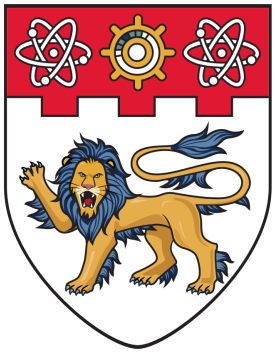
Assignment 1

Intelligent Agents



NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

Teng Yao Long

U2121909D

Contents

1 Introduction

In this assignment, we explore classic Reinforcement Learning (RL) algorithms Policy Iteration and Value Iteration [1], and the impact of the number of iterations on their utility estimates. We first explore the implementation for the environment in Section 2 since it is universal for both Value Iteration and Policy Iteration, followed by Value Iteration in Section 3 and Policy Iteration in Section 4. The bonus question is then answered in the last section. My implementation consists of 2 classes: the GridWorld environment class and the Agent class. We build environments following the de facto standard of OpenAI gym [2], where the most important method in the environment class is the *step*() method, which that takes an action a_t of the agent and updates state from s_t to s_{t+1} and returns reward r_t. The Agent class implements the Value Iteration and Policy Iteration algorithms respectively, which will be further discussed in the following sections. This assignment was done in Python and we use the terms utility and value interchangeably in this document. Additionally we calculate the convergence threshold $\epsilon\left(\frac{1-\gamma}{\gamma}\right) = 0.00101$ where $\epsilon = c * R_{max}$ by setting $c = 0.1$.

2 GridWorld Environment

The GridWorld class creates an instance of the GridWorld environment. The GridWorld class accepts parameters as shown in Figure 1:

- **tile_reward**: a dictionary specifying the reward of each tile colour.
- **map**: a 2-D array specifying the colour of each tile.
- **values_map**: a 2-D array of the initial values of each tile, with dimensions equal to that of **map**.

```
class GridWorld:

    def __init__(self, tile_reward= TILE_REWARD, map = None, size = None):
        self.tile_reward = tile_reward
        if map == None:
            self.map = [["Green", "Wall", "Green", "White", "White", "Green"],
                    ["White", "Brown", "White", "Green","Wall","Brown"],
                    ["White","White", "Brown", "White", "Green","White"],
                    ["White", "White", "White", "Brown", "White", "Green"],
                    ["White", "Wall","Wall","Wall","Brown","White"],
                    ["White","White","White","White","White","White"]]
        else:
            self.map = map
        if size == None:
            self.values_map = np.zeros((6,6))
        else:
            self.values_map = np.zeros((size, size))

        self.states = [(c, r) for r, row in enumerate(self.map) for c, tile in enumerate(row) if tile != "Wall"]
```

Figure 1: Code snippet of GridWorld constructor

Note that the user does not need to manually create a 2-D array if a custom maze is to be created, as it can easily be done with a nested for-loop. I also provide this function in my project.

The most important method of the GridWorld class is the *step* method, which takes an action a_t of the agent and updates state from s_t to s_{t+1} and returns reward r_t. It provides a one-step lookahead for the utility values required in the Bellman update.

```
41      def step(self, pos, action):
42          x, y = copy.deepcopy(pos)
43          x_, y_ = x,y
44
45          if self.is_valid_action(x, y, action) is False:
46              return pos, self.get_reward(pos)
47
48          if action == ACTIONS["UP"]:
49              y_ = y-1
50          elif action == ACTIONS["DOWN"]:
51              y_ = y+1
52          elif action == ACTIONS["LEFT"]:
53              x_ = x-1
54          elif action == ACTIONS["RIGHT"]:
55              x_ = x+1
56
57          reward = self.get_reward((x, y))
58          next_pos = (x_, y_)
59
60          return next_pos, reward
```

Figure 2: Code snippet of *step* method

Since we use Value Iteration to find the optimal policy, we implement the transition model in the Agent class for convenience while achieving the same desired outcome. However, it is understood that for algorithms where stochastic policies are used, it might be better to separate the transition model from the Agent class. Additional methods and their explanations are included in the Appendix.

3 Value Iteration

Value Iteration is a dynamic programming algorithm used in reinforcement learning and Markov Decision Processes (MDP) to find the optimal policy by iteratively improving the estimation of the value function, which estimates how good it is to be in a given state. The main idea is to systematically update the values of each state with the Bellman update rule given in (1) until they converge to approximately the optimal value.

$$V(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)\left[R(s,a,s') + \gamma V(s')\right] \tag{1}$$

Once the optimal value function is known, the optimal policy can be easily derived. The pseudocode of Value Iteration is given below.

