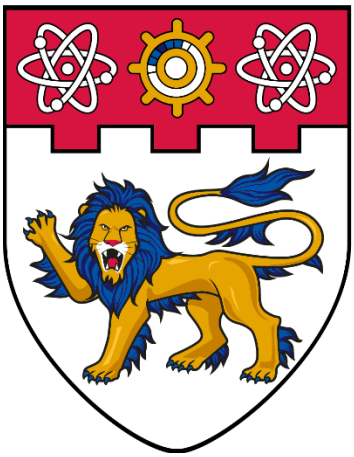


NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

# CE/CZ4046: INTELLIGENT AGENTS

## ASSIGNMENT 1: AGENT DECISION MAKING


**SINGH AISHWARYA (U1923952C)**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

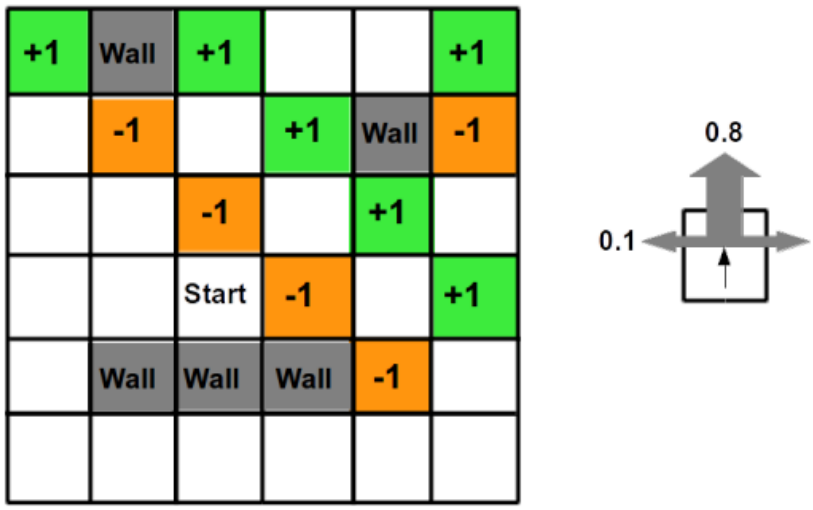
**NANYANG TECHNOLOGICAL UNIVERSITY**

# TABLE OF CONTENTS

# 1. INTRODUCTION

The notion of agent decision-making entails creating agents that can make decisions based on available information to attain the intended goals. We have been tasked with addressing a maze environment challenge in which an agent's behavior/decisions must be studied in the given maze (*Figure 1-1*), and our own complex maze –

*Figure 1-1: Given Maze Environment, Agent, Rewards, and Transition Probabilities*




The design of the grid environment results in an 80% likelihood that the preferred result will occur, whereas there is a 10% chance that the agent will move at right angles to their intended course. If proceeding with the movement would induce collision against any wall, then said action shall not be taken and the agent shall instead remain stationary. The white cell receives -0.04 points, the green obtains +1 point, and the brown cell receives -1 point. Because there are no terminal states in the task, the agent's state sequence is limitless.

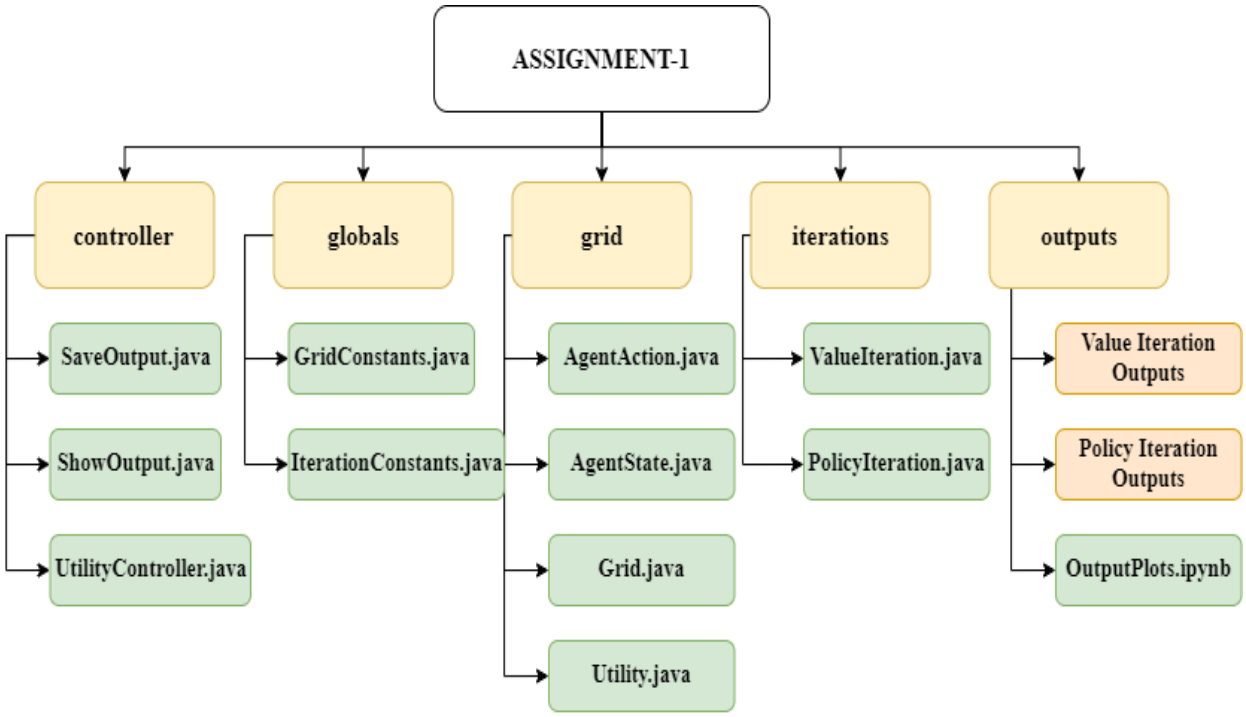
To solve this challenge, we must create algorithms that can navigate the maze environment while maximizing the cumulative reward. The program must assess the likelihood of traveling in various directions as well as the rewards associated with each cell. By constructing such an algorithm, we can learn a lot about how intelligent entities make decisions in complicated environments.

# 2. PROPOSED SOLUTION

## 2.1. Solution Architecture

*Figure 2-1* depicts the folder structure of the system's source code –

*Figure 2-1: Directory of source code*




To address the challenge posed in this assignment, two algorithms have been implemented to obtain the optimal policy for both the maze environments – Value Iteration and Policy Iteration. The details of the implemented packages and Java files can be found below –

- Package: controller –
    - *SaveOutput.java*: This class provides methods for saving experiment results in a CSV file.
    - *ShowOutput.java*: This class defines four methods to display the Grid, the policy, the utilities of all the (non-wall) states, and the utilities of all the states, in a grid format. Each method uses StringBuilder to format the output and prints it to the console.
    - *UtilityController.java*: This is a class that provides static methods for calculating maximum utilities and estimating the next utilities based on the Bellman equation.

- Package: globals –
    - *GridConstants.java*: The GridConstants class contains constants defining the dimensions, separators, and special cell placement of a grid, marked as final to prevent subclassing.
    - *IterationConstants.java*: This class holds constants related to the value and policy iteration algorithms, such as rewards for different cells, transition probabilities, discount factor, maximum reward, maximum error, number of times the Bellman algorithm is executed, and the upper bound on utility values.
- Package: grid –
    - *AgentAction.java*: This is an enum class that defines all possible actions an agent can take and includes a method to generate a random action from this set.
    - *AgentState.java*: This class represents a state in the grid environment with a reward and a Boolean flag to indicate if it's a wall or not.
    - *Grid.java*: This is a class that represents a grid environment consisting of a 2D array of states. It has a constructor to initialize the grid environment by building the grid and methods to get and set the grid, and to initialize the grid by setting rewards for each state based on their color, and marking walls as unreachable.
    - *Utility.java*: This is a utility class representing the utility value and action associated with a state, with methods to get and set the utility value and action, as well as compare utility values with another object in descending order.
- Package: iterations –
    - *ValueIteration.java*: This Java class implements the value iteration algorithm for the grid environment, calculates the utility values for each state in the grid, and displays the experiment results including the optimal policy and final utilities.
    - PolicyIteration.java: This is a Java implementation of the policy iteration algorithm, which iteratively computes the optimal policy for an agent in a grid environment.
- Folder: Outputs –
    - Value Iteration Outputs: This folder consists of the outputs of the different runs of the value iteration algorithm. The console output and the utilities of each run (with different ‘c’ values) are saved in respective subfolders.

- Policy Iteration Outputs: This folder consists of the outputs of the different runs of the policy iteration algorithm. The console output and the utilities of each run (with different ‘k’ values) are saved in respective subfolders.
- OutputPlots.ipynb: Python code to visualize the outputs from the different experiments (which are saved as CSV files) in different sub-directories.

### 2.2. Implementation Tools

The system has been implemented using Java version 14.0.2. The files were tested and run using Visual Studio Code, and the code snippets have been taken from the same. To visualize the CSV outputs generated by the Java code, Python libraries such as pandas, NumPy, and matplotlib have been used.

