NANYANG TECHNOLOGICAL UNIVERSITY

COURSE CODE	**:**	**CZ4046**
COURSE NAME	**:**	**INTELLIGENT AGENTS**
NAME	**:**	**HTET NAING**
MATRICULATION NO.	**:**	**U1620683D**
COURSEWORK	**:**	**ASSIGNMENT 1**

Table of Contents

1. Description of Implemented Solutions

1.1 Brief Explanation of the Source Code Files

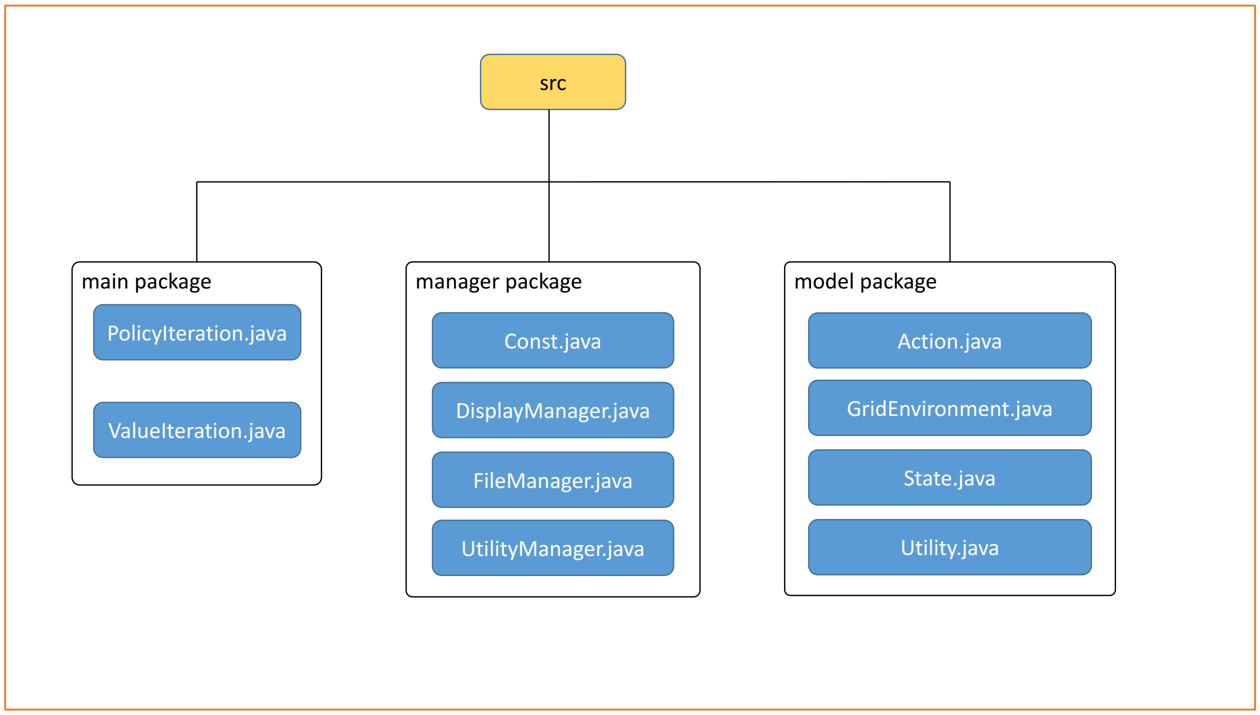


Figure 1.1: File structure of the source code

In this assignment, in order to find the optimal policy for the given maze environment, two algorithms, namely "Value Iteration" and "Policy Iteration" are implemented. The hierarchy of the implementation code is illustrated in the above figure.

The source code folder consists of three packages – *main*, *manger* and *model*. In the *main* package, the policy iteration and value iteration algorithms are implemented in their respective java files.

Under the *manager* package, there are four java files.

Const.java : stores all the constants used in the program such as Discount factor, Epsilon value, K value, different reward values and so on.

DisplayManager.java : is responsible for displaying all kind of console outputs that include grid environment, the parameter setup for the experiment, the total number of iterations required to find the best policy, the depiction of the optimal policy and the utilities of all states.

FileManager.java : helps save the estimated utility values of all states in an excel file after running the selected algorithm (either value iteration or policy iteration)

UtilityManager.java : is used for both calculating the best utility in a given state and estimating the next utility estimate in policy iteration.

In the *model* package, the following four java files are implemented.

Action.java : is an *enum* for representing different types of actions – Up, Down, Left, Right.

GridEnvironment.java : represents the maze environment with respective conditions in each state as shown in the assignment.

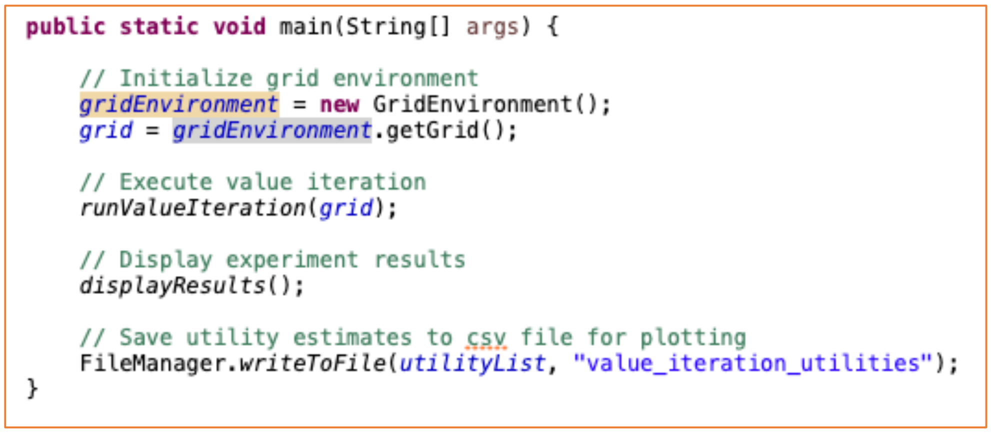
State.java : represents the state for a single grid; it could be reward values or wall.

Utility.java : is a *class* that represents the utility of a state given a certain action (i.e. Up, Down, Left, Right).

2. Implementation of Value Iteration

When the *ValueIteration.java* is run, the followings occurs:

1) The maze environment is initialized with respective reward values.
2) "Value Iteration" method is executed in order to find the optimal policy.
3) Both the experiment setup and experiment results after running the "Value Iteration" algorithm are displayed as the console output.
4) Utilities of all states that are recorded for each iteration is saved in an excel file (.csv) so that all these utility estimates can be plotted as a function of the number of iterations.

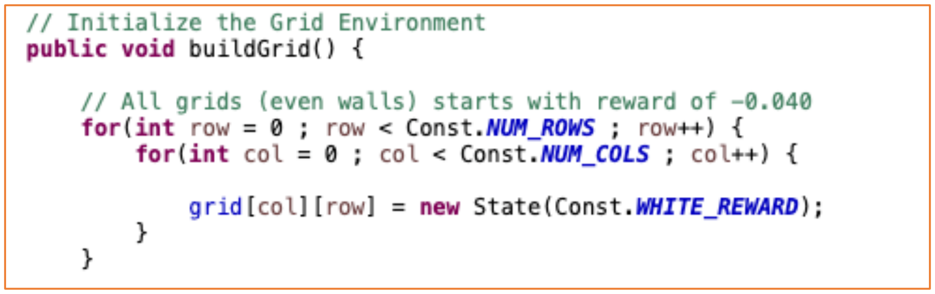


```java
public static void main(String[] args) {

    // Initialize grid environment
    gridEnvironment = new GridEnvironment();
    grid = gridEnvironment.getGrid();

    // Execute value iteration
    runValueIteration(grid);

    // Display experiment results
    displayResults();

    // Save utility estimates to csv file for plotting
    FileManager.writeToFile(utilityList, "value_iteration_utilities");
}
```

Figure 2.1: The main method for running "Value Iteration" algorithm

2.1 Initialization of the Maze Environment with Reward Values

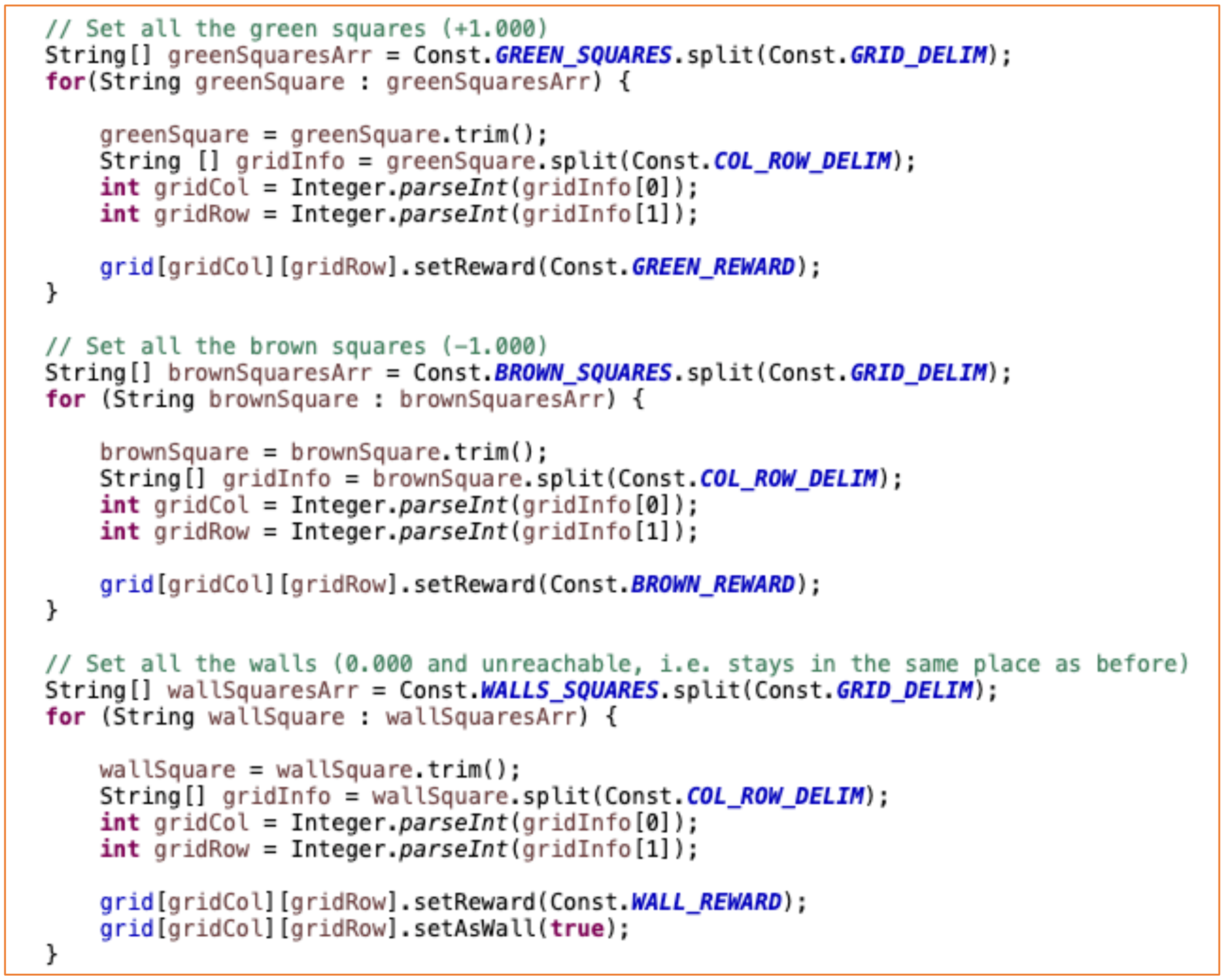
When an instance of *GridEnvironment* is created, the maze environment is constructed by invoking the *buildGrid()* method. Firstly, all the grids are filled up with new states with the default reward value of negative 0.04.



```java
// Initialize the Grid Environment
public void buildGrid() {

    // All grids (even walls) starts with reward of -0.040
    for(int row = 0 ; row < Const.NUM_ROWS ; row++) {
        for(int col = 0 ; col < Const.NUM_COLS ; col++) {

            grid[col][row] = new State(Const.WHITE_REWARD);
        }
    }
```

Figure 2.1.1: The initialization of the maze environment with default reward values

Then the coordinates for green square (+1.0), brown square (-1.0) and wall (0.0) are taken from *Const.java* and their reward values are also set in the corresponding states. Hence, a maze environment with the respective values as defined in the assignment is created with the implementation of the following code.



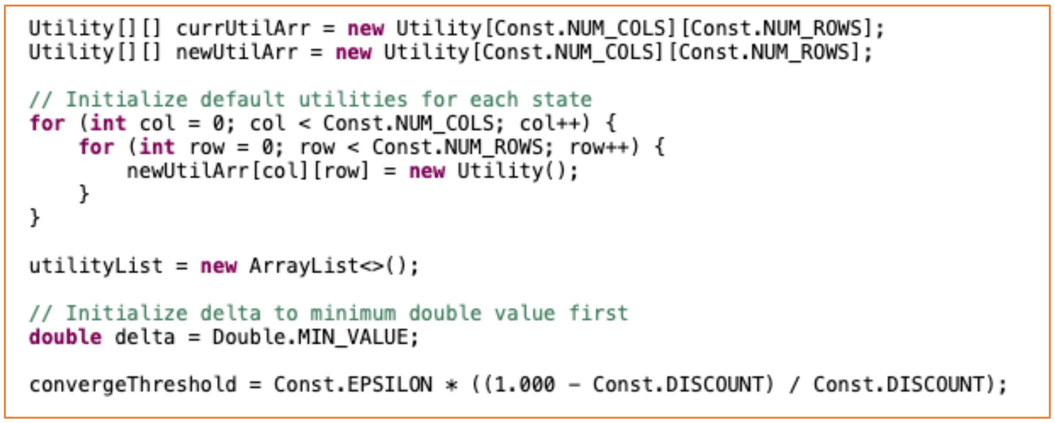
```
// Set all the green squares (+1.000)
String[] greenSquaresArr = Const.GREEN_SQUARES.split(Const.GRID_DELIM);
for(String greenSquare : greenSquaresArr) {

    greenSquare = greenSquare.trim();
    String [] gridInfo = greenSquare.split(Const.COL_ROW_DELIM);
    int gridCol = Integer.parseInt(gridInfo[0]);
    int gridRow = Integer.parseInt(gridInfo[1]);

    grid[gridCol][gridRow].setReward(Const.GREEN_REWARD);
}

// Set all the brown squares (-1.000)
String[] brownSquaresArr = Const.BROWN_SQUARES.split(Const.GRID_DELIM);
for (String brownSquare : brownSquaresArr) {

    brownSquare = brownSquare.trim();
    String[] gridInfo = brownSquare.split(Const.COL_ROW_DELIM);
    int gridCol = Integer.parseInt(gridInfo[0]);
    int gridRow = Integer.parseInt(gridInfo[1]);

    grid[gridCol][gridRow].setReward(Const.BROWN_REWARD);
}

// Set all the walls (0.000 and unreachable, i.e. stays in the same place as before)
String[] wallSquaresArr = Const.WALLS_SQUARES.split(Const.GRID_DELIM);
for (String wallSquare : wallSquaresArr) {

    wallSquare = wallSquare.trim();
    String[] gridInfo = wallSquare.split(Const.COL_ROW_DELIM);
    int gridCol = Integer.parseInt(gridInfo[0]);
    int gridRow = Integer.parseInt(gridInfo[1]);

    grid[gridCol][gridRow].setReward(Const.WALL_REWARD);
    grid[gridCol][gridRow].setAsWall(true);
}
```

Figure 2.1.2: The initialization of the maze environment with respective reward values for green square, brown square and wall

2.2 Value Iteration for Finding the Optimal Policy

Firstly, two vectors of utilities are created to store the current utilities of states and the updated utilities of states after each iteration. All the states will be initialized with utility values of 0.0 and unknown action declared as *null*.

Secondly, a list of utility is created to help record the estimated utility values for all states after each iteration. The list will be stored in an excel file after running the algorithm so that all these utility estimates can be plotted as a function of the number of iterations. The *delta* is initialized as the default *minimum double* value and it will be updated subsequently when the new updated delta is greater than the default value. The *convergence threshold* is calculated and used as the terminating condition to stop the Value Iteration algorithm. The implementation of the above can be seen in the following code.



