SC4003 Intelligent Agents

Assignment 1

Derrick Ng Choon Seng

U2122873F

Table of Contents

1. Introduction

By implementing the Markov Decision Process (MDP) framework, we can model decision-making in environments through an agent. Reinforcement Learning algorithms such as value iteration and policy iteration will be explored in this assignment in a GridWorld maze environment which consists of different elements such as positive, negative, wall and empty states. In the first part of this assignment, value iteration and policy iteration are implemented to compute the optimal policy and utilities of all states. The utilities of each state are then plot against the number of iterations to observe the convergence of the utilities. This assignment was implemented in Python, and codes used can be found in the appendix.

2. MDP Formulation

Environment/States

Referring to Appendices A-1 and A-2, the GridWorld environment is created by using a 2-dimensional array. Each state in the environment can be assigned 1 of 4 possible values:

- **p**: corresponds to the green square that the agent can move to.
- **n**: corresponds to the brown square that the agent can move to.
- **w**: corresponds to the wall square that the agent cannot move to. If the agent's outcome is to move to the wall state, it would remain in the same state.
- '': corresponds to the white square that the agent can move to.

There are no terminal states, and the agent's state sequences are infinite.

Actions and Transition Probabilities

The agent can take 4 possible actions: Up, Down, Left and Right. The outcome of each action has a probability of 0.8 to go in the intended direction, and a probability of 0.1 that it goes perpendicular to the intended direction on either side, which is the transition probability

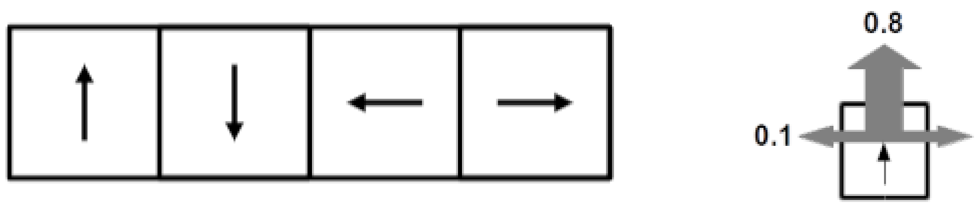


Figure 1: Actions and Outcome Probabilities

Rewards

At each possible state, the rewards are as follows:

- **p**: +1
- **n**: -1
- **w**: None
- **‘’**: -0.05

3. Part 1

A grid was created based on the maze environment provided in the assignment description, as per Appendix A-1, shown in Figure 2. Value iteration and policy iteration would then be used to obtain the optimal policy and utilities each state based on this environment.

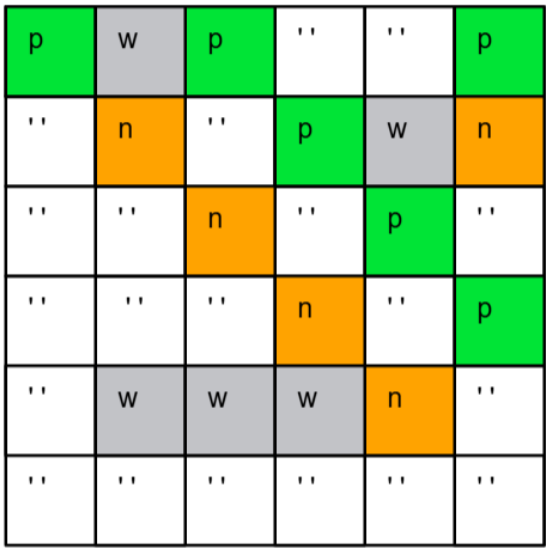


Figure 2: Diagram of Maze Environment

3.1. Value Iteration

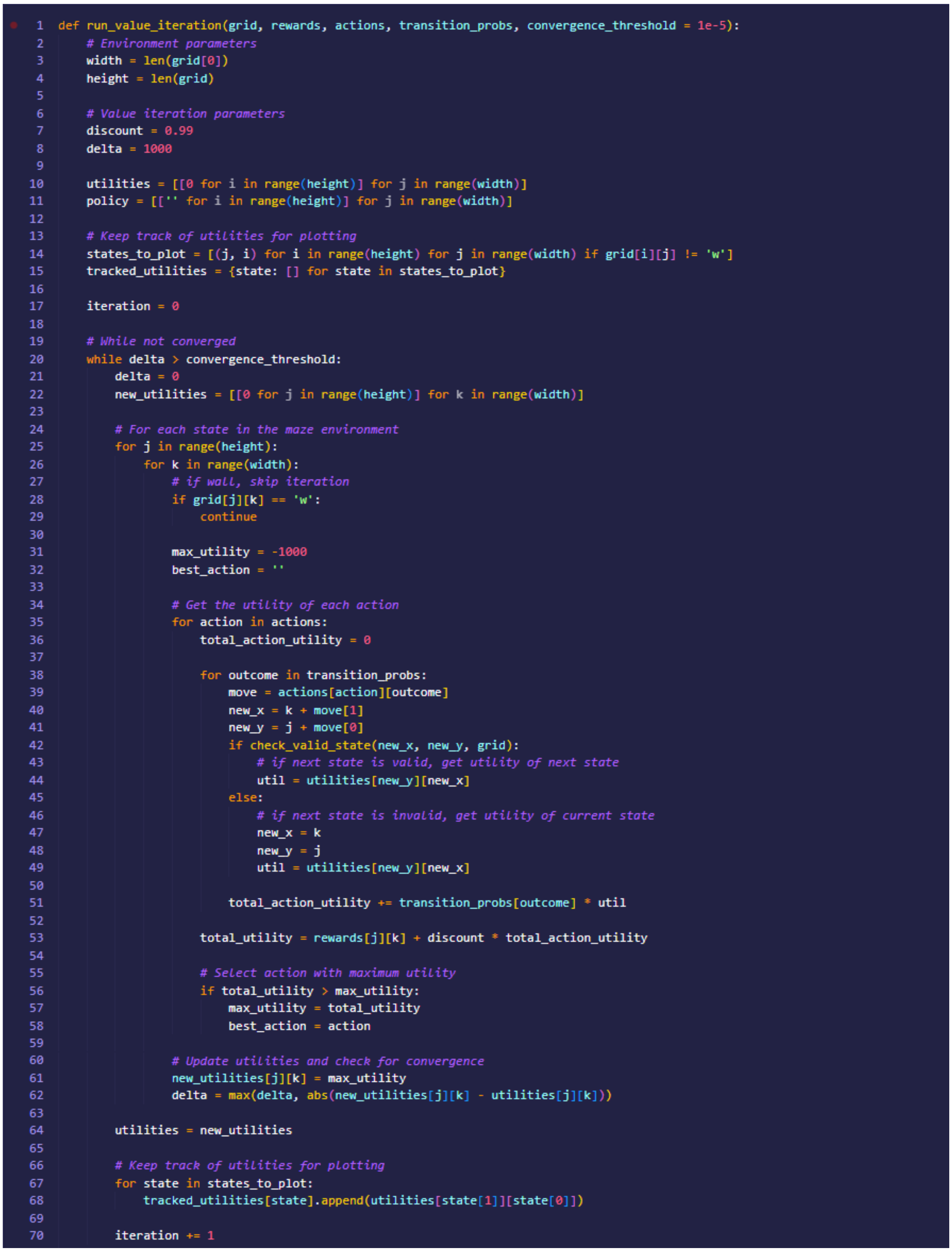
3.1.1. Description of Implemented Solution

Value iteration is an algorithm which iteratively improves the value estimates of each state by solving the Bellman Equation as shown in Equation 1. It considers the immediate rewards of being in the current state, and the future utilities of transitioning into the next state with a slight discount. The algorithm would iterate until the utilities converge. The optimal policy is then obtained by selecting actions for each state that leads to its neighbouring state with the highest utility estimate. Since there is no end state for this environment, convergence would be determined by the change in utility values from one iteration to the next, delta. Convergence would occur when delta is lower than a specified threshold value.

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

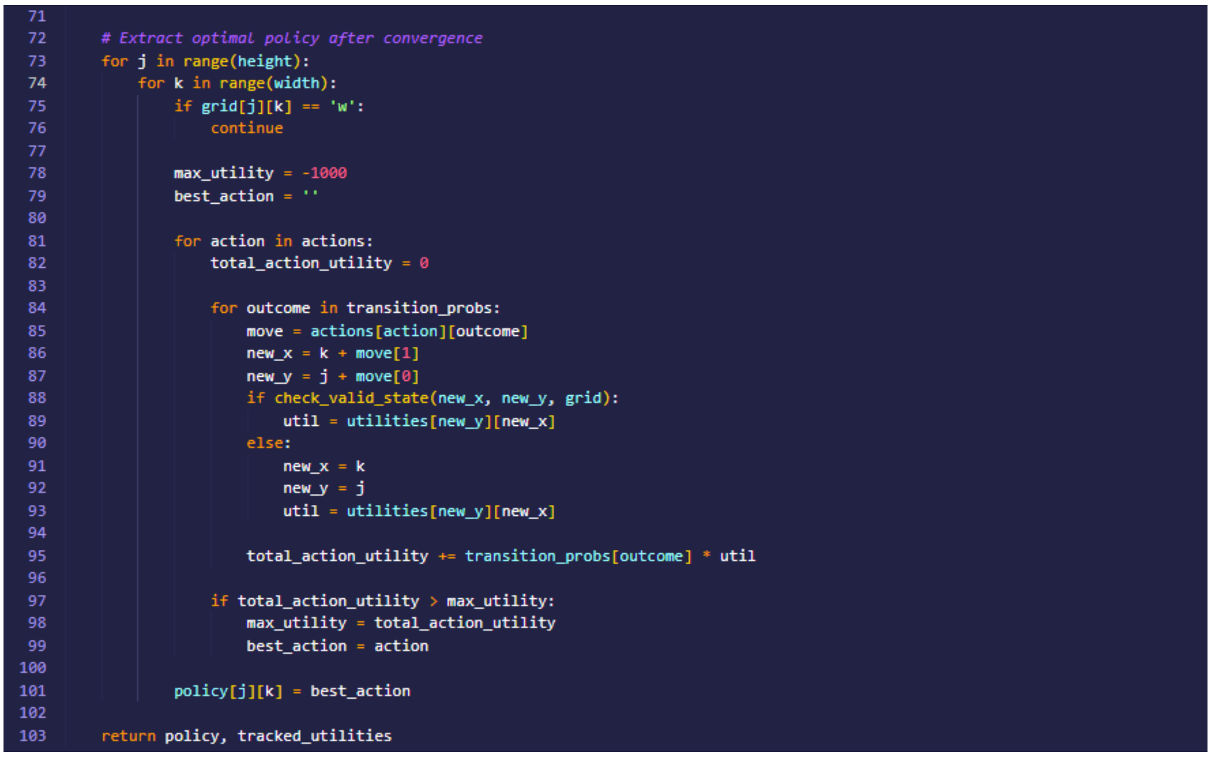
Equation 1: Value Iteration Update Equation

A discount factor of 0.99 was used as per the assignment description, and a convergence threshold value of 1e-5 was used, meaning the values converge when the difference in utilities between the previous and current iteration is less than 1e-5. Figures 3.1 and 3.2 are the implementation codes for value iteration used. Figure 3.1 shows the implementation of the main value iteration algorithm, and Figure 3.2 shows the extraction of the optimal policy after value iteration converges.



```python
def run_value_iteration(grid, rewards, actions, transition_probs, convergence_threshold = 1e-5):
    # Environment parameters
    width = len(grid[0])
    height = len(grid)

    # Value iteration parameters
    discount = 0.99
    delta = 1000

    utilities = [[0 for i in range(height)] for j in range(width)]
    policy = [['' for i in range(height)] for j in range(width)]

    # Keep track of utilities for plotting
    states_to_plot = [(j, i) for i in range(height) for j in range(width) if grid[i][j] != 'w']
    tracked_utilities = {state: [] for state in states_to_plot}

    iteration = 0

    # While not converged
    while delta > convergence_threshold:
        delta = 0
        new_utilities = [[0 for j in range(height)] for k in range(width)]

        # For each state in the maze environment
        for j in range(height):
            for k in range(width):
                # if wall, skip iteration
                if grid[j][k] == 'w':
                    continue

                max_utility = -1000
                best_action = ''

                # Get the utility of each action
                for action in actions:
                    total_action_utility = 0

                    for outcome in transition_probs:
                        move = actions[action][outcome]
                        new_x = k + move[1]
                        new_y = j + move[0]
                        if check_valid_state(new_x, new_y, grid):
                            # if next state is valid, get utility of next state
                            util = utilities[new_y][new_x]
                        else:
                            # if next state is invalid, get utility of current state
                            new_x = k
                            new_y = j
                            util = utilities[new_y][new_x]

                        total_action_utility += transition_probs[outcome] * util

                    total_utility = rewards[j][k] + discount * total_action_utility

                    # Select action with maximum utility
                    if total_utility > max_utility:
                        max_utility = total_utility
                        best_action = action

                # Update utilities and check for convergence
                new_utilities[j][k] = max_utility
                delta = max(delta, abs(new_utilities[j][k] - utilities[j][k]))

        utilities = new_utilities

        # Keep track of utilities for plotting
        for state in states_to_plot:
            tracked_utilities[state].append(utilities[state[1]][state[0]])

        iteration += 1
```

Figure 3.1: Value Iteration Code



```python

    # Extract optimal policy after convergence
    for j in range(height):
        for k in range(width):
            if grid[j][k] == 'w':
                continue

            max_utility = -1000
            best_action = ''

            for action in actions:
                total_action_utility = 0

                for outcome in transition_probs:
                    move = actions[action][outcome]
                    new_x = k + move[1]
                    new_y = j + move[0]
                    if check_valid_state(new_x, new_y, grid):
                        util = utilities[new_y][new_x]
                    else:
                        new_x = k
                        new_y = j
                        util = utilities[new_y][new_x]

                    total_action_utility += transition_probs[outcome] * util

                if total_action_utility > max_utility:
                    max_utility = total_action_utility
                    best_action = action

            policy[j][k] = best_action

    return policy, tracked_utilities
```

Figure 3.2: Value Iteration Code

3.1.2. Plot of Optimal Policy

After convergence, the optimal policy and actions are as shown in Figure 4. Refer to Appendix B-3 for the code outputs.

