

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

CZ4046: Intelligent Agents

Name	Nur Dilah Binte Zaini
Matriculation Number	U2022478K
Coursework	Assignment 1

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY**

Table of contents

Guide to executing Source Code	3
Download the Selected IDE	3
Steps in unpacking my project:	3
Steps in setting up the environment	4
Source Code	6
Initialization of the Maze Environment	9
Implementation of Value Iteration	10
Description	10
Plot of the Optimal Policy	14
Utility of all states	15
Figure 7: State Utility Values in a grid format for Value Iteration	16
Plot of Utility estimates as a function of the number of iterations	17
Implementation of Policy Iteration	18
Description	18
Plot of the Optimal Policy	22
Utility of all states	23
Plot of Utility estimates as a function of the number of iterations	25
Bonus Question	26
Complicated Maze Environment	26
1. Increase the number of states by increasing the scale of the environment	26
Findings with this experiment:	27
2. Generate different number of the states (Reward States, Penalty States) for the Maze environment	27
Findings with this experiment:	31
Findings from this experiment	34
Conclusion	35

Guide to executing Source Code

Download the Selected IDE

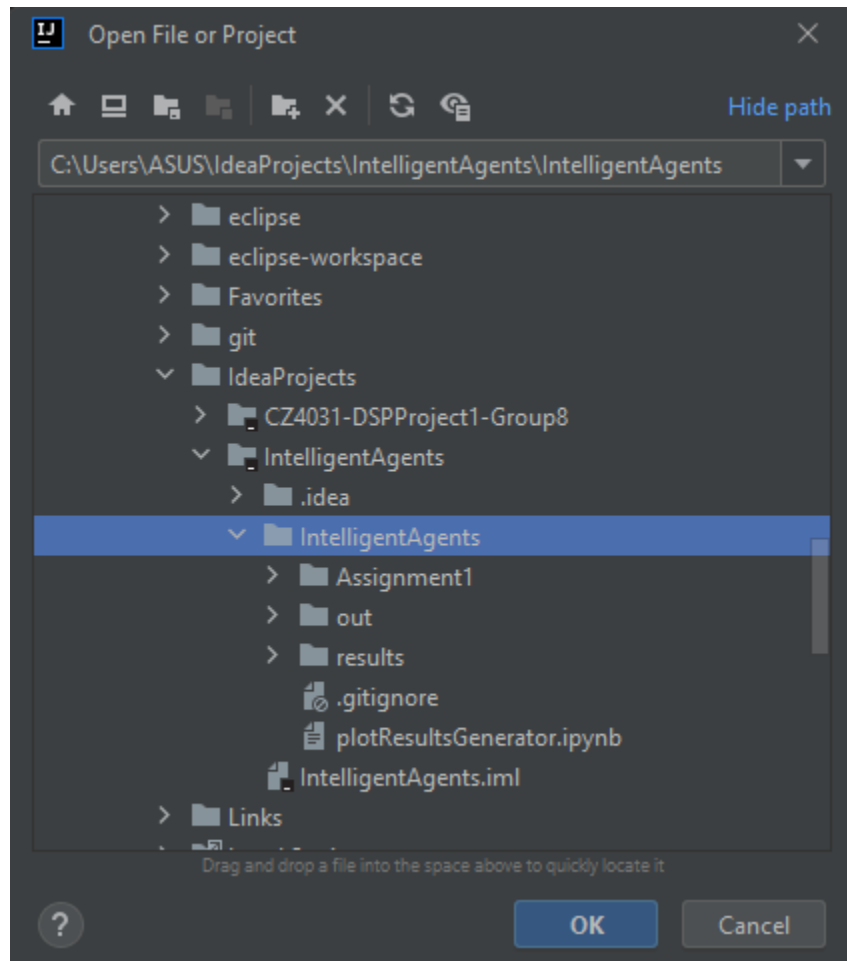
Download IntelliJ Community Edition which is the platform to execute the source code
<https://www.jetbrains.com/idea/download/#section=windows>

Steps in unpacking my project:

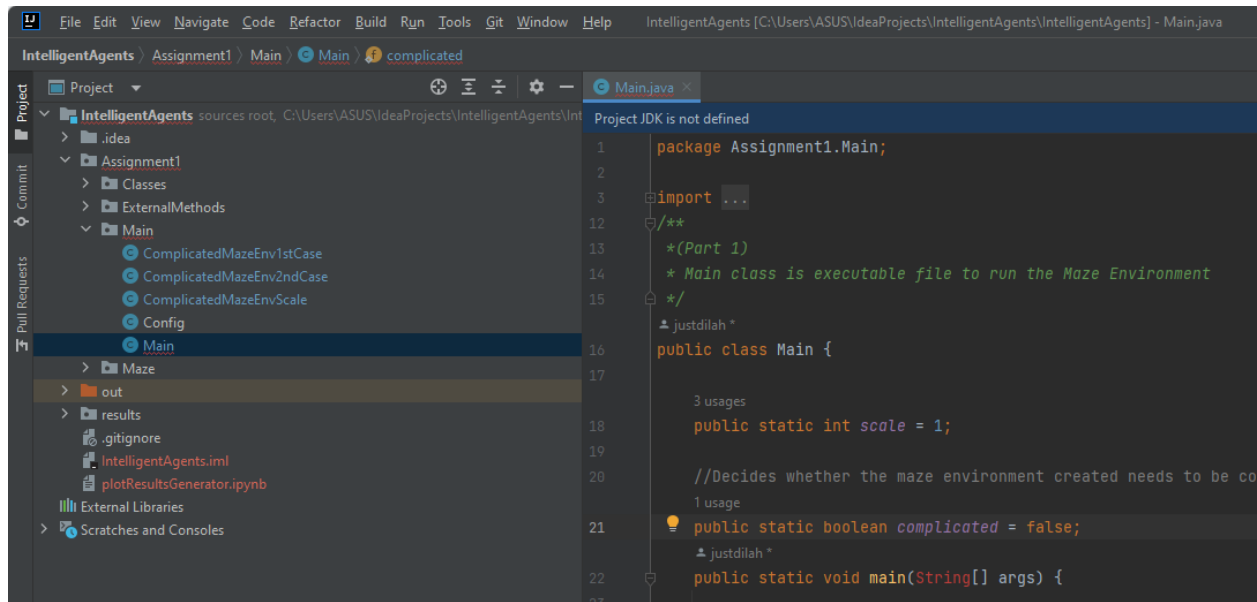
1. Download the Zaini_NurDilah_U2022478K.zip file that was submitted on NTULearn, and unzip it.
2. Inside the unzipped folder, the source code for our project is named IntelligentAgents.zip, put it into your IdeaProjects directory.
Eg: C:\Users\user\IdeaProjects

