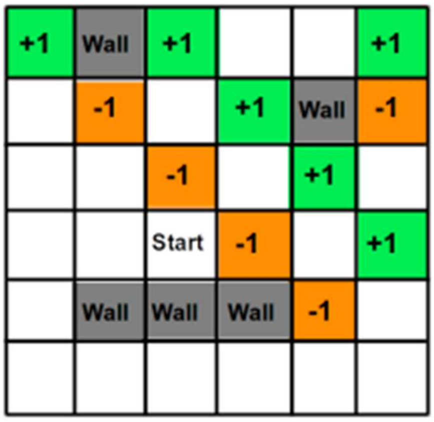
SCHOOL OF COMPUTER SCIENCE & ENGINEERING

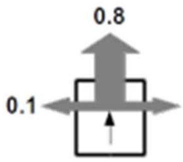
CZ4046 – Intelligent Agents

ASSIGNMENT 1

Name : Wu Rongxi
Matriculation Number: U2020719J

MAZE





PART 1: Value Iteration

I. Description of implemented solution

- Set up the maze by using for loop to initialize the size of the maze (6 x 6).

- Declare the coordinates of good states and the bad states as well as the corresponding rewards(+1) and penalty(-1) each time the agent moves(-0.04)

- Constants such as the discount factor(GAMMA = 0.99) used in the Bellman's equation and terminating factor for non-terminating states (EPSILON=10^-6), are being declared.

- To make the algorithm more dynamic, NUM_OF_ACTIONS is declared to be 4, with the list ACTIONS representing the 4 directions (UP, LEFT, DOWN, RIGHT), agent can choose to move towards. Each time the agent wants to move towards a certain direction, it has a possibility of 0.1 moving to the left and the right of that direction, this is made dynamic through taking the corresponding index of that direction (+1 to go RIGHT, -1 to go LEFT) mod 4, which will be useful later when finding the utility, for the best policy the agent can take at each state.

- All grids that are not WALLS or GOOD_STATE or BAD_STATE is being initialised as 0 at the start.

- **print_states(matrix)** function to print out the state containing the values for each state. This is useful for debugging as well as for the user to see how the values change as it approaches the optimal policy during the running of the program.

```python
def print_states(matrix):
    for row in range(BOARD_ROWS):
        for col in range(BOARD_COLS):
            if (row, col) in WALLS:
                print("WALL", end=" | ")
            else:
                print(str(matrix[row][col])[:6], end=" | ")

        print()
```

- **get_utility(states, row, col, action)** function returns the new state of the agent if it successfully moved, returns the original state if the agent is being blocked by barriers.

```python
def get_utility(states, row, col, action):      #to get the utility of that action in that state
    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]
```

- **evaluate_utility(states, row, col, action)** function will calculate the utility of each state for every iteration based on the utility value of previous iterations and return the value of that state in the iteration.

```python
def evaluate_utility(states, row, col, action):    #get best utility of that state
    utility = NT_REWARDS
    #non terminating states: even when at good states and bad states the IA continues to move
    if (row, col) in GOOD_STATES:
        utility = 1
    elif (row, col) in BAD_STATES:
        utility = -1
    utility += 0.1 * (GAMMA * get_utility(states,row,col,(action-1)%4))
    utility += 0.1 * (GAMMA * get_utility(states,row,col,(action+1)%4))
    utility += 0.8 * (GAMMA * get_utility(states,row,col,(action)))

    return utility
```

- **value_iteration(states)** function is the most main function of this algorithm:

```python
def value_iteration(states):     #find the value of each state
    iteration = 1
    while True:
        print("Iteration " , iteration)

        next_state = copy.deepcopy(states)   #take the previous states and clone it to modify it
        max_diff = 0
        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  # this is for terminating state


                utilities = []
                for action in range(NUM_OF_ACTIONS):
                    utilities.append(evaluate_utility(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
        print_states(states)
        if max_diff < EPSILON * (1-GAMMA)/GAMMA:     #stop iterating once max_diff between previous iteration and current iteration < epsilon
            break
        iteration += 1
    return states
```

