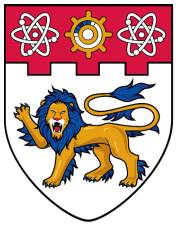

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4046 Intelligent Agents

Assignment 1

Name: Kyle Huang Junyuan

Matriculation: U1721717G

Table of Contents

Value Iteration (Part 1)

Description of Implemented Solutions

In Value Iteration, the utility of each state is first calculated using the Bellman equation and subsequently used to decide an optimal action.

The utility of a state is given by:

$$U(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, a)\, U(S')$$

The implementation can be briefly described through the following steps:

1. Initialise the utility of all states to 0 and their actions to *UP*
2. Import the maze environment
3. Calculate the threshold using the maximum error allowed and the discount factor
4. Repeat
    a. For each state: Calculate utility and set the action that maximises the expected utility of the subsequent state
    b. Find the maximum change in utility
5. Until maximum change exceeds the threshold

The following Figure 1 shows the code snippet of a Value Iteration run. It starts by defining a *threshold* and *maxChangeInUtility* value so that the do-while loop can be terminated once it crosses the predefined threshold.

The threshold value is defined by:

$$\begin{aligned} \boldsymbol{Threshold} &= \frac{\epsilon(1-\gamma)}{\gamma} \\ &= \frac{62(1-0.99)}{0.99} \\ &\approx 0.6262 \ldots \end{aligned}$$

Where ϵ represents the epsilon defined by:

$$\begin{aligned} \boldsymbol{Epsilon} &= C * Rmax \\ &= 62 * 1 \\ &= 62 \end{aligned}$$

In each iteration, the *calculateUtility* function is executed for every state in the maze and an action is set based on one that maximises the expected utility of the subsequent state. This function returns the maximum change in utility detected during the run and is then updated onto *maxChangeInUtility* if the value is larger. This helps to keep track of the overall maximum change in utility across all iterations.

```
private static void runValueIteration(Maze maze) {
    double threshold = EPSILON * ((1 - Constants.DISCOUNT_FACTOR) / Constants.DISCOUNT_FACTOR);
    double maxChangeInUtility = 0;
    int iteration = 1;

    do {
        maxChangeInUtility = 0;

        /* Runs 1 iteration */
        for (int c = 0; c < Constants.NUM_COL; c++) {
            for (int r = 0; r < Constants.NUM_ROW; r++) {
                Cell currCell = maze.getCell(new Coordinate(c, r));

                /* Find maximum change in utility */
                double changeInUtility = calculateUtility(currCell, maze);
                if (changeInUtility > maxChangeInUtility)
                    maxChangeInUtility = changeInUtility;
            }
        }

        iteration++;
    } while (maxChangeInUtility > threshold);
}
```

Figure 1: Code Snippet for Value Iteration

The following Figure 2 shows the code snippet of the *calculateUtility* function. It first iterates through all the 4 possible directions and stores them in the *subUtilities* array. The item with the maximum value is then identified using the *maxU* pointer so that the new action can be updated into each state (i.e. 0 = *UP*, 1 = *DOWN*, 2 = *LEFT*, 3 = *RIGHT*).

Subsequently, the calculated maximum expected utility and its action are updated into the state. The function will then return the difference between the previous and new utility values.

```
private static double calculateUtility(Cell currCell, Maze maze) {
    double[] subUtilities = new double[Coordinate.TOTAL_DIRECTIONS];

    /* 1. Find all possible utilities (i.e. 4 possible directions) */
    for (int dir = 0; dir < Coordinate.TOTAL_DIRECTIONS; dir++) {
        /* 1a. Sum up the 3 neighbours (i.e. UP, LEFT, RIGHT) */
        Cell[] neighbours = maze.getNeighboursOfCell(currCell, dir);
        double up = Constants.PROBABILITY_UP * neighbours[0].getUtility();
        double left = Constants.PROBABILITY_LEFT * neighbours[1].getUtility();
        double right = Constants.PROBABILITY_RIGHT * neighbours[2].getUtility();

        subUtilities[dir] = up + left + right;
    }

    /* 2. Find the maximum possible utility */
    int maxU = 0;
    for (int u = 1; u < subUtilities.length; u++) {
        if (subUtilities[u] > subUtilities[maxU])
            maxU = u;
    }

    /* 3. Set utility & policy of current cell */
    float currReward = currCell.getCellType().getReward();
    double prevUtility = currCell.getUtility();
    double newUtility = currReward + Constants.DISCOUNT_FACTOR * subUtilities[maxU];
    currCell.setUtility(newUtility);
    currCell.setPolicy(maxU);

    /* 4. Return the difference of prevUtility & newUtility */
    return (Math.abs(prevUtility - newUtility));
}
```

Figure 2: Code Snippet of calculateUtility function

Plot of Optimal Policy

The following Figure 3 shows the final plot of the optimal policies of all states after convergence on the **47th iteration** by using a C value of 62.

One may expect the optimal policy at (Col: 3, Row: 1) to be pointing *RIGHT* so that taking the next action will allow the agent to remain at the green state (since a wall is towards the right). However, the calculated optimal policy is pointing *UP*. A possible explanation here is that the state (3, 0) still has a higher possible maximum expected utility than compared to a *RIGHT* action.

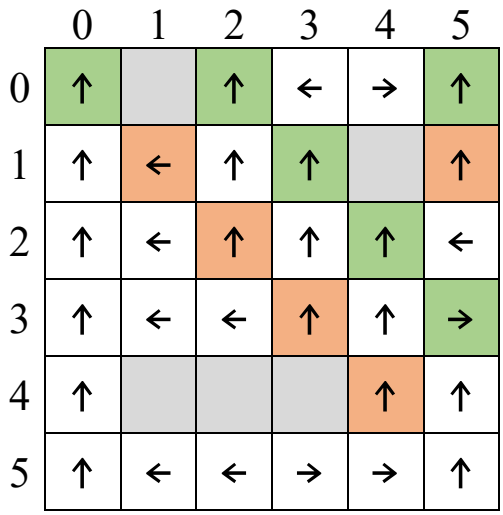

Figure 3: Plot of Optimal Policy for Value Iteration with C = 62

Utility of all States

The following Figure 4 shows the final utilities of all states after convergence on the **47th iteration** by using a C value of 62.