## 3. SOLUTION IMPLEMENTATION

This section discusses the implementation of the proposed solution.

### 3.1. Initializing the Grid Environment

A new Grid Environment is initialized using the *Grid.java* class. It denotes the context in which an agent will work. It holds a 2D array of AgentState objects that reflect the environment's states. The AgentState objects record state information such as the reward, whether the state is a wall or not, and so on. The *grid_builder()* method in this class is used to initialize the grid.

The *grid_builder()* function creates the grid by assigning rewards depending on the color of each state and identifying walls as inaccessible. The first loop assigns a -0.040 reward to each state in the grid (default value). The second loop, as described in *GridConstants.java*, assigns a reward of +1.000 to green cells. The third loop assigns a reward of -1.000 to brown cells and the fourth loop assigns a reward of 0 to all wall cells, setting their isWall attribute to true. This is illustrated in *Figure 3-1 –*

*Figure 3-1: Code Snippet: grid_builder() in Grid.java*

```java
/*
 * Initializes the grid by setting rewards for each state based on
 * their color, and marking walls as unreachable
 */
public void grid_builder() {
    // All cells start with -0.040 as reward
    for(int row = 0 ; row < GridConstants.TOTAL_NUM_ROWS ; row++) {
        for(int col = 0 ; col < GridConstants.TOTAL_NUM_COLS ; col++) {
            gridEnv[col][row] = new AgentState(IterationConstants.REWARD_WHITE);
        }
    }

    // Set +1.000 as the reward for green cells
    String[] green_cells = GridConstants.GREEN_CELLS.split(GridConstants.CELL_SEPARATOR);
    for(String green_cell : green_cells) {
        green_cell = green_cell.trim();
        String [] gridInfo = green_cell.split(GridConstants.ROW_COLUMN_SEPARATOR);
        int gridCol = Integer.parseInt(gridInfo[0]);
        int gridRow = Integer.parseInt(gridInfo[1]);
        gridEnv[gridCol][gridRow].setReward(IterationConstants.REWARD_GREEN);
    }

    // Set -1.000 as the reward for brown cells
    String[] brown_cells = GridConstants.BROWN_CELLS.split(GridConstants.CELL_SEPARATOR);
    for (String brown_cell : brown_cells) {

        brown_cell = brown_cell.trim();
        String[] gridInfo = brown_cell.split(GridConstants.ROW_COLUMN_SEPARATOR);
        int gridCol = Integer.parseInt(gridInfo[0]);
        int gridRow = Integer.parseInt(gridInfo[1]);
        gridEnv[gridCol][gridRow].setReward(IterationConstants.REWARD_BROWN);
    }

    // Set 0 for all the walls and mark them as unreachable - agent does not move
    String[] wall_cells = GridConstants.WALL_CELLS.split(GridConstants.CELL_SEPARATOR);
    for (String wall_cell : wall_cells) {

        wall_cell = wall_cell.trim();
        String[] gridInfo = wall_cell.split(GridConstants.ROW_COLUMN_SEPARATOR);
        int gridCol = Integer.parseInt(gridInfo[0]);
        int gridRow = Integer.parseInt(gridInfo[1]);
        gridEnv[gridCol][gridRow].setReward(IterationConstants.WALL_COLLISION);
        gridEnv[gridCol][gridRow].setAsWall(is_Wall: true);
    }
}
```

### 3.2. Utility Calculations

All the utility calculations are performed by the *UtilityController.java* class. It contains methods for calculating the utility of different actions, calculating the best utility for a state, and calculating the next set of utilities based on the Bellman equation.

The *calcBestUtil()* function considers the agent's current position, its existing utilities, and the current state of the grid. It computes and maintains the utility values of all potential actions (up, down, left, and right) in a list of Utility objects. The list of Utility objects is then sorted in decreasing order by utility value, and the Utility object with the greatest utility value is returned. This approach is essentially used to decide the optimum action to do at the present state based on the utility of the alternative options. This has been illustrated in *Figure 3-2* –

*Figure 3-2: Code Snippet: calcBestUtil() in UtilityController.java*

```java
/**
 * Calculates the utility all possible actions and returns action with maximum utility.
 *
 * @param col         The column index of the current state
 * @param row         The row index of the current state
 * @param curr_utils  The current utility array
 * @param grid        The grid of states
 * @return            The Utility object with the highest utility
 */

public static Utility calcBestUtil(final int col, final int row, final Utility[][] curr_utils, final AgentState[][] grid) {

    // Create a list to store the utilities for each action
    List<Utility> utils = new ArrayList<>();

    // Add the utilities for each possible action to the list
    utils.add(new Utility(AgentAction.UP, moveUpUtil(col, row, curr_utils, grid)));
    utils.add(new Utility(AgentAction.DOWN, moveDownUtil(col, row, curr_utils, grid)));
    utils.add(new Utility(AgentAction.LEFT, moveLeftUtil(col, row, curr_utils, grid)));
    utils.add(new Utility(AgentAction.RIGHT, moveRightUtil(col, row, curr_utils, grid)));

    // Sort the list of utilities based on the utility value
    Collections.sort(utils);

    // Return the Utility object with the maximum utility
    Utility best_util = utils.get(index: 0);
    return best_util;

}
```

The *calcActionUtil()* method takes as input an action, the current state, the current utility array for the action, and the grid of states, and calculates the utility of the given action for the current state. The method first initializes a Utility object to null. It then uses a switch statement to determine the type of the input *AgentAction* and calls a corresponding helper method (*moveUpUtil(), etc)* to

calculate the utility value for the given action. Finally, the method returns a new Utility object with the input *AgentAction* and the calculated utility value. This has been illustrated in *Figure 3-3* –

*Figure 3-3: Code Snippet: calcActionUtil() in UtilityController.java*

```java
/**
 * Calculates utility an action.
 *
 * @param action        The action to calculate the utility for
 * @param col           The column index of the current state
 * @param row           The row index of the current state
 * @param action_utils  The current utility array for the action
 * @param grid          The grid of states
 * @return              The Utility object for the given action
 */

public static Utility calcActionUtil(final AgentAction action, final int col, final int row, final Utility[][] action_utils,
final AgentState[][] grid) {

    Utility action_util = null;

    switch (action) {
        case UP:
        action_util = new Utility(AgentAction.UP, UtilityController.moveUpUtil(col, row, action_utils, grid));
        break;

        case DOWN:
        action_util = new Utility(AgentAction.DOWN, UtilityController.moveDownUtil(col, row, action_utils, grid));
        break;

        case LEFT:
        action_util = new Utility(AgentAction.LEFT, UtilityController.moveLeftUtil(col, row, action_utils, grid));
        break;

        case RIGHT:
        action_util = new Utility(AgentAction.RIGHT, UtilityController.moveRightUtil(col, row, action_utils, grid));
        break;
    }

    return action_util;

}
```

The *calcNextUtil()* method takes as input the current utility array and the grid of states and uses the Bellman equation to calculate the next set of utilities. This method is responsible for computing the utilities for each state in the grid, given the previous set of utilities and the state of the grid. It does this by repeatedly updating the utility values until convergence is reached, using the Bellman equation and the current policy. This has been illustrated in *Figure 3-4* –

*Figure 3-4: Code Snippet: calcNextUtil() in UtilityController.java*

```java
/**
 * Use Bellman Equation to calculate the next utility value.
 *
 * @param utils     The current utility list
 * @param grid      The grid of states
 * @return          The next utility list
 */

public static Utility[][] calcNextUtil(final Utility[][] utils, final AgentState[][] grid) {

    Utility[][] curr_utils = new Utility[GridConstants.TOTAL_NUM_COLS][GridConstants.TOTAL_NUM_ROWS];

    // Initializes the current utility array with new utility objects
    for (int col = 0; col < GridConstants.TOTAL_NUM_COLS; col++) {
        for (int row = 0; row < GridConstants.TOTAL_NUM_ROWS; row++) {
            curr_utils[col][row] = new Utility();
        }
    }

    Utility[][] new_utils = new Utility[GridConstants.TOTAL_NUM_COLS][GridConstants.TOTAL_NUM_ROWS];

    // Copies the utility and action values from the previous utility array to the new utility array
    for (int col = 0; col < GridConstants.TOTAL_NUM_COLS; col++) {
        for (int row = 0; row < GridConstants.TOTAL_NUM_ROWS; row++) {
            new_utils[col][row] = new Utility(utils[col][row].getAction(), utils[col][row].getUtil());
        }
    }

    int k = 0;
    do {
        UtilityController.updateUtils(new_utils, curr_utils);

        // Updates the utility for each state based on the action stated in the policy
        for (int row = 0; row < GridConstants.TOTAL_NUM_ROWS; row++) {
            for (int col = 0; col < GridConstants.TOTAL_NUM_COLS; col++) {
                if (!grid[col][row].isWall()) {
                    AgentAction action = curr_utils[col][row].getAction();
                    new_utils[col][row] = UtilityController.calcActionUtil(action, col, row, curr_utils, grid);
                }
            }
        }
        k++;
    } while(k < IterationConstants.K);
    return new_utils;

}
```

### 3.3. Value Iteration Algorithm

The implementation of this algorithm can be found in the *ValueIteration.java* class. Initially, two utility vectors are formed to hold the present utilities of states as well as the updated utilities of states after each iteration. All states will be started with utility values of 0.0 and unknown actions will be deemed null. Next, a utility list is generated to assist in recording the estimated utility

values for all states after each repetition. After executing the method, the list is saved in a CSV file to display the utility estimates as a function of the number of iterations. The delta is set to the default minimum double value and is updated when the newly updated delta is larger than the default value. The convergence threshold is computed and utilized as the stopping condition for the Value Iteration method. The constants for the algorithm have been defined in the *IterationConstants.java* and *GridConstants.java* files.