Every single iteration, the **states** matrix will be cloned to next_state (deepcopy function in copy library), max_diff will be set to zero which will be used to later compare with EPSILON to know when the while loop can terminate. Two for loops are used to move through every row and column, a list **Utilities** is being newly initialized in the for loop (BOARD_COLS), for every direction the agent may choose to move at a grid, the utility is being calculated using **evaluate_utility** and then appended into the **Utilities** list. The max value in the **Utilities** list will be the policy for the next_state.
The max_diff will be evaluated using ‘max_diff = max(max_diff, abs(next_state[row][col]-states[row][col]))’ to find the maximum difference state and next state have.
The program will terminate once max_diff is less than EPSILON.

- **get_policy(states)** function is used to get the policy by analysing the values of each grid around the agent. For example, if the value to the right of the agent is larger than that of the other 3 directions, the agent will move RIGHT in this case.

```python
def get_policy(states):    #find the best move the AI can take at that state
    policy = [[None] * BOARD_COLS for i in range(BOARD_ROWS)]
    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   #uncomment for terminating
            best_action = -99999
            best_utility = -99999
            for action in range(NUM_OF_ACTIONS):
                utility = evaluate_utility(states, row, col, action)
                if utility > best_utility:
                    best_utility = utility
                    best_action = action
            policy[row][col] = best_action

    return policy
```

II. OPTIMAL POLICY

```
***************Best Policy*********************


+1 (U) |  WALL  | +1 (L) |  Left  |  Left  | +1 (U) |
  Up   | -1 (L) |  Left  | +1 (L) |  WALL  | -1 (U) |
  Up   |  Left  | -1 (L) |   Up   | +1 (L) |  Left  |
  Up   |  Left  |  Left  | -1 (U) | +1 (U) |  Left  |
  Up   |  WALL  |  WALL  |  WALL  | -1 (U) |   Up   |
  Up   |  Left  |  Left  |  Left  |   Up   |   Up   |
```

The optimal policy after 1832 iterations

III. Utilities of all states (row, col)

(0, 0): 99.999
(0, 1): WALL
(0, 2): 95.045
(0, 3): 93.875
(0, 4): 92.654
(0, 5): 93.328
(1, 0): 98.393
(1, 1): 95.883
(1, 2): 94.544
(1, 3): 94.399
(1, 4): WALL
(1, 5): 90.918
(2, 0): 96.948
(2, 1): 95.586
(2, 2): 93.294
(2, 3): 93.191
(2, 4): 93.240
(2, 5): 91.878
(3, 0): 95.553
(3, 1): 94.452
(3, 2): 93.232
(3, 3): 91.239
(3, 4): 92.949
(3, 5): 91.626
(4, 0): 94.312
(4, 1): WALL
(4, 2): WALL
(4, 3): WALL
(4, 4): 90.533
(4, 5): 90.444
(5, 0): 92.937
(5, 1): 91.728

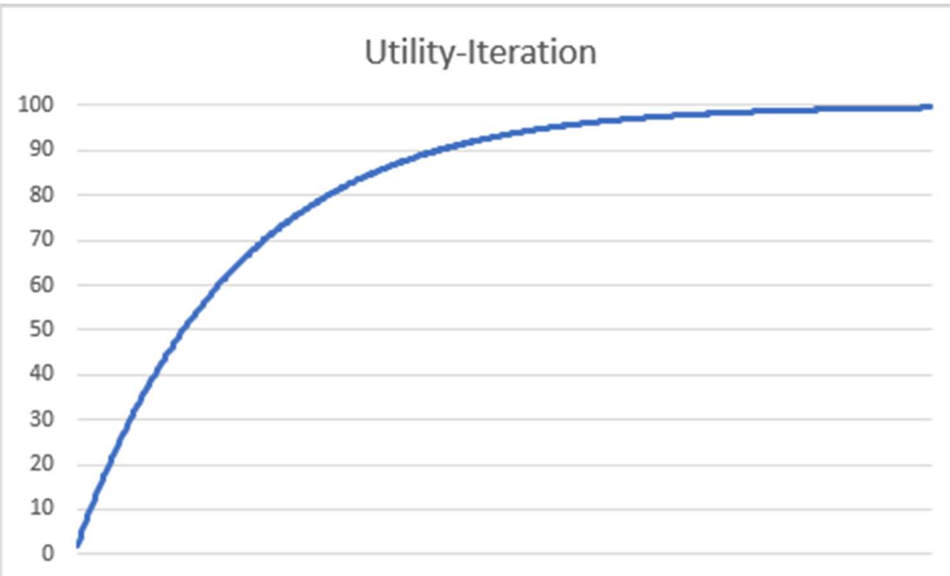
(5, 2): 90.535
(5, 3): 89.356
(5, 4): 89.246
(5, 5): 89.275