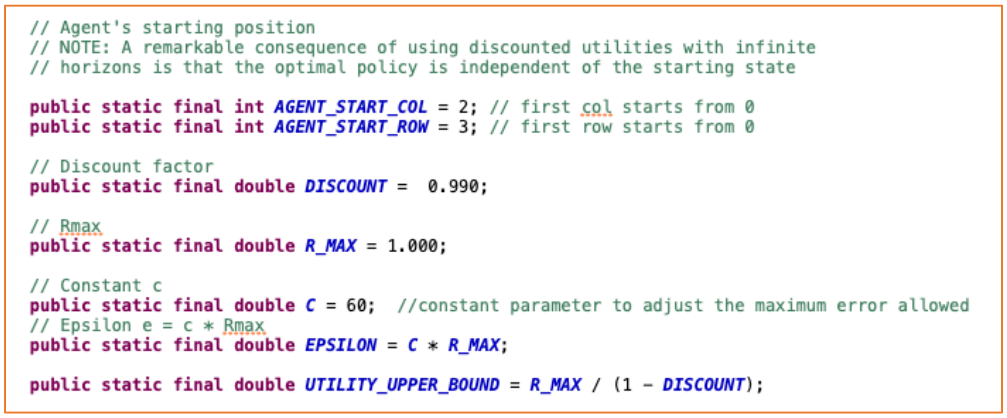
```java
Utility[][] currUtilArr = new Utility[Const.NUM_COLS][Const.NUM_ROWS];
Utility[][] newUtilArr = new Utility[Const.NUM_COLS][Const.NUM_ROWS];

// Initialize default utilities for each state
for (int col = 0; col < Const.NUM_COLS; col++) {
    for (int row = 0; row < Const.NUM_ROWS; row++) {
        newUtilArr[col][row] = new Utility();
    }
}

utilityList = new ArrayList<>();

// Initialize delta to minimum double value first
double delta = Double.MIN_VALUE;

convergeThreshold = Const.EPSILON * ((1.000 - Const.DISCOUNT) / Const.DISCOUNT);
```

Figure 2.2.1: The initialization of variables used in executing Value Iteration algorithm

The *Epsilon* value in the above figure represent the maximum error allowed in the utility of any state. The epsilon can be calculated by multiplying the arbitrary constant c and maximum reward value as follows:

*Epsilon = c * Rmax*

Based on different discount factors and different Epsilon values, the results of the optimal policy can vary. The outcome of the experiment when using different values of constant *"c"* will be analyzed in details later. All the constants used in this assignment such as Epsilon value, "c" constant, the maximum reward value, the Discount factor, the upper bound for the estimated utility of a state are also defined in *Const.java*.

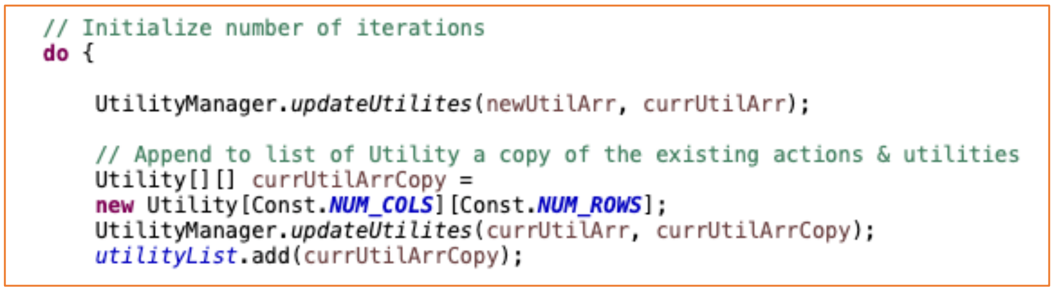


```java
// Agent's starting position
// NOTE: A remarkable consequence of using discounted utilities with infinite
// horizons is that the optimal policy is independent of the starting state

public static final int AGENT_START_COL = 2; // first col starts from 0
public static final int AGENT_START_ROW = 3; // first row starts from 0

// Discount factor
public static final double DISCOUNT =  0.990;

// Rmax
public static final double R_MAX = 1.000;

// Constant c
public static final double C = 60;  //constant parameter to adjust the maximum error allowed
// Epsilon e = c * Rmax
public static final double EPSILON = C * R_MAX;

public static final double UTILITY_UPPER_BOUND = R_MAX / (1 - DISCOUNT);
```

Figure 2.2.2: The initialization of constants used in executing Value Iteration algorithm

In this assignment, since there is no *fixed time N* defined for decision making, the nature of the problem is considered as having infinite horizons. "Infinite horizon" does not mean that all state sequences are infinite but it just means that there is no fixed deadline for finding the optimal policy for decision making. The remarkable consequence of using discounted utilities with infinite horizons is that the optimal policy is independent of the starting state. Therefore, the agent's starting position as in the given maze environment does not have any impact on the final optimal policy.

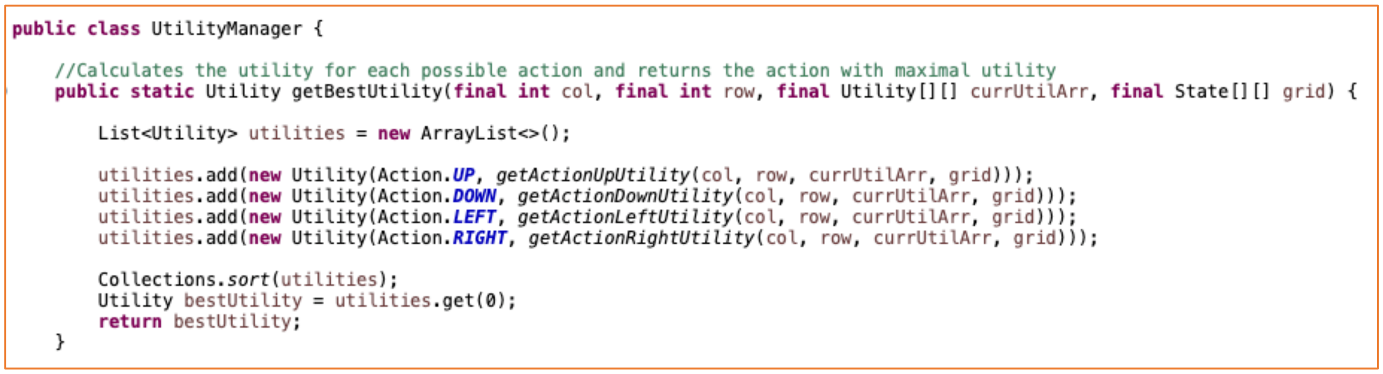
At the start of every iteration, the current utility vector will be updated with the best utility values achieved so far. Then this current utility vector that consists of the estimated utility values for all states will be added to the *utilityList* in order to keep track of utilities of all the states at each iteration for plotting the utility estimates as a function of number of iterations. They are implemented as follows:



```
// Initialize number of iterations
do {

    UtilityManager.updateUtilites(newUtilArr, currUtilArr);

    // Append to list of Utility a copy of the existing actions & utilities
    Utility[][] currUtilArrCopy =
    new Utility[Const.NUM_COLS][Const.NUM_ROWS];
    UtilityManager.updateUtilites(currUtilArr, currUtilArrCopy);
    utilityList.add(currUtilArrCopy);
```

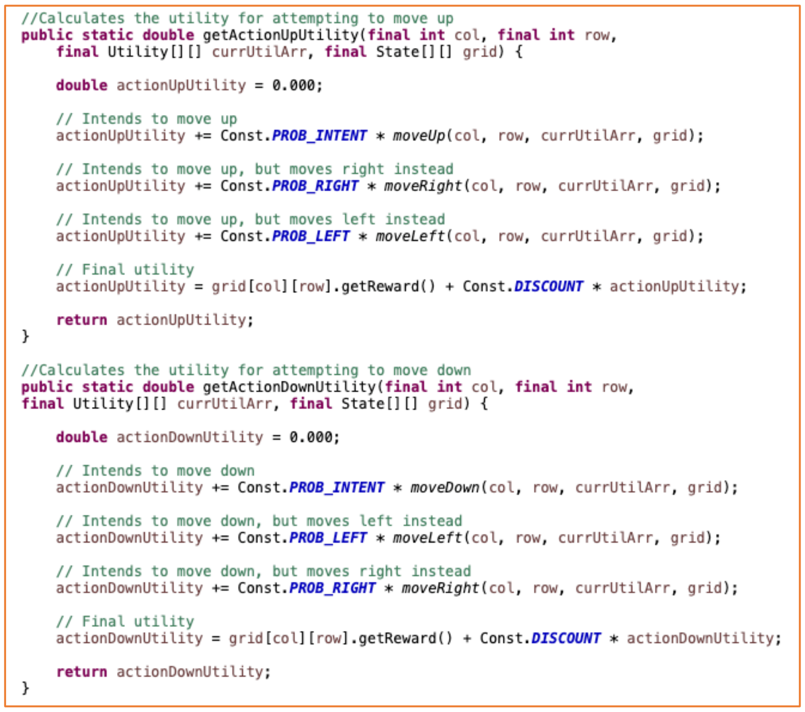
Figure 2.2.3: The update of current utility vector with the new best utilities

After that, during the current iteration, the best utility for each state is calculated. This is achieved by choosing the action that maximizes the expected utility of the subsequent state. The *UtilityManger.java* helps calculate the best utility derived from the selection of the best action. A list is created to hold the utilities produced by taking a certain action. Given a specific type of action, the corresponding utility is calculated and added to the list. The list is sorted in descending order based on its utility value. Thus, the topmost item in the list has the best utility. It is used to update the new best utility value for a given state. It is implemented as follows:



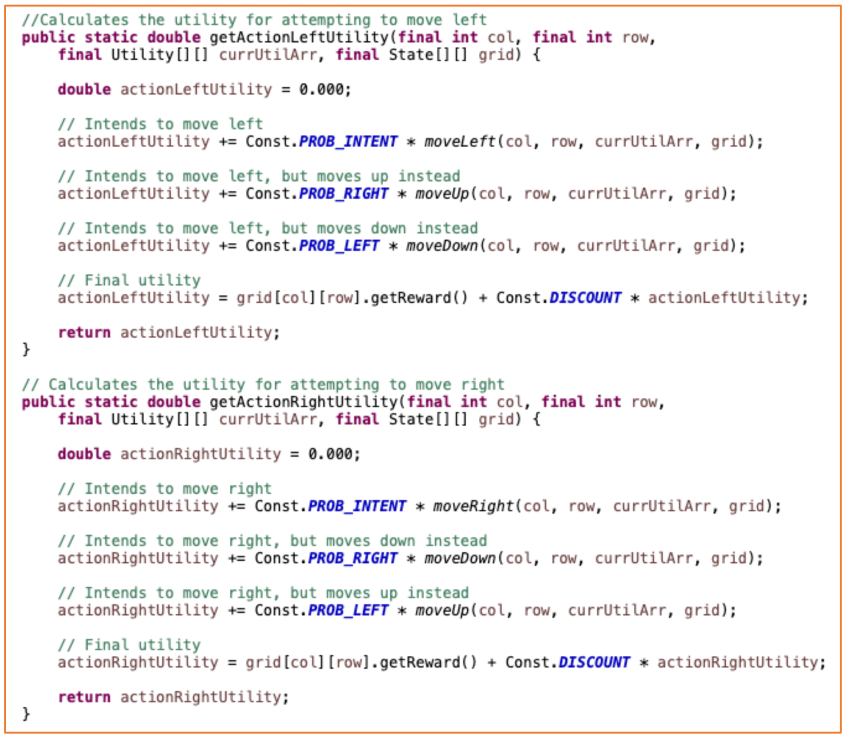
```java
public class UtilityManager {

    //Calculates the utility for each possible action and returns the action with maximal utility
    public static Utility getBestUtility(final int col, final int row, final Utility[][] currUtilArr, final State[][] grid) {

        List<Utility> utilities = new ArrayList<>();

        utilities.add(new Utility(Action.UP, getActionUpUtility(col, row, currUtilArr, grid)));
        utilities.add(new Utility(Action.DOWN, getActionDownUtility(col, row, currUtilArr, grid)));
        utilities.add(new Utility(Action.LEFT, getActionLeftUtility(col, row, currUtilArr, grid)));
        utilities.add(new Utility(Action.RIGHT, getActionRightUtility(col, row, currUtilArr, grid)));

        Collections.sort(utilities);
        Utility bestUtility = utilities.get(0);
        return bestUtility;
    }
```

Figure 2.2.4: The update of current utility vector with the new best utilities



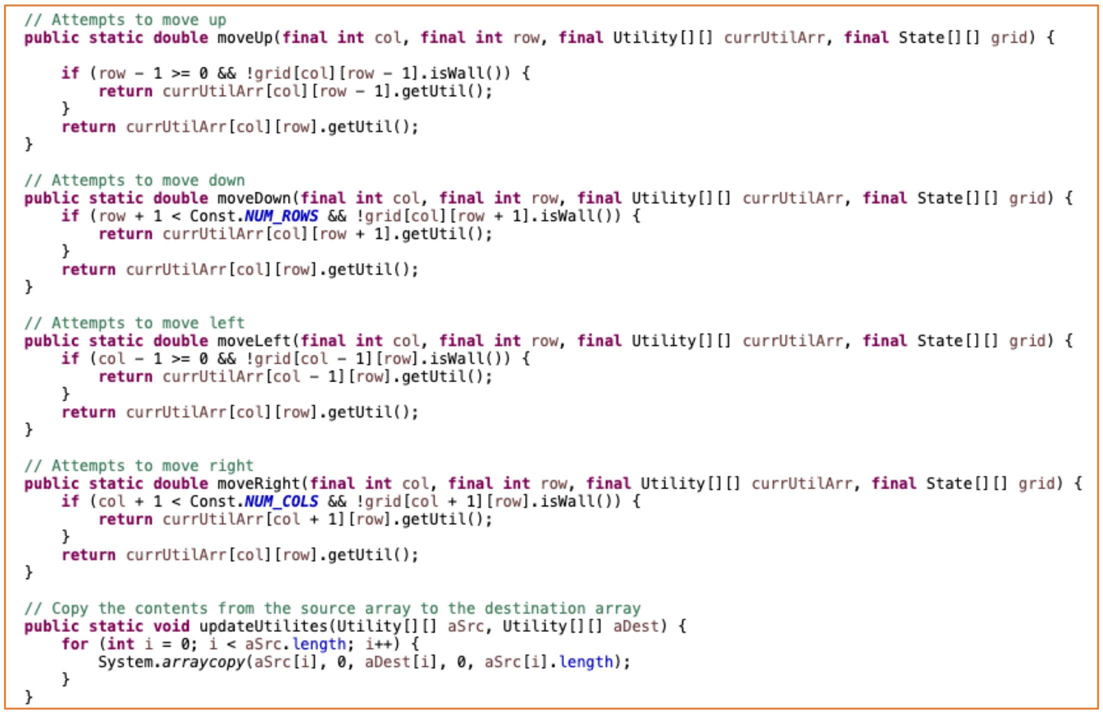
```java
//Calculates the utility for attempting to move up
public static double getActionUpUtility(final int col, final int row,
    final Utility[][] currUtilArr, final State[][] grid) {

    double actionUpUtility = 0.000;

    // Intends to move up
    actionUpUtility += Const.PROB_INTENT * moveUp(col, row, currUtilArr, grid);

    // Intends to move up, but moves right instead
    actionUpUtility += Const.PROB_RIGHT * moveRight(col, row, currUtilArr, grid);

    // Intends to move up, but moves left instead
    actionUpUtility += Const.PROB_LEFT * moveLeft(col, row, currUtilArr, grid);

    // Final utility
    actionUpUtility = grid[col][row].getReward() + Const.DISCOUNT * actionUpUtility;

    return actionUpUtility;
}

//Calculates the utility for attempting to move down
public static double getActionDownUtility(final int col, final int row,
final Utility[][] currUtilArr, final State[][] grid) {

    double actionDownUtility = 0.000;

    // Intends to move down
    actionDownUtility += Const.PROB_INTENT * moveDown(col, row, currUtilArr, grid);

    // Intends to move down, but moves left instead
    actionDownUtility += Const.PROB_LEFT * moveLeft(col, row, currUtilArr, grid);

    // Intends to move down, but moves right instead
    actionDownUtility += Const.PROB_RIGHT * moveRight(col, row, currUtilArr, grid);

    // Final utility
    actionDownUtility = grid[col][row].getReward() + Const.DISCOUNT * actionDownUtility;

    return actionDownUtility;
}
```

