NANYANG TECHNOLOGICAL UNIVERSITY

CZ4046 Intelligent Agents

Assignment 1 – Agent Decision Making

Jervis Chan Jun Yong
U1821352H

Table of Contents

1. Introduction

In this assignment, we are tasked to solve a Markov Decision Process (MDP). The MDP is composed as follows:

States - s

We are provided with a 6 x 6 grid environment (Figure 1.1).

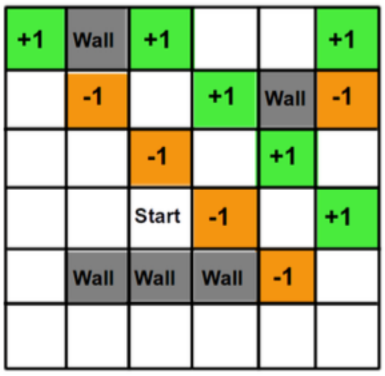


Figure 1.1. 6 x 6 Grid Environment

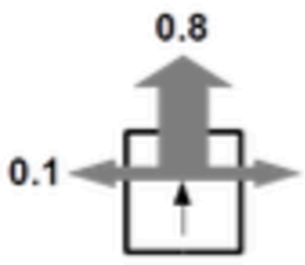
There are four kinds of squares in the grid: green, white, orange and wall. As an agent cannot be in the wall, there are 31 possible states. The states are defined by coordinates which are in the (row, col) format, with the top left corner being (0,0).

Actions – A(s)

In this maze, there are four possible actions, namely up, down, left and right. At each state, an agent can take three of the four possible actions. It can move in its intended direction or at right angle to the intended direction. If the move makes the agent walk into a wall, the agent will stay in the same place.

Transition model – P (s′ | s, a)

The transition model is stated as follows (Figure 1.2). The transition properties depend only on the current state and not on the previous history, demonstrating the Markov Property.



P (intended s′ | s, a) = 0.8
P (unintended s′ | s, a) = 0.1

Figure 1.2. Transition Model of Agent

Reward function – R(s)

The reward function gives us the reward given when the agent is a certain state.

R(white square) = -0.04
R(green square) = +1
R(brown square) = -1

Policy π(s)

The policy π(s) gives us the action that the agent will take in any given state. An optimal policy should maximise the expected utility over all possible state sequences. In this assignment, we aim to find the optimal policy to the MDP using **value iteration** and **policy iteration.**

As there are no terminal states, the agent's state sequence could be infinite. Thus, we will use a discount factor (γ) of 0.99 to ensure that the total utility will stay bounded and allow the algorithms to converge.

2. Value Iteration

Description of Implementation

In value iteration, we calculate the utility of each state and use the state utilities to decide on the optimal action in each state.

The implementation can be described in the following steps:

1. Create dictionaries which will store current and next utility values of each state in the MDP. The utility values of each state are initialised to 0.
2. Initialise variable which will hold the maximum change in utility at any iteration.
3. Calculate convergence threshold using the following formula:
   $$Convergence\ Threshold = \frac{\epsilon\,(1-\gamma)}{\gamma}$$
   ϵ represents Epsilon, which is the maximum allowed error in utility. It can be set at different values to see how the convergence of value iteration will be affected.
4. Iterate till convergence
   a. Perform Bellman Update for each state to calculate utility for the next state and set action that maximises expected utility. The equation for the Bellman Update is as follows:
   $$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s')$$
   b. Calculate maximum change in utility among all states and compare against convergence threshold. Convergence is reached when maximum change is less than the convergence threshold.

Code Implementation of Value Iteration

The above implementation can be seen in the code snippets below. Three functions are created:

1. *value_iteration* (Figure 2.1)
   This function is the main code implementation of the value iteration algorithm above.

2. *bellman_update* (Figure 2.2)
   This function is the implementation of the Bellman Update formula. For each action, the function will call get_expected_utility to calculate the utility of that action and determine what is the action that will return the maximum utility.

3. *get_expected_utility* (Figure 2.3)
   This function will be given the current state and action and returns the expected utility.

```python
def value_iteration(mdp, EPSILON):
    # LOCAL VARIABLES
    U_current, U_next = {}, {}  # U, U' - dictionary of utilities for states
    max_utility_change = 0 # δ - maximum change in utility at any iteration
    convergence_threshold = (EPSILON * (1 - mdp.discount))/mdp.discount

    # Supplementary dictionaries for visualisation
    U_iterations, optimal_policy, policy_loss= {}, {}, []

    for state in mdp.states:
        #initialise utility value of all states in U_current and U_next to be 0
        U_current[state] = 0
        U_next[state] = 0
        U_iterations[state] = []
        optimal_policy[state] = None

    num_iterations = 0
    converged = False

    while not converged:
        # U ← U’; δ ← 0
        for state in mdp.states:
            U_iterations[state].append(U_current[state])
            U_current[state] = U_next[state] # U ← U’

        max_utility_change = 0 # δ

        for state in mdp.states:
            updated_utility, best_action = bellman_update(mdp, state, U_current)
            U_next[state] = updated_utility
            optimal_policy[state] = best_action

            abs_utility_change = abs(U_next[state] - U_current[state])
            if abs_utility_change > max_utility_change:
                max_utility_change = abs_utility_change

        num_iterations += 1
        policy_loss.append(max_utility_change)
        if max_utility_change < convergence_threshold:
            converged = True

    return {'U_current': U_current, 'U_iterations': U_iterations, 'optimal_policy': optimal_policy, 'num_iterations' : num_iterations,'policy_loss': policy_loss}
```

Figure 2.1. Code Snippet of value_iteration Function

```python
def bellman_update(mdp, state, U_current):
    """
    Bellman Update:  Ui+1(s)←R(s)+γ max ∑ P(s’|s,a)Ui(s’)

    """
    max_expected_utility = float('-inf')
    best_action = None

    for action in mdp.actions:
        expected_utility = get_expected_utility(mdp, state, action, U_current)
        if expected_utility > max_expected_utility:
            max_expected_utility = expected_utility
            best_action = action

    utility = mdp.reward_function(state) + mdp.discount * max_expected_utility
    return (utility, best_action)
```

Figure 2.2. Code Snippet of bellman_update Function

```
def get_expected_utility(mdp, state, action, U_current):
    """
    Calculates sum of expected utility: ∑s’P(s’|s, a)U[s’]
    """
    sum_expected_utility = 0
    next_states = mdp.get_next_states(state, action)

    for next_state in next_states:
        probability = mdp.transition_model(state, action, next_state)
        actual_next_state = next_states[next_state]["actual"]
        sum_expected_utility += probability * U_current[actual_next_state]

    return sum_expected_utility
```