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

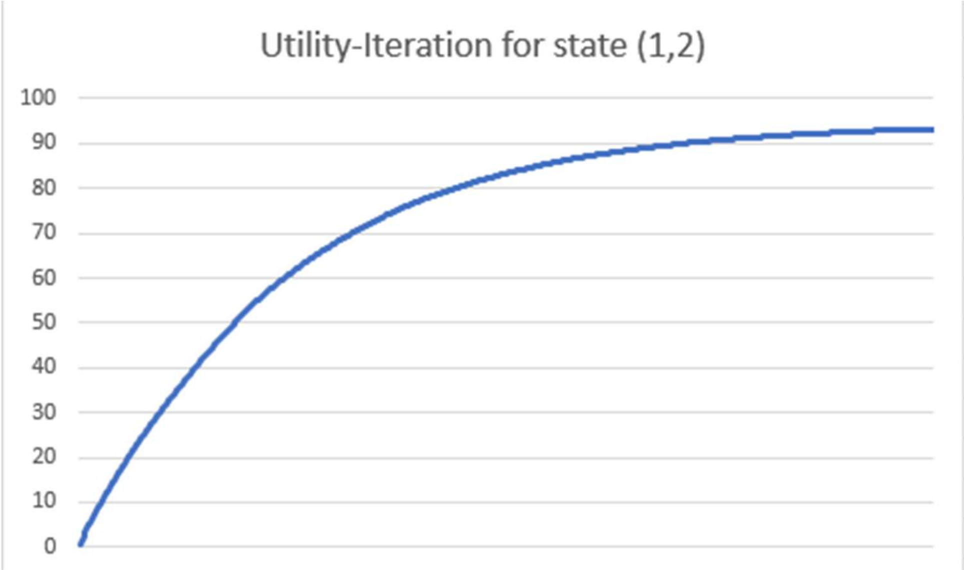
```
Iteration  1832
99.999 | WALL | 95.045 | 93.875 | 92.654 | 93.328 |
98.393 | 95.883 | 94.544 | 94.399 | WALL | 90.918 |
96.948 | 95.586 | 93.294 | 93.191 | 93.240 | 91.878 |
95.553 | 94.452 | 93.232 | 91.239 | 92.949 | 91.626 |
94.312 | WALL | WALL | WALL | 90.533 | 90.444 |
92.937 | 91.728 | 90.535 | 89.356 | 89.346 | 89.275 |
```