Algorithm 1 Value Iteration

function VALUE-ITERATION(mdp, ϵ)
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s'|s,a)$,
    rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
      δ, the maximum change in the utility of any state in an iteration
  repeat
    $U \leftarrow U'$
    $\delta \leftarrow 0$
    for each state s in S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U[s']$
      if $|U'[s] - U[s]| > \delta$ **then**
        $\delta \leftarrow |U'[s] - U[s]|$
      end if
    end for
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U
end function

3.1 Description of Implemented solution

In this section, we focus the discussion on the implementation of Value Iteration. Additional methods of the Agent class are discussed in the Appendix. The implementation of the Python code follows closely to that of the pseudocode introduced in Algorithm 1. We observe that since the assignment does not specify a terminal state and that the agent's state sequence is infinite, we can estimate the upper bound of a tile's utility with the sum to infinity geometric series:

$$\text{Max utility} \approx S_\infty = \frac{1}{1 - 0.99} = 100 \tag{2}$$

Through the experiments conducted in the later sections, we will notice that the results are consistent with this estimate. Additionally, since we have no fixed length trajectory and we use a discount factor γ, the optimal policy is independent of the starting state of the agent.

The Value Iteration implemented in Figure 3 is synchronous. As opposed to asynchronous methods, the synchronous implementation stores 2 copies of the value map as seen in line 124 of Figure 3. Each state is then updated with the Bellman Equation given in Equation (1) as seen in lines 129 to 135. *actionl* is a list of actions and *probs* is a list of the actions' respective probabilities. For example, if the action being evaluated currently is "Up", $actionl = [Up, Left, Right]$ and $probs = [0.8, 0.1, 0.1]$ according to the transition model. We then further use the actions to obtain the next state s_{t+1} and reward r via the *step* method, and update the values of each state with the Bellman Equation. Subsequently in lines 137

to 142, the maximum value over all actions is saved as the value of the current state, while the action that produced the maximum value is saved in the policy. In line 145, we keep track the historical state values for graph plotting later. The iteration continues until the maximum difference across all states between the old values and the new best values $\Delta \leq \theta$ where θ is a small user-specified positive value, since this means the algorithm is close to convergence. In experiments, the value of θ is set to 0.001, and 688 iterations are required to reach convergence.

```
112     def value_iteration(self):
113         theta = 0.00101
114         count = 0
115         all_states = self.policy.keys()
116         flag = True
117 
118         while flag:
119             count += 1
120             #if count == 36:  #specifies when to break iteration if we do not use theta as nreak condition
121                 #break
122             print(count)
123             delta = 0
124             v_tmp = copy.deepcopy(self.v)
125             for s in all_states:
126                 x, y = s
127                 max_a_value = -1
128 
129                 for action in self.actions.values():
130                     a_value = 0
131                     actionl, probs = self.get_action_probs(action)
132                     for a, p in zip(actionl, probs):
133                         s_, r = self.env.step(s, a)
134                         x_, y_ = s_                                    #we put r inside as p sums to 1 anyway
135                         a_value += p * (r + self.gamma * v_tmp[y_][x_]) #after iteration ends
136 
137                     if a_value > max_a_value:
138 
139                         max_a_value = a_value
140                         argmax_action = action
141                         self.policy[s] = argmax_action
142                         self.v[y][x] = max_a_value
143                         delta1= abs(v_tmp[y][x] - self.v[y][x])
144 
145                 self.value_history[(x, y)].append(self.v[y][x])
146 
147                 delta = max(delta, delta1)
148                 if delta < theta:
149                     flag = False
```

Figure 3: Value Iteration implementation

3.2 Plot of optimal policy

By simply observing the given environment in the assignment manual, we can notice that an agent can achieve maximum expected reward at the top left corner (0,0) by simply choosing the "Up" action, since if the move would make the agent walk into a wall, the agent stays in the same place as before. Therefore it is beneficial for the agent to reach this state in the long run so maximum reward can be attained. As expected from the experiment results in

Figure 4, we observe that the general direction the agent is moving is towards the top left corner. In the code, we encode "Up" as 0, "Down" as 1, "Left" as 2 and "Right" as 3.

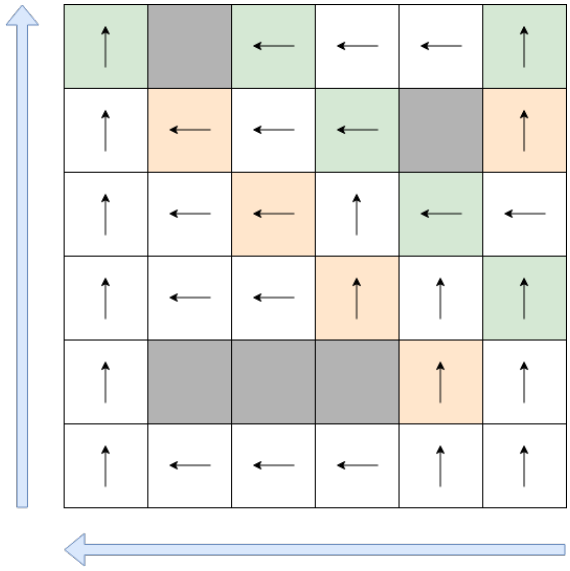


Figure 4: Plot of optimal policy

Console outputs are given in the Appendix, while the plot is given above in Figure 4

3.3 Utility of all states

As we observe in Figure 5, the estimate provided for the upper bound in Equation (2) is consistent with our obtained utilities.