Figure 2.2.5: Calculating the utility for moving towards Up and Down direction



```java
//Calculates the utility for attempting to move left
public static double getActionLeftUtility(final int col, final int row,
    final Utility[][] currUtilArr, final State[][] grid) {

    double actionLeftUtility = 0.000;

    // Intends to move left
    actionLeftUtility += Const.PROB_INTENT * moveLeft(col, row, currUtilArr, grid);

    // Intends to move left, but moves up instead
    actionLeftUtility += Const.PROB_RIGHT * moveUp(col, row, currUtilArr, grid);

    // Intends to move left, but moves down instead
    actionLeftUtility += Const.PROB_LEFT * moveDown(col, row, currUtilArr, grid);

    // Final utility
    actionLeftUtility = grid[col][row].getReward() + Const.DISCOUNT * actionLeftUtility;

    return actionLeftUtility;
}

// Calculates the utility for attempting to move right
public static double getActionRightUtility(final int col, final int row,
    final Utility[][] currUtilArr, final State[][] grid) {

    double actionRightUtility = 0.000;

    // Intends to move right
    actionRightUtility += Const.PROB_INTENT * moveRight(col, row, currUtilArr, grid);

    // Intends to move right, but moves down instead
    actionRightUtility += Const.PROB_RIGHT * moveDown(col, row, currUtilArr, grid);

    // Intends to move right, but moves up instead
    actionRightUtility += Const.PROB_LEFT * moveUp(col, row, currUtilArr, grid);

    // Final utility
    actionRightUtility = grid[col][row].getReward() + Const.DISCOUNT * actionRightUtility;

    return actionRightUtility;
}
```

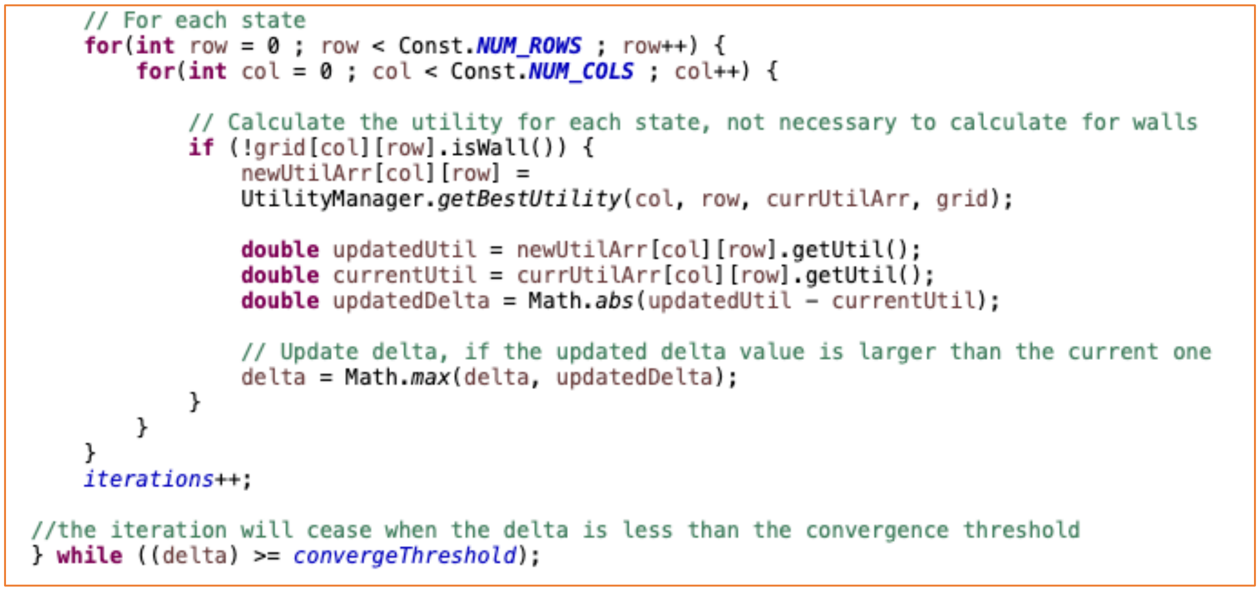
Figure 2.2.6: Calculating the utility for moving towards Left and Right direction



```java
// Attempts to move up
public static double moveUp(final int col, final int row, final Utility[][] currUtilArr, final State[][] grid) {

    if (row - 1 >= 0 && !grid[col][row - 1].isWall()) {
        return currUtilArr[col][row - 1].getUtil();
    }
    return currUtilArr[col][row].getUtil();
}

// Attempts to move down
public static double moveDown(final int col, final int row, final Utility[][] currUtilArr, final State[][] grid) {
    if (row + 1 < Const.NUM_ROWS && !grid[col][row + 1].isWall()) {
        return currUtilArr[col][row + 1].getUtil();
    }
    return currUtilArr[col][row].getUtil();
}

// Attempts to move left
public static double moveLeft(final int col, final int row, final Utility[][] currUtilArr, final State[][] grid) {
    if (col - 1 >= 0 && !grid[col - 1][row].isWall()) {
        return currUtilArr[col - 1][row].getUtil();
    }
    return currUtilArr[col][row].getUtil();
}

// Attempts to move right
public static double moveRight(final int col, final int row, final Utility[][] currUtilArr, final State[][] grid) {
    if (col + 1 < Const.NUM_COLS && !grid[col + 1][row].isWall()) {
        return currUtilArr[col + 1][row].getUtil();
    }
    return currUtilArr[col][row].getUtil();
}

// Copy the contents from the source array to the destination array
public static void updateUtilites(Utility[][] aSrc, Utility[][] aDest) {
    for (int i = 0; i < aSrc.length; i++) {
        System.arraycopy(aSrc[i], 0, aDest[i], 0, aSrc[i].length);
    }
}
```

Figure 2.2.7: Calculating the utility for taking Up, Down, Left and Right action

For each state, the new delta that is the difference between the new best utility and the current utility is calculated. Then if the new delta is greater than the current delta value so far, the current one will be updated with the new delta value. The above step will repeat for all states after which the program will check if the current delta value is less than the convergence threshold defined in *Figure 2.2.1*. If the delta is smaller than the threshold, it has reached the terminating condition. It also means that the utility of any state in the maze environment no longer changes by a lot as compared to the maximum error allowed (*Epsilon*). Otherwise, the algorithm will proceed to the next iteration to repeat the same procedure until the delta value becomes sufficiently small.



```
        // For each state
        for(int row = 0 ; row < Const.NUM_ROWS ; row++) {
            for(int col = 0 ; col < Const.NUM_COLS ; col++) {

                // Calculate the utility for each state, not necessary to calculate for walls
                if (!grid[col][row].isWall()) {
                    newUtilArr[col][row] =
                    UtilityManager.getBestUtility(col, row, currUtilArr, grid);

                    double updatedUtil = newUtilArr[col][row].getUtil();
                    double currentUtil = currUtilArr[col][row].getUtil();
                    double updatedDelta = Math.abs(updatedUtil - currentUtil);

                    // Update delta, if the updated delta value is larger than the current one
                    delta = Math.max(delta, updatedDelta);
                }
            }
        }
        iterations++;

    //the iteration will cease when the delta is less than the convergence threshold
    } while ((delta) >= convergeThreshold);
```

Figure 2.2.8: The partial implementation of Value Iteration algorithm

2.3 Plot of Optimal Policy and its Analysis with different constant "c" values

The experiment of the Value Iteration algorithm is performed based on the following constants:

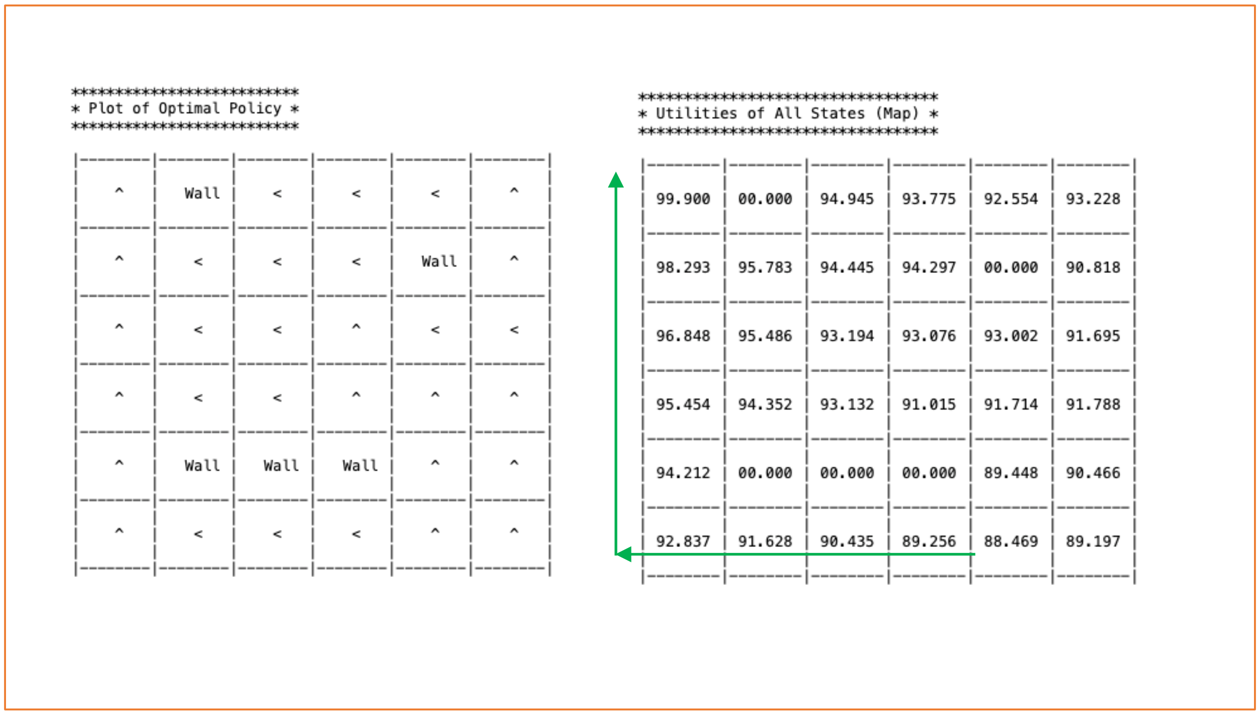
Discount Factor	:	0.99
Utility Upper Bound	:	100.00
Max Reward (Rmax)	:	+1.0
Epsilon	:	c * Rmax
Convergence threshold	:	Epsilon (1 – gamma) / gamma

where: *Epsilon* is the maximum error allowed in the utility of any state and

gamma is the discount factor

It should be noted that by setting the different values for "c", both the number of iterations required to find the optimal policy and the estimated utilities of all states will be different.

When the constant "c" is set to 0.1, the Epsilon value will become 0.1 and the convergence threshold is calculated to be 0.00101. After 688 iterations, the following optimal policy is achieved with the following utilities for all states.



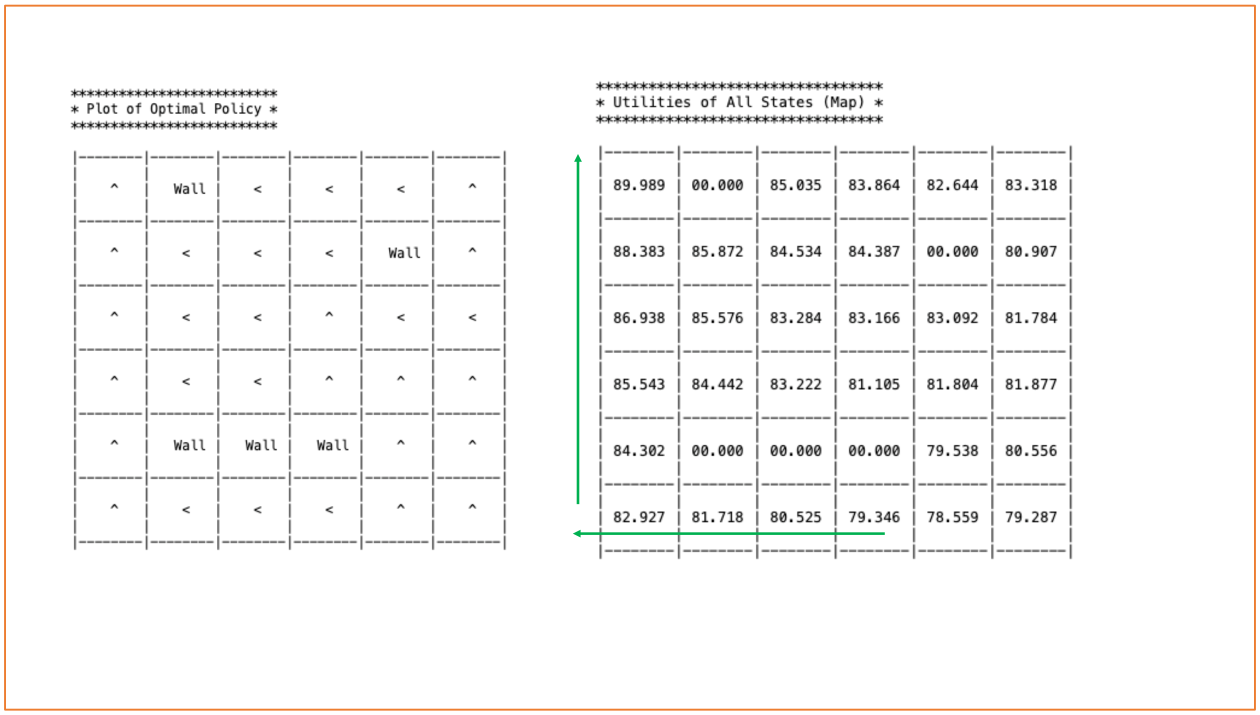
```
***************************
* Plot of Optimal Policy *
***************************

|--------|--------|--------|--------|--------|--------|
|   ^    |  Wall  |   <    |   <    |   <    |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   <    |   <    |  Wall  |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   <    |   ^    |   <    |   <    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   <    |   ^    |   ^    |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |  Wall  |  Wall  |  Wall  |   ^    |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   <    |   <    |   ^    |   ^    |
|--------|--------|--------|--------|--------|--------|
```

```
**********************************
* Utilities of All States (Map) *
**********************************

|--------|--------|--------|--------|--------|--------|
| 99.900 | 00.000 | 94.945 | 93.775 | 92.554 | 93.228 |
|--------|--------|--------|--------|--------|--------|
| 98.293 | 95.783 | 94.445 | 94.297 | 00.000 | 90.818 |
|--------|--------|--------|--------|--------|--------|
| 96.848 | 95.486 | 93.194 | 93.076 | 93.002 | 91.695 |
|--------|--------|--------|--------|--------|--------|
| 95.454 | 94.352 | 93.132 | 91.015 | 91.714 | 91.788 |
|--------|--------|--------|--------|--------|--------|
| 94.212 | 00.000 | 00.000 | 00.000 | 89.448 | 90.466 |
|--------|--------|--------|--------|--------|--------|
| 92.837 | 91.628 | 90.435 | 89.256 | 88.469 | 89.197 |
|--------|--------|--------|--------|--------|--------|
```

Figure 2.3.1: The plot of optimal policy when the constant "c" is set to 0.1

As indicated by the two green arrows in the above figure, it can clearly be seen that at every state, the algorithm "Value Iteration" chooses the best action that maximizes the utility of the subsequent states.

When the constant "c" is set to 10, the Epsilon value will become 10 and the convergence threshold is calculated to be 0.10101. After 230 iterations, the following optimal policy is achieved with the following utilities of all states.



