NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

AY2023/24 Semester 2

CC4046 - Intelligent Agents

Assignment 1: Agent Decision Making

Name	Matriculation Number
Dann, Wee Zi Jun	U2122790K

Setup Guide

Environment Context

Language: Java - High level, class-based, object-oriented programming language.
Build Tool: Apache Maven - Build automation tool to compile, build and test Java Programming Language projects.
IDE: Visual Studio Code - Streamlined source code editor that supports development operations such as debugging, task running, and version control.

Setup Environment

Download IDE

Download Visual Studio Code via https://code.visualstudio.com/download

Installation Steps

1. Install Java Development Kit (JDK 17+)

```
java --version
```

2. Install Apache Maven (Version 3.9+)

```
mvn -version
```

3. Clone the repository

```
git clone https://github.com/dannweeeee/CZ4046-Intelligent-Agents.git
```

4. Change directory to assignment-one

```
cd assignment-one
```

1. Compile assignment-one

```
mvn compile
```

Debugging Source Code

There are 4 executable files that will be runned for specific parts of assignment 1

1. **Main.java**
   Executable file for the maze environment built for Part 1 - Contains the optimal policy and the utilities of all (non-wall) states using value iteration and policy iteration.

```
mvn exec:java -Dexec.mainClass="cz4046.main.Main"
```

2. **ComplicatedMazeOne.java**
   Executable file for the maze environment built for Part 2 (Same Category States).

```
mvn exec:java -Dexec.mainClass="cz4046.main.ComplicatedMazeOne”
```

3. **ComplicatedMazeTwo.java**
   Executable file for the maze environment for Part 2 (Different Category States).

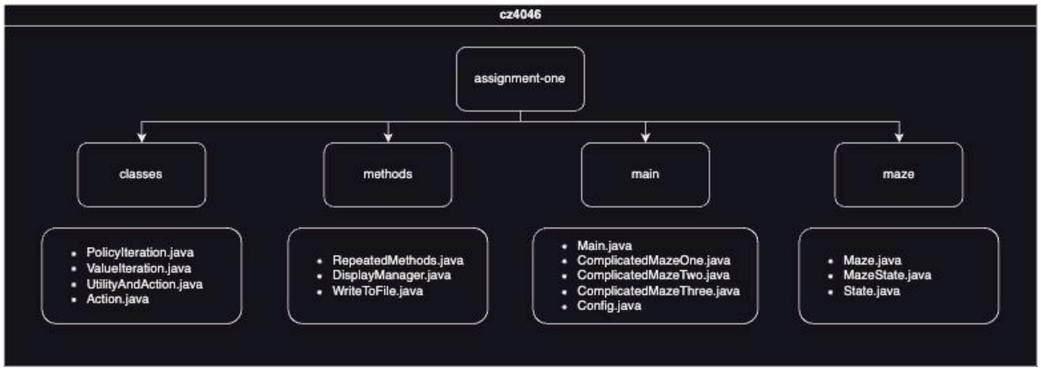
```
mvn exec:java -Dexec.mainClass="cz4046.main.ComplicatedMazeTwo"
```

4. **ComplicatedMazeThree.java**
   Executable file for the maze environment for Part 2 (Larger Scale).

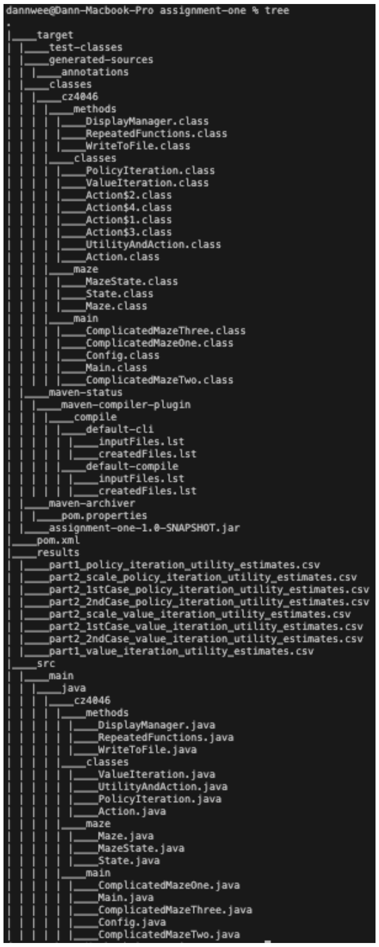
```
mvn exec:java -Dexec.mainClass="cz4046.main.ComplicatedMazeThree"
```

Source Code

The source code is decomposed into 4 major folder structures, namely Classes, Methods, Main & Maze. Each of these folder structures contains java files that facilitates the solution to the Agent Decision Making problem.



The tree structure of the source code is shown as follows:



```
dannwee@Dann-Macbook-Pro assignment-one % tree
.
|____target
| |____test-classes
| |____generated-sources
| | |____annotations
| |____classes
| | |____cz4046
| | | |____methods
| | | | |____DisplayManager.class
| | | | |____RepeatedFunctions.class
| | | | |____WriteToFile.class
| | | |____classes
| | | | |____PolicyIteration.class
| | | | |____ValueIteration.class
| | | | |____Action$2.class
| | | | |____Action$4.class
| | | | |____Action$1.class
| | | | |____Action$3.class
| | | | |____UtilityAndAction.class
| | | | |____Action.class
| | | |____maze
| | | | |____MazeState.class
| | | | |____State.class
| | | | |____Maze.class
| | | |____main
| | | | |____ComplicatedMazeThree.class
| | | | |____ComplicatedMazeOne.class
| | | | |____Config.class
| | | | |____Main.class
| | | | |____ComplicatedMazeTwo.class
| |____maven-status
| | |____maven-compiler-plugin
| | | |____compile
| | | | |____default-cli
| | | | | |____inputFiles.lst
| | | | | |____createdFiles.lst
| | | | |____default-compile
| | | | | |____inputFiles.lst
| | | | | |____createdFiles.lst
| |____maven-archiver
| | |____pom.properties
| |____assignment-one-1.0-SNAPSHOT.jar
|____pom.xml
|____results
| |____part1_policy_iteration_utility_estimates.csv
| |____part2_scale_policy_iteration_utility_estimates.csv
| |____part2_1stCase_policy_iteration_utility_estimates.csv
| |____part2_2ndCase_policy_iteration_utility_estimates.csv
| |____part2_scale_value_iteration_utility_estimates.csv
| |____part2_1stCase_value_iteration_utility_estimates.csv
| |____part2_2ndCase_value_iteration_utility_estimates.csv
| |____part1_value_iteration_utility_estimates.csv
|____src
| |____main
| | |____java
| | | |____cz4046
| | | | |____methods
| | | | | |____DisplayManager.java
| | | | | |____RepeatedFunctions.java
| | | | | |____WriteToFile.java
| | | | |____classes
| | | | | |____ValueIteration.java
| | | | | |____UtilityAndAction.java
| | | | | |____PolicyIteration.java
| | | | | |____Action.java
| | | | |____maze
| | | | | |____Maze.java
| | | | | |____MazeState.java
| | | | | |____State.java
| | | | |____main
| | | | | |____ComplicatedMazeOne.java
| | | | | |____Main.java
| | | | | |____ComplicatedMazeThree.java
| | | | | |____Config.java
| | | | | |____ComplicatedMazeTwo.java
```

Classes

This folder contains the classes that defines various components and algorithms related to reinforcement learning and Markov decision processes.

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 actions of the states.

Methods

This folder contains utility classes with methods commonly utilized throughout the project.

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.

Main

This folder serves as the entry point for executing different components and environments related to the assignment.

Class	Descriptions
Main.java	To execute the maze environment built for Part 1 of the assignment.
ComplicatedMazeOne.java	To execute the maze environment built for Part 2 of the assignment (Same States).
ComplicatedMazeTwo.java	To execute the maze environment built for Part 2 of the assignment (Different States).
ComplicatedMazeThree.java	To execute the maze environment built for Part 2 of the assignment (Larger Scale).
Config.java	Stores the constant variables.

Maze

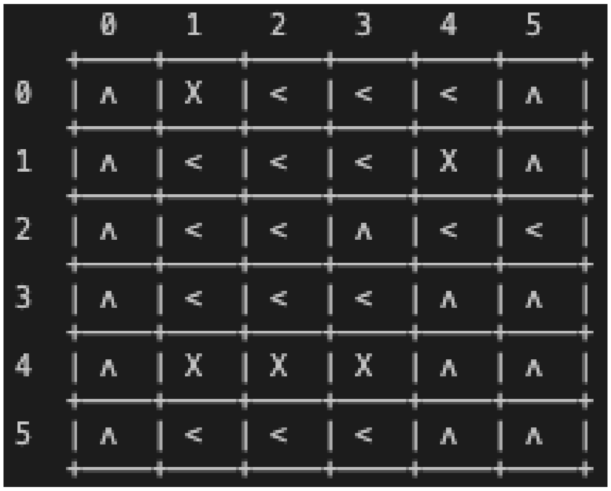
This folder contains classes related to initializing and managing the maze environment for the assignment.