Utility values after 1832 iterations

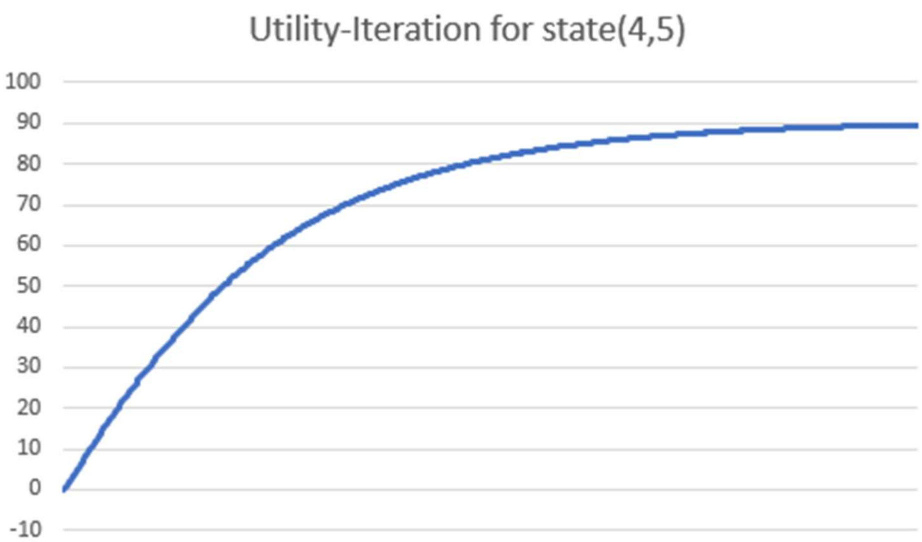
Graph-plot for Utility-Iteration



The increase in utility value converges as it approaches 100 (state (0,0))



Graph Plot utility value for state (1,2)



Graph Plot utility value for state (1,2)

PART 2: Policy Iteration

I. **Description of implemented solution** *(first 7 points of policy iteration is the same as those of value iteration.)*

1. Set up the maze by using for loop to initialize the size of the maze (6 x 6).

2. Declare the coordinates of good states and the bad states as well as the corresponding rewards(+1) and penalty(-1) each time the agent moves(-0.04)

3. Constants such as the discount factor(GAMMA = 0.99) used in the Bellman's equation and terminating factor for non-terminating states (EPSILON=10^-6), are being declared.

4. To make the algorithm more dynamic, NUM_OF_ACTIONS is declared to be 4, with the list ACTIONS representing the 4 directions (UP, LEFT, DOWN, RIGHT), agent can choose to move towards. Each time the agent wants to move towards a certain direction, it has a possibility of 0.1 moving to the left and the right of that direction, this is made dynamic through taking the corresponding index of that direction (+1 to go RIGHT, -1 to go LEFT) mod 4, which will be useful later when finding the utility, for the best policy the agent can take at each state.

5. All grids that are not WALLS or GOOD_STATE or BAD_STATE is being initialised as 0 at the start.

6. **get_utility(states, row, col, action)** function returns the new state of the agent if it successfully moved, returns the original state if the agent is being blocked by barriers.

7. **evaluate_utility(states, row, col, action)** function will calculate the utility of each state with the help of **get_utility** for every iteration based on the utility value of previous iterations and return the value of that state in the iteration.

8. **print_states(matrix)** function to print out the state containing the values for each state. This is useful for debugging as well as for the user to see how the values change as it approaches the optimal policy during the running of the program.

9. In policy iteration, a set of policies is being randomized at the beginning instead of starting from zero like value iteration, the agent will decide if it is better to follow the original policy or to find a better policy based on the calculated utility. Functions such as **policy_making(policy, state), evaluate_utility(states, row, col, action)** and **get_utility(states, row, col, action)** will be used in the function. The iteration will terminate once the best policy is found (no changes is made between current and previous policy)

10. **policy_making(policy, states)**:

```python
def policy_making(policy, states):   #finding a utility given the current policy and state
    max_diff = 0
    while True:
        next_state = copy.deepcopy(states)    #clone the states to next_state
        max_diff = 0
        for row in range(BOARD_ROWS):
            for col in range(BOARD_COLS):
                next_state[row][col] = evaluate_utility(states, row, col, policy[row][col])
                max_diff = max(max_diff, abs(next_state[row][col] - states[row][col]))  #find max diff between current
                                                                                         #state and previous state
        states = next_state
        #print("change")
        if max_diff < EPSILON * (1 - GAMMA)/GAMMA:
            break
    return states
```

This function is like the first part of the value_iteration function, the **states** matrix will be cloned to next_state (deepcopy function in copy library), max_diff will be set to zero which will be used to later compare with EPSILON to know when the while loop can terminate. It will find the utilities in the state for their current policy and return it.