99.90068522	0	94.89609657	94.76432259	93.53160891	93.1617994
99.32832311	97.76231894	95.44108761	94.24181132	0	92.74707229
97.86886722	96.49303648	95.1475818	94.05853676	92.93338125	92.66318125
96.46011866	95.34764866	94.11537757	92.94639889	92.68291459	91.70683124
95.20626038	0	0	0	91.36372876	91.42267075
93.81732594	92.5964202	91.39073772	90.20008869	89.4048257	90.14077672

Figure 5: Plot of utility of all states

3.4 Plot of utility estimates as a function of number of iterations

As mentioned before, we set θ to be 0.001. As a result, a total of 688 iterations are required to reach convergence. The utility estimates as a function of the number of iterations is plotted in Figure 6 below.

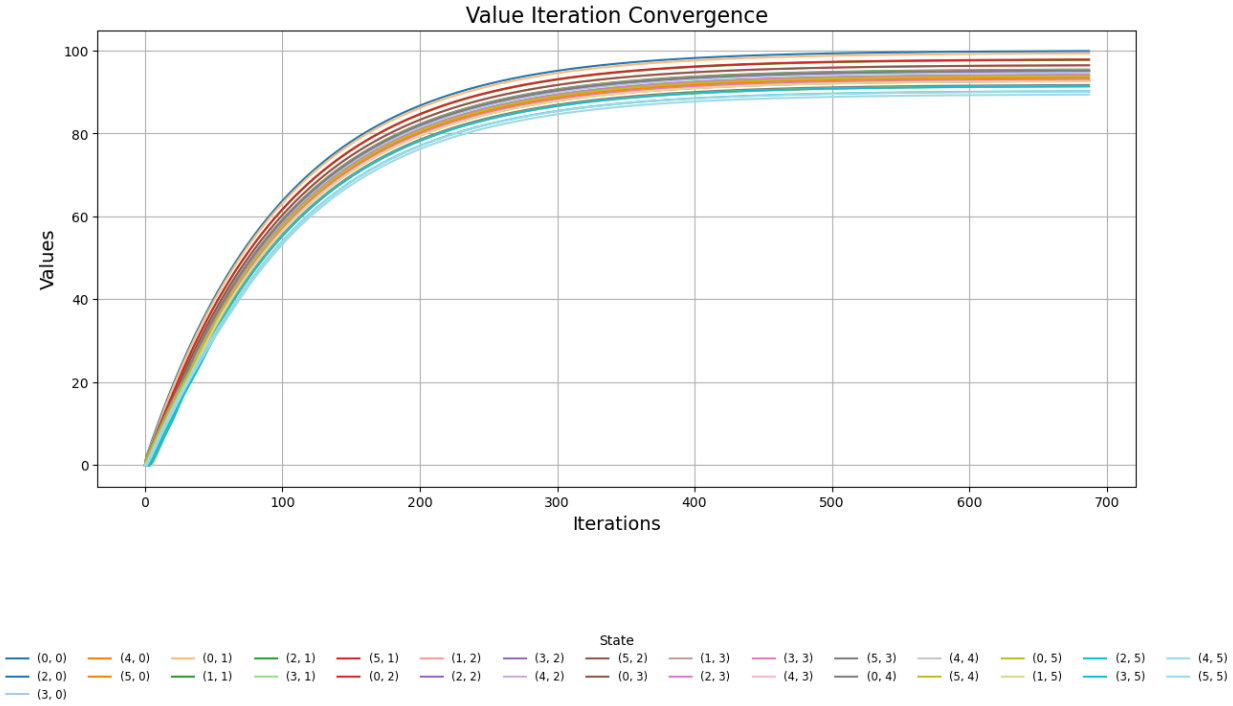


Figure 6: Plot of utility estimates as a function of the number of iterations

From Figure 6, we can see that the convergence process is relatively smooth, and increases at a decreasing rate as the number of iterations increase, and eventually plateaus to the optimal utility value.