Class	Descriptions
Maze.java	Initializes the maze environment.
MazeState.java	Stores the *state type, reward value, x and y coordinates.*
State.java	Stores the states - *penalty, empty reward, wall, start.*

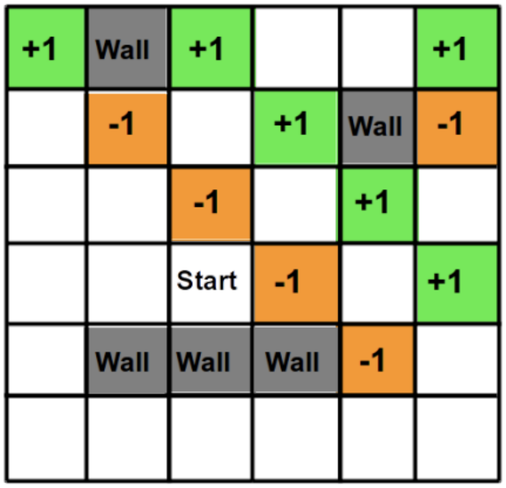
Maze Environment

The maze is represented by a 2-Dimensional array. For each coordinate, it holds a state. The start state as shown below is at coordinates (3,2). 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 [brown]
- Wall State: 0.00 [grey]



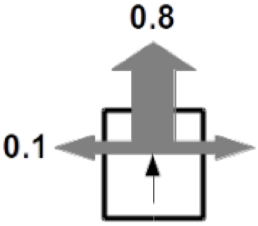
```
     0   1   2   3   4   5
   +---+---+---+---+---+---+
0  | ^ | X | < | < | < | ^ |
   +---+---+---+---+---+---+
1  | ^ | < | < | < | X | ^ |
   +---+---+---+---+---+---+
2  | ^ | < | < | ^ | < | < |
   +---+---+---+---+---+---+
3  | ^ | < | < | < | ^ | ^ |
   +---+---+---+---+---+---+
4  | ^ | X | X | X | ^ | ^ |
   +---+---+---+---+---+---+
5  | ^ | < | < | < | ^ | ^ |
   +---+---+---+---+---+---+
```



+1	Wall	+1			+1
	-1		+1	Wall	-1
		-1		+1	
		Start	-1		+1
	Wall	Wall	Wall	-1	

The transition model is as follows:

- The intended outcome occurs with a probability of 0.8 and 0.1 as depicted by the diagram below.



Value Iteration Implementation

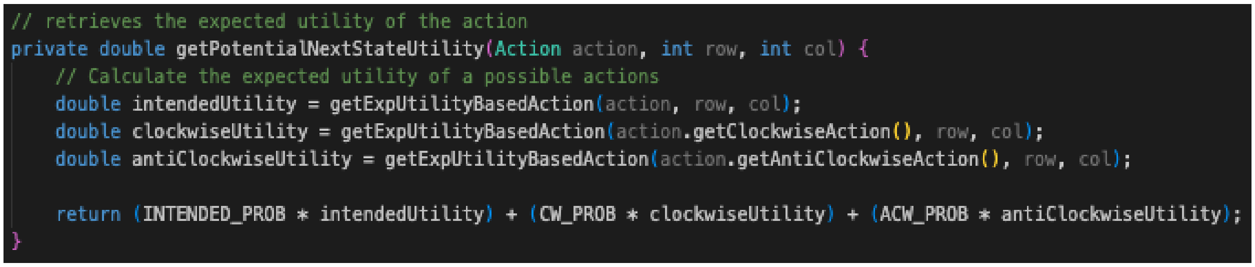
The value iteration function is defined in *ValueIteration.java* with the following Bellman equation used:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

The Bellman equation serves as the cornerstone of the value iteration method for solving Markov Decision Processes (MDPs). In a maze environment with n possible states, each state corresponds to a unique Bellman equation. Thus, there are *n* Bellman equations in total. Solving these equations entails employing an iterative approach. Initially, each state is initialized with a utility of 0 and a null action, utilizing a 2-D array represented by <UtilityAndAction>. During each iteration, a Bellman update is applied to adjust the utility of each state based on the utilities of its neighboring states.

Bellman Equation in Source Code:

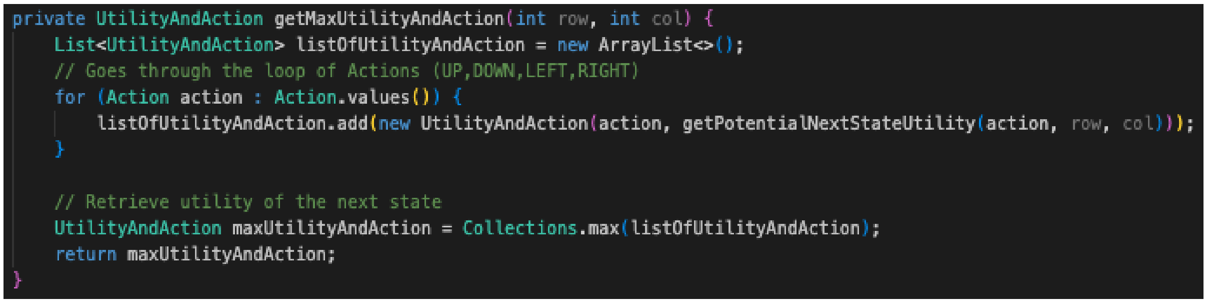
1. Retrieves the expected utility based on each available action the agent can take:



```java
// retrieves the expected utility of the action
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);
}
```

Code snippet for computation of Expected Utility

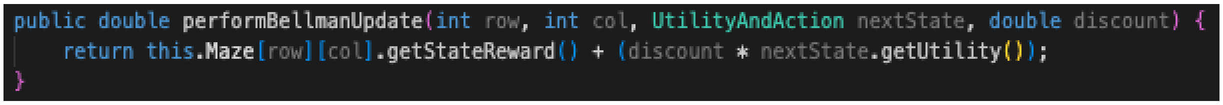
2. Select the optimal action that the agent will take based on the maximum utility for the next state:



```java
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;
}
```

Code snippet for Maximum Expected Utility

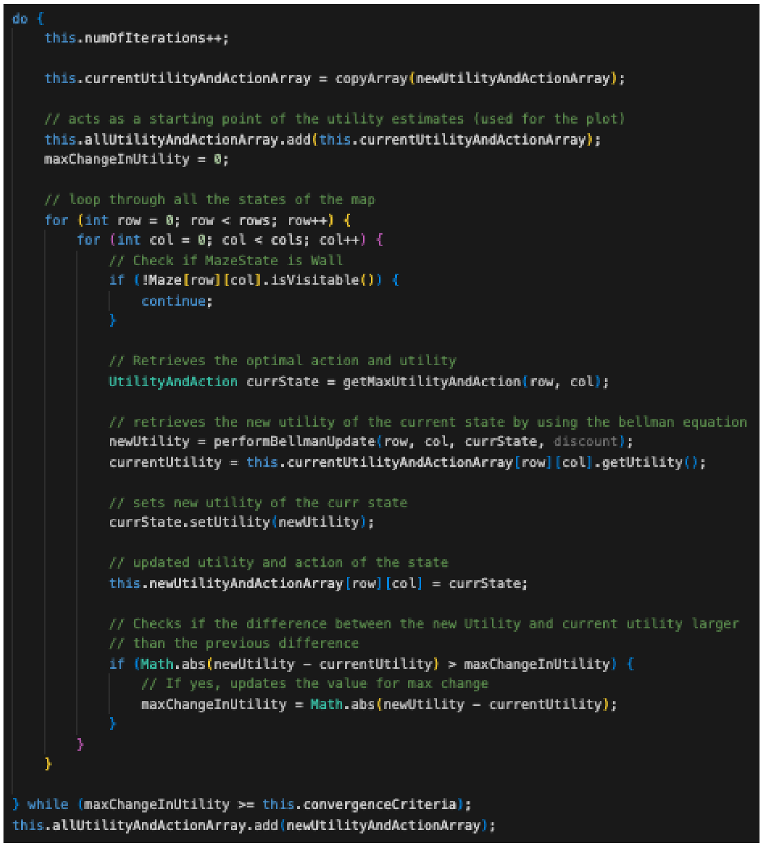
3. Bellman update process involves computing the new value of the state utility by substituting the current state utility value with the expected discounted utility of the next state:



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

Code snippet for Bellman Update

The Bellman update iterates until it converges to the optimal value function. Convergence to the optimal value function is reached when the maximum change in utility value among all states in a particular iteration falls below the convergence threshold criteria.



```java
do {
    this.numOfIterations++;

    this.currentUtilityAndActionArray = copyArray(newUtilityAndActionArray);

    // acts as a starting point of the utility estimates (used for the plot)
    this.allUtilityAndActionArray.add(this.currentUtilityAndActionArray);
    maxChangeInUtility = 0;

    // 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);
            }
        }
    }

} while (maxChangeInUtility >= this.convergenceCriteria);
this.allUtilityAndActionArray.add(newUtilityAndActionArray);
```

Code snippet for Value Iteration

The convergence criteria is determined by the formula: Epsilon * ((1 - discount) / discount). Here, Epsilon denotes the maximum permissible error for the utility of any state. To compute Epsilon, a constant variable 'c' is multiplied by the maximum reward value.

