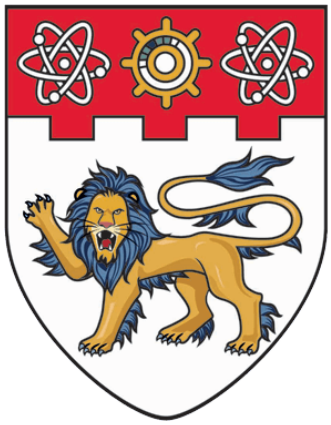

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4046 Intelligent Agents
Assignment 1

Name	Kong Jie Wei
Matriculation Number	U1720017C

Table of Contents

1. Descriptions of Source code and File Structure

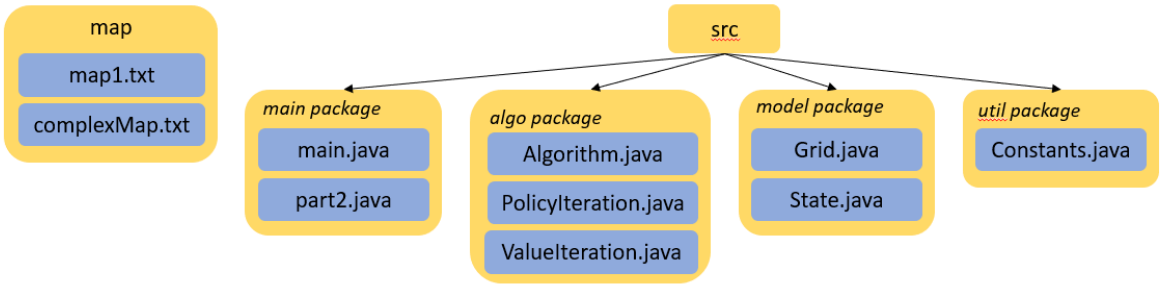

Figure 1.1 Source code and file structure

To obtain the optimal policy in a given maze environment, 2 algorithms: Value Iteration and Policy Iteration, are implemented. My implemented source code and its file structure is shown in Figure 1.1 above.

Map folder consist of text files which represents a certain maze environment.

- "E" – white, empty spaces
- "W" – wall
- "S" – start location
- "P" – green, positive reward spaces
- "M" – brown, negative reward spaces

The source code folder consists of 4 packages – main, algo, model, util.

1. main:
   a. main.java – main entry point of the program to run the policy or value iteration algorithm.
   b. part2.java – main entry point of the program which experiments on the effect of maze environment complexity on value and policy iteration algorithms.
2. algo:
   a. Algorithm.java – abstract class which contains common methods that will be used in both algorithms, such as displaying the policy, displaying the utility estimates etc.
   b. ValueIteration.java – implementation of value iteration algorithm, subclass of Algorithm.
   c. PolicyIteration.java – implementation of policy iteration algorithm, subclass of Algorithm.
3. model:
   a. Grid.java – represents the maze environment, each cell in the grid is a state object.
   b. State.java – represents a state of a single cell in the grid, and it could be a reward value or a wall.
4. util:
   a. Constants.java – provides the constants used in the program such as discount factor, probability of intended direction, reward values, actions, etc.

2. Value Iteration

2.1 Descriptions of Implemented Solutions

My implementation of ValueIteration.java is as follows:

1. Initialisation of algorithm variables and maze environment. Each cell will be initialised with its respective state object.

```java
 * Constructor.
 * @param path path to read maze environment from
 */
public Algorithm(String path) {
    // initialisation
    grid = new Grid(path);
    utilities = new double[grid.MAX_ROW][grid.MAX_COL]; // initialise utilities to 0
    policy = new Constants.Actions[grid.MAX_ROW][grid.MAX_COL];
    history = new ArrayList<double[][]>(Constants.I);
}
```

Figure 2.1 1 Algorithm Initialisation

```java
 * Constructor.
 * @param path path to read grid map from
 * @exception FileNotFoundException if no file found on the given path
 */
public Grid(String path) {
    File file = new File(path);
    Scanner sc = null;
    try {
        sc = new Scanner(file);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    // read the text file
    while (sc.hasNextLine()) {
        String[] row = sc.nextLine().split("");
        State[] states = new State[row.length];
        for (int i=0; i<row.length; i++) {
            states[i] = new State(row[i]);
        }
        grid.add(states);
    }
    MAX_ROW = grid.size();
    MAX_COL = grid.get(0).length;
    System.out.println("Grid of " + MAX_ROW + "x" + MAX_COL + " loaded");
}
```

Figure 2.1 2 Grid Initialisation

```java
public State(String s) {
    switch (s) {
        case "P": setWall(false); setReward(Constants.GREEN_REWARD); break;
        case "M": setWall(false); setReward(Constants.BROWN_REWARD); break;
        case "W": setWall(true); break;
        default: setWall(false); setReward(Constants.WHITE_REWARD); break;
    }
}
```

Figure 2.1 3 State Initialisation

```java
public class Constants {
    // Reward values
    public static double GREEN_REWARD = +1.00;
    public static double BROWN_REWARD = -1.00;
    public static double WHITE_REWARD = -0.04;
```

Figure 2.1.4 Reward Values

Figure 2.1.2 shows that when a Grid object is instantiated, it reads the text file specified by the input parameter `path` and populates the grid with its respective states. Figure 2.1.3 and Figure 2.1.4 shows that the respective states are filled with the following attributes:

- boolean value of whether it is a wall or not.
- Reward values from Constants.java, +1.0 for green cells, -1.0 for brown cells and -0.04 for white cells.