4 Policy Iteration

Policy Iteration is a method used in reinforcement learning to find the optimal policy for a given Markov Decision Process (MDP). It involves two main steps: policy evaluation and policy improvement, which are iteratively repeated until the policy converges to the optimal policy. Policy iteration and Value iteration are actually very similar and in fact, policy iteration with one-step policy evaluation can be thought of to be the same as Value Iteration [3]. In general, Policy Iteration converges faster than Value Iteration. The pseudocode followed in the implementation of Policy Iteration is given below.

In my implementation for Policy Evaluation, the commonly used iterative method is used instead of using linear algebra to directly find the solution since the state space is already

Algorithm 2 Policy Iteration

function POLICY-ITERATION(mdp)
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s'|s,a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random
  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s in S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U[s'] > \sum_{s'} P(s'|s,\pi[s])U[s']$ **then**
        $\pi[s] \leftarrow \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a)U[s']$
        $unchanged? \leftarrow$ false
      end if
    end for
  until $unchanged?$
  return π
end function

sizeable for a 6x6 grid. This approach is mentioned in the Artificial Intelligence textbook Page 657 [4] referenced in the assignment.

4.1 Description of Implemented solution

In this section, we focus the discussion on the implementation of Policy Iteration. The implementation of the Python code follows closely to that of the pseudocode introduced in Algorithm 2.

```
116     def evaluate_policy(self):
117         theta = .001
118         count = 0
119         all_states = self.policy.keys()
120         delta = np.inf
121         while delta >= theta:
122             count+=1
123             #print(count)
124             delta = 0
125             v_tmp = copy.deepcopy(self.v)
126             for s in all_states:
127                 x, y = s
128 
129                 actions, probs = self.get_action_probs(self.policy[s])
130                 value = 0
131                 for a, p in zip(actions,probs):
132                     s_, r = self.env.step(s, a)
133                     x_, y_ = s_
134                     value += p*(r + self.gamma * v_tmp[y_][x_]) #we can put r inside as p sums to 1 anyway
135 
136                 self.v[y][x] = value
137                 self.value_history[(x,y)].append(value) #uncomment this if we want to record value history at every
138                                                         #policy evaluation step
139                 delta = max(delta, abs(v_tmp[y][x] - self.v[y][x]))
140 
141             if delta <= theta:
142                 for s in all_states:
143                     x, y = s
144                     #self.value_history[(x, y)].append(self.v[y][x])
```

Figure 7: Code snippet of Policy Evaluation

The above Figure 7 shows the implementation of the policy evaluation step. Policy evaluation as it name suggests, evaluates the utilities of each state under a given policy. Similar to Value Iteration, we keep 2 copies of the value map to determine the occurrence of convergence when the maximum difference across all states between the old values and the new best values $\Delta \leq \theta$. The crux of the policy evaluation algorithm is from line 126 to 136, where we calculate the value of each state under the current policy. The explanations for each step is similar to that of Value Iteration given in Section 3.1. The only difference is that instead of getting the maximum utility over all actions, we simply update the current state value with the utility obtained under our current policy. In my code implementation, we can choose to record the state values history at each step of policy evaluation or at the end of policy evaluation (convergence occurs).

```
148     def improve_policy(self):
149         is_stable = True
150         all_states = self.policy.keys()
151         for s in all_states:
152             old_pi = copy.deepcopy(self.policy[s])
153             argmax_action = None
154             max_a_value = -1
155 
156             for action in self.actions.values():
157                 a_value = 0
158                 actionl, probs = self.get_action_probs(action)
159                 for a, p in zip(actionl,probs):
160                     s_, r = self.env.step(s, a)
161                     x_, y_ = s_
162                     a_value += p*(self.gamma * self.v[y_][x_])
163 
164                 if a_value > max_a_value:
165 
166                     max_a_value = a_value
167                     argmax_action = action
168                     self.policy[s] = argmax_action
169 
170             if old_pi != argmax_action:
171                 is_stable = False
172 
173         return is_stable
```

Figure 8: Code snippet of Policy Improvement

In the Policy Improvement step, we update the policy by making it greedy with respect to the current value function. This means for each state, choose the action that maximises the expected return using the current value function. This is done in line 156 to line 168 in Figure 8, where we get the action that maximises the expected return using the current value function, across all states (line 151). Subsequently, if the policy does not change during the improvement step, then it is considered optimal and the algorithm stops. If it does change, the new policy is evaluated and improved upon iteratively until convergence. This is done in line 170 to 171, where if the policy remains the same over all states in the Policy Improvement step, policy is said to be stable and convergence occurs.

```
176 if __name__ == '__main__':
177     env = GridWorld()
178     gamma = 0.99
179     is_stable = False
180     count = 0
181
182     agent = Agent(env, gamma)
183     print("Iterations:")
184     while is_stable == False:
185         agent.evaluate_policy()
186         is_stable = agent.improve_policy()
187         count += 1
188         print(count)
```

Figure 9: Code snippet of Policy Iteration

Finally, we put the Policy Evaluation step and Policy Improvement step together to form the Policy Iteration algorithm in Figure 9 lines 185 to 186, and finish the iteration when convergence occurs.