11. **policy_iteration(policy, state)** function is used to discover a better policy given the current policy and state:

```python
def policy_iteration(policy, state):      #find a better policy or original policy remains based on utility value
    iteration = 1
    while True:
        state = policy_making(policy, state)        #return the state with corresponding utility
        modified = 0     #policy not being modified yet; 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 = evaluate_utility(state, row, col, action)   #find the best utility and corresponding action
                    if ut > best_utility:
                        best_action = action
                        best_utility = ut

                if best_utility > evaluate_utility(state , row, col, policy[row][col]):   #update utility and policy
                    policy[row][col] = best_action
                    modified = 1   #policy is being modified

        print("Iteration ", iteration)
        print_policy(policy)
        #print(policy)

        if modified == 0:   #final policy will be generated when changes is not needed anymore
            break
        iteration += 1

    return policy, state
```

In every single iteration, it will first find out the utility of the current policy, **modified** will be set to 0(not changed yet), proceed to find the action of highest utility value using evaluate utility, if the current policy's utility value is lower than the best utility value, the policy will be updated for that grid(**modified =1)** and the cycle repeats until all rows and columns are being checked through. If the current iteration's policy is no different from the previous iteration's policy the program will terminate(**modified remains 0)** and return the final policy and state utility values.

12. Every time the policy is being randomized at the beginning of the program, so there is no fixed number of iterations for the agent to find the optimal policy, however the number of times of iteration is generally kept under **10**

II. OPTIMAL POLICY

```
***************Best Policy*********************

+1 (U) |  WALL  | +1 (U) |  Left  |  Right | +1 (U) |
  Up   | -1 (L) |   Up   | +1 (R) |  WALL  | -1 (U) |
  Up   |  Left  | -1 (U) |   Up   | +1 (U) |  Left  |
  Up   |  Left  |  Left  | -1 (R) | +1 (U) |  Left  |
  Up   |  WALL  |  WALL  |  WALL  | -1 (U) |   Up   |
  Up   |  Left  |  Left  |  Right |   Up   |   Up   |
```

The optimal policy after 5 iterations

III. Utilities of all states

(0, 0): 1
(0, 1): WALL
(0, 2): 1
(0, 3): 93.875
(0, 4): 92.654
(0, 5): 1
(1, 0): 98.393
(1, 1): -1
(1, 2): 94.544
(1, 3): 1
(1, 4): WALL
(1, 5): -1
(2, 0): 96.948
(2, 1): 95.586
(2, 2): -1
(2, 3): 93.191
(2, 4): 1
(2, 5): 91.878
(3, 0): 95.553
(3, 1): 94.452
(3, 2): 93.232
(3, 3): -1
(3, 4): 1
(3, 5): 91.626
(4, 0): 94.312
(4, 1): WALL
(4, 2): WALL
(4, 3): WALL
(4, 4): -1
(4, 5): 90.444
(5, 0): 92.937
(5, 1): 91.728
(5, 2): 90.535
(5, 3): 89.356

