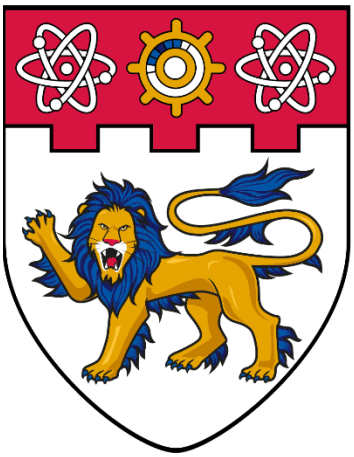


NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

School of Computer Science and Engineering

CE4046 Intelligent Agents

Assignment 1

Name: Lumlertluksanachai Pongpakin

Matriculation Number: U2023344C

Part 1 Value Iteration

I. Descriptions of implemented solutions

To commence, the maze is initialized utilizing a for loop to establish its dimensions as 6 x 6. Subsequently, the coordinates of advantageous and disadvantageous states are declared, and their corresponding rewards of +1 and penalties of -1, respectively, are assigned to each agent move (-0.04). Furthermore, constants such as the discount factor (GAMMA = 0.99) utilized in Bellman's equation, and the terminating factor for non-terminating states (EPSILON = 10^{-6}) are declared. To augment the algorithm's versatility, NUM_OF_ACTIONS is set to 4, with the list ACTIONS signifying the four directions (UP, LEFT, DOWN, RIGHT).

```
BOARD_ROWS = 6
BOARD_COLS = 6
GOOD_STATES = [(0, 0), (0, 2), (0, 5), (1, 3), (2, 4), (3, 4)]
BAD_STATES = [(1, 1), (1, 5), (2, 2), (3, 3), (4, 4)]
START_STATE = (3, 2)
WALLS = [(0, 1), (1, 4), (4, 1), (4, 2), (4, 3)]
GAMMA = 0.99
EPSILON = 1e-6
REWARD = -0.04
NUM_OF_ACTIONS = 4
ACTIONS = [(-1, 0), (0, -1), (1, 0), (0, 1)]
```

Following this, a function is created that returns the new state of the agent if it successfully moved and returns the original state if the agent is blocked by a wall.

```
def get_state(states, row, col, action):
    # Get the state of taking the given action
    temp_row = row + ACTIONS[action][0]
    temp_col = col + ACTIONS[action][1]
    if temp_row < 0 or temp_row >= BOARD_ROWS or temp_col < 0 or temp_col >= BOARD_COLS or (temp_row, temp_col) in WALLS:
        return states[row][col]
    else:
        return states[temp_row][temp_col]
```

Next, a function is developed to calculate the utility of each state for every iteration based on the utility value of previous iterations. It returns the value of that state in the iteration when it has a possibility of 0.1 to move left and right of the desired direction.

```
def utility_score(states, row, col, action):
    # Evaluate the expected utility score of taking the given action in the given state
    utility = REWARD
    if (row, col) in GOOD_STATES:
        utility = 1
    elif (row, col) in BAD_STATES:
        utility = -1
    utility += 0.1 * (GAMMA * get_state(states, row, col, (action - 1) % 4))
    utility += 0.1 * (GAMMA * get_state(states, row, col, (action + 1) % 4))
    utility += 0.8 * (GAMMA * get_state(states, row, col, action))
    return utility
```

Subsequently, a function is employed to obtain the policy by analyzing the values of each grid around the agent. For instance, if the value of moving up of the agent is greater than that of the other three directions, the agent will move up in this case.

```
def get_policy(states):
    # Get the optimal policy for each state
    policy = np.zeros((BOARD_ROWS, BOARD_COLS))
    for row in range(BOARD_ROWS):
        for col in range(BOARD_COLS):
            if (row, col) in GOOD_STATES or (row, col) in BAD_STATES or (row, col) in WALLS:
                continue
            best_action = None
            best_utility = float('-inf')
            for action in range(NUM_OF_ACTIONS):
                utility = utility_score(states, row, col, action)
                if utility > best_utility:
                    best_utility = utility
                    best_action = action
            policy[row][col] = best_action
    return policy
```