4.2 Plot of optimal policy

The final optimal policy obtained from Policy Iteration is identical to that of the one we obtained from Value Iteration, which is good news. The console output is given in the Appendix, while the plot is given below in Figure 10. Similarly, we see that the agent tries to move to the top left corner, which is the best position due to reasons explained in Section 3.2.

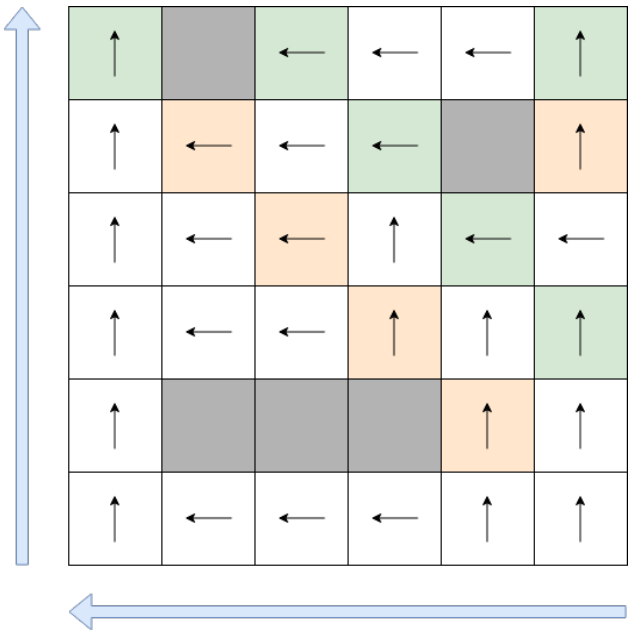


Figure 10: Plot of optimal policy

4.3 Utility of all states

Similarly, we observe that the upper limit of our utilities are bounded by our estimation in Equation (2).

99.99985837	0	95.0453156	93.87485904	92.65447219	93.32783829
98.39321988	95.88287575	94.54485674	94.39757192	0	90.9171839
96.94835855	95.58628612	93.29428598	93.17612334	93.10219816	91.79458046
95.55369747	94.45235217	93.23240379	91.11509146	91.81417792	91.88773216
94.31237778	0	0	0	89.54805947	90.56623357
92.93733269	91.728636	90.53501034	89.3562678	88.56848786	89.29683174

Figure 11: Plot of utilities of all states

4.4 Plot of utility estimates as a function of number of iterations

Similar to Value Iteration, we set θ to be 0.001. We require a total of 5 policy iterations to reach convergence, which is much less than that of Value Iteration. However, the number of Policy Evaluation steps required is 1341. The plots where we show the utility estimates at each step of Policy Evaluation and at the end of Policy Evaluation are shown for completeness.