Steps in setting up the environment

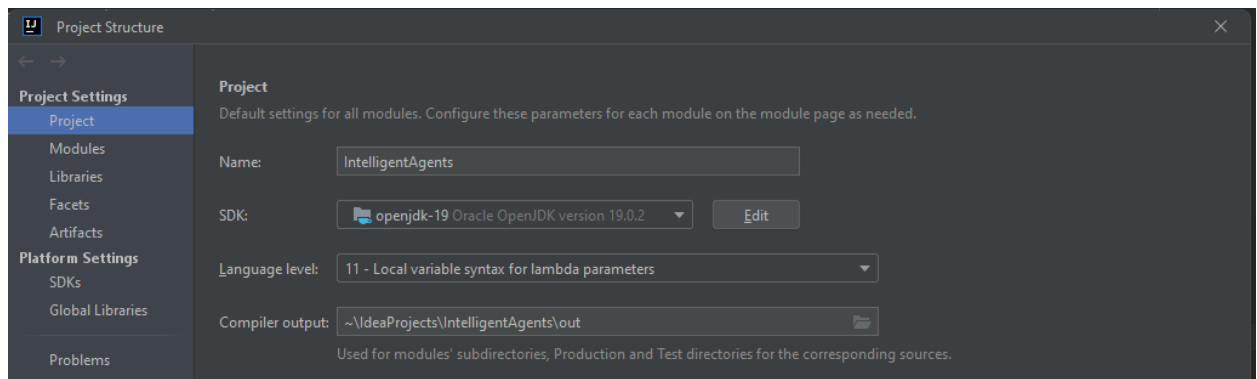
1. Unzip the IntelligentAgents.zip file in your IdeaProjects directory
2. Open IntelliJ and do the following steps:
 - a. Click on the open button which is found in the top-right-hand corner of the UI.
 - b. Search for your IdeasProjects directory and select the unzipped IntelligentAgents folder.




3. Once the project has been successfully opened in IntelliJ and you will navigate to Main.java, the error shown below will pop up.



4. You will have to define your Project JDK, go to File > Project Structure which is shown on the top navigation bar, and make the following changes to the SDK and Language Level:



5. Once you have applied the changes, execute the code on the IntelliJ interface by clicking on the  on the top navigation bar.

Take Note: There are 4 executable files that you will be running which are:

- Main.java: Executable file for the maze environment built for **Part 1** of the assignment
- ComplicatedMazeEnvScale.java: Executable file for the maze environment built for **Part 2** of the assignment (Scale)
- ComplicatedMazeEnv1stCase.java: Executable file for maze environment built for **Part 2** of the assignment (1st Case: Each category of states (Penalty, Reward, Wall) has an equal number of states)
- ComplicatedMazeEnv2ndCase.java: Executable file for the maze environment built for **Part 2** of the assignment (2nd Case: Modify the number of Penalty states and Rewards states)

Source Code

The source code is broken into different classes. The hierarchy of the source code is shown as follows:

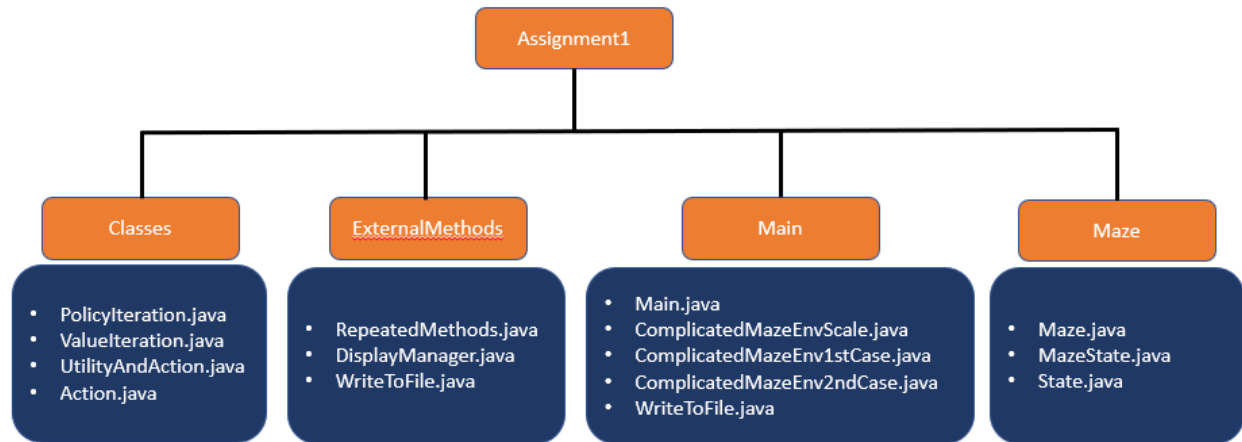


Figure 1: Hierarchy of the source code

- Package Classes

Class	Descriptions
Action.java	Contains the transition probabilities and the actions that are available for the agents to use
PolicyIteration.java	Used to build the modified Policy Iteration algorithm
ValueIteration.java:	Used to build the Value Iteration algorithm
UtilityAndAction.java:	Holds the utility value and action of the state

- Package ExternalMethods

Class	Descriptions
RepeatedMethods.java	Stores functions that are used in most of the classes
DisplayManager.java	Used to display the maps, actions, and states in a presentable manner
WriteToFile.java:	To store the list of utility estimates in a CSV file which is used to plot utility estimates as a function of iteration

- Package Main

Class	Descriptions
Main.java:	To execute the maze environment built for Part 1 of the assignment
ComplicatedMazeEnvScale.java	To execute the maze environment built for Part 2 of the assignment (Scale)
ComplicatedMazeEnv1stCase.java	To execute the maze environment built for Part 2 of the assignment (1st Case)
ComplicatedMazeEnv2ndCase.java	To execute the maze environment built for Part 2 of the assignment (2nd Case)
Config.java	Stores the constant variables