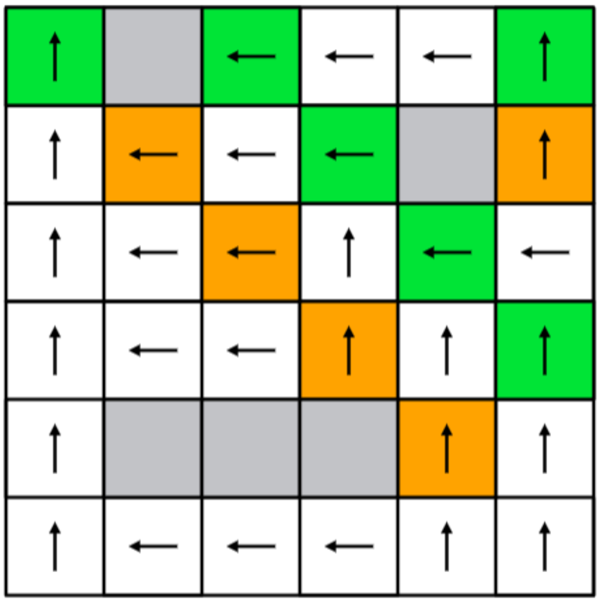
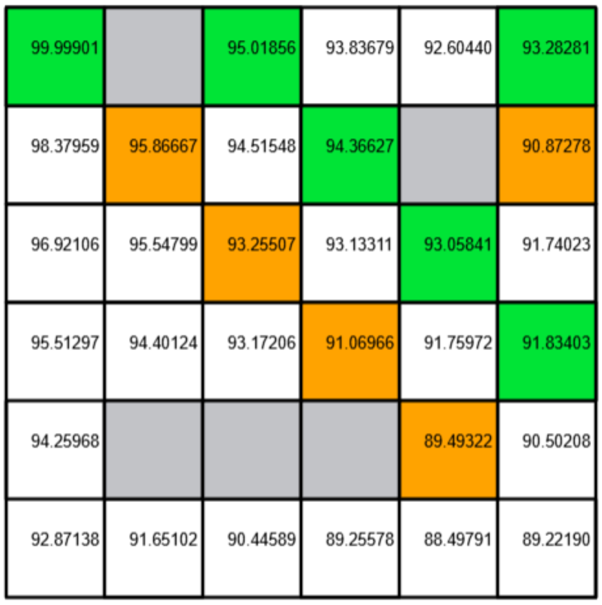


Figure 4: Optimal Policy after Value Iteration

3.1.3. Utilities of All States

After convergence, the final utilities for each state are observed in Figure 5. Values have been truncated to 5 decimal places. The top left grid has the highest utility value of 99.99901, which is because it can remain in the same state and accumulate rewards. By choosing to go up, all possible outcomes would lead to it remaining in the state. Refer to Appendix B-4 for the code outputs.



99.99901	▨	95.01856	93.83679	92.60440	93.28281
98.37959	95.86667	94.51548	94.36627	▨	90.87278
96.92106	95.54799	93.25507	93.13311	93.05841	91.74023
95.51297	94.40124	93.17206	91.06966	91.75972	91.83403
94.25968	▨	▨	▨	89.49322	90.50208
92.87138	91.65102	90.44589	89.25578	88.49791	89.22190

Figure 5: Plot of Utilities of Each State

3.1.4. Plot of utility estimates as a function of the number of iterations

As mentioned in Section 3.1.1, the convergence threshold value was set to 1e-5 or 0.00001. After 1147 iterations, the values converged and was considered optimal. However, from observing the plot in Figure 6, it seems that the values have started to stabilise after 600 iterations, with a maximum utility estimate of 100. The convergence likely took longer due to the higher accuracy of a smaller convergence threshold value.

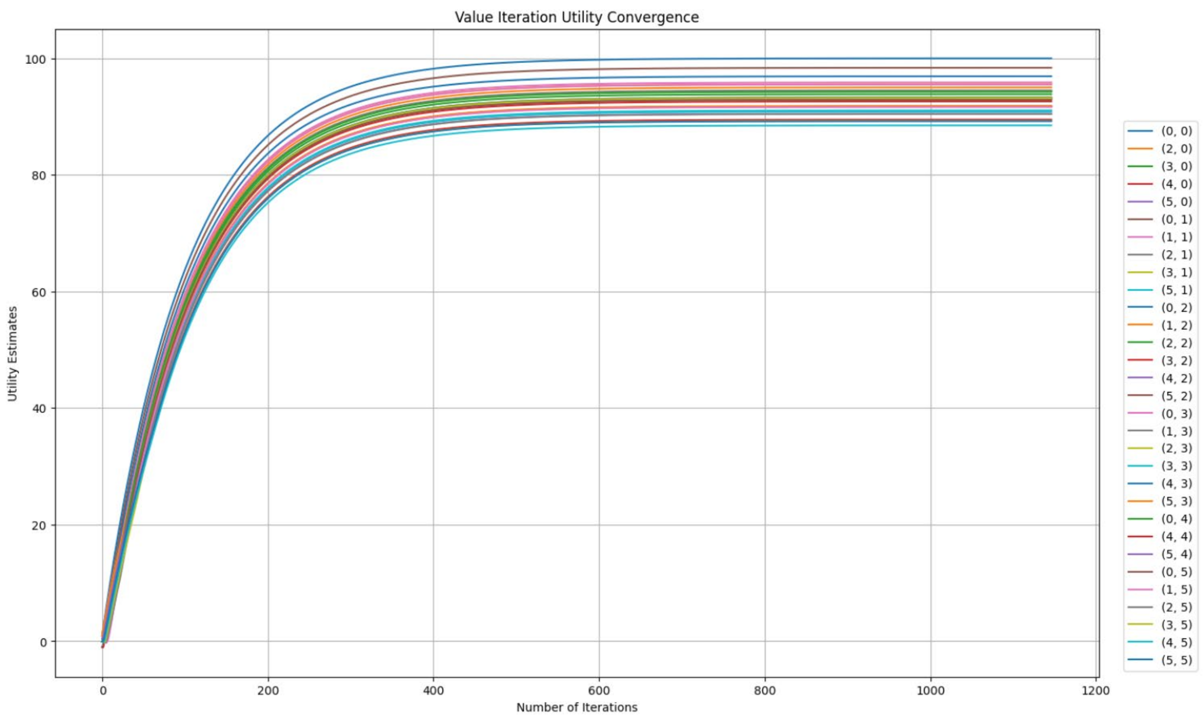


Figure 6: Plot of Utility Estimates against Iterations

3.2. Policy Iteration

3.2.1. Description of Implemented Solution

Policy iteration consists of 2 components, policy evaluation and policy improvement.

Policy evaluation is when the utilities for all states are calculated using the current policy by iterating until the utilities converge, as per Equation 2. Policy improvement is when the policy is updated by updating the optimal actions to take based on the new utilities calculated in policy evaluation, as per Equation 3.

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

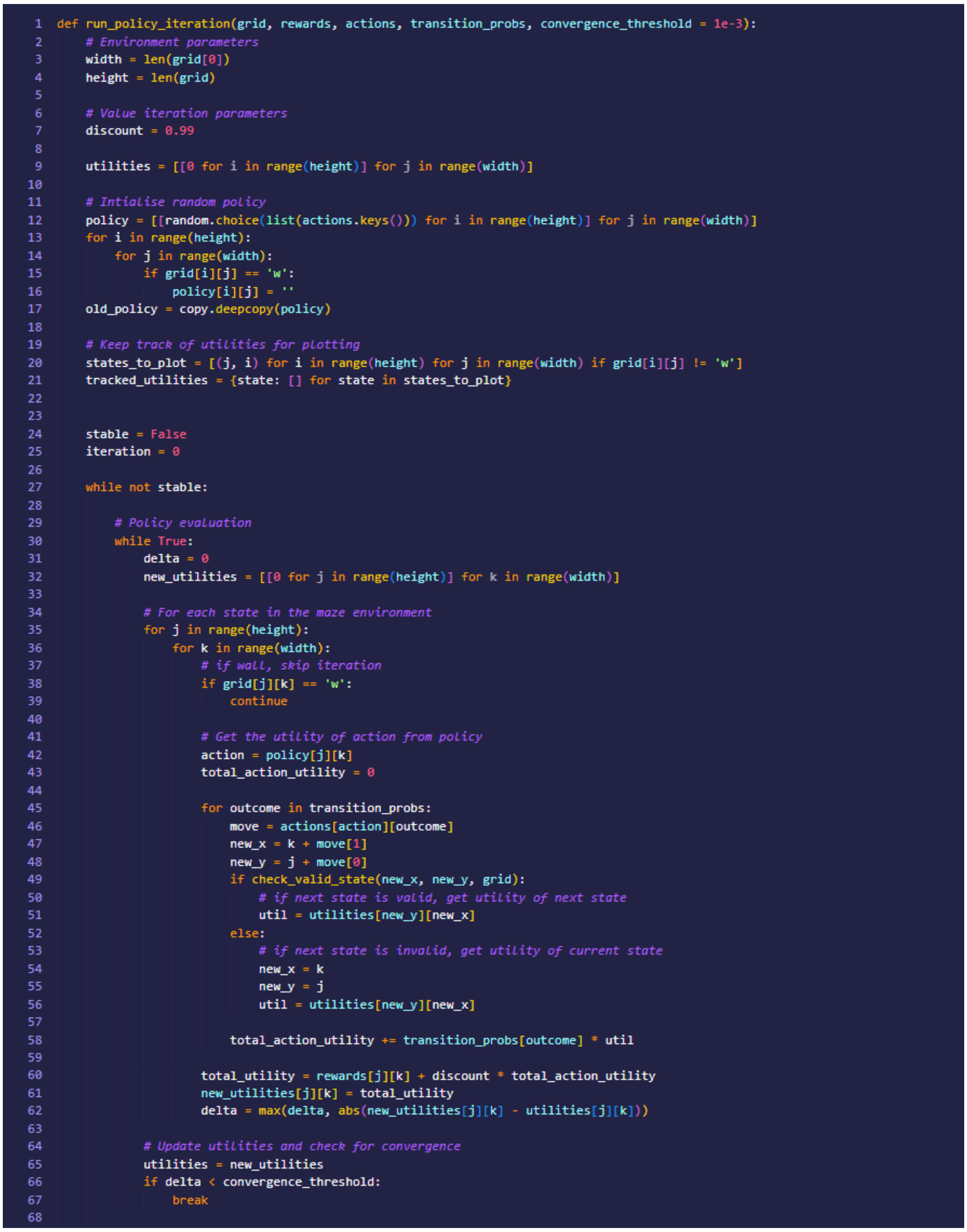
Equation 2: Policy Evaluation Equation

$$\pi^{i+1}(s) = \underset{a \in A(s)}{\arg\max} \sum_{s'} P(s' | s, a) U^{\pi_i}(s')$$