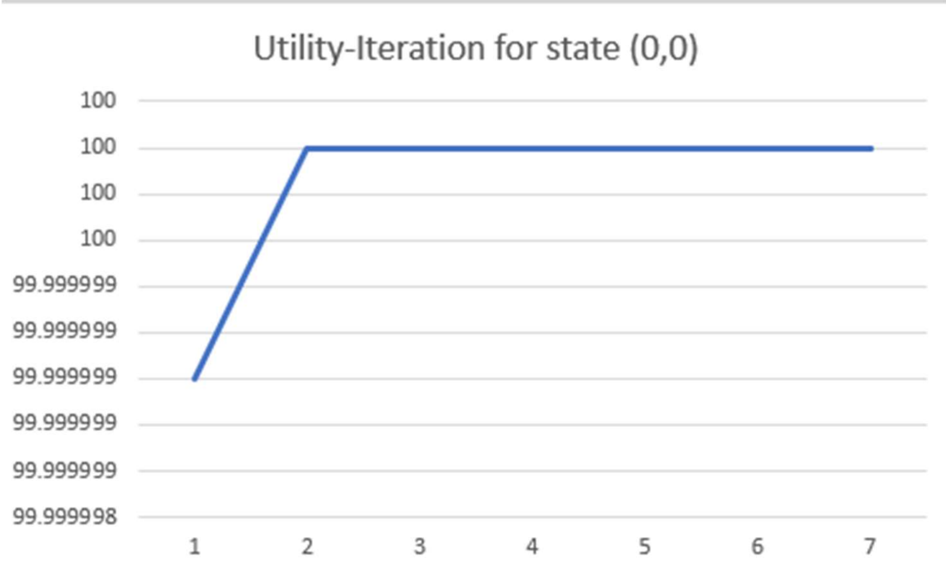
(5, 4): 89.346
(5, 5): 89.275

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

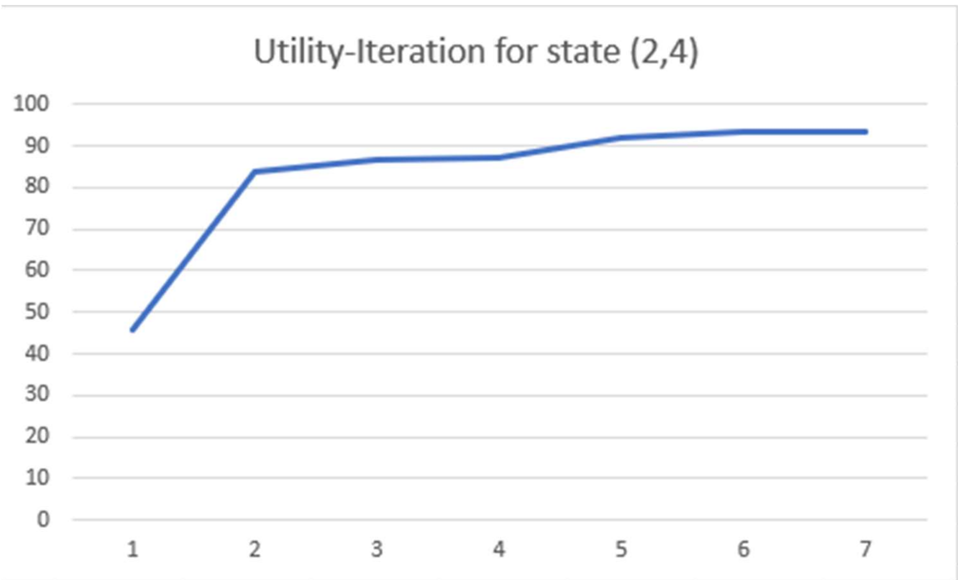
```
 1    | WALL |  1    | 93.875 | 92.654 |  1    |
98.393 |  -1   | 94.544 |  1    | WALL |  -1   |
96.948 | 95.586 |  -1   | 93.191 |  1    | 91.878 |
95.553 | 94.452 | 93.232 |  -1   |  1    | 91.626 |
94.312 | WALL | WALL | WALL |  -1   | 90.444 |
92.937 | 91.728 | 90.535 | 89.356 | 89.346 | 89.275 |
```

