Assignment 1

Part 1 Value Iteration

Descriptions of implemented solutions

The implementation is done in python. It follows strictly to the Bellman update formula given in the lecture notes and textbook:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s') \quad \dots\dots equation\ (1)$$

Although there is another alternative but equivalent way to formulate the method (I saw the formulation while I was doing research online). Basically, the alternative formulation takes the reward not as the reward of the current state, but rather the rewards of the possible states for next iteration, therefore during the update it takes the maximum of the expectation value of the sum of reward plus the discounted utility of the next state given different action:

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')] \quad \dots\dots equation\ (2)$$

Back to the code itself, there are several main blocks operation for value iteration technique, which is easier to work with if they are treated as functions. As such, there are three functions in the code to facilitate the computation. They are Next_state_space, Trans_prob and Qoptimal.

Next_state_space takes in current state and a given specific action and output all possible states due to that action. The current state is described by indices i and j which are the indices of a nested for loop iterating through a 2-dimensional matrix, which has exactly the same dimensions as the given grid world. Action is picked from a list 'Actionspace', in which all possible actions – 'up', 'down', 'left' and 'right' are stored.

Trans_prob takes the current state, a specific action and one of the possible next state as inputs and output the corresponding probability of transiting from the given current state to the given next state, through the given action. The action must be given otherwise the output value may not be the same, as it is possible to landing on the same next state from the same current state through a different action.

Finally, Qoptimal takes the current state, the whole actionspace, a dictionary named 'policy_space' and another two-dimensional matrix called 'value'. 'Value' also has the same dimensions as the grid world. Each of its entry is the corresponding expected utility of one state in the grid world. For value iteration, the entries are initialized with '0' value. 'policy_ space' is effectively the policy for the agent to take action. The keys of 'policy_ space' are every available state in the grid world (some states are not

available for agent to transit to, i.e. wall) and the values of keys are the actions the agent is supposed to take on that corresponding state(key). At the beginning, 'policy_ space' is an empty dictionary as the agent has not learnt an policy yet. The output of Qoptimal is 'Qopt' and 'policy_space'. 'Qopt' is the highest expected utility. It is obtained by iterating through all the actions in 'actionspace' to compute the corresponding expected utilities and pick the highest. The corresponding action of achieving the highest utility is considered to be the optimal policy of the current iteration. Since there are possibilities of getting the same highest expected utility by taking different actions, in such event all such action will be recorded to update 'policy_ space'. In other words, the optimal policy may not be unique.

There is another variable passed into each function, called 'reward'. Reward is also a matrix of the same dimensions as the grid world. Each of its entry correspond of the reward of each state of the grid world. For those entries corresponding to 'wall', rewards are defined to have value '9' to be distinguished of other rewards (for all intents and purposes, the choice of 'wall' reward value is arbitrary). 'Reward' is passed through all functions as it is used in every function.

To carry out value iteration, we update the value of each entry of 'value' matrix. It is achieved by another nested loop. The update rule is such that if the reward value of that entry is not 9 (namely not a 'wall' state), we do the following:

```
Qopt,policy_space=Qoptimal(i,j,actionspace,policy_space,value)
value[i,j]=reward[i,j]+discount*Qopt
```

the first line is to obtain the highest expected utility and the updated 'policy_space'. The second line uses the highest expected utility to update the 'value' matrix according to equation (1), where U_{i+1} is the updated value and

$$\max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

is Qopt. After iterating though the entire 'value' matrix, it is considered as one round of update. We need to repeat the whole process until all the entries of 'value' matrix is converged. Following the estimation method in the textbook, we set an error threshold ε, which itself is determined by an empirical scaling of the maximum reward possible. If the absolute difference between U_{i+1} and U_i is less than $\varepsilon(1-\gamma)/\gamma$, then we terminate the iteration. By trial and error, I obtained that with $\varepsilon = 0.01$, after 917 rounds of such process, the 'value' matrix is roughly converged. The corresponding 'value' matrix and 'policy_space' are:

99.99		95.04	93.87	92.65	93.32
98.38	95.87	94.54	94.39		90.91
96.94	95.58	93.28	93.17	93.09	91.79
95.54	94.44	93.22	91.11	91.81	91.88
94.3				89.54	90.56
92.93	91.72	90.53	89.35	88.56	89.29