I opted to assign the variable 'c' a value of 0.100. With a discount factor of 0.99, the iteration count is minimized for c = 0.100 compared to other values of 'c', while still converging to the accurate utilities.

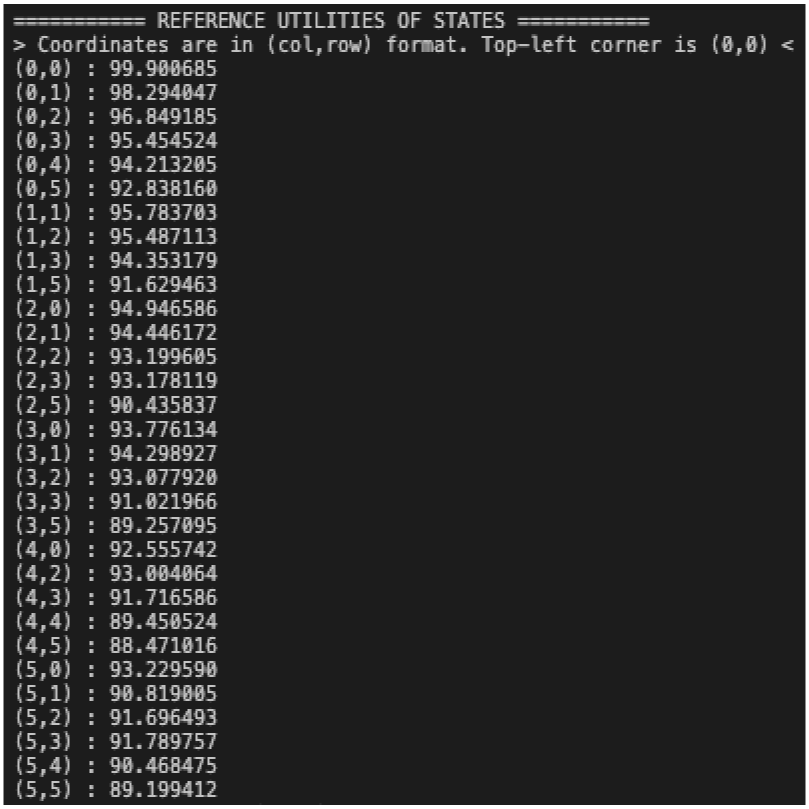
Upon reaching the optimal value function, the 2-D array <UtilityAndAction> will store the highest utility for each state alongside its corresponding optimal action.

Optimal Policy Plot for Value Iteration

At the initial position of (3, 2), after 688 iterations, the maximum change in utility value across all states in that specific iteration falls below the convergence threshold, indicating that equilibrium has been attained.

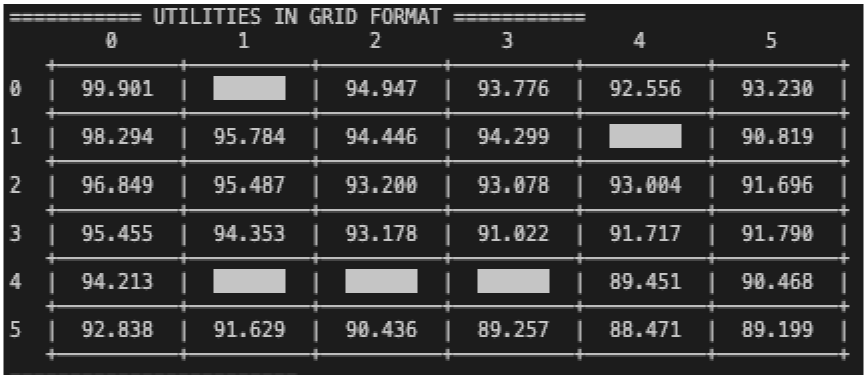
	0	1	2	3	4	5
0	^	⬛	<	<	<	^
1	^	<	<	<	⬛	^
2	^	<	<	^	<	<
3	^	<	<	<	^	^
4	^	⬛	⬛	⬛	^	^
5	^	<	<	<	^	^

Reference Utilities of 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
```

Code Snippet of Reference Utilities of States for Value Iteration



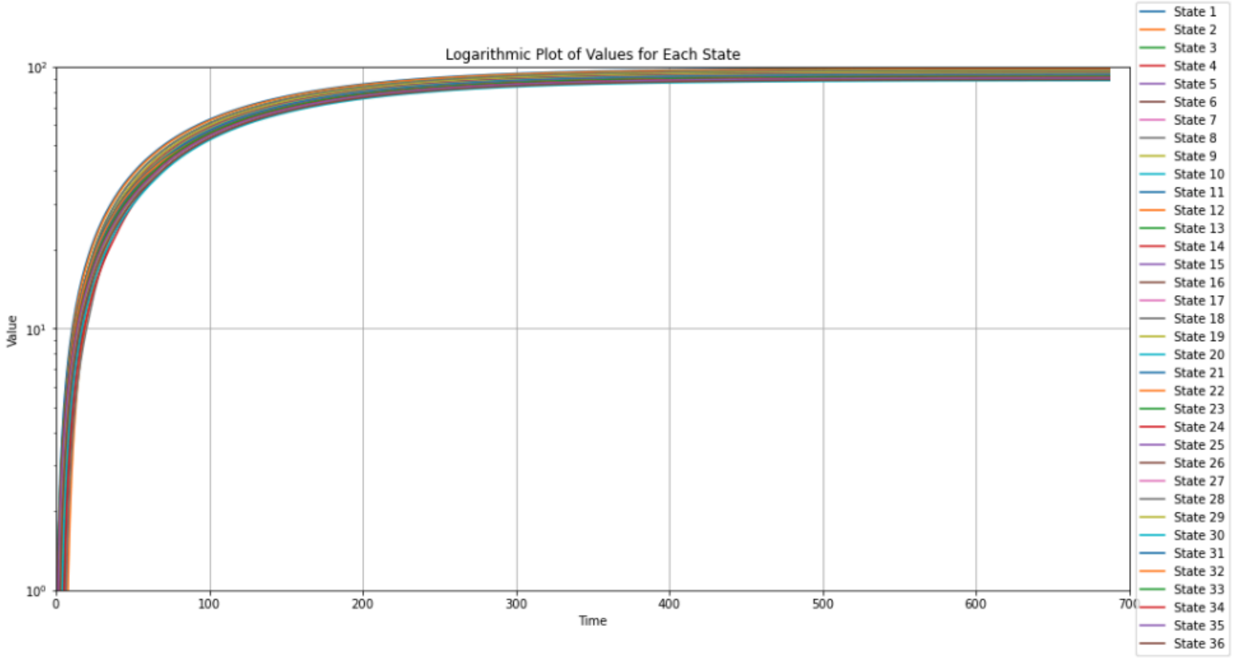
```
=========== UTILITIES IN GRID FORMAT ===========
          0          1          2          3          4          5
     +----------+----------+----------+----------+----------+----------+
0    |  99.901  |  ██████  |  94.947  |  93.776  |  92.556  |  93.230  |
     +----------+----------+----------+----------+----------+----------+
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  |
     +----------+----------+----------+----------+----------+----------+
```

Code Snippet of Reference Utilities of States in Grid Format for Value Iteration

Graph of Utility Estimates over Iterations (Value Iteration)

The graph depicted below illustrates the convergence of state utility to equilibrium by the 688th iteration. Through value iteration, the system eventually stabilizes, yielding a singular solution set for the Bellman equations. Notably, states with a constant utility value of 0 are identified as wall states.

Method	Value Iteration
No. of Iterations	688
Convergence Criteria	0.0010101010101010112



Graph of Utility Estimates over Iterations for Value Iteration

Policy Iteration Implementation

Before delving into the implementation of policy iteration, it's crucial to understand two key variables within the Policy Iteration function:

1. A **boolean variable named 'unchanged'** is employed to monitor changes in the policy. Initially set to true, it signifies stability in the policy. Whenever there's an update in the policy, 'unchanged' is set to false. The algorithm detects a policy change when the state utility derived from the optimal policy action surpasses the current state utility. Once 'unchanged' remains true, it indicates no further policy changes, prompting the algorithm to terminate.

2. The **constant variable 'K'** signifies the number of iterations during the simplified Bellman update in the Policy Evaluation step.

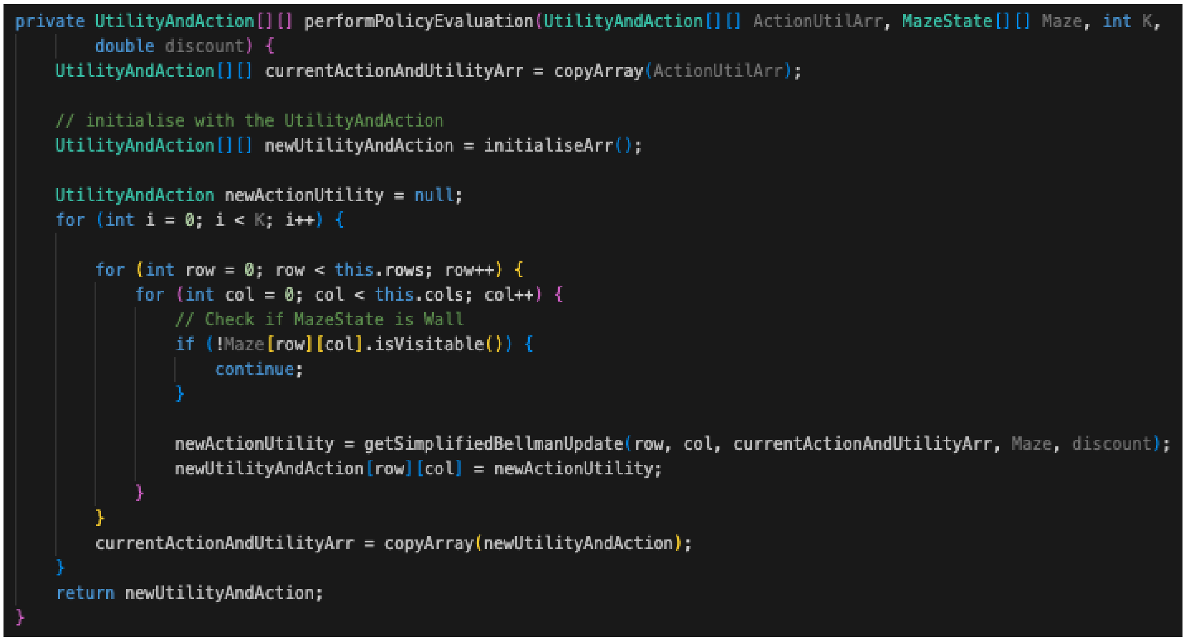
In the initialization phase, each state's utility is set to 0, with a random action assigned using a <UtilityAndAction> 2-D array to represent these values.

The policy iteration function is encapsulated within PolicyIteration.java. This process consists of two main stages:

- **Policy Evaluation:** Executes the simplified Bellman update K times.
- **Policy Improvement:** Determines the new maximum expected utility for the current state in each iteration.

In the Policy Evaluation phase:

By iteratively applying the simplified Bellman update K times, the algorithm computes the utilities for each state according to the current policy. This process disregards the expected maximum utility action in the Bellman Update, as the action in each state remains fixed by the policy.

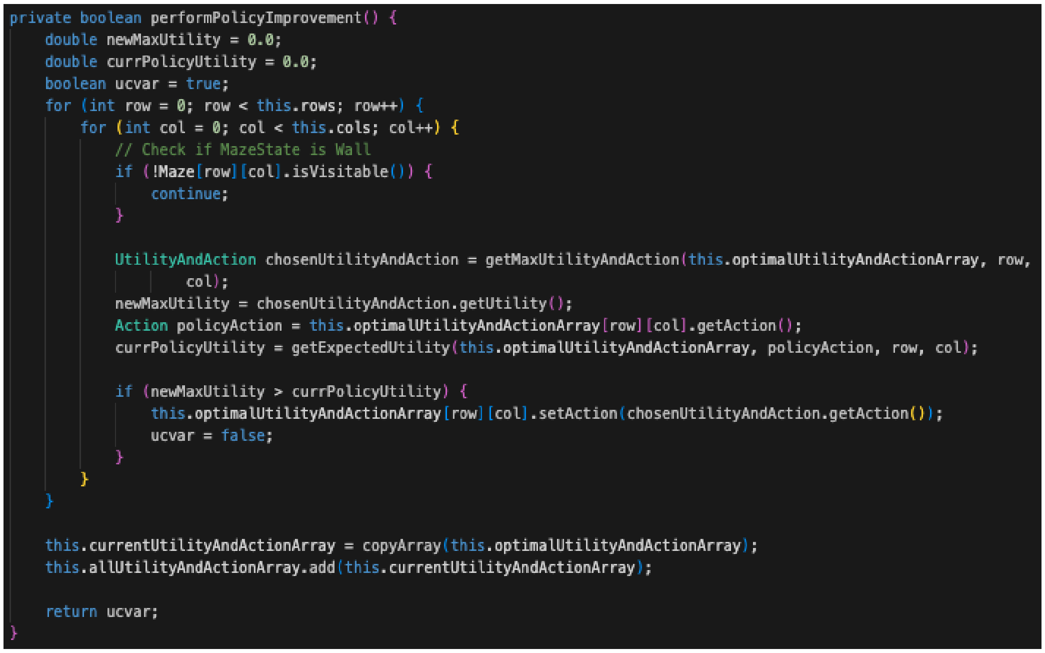