	0	1	2	3	4	5
0	38.271		36.707	36.137	35.560	36.777
1	37.368	35.552	36.137	36.783		34.932
2	36.618	35.926	34.696	36.117	36.817	36.170
3	35.901	35.395	34.816	34.574	36.135	36.841
4	35.256				34.497	36.070
5	34.544	33.920	33.304	33.164	34.307	35.372

Figure 4: Utility of all states for Value Iteration with C = 62

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

The following Figure 5 shows the plot of utility estimates as a function of the number of iterations. The values in this graph have been normalised to better depict how utility values are changing across iterations.

As compared to the Policy Iteration algorithm which converged at the 11th iteration, this Value Iteration run only converged much later at the **47th iteration**.

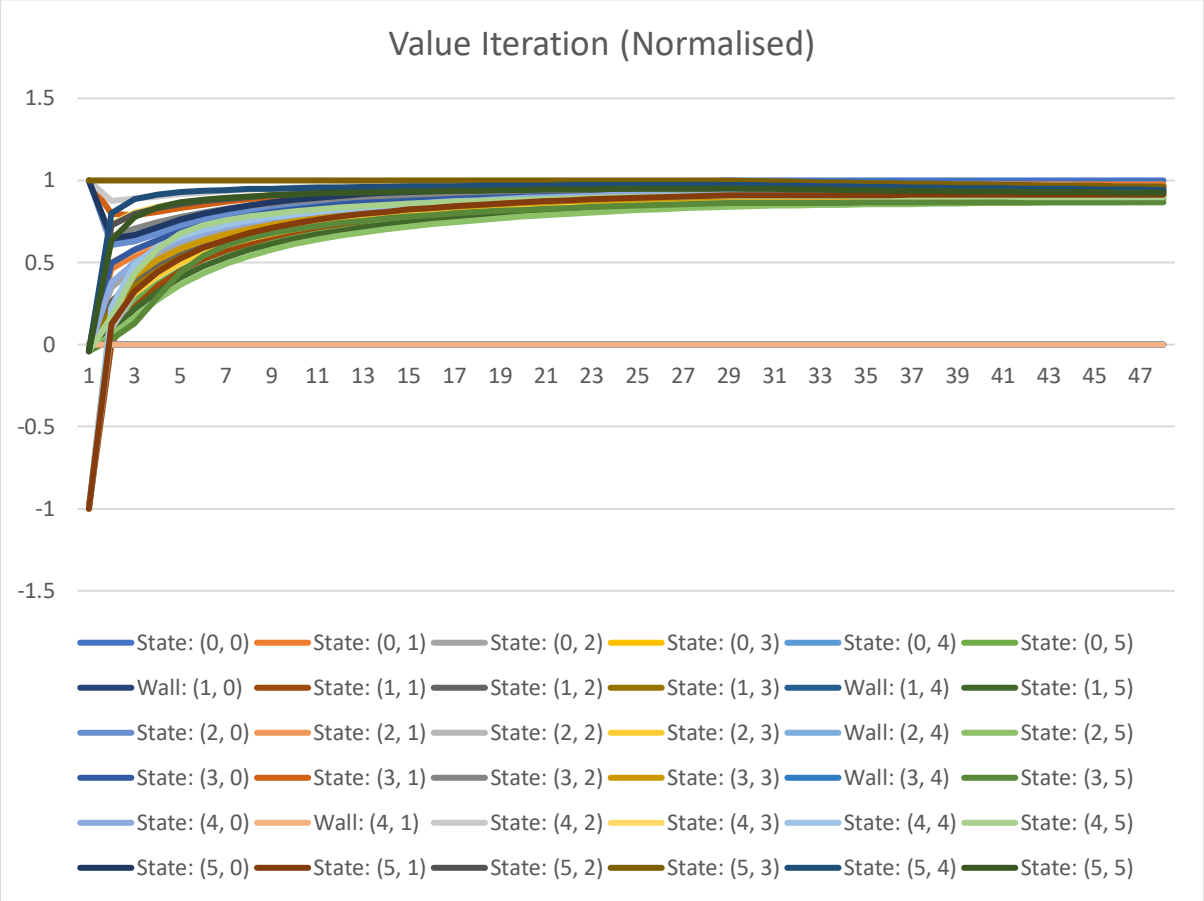

Figure 5: Plot of Utility Estimate across iterations for Value Iteration

Policy Iteration (Part 1)

Description of Implemented Solutions

In Policy Iteration, both Policy Evaluation and Policy Improvement are repeated until the improvement step yields no change in utilities.

A Policy Evaluation step calculates the utility of each state for a given policy while a Policy Improvement step calculates a new policy using a one-step look-ahead for calculating the maximum expected utility of the subsequent state.

The Policy Evaluation step can be implemented with a simplified Bellman update for large state spaces. It is repeated $k = 4$ times to produce the next utility estimate.