It is worth noting that the agent's starting state has no impact on the determined optimal policy because there is no definite time or a predetermined deadline for determining the best decision to be made. The current utility vector will be updated at the start of each iteration using the best utility values obtained thus far. The current utility vector, which comprises the estimated utility values for all states, will then be added to the utility list to maintain track of utilities for all states at each iteration for showing utility estimates as a function of iteration number. The best utility for each state is then computed during the current iteration. This is accomplished by selecting the action that maximizes the predicted utility of the next state, which is assisted by the *UtilityController.java* class (described in Section 3.2).

The new delta, which is the difference between the best new utility and the present utility, is determined for each state. If the new delta value is larger than the current delta value, the current one will be equated to the new delta value. The preceding process will be repeated for all states, and the program will then check to see if the current delta value is smaller than the convergence threshold. If the value of the delta is less than the threshold, the terminating condition has been met. It also implies that the use of any state in the maze environment no longer varies significantly in relation to Epsilon. Otherwise, the algorithm will go on to the next iteration and repeat the operation until the delta value is sufficiently minimal.

The value iteration algorithm has been illustrated in *Figure 3-5*:

*Figure 3-5: Code Snippet: Value Iteration Algorithm*

```
    Utility[][] currUtilArr = new Utility[globals.GridConstants.TOTAL_NUM_COLS][globals.GridConstants.TOTAL_NUM_ROWS];
    Utility[][] newUtilArr = new Utility[globals.GridConstants.TOTAL_NUM_COLS][globals.GridConstants.TOTAL_NUM_ROWS];

    // Initialize default utilities for each state
    for (int col = 0; col < globals.GridConstants.TOTAL_NUM_COLS; col++) {
        for (int row = 0; row < globals.GridConstants.TOTAL_NUM_ROWS; row++) {
            newUtilArr[col][row] = new Utility();
        }
    }

    utilityList = new ArrayList<>();

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

    convergeThreshold = globals.IterationConstants.MAXIMUM_ALLOWABLE_DISCOUNTED_ERROR *
    ((1.000 - globals.IterationConstants.DISCOUNT_FACTOR) / globals.IterationConstants.DISCOUNT_FACTOR);

    // Initialize number of iterations
    do {

        UtilityController.updateUtils(newUtilArr, currUtilArr);

        delta = Double.MIN_VALUE;

        // Append to list of Utility a copy of the existing actions & utilities
        Utility[][] currUtilArrCopy =
        new Utility[globals.GridConstants.TOTAL_NUM_COLS][globals.GridConstants.TOTAL_NUM_ROWS];
        UtilityController.updateUtils(currUtilArr, currUtilArrCopy);
        utilityList.add(currUtilArrCopy);

        // For each state
        for(int row = 0 ; row < globals.GridConstants.TOTAL_NUM_ROWS ; row++) {
            for(int col = 0 ; col < globals.GridConstants.TOTAL_NUM_COLS ; col++) {

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

### 3.4. Policy Iteration Algorithm

The implementation of this algorithm can be found in *PolicyIteration.java* class. At each cycle, one vector of utility is initialized to record the current utilities of states, and one policy vector indexed by the state is established to hold the updated utilities of states. The policy vector's states will all be started with utility values of 0.0 and a random action.

During each cycle, a utility list is constructed to assist record the projected utility values for all states. When the policy improvement provides no change in the utilities, a Boolean flag is utilized to determine if the algorithm should be terminated. The new utility values are computed using the Bellman equation based on the present utilities and actions of all states. To compute a sufficiently reasonable approximation of the utility, ‘K’ value iteration steps are conducted.

First, the current utility list is set to 0.0, which temporarily holds utilities while a simpler "Bellman update" is performed. The current policy actions and utility values are used to generate a new utility vector, and its states are initialized. The current utility array will be changed at the start of every Kth iteration. The new utility of each state is determined using the *UtilityController* class method at each Kth iteration depending on the action.

After calculating the estimated utilities of all states, the utility value for selecting the action that maximizes the future state is determined. It is then compared to the projected utility to assess if the existing policy is adequate. If the utility obtained by executing the optimal action exceeds the current predicted utility value, the policy of that specific state will be modified with the new policy that may possibly maximize its succeeding states. The preceding stages will be repeated until the algorithm has discovered the optimum policy.

The policy iteration algorithm has been illustrated in *Figure 3-6*:

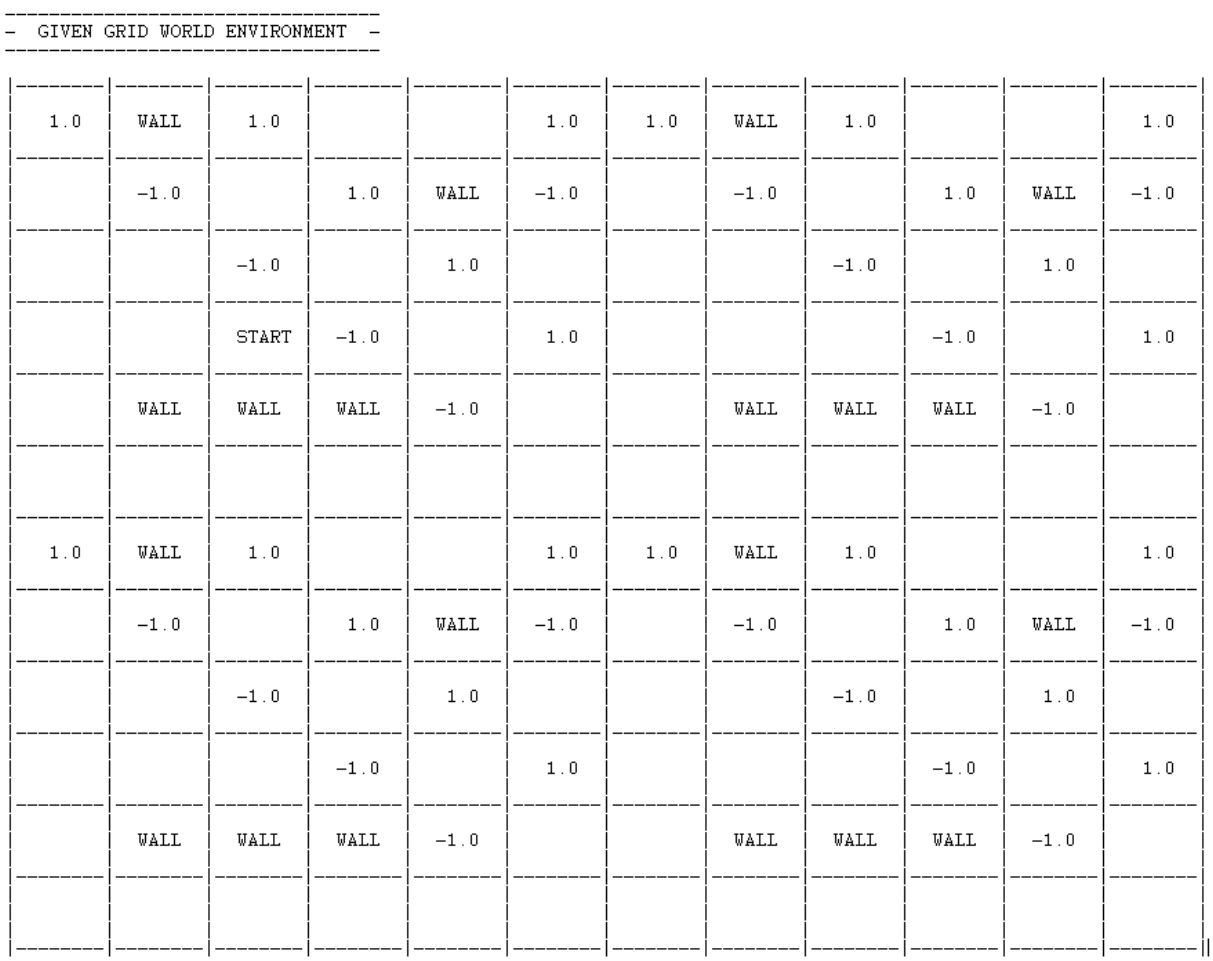
*Figure 3-6: Code Snippet: Policy Iteration Algorithm*

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

    Utility[][] currUtilArr = new Utility[globals.GridConstants.TOTAL_NUM_COLS][globals.GridConstants.TOTAL_NUM_ROWS];
    Utility[][] newUtilArr = new Utility[globals.GridConstants.TOTAL_NUM_COLS][globals.GridConstants.TOTAL_NUM_ROWS];

    // Initialize default utilities and policies for each state
    for (int col = 0; col < globals.GridConstants.TOTAL_NUM_COLS; col++) {
        for (int row = 0; row < globals.GridConstants.TOTAL_NUM_ROWS; row++) {
            newUtilArr[col][row] = new Utility();
            if (!grid[col][row].isWall()) {
                AgentAction randomAction = AgentAction.getRandomAction();
                newUtilArr[col][row].setAction(randomAction);
            }
        }
    }

    // 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;

    do {

        UtilityController.updateUtils(newUtilArr, currUtilArr);

        // Append to list of Utility a copy of the existing actions & utilities
        Utility[][] currUtilArrCopy =
        new Utility[globals.GridConstants.TOTAL_NUM_COLS][globals.GridConstants.TOTAL_NUM_ROWS];
        UtilityController.updateUtils(currUtilArr, currUtilArrCopy);
        utilityList.add(currUtilArrCopy);


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

        unchanged = true;

        // For each state - Policy improvement
        for (int row = 0; row < globals.GridConstants.TOTAL_NUM_ROWS; row++) {
            for (int col = 0; col < globals.GridConstants.TOTAL_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 =
                    UtilityController.calcBestUtil(col, row, newUtilArr, grid);

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

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

    } while (!unchanged);
}
```