Figure 1 Approximately Converged Value (Expected Utility) Table

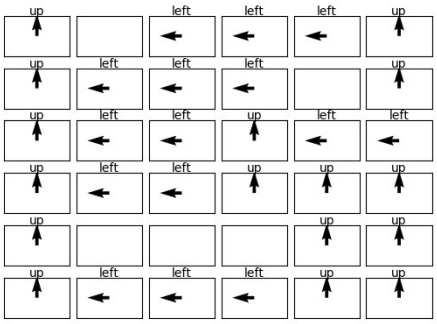

Figure 2 Optimal Policy

Futher increasing the number of iterations only changes the value from third decimal onwards for the initial state [4,3]. We can observe the utility estimates versus number of iteration plot of initial state and several other states:

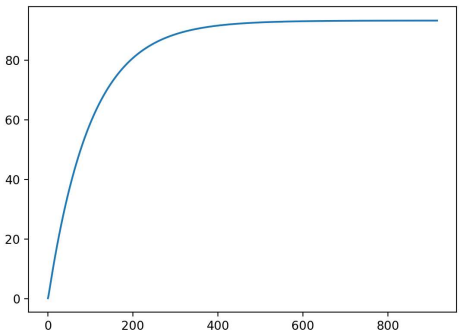

Figure 3 Initial State [4,3] (-0.04 reward)

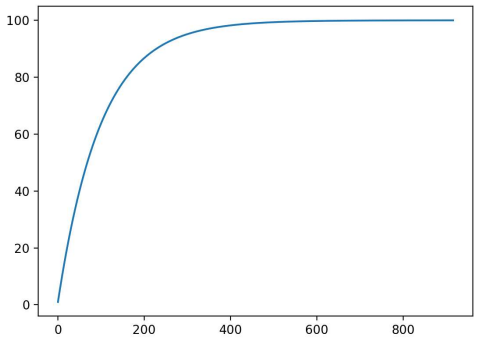

Figure 4 State [1,1] (Top-Left Corner, 1.0 reward)

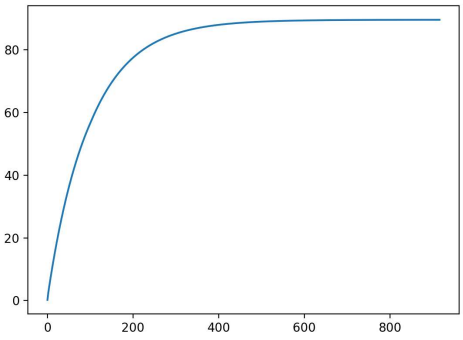

Figure 5 State [5,5] (-1 reward)

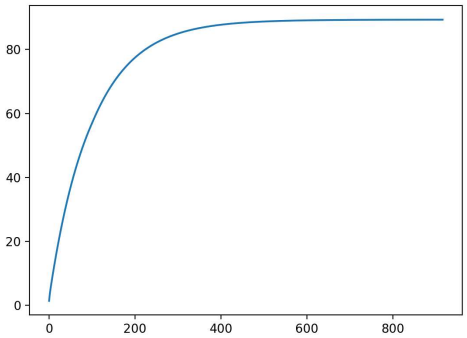

Figure 6 State[6,6] (-0.04 reward)

For all intents and purposes, it is safe to conclude that the utility of all states can be approximated with the utility values at 917th iteration and the optimal policy is the policy of that same iteration.

However, we can obtain the same 'policy_space' way before 'value' matrix converged:

65.19		61.56	60.72	59.83	60.99
63.98	61.86	60.96	61.23		58.91
62.93	61.94	60.03	60.38	60.68	59.77
61.92	61.15	60.27	58.67	59.75	60.18
61.01				57.84	59.18
60.01	59.13	58.26	57.41	57.19	58.23

Figure 7 Value (Expected Utility) Table at 105th Iteration

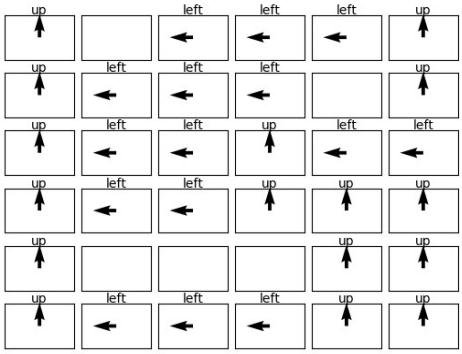

Figure 8 Policy at 105th Iteration

This is just after 105 rounds of iteration. By comparison, the 'policy_space' the identical and 'value' matrix is very different.