The Bellman update is given by:

$$U_{i+1(s)} \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

The implementation can be briefly described through the following steps:

1. Initialise the utility of all states to 0 and their actions to *UP*
2. Import the maze environment
3. Initialise a *didChange* Boolean to keep track of any changes to the policy
4. Repeat
    a. Set *didChange* to false
    b. For each state: Perform a Policy Evaluation for k times
    c. For each state: Perform a Policy Improvement
        i. If the maximum possible utility is greater than the current utility based on the existing policy:
            1. Set the new action that maximises the expected utility of the subsequent state
            2. Set *didChange* to true
5. Until *didChange* is false

The following Figure 6 shows the code snippet of a Policy Iteration run. It starts off by initialising a *didChange* Boolean variable so that the do-while loop can be terminated later. It then performs a *policyEvaluation* and *policyImprovement* for all states. The *policyImprovement* function will return true whenever any policy is updated. These 2 steps are then repeated until the improvement step yields no changes.

```
private static void runPolicyIteration(Maze maze) {
    boolean didChange;
    int iteration = 1;

    do {
        didChange = false;

        /* 1. Policy Evaluation */
        policyEvaluation(maze, K);

        /* 2. Policy Improvement */
        for (int c = 0; c < Constants.NUM_COL; c++) {
            for (int r = 0; r < Constants.NUM_ROW; r++) {
                Cell currCell = maze.getCell(new Coordinate(c, r));

                boolean changed = policyImprovement(currCell, maze);

                if (changed)
                    didChange = true;
            }
        }

        iteration++;
    } while (didChange);
}
```

Figure 6: Code Snippet for Policy Iteration

The following Figure 7 shows the code snippet shows the *policyEvaluation* function. In each iteration, the *policyEvaluation* function performs a Bellman update repeatedly for K times. This will calculate and update the utilities of all states based on the current policy.

```
private static void policyEvaluation(Maze maze, int k) {
    for (int i = 0; i < k; i++) {
        for (int c = 0; c < Constants.NUM_COL; c++) {
            for (int r = 0; r < Constants.NUM_ROW; r++) {
                /* 1. Get current reward & policy */
                Cell currCell = maze.getCell(new Coordinate(c, r));

                /*
                 * 2. Sum up the 3 neighbours (i.e. UP, LEFT, RIGHT) based on the current policy
                 */
                Cell[] neighbours = maze.getNeighboursOfCell(currCell);
                double up = Constants.PROBABILITY_UP * neighbours[0].getUtility();
                double left = Constants.PROBABILITY_LEFT * neighbours[1].getUtility();
                double right = Constants.PROBABILITY_RIGHT * neighbours[2].getUtility();

                /* 3. Update Utility */
                float reward = currCell.getCellType().getReward();
                currCell.setUtility(reward + Constants.DISCOUNT_FACTOR * (up + left + right));
            }
        }
    }
}
```

Figure 7: Code Snippet of policyEvaluation function

The following Figure 8 shows the code snippet of the *policyImprovement* function. It first iterates through all the 4 possible directions and stores them in the *subUtilities* array. The item with the maximum value is then identified using the *maxU* pointer so that the new action can be updated into each state (i.e. 0 = *UP*, 1 = *DOWN*, 2 = *LEFT*, 3 = *RIGHT*).

Subsequently, it compares the newly calculated possible maximum expected utility against the current utility. Only if the new utility is larger than the current utility, will the policy be updated. A Boolean value of true will then be returned if a policy update is performed. Else, a false will be returned.

```
private static boolean policyImprovement(Cell currCell, Maze maze) {
    /* 1. Find the maximum possible sub-utility */
    double[] maxSubUtility = new double[Coordinate.TOTAL_DIRECTIONS];
    for (int dir = 0; dir < Coordinate.TOTAL_DIRECTIONS; dir++) {
        Cell[] neighbours = maze.getNeighboursOfCell(currCell, dir);
        double up = Constants.PROBABILITY_UP * neighbours[0].getUtility();
        double left = Constants.PROBABILITY_LEFT * neighbours[1].getUtility();
        double right = Constants.PROBABILITY_RIGHT * neighbours[2].getUtility();

        maxSubUtility[dir] = up + left + right;
    }

    int maxSU = 0;
    for (int m = 1; m < maxSubUtility.length; m++) {
        if (maxSubUtility[m] > maxSubUtility[maxSU])
            maxSU = m;
    }

    /* 2. Current sub-utility */
    Cell[] neighbours = maze.getNeighboursOfCell(currCell);
    double up = Constants.PROBABILITY_UP * neighbours[0].getUtility();
    double left = Constants.PROBABILITY_LEFT * neighbours[1].getUtility();
    double right = Constants.PROBABILITY_RIGHT * neighbours[2].getUtility();

    double currSubUtility = up + left + right;

    /* 3. Update policy? */
    if (maxSubUtility[maxSU] > currSubUtility) {
        currCell.setPolicy(maxSU);
        return true;
    } else {
        return false;
    }
}
```

Figure 8: Code Snippet of policyImprovement function

Plot of Optimal Policy

The following Figure 3 shows the final plot of the optimal policies of all states after convergence on the **11th iteration** by using a *K* value of 4.

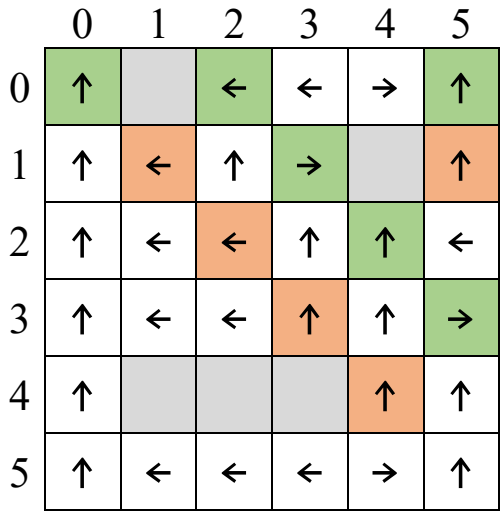

Figure 9: Plot of Optimal Policy for Policy Iteration with K = 4

Utility of all States

The following Figure 10 shows the final utilities of all states after convergence on the **11th iteration** by using a *K* value of 4.