The next function is employed to conduct the value iteration to update the value function, which maps each state in the environment to a value representing the expected cumulative reward that the agent can achieve from that state. The value function is updated using the Bellman equation, which expresses the relationship between the value of a state and the values of its neighboring states. During each iteration, the agent calculates the value of each state using the current estimate of the value function. It then updates the value function by setting the value of each state to the maximum expected reward that the agent can achieve from that state based on the current estimate of the values of its neighboring states. This process is repeated until the values of all states converge to their true values. Once the value function has been learned, the agent can use it to derive a policy that specifies the optimal action to take in each state to maximize cumulative reward. The policy is obtained by selecting the action that has the highest expected reward, based on the current estimate of the value function. The policy is then used to guide the agent's actions in the environment. The iteration process concludes when the values of all states converge to their true values.

```
utility_value_00 = []
utility_value_32 = []
utility_value_55 = []
def value_iteration(states):
    # find the value of the state until optimal
    iteration = 1
    while True:
        print("Iteration " , iteration)
        next_state = copy.deepcopy(states)
        max_diff = 0
        for row in range(BOARD_ROWS):
            for col in range(BOARD_COLS):
                utilities = []
                for action in range(NUM_OF_ACTIONS):
                    utilities.append(utility_score(states, row, col, action))

                next_state[row][col] = max(utilities)
                max_diff = max(max_diff,abs(next_state[row][col]-states[row][col]))

        states = next_state
        utility_value_00.append(states[0][0])
        utility_value_32.append(states[3][2])
        utility_value_55.append(states[5][5])
        print_states(states)

        if max_diff < EPSILON * (1-GAMMA)/GAMMA:
            break
        iteration += 1
    return states
```

II. Plot of optimal policy

One can obtain the optimal policy by executing the value iteration algorithm with the initial map as input.

```
all_states = value_iteration(all_states)
optimal_policy = get_policy(all_states)
```

Additionally, a "print policy" function has been developed to facilitate the visualization of the optimal policy.

```
def print_policy(matrix):
    for row in range(BOARD_ROWS):
        print("|", end="")
        for col in range(BOARD_COLS):
            if (row, col) in WALLS:
                print(" Wall ", end="|")
            else:
                if matrix[row][col] == 0:
                    print("  Up  ", end="|")
                elif matrix[row][col] == 1:
                    print(" Left ", end="|")
                elif matrix[row][col] == 2:
                    print(" Down ", end="|")
                elif matrix[row][col] == 3:
                    print(" Right", end="|")
        print()
```

```
Optimal Policy

|  Up   | Wall |  Up   | Left | Left |  Up   |
|  Up   |  Up  | Left |  Up  | Wall |  Up   |
|  Up   | Left |  Up  |  Up  |  Up  | Left |
|  Up   | Left | Left |  Up  |  Up  | Left |
|  Up   | Wall | Wall | Wall |  Up  |  Up  |
|  Up   | Left | Left | Left |  Up  |  Up  |
```

III. Utilities of all states

```
Iteration  1832
|99.99| Wall|95.04|93.87|92.65|93.32|
|98.39|95.88|94.54|94.39| Wall|90.91|
|96.94|95.58|93.29|93.19|93.24|91.87|
|95.55|94.45|93.23|91.23|92.94|91.62|
|94.31| Wall| Wall| Wall|90.53|90.44|
|92.93|91.72|90.53|89.35|89.34|89.27|
```

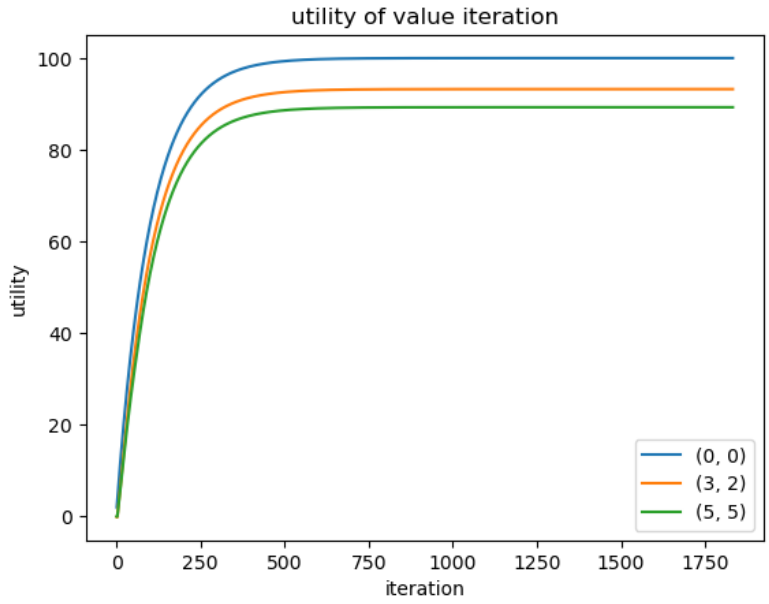
The utilities of states are represented by their coordinates in (row, col) format, with the origin located at the top left corner.