- Package Maze

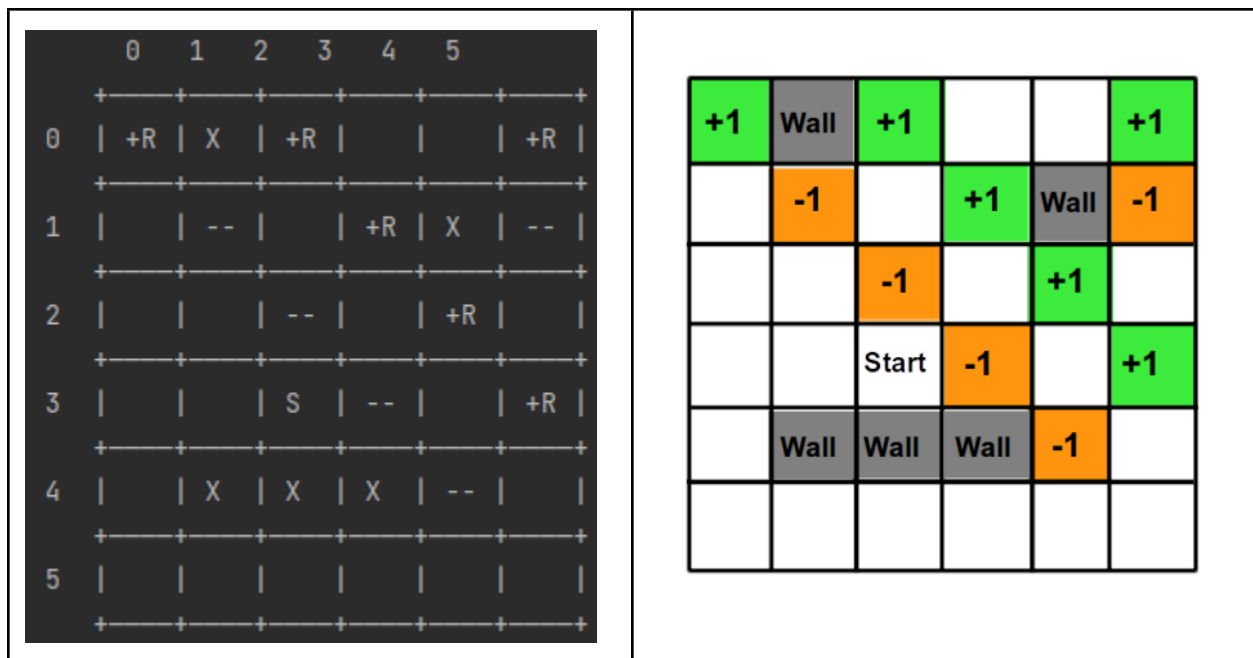
Class	Descriptions
Maze.java:	Creates the maze environment

MazeState.java	Stores the state type, reward value, the x, and y coordinates
State.java	Stores the states (penalty, empty, reward, wall, start)

Initialization of the Maze Environment

The maze is represented in a 2-D array. For each coordinate, it holds a state. For example, the coordinate at (3,2) holds the start state. The reward for each state is as follows:

- Start State: 0.00
- Empty State: -0.04 [white]
- Reward State: +1.00 [green]
- Penalty State: -1.00 [orange]
- Wall State: 0.00 [grey]



Transition model is as follows:

- The intended outcome occurs with a probability of 0.8
- The clockwise and the anti-clockwise direction of the intended outcome occurs with a probability of 0.1

Implementation of Value Iteration

Description

The value iteration function is defined in ValueIteration.java. The Bellman equation is used for value iteration:

$$U(s) = R(s) + \gamma \max_a \sum P(s'|s, a)U(s')$$

The Bellman equation is the main basis of the value iteration for solving MDPs. If there are n possible states in the maze environment, each state will have a bellman equation. Hence, there will be n bellman equations. To solve n bellman equations, an iterative approach will be used. Each state is initialized with 0 as the utility and null as the action using <UtilityAndAction> 2-D array to represent those values. For each iteration, a bellman update is used to update each state utility from the utilities of its neighbors.

Breakdown of the Bellman equation

- Calculates the expected utility based on each available action the agent can take:

```
//retrieves the expected utility of the action
1 usage  2 justdillah *
private double getPotentialNextStateUtility(Action action, int row, int col) {
    //Calculate the expected utility of a possible actions
    double intendedUtility = getExpUtilityBasedAction(action, row, col);
    double clockwiseUtility = getExpUtilityBasedAction(action.getClockwiseAction(), row, col);
    double antiClockwiseUtility = getExpUtilityBasedAction(action.getAntiClockwiseAction(), row, col);

    return (INTENDED_PROB * intendedUtility) + (CW_PROB * clockwiseUtility) + (ACW_PROB * antiClockwiseUtility);
}
```

Figure 2: Code snippet for the computation of Expected Utility function

- With the values retrieved from the method above, the maximum expected utility for the next state is selected which is the optimal action that the agent will take

```
1 usage  justdilah
private UtilityAndAction getMaxUtilityAndAction(int row, int col) {
    List<UtilityAndAction> listOfUtilityAndAction = new ArrayList<>();
    //Goes through the loop of Actions (UP,DOWN,LEFT,RIGHT)
    for (Action action : Action.values()) {
        listOfUtilityAndAction.add(new UtilityAndAction(action, getPotentialNextStateUtility(action, row, col)));
    }

    //Retrieve utility of the next state
    UtilityAndAction maxUtilityAndAction = Collections.max(listOfUtilityAndAction);
    return maxUtilityAndAction;
}
```

Figure 3: Code snippet for the selecting of Maximum Expected Utility function

- Bellman update is then executed that is, the new value of the state utility is computed by plugging in the current state utility value with the expected discounted utility of the next state.

```
1 usage  justdilah
public double performBellmanUpdate(int row,int col, UtilityAndAction nextState, double discount){
    return this.Maze[row][col].getStateReward() + (discount * nextState.getUtility());
}
```

Figure 4: Code snippet of Bellman Update function

The bellman update continues until it converges to the optimal value function. To achieve the optimal value function, the maximum change of utility value amongst all the states in that particular iteration falls below convergence criteria.

```

// loop through all the states of the map
for (int row = 0; row < rows; row++) {
    for (int col = 0; col < cols; col++) {
        // Check if MazeState is Wall
        if (!Maze[row][col].isVisitable()) {
            continue;
        }

        //Retrieves the optimal action and utility
        UtilityAndAction currState = getMaxUtilityAndAction(row, col);

        //retrieves the new utility of the current state by using the bellman equation
        newUtility = performBellmanUpdate(row,col,currState,discount);
        currentUtility = this.currentUtilityAndActionArray[row][col].getUtility();

        //sets new utility of the curr state
        currState.setUtility(newUtility);

        //updated utility and action of the state
        this.newUtilityAndActionArray[row][col] = currState;

        //Checks if the difference between the new Utility and current utility larger than the previous difference
        if (Math.abs(newUtility - currentUtility) > maxChangeInUtility) {
            //If yes, updates the value for max change
            maxChangeInUtility = Math.abs(newUtility - currentUtility);
        }
    }
}

```

Figure 5: Code snippet of Value Iteration function

The convergence criteria are calculated: $\text{Epsilon} * ((1 - \text{discount}) / \text{discount})$. Epsilon represents the maximum error allowable for the utility of a given state. To calculate epsilon, a constant variable c is multiplied by the maximum reward value.

I decided to set the value of the variable c as 0.100. Based on the graph shown below, with the discount factor of 0.99, the number of iterations is the lowest for $c = 0.100$ among all the different values of c shown in the reference book and still converges to the correct utilities.

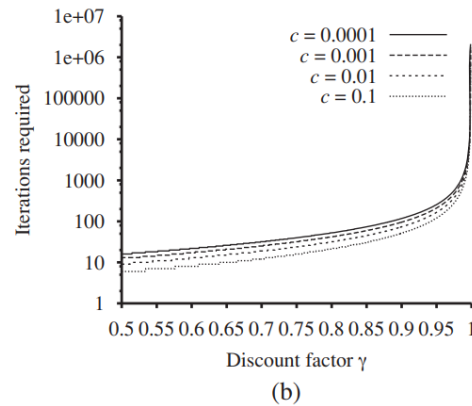


Figure 6: Screenshot is taken from the reference book

Once the function reaches the optimal value function, the $\langle \text{UtilityAndAction} \rangle$ 2-D array will contain all of the state's highest utility with its optimal actions.