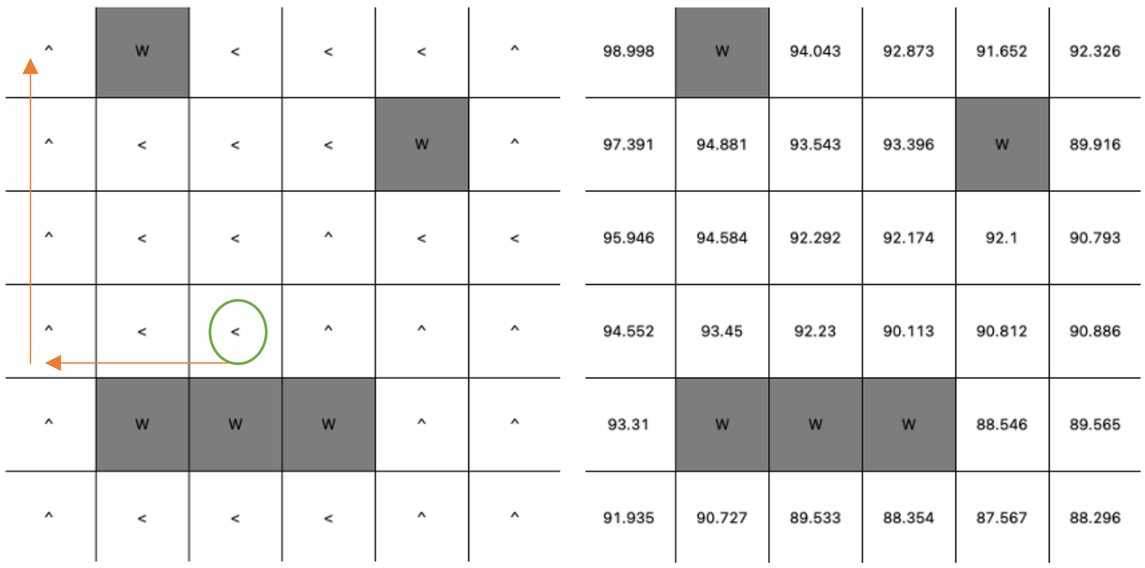
Figure 2.3. Code Snippet of get_expected_utility Function

Results

The following plots are obtained using ϵ value of 1 and γ of 0.99. With these values, convergence was reached after 459 iterations.

Plot of Optimal Policy

Figure 2.4 shows the final plot of the optimal policies of all states after convergence of the value iteration algorithm.



^	W	<	<	<	^
^	<	<	<	W	^
^	<	<	^	<	<
^	<	<	^	^	^
^	W	W	W	^	^
^	<	<	<	^	^

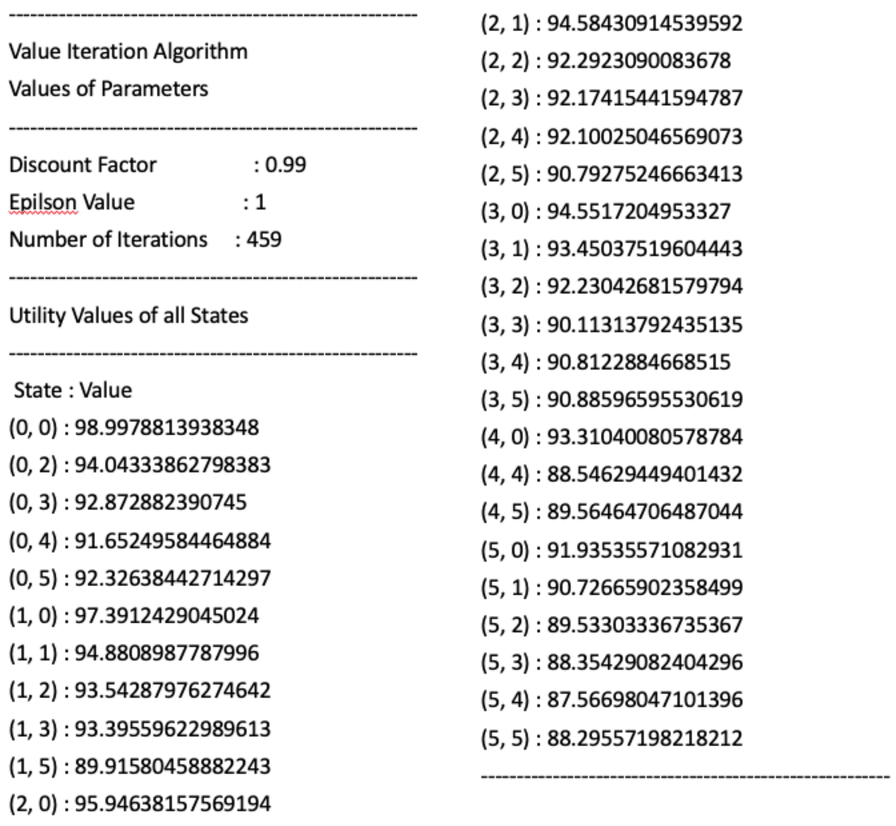
98.998	W	94.043	92.873	91.652	92.326
97.391	94.881	93.543	93.396	W	89.916
95.946	94.584	92.292	92.174	92.1	90.793
94.552	93.45	92.23	90.113	90.812	90.886
93.31	W	W	W	88.546	89.565
91.935	90.727	89.533	88.354	87.567	88.296

Figure 2.4. Value Iteration Optimal Policy Plot with $\epsilon = 1$

As seen from Figure 2.4, the optimal policy given the starting point of (3,2) is to move left then move to the top towards the reward.

Utilities of All States

Figure 2.5 shows the final policies of all states after convergence of the value iteration algorithm.



```
-------------------------------------------------------
Value Iteration Algorithm
Values of Parameters
-------------------------------------------------------
Discount Factor            : 0.99
Epilson Value              : 1
Number of Iterations     : 459
-------------------------------------------------------
Utility Values of all States
-------------------------------------------------------
 State : Value
(0, 0) : 98.9978813938348
(0, 2) : 94.04333862798383
(0, 3) : 92.872882390745
(0, 4) : 91.65249584464884
(0, 5) : 92.32638442714297
(1, 0) : 97.3912429045024
(1, 1) : 94.8808987787996
(1, 2) : 93.54287976274642
(1, 3) : 93.39559622989613
(1, 5) : 89.91580458882243
(2, 0) : 95.94638157569194
(2, 1) : 94.58430914539592
(2, 2) : 92.2923090083678
(2, 3) : 92.17415441594787
(2, 4) : 92.10025046569073
(2, 5) : 90.79275246663413
(3, 0) : 94.5517204953327
(3, 1) : 93.45037519604443
(3, 2) : 92.23042681579794
(3, 3) : 90.11313792435135
(3, 4) : 90.8122884668515
(3, 5) : 90.88596595530619
(4, 0) : 93.31040080578784
(4, 4) : 88.54629449401432
(4, 5) : 89.56464706487044
(5, 0) : 91.93535571082931
(5, 1) : 90.72665902358499
(5, 2) : 89.53303336735367
(5, 3) : 88.35429082404296
(5, 4) : 87.56698047101396
(5, 5) : 88.29557198218212
-------------------------------------------------------
```

Figure 2.5. Final Utility of All States with $\epsilon = 1$

Plot of Utility Estimates as a Function of Number of Iterations

Figure 2.6 shows the plot of utility estimates against number of iterations of the value iteration algorithm.