Equation 3: Policy Improvement Equation

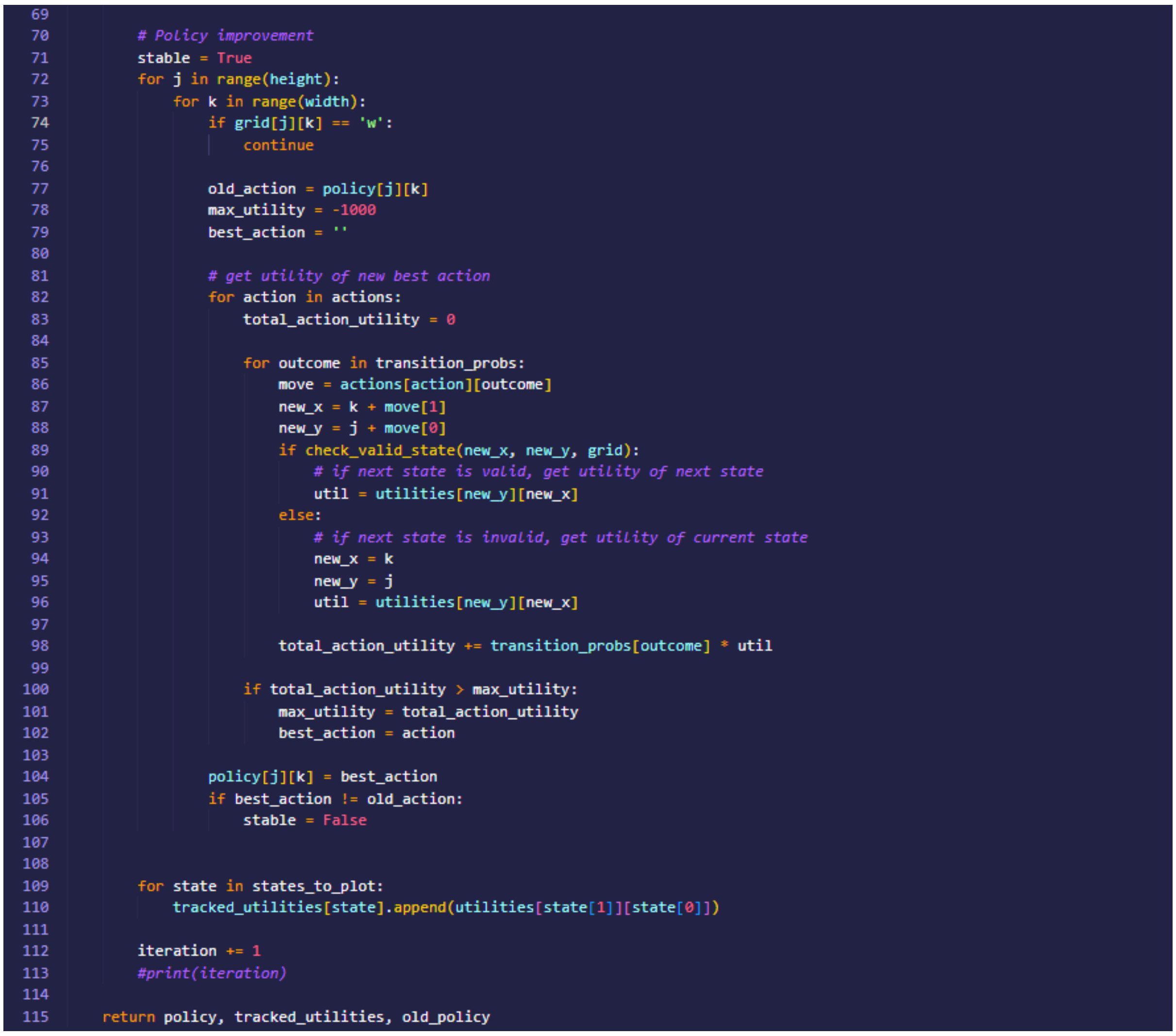
To implement policy iteration, a random policy is first initialised, then policy evaluation and improvement are repeated until there are no changes to the policy between the previous and current iteration. A convergence threshold of 1e-5 was used for the convergence of the policy evaluation component and a discount factor of 0.99 was also used as per the assignment description.

Figures 7.1 and 7.2 are the implementation codes for policy iteration used. Figure 7.1 shows the implementation of policy evaluation, and Figure 7.2 shows the implementation of policy improvement.



```python
def run_policy_iteration(grid, rewards, actions, transition_probs, convergence_threshold = 1e-3):
    # Environment parameters
    width = len(grid[0])
    height = len(grid)

    # Value iteration parameters
    discount = 0.99

    utilities = [[0 for i in range(height)] for j in range(width)]

    # Intialise random policy
    policy = [[random.choice(list(actions.keys())) for i in range(height)] for j in range(width)]
    for i in range(height):
        for j in range(width):
            if grid[i][j] == 'w':
                policy[i][j] = ''
    old_policy = copy.deepcopy(policy)

    # Keep track of utilities for plotting
    states_to_plot = [(j, i) for i in range(height) for j in range(width) if grid[i][j] != 'w']
    tracked_utilities = {state: [] for state in states_to_plot}


    stable = False
    iteration = 0

    while not stable:

        # Policy evaluation
        while True:
            delta = 0
            new_utilities = [[0 for j in range(height)] for k in range(width)]

            # For each state in the maze environment
            for j in range(height):
                for k in range(width):
                    # if wall, skip iteration
                    if grid[j][k] == 'w':
                        continue

                    # Get the utility of action from policy
                    action = policy[j][k]
                    total_action_utility = 0

                    for outcome in transition_probs:
                        move = actions[action][outcome]
                        new_x = k + move[1]
                        new_y = j + move[0]
                        if check_valid_state(new_x, new_y, grid):
                            # if next state is valid, get utility of next state
                            util = utilities[new_y][new_x]
                        else:
                            # if next state is invalid, get utility of current state
                            new_x = k
                            new_y = j
                            util = utilities[new_y][new_x]

                        total_action_utility += transition_probs[outcome] * util

                    total_utility = rewards[j][k] + discount * total_action_utility
                    new_utilities[j][k] = total_utility
                    delta = max(delta, abs(new_utilities[j][k] - utilities[j][k]))

            # Update utilities and check for convergence
            utilities = new_utilities
            if delta < convergence_threshold:
                break

```

Figure 7.1: Policy Iteration Code



```python

        # Policy improvement
        stable = True
        for j in range(height):
            for k in range(width):
                if grid[j][k] == 'w':
                    continue

                old_action = policy[j][k]
                max_utility = -1000
                best_action = ''

                # get utility of new best action
                for action in actions:
                    total_action_utility = 0

                    for outcome in transition_probs:
                        move = actions[action][outcome]
                        new_x = k + move[1]
                        new_y = j + move[0]
                        if check_valid_state(new_x, new_y, grid):
                            # if next state is valid, get utility of next state
                            util = utilities[new_y][new_x]
                        else:
                            # if next state is invalid, get utility of current state
                            new_x = k
                            new_y = j
                            util = utilities[new_y][new_x]

                        total_action_utility += transition_probs[outcome] * util

                    if total_action_utility > max_utility:
                        max_utility = total_action_utility
                        best_action = action

                policy[j][k] = best_action
                if best_action != old_action:
                    stable = False


        for state in states_to_plot:
            tracked_utilities[state].append(utilities[state[1]][state[0]])

        iteration += 1
        #print(iteration)

    return policy, tracked_utilities, old_policy
```

Figure 7.2: Policy Iteration Code

3.2.2. Plot of Optimal Policy

The initial random policy is as shown in Figure 8.

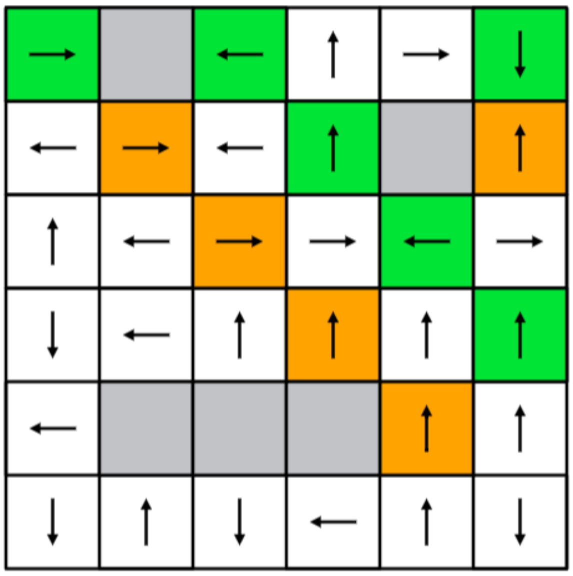


Figure 8: Initial Random Policy

After convergence, we can see that the policy has changed, and the optimal policy is as shown in Figure 9. The optimal policy obtained is the same as the policy obtained through value iteration, which shows that both algorithms are working and able to produce an optimal policy. Refer to Appendix B-7 for the code outputs.

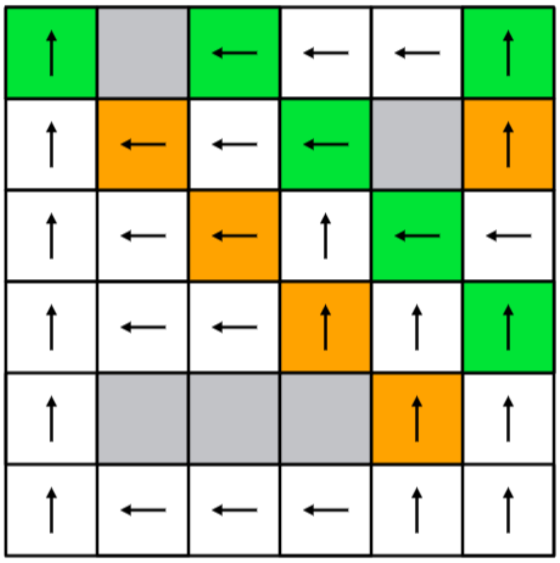
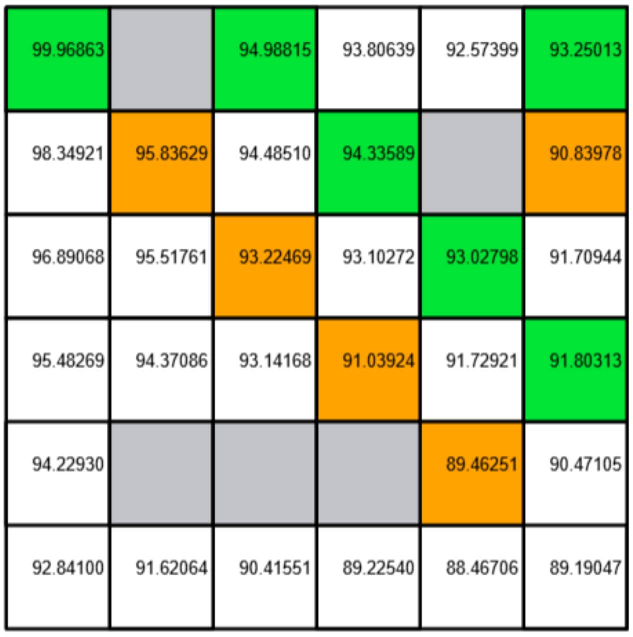


Figure 9: Optimal Policy after Policy Iteration

3.2.3. Utilities of All States

After stabilising, the final utilities for each state are observed in Figure 10. Values have been truncated to 5 decimal places. Similar to the results in value iteration, the top left grid has the highest utility value by using the same rationale. Refer to Appendix B-8 for the code outputs.



99.96863		94.98815	93.80639	92.57399	93.25013
98.34921	95.83629	94.48510	94.33589		90.83978
96.89068	95.51761	93.22469	93.10272	93.02798	91.70944
95.48269	94.37086	93.14168	91.03924	91.72921	91.80313
94.22930				89.46251	90.47105
92.84100	91.62064	90.41551	89.22540	88.46706	89.19047

Figure 10: Plot of Utilities of Each State

3.2.4. Plot of utility estimates as a function of the number of iterations

From Figure 11, we can see that the optimal policy stabilises after 4 iterations where there is no longer any change in optimal actions. Similar to value iteration, the maximum utility estimate is capped at 100. This is not a direct comparison against value iteration, as policy iteration consists of both policy evaluation and improvement, where policy evaluation has its own number of iterations for the utility estimates to converge before moving on to policy improvement. We can observe from the plot that the change in utility estimates between iterations are sudden and not as smooth compared to value iteration, due to the fact that the policy changes after policy improvement and the better actions would lead to better utilities.