This is round 104:

64.84		61.23	60.4	59.51	60.68
63.63	61.52	60.62	60.9		58.6
62.58	61.6	59.69	60.05	60.35	59.45
61.58	60.81	59.93	58.34	59.43	59.87
60.67				57.53	58.87
59.68	58.8	57.94	57.08	56.88	57.92

Figure 9 Value (Expected Utility) Table at 104th Iteration

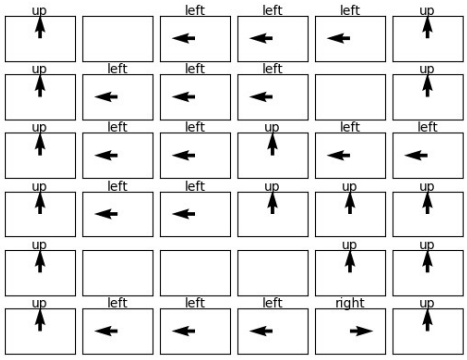

Figure 10 Policy at 104th Iteration

It can be observed that the 'policy_space' is different from the converged version by one different action at state [6,5].

However, whether 'policy_space' changed from 106th iteration to 499th iteration remains unclear. It is possible that some actions changed but changed back again before 500th iteration.

Another interesting observation is that at 800th iteration, the state with the lowest expected utility is not a state with lowest reward (-1); rather it is a state with reward -0.04. After further thinking, it is possible as the reward of some state is given only when the agent comes from other states to that state. In some sense, the neighbouring states of states with reward -1 is more dangerous than that state per se. However, the situation gets more involved when other factors in the topology of the grid world is accounted for. We can't get a straightforward intuition as which states have higher expected utilities. The only when is to compute through either policy iteration or value iteration.

Part 2 Policy Iteration

Descriptions of implemented solutions

Policy iteration is described by the following update rule:

$$\pi^{i+1}(s) = \arg\max_{a \in \mathrm{A(s)}} \sum_{s'} P(s'|s,a) U^{\pi_i}(s') \quad \dots\dots equation\ (3)$$

It means that we update the current policy π by finding the set of actions that can maximize the expected utility which is obtained based on the current policy π.

The relation between the utility of the current policy and the current policy itself can be expressed as:

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi(s)) U^{\pi}(s') \quad \dots\dots equation\ (4)$$

It can be solved either by repeatedly updating the expected utility U^{π} until it approximately converges or by solving a system of linear equations. We chose the latter method because it gives closed-form solution.

In terms of coding, those three functions used in value iteration are used here as well. The differences are as follow:

Instead of having an empty 'policy_space', we define an initial 'policy_space' for which dictates what action to take for each state. The choice of actions is arbitrary. Here, we initialize it such that for all states, the action to take is to go 'up'. Based on this policy, we can form a system of linear equation for each available state, all follow the form of equation 4. As mentioned before, we need to solve this set of equations to find the expected utility of the initial policy. Since the system comprises of the same number of questions as the same number of unknown variables (expected utility for each state), both of which are equal to the number of available states (excluding 'wall' state), the linear system is perfectly determined and can be solved. An orthodox way would be using Gaussian elimination, but in practice, it was solved here by forming a corresponding matrix with the coefficients of all unknown variable, taking its inverse and multiplied with the column vector formed by the constants in the linear system. To illustrate, let the expected utilities of each state be $x_0, x_1, \dots, x_{30}$(there are 31 available states for the given question). Rearrange equation 4, we have:

$$\begin{gathered} c_{0,0}x_0 + c_{0,1}x_1 + \cdots + c_{0,31}x_{31} = c_{0,32} \\ c_{1,0}x_0 + c_{1,1}x_1 + \cdots + c_{1,31}x_{31} = c_{1,32} \\ \vdots \\ c_{31,0}x_0 + c_{31,1}x_1 + \cdots + c_{31,31}x_{31} = c_{31,32} \dots\dots equations\ (5) \end{gathered}$$

The matrix form of this linear system is:

$$\begin{bmatrix} c_{0,0} & \cdots & c_{0,31} \\ \vdots & \ddots & \vdots \\ c_{31,0} & \cdots & c_{31,31} \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_{31} \end{bmatrix} = \begin{bmatrix} c_{0,32} \\ \vdots \\ c_{31,32} \end{bmatrix} \ldots\ldots \ equations\ (6)$$

The closed-form solution is:

$$\begin{bmatrix} x_0 \\ \vdots \\ x_{31} \end{bmatrix} = \begin{bmatrix} c_{0,0} & \cdots & c_{0,31} \\ \vdots & \ddots & \vdots \\ c_{31,0} & \cdots & c_{31,31} \end{bmatrix}^{-1} \begin{bmatrix} c_{0,32} \\ \vdots \\ c_{31,32} \end{bmatrix} \ldots\ldots \ equations\ (7)$$

As such we can obtain the expected utilities of the initial policy.