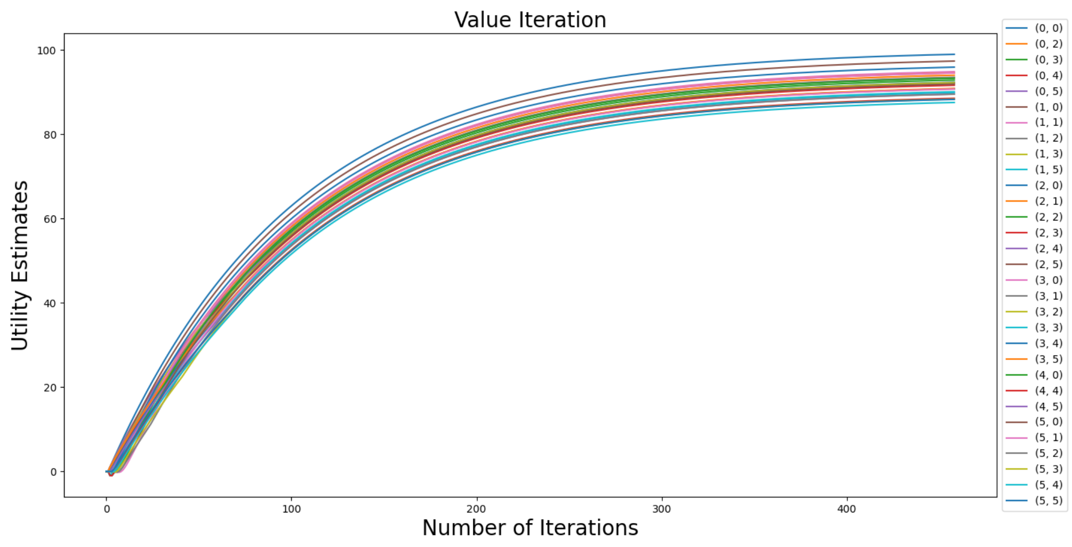


Figure 2.6. Plot of Utility Estimates Against Number of Iterations

Studying Effects of ϵ Against Convergence Iteration

In addition to performing value iteration to obtain the optimal policy, we studied the effects of ϵ on convergence. This was done by varying the value of ϵ was varied and seeing how many iterations it takes to achieve convergence.

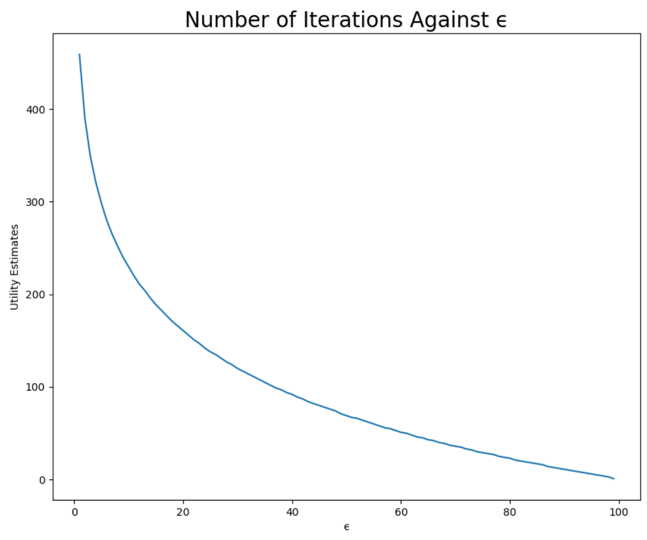


Figure 2.7. Plot of Utility Estimates Against Number of Iterations

As seen from Figure 2.7, reducing ϵ will help to reduce the number of iterations it takes for the value iteration to converge. This directly relates to allowing a larger margin of error. However, this larger margin of error could also result in incorrect policies. Thus, more experimentation could be done within that range to find the optimal ϵ which can balance the correctness of the results and the time taken for convergence.

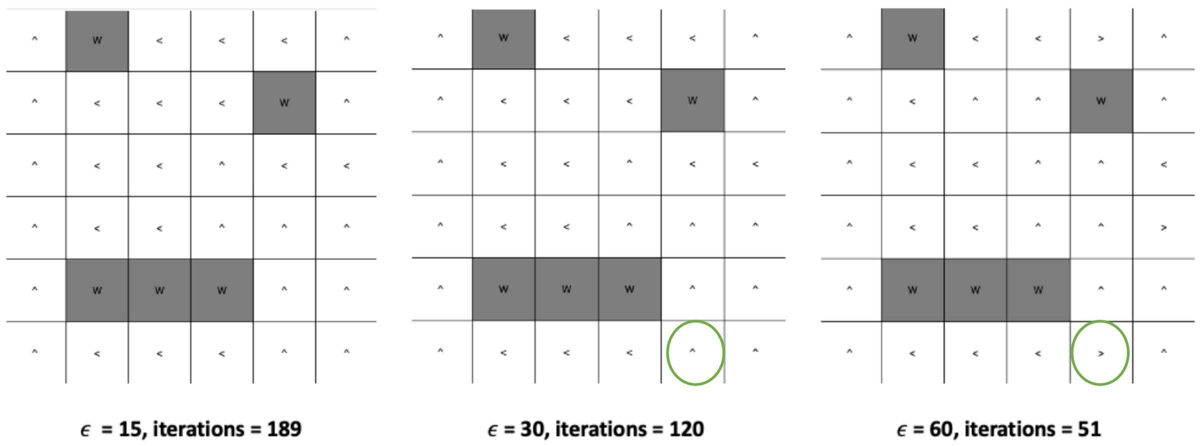

ϵ = 15, iterations = 189
ϵ = 30, iterations = 120
ϵ = 60, iterations = 51

Figure 2.8. Plot of Optimal Policy, ϵ = 15, 30, 60

As seen from Figure 2.8, the optimal policies for all states for ϵ = 15 and 30 are the same as ϵ = 1. However, when ϵ gets too large, such as ϵ = 60, we can observe differences in the optimal policy among some states.

3. Policy Iteration

Description of Implementation

In policy iteration, we start from an initial policy π_0, and alternate between **Policy Evaluation** and **Policy Improvement** until Policy Improvement yields no change in utilities.

In **policy evaluation**, we are given a policy π_i, and we calculate the utility of each state if that policy were to be executed. As the given policy helps specify the action in the current state, we can use a simplified Bellman Update as follows:

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

This is repeated *k* times to produce the next utility estimate. The value of *k* can be experimented to see how it affects convergence.