```java
private UtilityAndAction[][] performPolicyEvaluation(UtilityAndAction[][] ActionUtilArr, MazeState[][] Maze, int K,
        double discount) {
    UtilityAndAction[][] currentActionAndUtilityArr = copyArray(ActionUtilArr);

    // 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;
}
```

Code snippet for Policy Evaluation

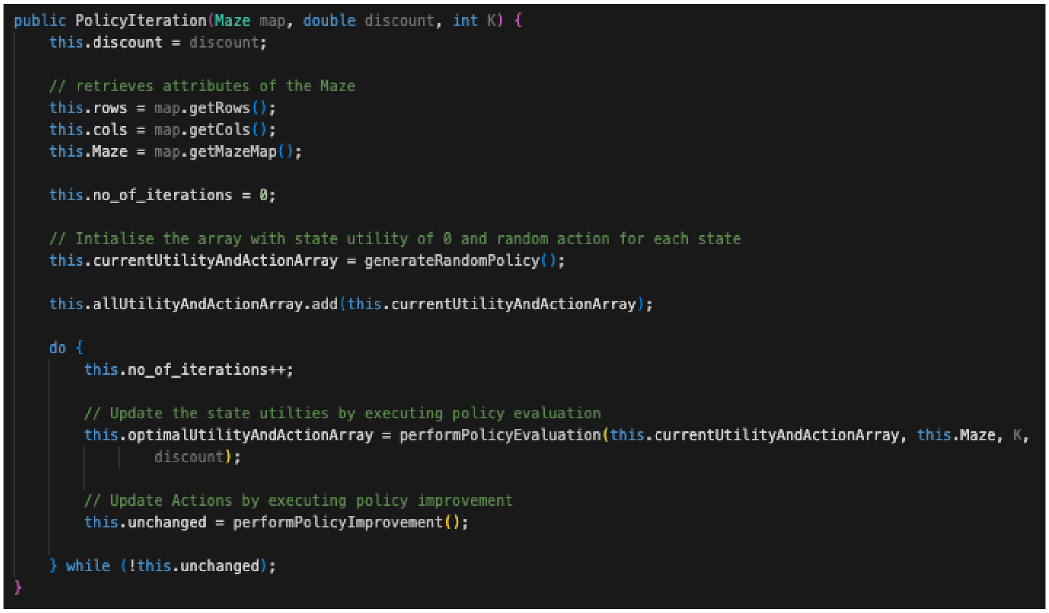
In the Policy Improvement phase:

After completing the policy evaluation, the policy improvement function is invoked. It computes the expected utility for each potential action (up, down, left, right) and selects the action with the highest expected utility from among them. Subsequently, the newly acquired utility is compared with the current utility obtained from the Policy Evaluation Step. If the current utility is lower than the new utility, it indicates that the policy has ceased to update. This condition sets the 'unchanged' variable to true, leading to the termination of the algorithm.



```java
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;
            }
        }
    }

    this.currentUtilityAndActionArray = copyArray(this.optimalUtilityAndActionArray);
    this.allUtilityAndActionArray.add(this.currentUtilityAndActionArray);