Plot of the Optimal Policy

Starting position: (3,2)

After 688 iterations, the maximum change of utility value amongst all the states in that particular iteration falls below convergence criteria which means it has reached the equilibrium.

	0	1	2	3	4	5
0	Λ		<	<	<	Λ
1	Λ	<	<	<		Λ
2	Λ	<	<	Λ	<	<
3	Λ	<	<	<	Λ	Λ
4	Λ				Λ	Λ
5	Λ	<	<	<	Λ	Λ

Utility of all states

```
===== REFERENCE UTILITIES OF STATES =====  
> Coordinates are in (col,row) format. Top-left corner is (0,0) <  
(0,0) : 99.900685  
(0,1) : 98.294047  
(0,2) : 96.849185  
(0,3) : 95.454524  
(0,4) : 94.213205  
(0,5) : 92.838160  
(1,1) : 95.783703  
(1,2) : 95.487113  
(1,3) : 94.353179  
(1,5) : 91.629463  
(2,0) : 94.946586  
(2,1) : 94.446172  
(2,2) : 93.199605  
(2,3) : 93.178119  
(2,5) : 90.435837  
(3,0) : 93.776134  
(3,1) : 94.298927  
(3,2) : 93.077920  
(3,3) : 91.021966  
(3,5) : 89.257095  
(4,0) : 92.555742  
(4,2) : 93.004064  
(4,3) : 91.716586  
(4,4) : 89.450524  
(4,5) : 88.471016  
(5,0) : 93.229590  
(5,1) : 90.819005  
  
(5,2) : 91.696493  
(5,3) : 91.789757  
(5,4) : 90.468475  
(5,5) : 89.199412
```

Figure 6: State Utility Values for Value Iteration






	0	1	2	3	4	5
0	99.901		94.947	93.776	92.556	93.230
1	98.294	95.784	94.446	94.299		90.819
2	96.849	95.487	93.200	93.078	93.004	91.696
3	95.455	94.353	93.178	91.022	91.717	91.790
4	94.213				89.451	90.468
5	92.838	91.629	90.436	89.257	88.471	89.199

Figure 7: State Utility Values in a grid format for Value Iteration

Plot of Utility estimates as a function of the number of iterations

As the graph below shows, the utility of each state has reached an equilibrium by the 688th iteration. The value iteration eventually converges to a unique set of solutions of the Bellman equations. The states who have a constant utility value of 0 are wall states.

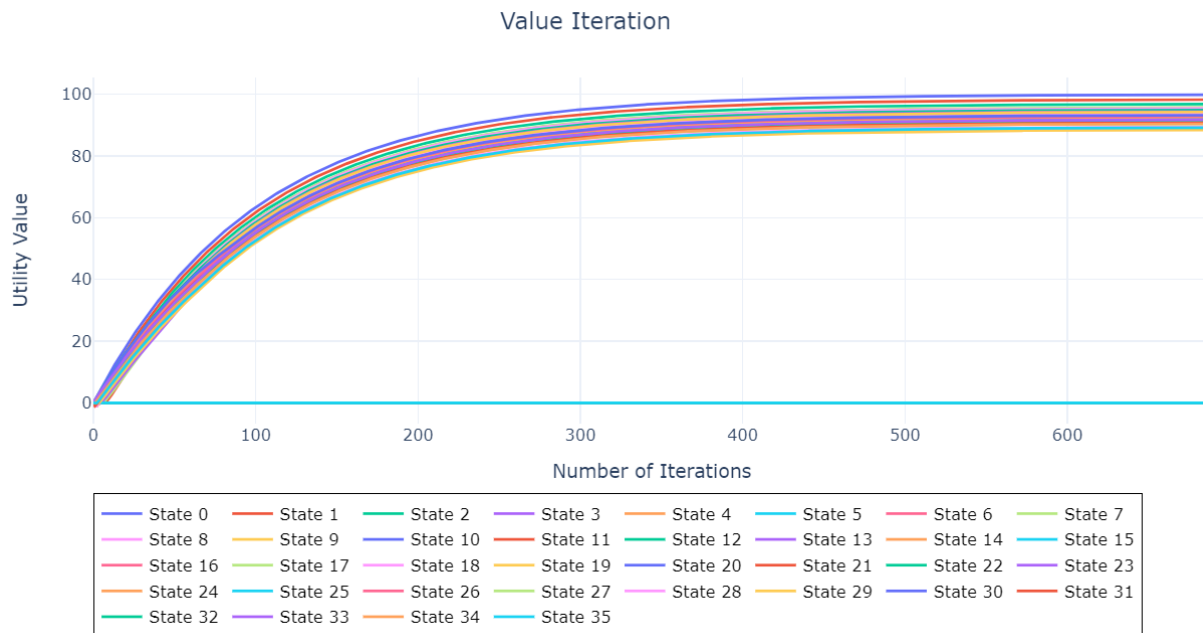


Figure 8: State Utility Values across iterations for Value Iteration

Method used	Value Iteration
No of iterations	688
Convergence Criteria	0.0010101010101010112

Implementation of Policy Iteration

Description

Before diving into the implementation of the policy iteration, there are two variables that are highly important in the Policy Iteration function:

- Boolean variable named '**unchanged**'
- Constant variable named '**K**'

The 'unchanged' variable is used to check if there are any changes to the policy. It is first initialized as true. Whenever there is an update in the policy, 'unchanged' variable will be set to false. The algorithm knows that there is a change in the policy if the state utility from the optimal policy action is higher than the current state utility and the unchanged variable will be changed to false. Once 'unchanged' remains true, it yields that there is no change in the policy and the algorithm terminates.

The 'K' variable represents the number of times the simplified bellman update is executed on the policy in the Policy Evaluation step.

Each state is initialized with 0 as the utility and a random action using <UtilityAndAction> 2-D array to represent those values.