	0	1	2	3	4	5
0	33.772		31.386	30.851	30.375	31.599
1	32.921	31.155	30.945	31.386		29.803
2	32.220	31.577	30.031	30.819	31.249	30.676
3	31.553	31.091	30.515	29.398	30.636	31.113
4	30.952				29.021	30.423
5	30.288	29.706	29.133	28.568	28.693	29.780

Figure 10: Utility of all states for Policy Iteration with K = 4

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

The following Figure 11 shows the plot of utility estimates as a function of the number of iterations. The values in this graph have been normalised to better depict how utility values are changing across iterations.

As compared to the Value Iteration algorithm which converged at the 47th iteration, this Policy Iteration run was able to achieve convergence earlier at the **11th iteration**.

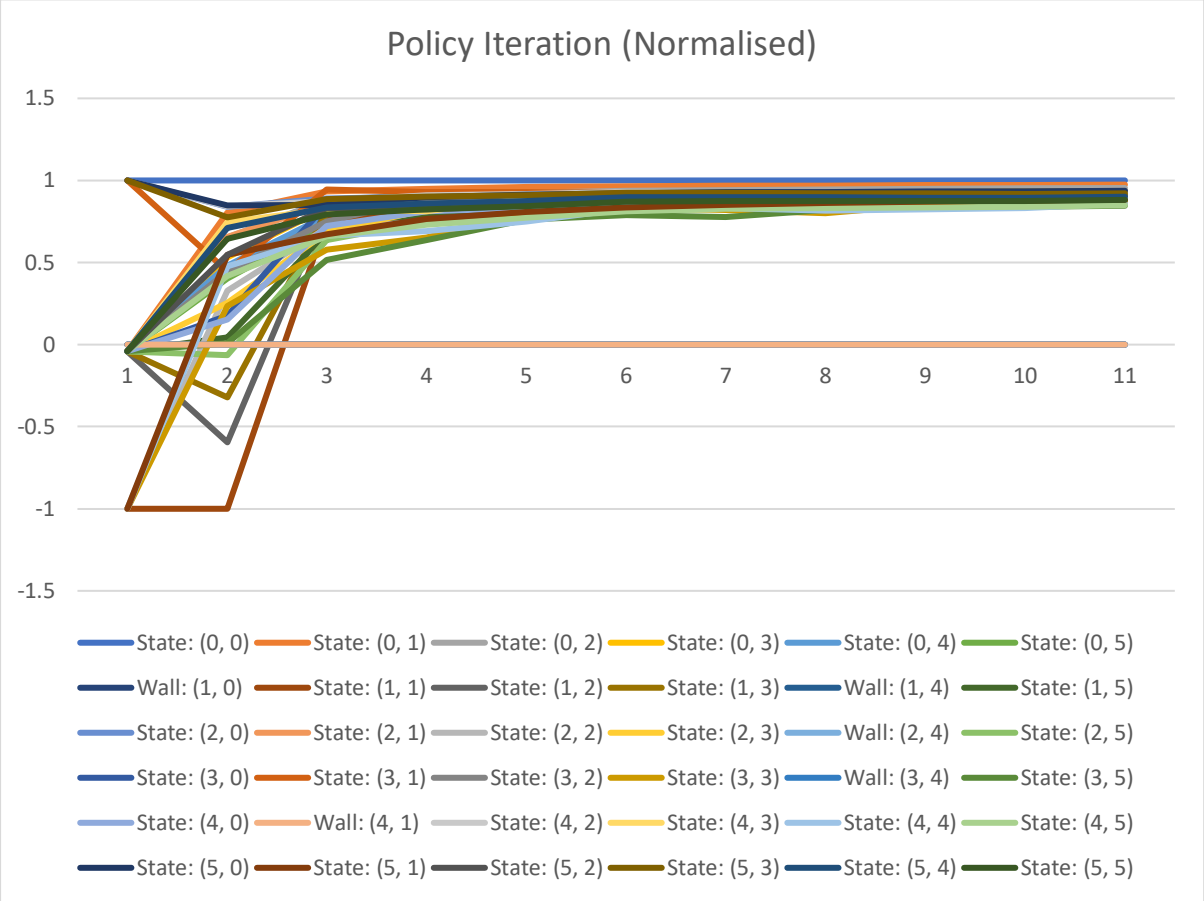

Figure 11: Plot of Utility Estimate across iterations for Policy Iteration

Bonus Question (Part 2)

Complicated Maze Environment

In this section, two complicated mazes – Figure 12 and Figure 13 have been designed to explore the difference in the **number of iterations**, **optimal policies** and **plot of utility estimates** by using Value Iteration and Policy Iteration algorithms.