Utility values after 5 iterations

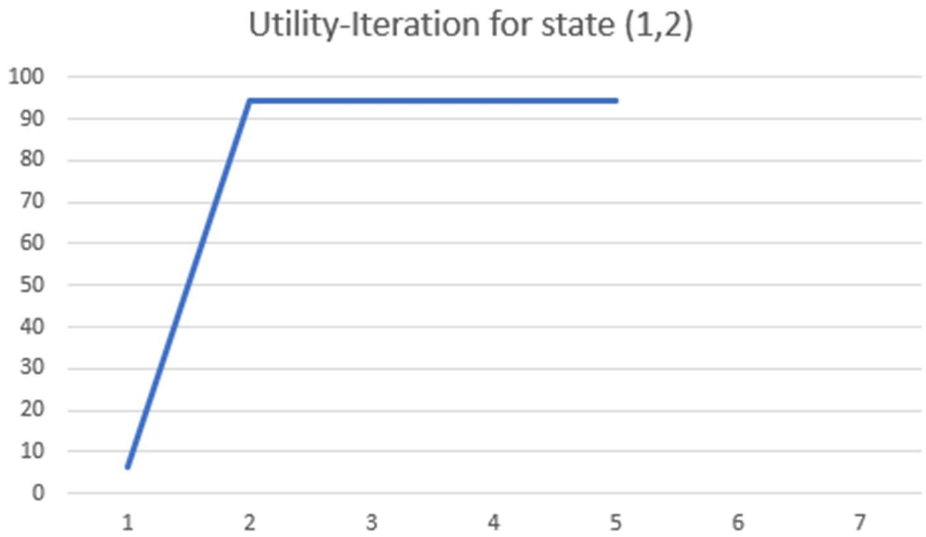
Graph-Plot for utility of policy iteration(random initial policy)



Best case scenario: initial policy is the correct policy.



Graph Plot utility value for state (2,4) after 7 iterations



Graph Plot utility value for state (1,2) after 5 iterations

V. Conclusions: Learning Outcomes

1. Policy iteration is faster than value iteration generally however each iteration takes longer since there is a need to check through the current policy again each time a policy is being modified, making it a O(n^2) time.

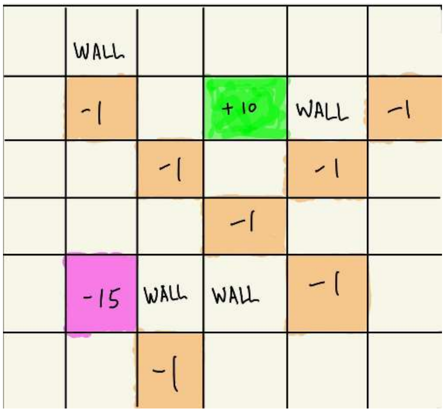
2. It is observed that when the cost for moving in each state(non-rewards/non-penalty) is being decreased from -0.04 to -0.01, the agent will take shorter time to converge and reach its optimal policy.

3. It is also observed that when GAMMA is being increased, the number of iterations will increase, this is because when GAMMA is a discount factor and when it is being increased, the impact of future rewards or penalty will be discounted more, resulting in lesser impact to the utility value, the agent can afford to try out more actions that may not be right.

4. Point 2 and 3 are interestingly similar to how humans learn, when the impact and consequences of children's actions are little, they can make more mistakes on the way to finding the right way to do things, whereas if great pressure is being put on the child such that their action will impact their future significantly, they will be forced to find the right way to do things quickly.

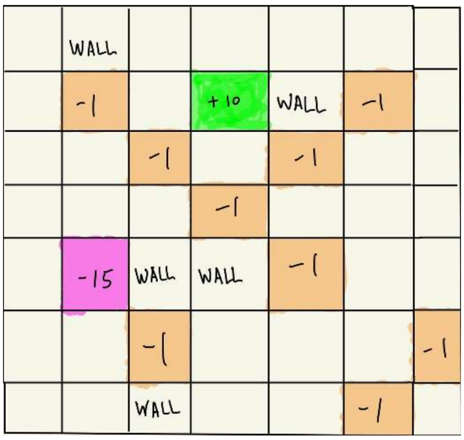
Bonus

New Maze 1



A maze where there are mostly penalty and only one state is the high reward(+10) and one state with heavy penalty(-15), the agent took 2034 iterations which is slightly longer than previous maze where there are relatively less penalty and more rewards, we can draw a conclusion that having more penalty may not help the agent to learn faster.

New Maze 2



When the maze is being expanded to be bigger, more penalty than rewards, the agent take about the same number of iterations: 2034 to find the right policy.

Conclusion

1.The utility converges slower when more penalty is present than rewards. Therefore, it is better to have more rewards than penalty to help the agent learn faster.

2.Slight changes to the size of the maze will not affect the time taken for the agent to find the optimal policy.