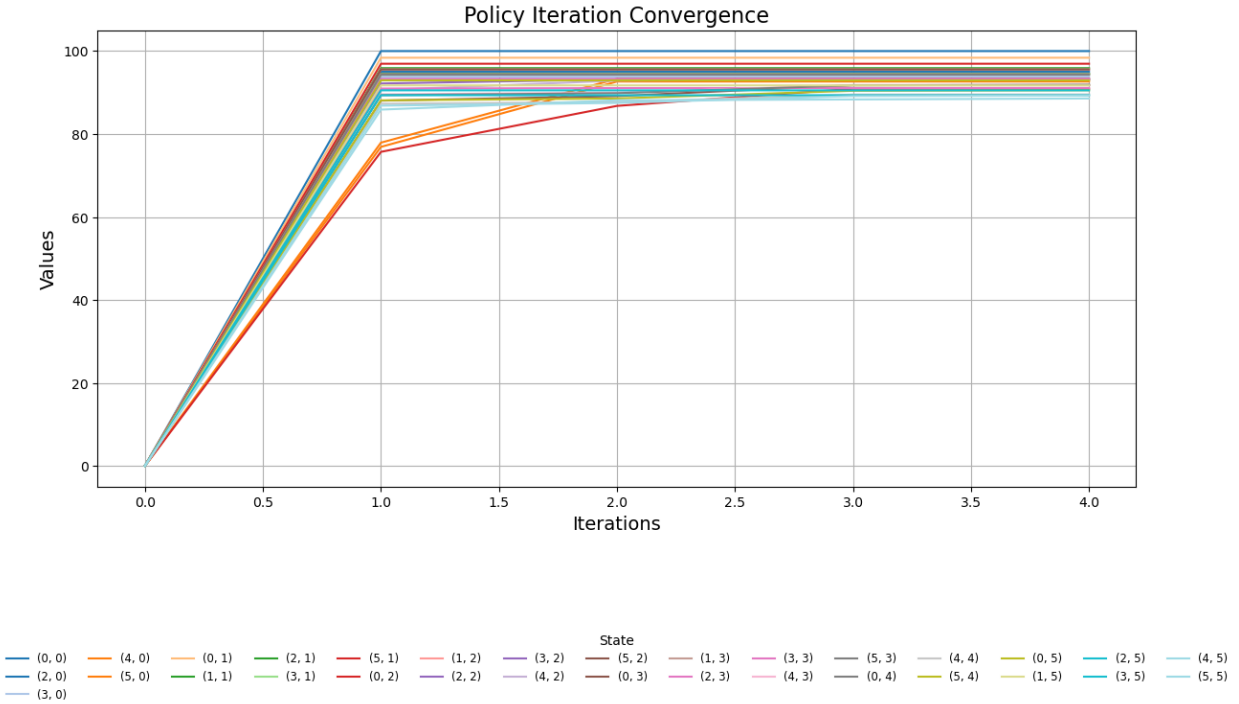


Figure 12: Plot of utility estimates **after** Policy Evaluation convergence as a function of the number of iterations

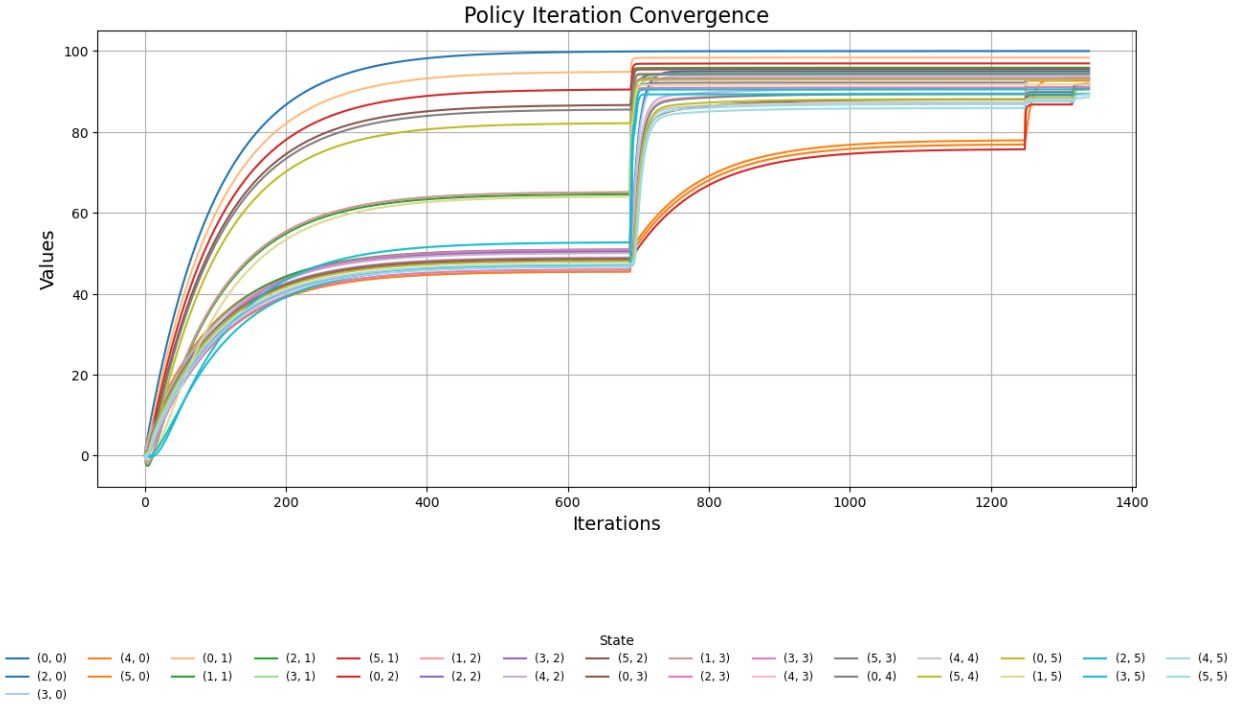


Figure 13: Plot of utility estimates **at each step** of Policy Evaluation as a function of the number of iterations

In Figure 12, we can see that the utility estimates slowly converges as the number of itera-

tions increases, eventually plateauing when enough iterations has occurred. Additionally, we can intuitively understand that the Policy Improvement step occurs at each iteration when the gradient between 2 lines change sharply. This can be more obviously seen in Figure 13, where there is a sharp increase in utilities at each policy improvement step. Also, we can see that within each Policy Evaluation step, the utility values under the current policy increases and eventually converges. These observations are nicely consistent with our understanding of the Policy Iteration algorithm.