    return ucvar;
}
```

Code snippet for Policy Improvement



```java
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 utilties 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);
}
```

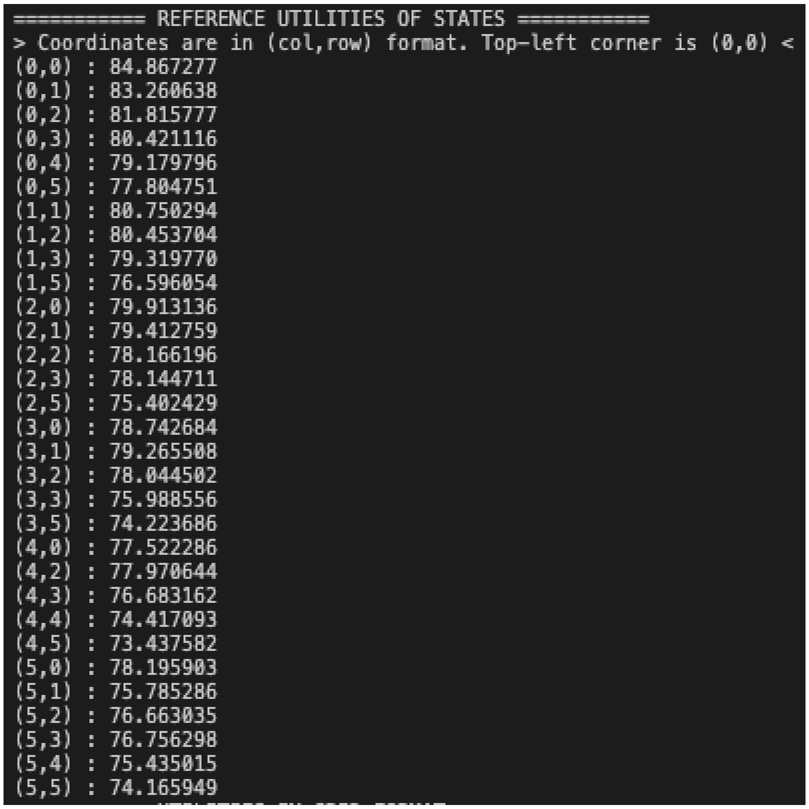
Code snippet of Policy Iteration Function

Optimal Policy Plot for Policy Iteration

At the starting position of (3, 2), with a discount factor of 0.99 and a maximum reward value of 1.0, a constant of 0.1 is utilized. Over the course of 30 executions of the Simplified Bellman update, the process is iterated 9 times. This involves the systematic evaluation and refinement of policies to optimize decision-making strategies, iteratively balancing short-term rewards with long-term goals under the given parameters.

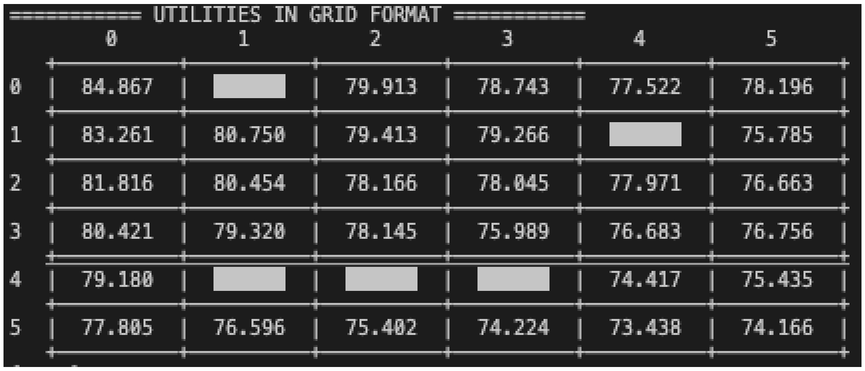
	0	1	2	3	4	5
0	^	⬛	<	<	<	^
1	^	<	<	<	⬛	^
2	^	<	<	^	<	<
3	^	<	<	<	^	^
4	^	⬛	⬛	⬛	^	^
5	^	<	<	<	^	^

Reference Utilities of States