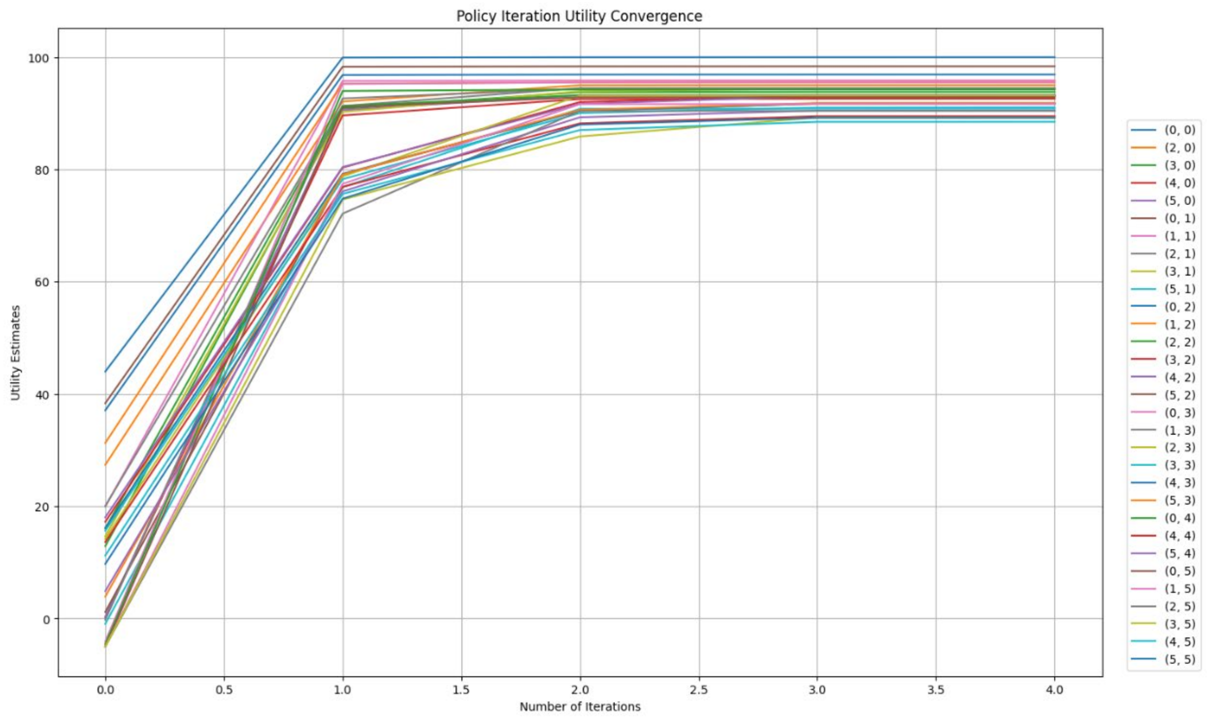


Figure 11: Plot of Utility Estimates against Iterations

4. Part 2

For the second part of the assignment, a more complicated maze environment was created by increasing the size of the environment and randomly initialising the elements for each state. A 10x10 maze was created, with random elements as seen in Figure 12. Refer to Appendix C-1.

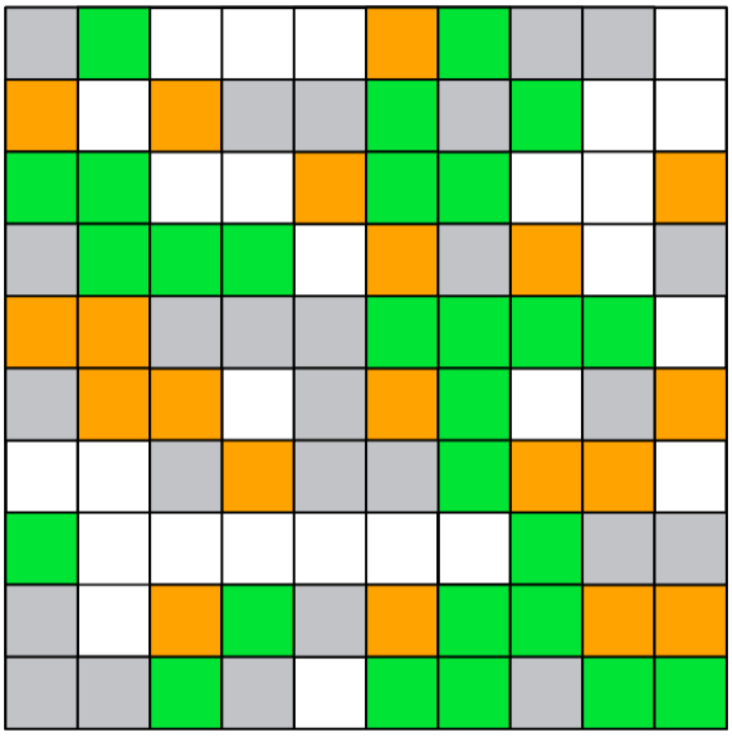


Figure 12: Diagram of Complicated Maze Environment

After implementing both value iteration and policy iteration on the new maze, both converged to the same optimal policy shown in Figure 13. Refer to Appendices C-3, C-4, C-7 and C-8.

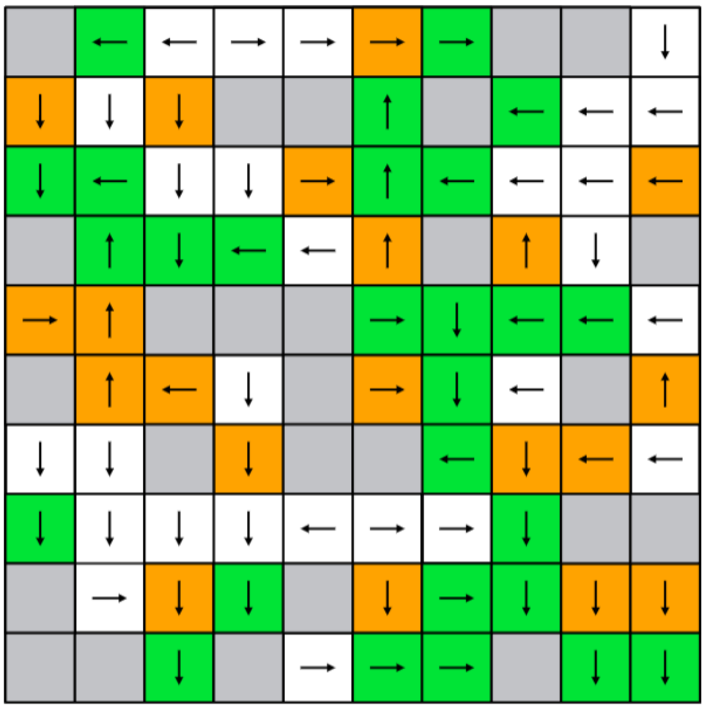


Figure 13: Optimal Policy

The plots of utility estimates as a function of iterations for both value iteration and policy iteration can be observed in Figures 14 and 15 respectively. The value iteration convergence seems to be similar as the previous one with a simpler maze environment. This could be due to the use of a large discount factor of 0.99 which causes the future rewards to be impactful, and a slight increase in maze size might not affect the convergence significantly. However, the convergence for policy iteration increased to 5 iterations. This shows that the complexity of the environment still affects the convergence of utilities, in this case the larger size of the maze causes the convergence of optimal policies to converge slightly slower.

