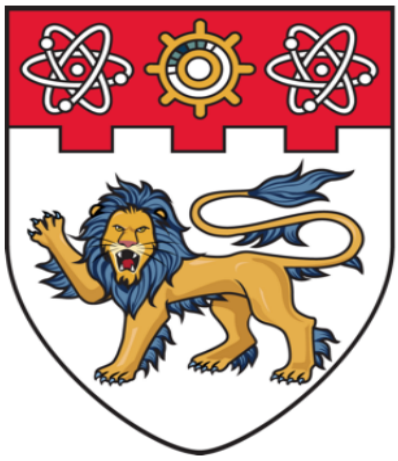


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Intelligent Agents
CZ4046

Assignment 1: Agent Decision Making

Ameeshi Gupta (U2023994H)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

Table of Contents

Table of Contents

1. Overview

1.1. Theory

Markov decision process (MDP) is defined as a stochastic decision-making process that uses a mathematical framework to model the decision-making of a dynamic system in scenarios where the results are either random or controlled by a decision maker

A policy is a mapping from states to actions. Finding an optimal policy leads to generating the maximum reward.

Given an MDP environment, we can use dynamic programming algorithms to compute optimal policies, which lead to the highest possible sum of future rewards at each state.

Dynamic programming algorithms work on the assumption that we have a perfect model of the environment's Markov's Decision Process (MDP). So, we're able to use a one-step look-ahead approach and compute rewards for all possible actions.

1.2. Project Description

In the assignment, we are given a static 6*6 grid-world environment with no terminal states. Each state has a reward (or a penalty) associated with it, and some states are walls which our agent can't cross. This means that if the move would make the agent walk into a wall, the agent stays in the same place as before.

We also have a starting state. There are no terminal states i.e. the agent's state sequence is infinite.

1.3. Tasks

The goal of this assignment is to find the optimal value for each state, as well as the optimal policy for the environment using value iteration and policy iteration methods.

1.4. The Environment

The environment is a 6x6 grid with the following features:
Number of Wall tiles: 5 wall tiles
Number of Reward tiles (green squares): 6 (a reward of +1)
Number of Penalty tiles (brown squares: 5 (a penalty of -1)
Remaining (white squares): 20 tiles (a penalty of -0.04)

The reward (or penalty) of a particular state is awarded when an agent transitions into that state. The agent can move in the following directions: up, down, right, left.

This environment is a Markov Decision Process (MDP) with the following transition model—if the agent intends to move in a particular direction, there is a 0.8 chance that it moves in that direction, and a 0.1 chance each that it moves in the direction either to the left or to the right of the intended direction. Moreover, there are no terminal states and the agent's action sequence can be infinite

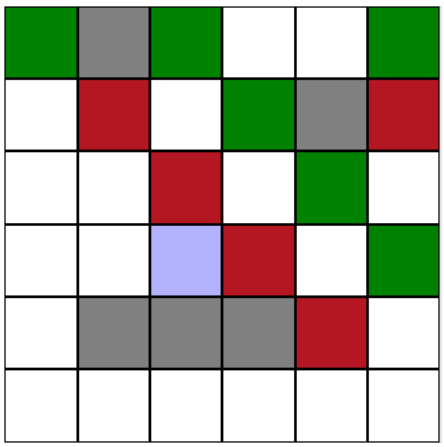
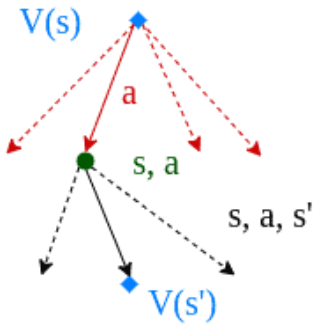


Figure 1: The Maze environment

2. Value Iteration

2.1. How it works

It is a method for finding the optimal value function by solving the **Bellman equations** iteratively. It uses the concept of dynamic programming to maintain a value function that approximates the optimal value function , iteratively improving until it converges to (or close to it).



We start with a random value function. At each step, we update it.

Input: MDP $M = \langle S, s_0, A, P_a(s' \mid s), r(s, a, s') \rangle$
Output: Value function V

Set V to arbitrary value function; e.g., $V(s) = 0$ for all s

Repeat
    $\Delta \leftarrow 0$
    For each $s \in S$
        $\underbrace{V'(s) \leftarrow \max_{a \in A(s)} \sum_{s' \in S} P_a(s' \mid s) \left[r(s, a, s') + \gamma V(s') \right]}_{\text{Bellman equation}}$
        $\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$
    $V \leftarrow V'$
Until $\Delta \leq \theta$

where
s is the current state
s' is the next state
$V(s)$ is the value of state s
$P(s'|s, a)$ is the probability of going from state s to state s' upon executing action a
$r(s, a, s')$ is the reward of going from state s to state s' upon executing action a
γ is the discount factor

Value iteration converges to the optimal policy as iterations continue: $V \rightarrow V^*$ *as* $i \rightarrow \infty$, where i is the number of iterations. So, given an infinite amount of iterations, it will be optimal.

Value iteration converges to the optimal value function V^* asymptotically, but in practice, the algorithm is stopped when the *residual* Δ reaches some pre-determined threshold – that is, when the largest change in the values between iterations is "small enough".

A policy can now be easily defined: in a state s, given V, choose the action with the highest expected reward using policy extraction. The loss of the result greedy policy is bound by $\frac{2\gamma\Delta}{1-\gamma}$.

