

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

CZ4046: Intelligent Agents

ASSIGNMENT 1

Prepared by:

GUCON NAILAH GINYILLE PABILONIA

U2021643H

Table of Content

PART 1: SOURCE CODE	2
Folder Hierarchy	2
Source Code Execution Guide	3
PART 1: VALUE ITERATION	5
Descriptions of Implemented Solutions	5
Plot of Optimal Policy	9
Utilities of All States	10
Plot of Utility Estimates as a function of the number of Iterations	11
PART 1: POLICY ITERATION	12
Descriptions of Implemented Solutions	12
Plot of Optimal Policy	17
Utilities of All States	18
Plot of Utility Estimates as a function of the number of Iterations	19
PART 2: BONUS QUESTION	20
Data Preparation	20
Approach & Experimentation	22
Results	24
How does the number of states and the complexity of the environment affect convergence?	25
How complex can you make the environment and still be able to learn the right policy?	25

PART 1: SOURCE CODE

Folder Hierarchy

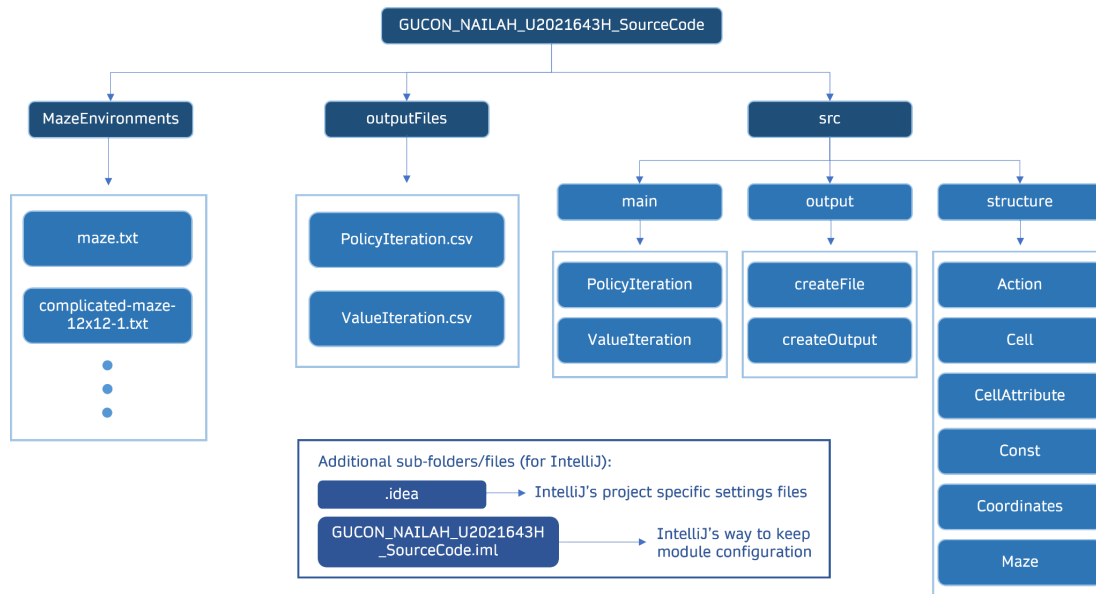


Figure 1: CZ4046 Assignment 1 Submission Folder Hierarchy

Main folder: GUCON_NAILAH_U2021643H_SourceCode

Sub-folders/files: MazeEnvironments, outputFiles, src, .idea, GUCON_NAILAH_U2021643H_SourceCode.iml

MazeEnvironments

Contains preset maze “maze.txt” for Part 1 and 20 other mazes of various sizes (e.g. “complicated-maze-12x12-1.txt”) for Part 2 (Bonus Question).

outputFiles

Contains the latest output files for plotting of PolicyIteration & ValueIteration.

src

main: Contains main programs to run, PolicyIteration & ValueIteration.

output: Contains Java classes to generate the output for PolicyIteration & ValueIteration.

structure: Contains Java classes that support the implementation of PolicyIteration & ValueIteration.

Source Code Execution Guide

To ensure the source code runs smoothly with no issues, please follow the instructions below.

Step 1: Download my Chosen IDE

Download IntelliJ Community Edition:

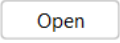
<https://www.jetbrains.com/idea/download/#section=windows>

Do take note of the directory of where your projects will be saved.
E.g. C:\Users\user\IdeaProjects

Step 2: Download my Source Code

- Download the `GUCON_NAILAH_U2021643H.zip` file that was submitted on NTULearn, and unzip it.
- Inside the unzipped folder, take my source code named `GUCON_NAILAH_U2021643H_SourceCode.zip`, put it into your IdeaProjects directory.

Step 3: Set up your Environment

- Unzip `GUCON_NAILAH_U2021643H_SourceCode.zip` file.
- Open IntelliJ:
 - Click on the  button on the top-right hand corner.
 - Find your IdeasProjects directory and select the unzipped `GUCON_NAILAH_U2021643H_SourceCode` folder.
- Once you have successfully loaded the project, you will need to define your Project JDK.