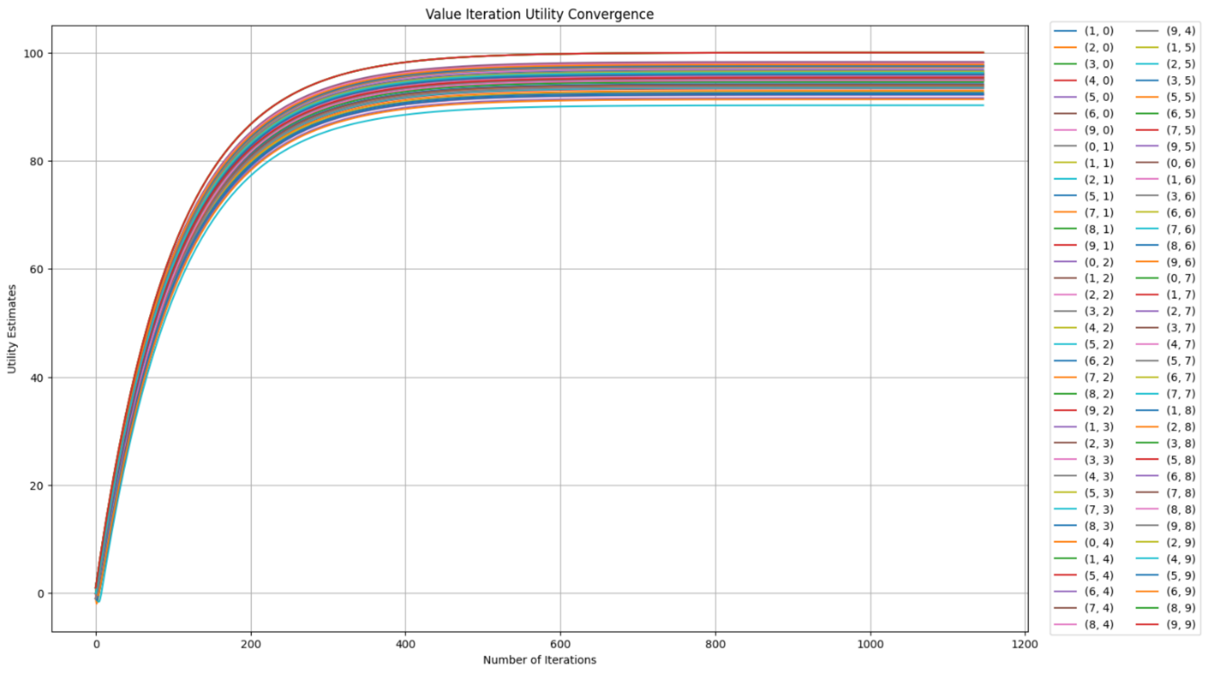


Figure 14: Plot of Utility Estimates against Iterations in Value Iteration

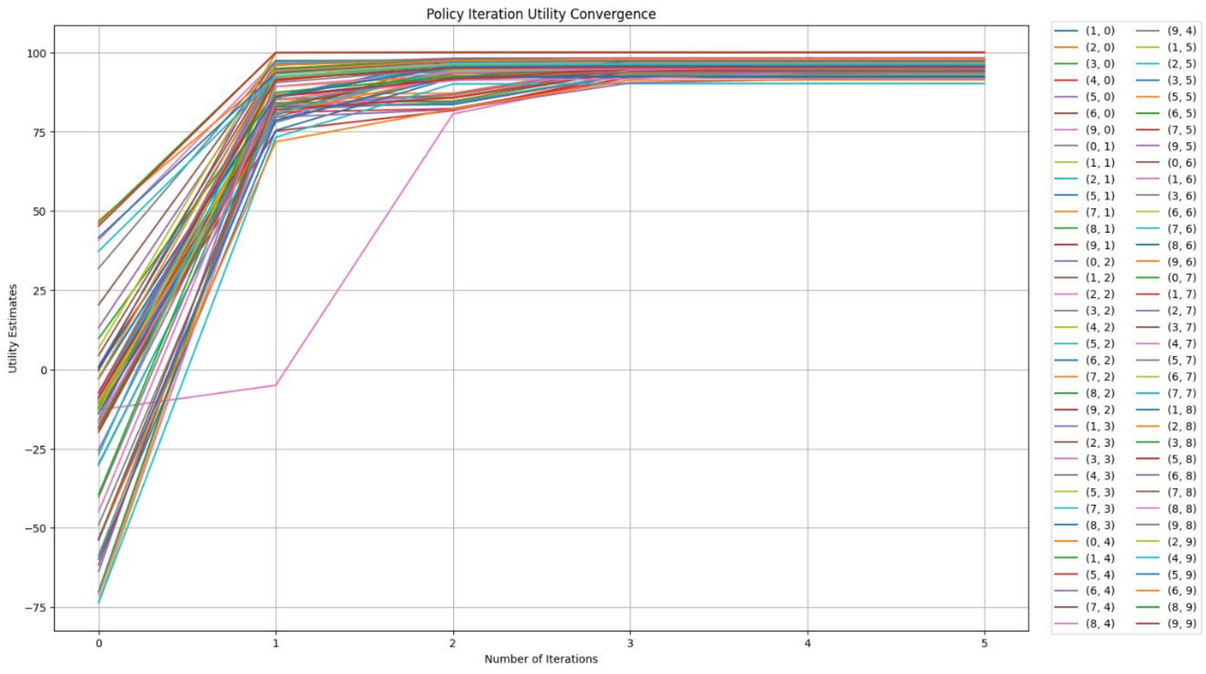


Figure 15: Plot of Utility Estimates against Iterations in Policy Iteration

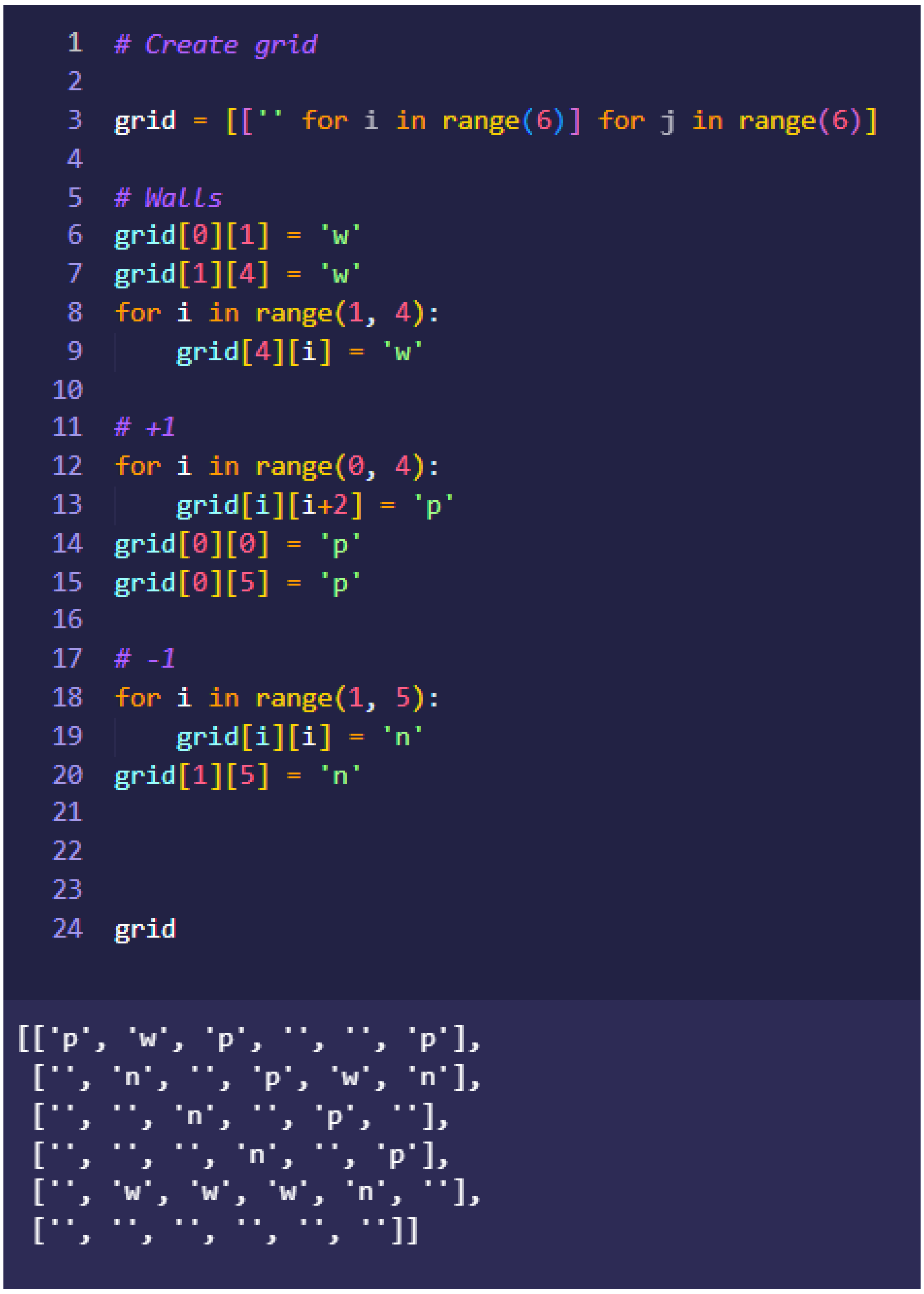
5. Conclusion

This report explored reinforcement learning algorithms, value iteration and policy iteration, in a GridWorld maze environment. Both algorithms successfully converged to provide the same optimal policy. By increasing the complexity of the maze environment, it could still convergence, with a slight reduction in the convergence speed. However, there are different ways to increase the complexity of the maze, and increasing the size is only one way. Increasing the complexity in other various ways could also be explored in the future to test the efficiency of value iteration and policy iteration as well.

Appendix

A. MDP

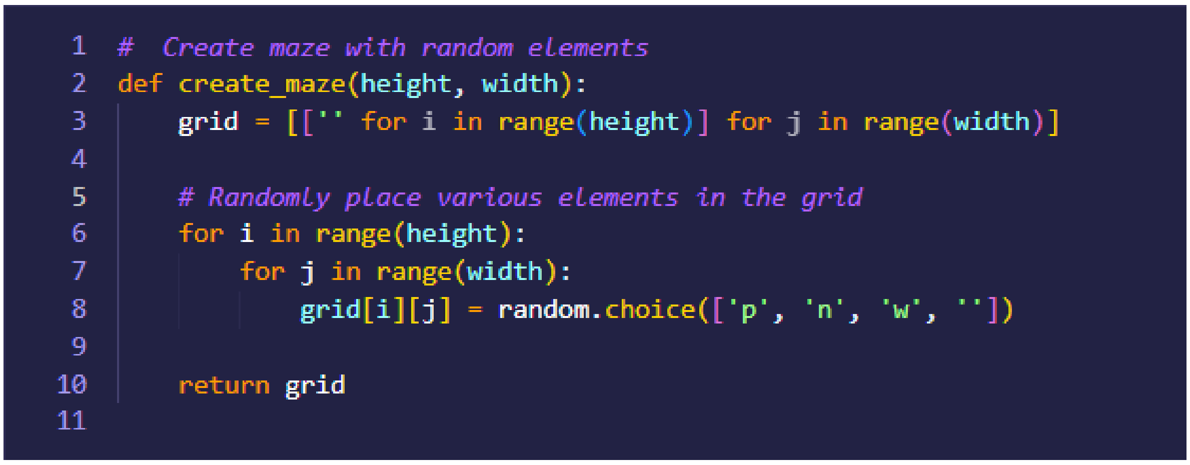
Appendix A-1: Creating the environment as per the assignment



```
# Create grid

grid = [['' for i in range(6)] for j in range(6)]

# Walls
grid[0][1] = 'w'
grid[1][4] = 'w'
for i in range(1, 4):
    grid[4][i] = 'w'

# +1
for i in range(0, 4):
    grid[i][i+2] = 'p'
grid[0][0] = 'p'
grid[0][5] = 'p'

# -1
for i in range(1, 5):
    grid[i][i] = 'n'
grid[1][5] = 'n'



grid
```

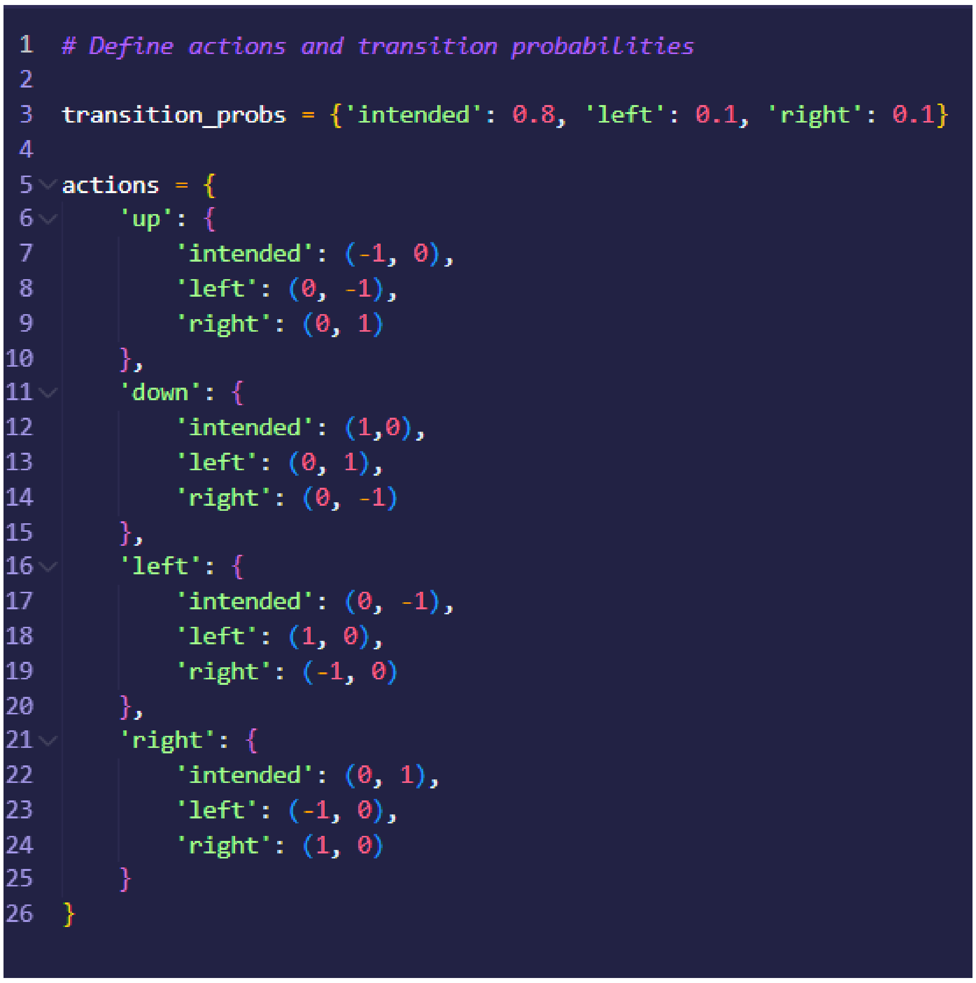
```
[['p', 'w', 'p', '', '', 'p'],
 ['', 'n', '', 'p', 'w', 'n'],
 ['', '', 'n', '', 'p', ''],
 ['', '', '', 'n', '', 'p'],
 ['', 'w', 'w', 'w', 'n', ''],
 ['', '', '', '', '', '']]
```

Appendix A-2: Function to create an environment with random elements for each state



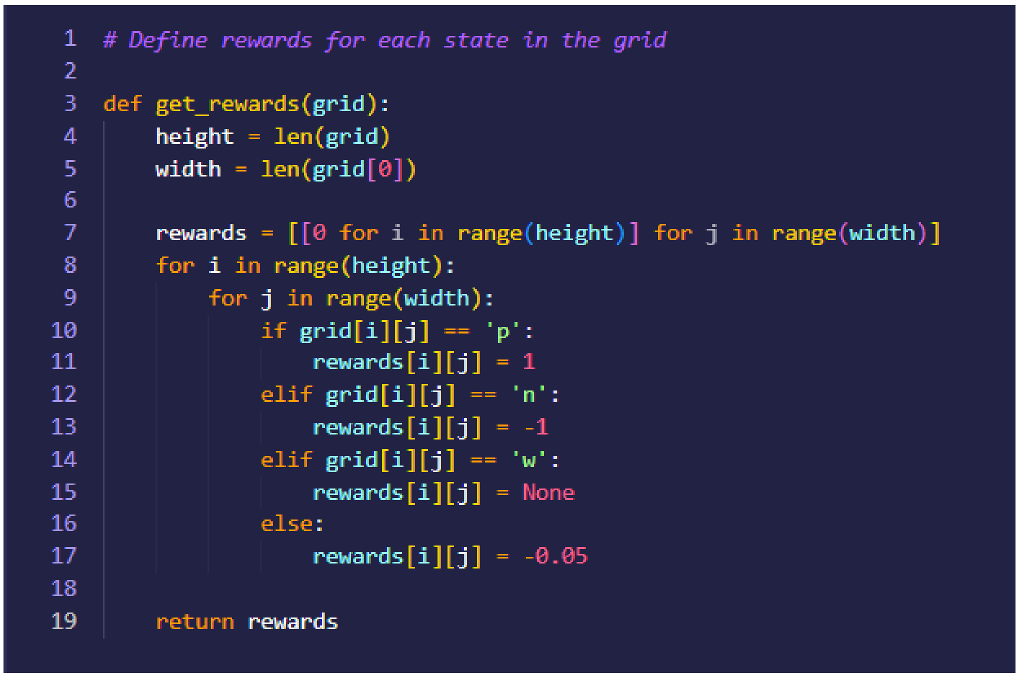
```python
#  Create maze with random elements
def create_maze(height, width):
    grid = [['' for i in range(height)] for j in range(width)]

    # Randomly place various elements in the grid
    for i in range(height):
        for j in range(width):
            grid[i][j] = random.choice(['p', 'n', 'w', ''])

    return grid