### 3.5. Part 2: Building a more complex maze

A more complex maze environment was constructed by scaling the given grid by different factors. This was performed by simply changing the number of rows and column constants in the *GridConstants.java* file. A separate function is added to the *Grid.java* file to duplicate the created grid environment with the walls, green cells, and brown cells. As a result, the number of states in the environment will expand. That suggests that the two methods will demand more computation time and memory to run.

*Figure 3-7: New Complex Maze (scaled by 2)*



# 4. RESULTS DISCUSSION

## 4.1. Part 1: Value Iteration

Results have been obtained using different values of c [0.01, 0.1, 1, 5, 10, 20, 40, 80]. The other constants used in the algorithm have been defined in *IterationConstants.java*. The figures below present the results for c values 0.01, 10, and 80. The remaining outputs have been added to the respective output sub-folders in the outputs/value_iteration_ouputs directory –

*Figure 4-1: c=0.01*

```
----------------------
-  IMPORTANT VALUES  -
----------------------

DISCOUNT FACTOR          :        0.99
UPPER BOUND OF UTILITY   :        100.00
MAXIMUM REWARD   :       1.0
CONSTANT 'c'             :        0.01
EPSILON (c * Rmax)       :        0.01
CONVERGENCE THRESHOLD    :        0.00010


-------------------------
- NUMBER OF ITERATIONS -
-------------------------

TOTAL NUMBER OF ITERATIONS: 917


-----------------------------------------
-  OUTPUT: UTILITY VALUES OF ALL STATES  -
-----------------------------------------

(0, 0): 99.989958
(0, 1): 98.383319
(0, 2): 96.938458
(0, 3): 95.543797
(0, 4): 94.302477
(0, 5): 92.927432
(1, 1): 95.872975
(1, 2): 95.576385
(1, 3): 94.442451
(1, 5): 91.718735
(2, 0): 95.035415
(2, 1): 94.534956
(2, 2): 93.284385
(2, 3): 93.222503
(2, 5): 90.525110
(3, 0): 93.864959
(3, 1): 94.387672
(3, 2): 93.166231
(3, 3): 91.105214
(3, 5): 89.346367
(4, 0): 92.644572
(4, 2): 93.092327
(4, 3): 91.804365
(4, 4): 89.538371
(4, 5): 88.559057
(5, 0): 93.318461
(5, 1): 90.907881
(5, 2): 91.784829
(5, 3): 91.878042
(5, 4): 90.556723
(5, 5): 89.287648
```

```
--------------------------------
-  OUTPUT: OPTIMAL POLICY PLOT  -
--------------------------------

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

--------------------------------
-  MAP OF ALL STATES' UTILITIES  -
--------------------------------

|--------|--------|--------|--------|--------|--------|
| 99.990 | 00.000 | 95.035 | 93.865 | 92.645 | 93.318 |
|--------|--------|--------|--------|--------|--------|
| 98.383 | 95.873 | 94.535 | 94.388 | 00.000 | 90.908 |
|--------|--------|--------|--------|--------|--------|
| 96.938 | 95.576 | 93.284 | 93.166 | 93.092 | 91.785 |
|--------|--------|--------|--------|--------|--------|
| 95.544 | 94.442 | 93.223 | 91.105 | 91.804 | 91.878 |
|--------|--------|--------|--------|--------|--------|
| 94.302 | 00.000 | 00.000 | 00.000 | 89.538 | 90.557 |
|--------|--------|--------|--------|--------|--------|
| 92.927 | 91.719 | 90.525 | 89.346 | 88.559 | 89.288 |
|--------|--------|--------|--------|--------|--------|
```

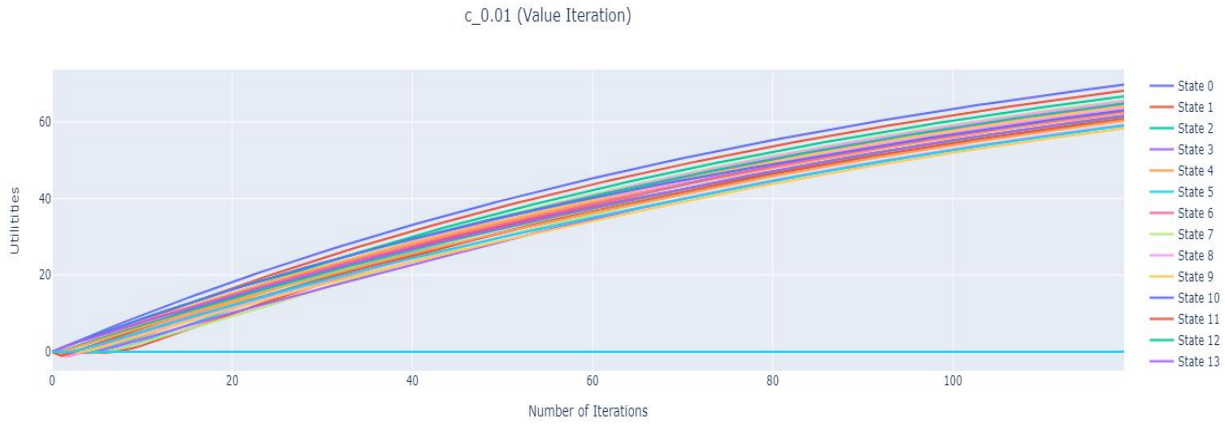



*Figure 4-2: c=10*

```
------------------------
-  IMPORTANT VALUES  -
------------------------

DISCOUNT FACTOR          :        0.99
UPPER BOUND OF UTILITY   :        100.00
MAXIMUM REWARD   :       1.0
CONSTANT 'c'             :        10.0
EPSILON (c * Rmax)       :        10.0
CONVERGENCE THRESHOLD    :        0.10101


--------------------------
- NUMBER OF ITERATIONS -
--------------------------

TOTAL NUMBER OF ITERATIONS: 230


-------------------------------------------
-  OUTPUT: UTILITY VALUES OF ALL 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
```

```
---------------------------------
-  OUTPUT: OPTIMAL POLICY PLOT  -
---------------------------------

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


---------------------------------
-  MAP OF ALL STATES' UTILITIES  -
---------------------------------

|--------|--------|--------|--------|--------|--------|
| 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 |
|--------|--------|--------|--------|--------|--------|
```