In terms of coding, a dictionary called position_pointer is employed to map the states in a 2-dimentional matrix to a 1-dimensional array to form linear equations. It is used in conjunction with next_state_space and trans_prob functions to create a list of dictionaries in the same order as the equations (5) first and then put in the c-matrix in equation (6).

Now, use equation 3 to evaluate which action can maximize the utility of each state. Then update the initial policy which those actions. Again, as in value iteration, the result might not be unique. It was coded in such a way that all possible optimal actions (policies) are added the dictionary as a list of strings.

Repeat the aforementioned procedure until 'policy_space' no longer changes. In this implementation, a while loop is used and it loop continues until the updated policy is the same as that of the previous iteration. In this case, unlike value iteration, it is guaranteed that the policy has converged. It is in fact quite straight forward to see why. If policy is no longer updated, the expected utility of each state under that policy will also not change, since the corresponding linear system is identical (hence identical solution). Therefore, the best action to take with the same expected utilities will also not change. In the case that initial policy is all state go 'up', it only took 4 iterations for the policy to converge (policy_space of 4th iteration is the same as that of the 5th iteration:

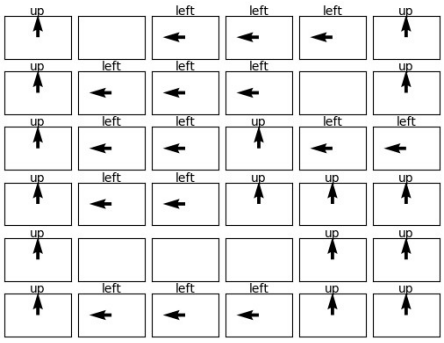

Figure 11 Optimal Policy by Policy Iteration

By comparison, it is identical to the solution obtained with value iteration.

100.0		95.05	93.88	92.65	93.33
98.39	95.88	94.54	94.4		90.92
96.95	95.59	93.29	93.18	93.1	91.79
95.55	94.45	93.23	91.12	91.81	91.89
94.31				89.55	90.57
92.94	91.73	90.54	89.36	88.57	89.3

Figure 12 Converged Expected Utility Table

The expected utilities of policy iteration are the true converged expected utilities. Compared to figure 1, the values at most differ by 1 at second decimal place. The means the expected utilities obtained by value is iteration with the defined stopping criteria is close enough.

We check the utility estimates of the same states as those chcekd in value iteration:

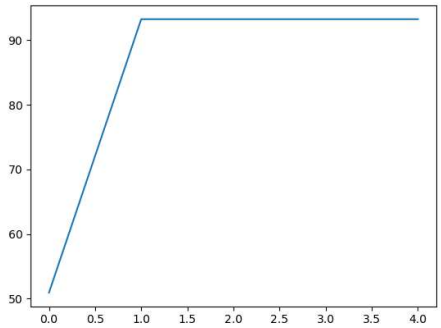

Figure 13 Initial State [4,3] (-0.04 reward)

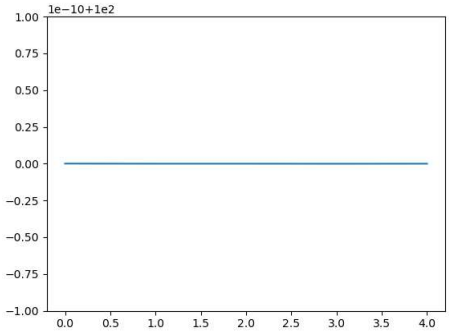

Figure 14 sState [1,1] (Top-Left Corner, 1.0 reward)

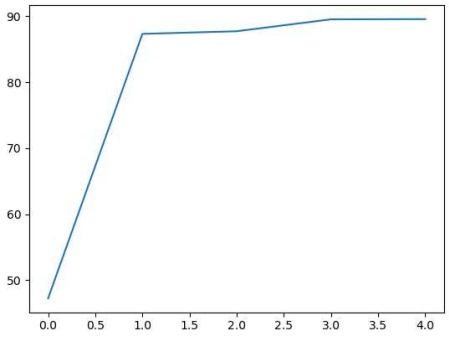

Figure 15 State [5,5] (-1 reward)