In **policy improvement**, we take the utilities calculated in policy evaluation and determine whether to change the policy by comparing if the max utility of all possible actions is higher than the utility calculated with the current policy.

The implementation can be described in the following steps:

1. Create dictionaries which will store utility values and policies of each state in the MDP. The utility values of each state are initialised to 0 and the initial policy is randomly selected.
2. Initialise *unchanged* Boolean variable to False to flag changes to policy.
3. Iterate till convergence
    a. Perform policy evaluation which calculates utility for each state using the current policy.
    b. Pass the utility values into policy improvement which will return new policy if there is a better policy among all possible actions. Convergence is reached when there is no change from policy improvement.

Code Implementation of Policy Iteration

The above implementation can be seen in the code snippets below. Three functions are created:

1. *policy_iteration* (Figure 3.1)
   This function is the main code implementation of the policy iteration algorithm above.

2. *policy_evaluation* (Figure 3.2)
   This function is the implementation of policy evaluation. The function will perform the simplified Bellman Update k times using a loop and return the updated policy values.

3. *policy_improvement* (Figure 3.3)
   This function is the implementation of policy improvement. The function will iterate through the different possible actions and determine the best possible action. It will then compare the utility value of the best action and the current policy and determine if a policy change has taken place.

```python
def policy_iteration(mdp, k):
    # LOCAL VARIRABLES
    # U , a vector of utilities for states in S , initially zero and π, a policy vector indexed by state, initially random
    utilities, policies = {}, {}

    # Supplementary dictionaries for visualisation
    U_iterations= {}

    # Initialise all states to 0 and policies to a random action
    for state in mdp.states:
        utilities[state] = 0
        policies[state] = mdp.actions[random.randint(0,3)] #the number will correspond to an action
        U_iterations[state] = [0]

    unchanged = False
    num_iterations = 0

    while not unchanged:
        utilities, new_iteration_utilities = policy_evaluation(policies, utilities, mdp, k)
        policies, unchanged = policy_improvement(mdp, utilities, policies, unchanged)
        num_iterations += 1

        for state in mdp.states:
            U_iterations[state].extend(new_iteration_utilities[state])

    return {'optimal_policy': policies, 'U_current': utilities,'U_iterations': U_iterations,'num_iterations': num_iterations }
```

Figure 3.1. Code Snippet of policy_iteration Function

```python
def policy_evaluation(policies, utilities, mdp, k):
    """
    Simplified Bellman Update
    """

    U_updated = {}
    new_iteration_utilities = {}

    for state in mdp.states:
        new_iteration_utilities[state] = []

    for i in range(int(k)):
        for state in mdp.states:
            # Calculate ∑s’P(s’|s, a)U[s’] where a is the action under π
            expected_utility = get_expected_utility(mdp, state, policies[state], utilities)

            # U_i+1(s) ← R(s) + γ ∑s’P (s'|s, π_i(s)) U_i(s')
            U_updated[state] = mdp.reward_function(state) + mdp.discount * expected_utility

       for state in mdp.states:
            utilities[state] = U_updated[state]

    for state in mdp.states:
       new_iteration_utilities[state].append(U_updated[state])

    return (utilities, new_iteration_utilities)
```

Figure 3.2. Code Snippet of policy_evaluation Function

```python
def policy_improvement(mdp, utilities, policies, unchanged):
    """
    Decide change if max ∑ a∈A(s) P(s′ |s,a) U[s′] > ∑ s' P(s′ |s,π[s]) U[s′]
    """
    #dictionary to store updated policies if policy change
    updated_policies = policies
    unchanged = True

    for state in mdp.states:
        # 1. Find max ∑ a∈A(s) P(s′ |s,a) U[s′]
        max_expected_utility = float('-inf')
        best_action = None

        for action in mdp.actions:
            expected_utility = get_expected_utility(mdp, state, action, utilities)
            if expected_utility > max_expected_utility:
                max_expected_utility = expected_utility
                best_action = action

        # 2. Find ∑ s' P(s′ |s,π[s]) U[s′]
        policy_utility = get_expected_utility(mdp, state, policies[state], utilities)

        # 3. Compare
            if max_expected_utility > policy_utility:
                updated_policies[state] = best_action
                unchanged = False

    return (updated_policies, unchanged)
```

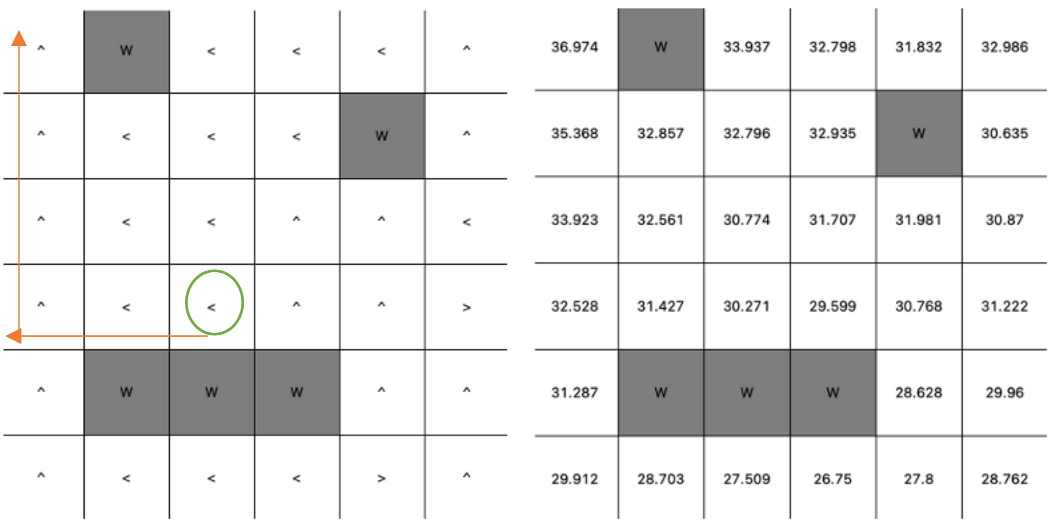
Figure 3.3. Code Snippet of policy_improvement Function

Results

With policy iteration, the number of iterations to convergence can take a range of values as the initial policy was randomised. The following plots are obtained using *k* value of 5, which reached convergence was after 10 iterations.

Plot of Optimal Policy

Figure 3.3 shows the plot of utility estimates against number of iterations of the value iteration algorithm.