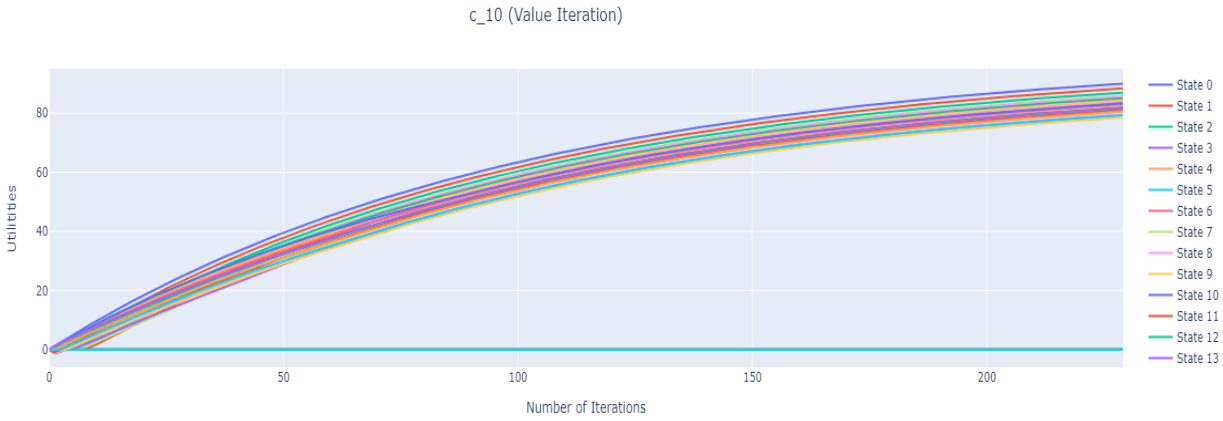
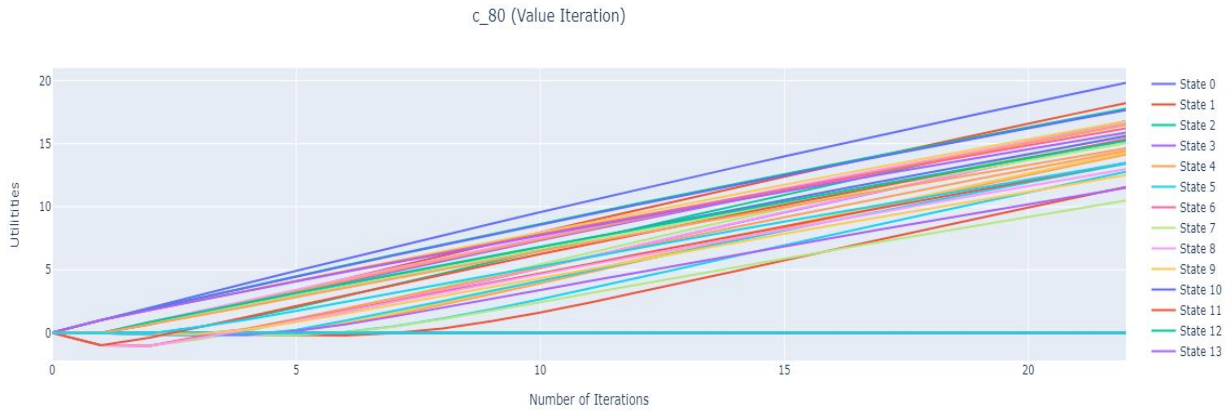



*Figure 4-3: c=80*

```
----------------------
-  IMPORTANT VALUES  -
----------------------

DISCOUNT FACTOR          :        0.99
UPPER BOUND OF UTILITY   :        100.00
MAXIMUM REWARD   :       1.0
CONSTANT 'c'             :        80.0
EPSILON (c * Rmax)       :        80.0
CONVERGENCE THRESHOLD    :        0.80808


-------------------------
- NUMBER OF ITERATIONS -
-------------------------

TOTAL NUMBER OF ITERATIONS: 23


-----------------------------------------
-  OUTPUT: UTILITY VALUES OF ALL STATES  -
-----------------------------------------

(0, 0): 19.836941
(0, 1): 18.230305
(0, 2): 16.785451
(0, 3): 15.390811
(0, 4): 14.149532
(0, 5): 12.774658
(1, 1): 15.719968
(1, 2): 15.423396
(1, 3): 14.289496
(1, 5): 11.566259
(2, 0): 17.766159
(2, 1): 16.531483
(2, 2): 14.439978
(2, 3): 13.416324
(2, 5): 10.503323
(3, 0): 16.630969
(3, 1): 16.791445
(3, 2): 15.598628
(3, 3): 13.475970
(3, 5): 11.496793
(4, 0): 16.518652
(4, 2): 16.238090
(4, 3): 15.055245
(4, 4): 12.990432
(4, 5): 12.506339
(5, 0): 17.672925
(5, 1): 15.321644
(5, 2): 15.236619
(5, 3): 15.892498
(5, 4): 14.630783
(5, 5): 13.471694
```

```
-----------------------------------
-  OUTPUT: OPTIMAL POLICY PLOT  -
-----------------------------------

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


-----------------------------------
-  MAP OF ALL STATES' UTILITIES  -
-----------------------------------

|--------|--------|--------|--------|--------|--------|
| 19.837 | 00.000 | 17.766 | 16.631 | 16.519 | 17.673 |
|--------|--------|--------|--------|--------|--------|
| 18.230 | 15.720 | 16.531 | 16.791 | 00.000 | 15.322 |
|--------|--------|--------|--------|--------|--------|
| 16.785 | 15.423 | 14.440 | 15.599 | 16.238 | 15.237 |
|--------|--------|--------|--------|--------|--------|
| 15.391 | 14.289 | 13.416 | 13.476 | 15.055 | 15.892 |
|--------|--------|--------|--------|--------|--------|
| 14.150 | 00.000 | 00.000 | 00.000 | 12.990 | 14.631 |
|--------|--------|--------|--------|--------|--------|
| 12.775 | 11.566 | 10.503 | 11.497 | 12.506 | 13.472 |
|--------|--------|--------|--------|--------|--------|
```




From the figures above, as the value of c increases, the convergence threshold increases, and the number of iterations decreases. It can also be observed that the output of optimal policy for c = 80 is different from the smaller values of c, where the agent seems to be colliding with the wall. That might be because the maximum error permitted has been increased to 80, causing the agent to make mistakes in selecting its best option. Moreover, while the number of iterations necessary to identify the best policy is lowered to 23 (in c = 80), the quality of decision-making suffers.

Furthermore, at each iteration step, the utility value for each state (excluding the wall) is adjusted using the Bellman algorithm. Because the discount factor is employed in the algorithm, the method is guaranteed to always converge to a unique solution of the Bellman equations when the discount factor is less than 1. As a result, in the above plots (graphs), all the predicted utility values steadily grow as the number of iterations increases until they approach an equilibrium where they begin to level out.

### 4.2. Part 1: Policy Iteration

Results have been obtained using different values of K [1, 5, 10, 20, 30, 40, 50, 80]. The other constants used in the algorithm have been defined in *IterationConstants.java*. The figures below present the results for K values 1, 20, and 80. The remaining outputs have been added to the respective output sub-folders in the outputs/policy_iteration_ouputs directory –

*Figure 4-4: K=1*

```
-----------------------
-  IMPORTANT VALUES  -
-----------------------

DISCOUNT FACTOR          :        0.99
UPPER BOUND OF UTILITY   :        100.00
MAXIMUM REWARD   :       1.0
CONSTANT 'c'             :        1.0
EPSILON (c * Rmax)       :        1.0
CONVERGENCE THRESHOLD    :        0.00000


-------------------------
- NUMBER OF ITERATIONS -
-------------------------

TOTAL NUMBER OF ITERATIONS: 7


-------------------------------------------
-  OUTPUT: UTILITY VALUES OF ALL STATES  -
-------------------------------------------

(0, 0): 5.8519851
(0, 1): 4.2651986
(0, 2): 2.8858371
(0, 3): 1.6591066
(0, 4): 0.70292326
(0, 5): 0.047352789
(1, 1): 1.8206828
(1, 2): 1.6638436
(1, 3): 0.77846975
(1, 5): -0.23407940
(2, 0): 5.3417235
(2, 1): 4.0202167
(2, 2): 1.9030151
(2, 3): 0.95595459
(2, 5): 0.033945084
(3, 0): 4.2655593
(3, 1): 4.9023365
(3, 2): 3.7445997
(3, 3): 1.7394146
(3, 5): 0.65506195
(4, 0): 4.1497362
(4, 2): 4.8578074
(4, 3): 3.7127864
(4, 4): 1.7223132
(4, 5): 1.5166637
(5, 0): 5.3040081
(5, 1): 2.9528008
(5, 2): 3.9065842
(5, 3): 4.8195180
(5, 4): 3.5640896
(5, 5): 2.4491057
```

```
---------------------------------
-  OUTPUT: OPTIMAL POLICY PLOT  -
---------------------------------

|--------|--------|--------|--------|--------|--------|
|   ^    |  Wall  |   ^    |   <    |   >    |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   ^    |   >    |  Wall  |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   ^    |   ^    |   ^    |   v    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   ^    |   ^    |   >    |   ^    |   >    |
|--------|--------|--------|--------|--------|--------|
|   ^    |  Wall  |  Wall  |  Wall  |   ^    |   ^    |
|--------|--------|--------|--------|--------|--------|
|   ^    |   <    |   >    |   >    |   >    |   ^    |
|--------|--------|--------|--------|--------|--------|

---------------------------------
-  MAP OF ALL STATES' UTILITIES  -
---------------------------------

|--------|--------|--------|--------|--------|--------|
| 05.852 | 00.000 | 05.342 | 04.266 | 04.150 | 05.304 |
|--------|--------|--------|--------|--------|--------|
| 04.265 | 01.821 | 04.020 | 04.902 | 00.000 | 02.953 |
|--------|--------|--------|--------|--------|--------|
| 02.886 | 01.664 | 01.903 | 03.745 | 04.858 | 03.907 |
|--------|--------|--------|--------|--------|--------|
| 01.659 | 00.778 | 00.956 | 01.739 | 03.713 | 04.820 |
|--------|--------|--------|--------|--------|--------|
| 00.703 | 00.000 | 00.000 | 00.000 | 01.722 | 03.564 |
|--------|--------|--------|--------|--------|--------|
| 00.047 | -00.23 | 00.034 | 00.655 | 01.517 | 02.449 |
|--------|--------|--------|--------|--------|--------|
```