The policy iteration function is defined in PolicyIteration.java. Policy iteration is separated into 2 parts for each loop:

- Policy Evaluation: Performs simplified bellman update for **K** times
- Policy Improvement: Calculates new maximum expected utility for the current state

For Policy Evaluation:

By performing the simplified Bellman update K times, it will compute all the utilities for each state based on the current policy. It does not use the expected maximum utility action in the Bellman Update as the action in each state is fixed by the policy.

```
usage justdilah
private UtilityAndAction[][] performPolicyEvaluation(UtilityAndAction[][] ActionUtilArr, MazeState[][] Maze, int K, double discount) {
    UtilityAndAction[][] currentActionAndUtilityArr = copyArray(ActionUtilArr);

    //initialise with the UtilityAndAction
    UtilityAndAction[][] newUtilityAndAction = initialiseArr();

    UtilityAndAction newActionUtility = null;
    for (int i=0; i<K; i++) {
        for (int row = 0; row < this.rows; row++) {
            for (int col = 0; col < this.cols; col++) {
                // Check if MazeState is Wall
                if (!Maze[row][col].isVisitable()) {continue;}

                newActionUtility = getSimplifiedBellmanUpdate(row, col, currentActionAndUtilityArr, Maze, discount);
                newUtilityAndAction[row][col] = newActionUtility;
            }
        }
        currentActionAndUtilityArr = copyArray(newUtilityAndAction);
    }
    return newUtilityAndAction;
}
```

Figure 9: Code snippet of Policy Evaluation function

For Policy Improvement:

Once the policy evaluation has been completed, the policy improvement function is executed. It will calculate the expected utility for every possible action (up, down, left, right) and retrieve the action with the maximum expected utility amongst all of the expected utilities. The new utility retrieved is then compared to the current utility which is the utility retrieved from Policy Evaluation Step. If the current utility is less than the new utility, it shows the policy is no longer updating and the unchanged variable is true which will terminate the algorithm.

```
1 usage  justdilah *
private boolean performPolicyImprovement(){
    double newMaxUtility = 0.0;
    double currPolicyUtility = 0.0;
    boolean ucvar = true;
    for (int row = 0; row < this.rows; row++){
        for (int col = 0; col < this.cols; col++) {
            // Check if MazeState is Wall
            if (!Maze[row][col].isVisitable()) {
                continue;
            }

            UtilityAndAction chosenUtilityAndAction = getMaxUtilityAndAction(this.optimalUtilityAndActionArray, row, col);
            newMaxUtility = chosenUtilityAndAction.getUtility();
            Action policyAction = this.optimalUtilityAndActionArray[row][col].getAction();
            currPolicyUtility = getExpectedUtility(this.optimalUtilityAndActionArray, policyAction, row, col);

            if (newMaxUtility > currPolicyUtility) {
                this.optimalUtilityAndActionArray[row][col].setAction(chosenUtilityAndAction.getAction());
                ucvar = false;
            }
        }
    }
}
```

Figure 10: Code snippet of Policy Improvement function

```

4 usages  justdilah *
public PolicyIteration(Maze map, double discount, int K) {
    this.discount = discount;

    //retrieves attributes of the Maze
    this.rows = map.getRows();
    this.cols = map.getCols();
    this.Maze = map.getMazeMap();

    this.no_of_iterations = 0;

    // Intialise the array with state utility of 0 and random action for each state
    this.currentUtilityAndActionArray = generateRandomPolicy();

    this.allUtilityAndActionArray.add(this.currentUtilityAndActionArray);

    do {
        this.no_of_iterations++;

        // Update the state utilities by executing policy evaluation
        this.optimalUtilityAndActionArray = performPolicyEvaluation(this.currentUtilityAndActionArray, this.Maze, K, discount);

        // Update Actions by executing policy improvement
        this.unchanged = performPolicyImprovement();
    } while (!this.unchanged);
}

```

Figure 11: Code snippet of Policy Iteration function

Plot of the Optimal Policy

Starting position: (3,2)

	0	1	2	3	4	5
0	Λ		<	<	<	Λ
1	Λ	<	<	<		Λ
2	Λ	<	<	Λ	<	<
3	Λ	<	<	<	Λ	Λ
4	Λ				Λ	Λ
5	Λ	<	<	<	Λ	Λ

Utility of all states

```
===== REFERENCE UTILITIES OF STATES =====  
> Coordinates are in (col,row) format. Top-left corner is (0,0) <  
(0,0) : 91.037138  
(0,1) : 89.430500  
(0,2) : 87.985638  
(0,3) : 86.590977  
(0,4) : 85.349658  
(0,5) : 83.974612  
(1,1) : 86.920156  
(1,2) : 86.623566  
(1,3) : 85.489632  
(1,5) : 82.765916  
(2,0) : 86.082963  
(2,1) : 85.582617  
(2,2) : 84.336057  
(2,3) : 84.314572  
(2,5) : 81.572290  
(3,0) : 84.912510  
(3,1) : 85.435362  
(3,2) : 84.214355  
(3,3) : 82.158417  
(3,5) : 80.393548  
(4,0) : 83.692108  
(4,2) : 84.140494  
(4,3) : 82.853002  
(4,4) : 80.586921  
(4,5) : 79.607402  
(5,0) : 84.365127  
(5,1) : 81.954425  
(5,2) : 82.832796  
  
(5,3) : 82.926057  
(5,4) : 81.604770  
(5,5) : 80.335700
```

Figure 12: State Utility Values for Policy Iteration






	0	1	2	3	4	5
0	83.747		78.791	77.621	76.400	77.066
1	82.140	79.630	78.292	78.145		74.655
2	80.695	79.333	77.046	76.924	76.850	75.541
3	79.301	78.199	77.024	74.868	75.562	75.634
4	78.059				73.296	74.313
5	76.684	75.476	74.282	73.103	72.316	73.044

Figure 13: State Utility Values in a grid format for Policy Iteration

Plot of Utility estimates as a function of the number of iterations

As the graph below shows, the utility of each state has reached an equilibrium by the 7th iteration. The states who have a constant utility value of 0 are wall states.