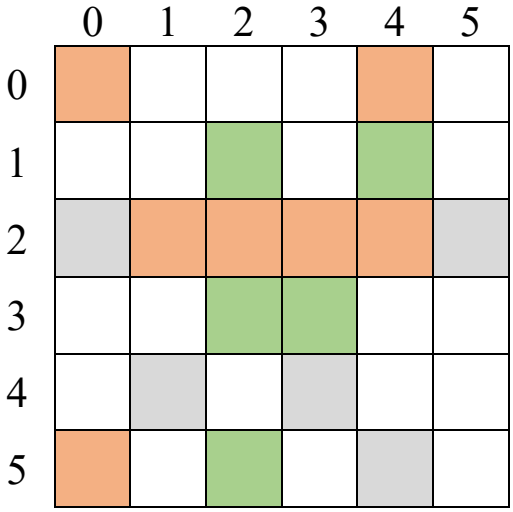

Figure 12: Complicated Maze Design 1

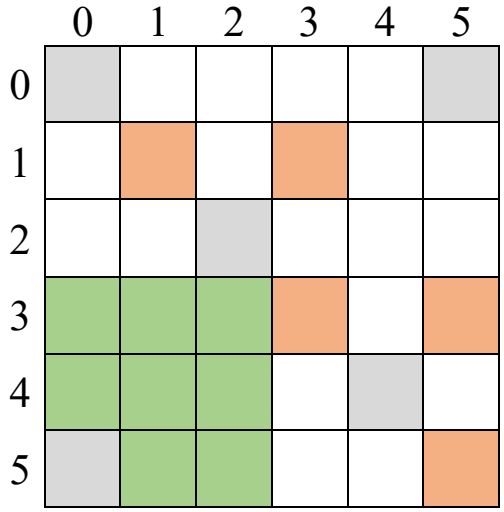

Figure 13: Complicated Maze Design 2

Without modifying the epsilon values, Figure 12: Complicated Maze Design 1 was able to converge at the **45th iteration** and the **4th iteration** when using the Value Iteration and Policy Iteration algorithm respectively. Convergence occurs slightly earlier as compared to the original maze design. As such, it can be deduced that simply increasing the number of brown states will not necessary achieve a more complicated maze design.

On the other hand, Figure 13: Complicated Maze Design 2 only converged much later at the **60th iteration** and the **37th iteration** when using the Value Iteration and Policy Iteration algorithm respectively. This may indicate a more complicated maze design. In this design, the green states are being surrounded by either wall or brown states which may require more iterations to find the most optimal policy.

Using Value Iteration

In Value Iteration, the plot of optimal policies obtained at the **45th iteration** for Complicated Maze Design 1 is shown in Figure 14 and the **60th iteration** for Complicated Maze Design 2 is shown in Figure 15.

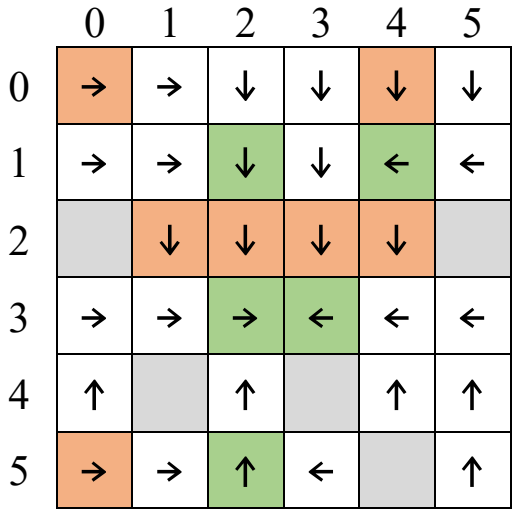

Figure 14: Optimal Policy of Complicated Maze Design 1 using Value Iteration

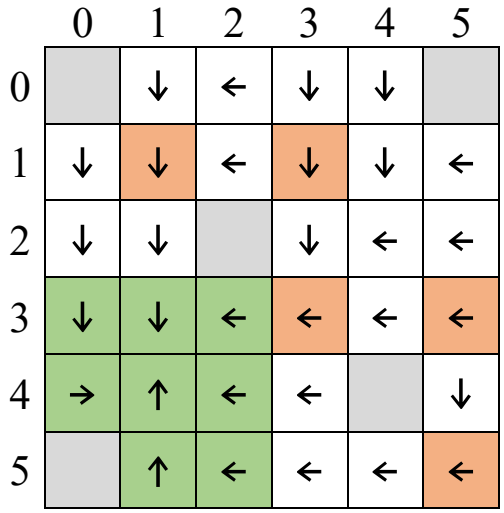

Figure 15: Optimal Policy of Complicated Maze Design 2 using Value Iteration

The following Figure 16 and Figure 17 shows the plot of utility estimates of Complicated Maze 1 and 2 respectively using the Value Iteration algorithm. The values in this graph have been normalised to better depict how utility values are changing across iterations.