2. `runValueIteration` method is executed to obtain the optimal policy.

```
 * value iteration algorithm
 */
private void runValueIteration() {
    int iterations = 1;
    double maxDelta;
    do {
        maxDelta = 0;
        maxDelta = getBestUtility(maxDelta);
        System.out.println("Iteration " +iterations+ " - Max Delta: "+maxDelta);
        double[][] currUtilities = new double[grid.MAX_ROW][grid.MAX_COL];
        copy2DArray(utilities, currUtilities);
        history.add(currUtilities); // store utility estimates for this iteration
        iterations++;
    } while (maxDelta >= Constants.CONVERGENCE_THRESH);
}
```

Figure 2.1.5 Do-While loop for Value Iteration Algorithm

```
// the maximum reward
public static final double R_MAX = Collections.max(
        new ArrayList<Double>(Arrays.asList(GREEN_REWARD, BROWN_REWARD, WHITE_RE-
WARD)));

// Constant parameter C to adjust
public static final double C = 0.1;

// formula for epsilon
public static final double EPSILON = C*R_MAX;

// convergence threshold formula
public static final double CONVERGENCE_THRESH = EPSILON*(1-DISCOUNT)/DISCOUNT;
```

Figure 2.1.6 Constants used in Value Iteration Algorithm

Figure 2.1.5 shows my implementation of the Value Iteration algorithm. The `maxDelta` variable stores the maximum change of utility values of all states in the grid, i.e. $maxDelta = \max(|U'(s_0) - U(s_0)|, \dots, |U'(s_n) - U(s_n)|)$, where s_0 is the state of the first cell and s_n is the state of the last cell. This value is returned by the method `getBestUtility`, shown in Figure 2.1.7 below. At every iteration, the utility values are being added into an ArrayList (`history`), to facilitate the plotting of utility estimates vs number of iterations graph. The value iteration algorithm terminates when the `maxDelta` value is less than the `CONVERGENCE_THRESH` constant in Constants.java, shown in Figure 2.1.6 above.

```
 * Function that calculates the utility values of actions and selects the optimal
one
 * @param delta a small value of type double, 0 or minimum value possible
 * @return maximum change in utility
 */
private double getBestUtility(double delta) {
    for (int row = 0; row<grid.MAX_ROW; row++)
        for (int col = 0; col < grid.MAX_COL; col++) {
            State state = grid.getGrid().get(row)[col];
            if (state.isWall()) continue; // skip wall
            HashMap<Constants.Actions, Double> actionUtilities = new HashMap<>(4);
            for (Constants.Actions intendedAction: Constants.Actions.values()) {
                Constants.Actions left, right;
                left = right = null;
                // set right and left of intended action
                switch (intendedAction) {
                    case U: left = Constants.Actions.L; right = Constants.Ac-
tions.R; break;
                    case L: left = Constants.Actions.D; right = Constants.Ac-
tions.U; break;
                    case R: left = Constants.Actions.U; right = Constants.Ac-
tions.D; break;
                    case D: left = Constants.Actions.R; right = Constants.Ac-
tions.L; break;
                }
                double intendUtility = calculateUtility(row, col, intendedAction);
                double leftUtility = calculateUtility(row, col, left);
                double rightUtility = calculateUtility(row, col, right);
                double utility = Constants.INTENDED_PROB*intendUtility + Con-
stants.RIGHT_ANGLE_PROB*leftUtility + Constants.RIGHT_ANGLE_PROB*rightUtility;
                utility = state.getReward() + Constants.DISCOUNT*utility;
                actionUtilities.put(intendedAction, utility);
            }
            Constants.Actions bestAction = null;
            double bestUtility = Collections.max(actionUtilities.values());
            for (Map.Entry<Constants.Actions, Double> map : actionUtilities.en-
trySet())
                if(map.getValue() == bestUtility) bestAction = map.getKey();
            policy[row][col] = bestAction;
            double newDelta = Math.abs(bestUtility - utilities[row][col]);
            delta = Math.max(newDelta, delta);
            utilities[row][col] = bestUtility;
        } return delta;
```