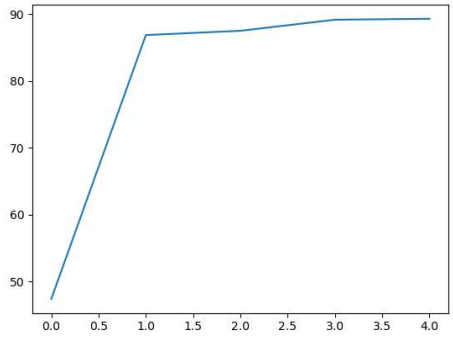

Figure 16 State [6,6] (-0.04 reward)

The utility of appeared in the plots above are those obtained by solving the linear system of equations. Therefore, iteration 0 corresponds to the initial policy given, iteration 1 corresponds to the first updated policy, so on and so forth.

In general we can see that the utility update of policy iteration is not as smooth as that of value iteration. This is expected, however, as it needs far less iteration to converge.

Note that in Figure 14, 0.00 corresponds to 100 due to its notation and despite looking almost identical, Figure 15 and Figure 16 have different utiltity values:

Figure 15: [47.2600427957861, 87.31343226398035, 87.71105302584384, 89.51627559522802, 89.54841310017954]
Figure 16: [47.37368644662352, 86.85734733050158, 87.5004632924173, 89.15492188535251, 89.29769058834738]

Last but not least, the last value of utility in each plot is the truly converged value. That is, if we further extend the plots, they will continues as perfectly falt lines. For the same reason, despite looking flat, the segment of Figure 13 plot and Figure 14 plot actually do not continue with the same value:

Figure 13: [50.9419130119095, 93.23254542196318, 93.23254542196317, 93.23254542196315, 93.23254542196317]

Figure 14: [100.00000000000007, 100.0, 100.0, 99.99999999999996, 100.0]

Part 3 More Complicated Grid World

Design:

-5		+1	+3	wall	-1	-2	-3	-1	-5
	-3		-5	wall	+2	wall	wall		
+1	wall					wall		+1	
	wall	+2	+1	wall	-2	wall	+1		
		+3	T1	+3	-2	T2	-2	+3	-2
	wall		+2	+2	-1	-2	wall		wall
	+3	+1			-3		+1		-1
	-2		-2	wall		-2		wall	
	+3	wall		+3			+3	-2	
-5		+3			+4		-2		-5

Figure 17 New Grid World

For those states didn't get labelled, the rewards are -0.1. 'T_1' and 'T_2' are two terminal states. If the agent enters those states, the game is over. 'T_1' has a reward of -20, 'T_2' has a reward of +20, 'Wall' states behave the same as before. The discount factor is set to be 0.9.

The above new grid world is more complex than the given one in the following aspects:

1. There are more number of available states for the agent to be in;
2. There are more variety of the rewards given for different states, ranging from -20 to 20;
3. There are two terminal states, for which if the agent steps in, the game will be over. However, one of the terminal states has reward far greater than any other state. The agent needs to gauge if staying in game or end game at certain point of time will result in more overall rewards gained;
4. There are many states with positive rewards surrounding the terminal states with -20 rewards. The agent can easily gain a lot of rewards there but it will risk stochastically stepping in the -20 terminal states;

Now there are some modification to be made for the original code in order for it to run on the new environment. Besides re-defining the 'value' and 'reward matrix, the most prominent change is adding in the terminal states. The difference of terminal states from other available states is that their expected value remains the same as their rewards throughout the game because it has no next state. This can be achieved by modifying Qoptimal function in both value iteration and policy iteration. For policy iteration, however, there are a few more things to be changed. For initialization of policy_space dictionary, since it is used to construct the linear system and the two terminal states do appear in the system (technically they even have their own respective equation with just rewards and no other variables, i.e. $U_{T_1} = -20$ and $U_{T_2} = 20$), they need to be defined in the dictionary. Therefore, we initialize the action on those two states to be 'nil', representing null action. For the same reason, the two terminal states need to be defined in position_pointer as well (which helps to map the states in a 2-dimensional matrix onto a 1-dimensional array to construct linear system). When creating the dictionary for next states and their respective transition probability, those two terminal states have empty dictionaries as placeholders and skipped Trans_prob and Next_state_space functions.