```
=========== REFERENCE UTILITIES OF STATES ===========
> Coordinates are in (col,row) format. Top-left corner is (0,0) <
(0,0) : 84.867277
(0,1) : 83.260638
(0,2) : 81.815777
(0,3) : 80.421116
(0,4) : 79.179796
(0,5) : 77.804751
(1,1) : 80.750294
(1,2) : 80.453704
(1,3) : 79.319770
(1,5) : 76.596054
(2,0) : 79.913136
(2,1) : 79.412759
(2,2) : 78.166196
(2,3) : 78.144711
(2,5) : 75.402429
(3,0) : 78.742684
(3,1) : 79.265508
(3,2) : 78.044502
(3,3) : 75.988556
(3,5) : 74.223686
(4,0) : 77.522286
(4,2) : 77.970644
(4,3) : 76.683162
(4,4) : 74.417093
(4,5) : 73.437582
(5,0) : 78.195903
(5,1) : 75.785286
(5,2) : 76.663035
(5,3) : 76.756298
(5,4) : 75.435015
(5,5) : 74.165949
```

Code Snippet of Reference Utilities of States for Policy Iteration



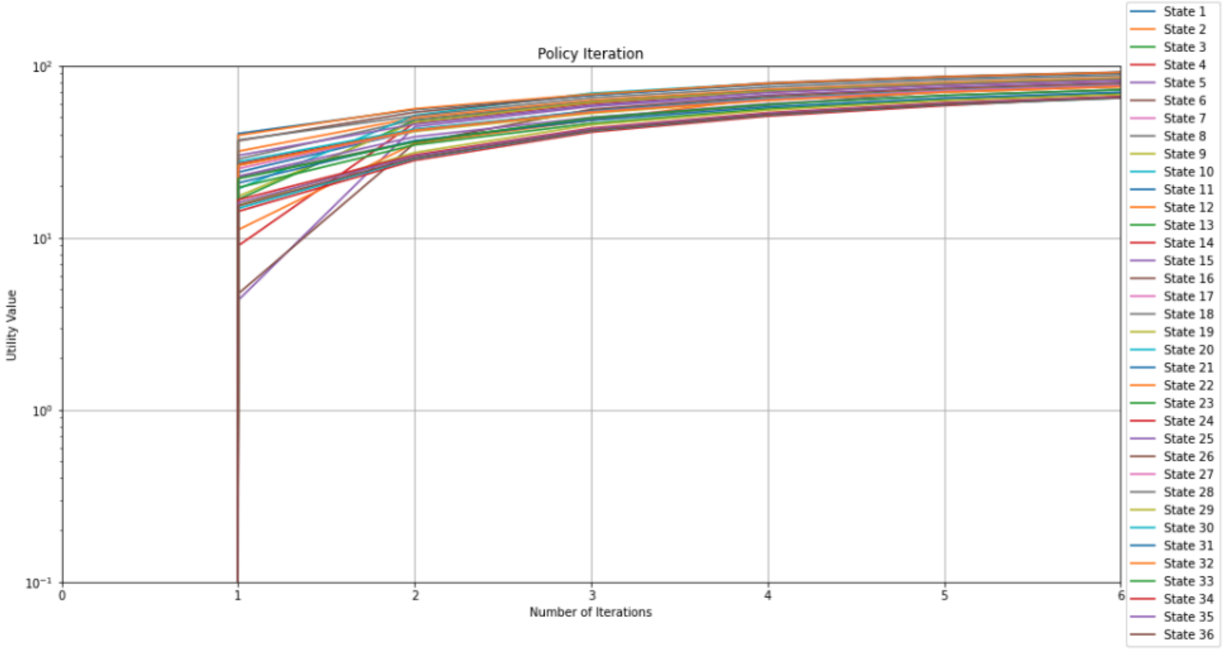
```
=========== UTILITIES IN GRID FORMAT ===========
        0        1        2        3        4        5
    +--------+--------+--------+--------+--------+--------+
0   | 84.867 | ██████ | 79.913 | 78.743 | 77.522 | 78.196 |
    +--------+--------+--------+--------+--------+--------+
1   | 83.261 | 80.750 | 79.413 | 79.266 | ██████ | 75.785 |
    +--------+--------+--------+--------+--------+--------+
2   | 81.816 | 80.454 | 78.166 | 78.045 | 77.971 | 76.663 |
    +--------+--------+--------+--------+--------+--------+
3   | 80.421 | 79.320 | 78.145 | 75.989 | 76.683 | 76.756 |
    +========+========+========+========+========+========+
4   | 79.180 | ██████ | ██████ | ██████ | 74.417 | 75.435 |
    +--------+--------+--------+--------+--------+--------+
5   | 77.805 | 76.596 | 75.402 | 74.224 | 73.438 | 74.166 |
    +--------+--------+--------+--------+--------+--------+
```

Code Snippet of Reference Utilities of States in Grid Format for Policy Iteration

Graph of Utility Estimates over Iterations (Policy Iteration)

The graph below illustrates the attainment of equilibrium in state utilities by the 7th iteration. Notably, states with a consistent utility value of 0 are designated as wall states.

Method Used	Policy Iteration
No. of Iterations	7



Graph of Utility Estimates over Iterations (Policy Iteration)

Bonus Question

Complicated Maze Environment

Before addressing how the convergence is influenced by the number of states and the complexity of the environment, I conducted several experiments.

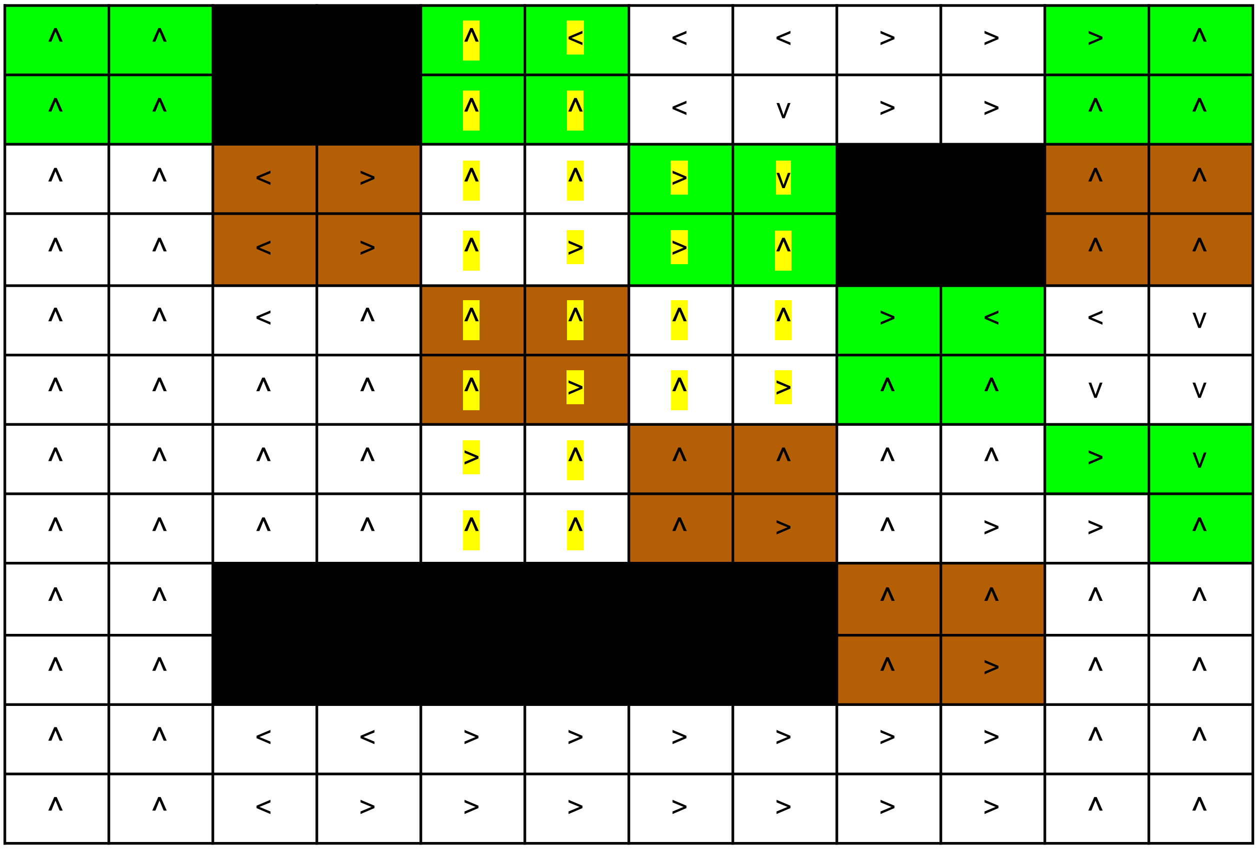
1. I augmented the number of states by enlarging the environment's scale.
2. I manipulated the count of states, including both Reward and Penalty states, within the Maze environment.

1. Augmentation of Environment Scale

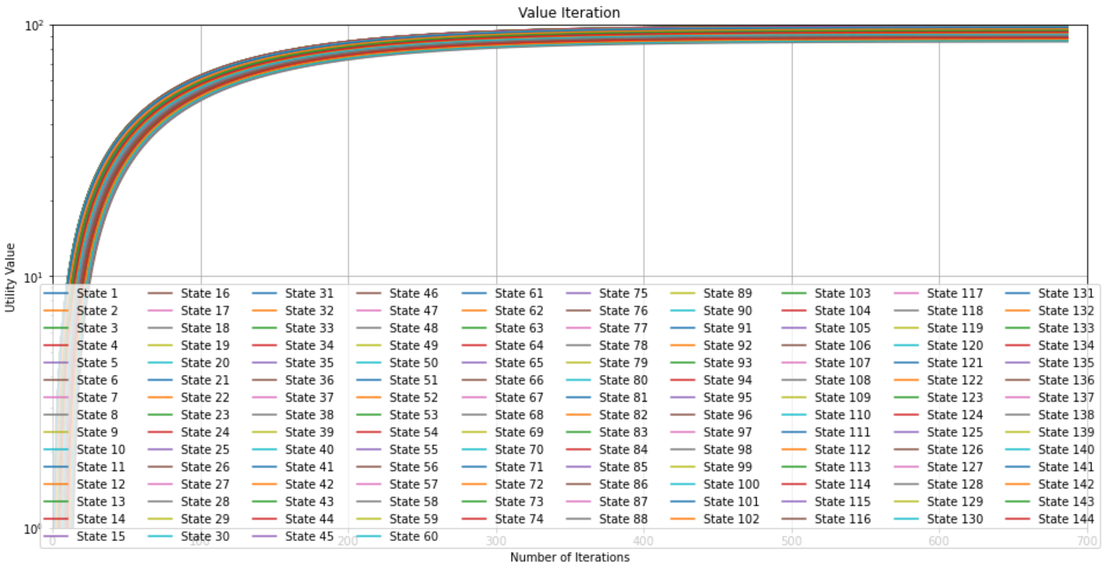
The scale is increased to 2. Hence, the total number of states is 144.

Optimal Policy Plot

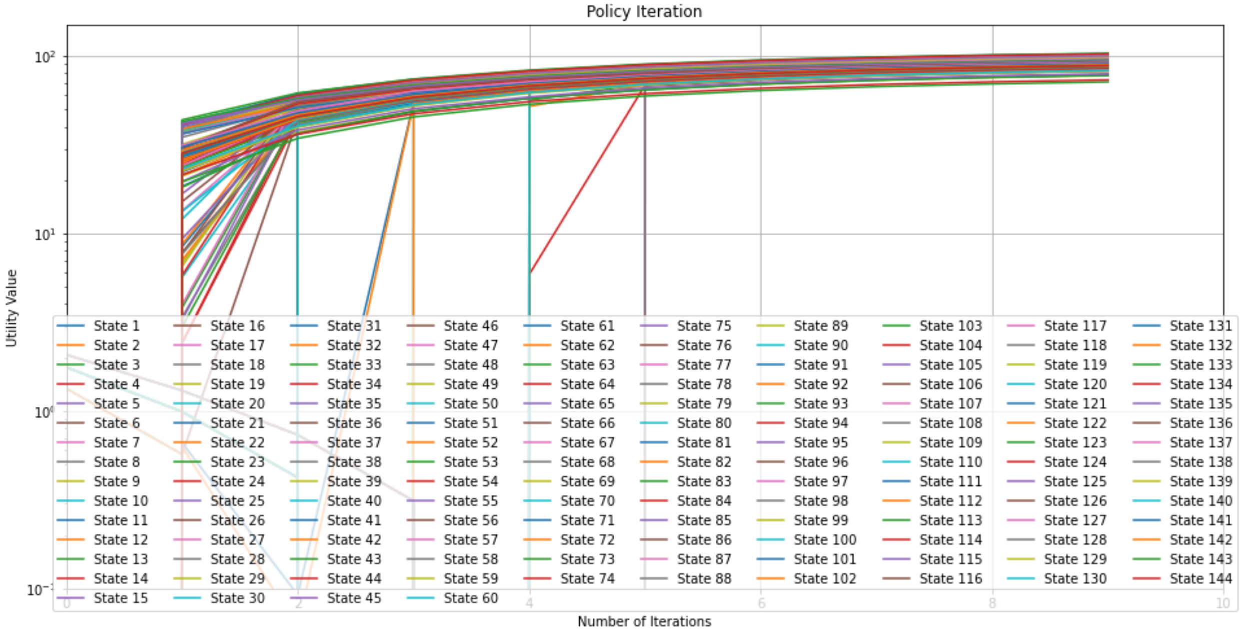
Starting Positions: (6,4), (6,5), (7,4), (7,5)



^	^			^	<	<	<	>	>	>	^
^	^			^	^	<	v	>	>	^	^
^	^	<	>	^	^	>	v			^	^
^	^	<	>	^	>	>	^			^	^
^	^	<	^	^	^	^	^	>	<	<	v
^	^	^	^	^	>	^	>	^	^	v	v
^	^	^	^	>	^	^	^	^	^	>	v
^	^	^	^	^	^	^	>	^	>	>	^
^	^							^	^	^	^
^	^							^	>	^	^
^	^	<	<	>	>	>	>	>	>	^	^
^	^	<	>	>	>	>	>	>	>	^	^



Method Used	Value Iteration
No. of Iterations	688
Convergence Criteria	0.0010101010101010112



Method Used	Policy Iteration
No. of Iterations	10

Observation with this experiment:

In the case of Value Iteration, despite the increase in the number of states from 31 to 124, the number of iterations remained constant.

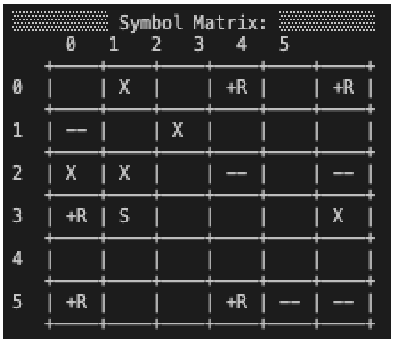
Conversely, with Policy Iteration, the number of iterations rose from 6 (as in Part 1) to 10. This suggests that the maze environment's complexity escalated for Policy Iteration, evidenced by the extended duration required for state utilities to stabilize compared to the original maze environment.

However, both Value Iteration and Policy Iteration algorithms successfully identified the optimal policy. Thus, the results indicate that augmenting the number of states does not necessarily hinder the agent from attaining the optimal policy.

2. Create Different Numbers of States (Reward States, Penalty States)

The rationale for conducting these test cases is to investigate whether variations in the number of unique states lead to significant impacts on the convergence rate. For instance, exploring the effects of altering the number of rewards from 6 to 5 provides insights into the potential impacts on convergence dynamics.

Case 1: Equal Number of States for Each Category

	**No. of Penalty States**	5
	No. of Reward States	5
	No. of Wall States	5
	No. of Start States	1
	Total No. of States	36