(0, 0): 99.99
(0, 1): Wall
(0, 2): 95.04
(0, 3): 93.87
(0, 4): 92.65
(0, 5): 93.32
(1, 0): 98.39
(1, 1): 95.58
(1, 2): 94.54
(1, 3): 94.39
(1, 4): Wall
(1, 5): 90.91
(2, 0): 96.94
(2, 1): 95.58
(2, 2): 93.29
(2, 3): 93.19
(2, 4): 93.24
(2, 5): 91.87
(3, 0): 95.55
(3, 1): 94.45
(3, 2): 93.23
(3, 3): 91.23
(3, 4): 92.94
(3, 5): 91.62
(4, 0): 94.31
(4, 1): Wall
(4, 2): Wall
(4, 3): Wall
(4, 4): 90.53
(4, 5): 90.44
(5, 0): 92.93
(5, 1): 91.72
(5, 2): 90.53
(5, 3): 89.35
(5, 4): 89.34
(5, 5): 89.27

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

In the previous section, the algorithm was run for a total of 1832 iterations, and the utility value of each iteration was recorded. However, due to the large number of coordinates, only a select few

were chosen for visualization purposes. Specifically, coordinates (0, 0) were represented by the blue line, (3, 2) by the orange line, and (5, 5) by the green line. These coordinates were chosen because the highest utility score was observed at (0, 0), the lowest utility score at (5, 5), and the median utility score at (3, 2). The resulting graph is displayed below for ease of interpretation.



Part 2 Policy Iteration

I. Descriptions of implemented solutions

The implementation of policy iteration is similar to that of value iteration, with the exception of the final stage, which involves the iteration process. Policy iteration is a two-step process that begins with policy evaluation. In this step, an initial policy is evaluated, and the value of each state is calculated using iterative methods such as the Bellman equation. The value of a state is defined as the expected sum of future rewards that can be obtained by following the policy from that state. The subsequent step is policy improvement. Once the value of each state has been evaluated, the policy is updated by selecting the action that maximizes the expected value of the next state. This stage is referred to as policy improvement. The process of policy evaluation and policy improvement is repeated until convergence, which is achieved when the policy no longer changes, and the policy is therefore deemed optimal.

```
def policy_making(policy, states):
    # find a utility from the given policy
    max_diff = 0
    while True:
        next_state = copy.deepcopy(states)
        max_diff = 0
        for row in range(BOARD_ROWS):
            for col in range(BOARD_COLS):
                next_state[row][col] = utility_score(states, row, col, policy[row][col])
                max_diff = max(max_diff, abs(next_state[row][col] - states[row][col]))

        states = next_state
        if max_diff < EPSILON * (1 - GAMMA)/GAMMA:
            break
    return states
```

```
utility_policy_00 = []
utility_policy_32 = []
utility_policy_55 = []
def policy_iteration(policy, state):
    # find better policy until optimal
    iteration = 1
    while True:
        state = policy_making(policy, state)
        modified = 0

        for row in range(BOARD_ROWS):
            for col in range(BOARD_COLS):
                best_action = None
                best_utility = -float("inf")
                for action in range(NUM_OF_ACTIONS):
                    ut = utility_score(state, row, col, action)
                    if ut > best_utility:
                        best_action = action
                        best_utility = ut

                if best_utility > utility_score(state , row, col, policy[row][col]):
                    policy[row][col] = best_action
                    modified = 1

        print("Iteration ", iteration)
        utility_policy_00.append(state[0][0])
        utility_policy_32.append(state[3][2])
        utility_policy_55.append(state[5][5])
        print_policy(policy)

        if modified == 0:
            break
        iteration += 1

    return policy, state
```

