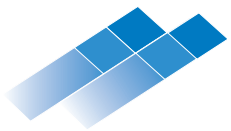
Technische Universität München
Lehrstuhl für Datenverarbeitung
Prof. Dr.-Ing. Klaus Diepold

Approximate Dynamic Programming and Reinforcement Learning

Programming Assignment

Alperen Gündogan, 03694565

January 30, 2019



1 Probabilistic Formulation of the Problem

- I implemented the transition probability matrix in the following form; $P[s][a] = (prob, next_state, reward, is\_done)$. The matrix consists of states s which are the cells of the maze and available actions a for the corresponding states. First of all, states of the 2-dimensional maze should turn into the 1D vector with the size of $nS = nrows * ncols$. Then, available actions i.e. check the walls for the action, is determined for each state. Then, probabilities which are defined in the assignment are implemented for all states with the corresponding next state i.e. successor state s' and reward value for the available actions.
- If the number of states is high and we have many available actions for each state, then the algorithm would require so much time until it converges. For example, if we have a 3-dimensional environment, we would need to convert the environment into states and specify all possible actions for each state. This operation requires so much memory to store all the state-action pairs.

2 Policy

- Policy is the two dimensional array with the size nS x nA where nA is the number of actions from the action space $C = \{up, right, down, left, idle\}$. Each row indicates the probabilities of actions on that state. For example, $\mu(s) = [0.2, 0.2, 0.2, 0.2, 0.2]$ where $\forall s \in S$ shows that the agent can take all the actions with equal probability and the sum is equal to 1. After the convergence of the algorithm, we would only have e.g. $\mu(s) = [0, 1, 0, 0, 0]$ (right) where the agent can have only one action.
- At the beginning of both algorithms, a uniform random policy is initialized.

3 Solution with Dynamic Programming

- For the maze problem, the state space is limited with the number of cells in the maze. I simply use $numpy.ndarray.flatten()$function which takes the maze array and return a copy of the array collapsed into one dimension.
- Q: How can you determine if the DP algorithms have converged? Think of a sensible numerically stable criteria and describe it.
  - Policy evaluation and VI formally requires infinite number of iteration to converge exactly to V^*. Therefore, I added a stopping condition for practical implementation. If the value function changes by only a small amount in a sweep, then the DP algorithms should be converged i.e. $\max_{s \in S} \mid V_{k+1}(s) - V_k(s) \mid < \theta$ with $\theta = 10^{-25}$.
- Do the two methods generate the same policy?

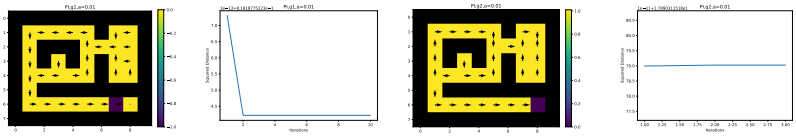
  - Yes, I got the same policy for $g_1, VI, \alpha = 0.9$ and $g_2, PI, \alpha = 0.9$.

- Has the changed α an influence on the final policy?
  - Yes. I would say that the effect of α on the final policy is also related with the cost function that we use. For example; for $g_1, VI, \alpha = 0.01$ and $g_1, PI, \alpha = 0.01$ I can have the same optimal policy as $\alpha = 0.5, 0.9$ however, for $g_2, VI, \alpha = 0.01$ and $g_2, PI, \alpha = 0.01$, I got different policies than other α values.

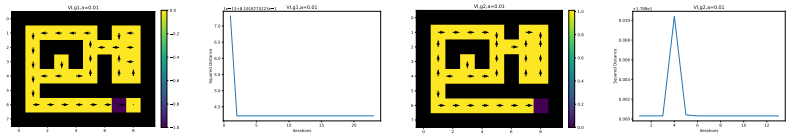
4 A study of Algorithm Properties

- The ground truth that I have obtained for cost functions is the same for both PI and VI at start state $[g_1, g_2] = [-0.81918, 18.90041]$.
- Which values of α did you use?
  - I tested the algorithms for $\alpha = [0.3, 0.7, 0.99]$ and both cost functions which you can find the error plots and heat map in the figures. As I understand from my results, suitable α value depends on how we evaluate the actions i.e. implementation of the cost function. For g_1 I get the optimal policy for different values of α. However, I only have the optimal policy for g_2 at $\alpha = 0.99$. The action at state $(x, y) = (4, 4)$ in the Figure 6 for both VI and PI, only converges to the optimal action for that state at $\alpha = 0.99$ for g_2. Therefore, one should test the algorithms for different α values to find the optimal policy and suitable discount factor for given cost functions.
- Is VI or PI better / faster? Why?
  - The convergence of policy iteration is faster than value iteration. For Policy iteration, we first apply the Bellman operator for the given policy until convergence to the value function of the given policy, this is called policy evaluation. Then, we apply policy improvement to improve the given policy by taking the action that maximizes the value function for every state. However, we can terminate the policy evaluation step with some tolerable error instead of waiting until convergence and perform policy improvement to have better policy. For value iteration, we again apply the optimal Bellman operator to the value function until convergence to optimal value and get the greedy policy for every state. VI can be interpreted as applying one iteration of the policy evaluation loop and then improve the policy i.e. the policy evaluation step is approximated with just one single iteration. Therefore, VI is slower and computationally heavier than PI.
  - I would say that PI is better than VI since it is computationally efficient as it often takes a fewer number of iterations to converge although each iteration is more computationally expensive.