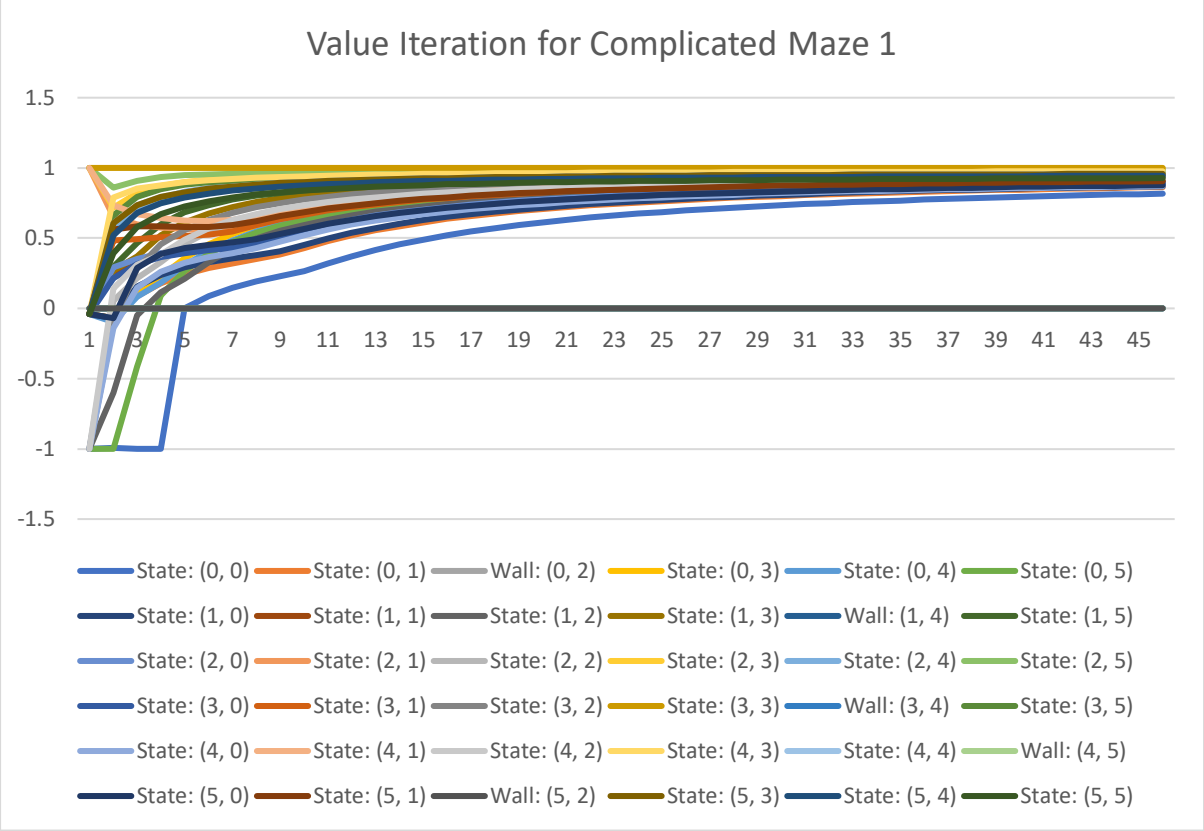

Figure 16: Utility Estimate across iterations for Value Iteration of Complicated Maze 1

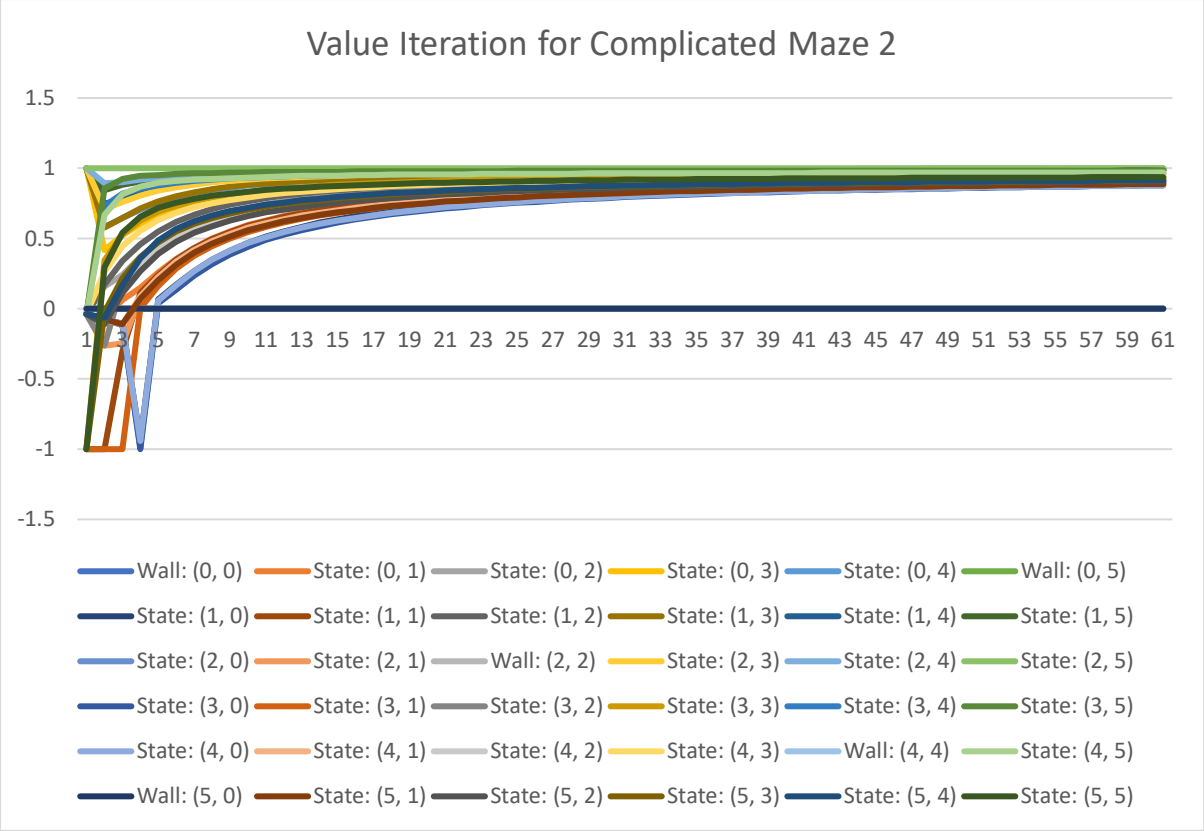

Figure 17: Utility Estimate across iterations for Value Iteration of Complicated Maze 2

Using Policy Iteration

In Policy Iteration, the plot of optimal policies obtained at the **4th iteration** for Complicated Maze Design 1 is shown in Figure 16 and the **37th iteration** for Complicated Maze Design 2 is shown in Figure 17.