```
░░░░░░░░░░ Symbol Matrix: ░░░░░░░░░░
     0    1    2    3    4    5
   +----+----+----+----+----+----+
0  |    | X  |    | +R |    | +R |
   +----+----+----+----+----+----+
1  | -- |    | X  |    |    |    |
   +----+----+----+----+----+----+
2  | X  | X  |    | -- |    | -- |
   +----+----+----+----+----+----+
3  | +R | S  |    |    |    | X  |
   +----+----+----+----+----+----+
4  |    |    |    |    |    |    |
   +----+----+----+----+----+----+
5  | +R |    |    | +R | -- | -- |
   +----+----+----+----+----+----+
```

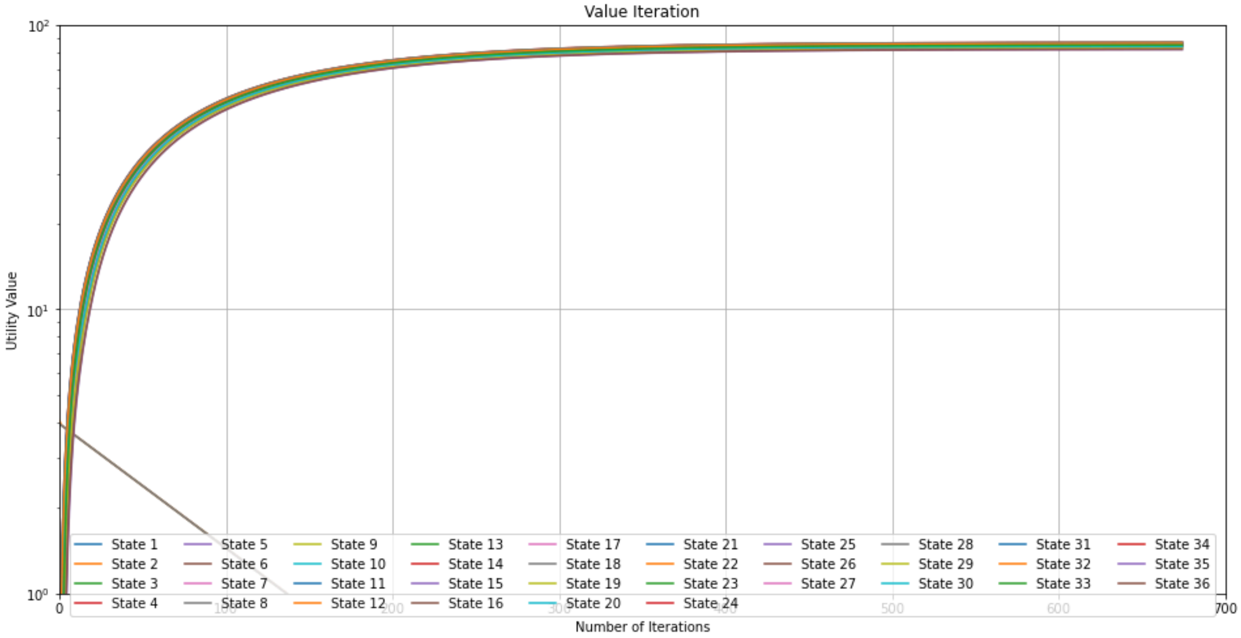
Value Iteration

Optimal Policy Plot

Starting Position: (3, 1)

	0	**1**	**2**	**3**	**4**	**5**
0	^	⬛	>	🟩 ^	>	🟩 ^
1	🟫 ^	>	⬛	^	^	^
2	⬛	⬛	v	🟫 ^	^	🟫 ^
3	🟩 ^	<	<	<	<	⬛
4	^	<	<	v	<	<
5	🟩 <	<	<	🟩 <	🟫 <	🟫 <

Graph of Utility Estimates over Iterations (Value Iteration)



Graph of Utility Estimates over Iterations (Value Iteration)

Method Used	Value Iteration
No. of Iterations	675
Convergence Criteria	0.0010101010101010112

Observation:

Based on the depicted graph, it becomes apparent that the values of certain utilities for the states approach negative values as they near the convergence criteria, instead of the anticipated positive values.

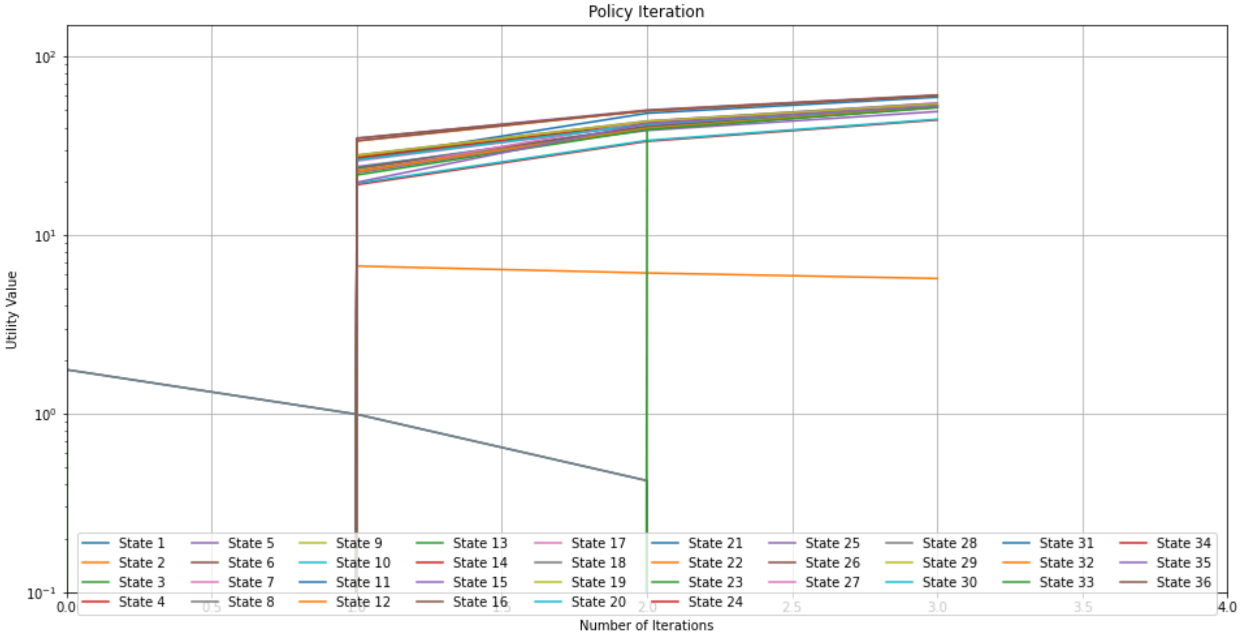
Policy Iteration

Optimal Policy Plot

Starting Position: (3, 1)

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

Graph of Utility Estimates over Iterations (Policy Iteration)



Graph of Utility Estimates over Iterations (Policy Iteration)

Method Used	Policy Iteration
No. of Iterations	4

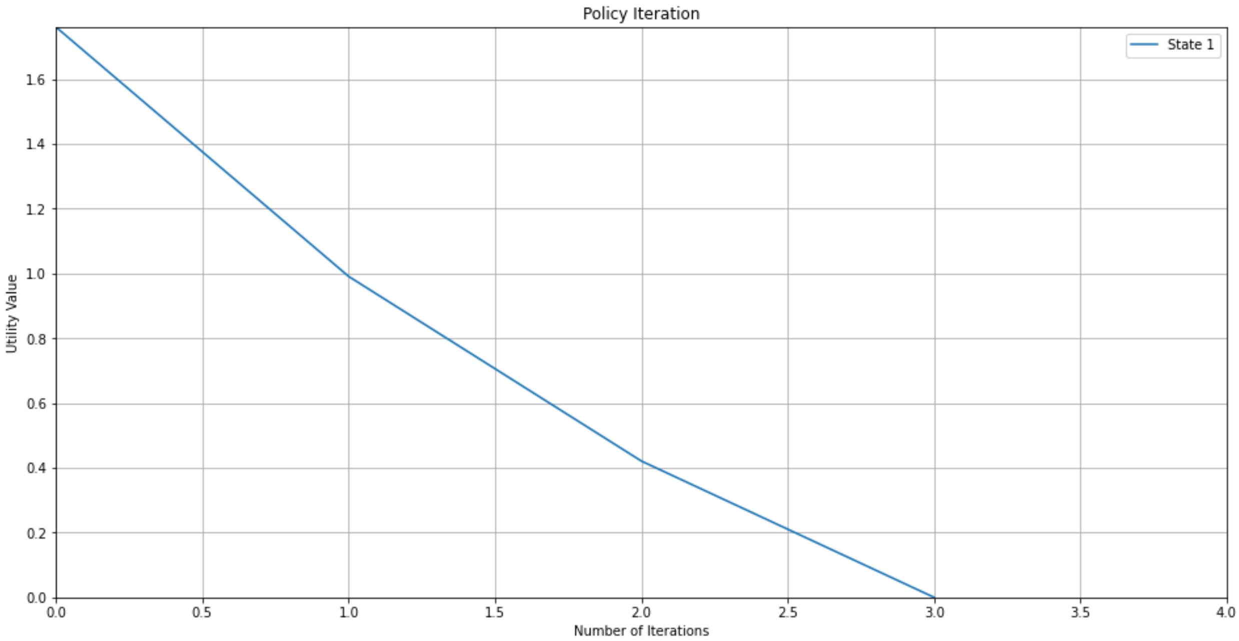
Observation:

In line with observations from Value Iteration, certain states exhibit negative values towards the end of the 8th iteration. These states stabilize their utility values earlier than those with positive utility values. Notably, the key disparity between Value Iteration and Policy Iteration lies in the behavior of state utilities: in Policy Iteration, as the algorithm progresses, state utilities initially holding negative values transition to positive values.

Key Insights:

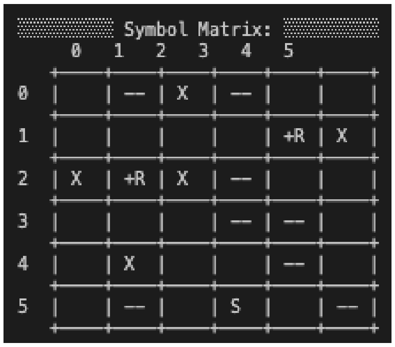
Several state utilities exhibit negative values due to the inability of these states to find a path to the reward state. For instance, State 1 is constrained by surrounding walls, rendering it devoid of any path to a reward state. Consequently, the only rewards it can accrue stem from penalty states and empty states within the confined area, leading to the manifestation of negative utility values.

Nevertheless, in both policy iteration and value iteration, despite starting from different points, the system successfully navigates to a reward state. Consequently, it attains the optimal policy, demonstrating its effectiveness regardless of the initial conditions.



Graph of Utility Estimates over Iterations for State 1 (Value Iteration)

Case 2: Change in Number of Penalty States and Reward 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	36