2.2. Implementation

The **utilities** array is the size of the **grid_size**. All the values are initially initialised to 0. The **policy** array has the same size as the utilities array. It is initialized with placeholders ' ' for non-wall cells. This array represents the policy, i.e., the recommended action to take in each state.

The **is_valid_state** function checks if a given state **(x, y)** is valid, meaning it's within the grid boundaries and not a wall.

Below is the python implementation:

```
# Initialize utilities and policy
utilities = np.zeros(grid_size)
policy = np.full(grid_size, ' ', dtype='<U5')  # Placeholder for non-wall cells

# Function to check if a state is valid (not a wall and within grid)
def is_valid_state(x, y):
    return (0 <= x < grid_size[0]) and (0 <= y < grid_size[1]) and (x, y) not in walls
```

Figure 2: initialisation step

Next, a function called **expected_utility** is defined which calculates the expected utility by considering the probabilities of moving in the specified direction (**action**) and the neighbouring cells. It returns the sum of the expected utilities for all neighbouring cells.

```python
# Calculate expected utility of performing an action from a state
def expected_utility(action, x, y):
    if action == 'up':
        forward = (x - 1, y)
        side1 = (x, y - 1)
        side2 = (x, y + 1)
    elif action == 'down':
        forward = (x + 1, y)
        side1 = (x, y - 1)
        side2 = (x, y + 1)
    elif action == 'left':
        forward = (x, y - 1)
        side1 = (x - 1, y)
        side2 = (x + 1, y)
    elif action == 'right':
        forward = (x, y + 1)
        side1 = (x - 1, y)
        side2 = (x + 1, y)

    utility_sum = 0
    for nx, ny in [forward, side1, side2]:
        if is_valid_state(nx, ny):
            p = prob_forward if (nx, ny) == forward else prob_side
            utility_sum += p * utilities[nx, ny]
        else:
            utility_sum += (prob_forward if (nx, ny) == forward else prob_side) * utilities[x, y]
    return utility_sum
```

Figure 3: function expected_utilities

The purpose of these implementations is to set up the initial environment for a grid-based problem solving approach.
Following this, **value_iteration** function is implemented. The python implementation of the function is below:

```python
    # Value iteration
    iteration = 0
    utilities_history = []  # Track utilities for plotting
    while True:
        delta = 0
        new_utilities = np.copy(utilities)
        for x in range(grid_size[0]):
            for y in range(grid_size[1]):
                if is_valid_state(x, y):
                    max_utility = -np.inf
                    for action in actions:
                        utility = rewards[x, y] + gamma * expected_utility(action, x, y)
                        if utility > max_utility:
                            max_utility = utility
                            policy[x, y] = action
                    delta = max(delta, abs(max_utility - utilities[x, y]))
                    new_utilities[x, y] = max_utility
        utilities = new_utilities
        utilities_history.append(utilities.copy())
        iteration += 1
        if delta < 0.01*(1-gamma)/gamma:  #Convergence criterion
            break
```

Figure 4: function defination for value iteration

The function iteratively updates the utility values for each state based on the Bellman equation, selecting the action that maximizes the expected utility. The algorithm terminates when the maximum change in utility values falls below a specified threshold, indicating convergence to the optimal solution. The iteration variable is initially set as 0 and it keeps incrementing the value by 1 till convergence is reached.

To achieve the optimal policy, we chose a **delta of 0. 01* (1-gamma)/gamma**—once the change in the value of each state becomes smaller than that value, we consider our algorithm to have converged.

2.3. Plot of optimal policy

The algorithm converged after 917 iterations with the following optimal policy –

```
Optimal Policy:
[['up' 'Wall' 'left' 'left' 'left' 'up']
 ['up' 'left' 'left' 'left' 'Wall' 'up']
 ['up' 'left' 'left' 'up' 'left' 'left']
 ['up' 'left' 'left' 'up' 'up' 'up']
 ['up' 'Wall' 'Wall' 'Wall' 'up' 'up']
 ['up' 'left' 'left' 'left' 'up' 'up']]
```

Figure 5: Optimal Policy obtained by Value Iteration after 917 iterations

2.4. Utilities of all states

The following value function V(s) value was obtained for each state after convergence:

```
Utilities of states:
[['99.99' 'Wall' '95.04' '93.87' '92.64' '93.32']
 ['98.38' '95.87' '94.54' '94.39' 'Wall' '90.91']
 ['96.94' '95.58' '93.28' '93.17' '93.09' '91.78']
 ['95.54' '94.44' '93.22' '91.11' '91.80' '91.88']
 ['94.30' 'Wall' 'Wall' 'Wall' '89.54' '90.56']
 ['92.93' '91.72' '90.53' '89.35' '88.56' '89.29']]
```