Since the R_{max} is 4 times of that in the given setup (excluding the terminal rewards) and we set the scaling factor to be the same, $\varepsilon = 0.04$. The result of value iteration is shown below:

12.6	21.21	25.02	27.62		11.65	7.79	3.21	7.82	3.29
13.3	15.38	20.96	18.43		15.14			11.44	10.13
13.24		18.06	16.06	13.98	13.25		12.6	13.33	11.51
13.69		18.45	15.63		10.51		14.4	14.12	12.18
15.73	14.93	17.15	-20.0	16.83	14.74	20.0	15.05	16.42	12.0
18.16		19.29	19.3	18.94	15.52	15.16		14.61	
20.82	24.51	22.11	19.25	16.99	17.36	17.55	19.53	16.79	13.57
18.24	19.02	19.22	18.77		23.97	19.9	21.44		14.02
20.24	24.15		24.11	28.2	27.94	24.74	24.76	19.32	16.1
14.55	22.81	25.95	24.63	28.26	32.34	27.92	22.34	19.48	11.51

Figure 18 Approximately Converged Value (Expected Utility) Table

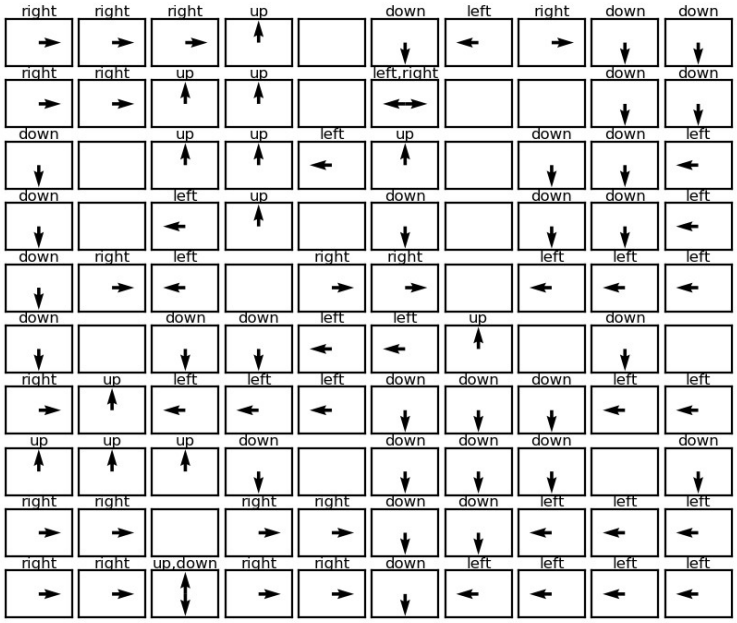

Figure 19 Optimal Policy

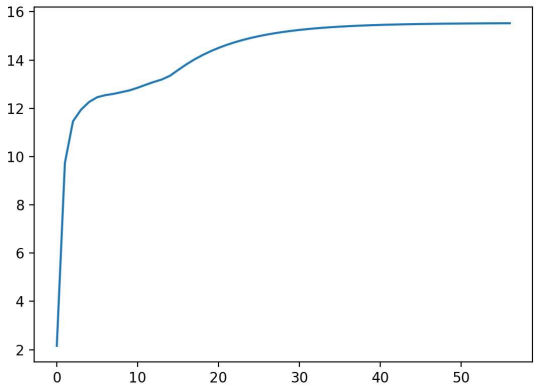

Figure 20 State [6,6] (-1 reward, $\varepsilon = 0.04$)

Surprisingly, despite the complexity of the new setup, it took much less iterations to converge (a bit more than 50 steps). This might be due to the fact that ε is larger now. However, when we impose $\varepsilon = 0.01$, the plot becomes:

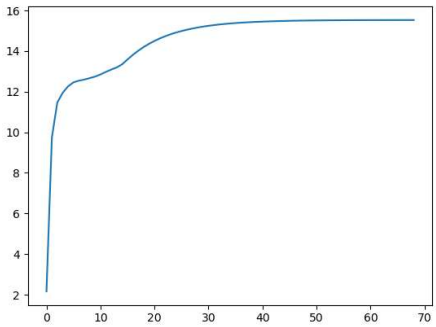

Figure 21 State [6,6] (-1 reward, $\varepsilon = 0.01$)

which did not extend by a whole lot. This might just be a coincidence. We plot a few more utility estimate to see if that is the case:

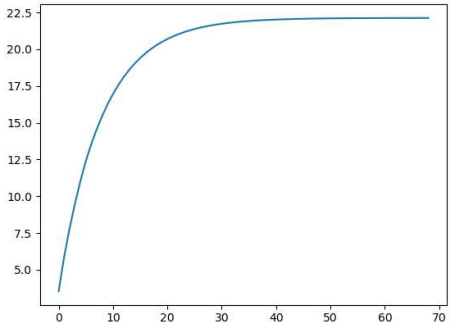

Figure 22 State [7,3] (+1 reward, $\varepsilon = 0.01$)