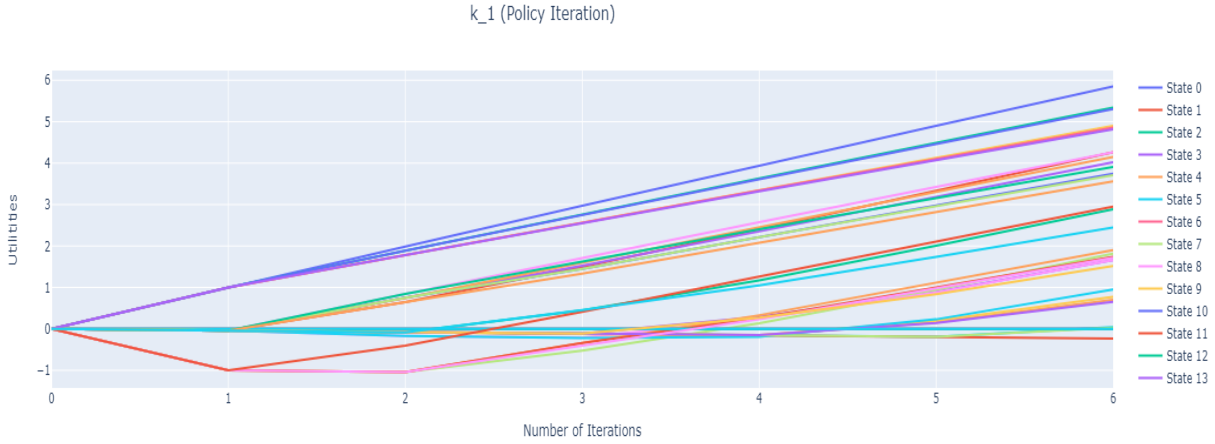


*Figure 4-5: K=20*

```
----------------------
-  IMPORTANT VALUES  -
----------------------

DISCOUNT FACTOR          :        0.99
UPPER BOUND OF UTILITY   :        100.00
MAXIMUM REWARD   :       1.0
CONSTANT 'c'             :        1.0
EPSILON (c * Rmax)       :        1.0
CONVERGENCE THRESHOLD    :        0.00000


-------------------------
- NUMBER OF ITERATIONS -
-------------------------

TOTAL NUMBER OF ITERATIONS: 6


-------------------------------------------
-  OUTPUT: UTILITY VALUES OF ALL STATES  -
-------------------------------------------

(0, 0): 62.059944
(0, 1): 60.453305
(0, 2): 59.008444
(0, 3): 57.613783
(0, 4): 56.372463
(0, 5): 54.997418
(1, 1): 57.942961
(1, 2): 57.646371
(1, 3): 56.512437
(1, 5): 53.788721
(2, 0): 57.107434
(2, 1): 56.605168
(2, 2): 55.354396
(2, 3): 55.292492
(2, 5): 52.595096
(3, 0): 55.937008
(3, 1): 56.458131
(3, 2): 55.236651
(3, 3): 53.141718
(3, 5): 51.416353
(4, 0): 54.716880
(4, 2): 55.162417
(4, 3): 53.871290
(4, 4): 51.605843
(4, 5): 50.401040
(5, 0): 55.398280
(5, 1): 52.988732
(5, 2): 53.856243
(5, 3): 53.949125
(5, 4): 52.627571
(5, 5): 51.333487
```

```
---------------------------------
-  OUTPUT: OPTIMAL POLICY PLOT  -
---------------------------------
```

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

```
---------------------------------
-  MAP OF ALL STATES' UTILITIES  -
---------------------------------
```

| | | | | | |
|---|---|---|---|---|---|
| 62.060 | 00.000 | 57.107 | 55.937 | 54.717 | 55.398 |
| 60.453 | 57.943 | 56.605 | 56.458 | 00.000 | 52.989 |
| 59.008 | 57.646 | 55.354 | 55.237 | 55.162 | 53.856 |
| 57.614 | 56.512 | 55.292 | 53.142 | 53.871 | 53.949 |
| 56.372 | 00.000 | 00.000 | 00.000 | 51.606 | 52.628 |
| 54.997 | 53.789 | 52.595 | 51.416 | 50.401 | 51.333 |

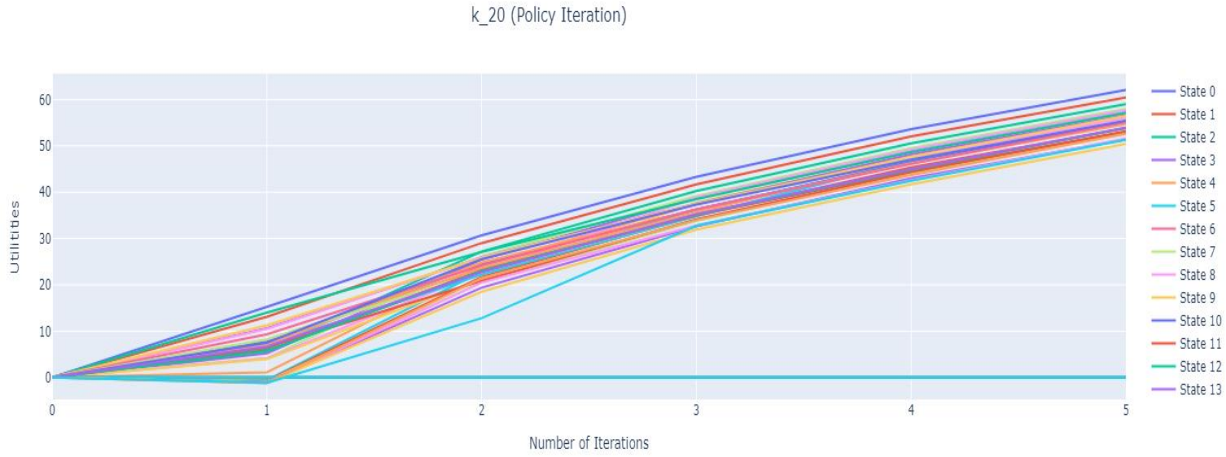


*Figure 4-6: K=80*

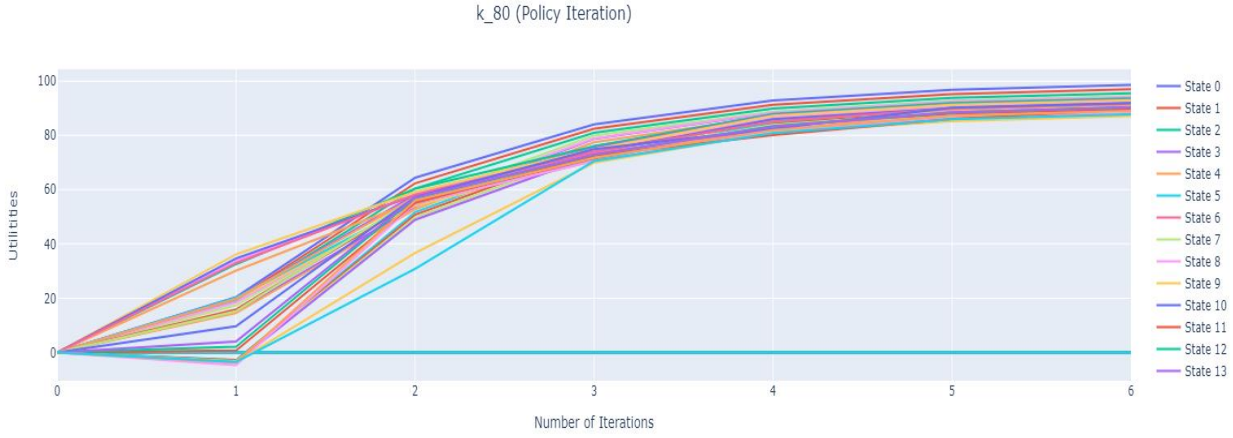
```
----------------------
-  IMPORTANT VALUES  -
----------------------

DISCOUNT FACTOR          :        0.99
UPPER BOUND OF UTILITY   :        100.00
MAXIMUM REWARD  :        1.0
CONSTANT 'c'             :        1.0
EPSILON (c * Rmax)       :        1.0
CONVERGENCE THRESHOLD    :        0.00000


-------------------------
- NUMBER OF ITERATIONS -
-------------------------

TOTAL NUMBER OF ITERATIONS: 7


-----------------------------------------
-  OUTPUT: UTILITY VALUES OF ALL STATES  -
-----------------------------------------

(0, 0): 98.570703
(0, 1): 96.964065
(0, 2): 95.519203
(0, 3): 94.124542
(0, 4): 92.883223
(0, 5): 91.508177
(1, 1): 94.453721
(1, 2): 94.157131
(1, 3): 93.023197
(1, 5): 90.299481
(2, 0): 93.616160
(2, 1): 93.115701
(2, 2): 91.865131
(2, 3): 91.803249
(2, 5): 89.105855
(3, 0): 92.445702
(3, 1): 92.968397
(3, 2): 91.746767
(3, 3): 89.684210
(3, 5): 87.927113
(4, 0): 91.225315
(4, 2): 91.671130
(4, 3): 90.369104
(4, 4): 88.090758
(4, 5): 87.105655
(5, 0): 91.899204
(5, 1): 89.488624
(5, 2): 90.349741
(5, 3): 90.314397
(5, 4): 89.007430
(5, 5): 87.750335
```

```
---------------------------------
-  OUTPUT: OPTIMAL POLICY PLOT  -
---------------------------------
```