5 Bonus Question

For the bonus question, we are tasked to design a more complicated maze environment of your own. However, it must be noted that for the same dimensions, there is no heuristic to determine how "complicated" a GridWorld maze is as the word "complicated" is subjective. Additionally, since we are tasked to determine how "complicated" a maze is affects the convergence of our algorithms, we cannot say that a longer time taken means a maze is more complicated, since that is the effect we are trying to determine in the first place. As such in my opinion, the complexity of the maze can only be increased objectively by increasing the dimensions of the maze first. That is, we guarantee that the maze is more complex in bigger dimensions since we need to sweep through more states. Additionally, in the original assignment manual, although we do not have a terminal state we have a best state (top-left corner). This greatly helps in propagating the utility estimates back since the value of that best state will stabilise quickly (like an "anchor" point). This means that other than increasing the dimensions of the maze, to make a maze more "complicated" than the maze given in the assignment, we can simply augment the maze to not have a fast converging best state.

We can create a new maze with a function that accepts the dimensions given in Figure 14. An example map is shown in Figure 15.

```python
def generate_map(n):
    # Define the colors and wall as strings
    elements = ["Brown", "White", "Wall", "Green"]

    # Generate a NxN 2D array with random selection of the elements
    map_nxn = np.random.choice(elements, size=(n, n), p=[0.25, 0.5, 0.15, 0.1])

    map_list = map_nxn.tolist()
    return map_list
```

Figure 14: Function to generate maps

```
['White', 'White', 'White', 'Green', 'White', 'White', 'White', 'Wall', 'Green', 'Wall']
['White', 'White', 'White', 'Brown', 'Brown', 'White', 'Brown', 'Brown', 'White', 'White']
['White', 'Brown', 'Wall', 'Green', 'Brown', 'White', 'White', 'Wall', 'Wall', 'White']
['White', 'White', 'White', 'Brown', 'Wall', 'White', 'Green', 'Brown', 'Wall', 'Green']
['White', 'White', 'White', 'Brown', 'White', 'White', 'White', 'Brown', 'Wall', 'Brown']
['White', 'Wall', 'White', 'Wall', 'Brown', 'Wall', 'Brown', 'Green', 'White', 'White']
['Green', 'White', 'Brown', 'Brown', 'White', 'Brown', 'White', 'White', 'White', 'White']
['Brown', 'White', 'Brown', 'Wall', 'Brown', 'White', 'White', 'Green', 'White', 'White']
['White', 'Wall', 'Brown', 'White', 'White', 'White', 'White', 'White', 'Brown', 'White']
['Wall', 'Brown', 'Wall', 'Green', 'White', 'Wall', 'White', 'White', 'Wall', 'White']
```

Figure 15: Example 10x10 map generated for bonus question

We verify our hypothesis with experiments. Experiment results for both algorithms show that even with an increased θ at 0.1 from 0.001 at the same dimensions, without a fast-stabilising best state, the number of policy iterations has exceeded 100,000. This is exacerbated when the dimensions are increased, where a mere increase to 10x10 grid has made the number of iterations required to exceed 1,000,000 (and still not converged). This is due a lack of an "anchor" point for Bellman updates to stably propagate across all states.

As expected, by increasing the complexity of the maze (via increasing the dimensions of the maze and then randomly populating) increasing the number of iterations needed for convergence since we need to iterate across more states. Given enough computing power and time, it is always possible to solve an environment's utilities. However from my experiment results, a 100x100 grid already excruciatingly increases the time taken. As the state space further increases, it will be wise to adopt other algorithms. If the user wants to use a similar algorithm, we can first try asynchronous updates for our algorithms instead of the current synchronous updates. If we want to further trade-off time taken with accurate estimations on the utility values, we can use model free algorithms such as Q-learning to get optimal policies. Deep Q-Networks[5] leverages the power of deep neural networks for approximation to allow us to get solutions faster. Additionally, we can use state-of-the-art algorithms like Proximal Policy Optimisation [1] if we prefer an on-policy algorithm. To further improve sample efficiency, deterministic policy gradient methods [6] with experience replay can also be used. Finally, if we want a more robust exploration strategy using entropy, Soft-Actor Critic can be adopted [7].