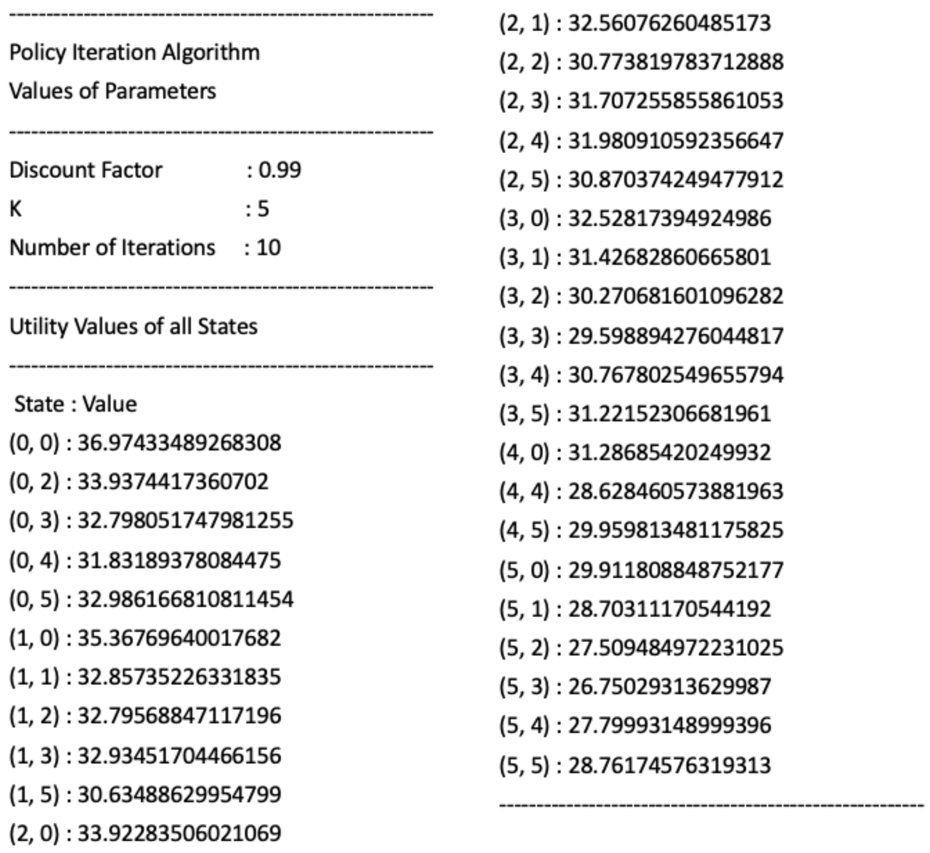
^	W	<	<	<	^
^	<	<	<	W	^
^	<	<	^	^	<
^	<	<	^	^	>
^	W	W	W	^	^
^	<	<	<	>	^

36.974	W	33.937	32.798	31.832	32.986
35.368	32.857	32.796	32.935	W	30.635
33.923	32.561	30.774	31.707	31.981	30.87
32.528	31.427	30.271	29.599	30.768	31.222
31.287	W	W	W	28.628	29.96
29.912	28.703	27.509	26.75	27.8	28.762

Figure 3.3 Policy Iteration Optimal Policy Plot with *k* = 5

As seen from Figure 3.3, the optimal policy for policy iteration, given the starting point of (3,2), is to move left then move to the top, towards the reward. This provides the same policy as value iteration.

<u>Utilities of All States</u>
Figure 3.4 shows the plot of utility estimates against number of iterations of the policy iteration algorithm. Depending on the number of iterations, the final utility value might vary even with the same *k* value. Thus, in policy iteration, it is more important to look at optimal policy.



```
--------------------------------------------------------
Policy Iteration Algorithm
Values of Parameters
--------------------------------------------------------
Discount Factor         : 0.99
K                       : 5
Number of Iterations    : 10
--------------------------------------------------------
Utility Values of all States
--------------------------------------------------------
 State : Value
(0, 0) : 36.97433489268308
(0, 2) : 33.9374417360702
(0, 3) : 32.798051747981255
(0, 4) : 31.83189378084475
(0, 5) : 32.986166810811454
(1, 0) : 35.36769640017682
(1, 1) : 32.85735226331835
(1, 2) : 32.79568847117196
(1, 3) : 32.93451704466156
(1, 5) : 30.63488629954799
(2, 0) : 33.92283506021069
(2, 1) : 32.56076260485173
(2, 2) : 30.773819783712888
(2, 3) : 31.707255855861053
(2, 4) : 31.980910592356647
(2, 5) : 30.870374249477912
(3, 0) : 32.52817394924986
(3, 1) : 31.42682860665801
(3, 2) : 30.270681601096282
(3, 3) : 29.598894276044817
(3, 4) : 30.767802549655794
(3, 5) : 31.22152306681961
(4, 0) : 31.28685420249932
(4, 4) : 28.628460573881963
(4, 5) : 29.959813481175825
(5, 0) : 29.911808848752177
(5, 1) : 28.70311170544192
(5, 2) : 27.509484972231025
(5, 3) : 26.75029313629987
(5, 4) : 27.79993148999396
(5, 5) : 28.76174576319313
--------------------------------------------------------
```

Figure 3.4. Final Utility of All States with $k = 5$

Plot of Utility Estimates as a Function of Number of Iterations

Figure 3.5 shows the plot of utility estimates against number of iterations of the value iteration algorithm.