Figure 6: Utilities of the states obtained by Value Iteration function

2.5. Plot of utility estimates

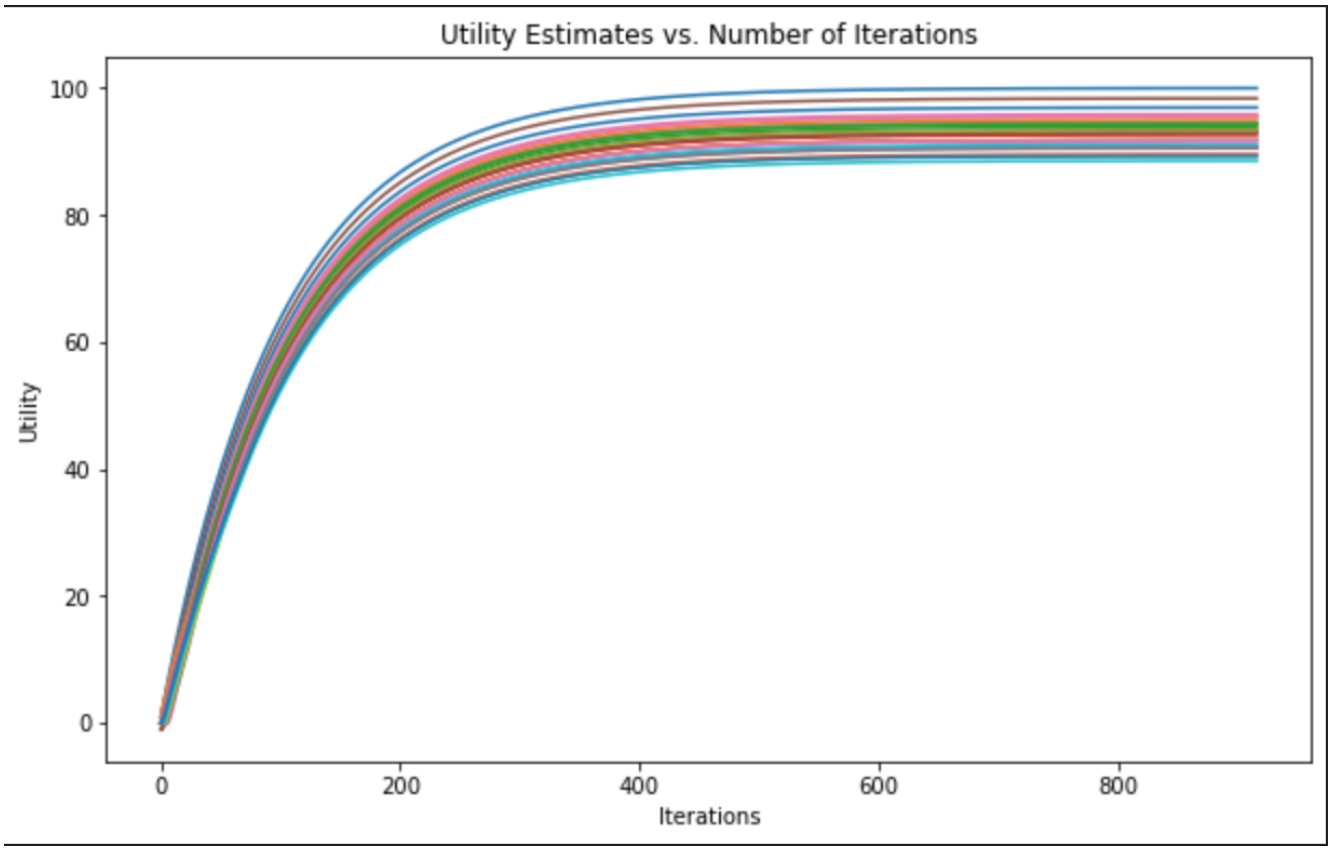
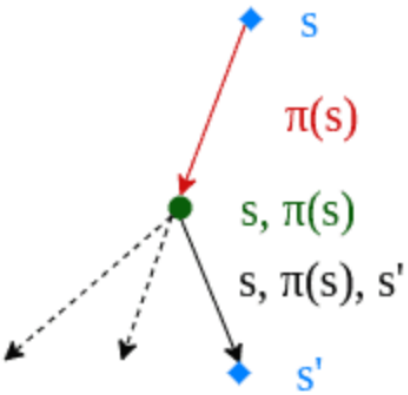


Figure 7: Plot of utility estimates as a function of the number of iterations

3. Policy Iteration

3.1. How it works

In policy iteration, we start by choosing an arbitrary policy π. The function then alternates between the following two steps—policy evaluation and policy improvement. We iteratively evaluate and improve the policy until convergence. In policy evaluation, the algorithm calculates the value function



Policy evaluation can be characterised as $V^{\pi}(s)$ as defined by the following equation:

$$V^{\pi}(s) = \sum_{s' \in S} P_{\pi(s)}(s' \mid s)\,[r(s,a,s') + \gamma\, V^{\pi}(s')]$$

where $V^{\pi}(s) = 0$ for terminal states.

The implementation is the same as value iteration except that we use the policy evaluation equation instead of the Bellman equation in **policy_iteration.**

Algorithm for Policy evaluation:

Input: π the policy for evaluation, V^{π} value function, and MDP $M = \langle S, s_0, A, P_a(s' \mid s), r(s,a,s') \rangle$
Output: Value function V^{π}

Repeat
  $\Delta \leftarrow 0$
  For each $s \in S$
    $\underbrace{V'^{\pi}(s) \leftarrow \sum_{s' \in S} P_{\pi(s)}(s' \mid s)\,[r(s,a,s') + \gamma\, V^{\pi}(s')]}_{\text{Policy evaluation equation}}$
    $\Delta \leftarrow \max(\Delta, |V'^{\pi}(s) - V^{\pi}(s)|)$
  $V^{\pi} \leftarrow V'^{\pi}$
Until $\Delta \leq \theta$

Algorithm for Policy Iteration:

Input: MDP $M = \langle S, s_0, A, P_a(s' \mid s), r(s, a, s') \rangle$
Output: Policy π

Set V^π to arbitrary value function; e.g., $V^\pi(s) = 0$ for all s.

Set π to arbitrary policy; e.g. $\pi(s) = a$ for all s, where $a \in A$ is an arbitrary action.

Repeat
    Compute $V^\pi(s)$ for all s using policy evaluation
    For each $s \in S$
        $\pi(s) \leftarrow \text{argmax}_{a \in A(s)} Q^\pi(s, a)$
Until π does not change

The policy iteration function computes an optimal π by performing a sequence of interleaved policy evaluations and improvements.