Figure 2.1.7 getBestUtility Method

```
 * @param row current row number of the grid
 * @param col current column number of the grid
 * @param action action to move
 * @return utility value of the action
 */
protected double calculateUtility(int row, int col, Constants.Actions action) {
    double utility = 0;
    // get the utility of the next state based on the action
    // if a wall is present, the utility is the current state (no change)
    switch (action) {
        case U: utility = (row-1 >= 0 && !grid.getGrid().get(row-
1)[col].isWall()) ? utilities[row-1][col] : utilities[row][col]; break;
        case L: utility = (col-1 >= 0 && !grid.getGrid().get(row)[col-
1].isWall()) ? utilities[row][col-1] : utilities[row][col]; break;
        case R: utility = (col+1 < grid.MAX_COL && !grid.get-
Grid().get(row)[col+1].isWall()) ? utilities[row][col+1] : utilities[row][col];
break;
        case D: utility = (row+1 < grid.MAX_ROW && !grid.get-
Grid().get(row+1)[col].isWall()) ? utilities[row+1][col] : utilities[row][col];
break;
    }
    return utility;
```

Figure 2.1.8 calculateUtility Method

The `getBestUtility` function in Figure 2.1.7 does the following:

a. Implementing the equation, $U_{i+1}(s) \leftarrow \mathrm{R(s)} + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$ for all states where γ is the discount factor and A(s) is the set of actions: up, down left, right.
b. To compute the utility of each action in a state, the `calculateUtility` function is executed as shown in Figure 2.1.8. It returns the utility when a specific action is taken from a state, and if an action would result in walking into a wall, the utility of the current position is returned.
c. The state which yields the biggest change in utility value will be stored and then returned.

3. The final utility and policy are displayed on the console.
4. The utility values of all states for each iteration were recorded and is written into a `.csv` file to facilitate the plotting of utility estimates as a function of the number of iterations.

2.2 Plot of Optimal Policy

The experiment for the Value Iteration algorithm is conducted with the following parameters:

- Discount factor, γ = 0.99
- C = 0.1
- Max reward R_MAX = +1.0
- Epsilon = C*R_MAX
- Convergence threshold = epsilon * $\frac{(1-\gamma)}{\gamma}$

```
Iteration 688 - Max Delta: 0.0010031795918337139
Final Policy:
|   U   | Wall |   L   |   L   |   L   |   U   |
|   U   |   L  |   L   |   L   | Wall  |   U   |
|   U   |   L  |   L   |   U   |   L   |   L   |
|   U   |   L  |   L   |   U   |   U   |   U   |
|   U   | Wall | Wall  | Wall  |   U   |   U   |
|   U   |   L  |   L   |   L   |   U   |   U   |
Discount factor: 0.99
Max Reward(R_MAX): 1.0
Constant C: 0.1
Epsilon: 0.1
Convergence Threshold: 0.010101010101011011
```

Figure 2.2 Plot of Optimal Policy – Value Iteration

Figure 2.2 was obtained using the constants as specified above. It displays number of iterations for value iteration to converge with its corresponding `maxDelta`, and the plot of optimal policy. For the plot of optimal policy, "U" refers to "Up" action, "L" refers to "Left" action, "R" refers to "Right" action, "D" refers to "Down" action and "Wall" refers to a wall being present at that particular cell in the grid.

Note that different values of constants C and discount factor will result in different values for the convergence threshold. This results in a different number of iterations required for the algorithm to converge, causing a different utility estimate and policy compared to the one displayed in Figure 2.2. The effects of changing constant C and discount factor on number of iterations to converge is illustrated in the table below.

C	Discount factor	Convergence Threshold	Number of Iterations to Converge
0.1	0.985	0.00152284263959391	431
0.1	0.980	0.002040816326530614	308
1	0.990	0.01010101010101011	459
1	0.985	0.0152284263959391	278
1	0.980	0.02040816326530614	194

My observation is that either increasing the constant C or decreasing the discount factor causes the convergence threshold value to increase, hence leading to a lesser number of iterations required for value iteration to converge.