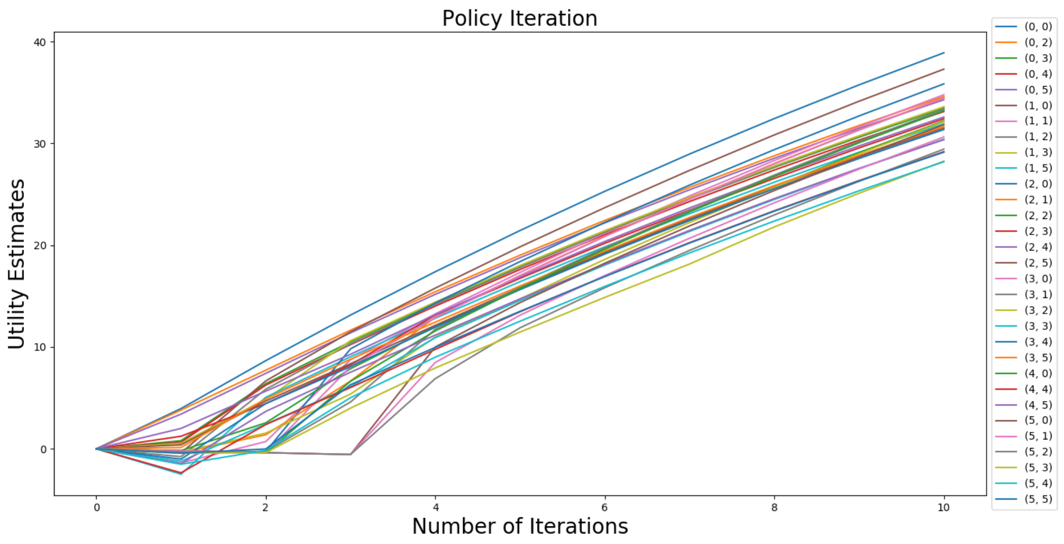
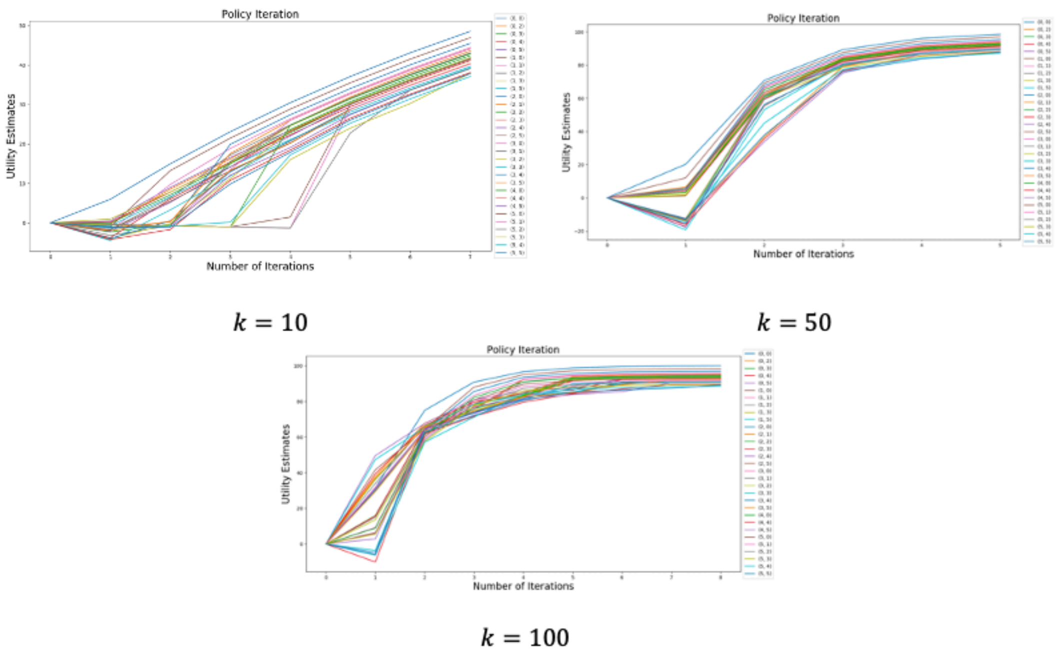


Figure 3.5. Plot of Utility Estimates Against Number of Iterations for k = 5

From Figure 3.5, we can see that the utility estimates against number of iterations for policy iteration follow a different shape compared to utility iteration. In fact, there is even cases where utility values go negative, whilst utility values for policy iteration are strictly increasing from 0. This could be attributed to the fact that random actions were used at the start for policy iteration while value iteration tries to find the best action at every iteration. In fact, as we increase the value k, we might observe that the shape increasingly becoming like that of the value iteration, with the exception of the first iteration (Figure 3.6).



$k = 10$ $k = 50$

$k = 100$

Figure 3.6. Plot of Utility Estimates Against Number of Iterations for k = 10, 50, 100

4. Bonus Questions

Description of Implementation

For the bonus question, the *generate_maze* function (Figure 4.1) was used to generate a more complex maze environment. A more complex maze can be designed by increasing the grid length of the grid or by changing the probability of different types of squares.

This function will be used to generate a maze which will be passed to the value and policy iteration functions. For value iteration, ϵ value of 1 was used, while for policy iteration, k value of 5 was used. This values were picked as they were able to obtain the correct optimal policies.

```python
def generate_maze(grid_length, special = None, wall = None):
    """
    Generates a maze with given grid length
    """
    if special and wall:
      w = 1 - special - wall
      g = o = special/2

    elif special and not wall:
       wall = (0.2/0.7) * (1-special)
       g = o = special/2
       w = (0.5/0.7) * (1-special)

    elif wall and not special:
       g = (0.15/0.8) * (1-wall)
       o = g
       w = (0.5/0.8) * (1-wall)

    elif not special and not wall:
       g = o = 0.15
       wall = 0.2
       w = 0.5

    random.seed()
    maze = []
    for row in range(grid_length):
        maze.append([])

        # WE ITERATE THROUGH EACH CELL AND ADD A CELL VALUE
        for _ in range(grid_length):
            random_color = random.random()

            if random_color < w:
                maze[row].append(' ')
            elif random_color < w + g:
                maze[row].append('G')
            elif random_color < w + g + o:
                maze[row].append('O')
            else:
                maze[row].append('W')

    return maze
```

Figure 4.1. Code Snippet of generate_maze Function

<u>Question: How Does The Number of States Affect Convergence</u>

The number of states available to the agent can be varied using the following methods:

1. Varying number of walls
2. Varying grid length

<u>Varying Number of Walls</u>

We change the wall probability provided to *generate_maze* function to vary the number of walls available in the grid. With more walls, the agent will have less available states to move in. This is implemented by varying the probability of walls while keeping grid size constant. The results from varying wall probability can be seen in Figure 5.1 and 5.2