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

```
-----------------------------------
-  MAP OF ALL STATES' UTILITIES  -
-----------------------------------
```

| | | | | | |
|---|---|---|---|---|---|
| 98.571 | 00.000 | 93.616 | 92.446 | 91.225 | 91.899 |
| 96.964 | 94.454 | 93.116 | 92.968 | 00.000 | 89.489 |
| 95.519 | 94.157 | 91.865 | 91.747 | 91.671 | 90.350 |
| 94.125 | 93.023 | 91.803 | 89.684 | 90.369 | 90.314 |
| 92.883 | 00.000 | 00.000 | 00.000 | 88.091 | 89.007 |
| 91.508 | 90.299 | 89.106 | 87.927 | 87.106 | 87.750 |



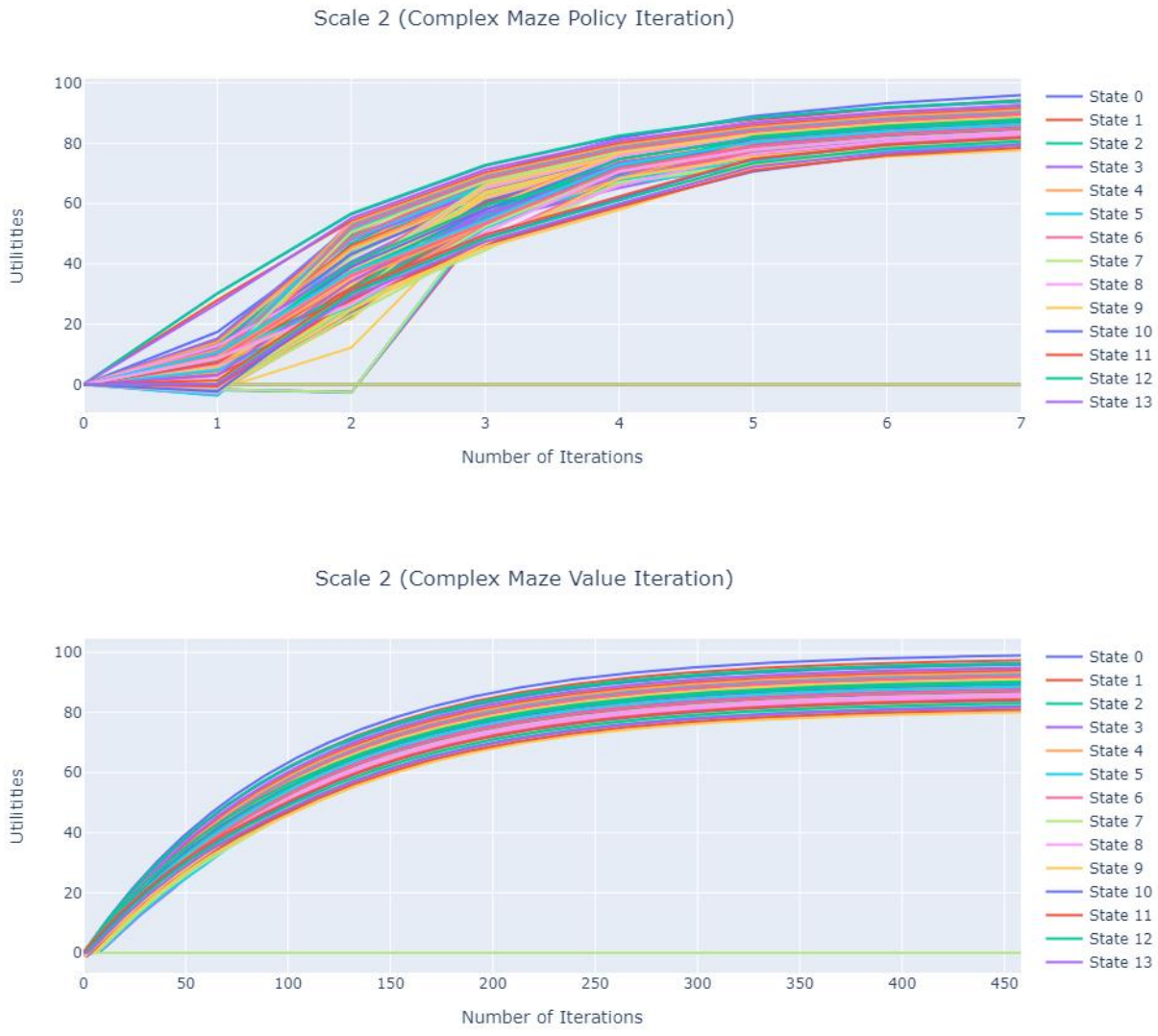
From the figures above, as the value of K increases, the final estimated utilities are seen to increase. It is most noticed when K increases from 1 to 20. This increase plateaus as the value of K increases further. Since the upper bound has been set to 100, the utility value never crosses this set number. It is worth noting that the equations used to obtain the new utilities are now linear because the "max" operator has been eliminated. To discover a sufficiently reasonable estimate of the utility, "K" simple value iteration steps are conducted. The resultant method outperforms value iteration in terms of efficiency. As a result, just 6 to 8 iterations are needed to determine the best policy, which is significantly fewer than the number of iterations required by the Value Iteration method.