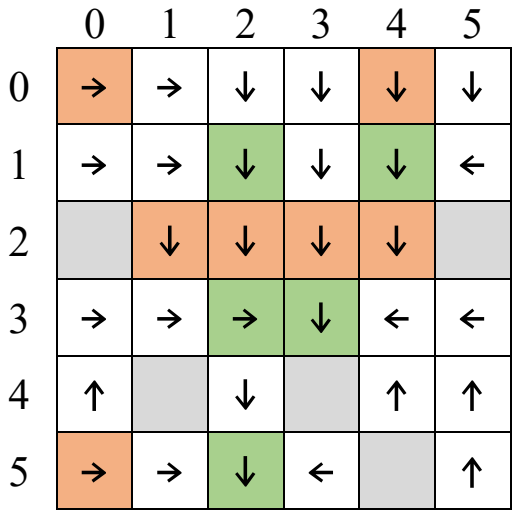

Figure 18: Optimal Policy of Complicated Maze Design 1 using Policy Iteration

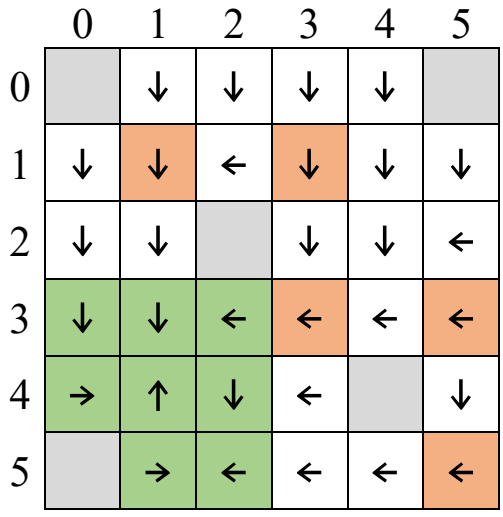

Figure 19: Optimal Policy of Complicated Maze Design 2 using Policy Iteration

The following Figure 20 and Figure 21 shows the plot of utility estimates of Complicated Maze 1 and 2 respectively using the Policy Iteration algorithm. The values in this graph have been normalised to better depict how utility values are changing across iterations.

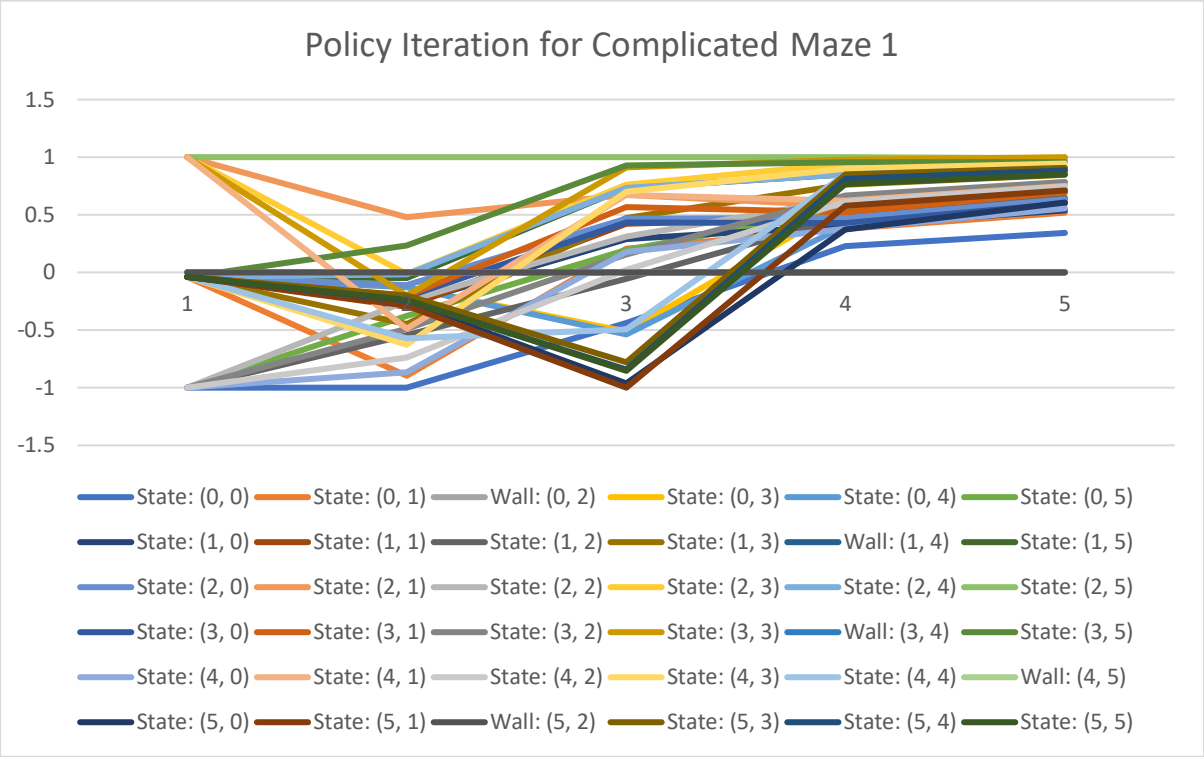

Figure 20: Utility Estimate across iterations for Policy Iteration of Complicated Maze 1