3.2. Implementation

Similar to the value iteration, we first initialise the utilities values to 0 and the policy values are initialised with placeholders **' '** for non-wall cells.

We then define a function called **expected_utilities** which calculates the expected utility by considering the probabilities of moving in the specified direction (**action**) and the neighbouring cells.

For the policy iteration method, we also define a variable called **initial_choice.** This array initializes a policy array with random actions for each position in a grid, where actions is a list of possible actions and grid_size defines the shape of the grid. It then iterates over a list of walls (positions in the grid that are walls) and sets the policy for these wall positions to None, indicating that no action is to be taken for wall states.

```
initial_policy = np.random.choice(actions, size=grid_size)
for wall in walls:
    initial_policy[wall] = None  # No policy for wall states
```

Figure 8: code snippet for initial_choice array

After the implementation of the initial steps as mentioned, **policy_evaluation** and **policy_improvement** functions are defined.

The **policy evaluation** function iteratively updates the utilities based on the expected return from following the given policy, until the change in utility values (delta) is smaller than a specified threshold, which ensures convergence to a stable set of utilities under the given policy.
The condition for breaking the loop, **delta < 0.01*(1-gamma)/gamma**, uses a factor of the discount rate **gamma** to determine when the utility estimates have sufficiently converged.

The **policy improvement** function iteratively updates a given policy based on the current utility values of states in a grid environment to maximize the expected utility of actions taken from each state. It loops through all states in the grid, skipping states that are not valid. For each valid state, it calculates the expected utility of taking each possible action from that state, given the current utility values. The action with the highest expected utility is then set as the new policy for that state. If the chosen action for any state is different from the action specified by the old policy, the **policy_stable** flag is set to **False**, indicating that the policy has changed.
The function returns the updated policy and a boolean indicating whether the policy was stable (i.e., no changes were made to the policy during the iteration), which is a key part of determining convergence in policy iteration algorithms.

Below are the implementations of the two functions in python:

```python
# Correct the policy_evaluation function to pass the utilities array
def policy_evaluation(policy, utilities):
    while True:
        delta = 0
        new_utilities = utilities.copy()
        for x in range(grid_size[0]):
            for y in range(grid_size[1]):
                if not is_valid_state(x, y):
                    continue
                utility = rewards[x, y] + gamma * expected_utility(policy[x, y], x, y, utilities)
                delta = max(delta, abs(utility - utilities[x, y]))
                new_utilities[x, y] = utility
        if delta < 0.01*(1-gamma)/gamma:
            break
        utilities = new_utilities
    return utilities
```

Figure 9: Implementation of policy_evaluation function

```python
# Correct the policy_improvement function to pass the utilities array
def policy_improvement(utilities, policy):
    policy_stable = True
    for x in range(grid_size[0]):
        for y in range(grid_size[1]):
            if not is_valid_state(x, y):
                continue
            old_action = policy[x, y]
            max_utility = -np.inf
            for action in actions:
                utility = rewards[x, y] + gamma * expected_utility(action, x, y, utilities)
                if utility > max_utility:
                    max_utility = utility
                    policy[x, y] = action
            if old_action != policy[x, y]:
                policy_stable = False
    return policy, policy_stable
```

Figure 10: Implementation of function policy_improvement

3.3. Plot of optimal policy

For policy iteration, the algorithm converged after 6 iterations.
The optimal policy obtained after convergence is –

```
Optimal Policy:
[['up' 'Wall' 'left' 'left' 'left' 'up']
 ['up' 'left' 'left' 'left' 'Wall' 'up']
 ['up' 'left' 'left' 'up' 'left' 'left']
 ['up' 'left' 'left' 'up' 'up' 'up']
 ['up' 'Wall' 'Wall' 'Wall' 'up' 'up']
 ['up' 'left' 'left' 'left' 'up' 'up']]
```

Figure 11: optimal policy from policy iteration

3.4. Utilities of all states

The value of the utilities of all the states obtained after 6 iterations are as follows –

```
Utilities of states:
[['100.00' 'Wall' '95.05' '93.87' '92.65' '93.33']
 ['98.39' '95.88' '94.54' '94.40' 'Wall' '90.92']
 ['96.95' '95.59' '93.29' '93.18' '93.10' '91.79']
 ['95.55' '94.45' '93.23' '91.12' '91.81' '91.89']
 ['94.31' 'Wall' 'Wall' 'Wall' '89.55' '90.57']
 ['92.94' '91.73' '90.54' '89.36' '88.57' '89.30']]
```