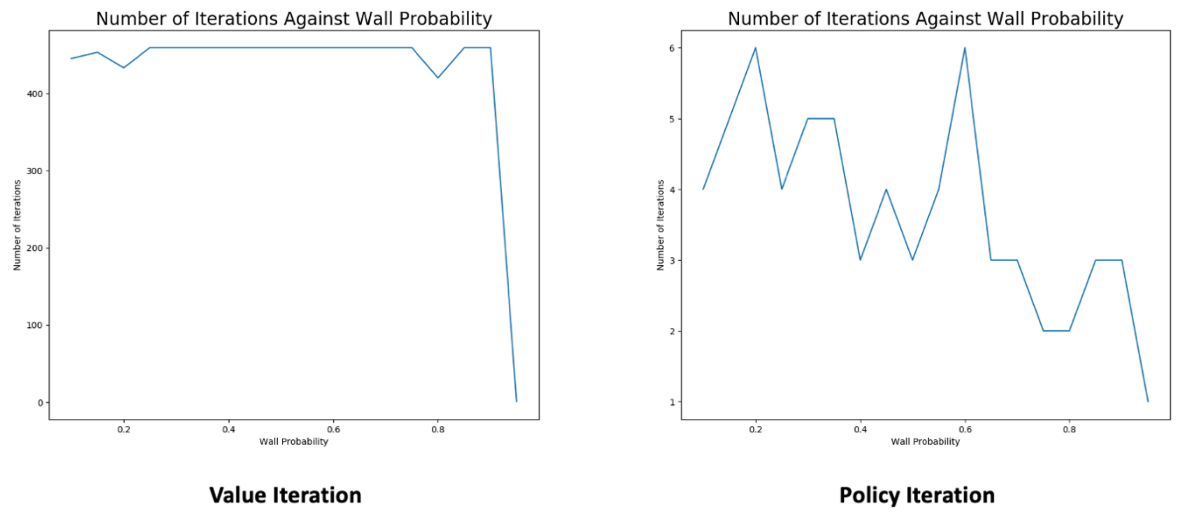


Figure 5.1. Convergence Iterations Against Wall Probability

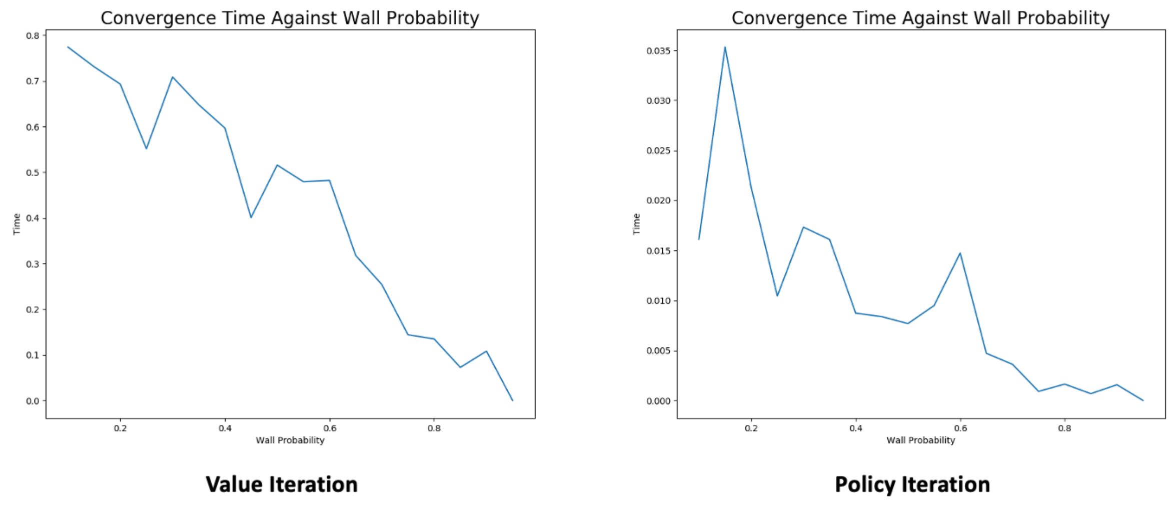


Figure 5.2. Convergence Time Against Wall Probability

In terms of convergence time, we can see that there is a decrease in convergence time as the wall probability (and in turn, number of states) increases. On the other hand, the number of iterations to reach convergence seems to remain the same for value iteration while there seems to be an increasing trend for policy iteration.

Varying Grid Length

We change the grid length provided to *generate_maze* function to vary the size of the maze while keeping the probabilities of the different square types forming the same. The results from varying wall probability can be seen in Figure 5.3 and 5.4.

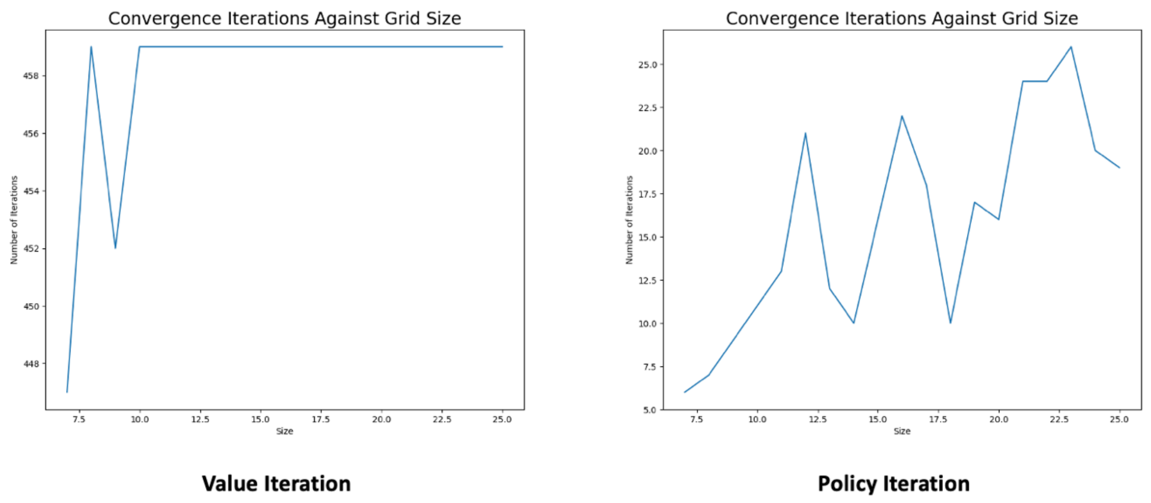


Figure 5.3. Convergence Iterations Against Grid Size

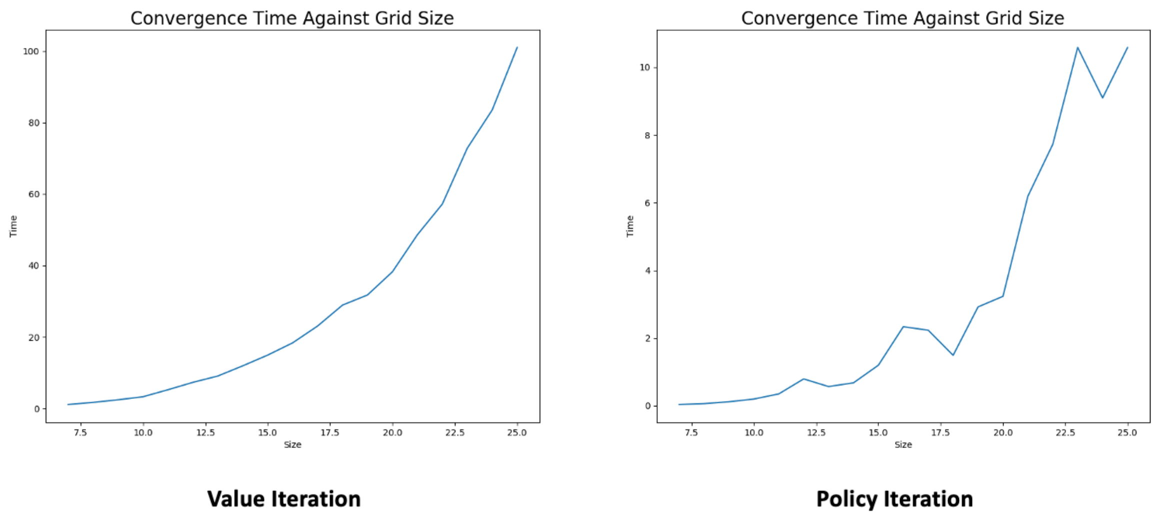


Figure 5.4. Convergence Time Against Grid Size

From Figure 5.3, we can see the number of iterations to reach convergence seems to remain the same for value iteration while there seems to be an increasing trend for policy iteration. On the other hand, in terms of convergence time in Figure 5.4, we can see that there is an increase in convergence time as the grid size (and in turn, number of states) increases. This is the same for both policy and value iteration. However, in terms of absolute time, value iteration takes a much longer time as compared to policy iteration. For instance, with a grid size of 25, the time taken for value iteration is approximately 100s while the time taken for policy iteration is approximately 10s.