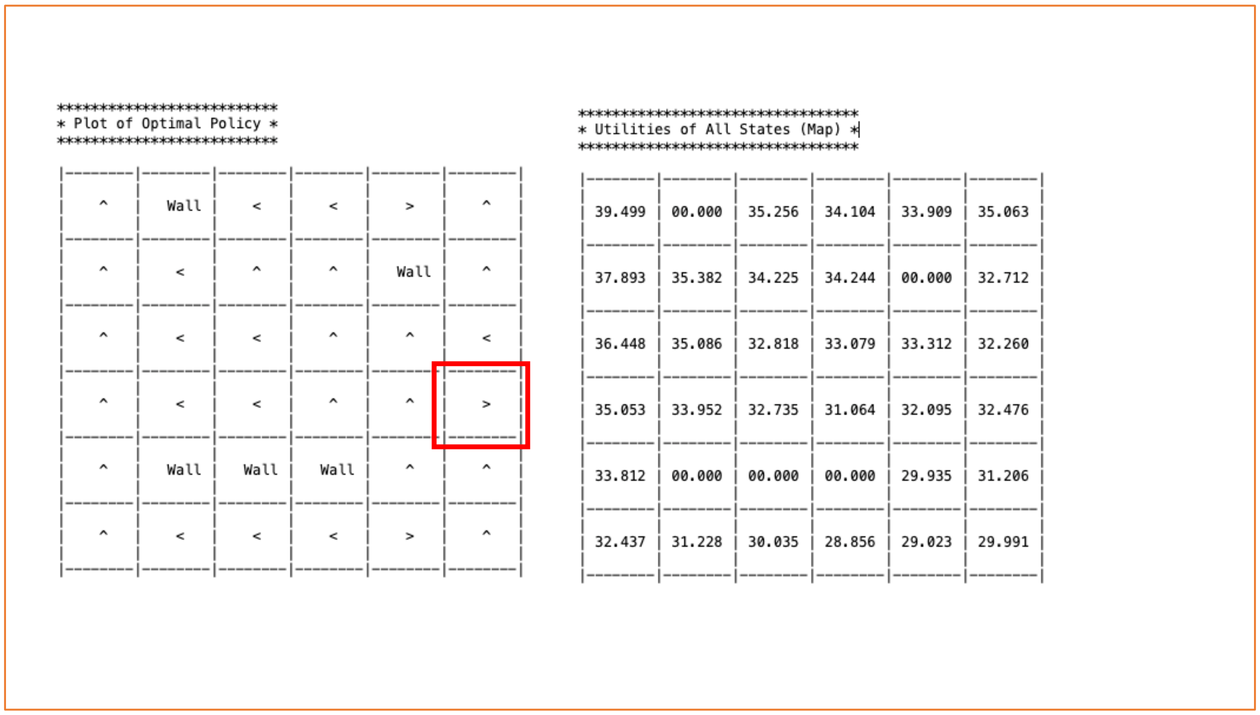
```
**************************
* Plot of Optimal Policy *
**************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |   <    |   <    |   <    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |  Wall  |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   <    |   <    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |  Wall  |  Wall  |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

```
*********************************
* Utilities of All States (Map) *
*********************************

|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 89.989  | 00.000  | 85.035  | 83.864  | 82.644  | 83.318  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 88.383  | 85.872  | 84.534  | 84.387  | 00.000  | 80.907  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 86.938  | 85.576  | 83.284  | 83.166  | 83.092  | 81.784  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 85.543  | 84.442  | 83.222  | 81.105  | 81.804  | 81.877  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 84.302  | 00.000  | 00.000  | 00.000  | 79.538  | 80.556  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 82.927  | 81.718  | 80.525  | 79.346  | 78.559  | 79.287  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
```

Figure 2.3.2: The plot of optimal policy when the constant "c" is set to 10

It should be observed that when c is set to 10, although the maximum error allowed (*Epsilon*) becomes higher (i.e. 10), the exact same optimal policy is achieved as compared to the result when c is set to 0.1.

When c is increased to 60, the optimal policy is no longer the same. It may be due to the fact that the maximum error allowed is also increased to 60 and the agent becomes prone to making mistakes in choosing its optimal decision. Moreover, although the number of iterations required to find the optimal policy is reduced to 51, the quality of the decision making is compromised. As highlighted in the red rectangle below, the agent chooses to go towards the wall instead of taking the action that maximizes the utilities of its subsequent states. It should move towards "Up" direction as shown in *Figure 2.3.1* and *Figure 2.3.2*.



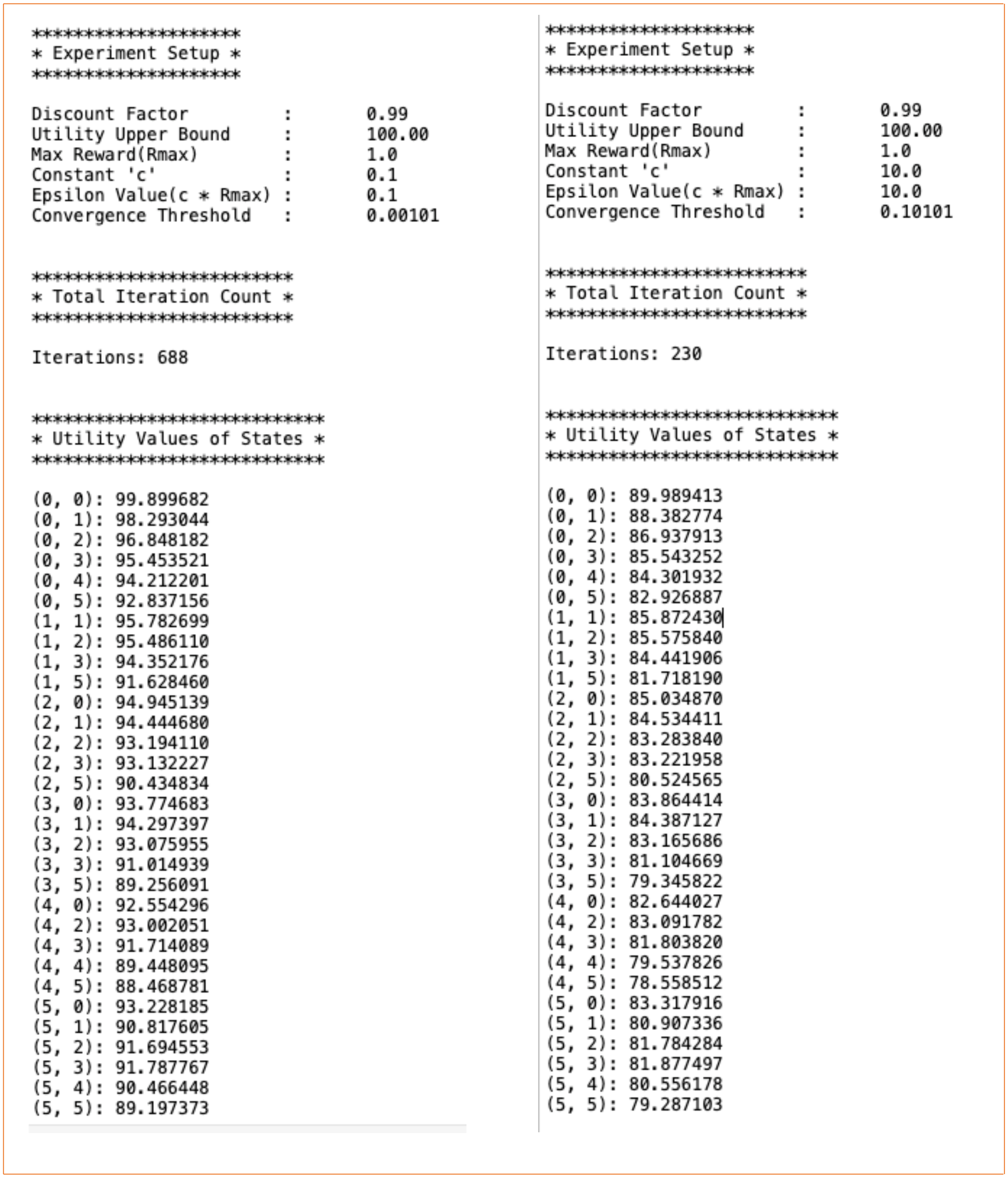
```
***************************
* Plot of Optimal Policy *
***************************

|--------|--------|--------|--------|--------|--------|
|   ^    |  Wall  |   <    |   <    |   >    |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   ^    |   ^    |  Wall  |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   <    |   ^    |   ^    |   <    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   <    |   ^    |   ^    |   >    |
|--------|--------|--------|--------|--------|--------|
|   ^    |  Wall  |  Wall  |  Wall  |   ^    |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   <    |   <    |   >    |   ^    |
|--------|--------|--------|--------|--------|--------|
```

```
**********************************
* Utilities of All States (Map) *
**********************************

|--------|--------|--------|--------|--------|--------|
| 39.499 | 00.000 | 35.256 | 34.104 | 33.909 | 35.063 |
|--------|--------|--------|--------|--------|--------|
| 37.893 | 35.382 | 34.225 | 34.244 | 00.000 | 32.712 |
|--------|--------|--------|--------|--------|--------|
| 36.448 | 35.086 | 32.818 | 33.079 | 33.312 | 32.260 |
|--------|--------|--------|--------|--------|--------|
| 35.053 | 33.952 | 32.735 | 31.064 | 32.095 | 32.476 |
|--------|--------|--------|--------|--------|--------|
| 33.812 | 00.000 | 00.000 | 00.000 | 29.935 | 31.206 |
|--------|--------|--------|--------|--------|--------|
| 32.437 | 31.228 | 30.035 | 28.856 | 29.023 | 29.991 |
|--------|--------|--------|--------|--------|--------|
```

Figure 2.3.3: The plot of optimal policy when the constant "c" is set to 60

To conclude, it is important to note that the constant values and variables used to find the optimal policy such as Discount Factor "gamma", Constant "c" and Epsilon play a crucial role when using the Value Iteration algorithm because they are used to calculate the convergence threshold which is the terminating condition that is used to determine when to stop finding the estimated utilities of all states.

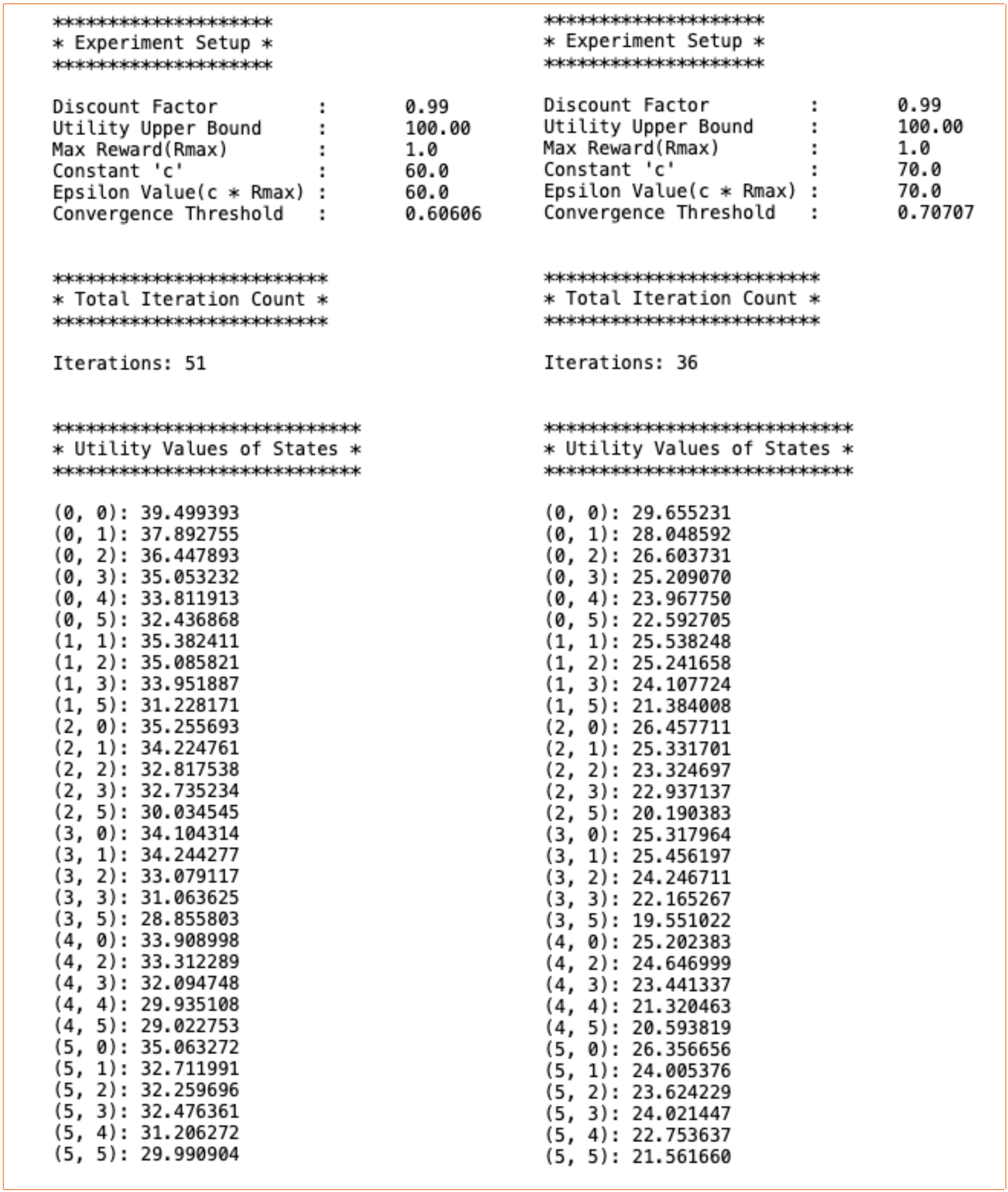
2.4 Utilities of All States



```
********************
* Experiment Setup *
********************

Discount Factor        :       0.99
Utility Upper Bound    :       100.00
Max Reward(Rmax)       :       1.0
Constant 'c'           :       0.1
Epsilon Value(c * Rmax) :      0.1
Convergence Threshold  :       0.00101


*************************
* Total Iteration Count *
*************************

Iterations: 688


****************************
* Utility Values of States *
****************************

(0, 0): 99.899682
(0, 1): 98.293044
(0, 2): 96.848182
(0, 3): 95.453521
(0, 4): 94.212201
(0, 5): 92.837156
(1, 1): 95.782699
(1, 2): 95.486110
(1, 3): 94.352176
(1, 5): 91.628460
(2, 0): 94.945139
(2, 1): 94.444680
(2, 2): 93.194110
(2, 3): 93.132227
(2, 5): 90.434834
(3, 0): 93.774683
(3, 1): 94.297397
(3, 2): 93.075955
(3, 3): 91.014939
(3, 5): 89.256091
(4, 0): 92.554296
(4, 2): 93.002051
(4, 3): 91.714089
(4, 4): 89.448095
(4, 5): 88.468781
(5, 0): 93.228185
(5, 1): 90.817605
(5, 2): 91.694553
(5, 3): 91.787767
(5, 4): 90.466448
(5, 5): 89.197373
```

```
********************
* Experiment Setup *
********************

Discount Factor        :       0.99
Utility Upper Bound    :       100.00
Max Reward(Rmax)       :       1.0
Constant 'c'           :       10.0
Epsilon Value(c * Rmax) :      10.0
Convergence Threshold  :       0.10101


*************************
* Total Iteration Count *
*************************

Iterations: 230


****************************
* Utility Values of States *
****************************

(0, 0): 89.989413
(0, 1): 88.382774
(0, 2): 86.937913
(0, 3): 85.543252
(0, 4): 84.301932
(0, 5): 82.926887
(1, 1): 85.872430
(1, 2): 85.575840
(1, 3): 84.441906
(1, 5): 81.718190
(2, 0): 85.034870
(2, 1): 84.534411
(2, 2): 83.283840
(2, 3): 83.221958
(2, 5): 80.524565
(3, 0): 83.864414
(3, 1): 84.387127
(3, 2): 83.165686
(3, 3): 81.104669
(3, 5): 79.345822
(4, 0): 82.644027
(4, 2): 83.091782
(4, 3): 81.803820
(4, 4): 79.537826
(4, 5): 78.558512
(5, 0): 83.317916
(5, 1): 80.907336
(5, 2): 81.784284
(5, 3): 81.877497
(5, 4): 80.556178
(5, 5): 79.287103
```

Figure 2.4.1: Utility Values when c is set to 0.1 (on the left) and set to 10 (on the right)



```
********************                   ********************
* Experiment Setup *                   * Experiment Setup *
********************                   ********************