Figure 12: Utilities of states from policy iteration

3.5. Plot of utility estimates

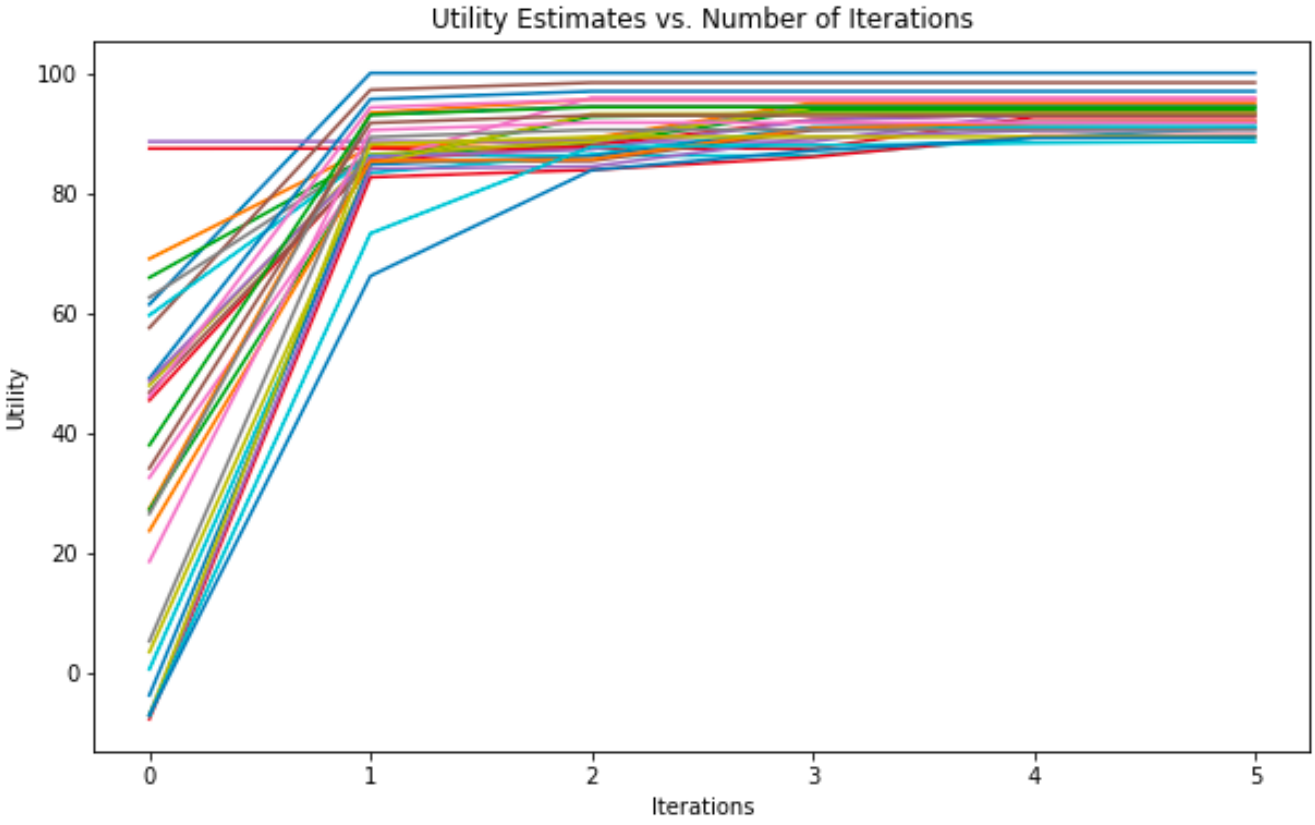


Figure 13: Plot of utility estimates as a function of number of iterations

4. Bonus Question

To design a more complicated maze environment, I define a function called **initialize_random_maze** which takes the grid size and randomly places the walls, reward squares and the penalty squares in the maze.

I have set the following probabilities for designing the maze environment:
Wall probability = 0.1
Reward tile probability = 0.02 with a reward of +1
Penalty tile probability = 0.02 with a penalty of -1
The remaining tiles are white with a penalty of 0.04.

I use multiple grid sizes for this task. The table below shows the grid sizes and the number of iterations the functions take to converge.

Grid_size	Number of Iterations	
	Value Iteration	Policy Iteration
6*6	891	4
30*30	893	10
500*500	786	9