For both experiments with number of states, it is more important to look at the effects with regards to convergence time as the trend is clearer. In both experiments, we can see that there is a positive correlation between convergence time and number of states. Additionally, the experiments also show that policy iteration seems better able to scale to handle larger number of states, as compared to value iteration.

<u>Question: How Does Increasing Complexity Affect Convergence</u>

In addition to changing the number of states, we can increase the complexity by varying the number of special squares (rewards and penalties) in the system. This is done by varying the probabilities of the special squares while keeping total grid size the same. Among the special squares, we keep the probability of reward and penalty squares equal. The results from varying wall probability can be seen in Figure 5.5 and 5.6.

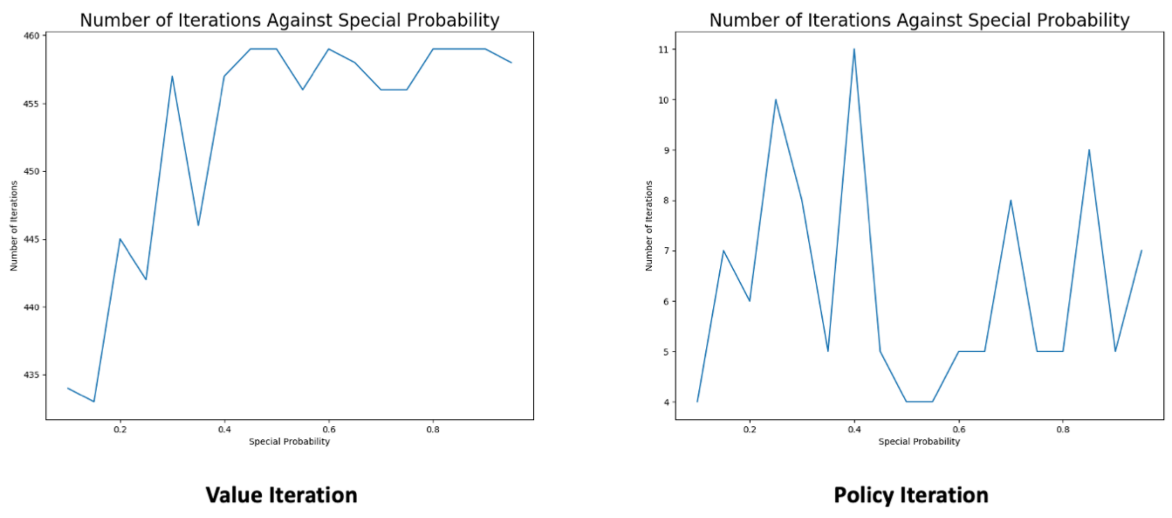


Figure 5.5. Convergence Iterations Against Special Probability

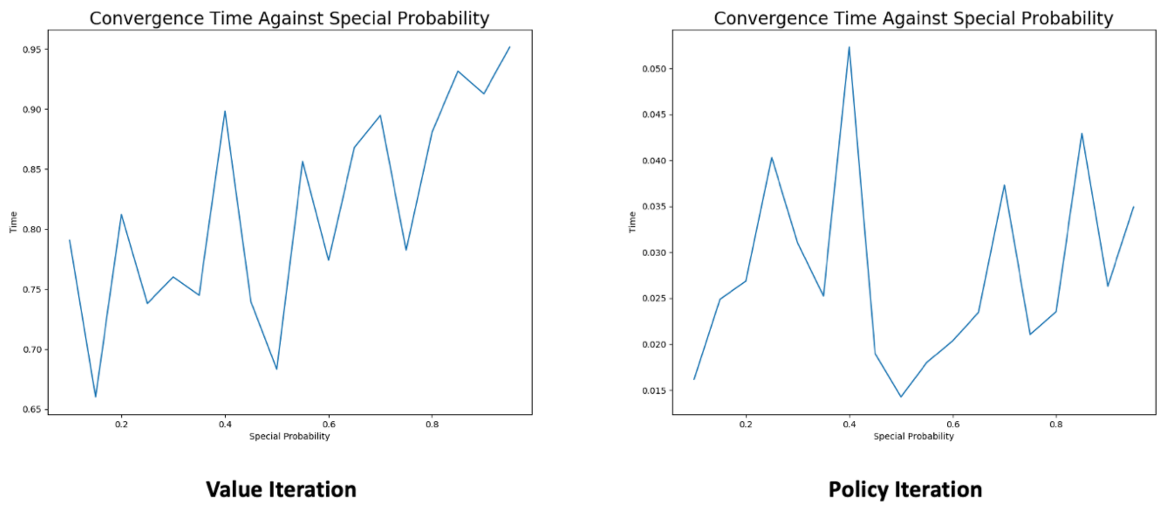


Figure 5.6. Convergence Time Against Special Probability

From Figure 5.5, we can see that there seems to be a positive correlation with number convergence iterations and the probability of special squares for value iteration. On the other hand, from Figure 5.6, there does not seem to be a clear relationship between convergence and probability of special squares. For both algorithms, convergence was reached even as the probability of special squares approach 1. Thus, if we take special squares as an indicator of environment complexity, we can conclude that the algorithms are still able to learn the right policy even an extremely complex environment.

<u>5. Source Code</u>

The source code comprises of the following python files:

1. *main.py*
   The file servers as the manager that will handle the logic required.

2. *constants.py*
   This file holds the constant values which will be used by other files and functions. This includes the maze grid, the discount factor and reward map.

3. *MDP.py*
   This file holds the class definition of the abstract MDP class.

4. *Maze.py*
   This file holds the class definition of the Maze class. The Maze will inherit from the MDP class and it is coded according to the MDP defined in the introduction.

5. *valueiteration.py*
   This file holds the implementation of the value iteration algorithm as described in Section 2.

6. *policyiteration.py*
   This file holds the implementation of the policy iteration algorithm as described in Section 3.

7. *bonus_functions.py*
   This file holds the implementation of the bonus functions to solve the bonus questions as described in Section 4.

8. *visualisations.py*
   This file holds the helper functions to produce the visualisations required.

6. Conclusion

In this assignment, we implemented both value iteration and policy iteration to find the optimal policy in the given MDP environment. As we use these algorithms in the bonus questions, we can see the effects of varying number of states and complexity on the convergence of these algorithms. One insight we can take away is that both algorithms are effective in converging to a policy even in complex environments. Another insight is that the policy iteration algorithm seems better able to scale with increasing state size of the MDP.