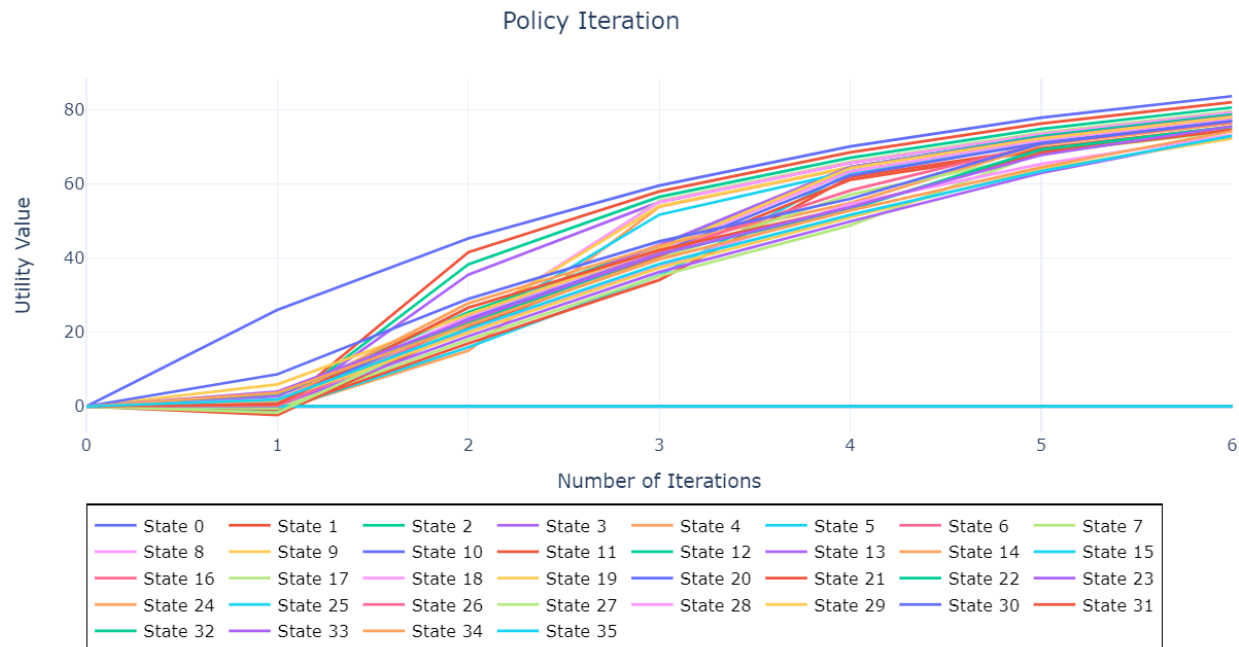


Figure 14: State Utility Values across iterations for Policy Iteration

Method used	Policy Iteration
No of iterations	7

Bonus Question

Complicated Maze Environment

Before answering the question on how the number of states and the complexity of the environment can affect convergence, I decided to perform a few experiments:

1. Increase the number of states by increasing the scale of the environment
2. Generate different number of the states (Reward States, Penalty States) for the Maze environment

1. Increase the number of states by increasing the scale of the environment

The scale is increased to 2. Hence, the total number of states is 144.

Plot of the Optimal Policy

Starting positions: (6,4),(6,5),(7,4),(7,5)

Λ	Λ			Λ	<	<	<	>	>	>	Λ
Λ	Λ			Λ	Λ	<	v	>	>	Λ	Λ
Λ	Λ	<	>	Λ	Λ	>	v			Λ	Λ
Λ	Λ	<	>	Λ	>	>	Λ			Λ	Λ
Λ	Λ	<	Λ	Λ	Λ	Λ	Λ	>	<	<	v
Λ	Λ	Λ	Λ	Λ	>	Λ	>	Λ	Λ	v	v
Λ	Λ	Λ	Λ	>	Λ	Λ	Λ	Λ	Λ	>	v
Λ	Λ	Λ	Λ	Λ	Λ	Λ	>	Λ	>	>	Λ
Λ	Λ							Λ	Λ	Λ	Λ
Λ	Λ							Λ	>	Λ	Λ
Λ	Λ	<	<	>	>	>	>	>	>	Λ	Λ
Λ	Λ	<	>	>	>	>	>	>	>	Λ	Λ

Method used	Value Iteration
No of iterations	688
Convergence Criteria	0.0010101010101010112

Method used	Policy Iteration
No of iterations	15

Findings with this experiment:

For Value Iteration, even though the number of states has increased from 36 to 144, the number of iterations remained the same.

For Policy Iteration, the number of iterations has increased from 7 (referred to as Part 1) to 15. This shows that the complexity of the maze environment has increased in terms of policy iteration as it took a longer time for the state utilities to reach equilibrium compared to the original maze environment.

Both Value Iteration and Policy Iteration algorithms are able to find the optimal policy. Hence, this shows that increasing the number of states does not necessarily prevents the agent from finding the optimal policy.

2. Generate different number of the states (Reward States, Penalty States) for the Maze environment

The reason behind performing these test cases: I would like to see if there are any significant impacts that will occur to the convergence rate when the number of unique states varies (eg. What impact can occur by changing the number of rewards from 6 to 5)

1st Case: Each category of states (Penalty, Reward, Wall) has an equal number of states

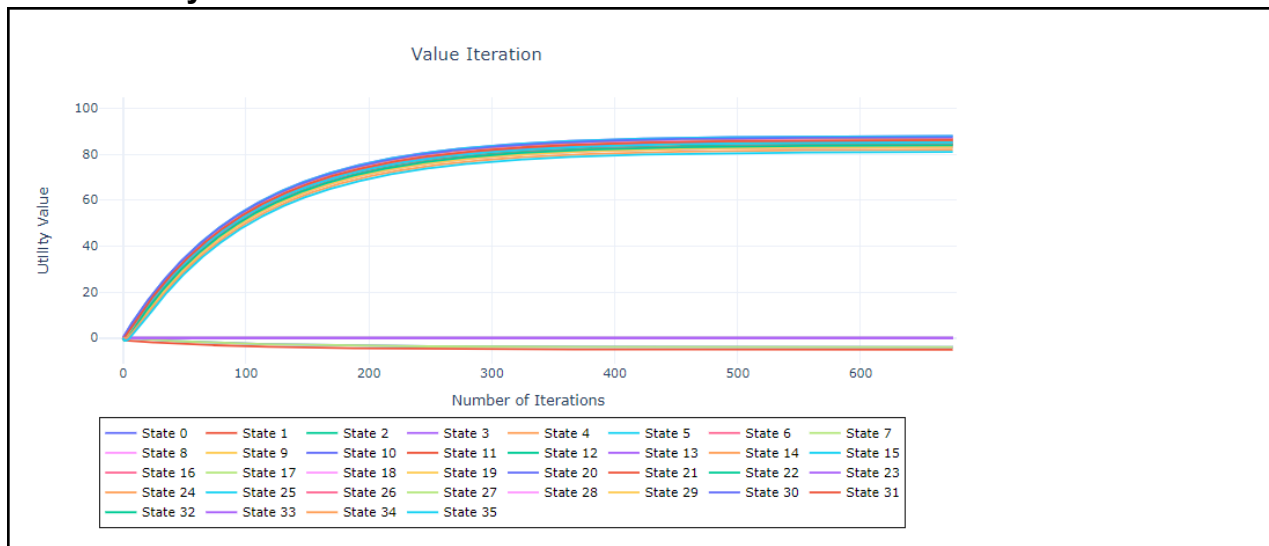
	No of Penalty States	5
	No of Reward States	5
	No of Wall States	5
	No of Start States	1
	Total No of States (including Empty States)	36

Value Iteration

Plot of the Optimal Policy
Starting Position: (3,1)

	0	1	2	3	4	5
0	Λ		>	Λ	>	Λ
1	Λ	>		Λ	Λ	Λ
2			v	Λ	Λ	Λ
3	Λ	<	<	<	<	
4	Λ	<	<	v	<	<
5	<	<	<	<	<	<

Plot of Utility estimates as a function of the number of iterations



Value Iteration

Number of Iterations: 675

Convergence Criteria: 0.0010101010101010112

Observations:

Based on the graph shown above, a couple of utilities for the states approaches the negative value instead of the positive value as it is nearing to the convergence criteria