6 Appendix

6.1 Environment

Additional methods of the environment class are presented in this section.

```
60      def get_reward(self, pos: tuple):
61          x, y = pos
62          # print(pos)
63          return self.tile_reward[self.map[y][x]]
64
            1 usage
65      def is_valid_action(self, x, y, action):
66
67          x_, y_ = x, y
68
69          if action == ACTIONS["UP"]:
70              y_ += -1
71          elif action == ACTIONS["DOWN"]:
72              y_ += 1
73          elif action == ACTIONS["LEFT"]:
74              x_ += -1
75          elif action == ACTIONS["RIGHT"]:
76              x_ += 1
77
78          if len(self.map) <= y_ or y_ < 0:
79              return False
80          elif len(self.map[0]) <= x_ or x_ < 0:
81              return False
82          elif self.map[y_][x_] == "Wall":
83              return False
84
85          return True
86
```

Figure 16: Additional methods in GridWorld class

As its name suggests, the *get_reward* method returns the reward after an agent goes to the specified state. Similarly, the *is_valid_action* checks if the action executed by the agent is valid; if invalid, the agent will remain in its current state after executing the action.

6.2 Value Iteration

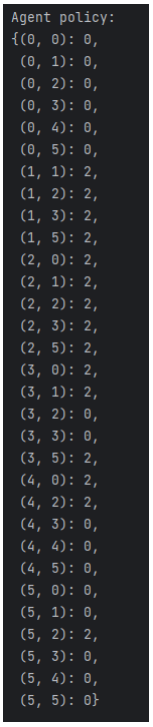


Figure 17: Console output of policy

The above figure shows the corresponding console output of the plot to Figure 4.

```
Values for each state:
[[99.90068522  0.         94.89609657 94.76432259 93.53160891 93.1617994 ]
 [99.32832311 97.76231894 95.44108761 94.24181132  0.         92.74707229]
 [97.86886722 96.49303648 95.1475818  94.05853676 92.93338125 92.66318125]
 [96.46011866 95.34764866 94.11537757 92.94639889 92.68291459 91.70683124]
 [95.20626038  0.          0.          0.         91.36372876 91.42267075]
 [93.81732594 92.5964202  91.39073772 90.20008869 89.4048257  90.14077672]]
```

Figure 18: Console output of utilities for each state

The above figure shows the corresponding console output of the plot to Figure 5.

6.3 Policy Iteration

```
Agent policy:
{(0, 0): 0,
 (0, 1): 0,
 (0, 2): 0,
 (0, 3): 0,
 (0, 4): 0,
 (0, 5): 0,
 (1, 1): 2,
 (1, 2): 2,
 (1, 3): 2,
 (1, 5): 2,
 (2, 0): 2,
 (2, 1): 2,
 (2, 2): 2,
 (2, 3): 2,
 (2, 5): 2,
 (3, 0): 2,
 (3, 1): 2,
 (3, 2): 0,
 (3, 3): 0,
 (3, 5): 2,
 (4, 0): 2,
 (4, 2): 2,
 (4, 3): 0,
 (4, 4): 0,
 (4, 5): 0,
 (5, 0): 0,
 (5, 1): 0,
 (5, 2): 2,
 (5, 3): 0,
 (5, 4): 0,
 (5, 5): 0}
```

Figure 19: Console output of policy

The above figure shows the corresponding console output of the plot to Figure 10.

```
Values for each state:
[[99.99985837  0.         95.0453156  93.87485904 92.65447219 93.32783829]
 [98.39321988 95.88287575 94.54485674 94.39757192  0.         90.9171839 ]
 [96.94835855 95.58628612 93.29428598 93.17612334 93.10219816 91.79458046]
 [95.55369747 94.45235217 93.23240379 91.11509146 91.81417792 91.88773216]
 [94.31237778  0.          0.          0.         89.54805947 90.56623357]
 [92.93733269 91.728636   90.53501034 89.3562678  88.56848786 89.29683174]]
```

Figure 20: Console output of utilities for each state

The above figure shows the corresponding console output of the plot to Figure 11.

References

[1] P Read Montague. "Reinforcement learning: an introduction, by Sutton, RS and Barto, AG". In: *Trends in cognitive sciences* 3.9 (1999), p. 360.

[2] Greg Brockman et al. "Openai gym (2016)". In: *arXiv preprint arXiv:1606.01540* 476 (2016).

[3] Steve Roberts. *Policy and value iteration*. Aug. 2022. URL: https://towardsdatascience.com/policy-and-value-iteration-78501afb41d2#:~:text=In%20Policy%20Iteration%2C%20at%20each,be%20the%20estimated%20state%20value..

[4] Stuart Russell and Peter Norvig. "Artificial Intelligence: a modern approach, 4th US ed". In: *University of California, Berkeley* (2021).

[5] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[6] David Silver et al. "Deterministic policy gradient algorithms". In: *International conference on machine learning*. Pmlr. 2014, pp. 387–395.

[7] Tuomas Haarnoja et al. "Soft actor-critic algorithms and applications". In: *arXiv preprint arXiv:1812.05905* (2018).