Discount Factor          :     0.99      Discount Factor          :     0.99
Utility Upper Bound      :     100.00    Utility Upper Bound      :     100.00
Max Reward(Rmax)         :     1.0       Max Reward(Rmax)         :     1.0
Constant 'c'             :     60.0      Constant 'c'             :     70.0
Epsilon Value(c * Rmax) :     60.0      Epsilon Value(c * Rmax) :     70.0
Convergence Threshold    :     0.60606   Convergence Threshold    :     0.70707


*************************              *************************
* Total Iteration Count *              * Total Iteration Count *
*************************              *************************

Iterations: 51                         Iterations: 36


****************************           ****************************
* Utility Values of States *           * Utility Values of States *
****************************           ****************************

(0, 0): 39.499393                      (0, 0): 29.655231
(0, 1): 37.892755                      (0, 1): 28.048592
(0, 2): 36.447893                      (0, 2): 26.603731
(0, 3): 35.053232                      (0, 3): 25.209070
(0, 4): 33.811913                      (0, 4): 23.967750
(0, 5): 32.436868                      (0, 5): 22.592705
(1, 1): 35.382411                      (1, 1): 25.538248
(1, 2): 35.085821                      (1, 2): 25.241658
(1, 3): 33.951887                      (1, 3): 24.107724
(1, 5): 31.228171                      (1, 5): 21.384008
(2, 0): 35.255693                      (2, 0): 26.457711
(2, 1): 34.224761                      (2, 1): 25.331701
(2, 2): 32.817538                      (2, 2): 23.324697
(2, 3): 32.735234                      (2, 3): 22.937137
(2, 5): 30.034545                      (2, 5): 20.190383
(3, 0): 34.104314                      (3, 0): 25.317964
(3, 1): 34.244277                      (3, 1): 25.456197
(3, 2): 33.079117                      (3, 2): 24.246711
(3, 3): 31.063625                      (3, 3): 22.165267
(3, 5): 28.855803                      (3, 5): 19.551022
(4, 0): 33.908998                      (4, 0): 25.202383
(4, 2): 33.312289                      (4, 2): 24.646999
(4, 3): 32.094748                      (4, 3): 23.441337
(4, 4): 29.935108                      (4, 4): 21.320463
(4, 5): 29.022753                      (4, 5): 20.593819
(5, 0): 35.063272                      (5, 0): 26.356656
(5, 1): 32.711991                      (5, 1): 24.005376
(5, 2): 32.259696                      (5, 2): 23.624229
(5, 3): 32.476361                      (5, 3): 24.021447
(5, 4): 31.206272                      (5, 4): 22.753637
(5, 5): 29.990904                      (5, 5): 21.561660
```

Figure 2.4.2: Utility Values when c is set to 60 (on the left) and set to 70 (on the right)

2.5 Plot of Utility Estimates as a function of the number of iterations

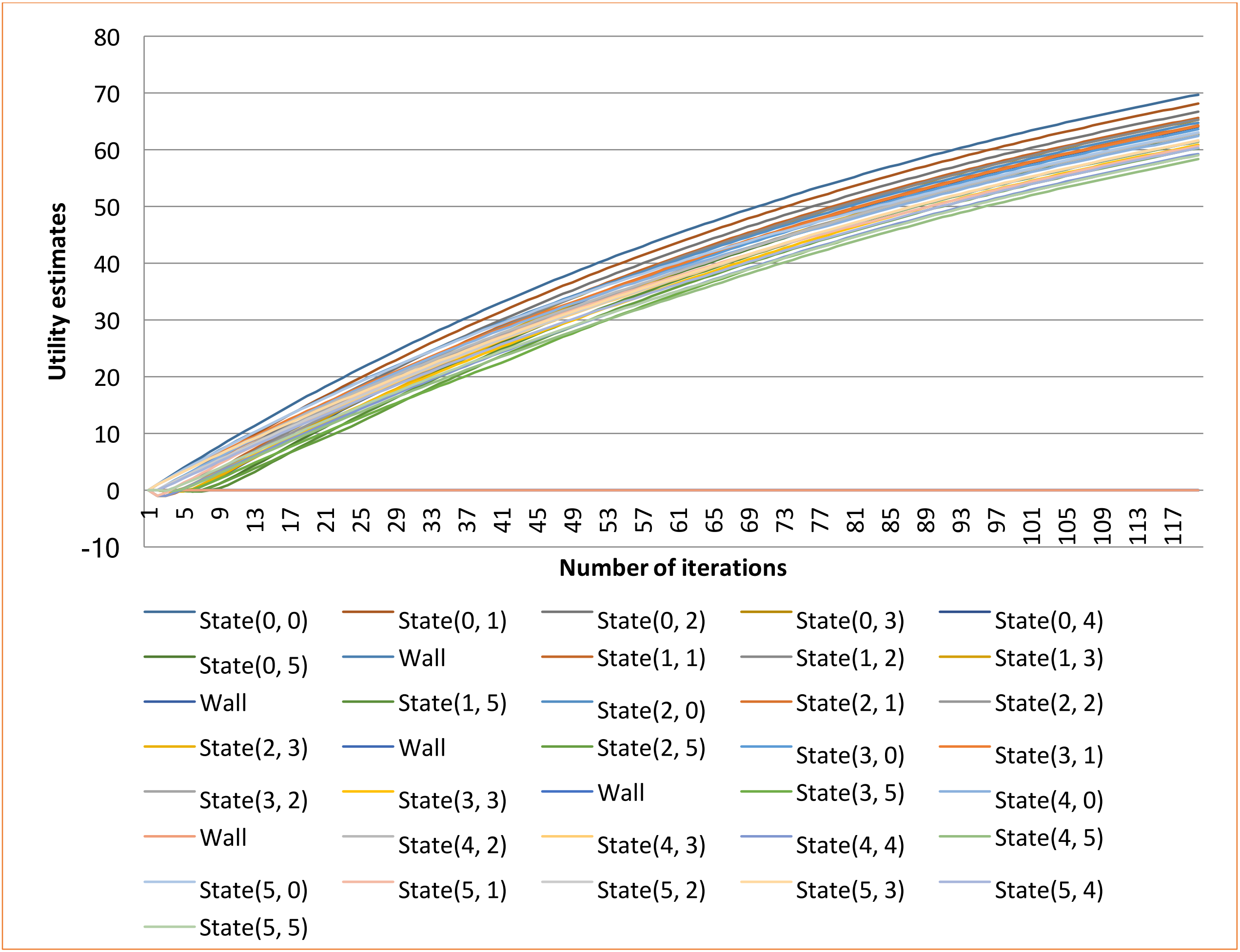


Figure 2.5.1: Plot of Utility Estimates as a function of the number of iterations when constant "c" is set to 30

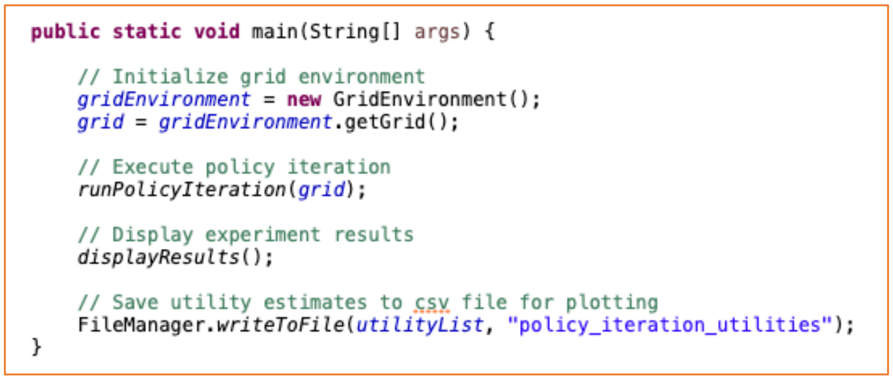
In order to analyze how the utilities estimates has changed as the number of iterations increase, the following variables and constant values are chosen and calculated for the experiment.

Discount Factor	:	0.99
Utility Upper Bound	:	100.00
Max Reward (Rmax)	:	+1.0
Constant 'c'	:	30
Epsilon Value (c*Rmax)	:	30
Convergence Threshold	:	0.30303

At each iteration step, the utility value for each state (except wall) will be updated according to the formula, namely "Bellman update". Theoretically, when the Bellman update is applied infinitely often to update the utility of a state, an equilibrium, will be reached; a condition where the final utility values are the "unique solutions" and the resulting policy is optimal. In practice, since the discount factor is used in "Bellman update", the algorithm is guaranteed to always converge to a unique solution of the Bellman equations whenever the discount factor is less than 1. Therefore, in the above figure, all the estimated utility values are gradually increasing as the number of iterations increase until it reaches the equilibrium where it starts to level off.

3. Implementation of Policy Iteration

When the *PolicyIteration.java* is run, the same program flow as described in the start of Section 2 occurs except that the program is now running the "Policy Iteration" algorithm.



```
public static void main(String[] args) {

    // Initialize grid environment
    gridEnvironment = new GridEnvironment();
    grid = gridEnvironment.getGrid();

    // Execute policy iteration
    runPolicyIteration(grid);

    // Display experiment results
    displayResults();

    // Save utility estimates to csv file for plotting
    FileManager.writeToFile(utilityList, "policy_iteration_utilities");
}
```

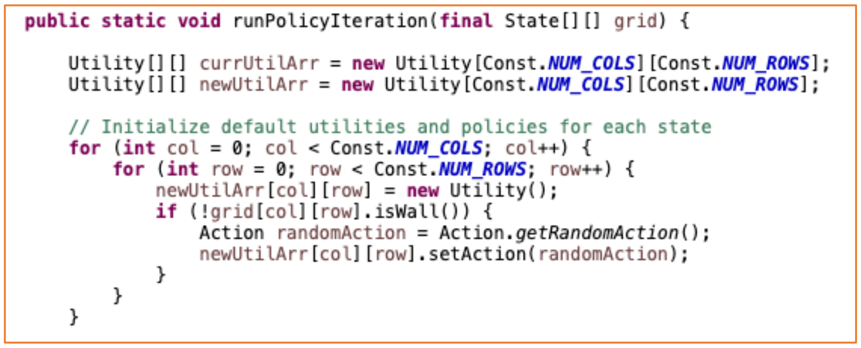
Figure 3.1: The main method for running "Policy Iteration" algorithm

3.1 Initialization of the Maze Environment with Reward Values

The maze environment is implemented in similar manner as described in the Section 2.1.

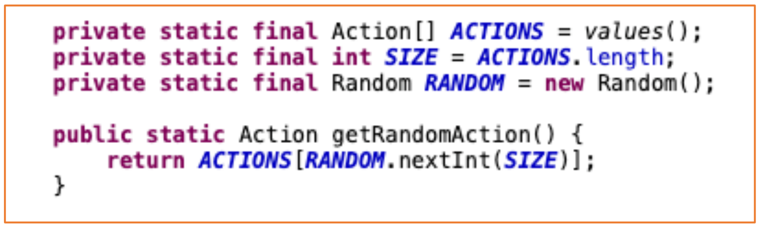
3.2 Policy Iteration for Finding the Optimal Policy

One vector of utilities *currUtilArr* is initialized to store the current utilities of states and one policy vector *newUtilArr* indexed by state is created to store the updated utilities of states after each iteration. All the states in the policy vector will be initialized with utility values of 0.0 and a random action.



```java
public static void runPolicyIteration(final State[][] grid) {

    Utility[][] currUtilArr = new Utility[Const.NUM_COLS][Const.NUM_ROWS];
    Utility[][] newUtilArr = new Utility[Const.NUM_COLS][Const.NUM_ROWS];

    // Initialize default utilities and policies for each state
    for (int col = 0; col < Const.NUM_COLS; col++) {
        for (int row = 0; row < Const.NUM_ROWS; row++) {
            newUtilArr[col][row] = new Utility();
            if (!grid[col][row].isWall()) {
                Action randomAction = Action.getRandomAction();
                newUtilArr[col][row].setAction(randomAction);
            }
        }
    }
```

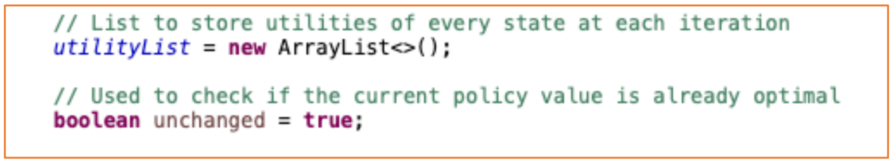
Figure 3.2.1: Random selection of an action that will be assigned to each state



```java
private static final Action[] ACTIONS = values();
private static final int SIZE = ACTIONS.length;
private static final Random RANDOM = new Random();

public static Action getRandomAction() {
    return ACTIONS[RANDOM.nextInt(SIZE)];
}
```

Figure 3.2.2: Random selection of an action that will be assigned to each state (Action.java)

A list of utility is created to help record the estimated utility values for all states after each iteration. The list will be stored in an excel file after running the algorithm so that all these utility estimates can be plotted as a function of the number of iterations. A Boolean flag, *unchanged*, is used to check if the algorithm should terminate when the policy improvement yields no change in the utilities. It is initialized as *true*. During each iteration of the algorithm, it will be set to *false* whenever the utility value of a state derived from the best policy action is larger than the current utility of the state.



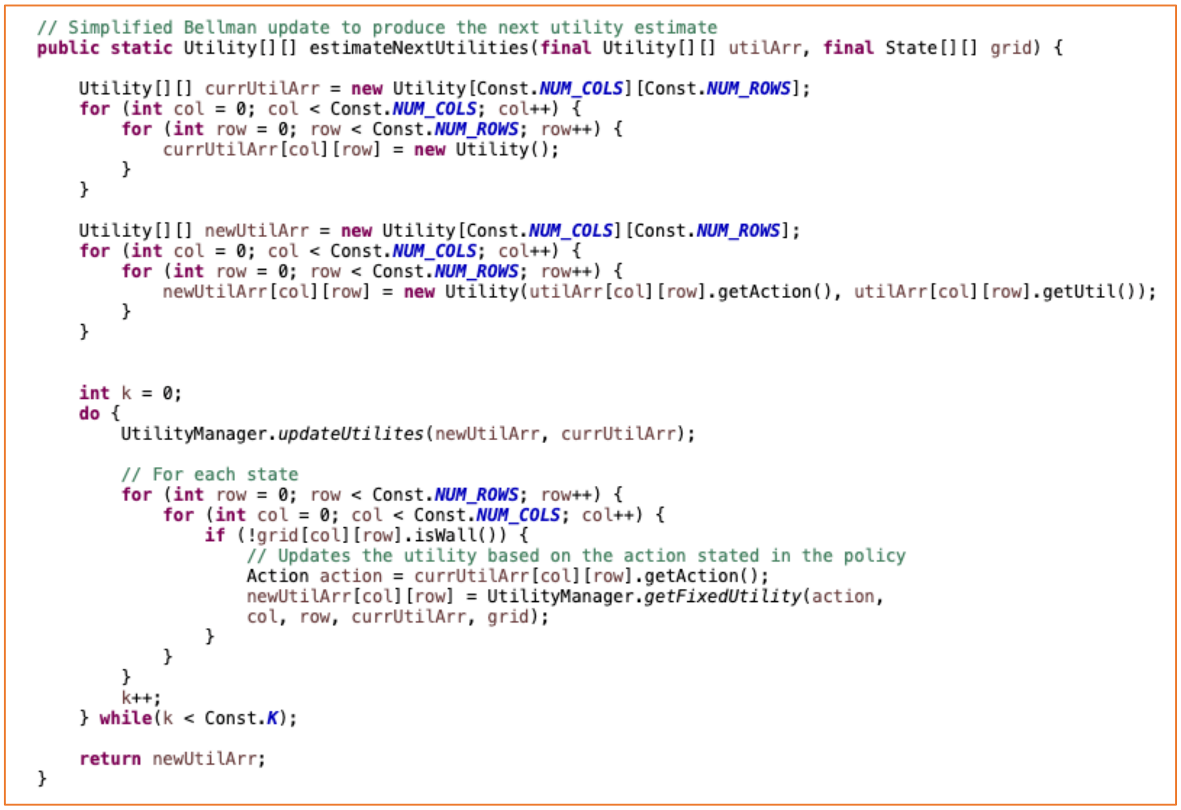
```java
// List to store utilities of every state at each iteration
utilityList = new ArrayList<>();