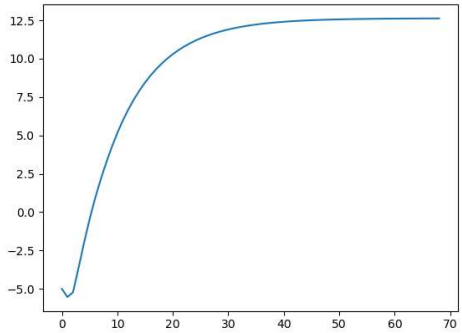

Figure 23 State [1,1] (-5 reward, $\varepsilon = 0.01$)

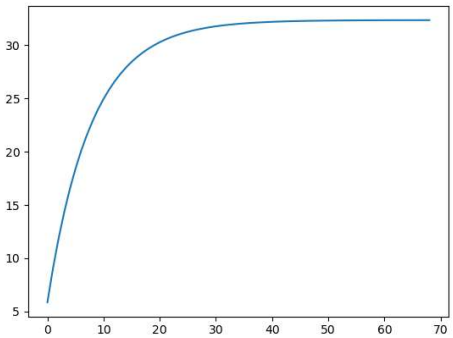

Figure 24 State [10,6] (+4 reward, $\varepsilon = 0.01$)

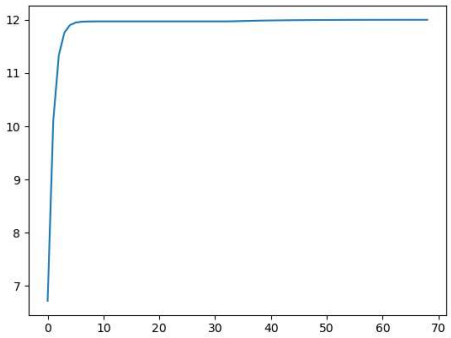

Figure 25 State [5,10] (-2 reward, $\varepsilon = 0.01$)

Apparently, quite a few states converged within 70 iterations with $\varepsilon = 0.01$. This could imply that sometimes, given a complex envirnment, the convergence is not necessarily slower than that of a simpler environment.

Further more, the utility estimate plot of state [6,6] is more as smooth as that for the given envirnment. This is very likely due to the fact that [6,6] is in the vicinity of a terminal state and hence the consequently arised complexty of the situtaion in different iterations. Most notably, for state [2,6] and [10,3] we now have a state with two competing optimal actions. This proved that the optimal policy indeed may not be unique.

Further observation tell us that there is no unique strategy to win under this setting. We can see in the states near T_1 the strategy is to get further away from T_1, and for quite a few states in the top right quadrant, following the optimla policy leads to T_2. But at some states far from the terminal states, say [1,4] and [10,6] and states in their vicinities, the optimal policy leads to stuck at those two states. It means it is not always good to go to T_2, collect 20 rewards and end game; but if the agent is near T_2 then that is the best shot.

The following is the results of policy iteration:

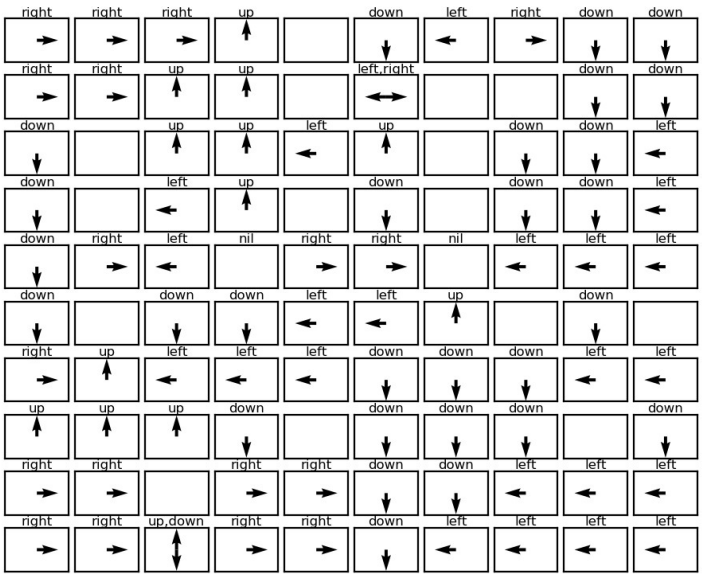

Figure 26 Optimal Policy by Policy Iteration

As we can see the optimal policy obtained through policy iteration is identical to that of value iteration. This time policy converged at the 5th iteration (6th iteration has the same policy as that of the 5th). So in this case policy iteration, still converged way faster that value iteration, despite value iteration converged faster than expected. Compared to the simpler grid world in part 1, policy iteration converged slower by one iteration, which is not too significant. On a sidenote, the 'nil' above two empty states are due to the fact that we initilized the actions of the two terminal states to be 'nil'.