```

Appendix A-3: Actions available and outcome probabilities



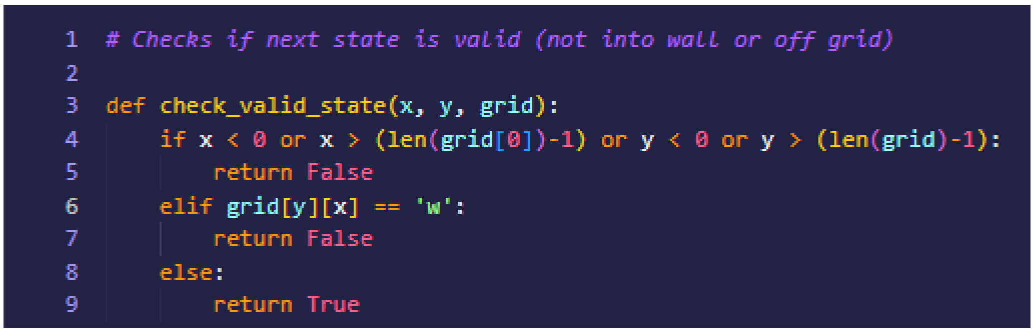
```python
# Define actions and transition probabilities

transition_probs = {'intended': 0.8, 'left': 0.1, 'right': 0.1}

actions = {
    'up': {
        'intended': (-1, 0),
        'left': (0, -1),
        'right': (0, 1)
    },
    'down': {
        'intended': (1,0),
        'left': (0, 1),
        'right': (0, -1)
    },
    'left': {
        'intended': (0, -1),
        'left': (1, 0),
        'right': (-1, 0)
    },
    'right': {
        'intended': (0, 1),
        'left': (-1, 0),
        'right': (1, 0)
    }
}
```

Appendix A-4: Function to get rewards for each state based on given grid



```python
# Define rewards for each state in the grid

def get_rewards(grid):
    height = len(grid)
    width = len(grid[0])

    rewards = [[0 for i in range(height)] for j in range(width)]
    for i in range(height):
        for j in range(width):
            if grid[i][j] == 'p':
                rewards[i][j] = 1
            elif grid[i][j] == 'n':
                rewards[i][j] = -1
            elif grid[i][j] == 'w':
                rewards[i][j] = None
            else:
                rewards[i][j] = -0.05

    return rewards
```

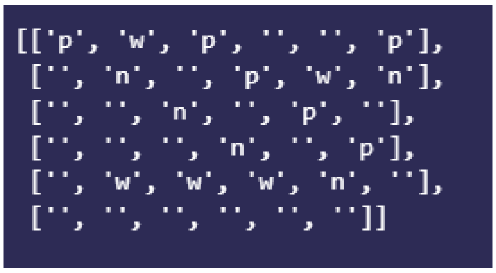
Appendix A-5: Function to check if action leads to valid state, used in value and policy iteration



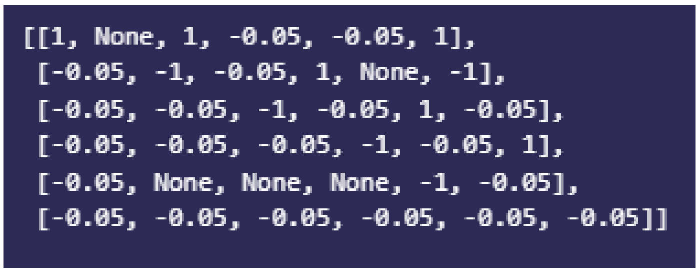
```python
# Checks if next state is valid (not into wall or off grid)

def check_valid_state(x, y, grid):
    if x < 0 or x > (len(grid[0])-1) or y < 0 or y > (len(grid)-1):
        return False
    elif grid[y][x] == 'w':
        return False
    else:
        return True
```

B. Part 1

Appendix B-1: Code outputs for states and rewards in part 1

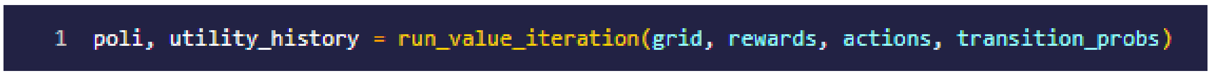


```
[['p', 'w', 'p', '', '', 'p'],
 ['', 'n', '', 'p', 'w', 'n'],
 ['', '', 'n', '', 'p', ''],
 ['', '', '', 'n', '', 'p'],
 ['', 'w', 'w', 'w', 'n', ''],
 ['', '', '', '', '', '']]
```



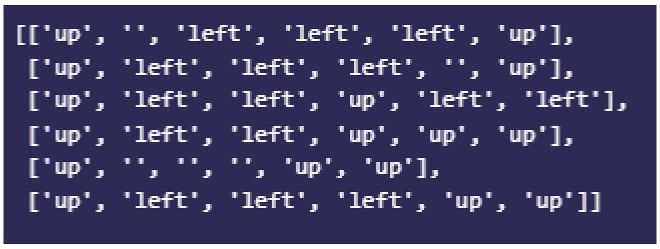
```
[[1, None, 1, -0.05, -0.05, 1],
 [-0.05, -1, -0.05, 1, None, -1],
 [-0.05, -0.05, -1, -0.05, 1, -0.05],
 [-0.05, -0.05, -0.05, -1, -0.05, 1],
 [-0.05, None, None, None, -1, -0.05],
 [-0.05, -0.05, -0.05, -0.05, -0.05, -0.05]]
```

Appendix B-2: Application of value iteration in part 1



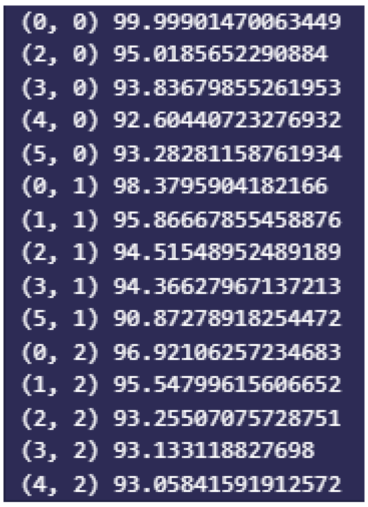
```
poli, utility_history = run_value_iteration(grid, rewards, actions, transition_probs)
```

Appendix B-3: Code output for optimal policy after value iteration in part 1

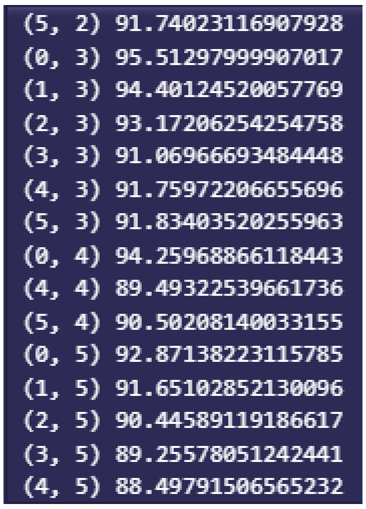


```
[['up', '', 'left', 'left', 'left', 'up'],
 ['up', 'left', 'left', 'left', '', 'up'],
 ['up', 'left', 'left', 'up', 'left', 'left'],
 ['up', 'left', 'left', 'up', 'up', 'up'],
 ['up', '', '', '', 'up', 'up'],
 ['up', 'left', 'left', 'left', 'up', 'up']]
```

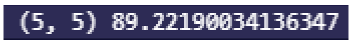
Appendix B-4: Code output for utility estimates of each state (x, y) after value iteration in part 1



```
(0, 0) 99.99901470063449
(2, 0) 95.0185652290884
(3, 0) 93.83679855261953
(4, 0) 92.60440723276932
(5, 0) 93.28281158761934
(0, 1) 98.3795904182166
(1, 1) 95.86667855458876
(2, 1) 94.51548952489189
(3, 1) 94.36627967137213
(5, 1) 90.87278918254472
(0, 2) 96.92106257234683
(1, 2) 95.54799615606652
(2, 2) 93.25507075728751
(3, 2) 93.133118827698
(4, 2) 93.05841591912572
```

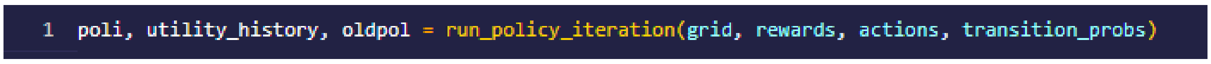


```
(5, 2) 91.74023116907928
(0, 3) 95.51297999907017
(1, 3) 94.40124520057769
(2, 3) 93.17206254254758
(3, 3) 91.06966693484448
(4, 3) 91.75972206655696
(5, 3) 91.83403520255963
(0, 4) 94.25968866118443
(4, 4) 89.49322539661736
(5, 4) 90.50208140033155
(0, 5) 92.87138223115785
(1, 5) 91.65102852130096
(2, 5) 90.44589119186617
(3, 5) 89.25578051242441
(4, 5) 88.49791506565232
```

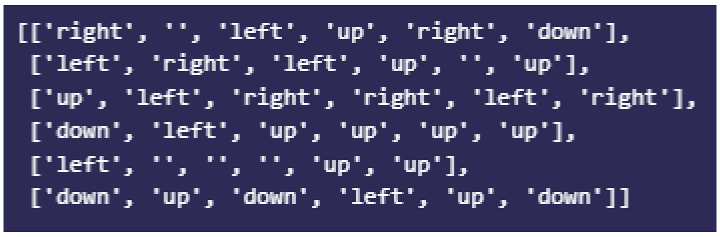


```
(5, 5) 89.22190034136347
```

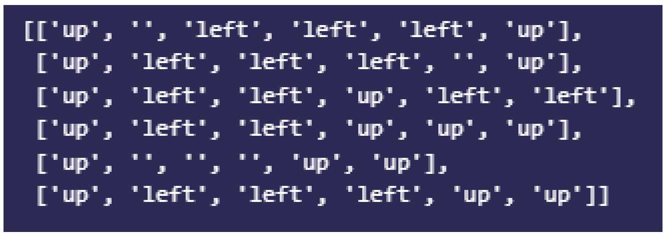
Appendix B-5: Application of policy iteration in part 1


```
poli, utility_history, oldpol = run_policy_iteration(grid, rewards, actions, transition_probs)
```

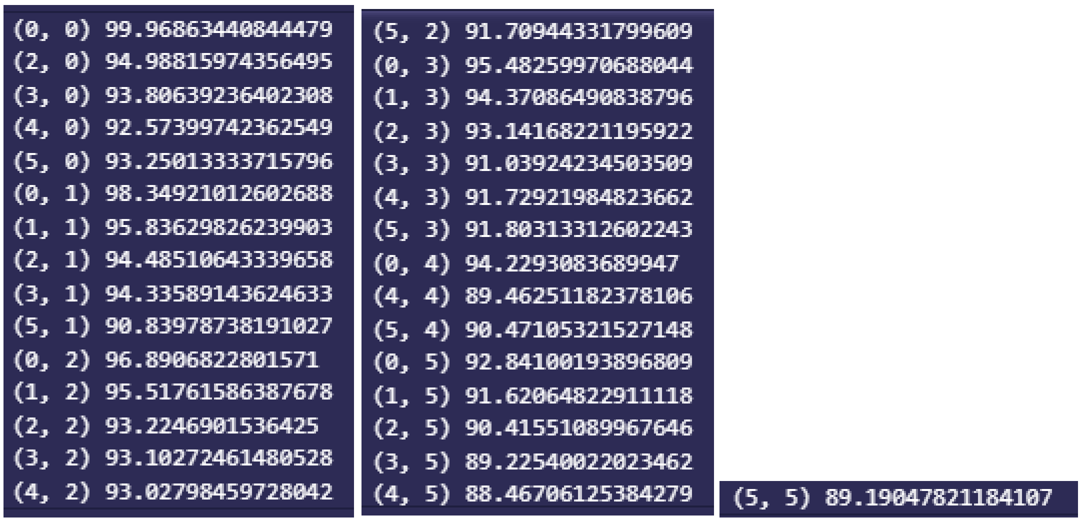
Appendix B-6: Code output for initial random policy in part 1


```
[['right', '', 'left', 'up', 'right', 'down'],
 ['left', 'right', 'left', 'up', '', 'up'],
 ['up', 'left', 'right', 'right', 'left', 'right'],
 ['down', 'left', 'up', 'up', 'up', 'up'],
 ['left', '', '', '', 'up', 'up'],
 ['down', 'up', 'down', 'left', 'up', 'down']]
```

Appendix B-7: Code output for optimal policy after policy iteration in part 1


```
[['up', '', 'left', 'left', 'left', 'up'],
 ['up', 'left', 'left', 'left', '', 'up'],
 ['up', 'left', 'left', 'up', 'left', 'left'],
 ['up', 'left', 'left', 'up', 'up', 'up'],
 ['up', '', '', '', 'up', 'up'],
 ['up', 'left', 'left', 'left', 'up', 'up']]
```