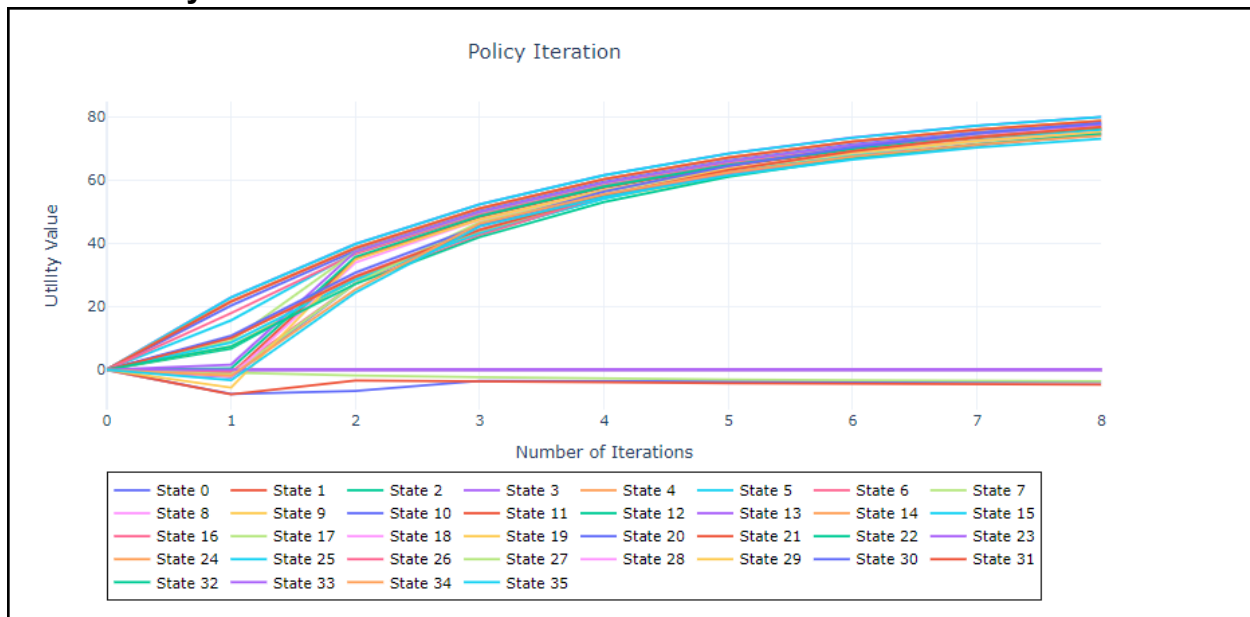
Policy Iteration

Plot of the Optimal Policy

Starting Position: (3,1)

	0	1	2	3	4	5
0	Λ		>	Λ	>	Λ
1	Λ	>		Λ	Λ	Λ
2			v	Λ	Λ	Λ
3	Λ	<	<	<	Λ	
4	Λ	<	<	v	<	<
5	<	<	<	<	<	<

Plot of Utility estimates as a function of the number of iterations



Policy Iteration

Number of Iterations: 8

Observation:

Similar to findings in Value Iteration, some states have negative values as it reaches the end of the 8th iteration. Those states found their unchanged utility states earlier before the states that have positive utility values. The difference between the findings in Value Iteration and Policy Iteration is in Policy Iteration, as the algorithm continues, state utilities that hold a negative value change to a positive value.

Findings with this experiment:

There are a couple of state utilities with a negative value. The reason behind this is that those states are unable to find the path to the reward state. For example, state 1 is confined within the walls. There is no path from state 1 to a reward state. The only rewards that it is able to accumulate are only from the penalty states and empty states which are confined within the walls hence the negative utility values.

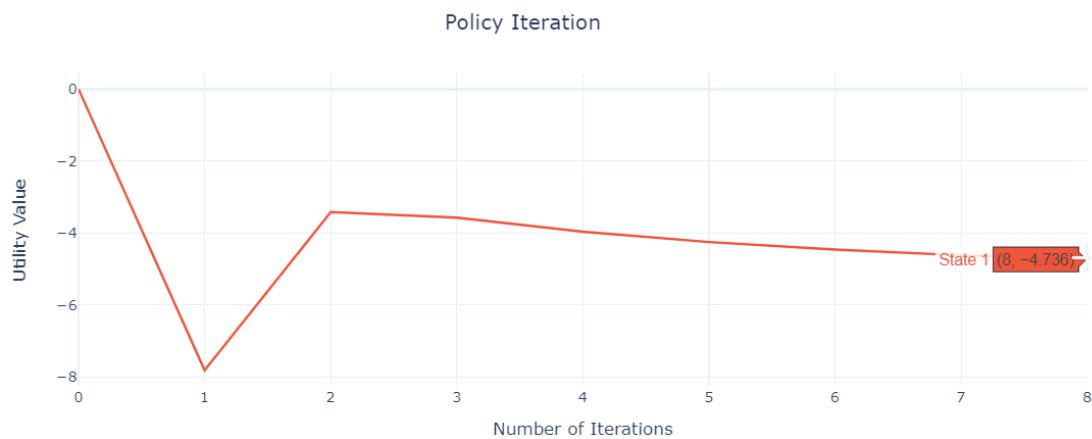


Figure 15: State Utility Values across iterations for Value Iteration for state 1

However, for both policy iteration and value iteration, from the starting point, it is able to find its way to a reward state. Therefore, it has still achieved the optimal policy.

2nd Case: Modify the number of Penalty states and Rewards states

	No of Penalty States	8
	No of Reward States	2
	No of Wall States	5
	No of Start States	1
	Total No of States (including Empty States)	36