// Used to check if the current policy value is already optimal
boolean unchanged = true;
```

Figure 3.2.3: The initialization of variables used in executing Policy Iteration algorithm

Based on the current utilities and actions of all states, the new utility values are estimated by using the simplified version of the Bellman equation. "k" number of simplified value iteration steps are performed to calculate a reasonably good approximation of the utilities.

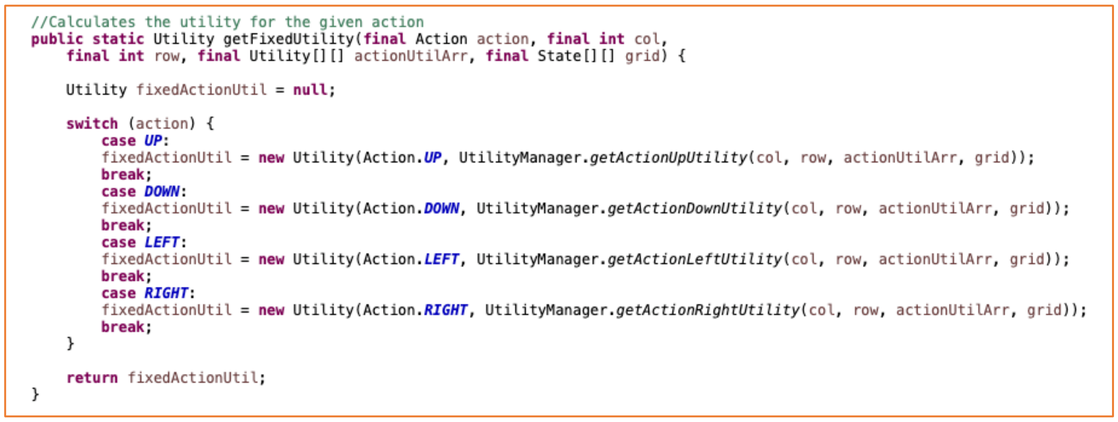
Firstly, *currUtilArr* is initialized with default utility values of 0.0. It will be used to hold the utilities of all states temporarily while performing simplified "Bellman update". A new utility vector is created and its states are initialized with the current policy actions and utility values. At the start of every *k-th* iteration, the current utility array *currUtilArr* will be updated with the new utilities estimated by the simplified version of the Bellman equations. At each *k-th* iteration, the new utility of each state is calculated based on the action defined in the current policy with the help of *getFixedUtility* method. The above procedure will repeat "k" number of times as defined in *Const.java*.

It is important to note that now these equations (to calculate the new best utilities) become linear because unlike in normal Bellman equations, the max operator has been removed, thus reducing the number of iterations required to find the optimal policy by a lot. The code implementation is shown below.



```java
// Simplified Bellman update to produce the next utility estimate
public static Utility[][] estimateNextUtilities(final Utility[][] utilArr, final State[][] grid) {

    Utility[][] currUtilArr = new Utility[Const.NUM_COLS][Const.NUM_ROWS];
    for (int col = 0; col < Const.NUM_COLS; col++) {
        for (int row = 0; row < Const.NUM_ROWS; row++) {
            currUtilArr[col][row] = new Utility();
        }
    }

    Utility[][] newUtilArr = new Utility[Const.NUM_COLS][Const.NUM_ROWS];
    for (int col = 0; col < Const.NUM_COLS; col++) {
        for (int row = 0; row < Const.NUM_ROWS; row++) {
            newUtilArr[col][row] = new Utility(utilArr[col][row].getAction(), utilArr[col][row].getUtil());
        }
    }


    int k = 0;
    do {
        UtilityManager.updateUtilites(newUtilArr, currUtilArr);

        // For each state
        for (int row = 0; row < Const.NUM_ROWS; row++) {
            for (int col = 0; col < Const.NUM_COLS; col++) {
                if (!grid[col][row].isWall()) {
                    // Updates the utility based on the action stated in the policy
                    Action action = currUtilArr[col][row].getAction();
                    newUtilArr[col][row] = UtilityManager.getFixedUtility(action,
                    col, row, currUtilArr, grid);
                }
            }
        }
        k++;
    } while(k < Const.K);

    return newUtilArr;
}
```

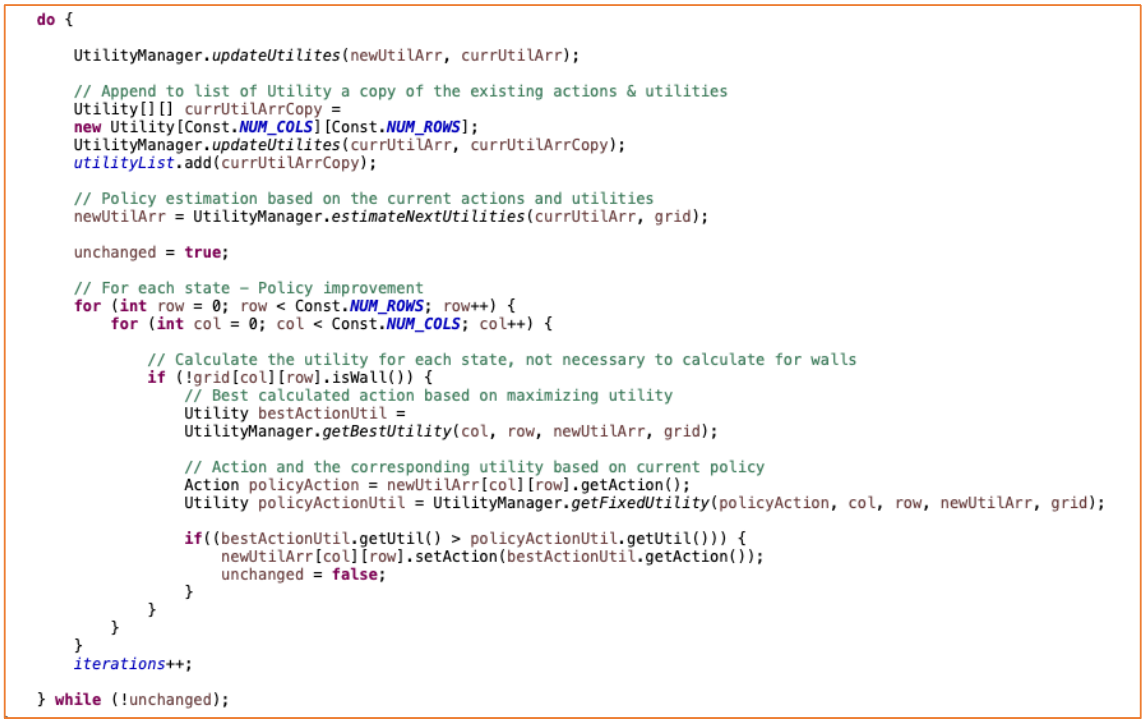
Figure 3.2.4: The implementation code for the simplified version of Bellman update (the iteration will continue k number of times.



```
//Calculates the utility for the given action
public static Utility getFixedUtility(final Action action, final int col,
    final int row, final Utility[][] actionUtilArr, final State[][] grid) {

    Utility fixedActionUtil = null;

    switch (action) {
        case UP:
        fixedActionUtil = new Utility(Action.UP, UtilityManager.getActionUpUtility(col, row, actionUtilArr, grid));
        break;
        case DOWN:
        fixedActionUtil = new Utility(Action.DOWN, UtilityManager.getActionDownUtility(col, row, actionUtilArr, grid));
        break;
        case LEFT:
        fixedActionUtil = new Utility(Action.LEFT, UtilityManager.getActionLeftUtility(col, row, actionUtilArr, grid));
        break;
        case RIGHT:
        fixedActionUtil = new Utility(Action.RIGHT, UtilityManager.getActionRightUtility(col, row, actionUtilArr, grid));
        break;
    }

    return fixedActionUtil;
}
```

Figure 3.2.5: Finding the fixed utility according to an action as defined in a policy

After finding the utilities of all states estimated by the simplified Bellman update, For each state, the utility value for choosing the action that maximizes the subsequent state is calculated. Then it is compared against the estimated utility to determine if the current policy is sufficient. If the utility achieved by taking the best action is larger than the current estimated utility value, then the policy of that particular state will be updated with new policy that will potentially maximize its subsequent states. The above steps will be repeated until the algorithm has found the optimal policy (i.e. the current estimated utility is no longer different from the utility achieved by taking the best action).



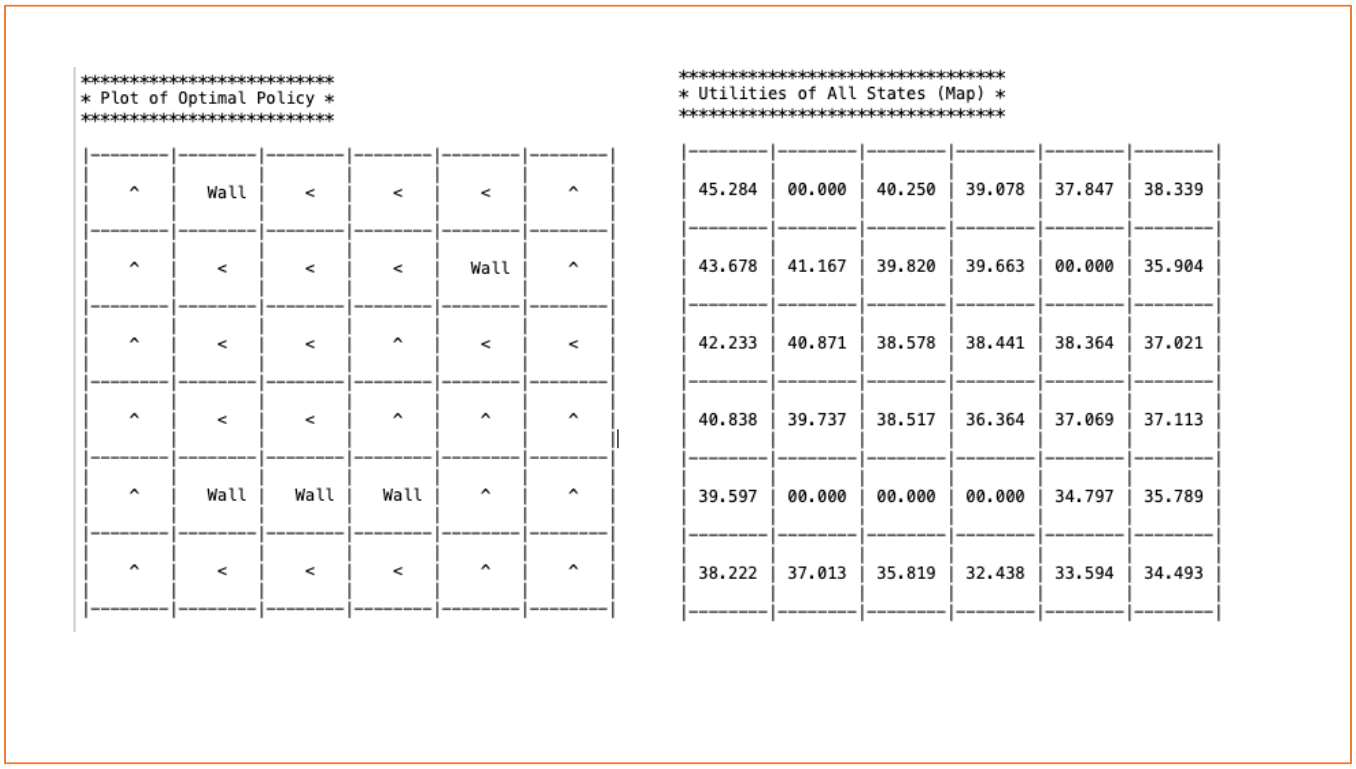
```
do {

    UtilityManager.updateUtilites(newUtilArr, currUtilArr);

    // Append to list of Utility a copy of the existing actions & utilities
    Utility[][] currUtilArrCopy =
    new Utility[Const.NUM_COLS][Const.NUM_ROWS];
    UtilityManager.updateUtilites(currUtilArr, currUtilArrCopy);
    utilityList.add(currUtilArrCopy);

    // Policy estimation based on the current actions and utilities
    newUtilArr = UtilityManager.estimateNextUtilities(currUtilArr, grid);

    unchanged = true;

    // For each state - Policy improvement
    for (int row = 0; row < Const.NUM_ROWS; row++) {
        for (int col = 0; col < Const.NUM_COLS; col++) {

            // Calculate the utility for each state, not necessary to calculate for walls
            if (!grid[col][row].isWall()) {
                // Best calculated action based on maximizing utility
                Utility bestActionUtil =
                UtilityManager.getBestUtility(col, row, newUtilArr, grid);

                // Action and the corresponding utility based on current policy
                Action policyAction = newUtilArr[col][row].getAction();
                Utility policyActionUtil = UtilityManager.getFixedUtility(policyAction, col, row, newUtilArr, grid);

                if((bestActionUtil.getUtil() > policyActionUtil.getUtil())) {
                    newUtilArr[col][row].setAction(bestActionUtil.getAction());
                    unchanged = false;
                }
            }
        }
    }
    iterations++;

} while (!unchanged);
```

Figure 3.2.6: Implementation of the modified "Policy Iteration" algorithm

The resulting algorithm is called "modified policy iteration". In this assignment, "Policy Iteration" algorithm is modified version and it is implemented as shown in the above figure.

3.3 Plot of Optimal Policy and its Analysis with different constant "k" values

When the number of simplified value iteration steps defined as "k" is set to 10, the total number of iterations required to find the optimal policy vary from 7 to 15 depending on the starting policy of each state since all the states are initialized with a randomly chosen policy.

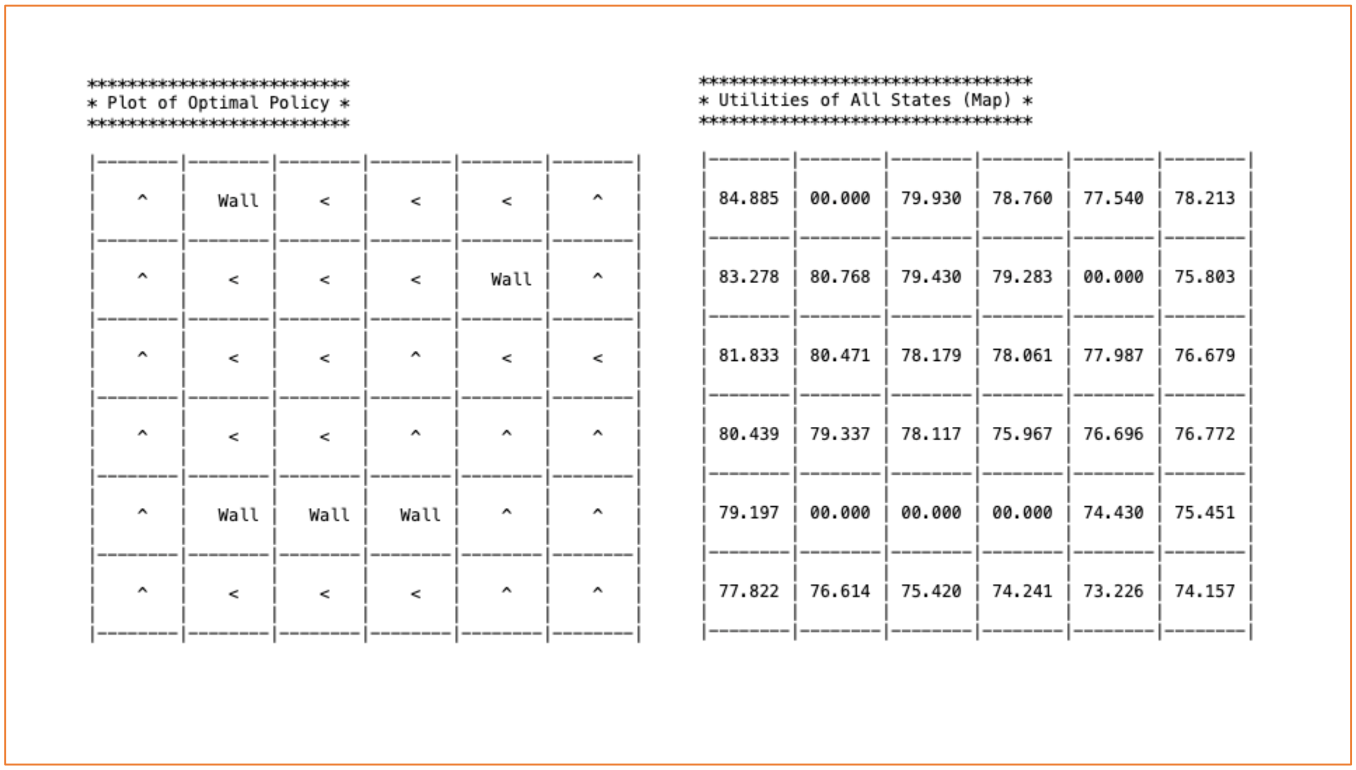