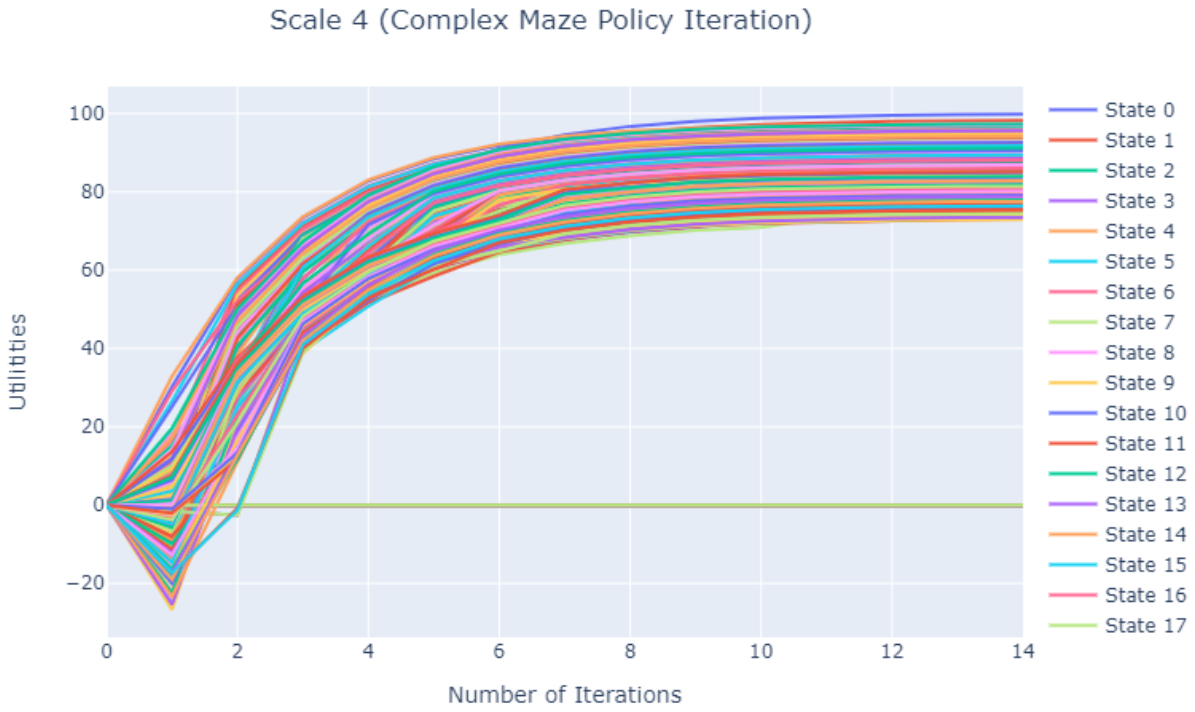
### 4.3. Part 2: Complex Maze

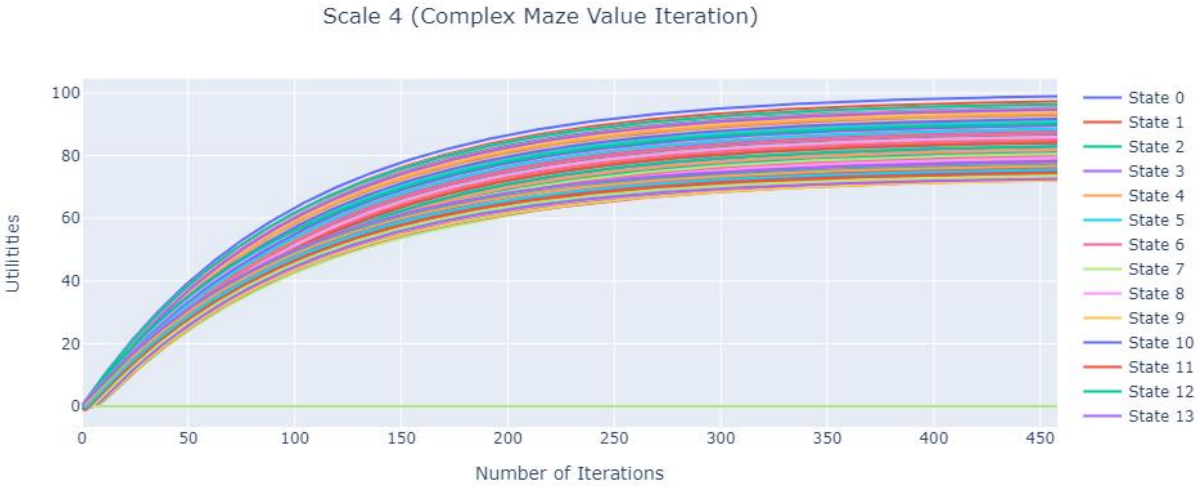
The results obtained for this part can be run and observed from the *ComplexMaze.java* file in the iterations sub-directory. Because both algorithms require updating the utility or policy for all states at once, increasing the size of the maze has a significant impact on the performance of the two algorithms in terms of both space complexity and time complexity, particularly when Bellman algorithms are performed at the start of each iteration. When scaling up the maze environment, it is discovered that both methods require a long time to identify the best policy. As a result, given the existing implementation, the larger the space complexity of the environment, the higher the time complexity of convergence in finding the best policy. This however can be mitigated by selecting any subset of states and applying either type of updating policy improvement or simplified value iteration to that subset – asynchronous policy iteration.

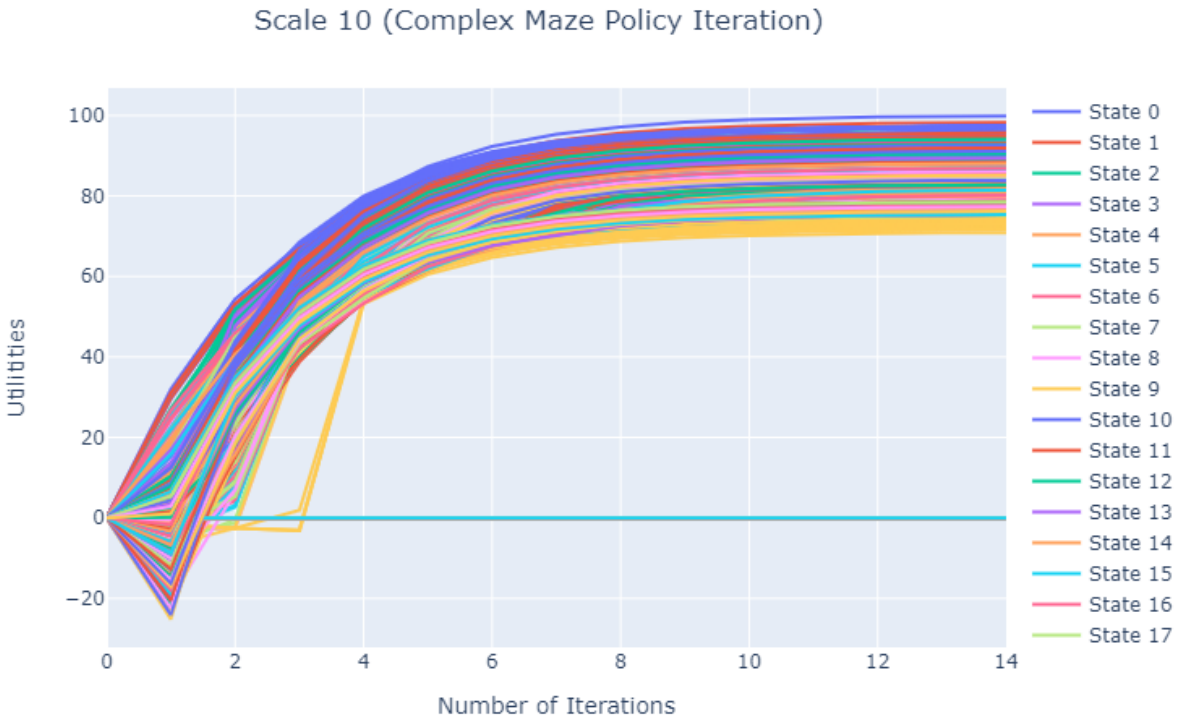
The plots below depict the outputs of value and policy iterations by scaling the grid to 2, 4 and 10 (the system crashes on scale 100). The values of c and K are set as 1 and 50 respectively –

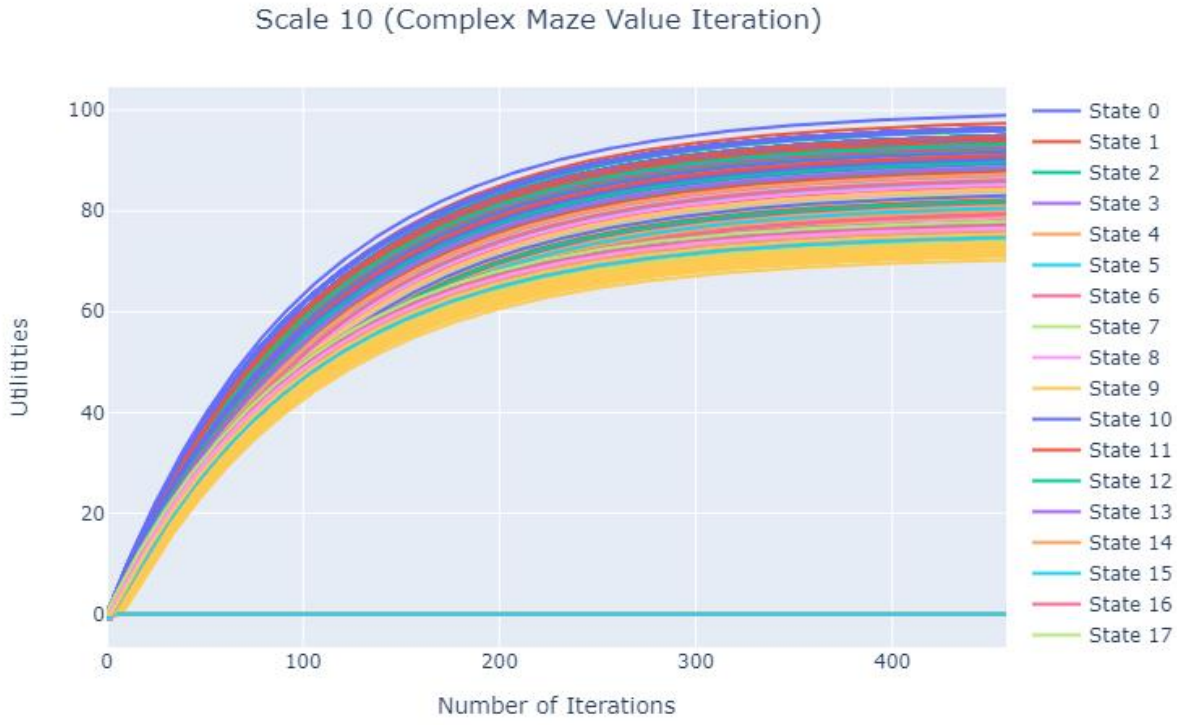
*Figure 4-7: Complex Maze: Outputs – Value and Policy Iterations*




Scale 4 (Complex Maze Policy Iteration)
Utilities
Number of Iterations
100
80
60
40
20
0
−20
0
2
4
6
8
10
12
14
State 0
State 1
State 2
State 3
State 4
State 5
State 6
State 7
State 8
State 9
State 10
State 11
State 12
State 13
State 14
State 15
State 16
State 17


Scale 4 (Complex Maze Value Iteration)
Utilities
Number of Iterations
100
80
60
40
20
0
0
50
100
150
200
250
300
350
400
450
State 0
State 1
State 2
State 3
State 4
State 5
State 6
State 7
State 8
State 9
State 10
State 11
State 12
State 13


Scale 10 (Complex Maze Policy Iteration)
Utilities
Number of Iterations
100
80
60
40
20
0
−20
0
2
4
6
8
10
12
14
State 0
State 1
State 2
State 3
State 4
State 5
State 6
State 7
State 8
State 9
State 10
State 11
State 12
State 13
State 14
State 15
State 16
State 17


Scale 10 (Complex Maze Value Iteration)
Utilities
Number of Iterations
100
80
60
40
20
0
0
100
200
300
400
State 0
State 1
State 2
State 3
State 4
State 5
State 6
State 7
State 8
State 9
State 10
State 11
State 12
State 13
State 14
State 15
State 16
State 17