2.3 Utilities of all States

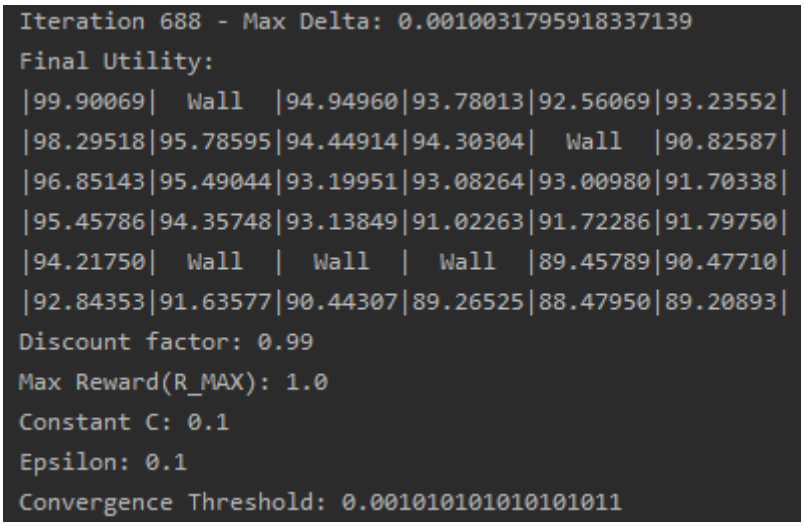

Figure 2.3 Plot of utility estimates of all states – Value Iteration

Utility Estimates (row, column) - Discount factor = 0.99, Constant C = 0.1, Rmax = 1.0					
(0,0): 99.901	(1,0): 98.295	(2,0): 96.851	(3,0): 95.458	(4,0): 94.218	(5,0): 92.844
(0,1): Wall	(1,1): 95.786	(2,1): 95.490	(3,1): 94.357	(4,1): Wall	(5,1): 91.636
(0,2): 94.950	(1,2): 94.449	(2,2): 93.200	(3,2): 93.138	(4,2): Wall	(5,2): 90.443
(0,3): 93.780	(1,3): 94.303	(2,3): 93.083	(3,3): 91.023	(4,3): Wall	(5,3): 89.265
(0,4): 92.561	(1,4): Wall	(2,4): 93.010	(3,4): 91.723	(4,4): 89.458	(5,4): 88.480
(0,5): 93.236	(1,5): 90.826	(2,5): 91.703	(3,5): 91.798	(4,5): 90.477	(5,5): 89.209

Figure 2.3 and the table above displays the utilities estimates of every non-wall state. It was obtained using the constants as specified in 2.2 Plot of Optimal Policy.

2.4 Plot of Utility Estimates as a Function of the Number of Iterations

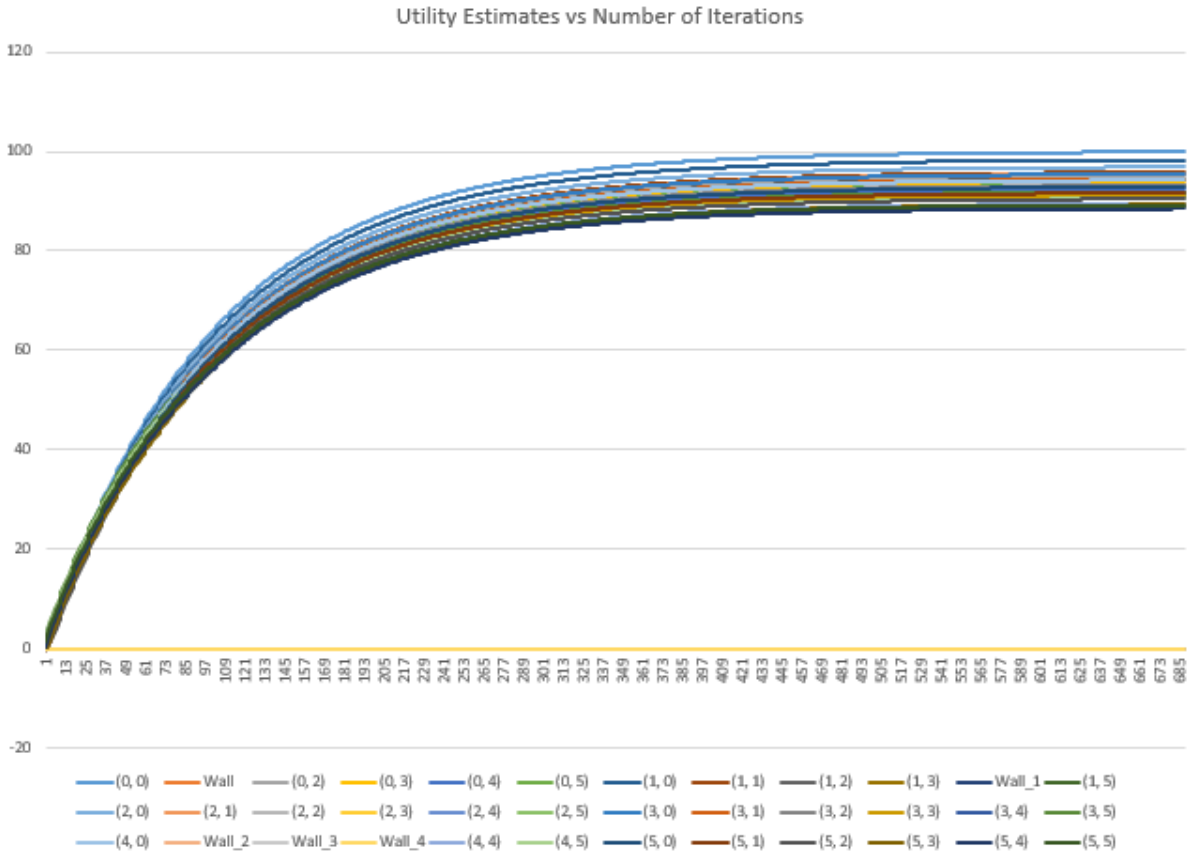

Figure 2.4 Graph of Utility Estimates vs Number of Iterations – Value Iteration

Figure 2.4 was obtained using the constants as specified above in 2.2 Plot of Optimal Policy.

