergent value is not fool-proof, or if there is a glitch in the code. It may be the case, that if I only cap the max reward for the goal state, it is more advantageous to then go directly next to it as my code would allow the continuous collection of goals for those states.