12.63	21.24	25.06	27.66		11.66	7.8	3.21	7.82	3.29
13.33	15.41	20.99	18.47		15.16			11.44	10.13
13.25		18.09	16.08	14.0	13.27		12.61	13.34	11.51
13.7		18.47	15.65		10.51		14.4	14.12	12.18
15.75	14.95	17.16	-20.0	16.84	14.74	20.0	15.05	16.42	12.0
18.17		19.31	19.31	18.95	15.53	15.16		14.63	
20.84	24.52	22.13	19.26	17.0	17.38	17.57	19.55	16.8	13.58
18.26	19.03	19.23	18.8		23.99	19.92	21.46		14.03
20.26	24.17		24.13	28.22	27.97	24.76	24.77	19.33	16.12
14.58	22.84	25.98	24.65	28.28	32.36	27.94	22.36	19.49	11.52

Figure 27 Converged Expected Utility Table

By comparison, the values obtained by value iteration with the defined stopping criteria differ from that obtained by policy iteration from 2 decimal place onwards. We consider they are close enough.

Finally, we plot the utility estimates for the states plotted for value iteration as well:

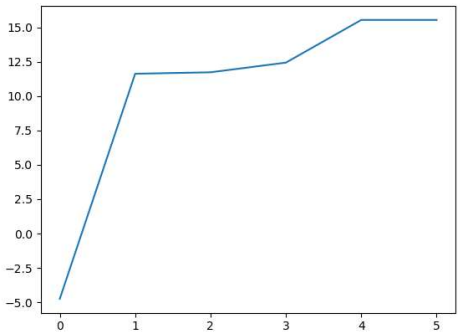

Figure 28 State [6,6] (-1 reward)

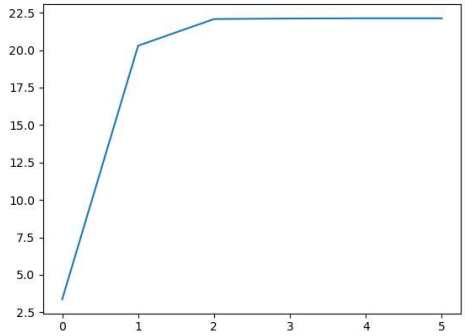

Figure 29 State [7,3] (+1 reward)

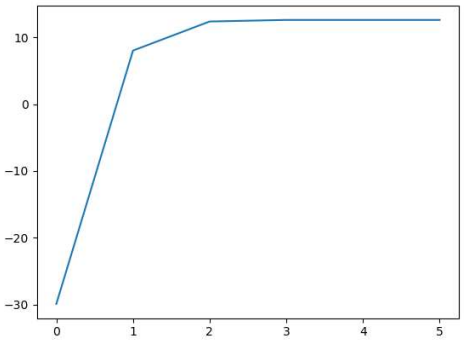

Figure 30 State [1,1] (-5 reward)

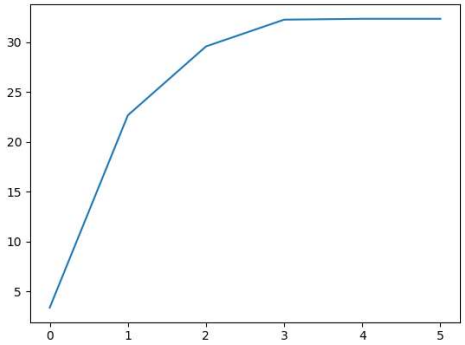

Figure 31 State [10,6] (+4 reward)

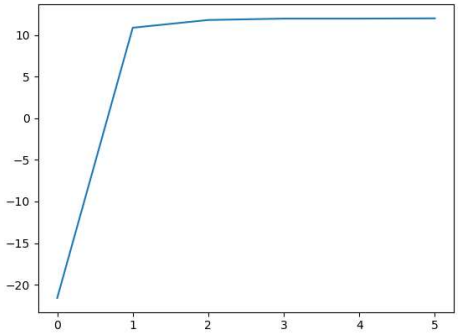

Figure 32 State [5,10] (-2 reward)