Plot for 6*6 random maze –

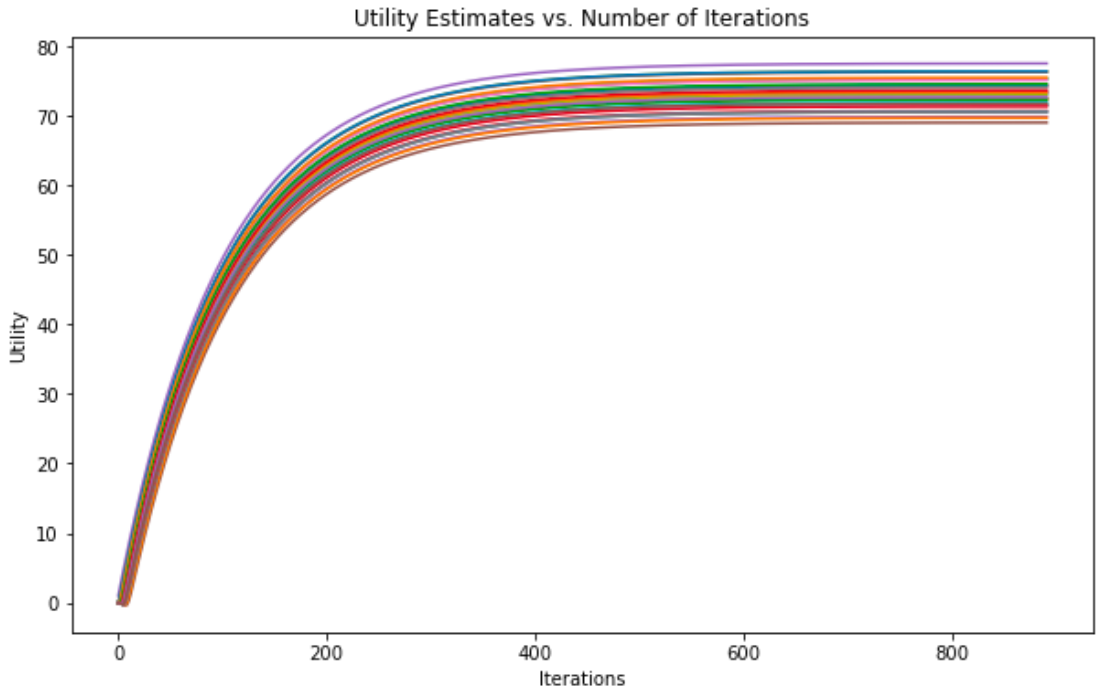


Figure 14: Plot for value iteration

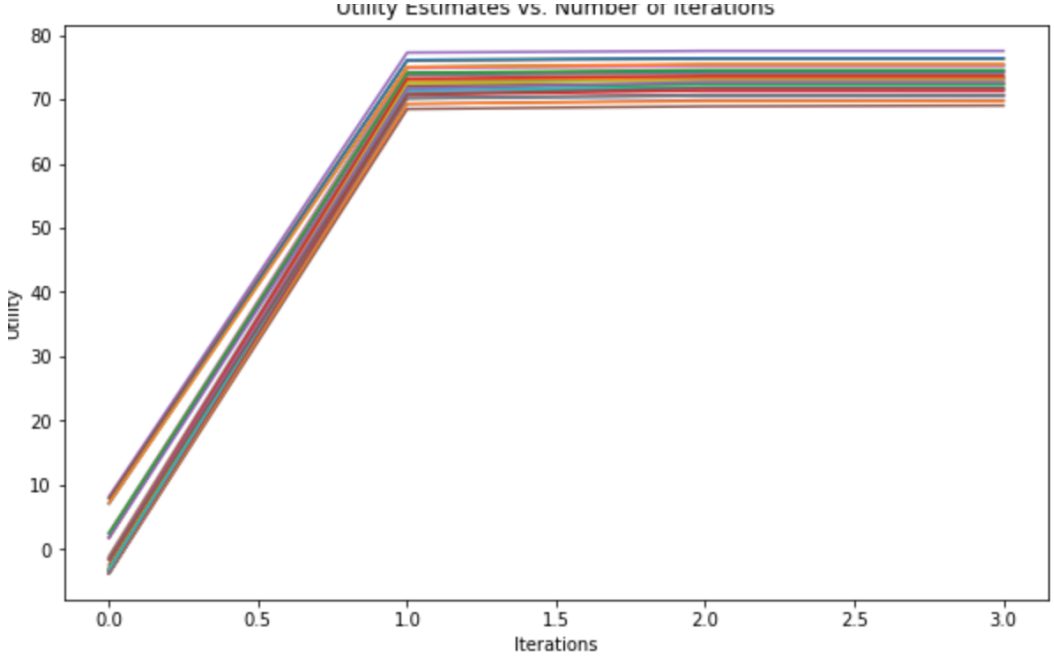


Figure 15: plot for policy iteration

Plot for 30*30 random maze –

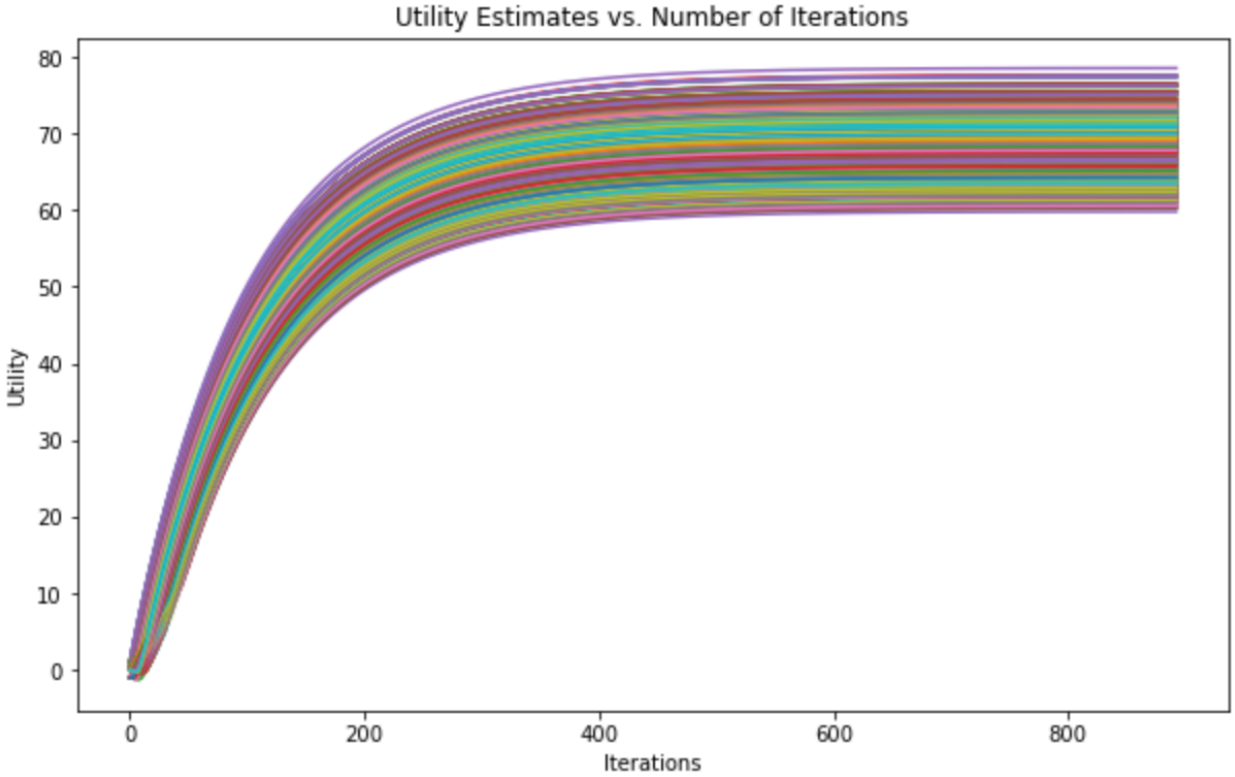


Figure 16: plot of value iteration

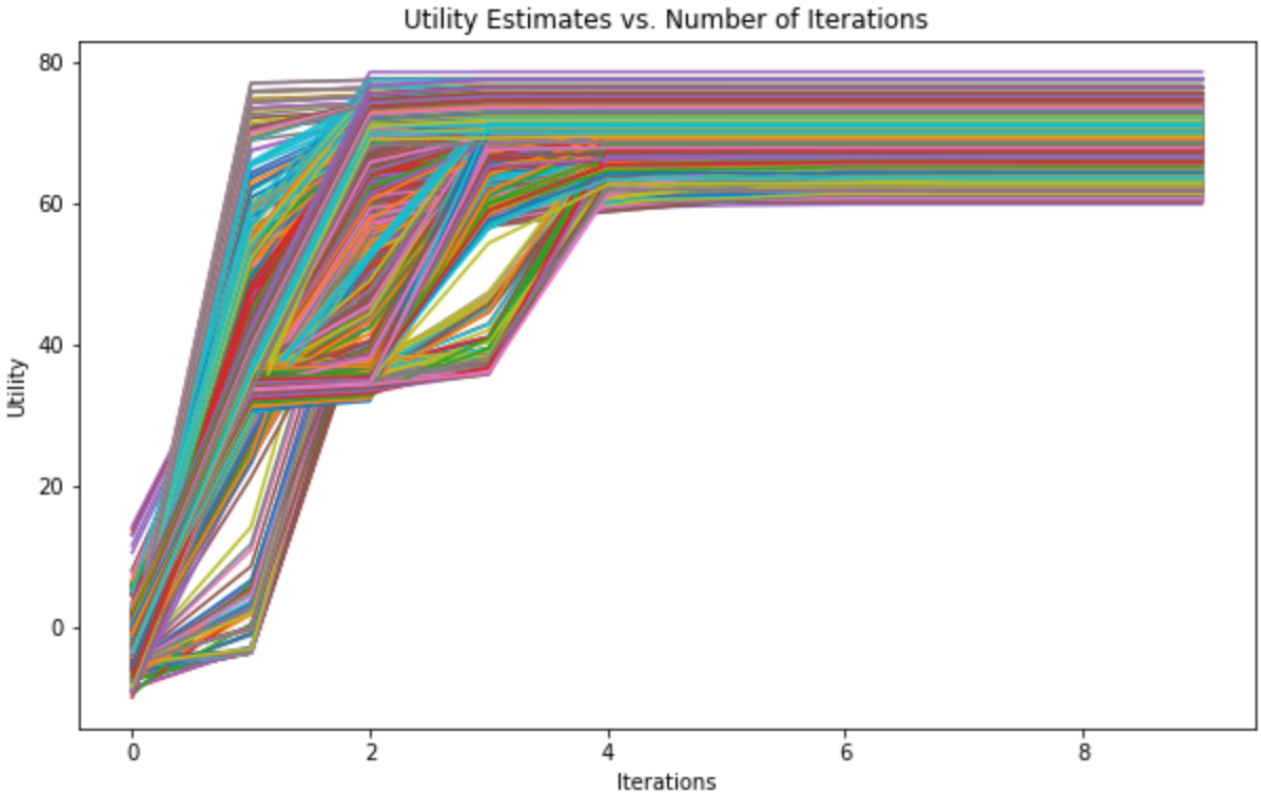


Figure 17: plot for policy iteration