3. Policy Iteration

3.1 Descriptions of Implemented Solutions

My implementation of PolicyIteration.java is as follows:

1. Initialisation of algorithm variables and maze environment. Each cell will be initialised with its respective state object. The implementation of this is the same as for Value Iteration (see Figure 2.1.1 to Figure 2.1.4).
2. `runPolicyIteration` method is executed to obtain the optimal policy.

```
 * policy iteration algorithm
 */
private void runPolicyIteration() {
    initPolicy(); //initialise policy
    boolean change;
    int iterations = 1;
    double[][] initialUtility = new double[grid.MAX_ROW][grid.MAX_COL];
    ValueIteration.copy2DArray(utilities, initialUtility);
    history.add(initialUtility); // initial
    do {
        Constants.Actions[][] oldPolicy = new Constants.Ac-
tions[grid.MAX_ROW][grid.MAX_COL];
        copy2DArray(policy, oldPolicy);
        evaluatePolicy();
        policyImprovement();
        change = comparePolicy(oldPolicy, policy);
        System.out.println("Iteration " +iterations+ " - Change: "+change);
        iterations++;
    } while (change);
}
```

Figure 3.1.1 Do-While loop for Policy Iteration Algorithm

```
 * Function that initialises the grid policies with a random action
 */
private void initPolicy() {
    List<Constants.Actions> actions = Arrays.asList(Constants.Actions.values());
    Random r = new Random();
    for (int row = 0; row < grid.MAX_ROW; row++)
        for (int col = 0; col < grid.MAX_COL; col++) {
            State state = grid.getGrid().get(row)[col];
            if (state.isWall()) continue; // skip wall
            int i = r.nextInt(actions.size()); // to set policy randomly
            policy[row][col] = actions.get(i); // set initial policy
        }
}
```

Figure 3.1.2 Initialising the Policy

```
 * policy evaluation algorithm, implementing simplified Bellman Equation
 */
private void evaluatePolicy() {
    int iterations = 1;
    do {
        for (int row = 0; row<grid.MAX_ROW; row++)
            for (int col = 0; col < grid.MAX_COL; col++) {
                State state = grid.getGrid().get(row)[col];
                if (state.isWall()) continue; //skip wall
                Constants.Actions intendedAction = policy[row][col];
                Constants.Actions left, right;
                left = right = null;
                // set right and left of intended action
                switch (intendedAction) {
                    case U: left = Constants.Actions.L; right = Constants.Ac-
tions.R; break;
                    case L: left = Constants.Actions.D; right = Constants.Ac-
tions.U; break;
                    case R: left = Constants.Actions.U; right = Constants.Ac-
tions.D; break;
                    case D: left = Constants.Actions.R; right = Constants.Ac-
tions.L; break;
                }
                double intendUtility = calculateUtility(row, col, intendedAction);
                double leftUtility = calculateUtility(row, col, left);
                double rightUtility = calculateUtility(row, col, right);
                double utility = Constants.INTENDED_PROB*intendUtility + Con-
stants.RIGHT_ANGLE_PROB*leftUtility + Constants.RIGHT_ANGLE_PROB*rightUtility;
                utilities[row][col] = state.getReward() + Constants.DISCOUNT*util-
ity;
            }
        double[][] currUtilities = new double[grid.MAX_ROW][grid.MAX_COL];
        ValueIteration.copy2DArray(utilities, currUtilities);
        history.add(currUtilities); // store utility estimates of this iteration
    } while (++iterations <= Constants.I);
}
```

Figure 3.1.3 Policy Evaluation – simplified Bellman Equation

```
// Constant I - number of times simplified Bellman update
// is executed to produce the next utility estimate
public static final int I = 50;
```

Figure 3.1.4 Constant used in Policy Iteration Algorithm

```
 * Function that compares the 2 policies. Returns false when there is no change
 */
private boolean comparePolicy(Constants.Actions[][] oldPolicy, Constants.Ac-
tions[][] newPolicy) {
    for (int row=0; row<oldPolicy.length; row++) for (int col=0;
col<oldPolicy[row].length; col++) {
        if (oldPolicy[row][col] == null) continue;
        else if (oldPolicy[row][col] != newPolicy[row][col]) {
            return true; // return true when there is a difference
        }
    } return false; // return false when there is no change
}
```