```
***************************
* Plot of Optimal Policy *
***************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |   <    |   <    |   <    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |  Wall  |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   <    |   <    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |  Wall  |  Wall  |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

```
**********************************
* Utilities of All States (Map) *
**********************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 45.284 | 00.000 | 40.250 | 39.078 | 37.847 | 38.339 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 43.678 | 41.167 | 39.820 | 39.663 | 00.000 | 35.904 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 42.233 | 40.871 | 38.578 | 38.441 | 38.364 | 37.021 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 40.838 | 39.737 | 38.517 | 36.364 | 37.069 | 37.113 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 39.597 | 00.000 | 00.000 | 00.000 | 34.797 | 35.789 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 38.222 | 37.013 | 35.819 | 32.438 | 33.594 | 34.493 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

Figure 3.3.1: The plot of optimal policy when "k" value is set to 10; total iterations required: 7-15

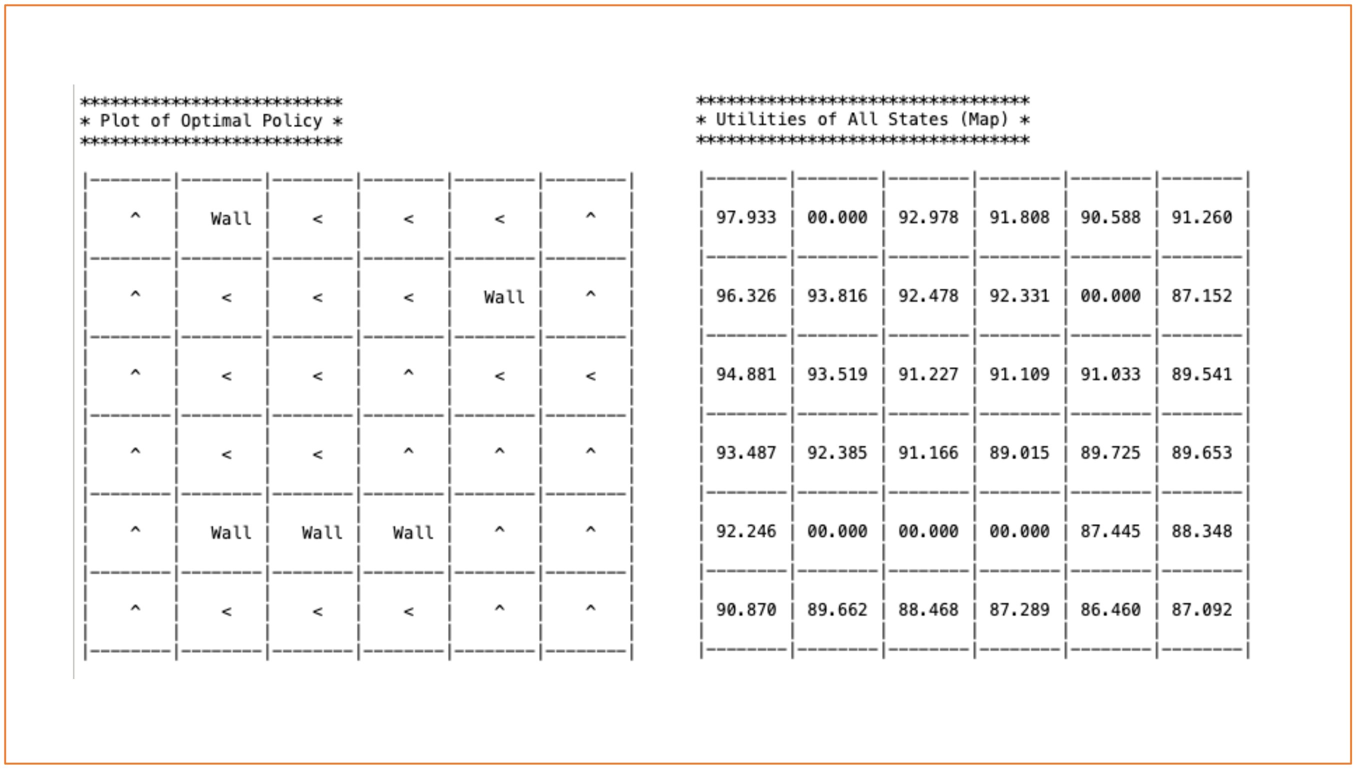
When "k" is set to 50, the total number of iterations required vary from 5 to 8 depending on the starting policy of each state since all the states are initialized with a randomly chosen policy. The number of iterations required become halved and utilities of all states become doubled as compared to the above experiment when k is set to 10.



```
***************************
* Plot of Optimal Policy *
***************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |   <    |   <    |   <    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |  Wall  |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   <    |   <    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |  Wall  |  Wall  |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

```
**********************************
* Utilities of All States (Map) *
**********************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 84.885 | 00.000 | 79.930 | 78.760 | 77.540 | 78.213 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 83.278 | 80.768 | 79.430 | 79.283 | 00.000 | 75.803 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 81.833 | 80.471 | 78.179 | 78.061 | 77.987 | 76.679 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 80.439 | 79.337 | 78.117 | 75.967 | 76.696 | 76.772 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 79.197 | 00.000 | 00.000 | 00.000 | 74.430 | 75.451 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 77.822 | 76.614 | 75.420 | 74.241 | 73.226 | 74.157 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

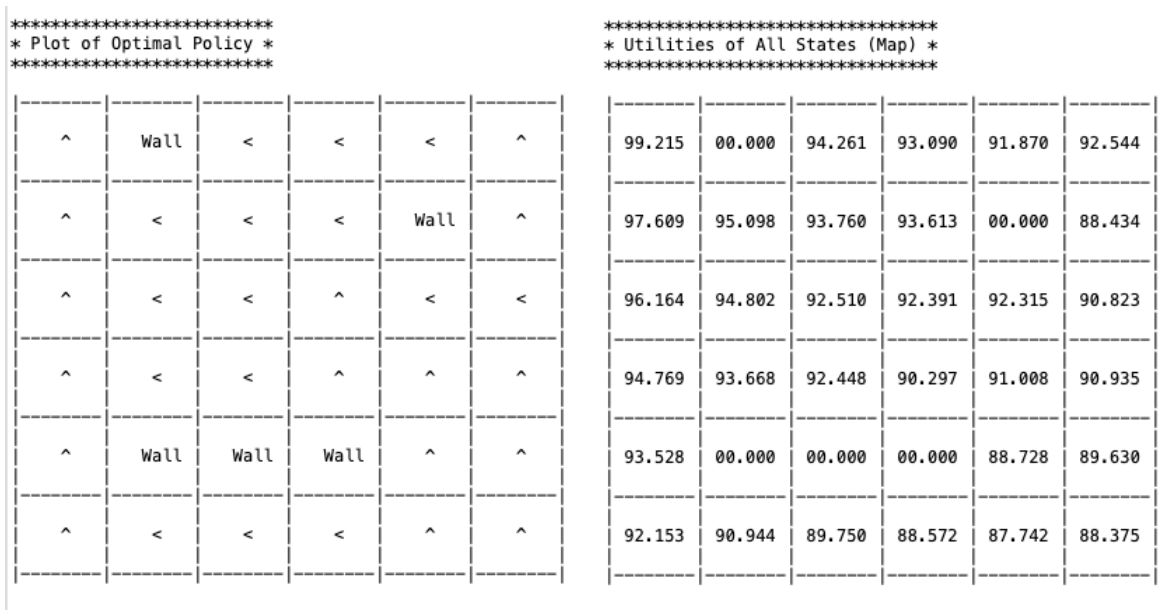
Figure 3.3.2: The plot of optimal policy when "k" value is set to 50; total iterations required: 5-8



```
***************************
* Plot of Optimal Policy *
***************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |   <    |   <    |   <    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |  Wall  |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   <    |   <    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |  Wall  |  Wall  |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

```
**********************************
* Utilities of All States (Map) *
**********************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 97.933 | 00.000 | 92.978 | 91.808 | 90.588 | 91.260 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 96.326 | 93.816 | 92.478 | 92.331 | 00.000 | 87.152 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 94.881 | 93.519 | 91.227 | 91.109 | 91.033 | 89.541 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 93.487 | 92.385 | 91.166 | 89.015 | 89.725 | 89.653 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 92.246 | 00.000 | 00.000 | 00.000 | 87.445 | 88.348 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 90.870 | 89.662 | 88.468 | 87.289 | 86.460 | 87.092 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

Figure 3.3.3: The plot of optimal policy when "k" value is set to 75; total iterations required: 5-7



```
**************************
* Plot of Optimal Policy *
**************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |   <    |   <    |   <    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |  Wall  |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   <    |   <    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   ^    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |  Wall  |  Wall  |  Wall  |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|   ^    |   <    |   <    |   <    |   ^    |   ^    |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

```
*********************************
* Utilities of All States (Map) *
*********************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 99.215 | 00.000 | 94.261 | 93.090 | 91.870 | 92.544 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 97.609 | 95.098 | 93.760 | 93.613 | 00.000 | 88.434 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 96.164 | 94.802 | 92.510 | 92.391 | 92.315 | 90.823 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 94.769 | 93.668 | 92.448 | 90.297 | 91.008 | 90.935 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 93.528 | 00.000 | 00.000 | 00.000 | 88.728 | 89.630 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 92.153 | 90.944 | 89.750 | 88.572 | 87.742 | 88.375 |
|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
```

Figure 3.3.4: The plot of optimal policy when "k" value is set to100; total iterations required: 5-7

As it can be observed in *Figure 3.3.1*, *Figure 3.3.2*, *Figure 3.3.3* and *Figure 3.3.4*, by increasing the number of "k", the final estimated utilities increase. The increase in utility values is notable when k is set to 10 and when k is set to 50. However, as "k" is increased further to 75 and 100, the changes in final utilities become subtle. Since the algorithm uses discount factor, the utility upper bound is always limited to 100 as calculated by the formula below:

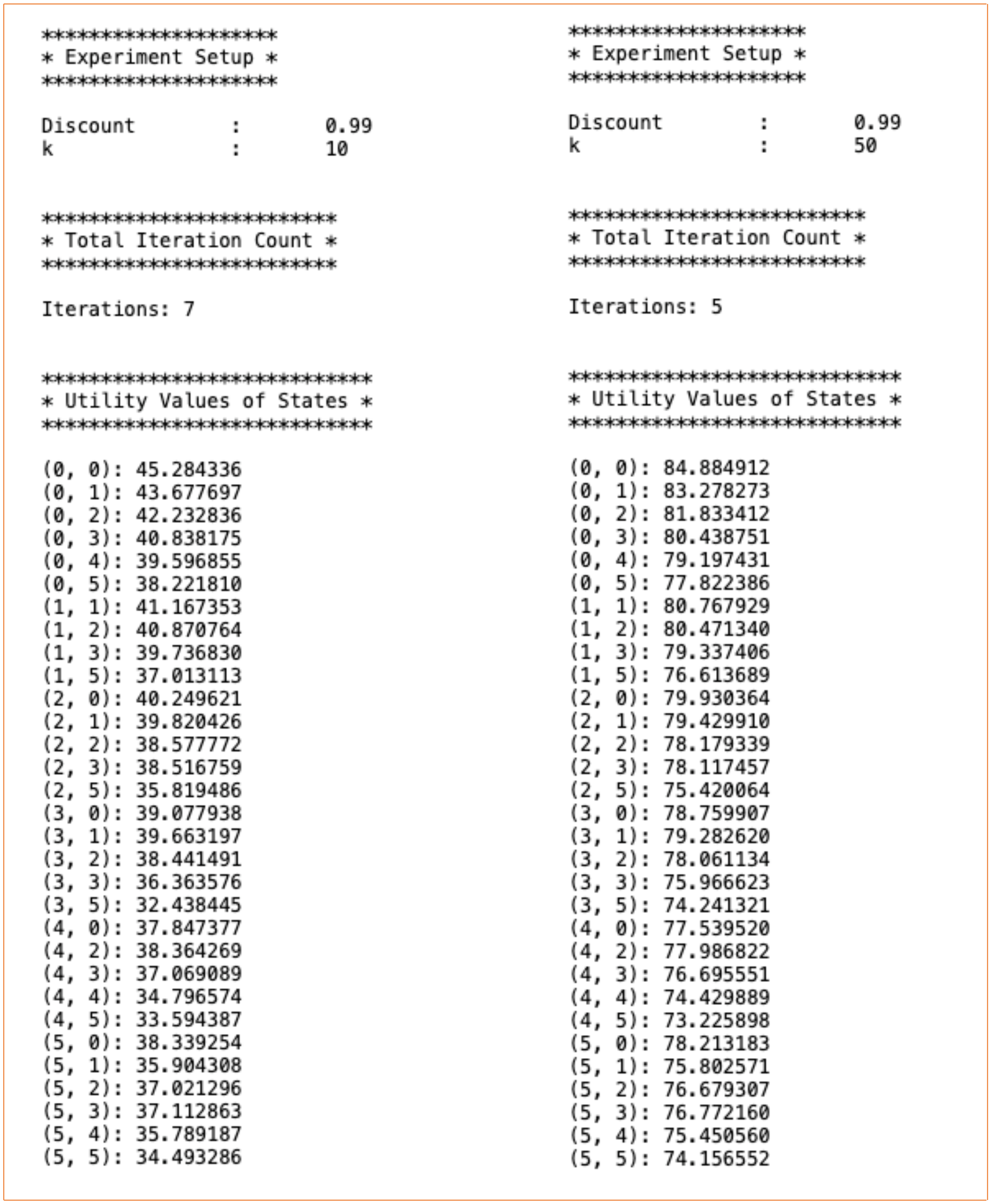
Utility Upper Bound = Rmax / (1 – gamma)

where: *gamma* is discount factor

Rmax is maximum reward

Thus, the utility value will never increase to more than 100. Moreover, the number of iterations required to find the optimal policy are also not much different when k is set to 50, 75 and 100 while still achieving the most optimal policy. It could be verified by comparing with the optimal policy plot in *Figure 2.3.1* and *Figure 2.3.2*.