Plot for 500*500 random maze –

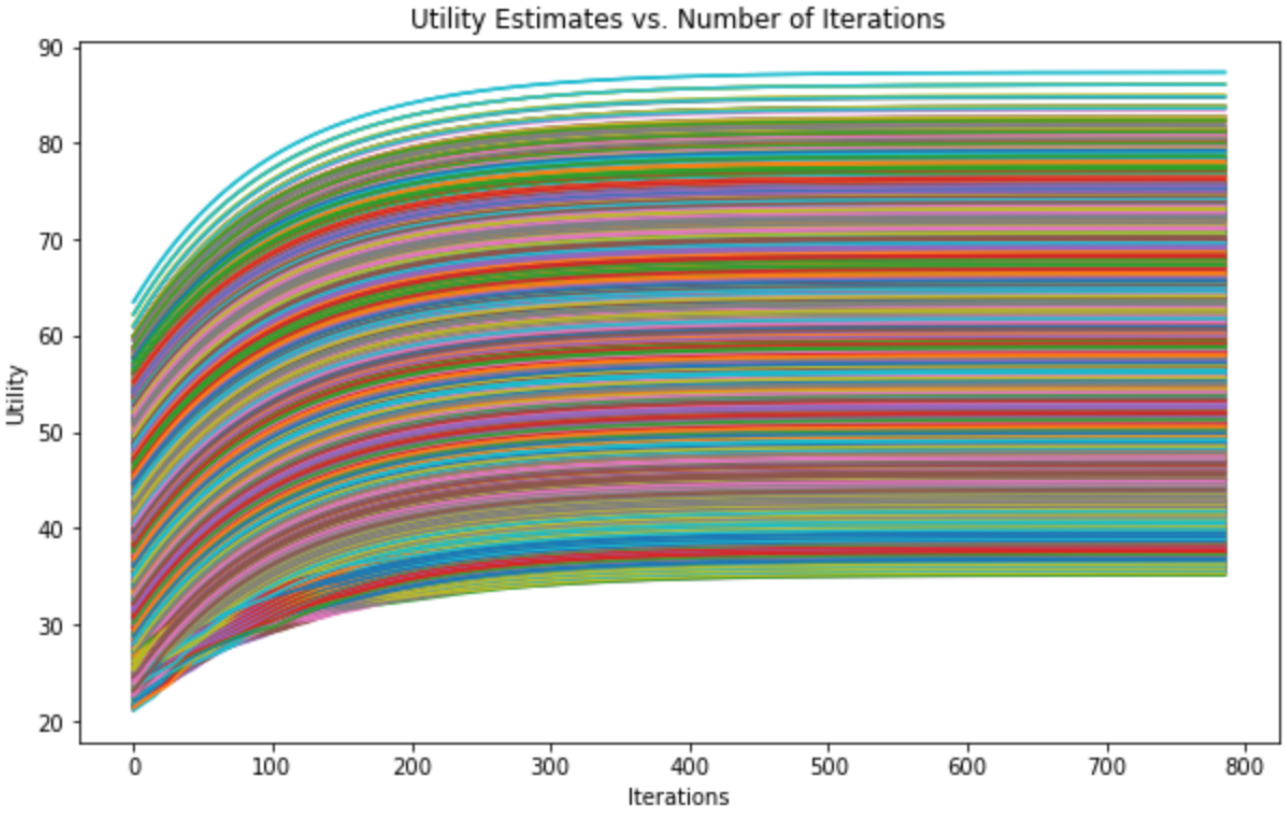


Figure 18: plot for value iteration

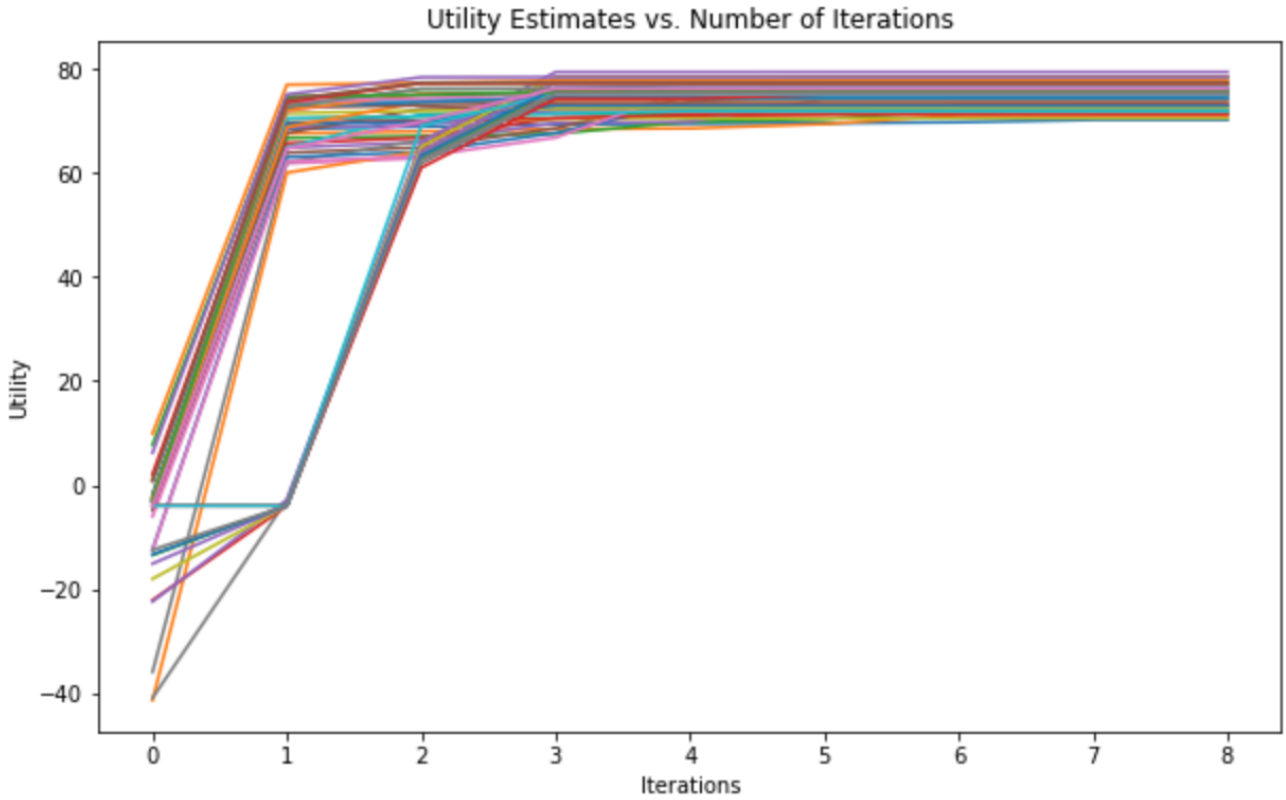


Figure 19: plot for policy iteration

It is observed that as the grid size increases, the iterations taken to converge for value iteration and policy iteration increases.

The complexity of the model increases significantly with the increase in the number of states. At size = 500, the test runs take a significant amount of time (> 10 minutes) to complete. It shows that the computational time is getting infeasible with the increase in size.

With this, it can be concluded that for the implemented algorithms, they can feasibly learn the right policy for the arenas with a complexity <= 500 X 500.