Figure 3.1.5 comparePolicy Method

```
 * policy improvement algorithm, to calculate new policy based on the updated
utilities
 */
private void policyImprovement() {
    for (int row=0; row<grid.MAX_ROW; row++)
        for (int col=0; col<grid.MAX_COL; col++) {
            State state = grid.getGrid().get(row)[col];
            if (state.isWall()) continue; // skip wall
            HashMap<Constants.Actions, Double> expectedUtilities = new
HashMap<>(4);
            for (Constants.Actions action : Constants.Actions.values()) {
                double expectedUtility = 0;
                // get utility value of each action
                switch (action) {
                    case U: expectedUtility = Constants.INTENDED_PROB*calculateU-
tility(row, col, Constants.Actions.U)
                            + Constants.RIGHT_ANGLE_PROB*calculateUtility(row,
col, Constants.Actions.L)
                            + Constants.RIGHT_ANGLE_PROB*calculateUtility(row,
col, Constants.Actions.R); break;
                    case L: expectedUtility = Constants.INTENDED_PROB*calculateU-
tility(row, col, Constants.Actions.L)
                            + Constants.RIGHT_ANGLE_PROB*calculateUtility(row,
col, Constants.Actions.D)
                            + Constants.RIGHT_ANGLE_PROB*calculateUtility(row,
col, Constants.Actions.U); break;
                    case R: expectedUtility = Constants.INTENDED_PROB*calculateU-
tility(row, col, Constants.Actions.R)
                            + Constants.RIGHT_ANGLE_PROB*calculateUtility(row,
col, Constants.Actions.U)
                            + Constants.RIGHT_ANGLE_PROB*calculateUtility(row,
col, Constants.Actions.D); break;
                    case D: expectedUtility = Constants.INTENDED_PROB*calculateU-
tility(row, col, Constants.Actions.D)
                            + Constants.RIGHT_ANGLE_PROB*calculateUtility(row,
col, Constants.Actions.R)
                            + Constants.RIGHT_ANGLE_PROB*calculateUtility(row,
col, Constants.Actions.L); break;
                }
                expectedUtilities.put(action, expectedUtility);
            }
            // find action with the best utility
            double maxExpectedUtility = Collections.max(expectedUtilities.val-
ues());
            Constants.Actions updatedAction = null;
            for (Map.Entry<Constants.Actions, Double> map : expectedUtilities.en-
trySet())
                if(map.getValue() == maxExpectedUtility)
                    updatedAction = map.getKey();
            policy[row][col] = updatedAction; // update policy
        }
}
```

Figure 3.1.6 Policy Improvement Algorithm

Figure 3.1.1 shows my implementation of the Policy Iteration algorithm. The policy is first initialised using the `initPolicy` method, shown in Figure 3.1.2 above. The `initPolicy` method initialises every state that is not a wall with a random action. Once the initialisation step is done, at every iteration, the previous policy is first stored in a variable named `oldPolicy`. Next, the policy will be evaluated using the `evaluatePolicy` method, shown in Figure 3.1.3 above. This method estimates

the utility of every state using the simplified Bellman equation, by iterating it through a constant `I` times, specified in Constants.java shown in Figure 3.1.4. The `calculateUtility` method is same as the one used in ValueIteration.java (see Figure 2.1.8). Once an iteration of policy evaluation is finished, the `policyImprovement` method, shown in Figure 3.1.6, is executed. This method search for the best policy based on the updated utilities after the `evaluatePolicy` method and update the policy such that it maximises the utilities for every state. Lastly, the updated policy will be compared with the `oldPolicy`, using the method `comparePolicy`, shown in Figure 3.1.5. This method returns `true` when there is a difference in between the 2 policies, else it returns `false`. The value returned by `comparePolicy` then updates the boolean variable `change`. The policy iteration algorithm terminates once the variable `change` is `false`.

3. The final utility and policy are displayed on the console.
4. The utility values of all states for each iteration were recorded and is written into a `.csv` file to facilitate the plotting of utility estimates as a function of the number of iterations.

3.2 Plot of Optimal Policy

The experiment for the Policy Iteration algorithm is conducted with the following parameters:

- Discount factor, γ = 0.99
- I = 50

```
Iteration 7 - Change: false
Final Policy:
|   U   | Wall |   L   |   L   |   L   |   U   |
|   U   |   L  |   L   |   L   | Wall  |   U   |
|   U   |   L  |   L   |   U   |   L   |   L   |
|   U   |   L  |   L   |   U   |   U   |   U   |
|   U   | Wall | Wall  | Wall  |   U   |   U   |
|   U   |   L  |   L   |   L   |   U   |   U   |
Constant I: 50
```

Figure 3.2 Plot of Optimal Policy – Policy Iteration

Figure 3.2 was obtained using the constants as specified above. It displays number of iterations for policy iteration to converge and the plot of optimal policy. Refer to 2.2 Plot of Optimal Policy to understand the symbols shown in Figure 3.2.

Note that for policy iteration, the total number of iterations might differ even though the same parameters are used. The table below documents my observations for the number of iterations needed to find the optimal policy.

No. of times Ran	Discount Factor	I	Iterations needed
10	0.99	50	7 to 9
	0.99	25	12 to 14
	0.98	50	5 to 10
	0.98	25	4 to 11

Some observations are:

- Decreasing the number of iterations for the simplified Bellman equation increases the number of iterations required for policy iteration to converge.
- Decreasing the discount factor increases the variance for the number of iterations required for policy iteration to converge.

3.3 Utilities of all States

```
Iteration 12 - Change: false
Final Utility:
|95.09591|  Wall  |90.31209|89.19019|88.01666|88.73691|
|93.54521|91.09000|89.81163|89.72272|  Wall  |86.37224|
|92.15548|90.84666|88.60749|88.55256|88.53123|87.27723|
|90.81467|89.76065|88.58816|86.54144|87.29398|87.41531|
|89.62074|  Wall  |  Wall  |  Wall  |85.07847|86.13903|
|88.29841|87.13610|85.98841|84.85513|84.14497|84.91463|
Constant I: 25
```