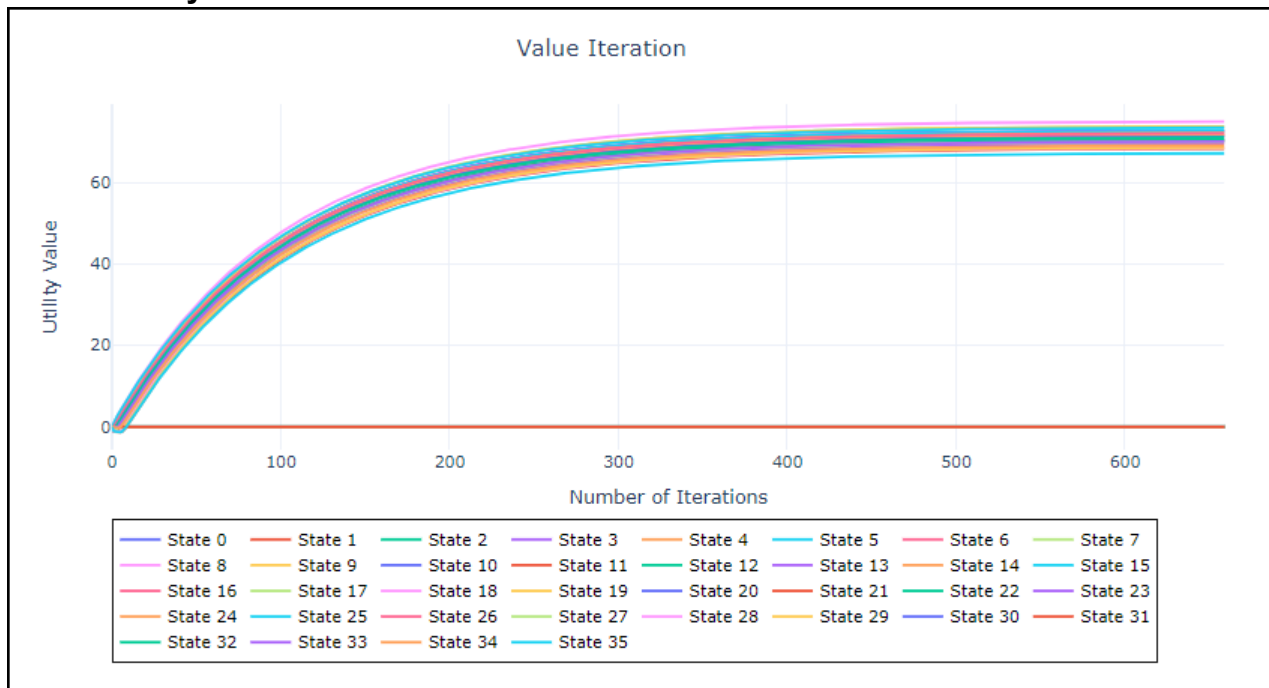
Value Iteration

Plot of the Optimal Policy

Starting Position: (5,3)

	0	1	2	3	4	5
0	v	v		>	v	<
1	>	v	<	>	>	
2		<		>	Λ	<
3	>	Λ	<	<	Λ	Λ
4	Λ		Λ	<	<	Λ
5	Λ	<	Λ	Λ	<	Λ

Plot of Utility estimates as a function of the number of iterations



Value Iteration

Number of Iterations: 659

Convergence Criteria: 0.0010101010101010112

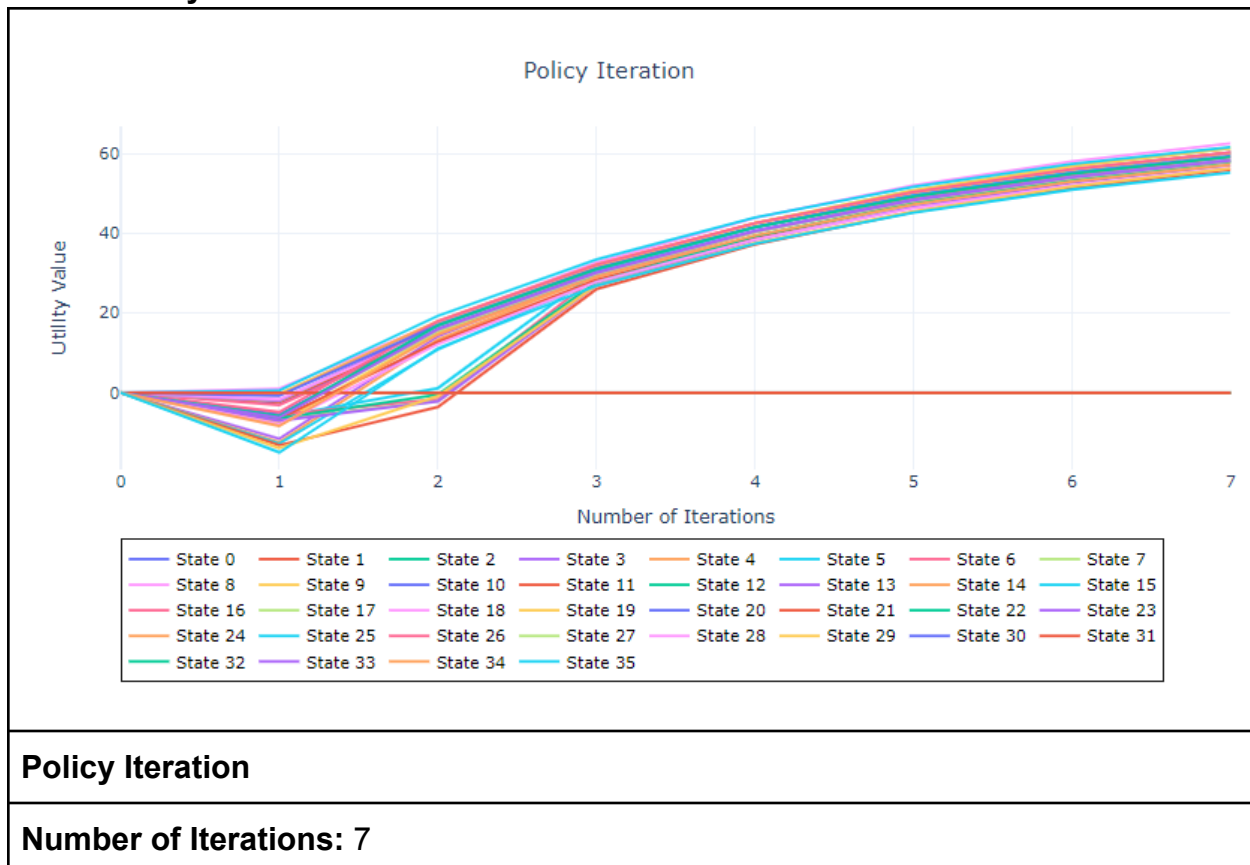
Policy Iteration

Plot of the Optimal Policy

Starting Position: (5,3)

	0	1	2	3	4	5
0	v	v		>	v	<
1	>	v	<	>	>	
2		<		>	^	<
3	>	^	<	<	^	^
4	^		^	<	<	^
5	^	<	^	^	<	^

Plot of Utility estimates as a function of the number of iterations



Findings from this experiment

For Policy Iteration, the number of iterations has decreased from 8 (from 1st Case) to 7 (from 2nd Case).

For value iteration, no matter how many changes I made to the number of walls or penalties, or rewards and to execute those runs, the number of iterations will not be beyond 688 iterations. It will either generate a number of iterations lower than 688 iterations or 688 iterations. It seems that the maximum number of iterations is 688 till it hits the convergence criteria for value iteration.

Increasing the number of penalty states and decreasing the number of reward states does not necessarily lead to a more complicated maze environment. The number of iterations taken for the 2nd case (Value Iteration and Policy Iteration) is lower than the number of iterations for the 1st case (Value Iteration and Policy Iteration).

Conclusion

Based on the results achieved from the complicated maze environments, as long as the maze is well structured, there will be a right policy that the agent can follow to achieve maximized expected utility. If the start state is confined within the wall and the reward states are located outside of those boundaries, the agent will not be able to achieve the optimal policy. As it stays within those walls, it will only accumulate negative or 0-value rewards and will not be able to increase the utility value.