```
░░░░░░░░░░ Symbol Matrix: ░░░░░░░░░░
      0    1    2    3    4    5
    +----+----+----+----+----+----+
0   |    | -- | X  | -- |    |    |
    +----+----+----+----+----+----+
1   |    |    |    |    | +R | X  |
    +----+----+----+----+----+----+
2   | X  | +R | X  | -- |    |    |
    +----+----+----+----+----+----+
3   |    |    |    | -- | -- |    |
    +----+----+----+----+----+----+
4   |    | X  |    |    | -- |    |
    +----+----+----+----+----+----+
5   |    | -- |    | S  |    | -- |
    +----+----+----+----+----+----+
```

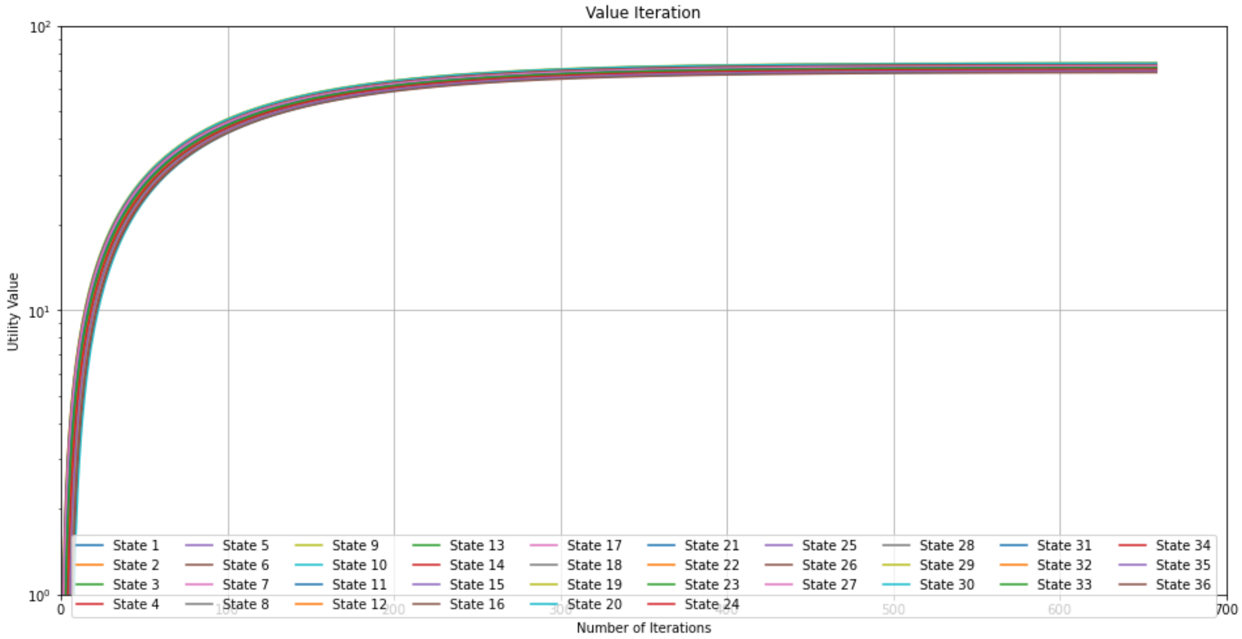
Value Iteration

Optimal Policy Plot

Starting Position: (5, 3)

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

Graph of Utility Estimates over Iterations (Value Iteration)



Graph of Utility Estimates over Iterations (Value Iteration)

Method Used	Value Iteration
No. of Iterations	659
Convergence Criteria	0.0010101010101010112

Observation:

In value iteration, regardless of adjustments made to factors such as the number of walls, penalties, or rewards, the number of iterations remains consistent. The iterations consistently cap at 688 or below until meeting the convergence criteria, suggesting a maximum iteration threshold for value iteration.

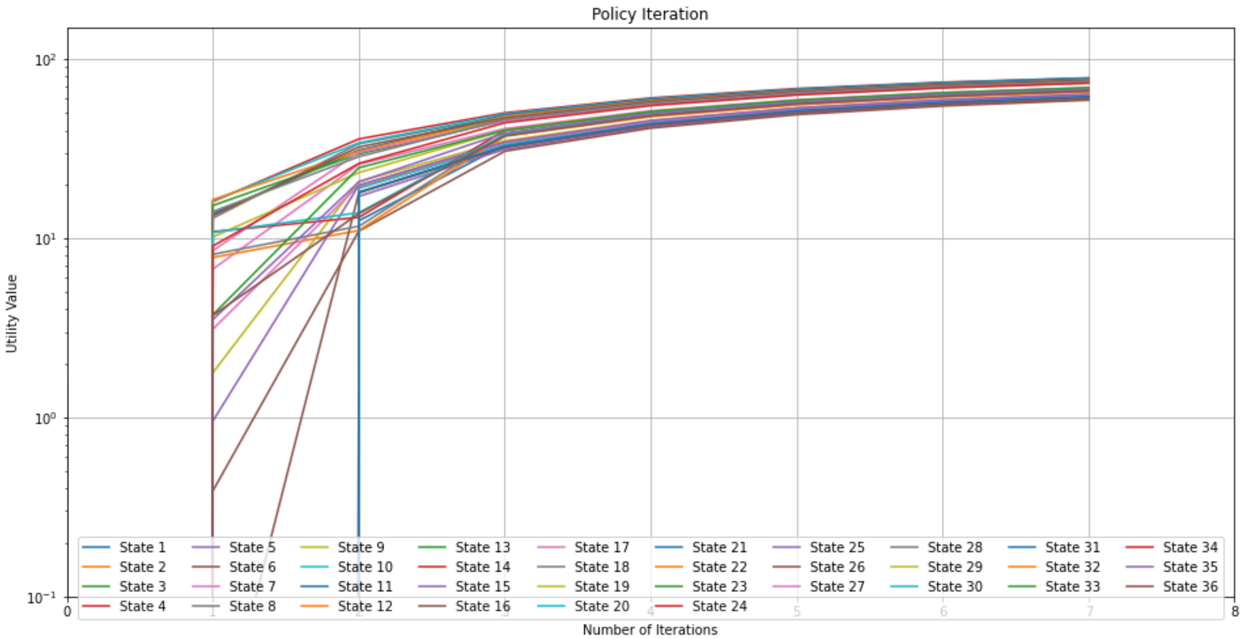
Policy Iteration

Optimal Policy Plot

Starting Position: (5, 3)

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

Graph of Utility Estimates over Iterations (Policy Iteration)



Graph of Utility Estimates over Iterations (Policy Iteration)

Method Used	Policy Iteration
No. of Iterations	8

Observation:

In Policy Iteration, the number of iterations has decreased from 8 in the 1st Case to 7 in the 2nd Case.

Conclusion

The findings derived from complex maze environments suggest that a well-structured maze inherently harbors an optimal policy for the agent to pursue, maximizing expected utility. However, if the start state is enclosed within walls and the reward states lie beyond those confines, the agent cannot attain the optimal policy. Remaining within these walls confines the agent to accumulating either negative or zero-value rewards, thus impeding any increase in utility value.