Figure 3.3.2 Plot of utility estimates of all states – Policy Iteration (I = 25, γ = 0.99)

```
Iteration 7 - Change: false
Final Utility:
|96.53733|  Wall  |91.70333|90.56716|89.37985|90.08649|
|94.97018|92.49877|91.20287|91.09680|  Wall  |87.70833|
|93.56425|92.23978|89.98508|89.91156|89.87478|88.60506|
|92.20761|91.13968|89.95324|87.88578|88.62263|88.72996|
|90.99975|  Wall  |  Wall  |  Wall  |86.39228|87.44044|
|89.66193|88.48599|87.32479|86.17815|85.44531|86.20290|
Constant I: 50
```

Figure 3.3.1 Plot of utility estimates of all states – Policy Iteration (I = 50, γ = 0.99)

```
Iteration 7 - Change: false
Final Utility:
|99.96011|  Wall  |95.00695|93.83689|92.61689|93.29115|
|98.35392|95.84403|94.50649|94.35969|  Wall  |90.88095|
|96.90951|95.54787|93.25630|93.13866|93.06518|91.75812|
|95.51529|94.41433|93.19477|91.07805|91.77764|91.85170|
|94.27435|  Wall  |  Wall  |  Wall  |89.51205|90.53075|
|92.89974|91.69142|90.49817|89.31979|88.53311|89.26204|
Constant I: 125
```

Figure 3.3.3 Plot of utility estimates of all states – Policy Iteration (I = 125, γ = 0.99)

```
Iteration 7 - Change: false
Final Utility:
|99.99739|  Wall  |95.04294|93.87251|92.65214|93.32606|
|98.39078|95.88046|94.54248|94.39522|  Wall  |90.91550|
|96.94595|95.58390|93.29193|93.17381|93.09993|91.79247|
|95.55132|94.44999|93.23007|91.11282|91.81200|91.88570|
|94.31002|  Wall  |  Wall  |  Wall  |89.54603|90.56441|
|92.93500|91.72633|90.53273|89.35401|88.56674|89.29536|
Constant I: 150
```

Figure 3.3.4 Plot of utility estimates of all states – Policy Iteration (I = 150, γ = 0.99)

Figure 3.3.1 to Figure 3.3.4 above shows the plot of utility estimates of all states for 4 different values of constant `I`, 25, 50, 125 and 150 respectively. It can be observed from these plot of utility estimates that `I` as increase, the utility estimates of each state increase as well. However, for higher values of `I`, for example `I` = 125 vs `I` = 150, the difference in the utility estimates of each state is much less compared to `I` = 25 vs `I` = 50. Also, the utility value will never exceed the utility upper bound, $\frac{R\_MAX}{1-\gamma} = \frac{1}{1-0.99} = 100.$

Utility Estimates (row, column) - Discount factor = 0.99, Constant I = 50					
(0,0): 96.153	(1,0): 94.590	(2,0): 93.189	(3,0): 91.836	(4,0): 90.632	(5,0): 89.298
(0,1): Wall	(1,1): 92.123	(2,1): 91.868	(3,1): 90.772	(4,1): Wall	(5,1): 88.126
(0,2): 91.332	(1,2): 90.832	(2,2): 89.618	(3,2): 89.589	(4,2): Wall	(5,2): 86.969
(0,3): 90.200	(1,3): 90.730	(2,3): 89.549	(3,3): 87.527	(4,3): Wall	(5,3): 85.825
(0,4): 89.016	(1,4): Wall	(2,4): 89.517	(3,4): 88.268	(4,4): 86.042	(5,4): 85.099
(0,5): 89.727	(1,5): 87.352	(2,5): 88.251	(3,5): 88.379	(4,5): 87.093	(5,5): 85.859