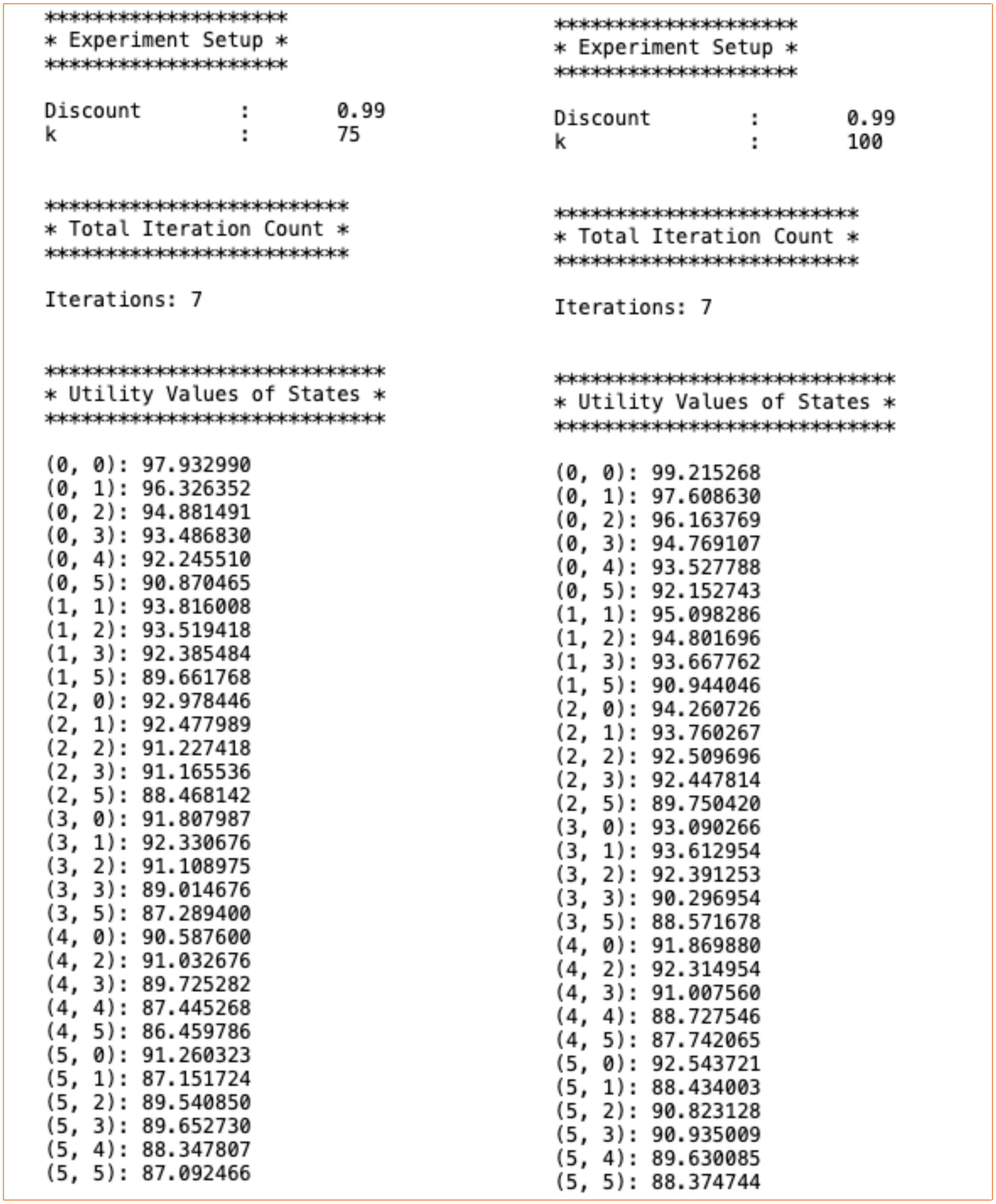
3.4 Utilities of All States



```
********************
* Experiment Setup *
********************

Discount        :       0.99
k               :       10


*************************
* Total Iteration Count *
*************************

Iterations: 7


****************************
* Utility Values of States *
****************************

(0, 0): 45.284336
(0, 1): 43.677697
(0, 2): 42.232836
(0, 3): 40.838175
(0, 4): 39.596855
(0, 5): 38.221810
(1, 1): 41.167353
(1, 2): 40.870764
(1, 3): 39.736830
(1, 5): 37.013113
(2, 0): 40.249621
(2, 1): 39.820426
(2, 2): 38.577772
(2, 3): 38.516759
(2, 5): 35.819486
(3, 0): 39.077938
(3, 1): 39.663197
(3, 2): 38.441491
(3, 3): 36.363576
(3, 5): 32.438445
(4, 0): 37.847377
(4, 2): 38.364269
(4, 3): 37.069089
(4, 4): 34.796574
(4, 5): 33.594387
(5, 0): 38.339254
(5, 1): 35.904308
(5, 2): 37.021296
(5, 3): 37.112863
(5, 4): 35.789187
(5, 5): 34.493286
```

```
********************
* Experiment Setup *
********************

Discount        :       0.99
k               :       50


*************************
* Total Iteration Count *
*************************

Iterations: 5


****************************
* Utility Values of States *
****************************

(0, 0): 84.884912
(0, 1): 83.278273
(0, 2): 81.833412
(0, 3): 80.438751
(0, 4): 79.197431
(0, 5): 77.822386
(1, 1): 80.767929
(1, 2): 80.471340
(1, 3): 79.337406
(1, 5): 76.613689
(2, 0): 79.930364
(2, 1): 79.429910
(2, 2): 78.179339
(2, 3): 78.117457
(2, 5): 75.420064
(3, 0): 78.759907
(3, 1): 79.282620
(3, 2): 78.061134
(3, 3): 75.966623
(3, 5): 74.241321
(4, 0): 77.539520
(4, 2): 77.986822
(4, 3): 76.695551
(4, 4): 74.429889
(4, 5): 73.225898
(5, 0): 78.213183
(5, 1): 75.802571
(5, 2): 76.679307
(5, 3): 76.772160
(5, 4): 75.450560
(5, 5): 74.156552
```

Figure 3.4.1: Utility Values when k is set to 10 (on the left) and set to 50 (on the right)



```
********************
* Experiment Setup *
********************

Discount        :       0.99
k               :       75


*************************
* Total Iteration Count *
*************************

Iterations: 7


****************************
* Utility Values of States *
****************************

(0, 0): 97.932990
(0, 1): 96.326352
(0, 2): 94.881491
(0, 3): 93.486830
(0, 4): 92.245510
(0, 5): 90.870465
(1, 1): 93.816008
(1, 2): 93.519418
(1, 3): 92.385484
(1, 5): 89.661768
(2, 0): 92.978446
(2, 1): 92.477989
(2, 2): 91.227418
(2, 3): 91.165536
(2, 5): 88.468142
(3, 0): 91.807987
(3, 1): 92.330676
(3, 2): 91.108975
(3, 3): 89.014676
(3, 5): 87.289400
(4, 0): 90.587600
(4, 2): 91.032676
(4, 3): 89.725282
(4, 4): 87.445268
(4, 5): 86.459786
(5, 0): 91.260323
(5, 1): 87.151724
(5, 2): 89.540850
(5, 3): 89.652730
(5, 4): 88.347807
(5, 5): 87.092466
```

```
********************
* Experiment Setup *
********************

Discount        :       0.99
k               :       100


*************************
* Total Iteration Count *
*************************

Iterations: 7


****************************
* Utility Values of States *
****************************

(0, 0): 99.215268
(0, 1): 97.608630
(0, 2): 96.163769
(0, 3): 94.769107
(0, 4): 93.527788
(0, 5): 92.152743
(1, 1): 95.098286
(1, 2): 94.801696
(1, 3): 93.667762
(1, 5): 90.944046
(2, 0): 94.260726
(2, 1): 93.760267
(2, 2): 92.509696
(2, 3): 92.447814
(2, 5): 89.750420
(3, 0): 93.090266
(3, 1): 93.612954
(3, 2): 92.391253
(3, 3): 90.296954
(3, 5): 88.571678
(4, 0): 91.869880
(4, 2): 92.314954
(4, 3): 91.007560
(4, 4): 88.727546
(4, 5): 87.742065
(5, 0): 92.543721
(5, 1): 88.434003
(5, 2): 90.823128
(5, 3): 90.935009
(5, 4): 89.630085
(5, 5): 88.374744
```

Figure 3.4.2: Utility Values when k is set to 75 (on the left) and set to 100 (on the right)

3.5 Plot of Utility Estimates as a function of the number of iterations

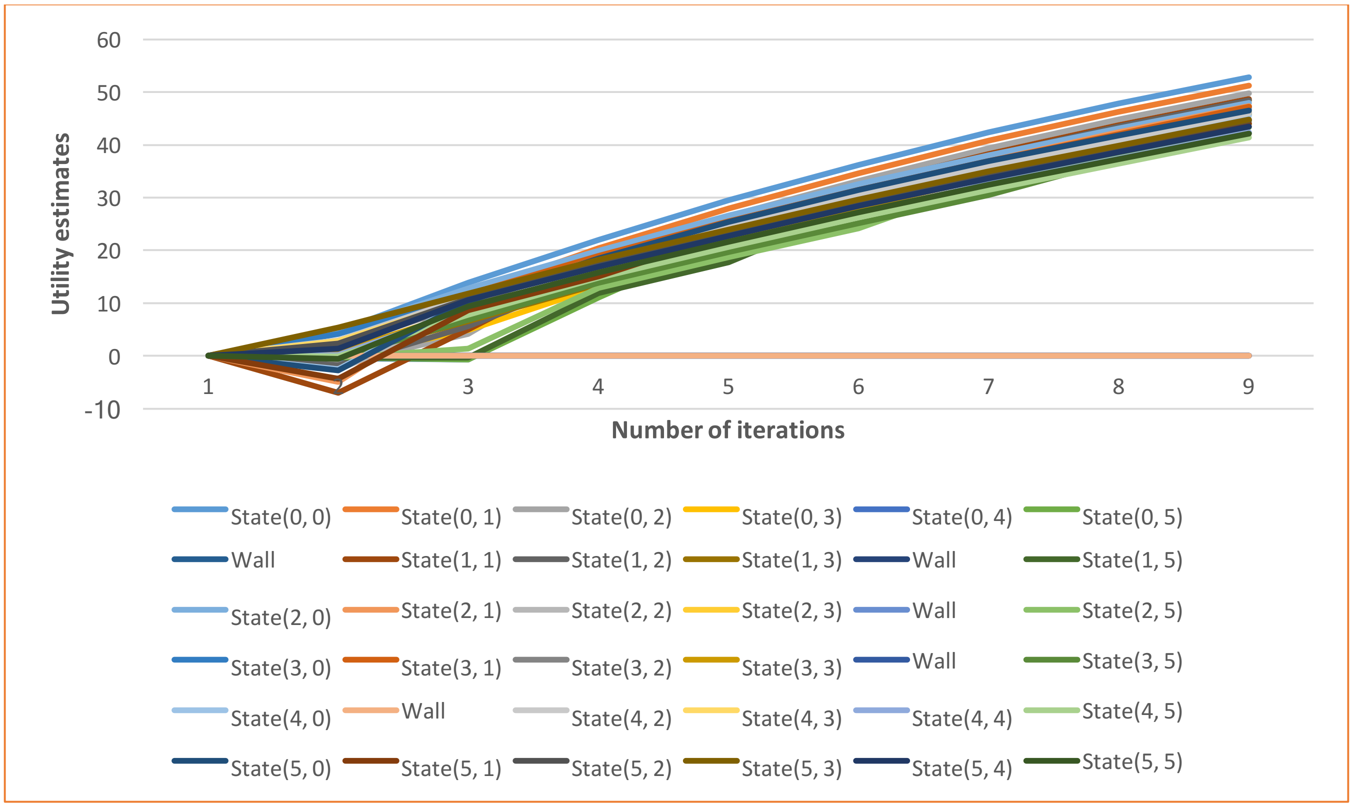


Figure 3.5.1: Plot of Utility Estimates as a function of the number of iterations when "k" is set to repeat for 10 times

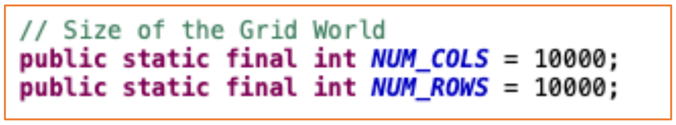
The following variables and constant values are chosen and calculated for the experiment in order to analyze how the utilities estimates has changed as the number of iterations increase

Discount Factor	:	0.99
Utility Upper Bound	:	100.00
Max Reward (Rmax)	:	+1.0
'k' repetitions	:	10

At each iteration step, the utility value for each state (except wall) will be updated according to the simplified version of the "Bellman update". It is important to note that the equations that are now used to find the new utilities are linear because the "max" operator has been removed. "k" number of simplified value iteration steps are performed to find a reasonably good approximation of the utilities. The resulting algorithm, "modified policy iteration", is much more efficient than standard policy iteration or value iteration. Consequently, only 5 to 8 iterations (as described in *Figure 3.3.2*, *Figure 3.3.3*, *Figure 3.3.4*) are required to find the optimal policy, much less than the number of iterations required in "Value Iteration" algorithm.

4. Design of More Complicated Maze Environment

In this section, the performance of the two algorithms, "Value Iteration" and modified "Policy Iteration", will be tested by running them in a more complicated maze environment. A more complicated maze can be implemented merely by increasing the size of the maze environment. It can be done by modifying the number of columns and rows defined in *Const.java*. By doing so, the number of states that exist in the environment will also increase. It implicitly implies longer calculation time and larger memory required to run the two algorithms. It will be analyzed more in details later.

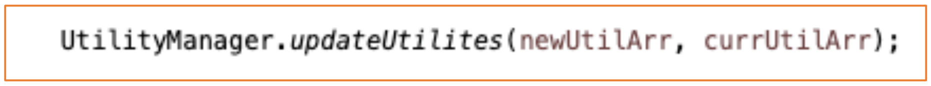


```
// Size of the Grid World
public static final int NUM_COLS = 10000;
public static final int NUM_ROWS = 10000;
```

Figure 4.1: Implementation of a more complicated maze

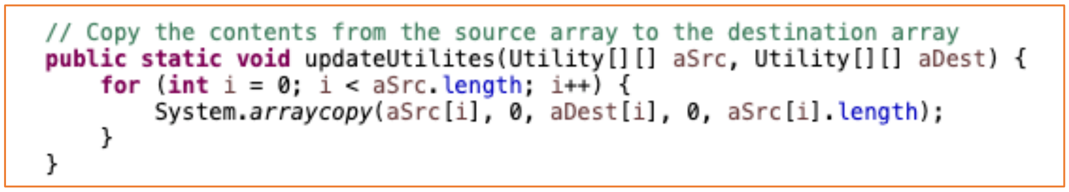
4.1 Impact of the Space Complexity of the Maze Environment on Convergence Rate

Since both algorithms require updating the utility or policy for all states at once, by increasing the size of the maze to "10, 000 x 10, 000", it will severely affect the performance of the two algorithms in terms of both space complexity and time complexity especially when it performs the "Bellman update" at the start of each iteration.



```
UtilityManager.updateUtilites(newUtilArr, currUtilArr);
```

Figure 4.1.1: Bellman update method that is used in both "Value Iteration" and "Policy Iteration"



```java
// Copy the contents from the source array to the destination array
public static void updateUtilites(Utility[][] aSrc, Utility[][] aDest) {
    for (int i = 0; i < aSrc.length; i++) {
        System.arraycopy(aSrc[i], 0, aDest[i], 0, aSrc[i].length);
    }
}
```

Figure 4.1.2: Bellman update method that is used in both "Value Iteration" and "Policy Iteration"

After upsizing the maze environment, it is found that the execution of the two algorithms takes excruciatingly long for both algorithms to find the optimal policy. Therefore, it is important to note that given the current way of implementation, the higher the space complexity of the environment, the higher the time complexity of the convergence in finding the optimal policy.

However, it turns out that the above issue in terms of space and time complexity could be alleviated by selectively picking any subset of states and apply either kind of updating policy improvement or simplified value iteration to that subset. Such algorithm is known as "asynchronous policy iteration". The "asynchronous policy iteration" is guaranteed to converge to an optimal policy given certain conditions on the initial policy and initial utility function. Based on such approach, much more heuristic algorithms can be designed so that it can concentrate on updating the values of states that are likely to be reached by a good policy.

4.2 Impact of Fixed Time Limit N on Learning the Right Policy

The maze can also be made more complicated by introduction a new concept of having fixed time limit or in other words, "finite horizon". In the above sections, the agent does not have to deal with time limit since it is making decision in "infinite horizon". There is no fixed time limit in infinite horizon and hence, there is no reason to behave differently in the same state and the optimal policy is stationary. However, this is not the case when the agent has to make decision in a finite horizon where the optimal action in a given state could change over time. Hence, in such scenario, the final optimal policy will no longer be the same as the one that has been achieved in the above sections.

4.3 Impact of Discount Factor on Learning the Right Policy

Despite all of the limitations of the two algorithms that are described in Section 4.1 and Section 4.2, the agent will still be able to learn the right policy as long as the discount factor is less than 1. In this assignment, it is stated that there are no terminal states and the agent's state sequence is infinite. Consider a case where the discount factor value is set to 1 in infinite horizon where the environment does not contain a terminal state (just as the one in this assignment). Then all environment histories will infinitely long and utilities with additive, undiscounted rewards will generally be infinite. It can be bottleneck for both the space complexity and time complexity of the two algorithms since they will become unpredictably high. Thus, it is important to always set the discount factor (*gamma*) to be less than 1 and with discounted rewards, the utility of an infinite sequence is always finite.

Since the two algorithms are implemented with Bellman update that is essentially based on the contraction by a factor of *gamma* on the space of utility vectors, it is guaranteed to always converges to a unique solution of the Bellman equations whenever *gamma* is less than 1. In other words, the discount factor has to be less than 1 in order to learn the right policy in infinite horizons.