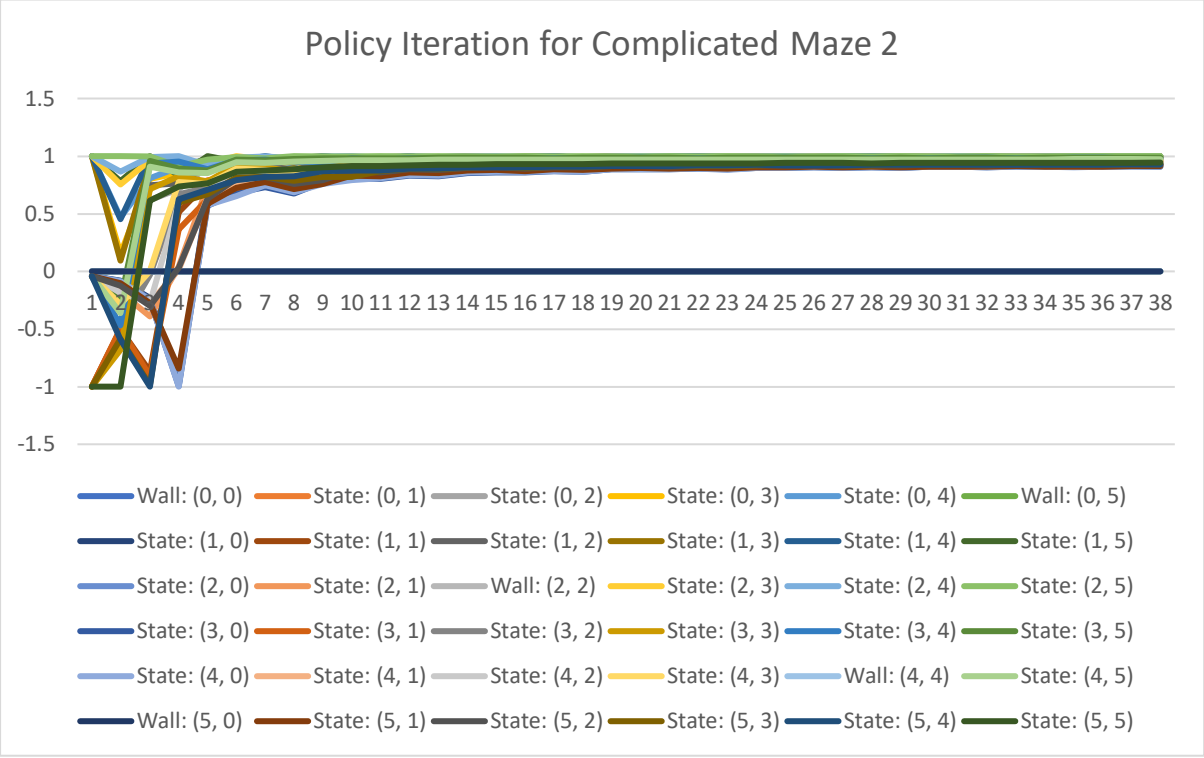

Figure 21: Utility Estimate across iterations for Policy Iteration of Complicated Maze 2

Larger Maze Environment

A more complicated maze environment can be created by increasing the size of the maze environment. This is simply performed by modifying *NUM_COL* and *NUM_ROW* in the constants defined in Constants.java as shown in Figure 18.

```
public final static int NUM_COL = 6;
public final static int NUM_ROW = 6;
```

Figure 22: Code Snippet of Constants.java

The following Figure 19 shows a sample txt file to customise a maze environment. Every new Maze initialisation can be made to import this txt file to specify white, brown, green and wall states.

```
G X G W W G
W B W G X B
W W B W G W
W W W B W G
W X X X B W
W W W W W W
```

Figure 23: Txt file to customise maze environment

Upon increasing the size of the maze environment above 500 columns by 500 rows, it takes a much longer time for both Value Iteration and Policy Iteration algorithm to find the optimal policy.

Source Code (Part 1)

Overview

The following Figure 12 shows an overview of the source code structure. There exist two main programs to run Value Iteration and Policy Iteration respectively. They are supported by multiple entity classes which include Cell, Maze and Constants. A Cell extends Coordinate and composes CellType and Policy.

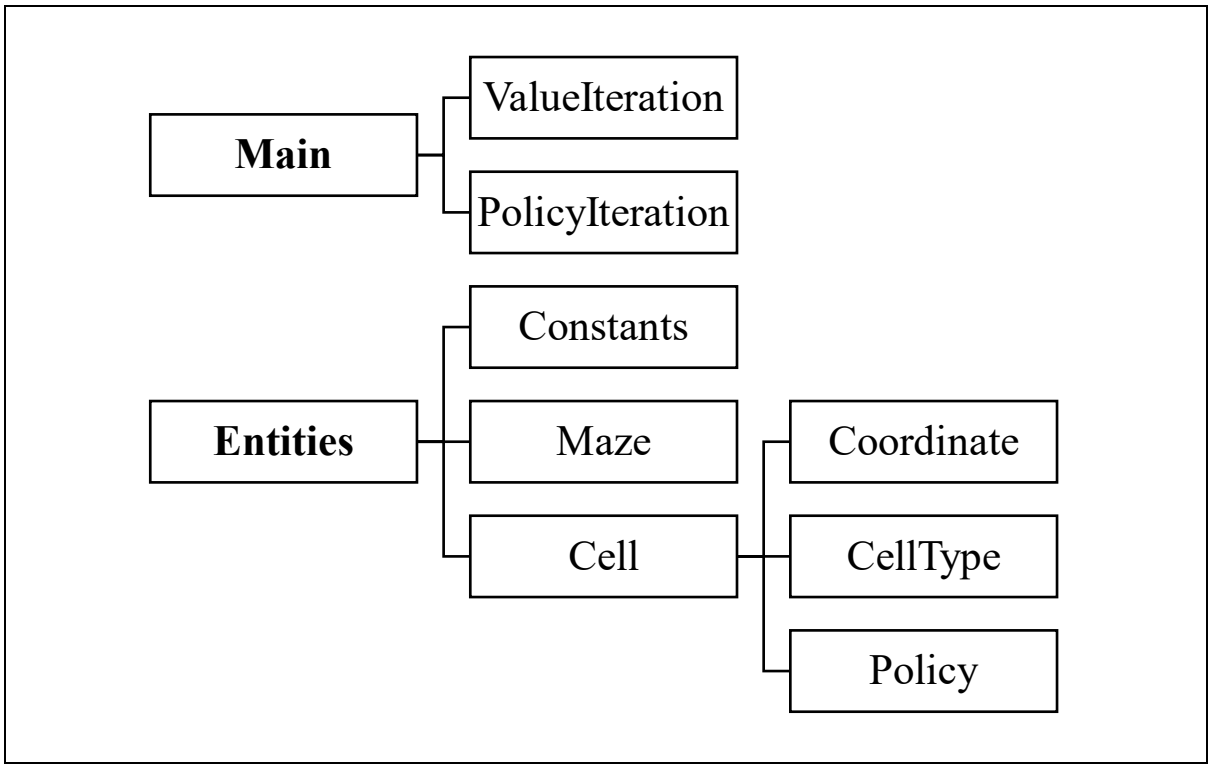

Figure 24: Structure of Source Code

Source Code

Please refer to the attached ZIP file containing the full set of source code.