5 Plots

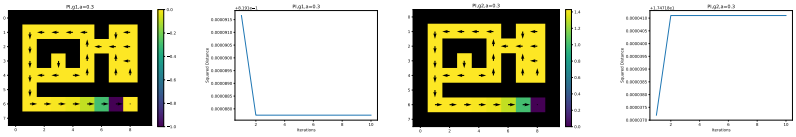


(a) PI, g1 (b) PI, g1 (c) PI, g2 (d) PI, g2

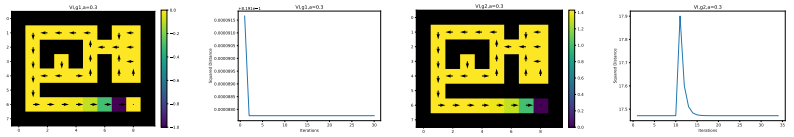


(e) VI, g1 (f) VI, g1 (g) VI, g2 (h) VI, g2

Figure 1: Discount Factor = 0.01

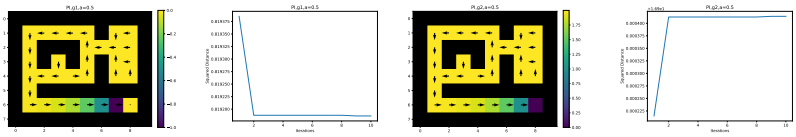


(a) PI, g1 (b) PI, g1 (c) PI, g2 (d) PI, g2

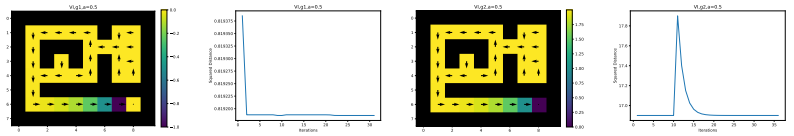


(e) VI, g1 (f) VI, g1 (g) VI, g2 (h) VI, g2

Figure 2: Discount Factor = 0.3

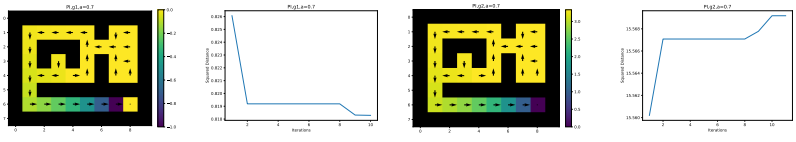


(a) PI, g1 (b) PI, g1 (c) PI, g2 (d) PI, g2

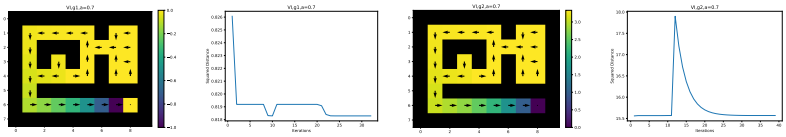


(e) VI, g1 (f) VI, g1 (g) VI, g2 (h) VI, g2

Figure 3: Discount Factor = 0.5

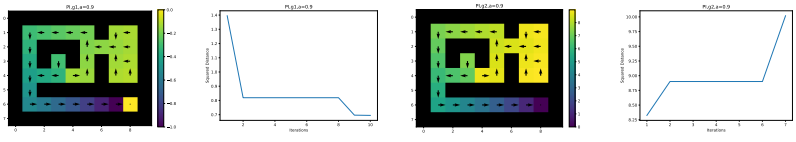


(a) PI, g1 (b) PI, g1 (c) PI, g2 (d) PI, g2

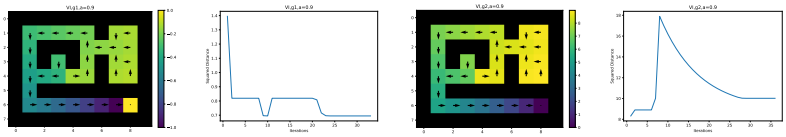


(e) VI, g1 (f) VI, g1 (g) VI, g2 (h) VI, g2

Figure 4: Discount Factor = 0.7

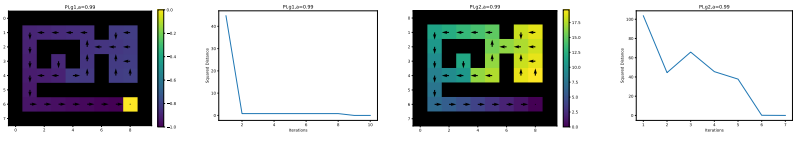


(a) PI, g1 (b) PI, g1 (c) PI, g2 (d) PI, g2

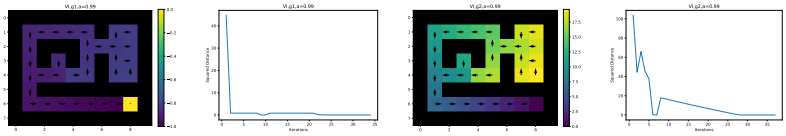


(e) VI, g1 (f) VI, g1 (g) VI, g2 (h) VI, g2

Figure 5: Discount Factor = 0.9



(a) PI, g1 (b) PI, g1 (c) PI, g2 (d) PI, g2



(e) VI, g1 (f) VI, g1 (g) VI, g2 (h) VI, g2

Figure 6: Discount Factor = 0.99