3.4 Plot of Utility Estimates as a Function of the Number of Iterations

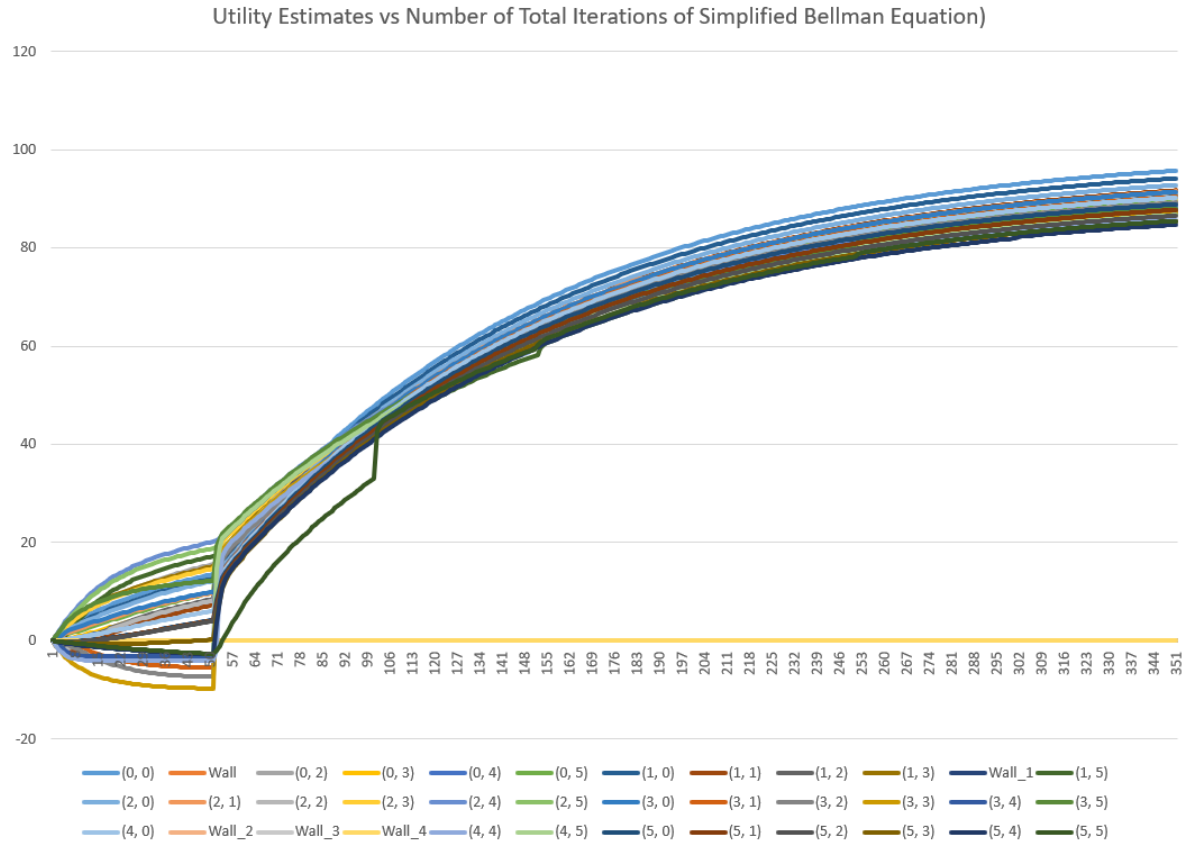

Figure 3.4 Graph of Utility Estimates vs Number of Iterations – Policy Iteration

Figure 3.4 was obtained using the constants as specified above in 3.2 Plot of Optimal Policy.

4. Part 2

4.1 Designing a More Complex Maze Environment

A new Java source code is added in the main package – "part2.java". This java file contains the source code which experiments on the effect of maze environment complexity on the convergence of both value and policy iteration algorithms.

```
 * This function generates a maze environment randomly.
 * @param path txt file where the maze environment will be written to
 * @param size row or column size to generate the environment.
 */
public static void generateMaze(String path, int size) {
    try {
        FileWriter txt = new FileWriter(path);
        Random r = new Random();
        int[] startPos = new int[]{r.nextInt(size), r.nextInt(size)};
        ArrayList<String> values = new ArrayList<String>(4);
        values.add("M"); values.add("P"); values.add("W"); values.add("E");

        for (int row = 0; row < size; row++) {
            for (int col = 0; col < size; col++) {
                int i = r.nextInt(values.size());
                String toWrite = values.get(i);
                if (row == startPos[0] && col == startPos[1])
                    toWrite = "S";
                txt.write(toWrite);
            } txt.write("\n");
        }
        txt.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 4.1 Function to randomly generate Maze Environment

The function `generateMaze` as shown above in Figure 4.1 generates a maze environment randomly. The parameter `size` is used to control the number of rows and columns of the generated maze environment. To increase the generated maze complexity, a larger `size` value can be passed into this function.

4.2 Effect of Maze Environment Complexity on Value Iteration

To experiment the effect of maze environment complexity on the value iteration algorithm, the same parameters as 2.2 Plot of Optimal Policy are used on 100x100 to 500x500 maze environments.

Maze Environment	Number of Iteration to Converge	Execution Time
6x6	688	64ms
100x100	688	1537ms
200x200	688	5263ms
300x300	688	9929ms
400x400	688	17725ms
500x500	688	32662ms

From this experiment, we can observe that at every 100x100 increase in the complexity and size of the maze environment, the value iteration algorithm still manages to converge with the same number of iterations, however the execution time approximately doubles. Therefore, we can conclude that for value iteration, as the complexity and size of the maze environment increases, the number of

iterations required to converge remains the same however the execution time needed for that increases exponentially.

4.3 Effect of Maze Environment Complexity on Policy Iteration

To experiment the effect of maze environment complexity on the policy iteration algorithm, the same parameters as 3.2 Plot of Optimal Policy are used on various maze environment complexity as shown below.

Maze Environment	Number of Iteration to Converge	Execution Time
6x6	7	15ms
10x10	10	31ms
20x20	14-56	41ms-94ms
40x40	16-64	78ms-219ms
50x50	57	297ms
100x100	65	1188ms
200x200	67	4470ms
300x300	69	10047ms
400x400	68	18236ms
500x500	69	28830ms

From this experiment, I observed that the number of iterations required for policy iteration algorithm to converge varies greatly as the maze environment sizes increases from around 20x20 to 40x40. More complex maze environment after 40x40 has a lesser variance for the number of iterations required for policy iteration to converge. Generally, as the maze environment complexity increases, the number of iterations required for policy iteration algorithm to converge increases, and the execution time for it increases as well.