Appendix B-8: Code output for utility estimates of each state (x, y) after policy iteration in part 1

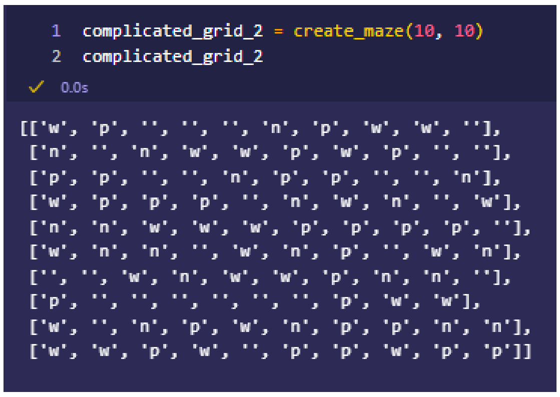

```
(0, 0) 99.96863440844479
(2, 0) 94.98815974356495
(3, 0) 93.80639236402308
(4, 0) 92.57399742362549
(5, 0) 93.25013333715796
(0, 1) 98.34921012602688
(1, 1) 95.83629826239903
(2, 1) 94.48510643339658
(3, 1) 94.33589143624633
(5, 1) 90.83978738191027
(0, 2) 96.8906822801571
(1, 2) 95.51761586387678
(2, 2) 93.2246901536425
(3, 2) 93.10272461480528
(4, 2) 93.02798459728042
```

```
(5, 2) 91.70944331799609
(0, 3) 95.48259970688044
(1, 3) 94.37086490838796
(2, 3) 93.14168221195922
(3, 3) 91.03924234503509
(4, 3) 91.72921984823662
(5, 3) 91.80313312602243
(0, 4) 94.2293083689947
(4, 4) 89.46251182378106
(5, 4) 90.47105321527148
(0, 5) 92.84100193896809
(1, 5) 91.62064822911118
(2, 5) 90.41551089967646
(3, 5) 89.22540022023462
(4, 5) 88.46706125384279
```

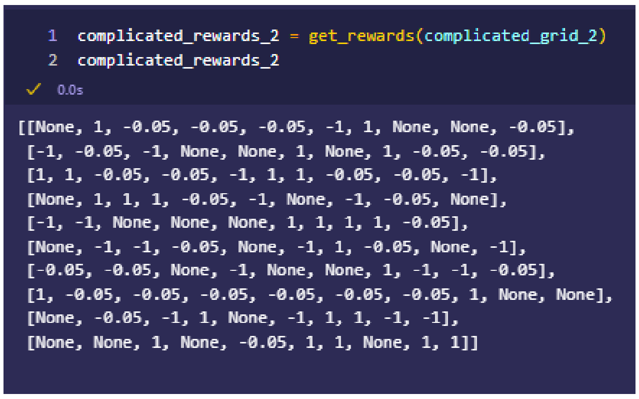
```
(5, 5) 89.19047821184107
```

C. Part 2

Appendix C-1: Code output for environment states and rewards in part 2

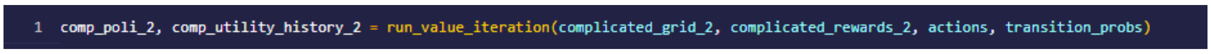


```
complicated_grid_2 = create_maze(10, 10)
complicated_grid_2
✓ 0.0s

[['w', 'p', '', '', '', 'n', 'p', 'w', 'w', ''],
 ['n', '', 'n', 'w', 'w', 'p', 'w', 'p', '', ''],
 ['p', 'p', '', '', 'n', 'p', 'p', '', '', 'n'],
 ['w', 'p', 'p', 'p', '', 'n', 'w', 'n', '', 'w'],
 ['n', 'n', 'w', 'w', 'w', 'p', 'p', 'p', 'p', ''],
 ['w', 'n', 'n', '', 'w', 'n', 'p', '', 'w', 'n'],
 ['', '', 'w', 'n', 'w', 'w', 'p', 'n', 'n', ''],
 ['p', '', '', '', '', '', '', 'p', 'w', 'w'],
 ['w', '', 'n', 'p', 'w', 'n', 'p', 'p', 'n', 'n'],
 ['w', 'w', 'p', 'w', '', 'p', 'p', 'w', 'p', 'p']]
```



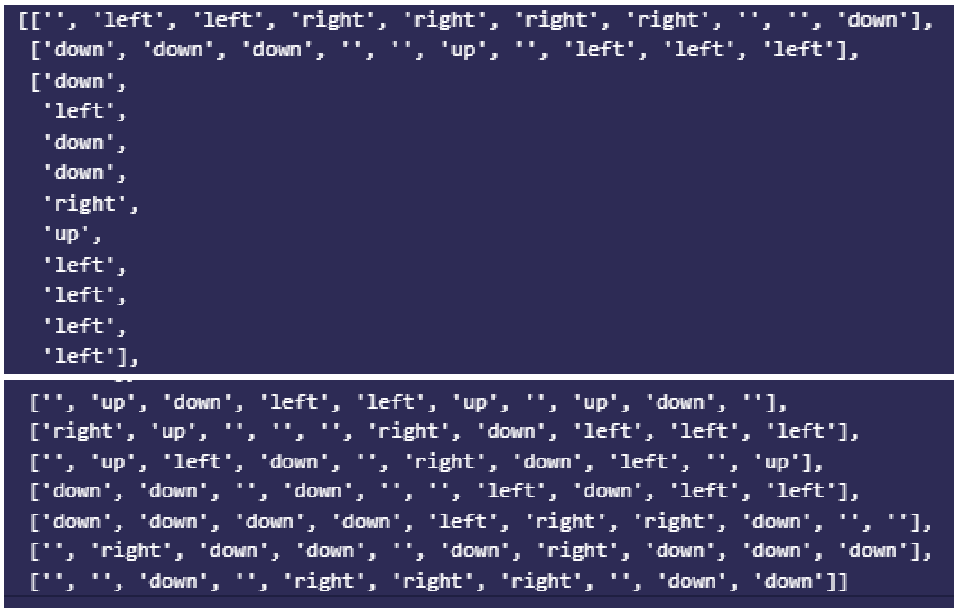
```
complicated_rewards_2 = get_rewards(complicated_grid_2)
complicated_rewards_2
✓ 0.0s

[[None, 1, -0.05, -0.05, -0.05, -1, 1, None, None, -0.05],
 [-1, -0.05, -1, None, None, 1, None, 1, -0.05, -0.05],
 [1, 1, -0.05, -0.05, -1, 1, 1, -0.05, -0.05, -1],
 [None, 1, 1, 1, -0.05, -1, None, -1, -0.05, None],
 [-1, -1, None, None, None, 1, 1, 1, 1, -0.05],
 [None, -1, -1, -0.05, None, -1, 1, -0.05, None, -1],
 [-0.05, -0.05, None, -1, None, None, 1, -1, -1, -0.05],
 [1, -0.05, -0.05, -0.05, -0.05, -0.05, -0.05, 1, None, None],
 [None, -0.05, -1, 1, None, -1, 1, 1, -1, -1],
 [None, None, 1, None, -0.05, 1, 1, None, 1, 1]]
```

Appendix C-2: Application of value iteration in part 2



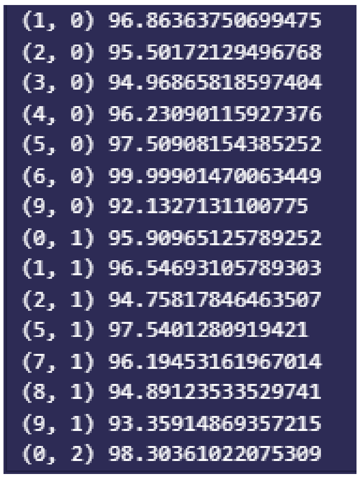
```
comp_poli_2, comp_utility_history_2 = run_value_iteration(complicated_grid_2, complicated_rewards_2, actions, transition_probs)
```

Appendix C-3: Code output for optimal policy after value iteration in part 2

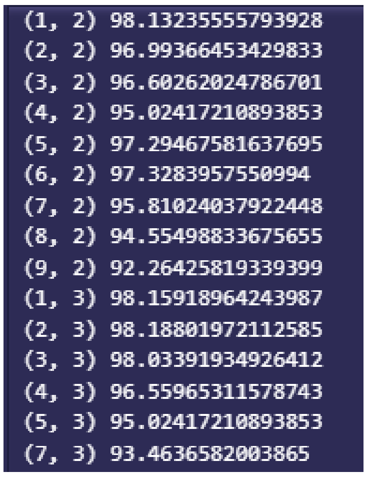


```
[['', 'left', 'left', 'right', 'right', 'right', 'right', '', '', 'down'],
 ['down', 'down', 'down', '', '', 'up', '', 'left', 'left', 'left'],
 ['down',
  'left',
  'down',
  'down',
  'right',
  'up',
  'left',
  'left',
  'left',
  'left'],
 ['', 'up', 'down', 'left', 'left', 'up', '', 'up', 'down', ''],
 ['right', 'up', '', '', '', 'right', 'down', 'left', 'left', 'left'],
 ['', 'up', 'left', 'down', '', 'right', 'down', 'left', '', 'up'],
 ['down', 'down', '', 'down', '', '', 'left', 'down', 'left', 'left'],
 ['down', 'down', 'down', 'down', 'left', 'right', 'right', 'down', '', ''],
 ['', 'right', 'down', 'down', '', 'down', 'right', 'down', 'down', 'down'],
 ['', '', 'down', '', 'right', 'right', 'right', '', 'down', 'down']]
```

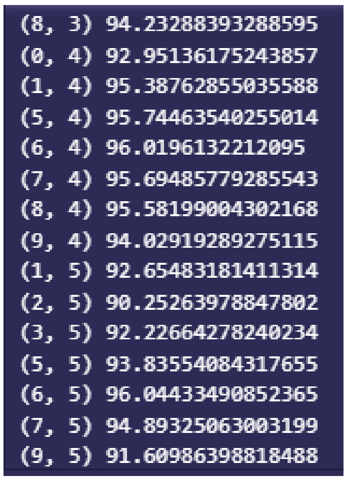
Appendix C-4: Code output for utility estimates of each state (x, y) after value iteration in part 2



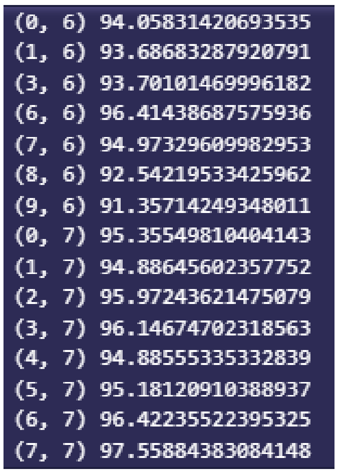
```
(1, 0) 96.86363750699475
(2, 0) 95.50172129496768
(3, 0) 94.96865818597404
(4, 0) 96.23090115927376
(5, 0) 97.50908154385252
(6, 0) 99.99901470063449
(9, 0) 92.1327131100775
(0, 1) 95.90965125789252
(1, 1) 96.54693105789303
(2, 1) 94.75817846463507
(5, 1) 97.5401280919421
(7, 1) 96.19453161967014
(8, 1) 94.89123533529741
(9, 1) 93.35914869357215
(0, 2) 98.30361022075309
```



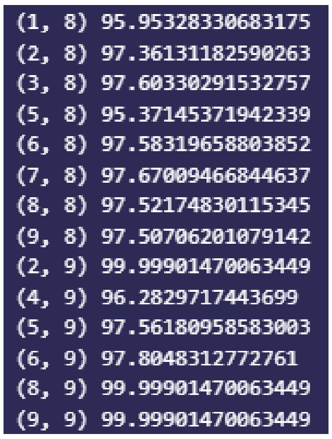
```
(1, 2) 98.13235555793928
(2, 2) 96.99366453429833
(3, 2) 96.60262024786701
(4, 2) 95.02417210893853
(5, 2) 97.29467581637695
(6, 2) 97.3283957550994
(7, 2) 95.81024037922448
(8, 2) 94.55498833675655
(9, 2) 92.26425819339399
(1, 3) 98.15918964243987
(2, 3) 98.18801972112585
(3, 3) 98.03391934926412
(4, 3) 96.55965311578743
(5, 3) 95.02417210893853
(7, 3) 93.4636582003865
```



```
(8, 3) 94.23288393288595
(0, 4) 92.95136175243857
(1, 4) 95.38762855035588
(5, 4) 95.74463540255014
(6, 4) 96.0196132212095
(7, 4) 95.69485779285543
(8, 4) 95.58199004302168
(9, 4) 94.02919289275115
(1, 5) 92.65483181411314
(2, 5) 90.25263978847802
(3, 5) 92.22664278240234
(5, 5) 93.83554084317655
(6, 5) 96.04433490852365
(7, 5) 94.89325063003199
(9, 5) 91.60986398818488
```