II. Plot of optimal policy

```
Optimal Policy

|  Up  | Wall |  Up  | Left | Left |  Up  |
|  Up  |  Up  | Left |  Up  | Wall |  Up  |
|  Up  | Left |  Up  |  Up  |  Up  | Left |
|  Up  | Left | Left |  Up  |  Up  | Left |
|  Up  | Wall | Wall | Wall |  Up  |  Up  |
|  Up  | Left | Left | Left |  Up  |  Up  |
```

III. Utilities of all states

```
|99.99| Wall|95.04|93.87|92.65|93.32|
|98.39|95.88|94.54|94.39| Wall|90.91|
|96.94|95.58|93.29|93.19|93.24|91.87|
|95.55|94.45|93.23|91.23|92.94|91.62|
|94.31| Wall| Wall| Wall|90.53|90.44|
|92.93|91.72|90.53|89.35|89.34|89.27|
```

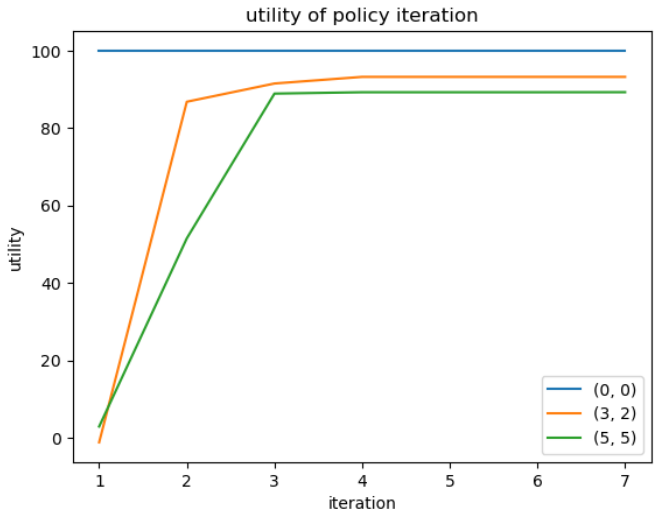
The utilities of states are represented by their coordinates in (row, col) format, with the origin located at the top left corner.

(0, 0): 99.99
(0, 1): Wall

(0, 2): 95.04
(0, 3): 93.87
(0, 4): 92.65
(0, 5): 93.32
(1, 0): 98.39
(1, 1): 95.58
(1, 2): 94.54
(1, 3): 94.39
(1, 4): Wall
(1, 5): 90.91
(2, 0): 96.94
(2, 1): 95.58
(2, 2): 93.29
(2, 3): 93.19
(2, 4): 93.24
(2, 5): 91.87
(3, 0): 95.55
(3, 1): 94.45
(3, 2): 93.23
(3, 3): 91.23
(3, 4): 92.94
(3, 5): 91.62
(4, 0): 94.31
(4, 1): Wall
(4, 2): Wall
(4, 3): Wall
(4, 4): 90.53
(4, 5): 90.44
(5, 0): 92.93
(5, 1): 91.72
(5, 2): 90.53
(5, 3): 89.35
(5, 4): 89.34
(5, 5): 89.27

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

The plotting template remains consistent with the previous section. However, as a result of the randomization of the initial policy, the graph will not be identical each time the code is executed. The following example displays a sequence of seven iterations.



Part 3 Bonus Questions

New Maze

	-1	-1	-3	5	-3
Wall		Wall	-3		Wall
	-1			Wall	-3
-1	start	-1	-3		5
	-1			Wall	-1
1		Wall	-1		1

The maze in question offers a notably elevated reward (+5) and penalty (-3), yielding a relatively high ratio of penalty to reward. Furthermore, this maze boasts a higher density than its predecessor. Despite these added complexities, the agent succeeded in learning the optimal policy, albeit requiring 1961 iterations, a 7% increase from the previous maze's performance.

In summary, the added complexity of the maze did not prevent the agent from eventually learning the correct policy. However, the larger number of states and heightened intricacy of the maze translated into a longer learning time and greater difficulty in achieving utility convergence.