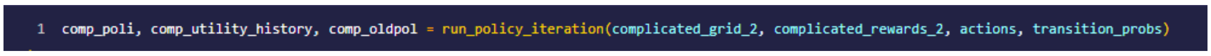


```
(0, 6) 94.05831420693535
(1, 6) 93.68683287920791
(3, 6) 93.70101469996182
(6, 6) 96.41438687575936
(7, 6) 94.97329609982953
(8, 6) 92.54219533425962
(9, 6) 91.35714249348011
(0, 7) 95.35549810404143
(1, 7) 94.88645602357752
(2, 7) 95.97243621475079
(3, 7) 96.14674702318563
(4, 7) 94.88555335332839
(5, 7) 95.18120910388937
(6, 7) 96.42235522395325
(7, 7) 97.55884383084148
```



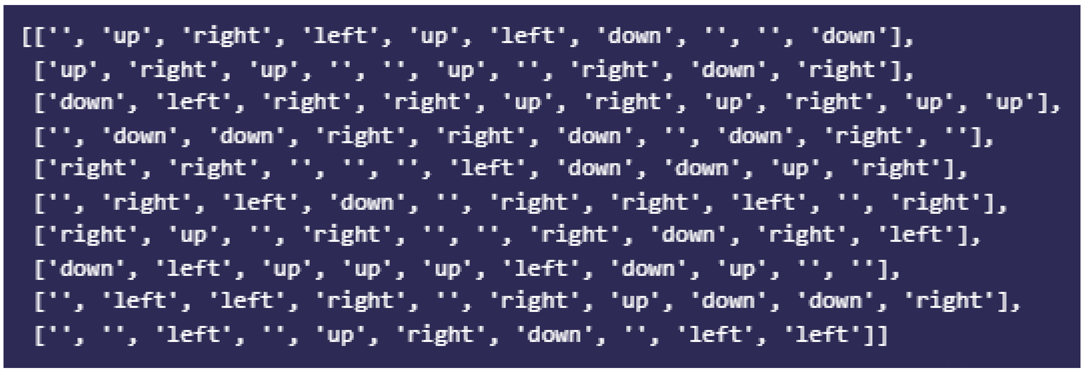
```
(1, 8) 95.95328330683175
(2, 8) 97.36131182590263
(3, 8) 97.60330291532757
(5, 8) 95.37145371942339
(6, 8) 97.58319658803852
(7, 8) 97.67009466844637
(8, 8) 97.52174830115345
(9, 8) 97.50706201079142
(2, 9) 99.99901470063449
(4, 9) 96.2829717443699
(5, 9) 97.5618095858303
(6, 9) 97.8048312772761
(8, 9) 99.99901470063449
(9, 9) 99.99901470063449
```

Appendix C-5: Application of policy iteration in part 2



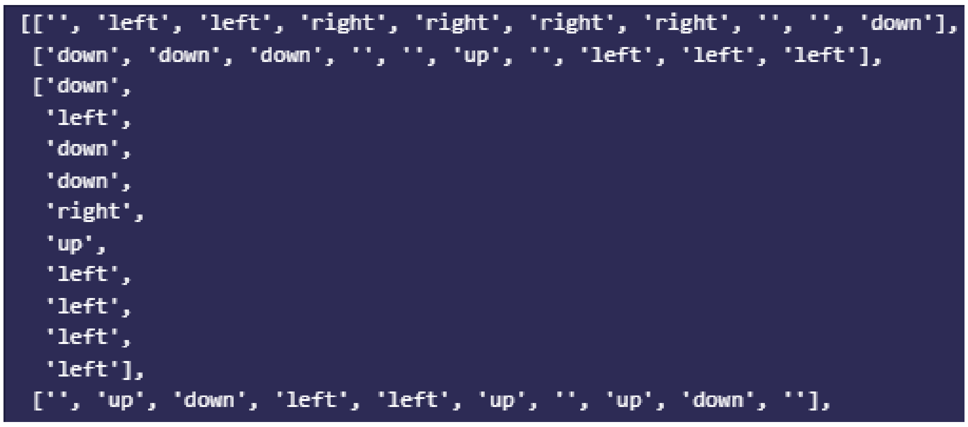
```
comp_poli, comp_utility_history, comp_oldpol = run_policy_iteration(complicated_grid_2, complicated_rewards_2, actions, transition_probs)
```

Appendix C-6: Code output for initial random policy in part 2

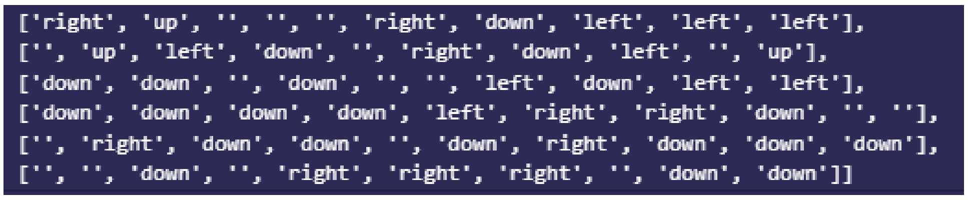


```
[['', 'up', 'right', 'left', 'up', 'left', 'down', '', '', 'down'],
 ['up', 'right', 'up', '', '', 'up', '', 'right', 'down', 'right'],
 ['down', 'left', 'right', 'right', 'up', 'right', 'up', 'right', 'up', 'up'],
 ['', 'down', 'down', 'right', 'right', 'down', '', 'down', 'right', ''],
 ['right', 'right', '', '', '', 'left', 'down', 'down', 'up', 'right'],
 ['', 'right', 'left', 'down', '', 'right', 'right', 'left', '', 'right'],
 ['right', 'up', '', 'right', '', '', 'right', 'down', 'right', 'left'],
 ['down', 'left', 'up', 'up', 'up', 'left', 'down', 'up', '', ''],
 ['', 'left', 'left', 'right', '', 'right', 'up', 'down', 'down', 'right'],
 ['', '', 'left', '', 'up', 'right', 'down', '', 'left', 'left']]
```

Appendix C-7: Code output for optimal policy after policy iteration in part 2

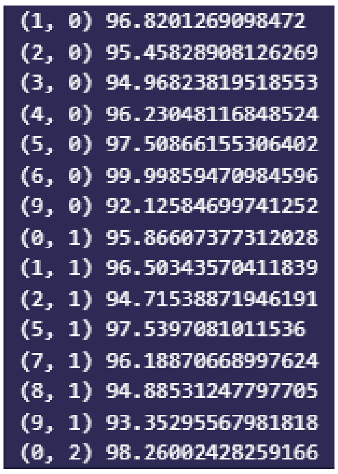


```
[['', 'left', 'left', 'right', 'right', 'right', 'right', '', '', 'down'],
 ['down', 'down', 'down', '', '', 'up', '', 'left', 'left', 'left'],
 ['down',
  'left',
  'down',
  'down',
  'right',
  'up',
  'left',
  'left',
  'left',
  'left'],
 ['', 'up', 'down', 'left', 'left', 'up', '', 'up', 'down', ''],
```

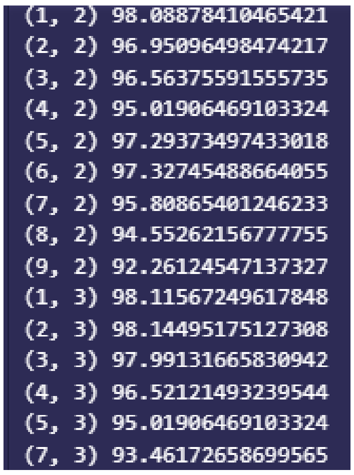


```
['right', 'up', '', '', '', 'right', 'down', 'left', 'left', 'left'],
['', 'up', 'left', 'down', '', 'right', 'down', 'left', '', 'up'],
['down', 'down', '', 'down', '', '', 'left', 'down', 'left', 'left'],
['down', 'down', 'down', 'down', 'left', 'right', 'right', 'down', '', ''],
['', 'right', 'down', 'down', '', 'down', 'right', 'down', 'down', 'down'],
['', '', 'down', '', 'right', 'right', 'right', '', 'down', 'down']]
```

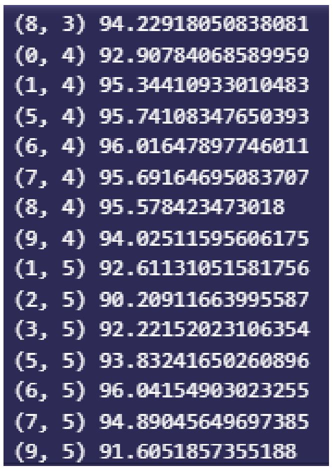
Appendix C-8: Code output for utility estimates of each state (x, y) after policy iteration in part 2



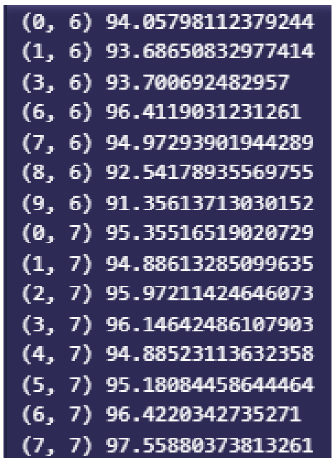
```
(1, 0) 96.8201269098472
(2, 0) 95.45828908126269
(3, 0) 94.96823819518553
(4, 0) 96.23048116848524
(5, 0) 97.50866155306402
(6, 0) 99.99859470984596
(9, 0) 92.12584699741252
(0, 1) 95.86607377312028
(1, 1) 96.50343570411839
(2, 1) 94.71538871946191
(5, 1) 97.5397081011536
(7, 1) 96.18870668997624
(8, 1) 94.88531247797705
(9, 1) 93.35295567981818
(0, 2) 98.26002428259166
```



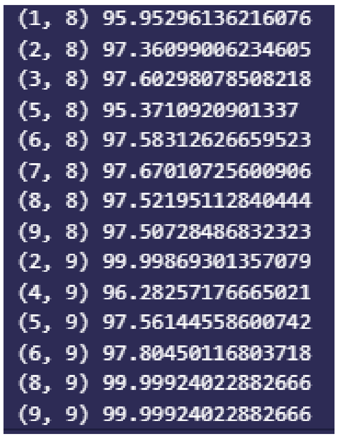
```
(1, 2) 98.08878410465421
(2, 2) 96.95096498474217
(3, 2) 96.56375591555735
(4, 2) 95.01906469103324
(5, 2) 97.29373497433018
(6, 2) 97.32745488664055
(7, 2) 95.80865401246233
(8, 2) 94.55262156777755
(9, 2) 92.26124547137327
(1, 3) 98.11567249617848
(2, 3) 98.14495175127308
(3, 3) 97.99131665830942
(4, 3) 96.52121493239544
(5, 3) 95.01906469103324
(7, 3) 93.46172658699565
```



```
(8, 3) 94.22918050838081
(0, 4) 92.90784068589959
(1, 4) 95.34410933010483
(5, 4) 95.74108347650393
(6, 4) 96.01647897746011
(7, 4) 95.69164695083707
(8, 4) 95.578423473018
(9, 4) 94.02511595606175
(1, 5) 92.61131051581756
(2, 5) 90.20911663995587
(3, 5) 92.22152023106354
(5, 5) 93.83241650260896
(6, 5) 96.04154903023255
(7, 5) 94.89045649697385
(9, 5) 91.6051857355188
```



```
(0, 6) 94.05798112379244
(1, 6) 93.68650832977414
(3, 6) 93.700692482957
(6, 6) 96.4119031231261
(7, 6) 94.97293901944289
(8, 6) 92.54178935569755
(9, 6) 91.35613713030152
(0, 7) 95.35516519020729
(1, 7) 94.88613285099635
(2, 7) 95.97211424646073
(3, 7) 96.14642486107903
(4, 7) 94.88523113632358
(5, 7) 95.18084458644464
(6, 7) 96.4220342735271
(7, 7) 97.55880373813261
```



```
(1, 8) 95.95296136216076
(2, 8) 97.36099006234605
(3, 8) 97.60298078508218
(5, 8) 95.3710920901337
(6, 8) 97.58312626659523
(7, 8) 97.67010725600906
(8, 8) 97.52195112840444
(9, 8) 97.50728486832323
(2, 9) 99.99869301357079
(4, 9) 96.28257176665021
(5, 9) 97.56144558600742
(6, 9) 97.80450116803718
(8, 9) 99.99924022882666
(9, 9) 99.99924022882666
```