- On the top navigation bar, go to File > Project Structure and make the following changes to the following SDK:

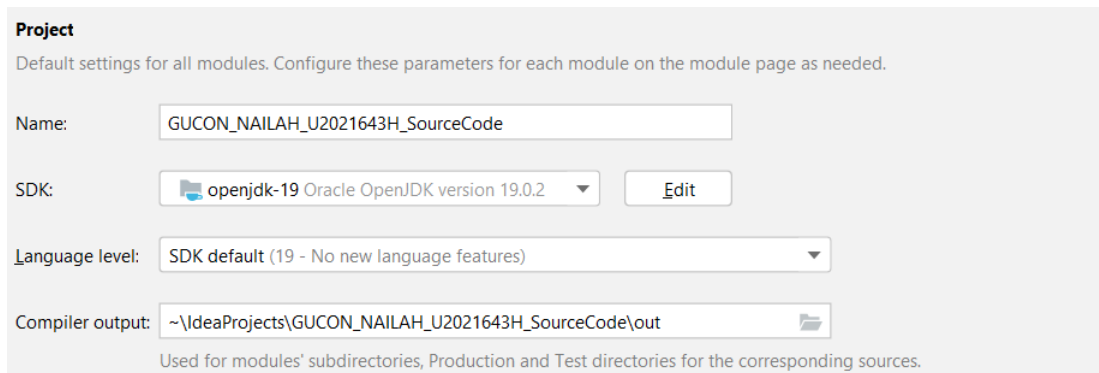



Figure 2: IntelliJ Project Structure Requirements

Step 4: Run ValueIteration OR PolicyIteration program

Once you apply the changes, open either `ValueIteration.java` OR `PolicyInteraction.java` (*depending on which you would like to view the output of*), and run the code on the IntelliJ interface by clicking on the  on the top navigation bar.

PART 1: VALUE ITERATION

Disclaimer: Comments & display codes e.g. `System.out.println()` present in the source code may be omitted to emphasise more important points in the report.

Descriptions of Implemented Solutions

Overview

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

- It represents the maximum expected utility that can be obtained starting from that state and following the optimal action.
- At each iteration, the algorithm updates the value function for each state by taking the maximum over all possible actions and their expected returns. This process is repeated until the value function converges to the optimal value function.
- Once the optimal value function is obtained, the optimal policy can be derived by taking the action with the maximum expected return for each state.

```
public class ValueIteration {
    private final static float C = 0.1;
    private final static float R_MAX = 1;
    private final static float EPSILON = C * R_MAX;

    public static void main(String[] args) {
        Maze maze = new Maze("maze.txt");
        runValueIteration(maze);
    }
}
```

Figure 3: Declaration of necessary variables for Value Iteration

Before Value Iteration can be done on a given maze, the following variables are declared:

- `C` : a constant value (see “How C is calculated” section below for details)
- `R_MAX` : maximum reward in the given environment, which is 1
- `EPSILON` : the maximum error allowable for a given state, before accounting for the discount factor; calculated by multiply `C` with `R_MAX`

The `main` function loads the given maze and calls the `runValueIteration` function.

How C is calculated

C is a constant value, determined after making observations based on the content given in Chapter 17.2 of the reference book, “Artificial Intelligence: A Modern Approach” by S. Russell and P. Norvig. Prentice-Hall, third edition, 2010.

The Figure 4 shows a line graph that depicts the number of value iterations k required to guarantee an error of at most $\varepsilon = C \cdot R_{\max}$, for different values of C , as a function of the discount factor γ .

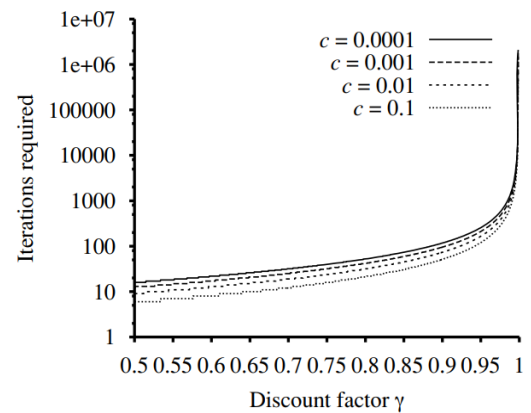


Figure 4: Diagram in reference book that shows different values of C used

It can be seen on the diagram, as C gets smaller, the number of iterations needed for convergence increases. Therefore, $C = 0.1$ will be used in the current implementation as it yields the same optimal policy with the least number of iterations to converge for the same maze.

Code Implementation for Value Iteration

```
private static void runValueIteration(Maze maze) {
    int iteration = 1;
    double maxChangeInUtility = 0;
    // calculates the threshold for stopping the algorithm
    double threshold = EPSILON * ((1 - Const.DISCOUNT_FACTOR) / Const.DISCOUNT_FACTOR);
    createOutput output = new createOutput("ValueIteration", maze);

    ...
}
```

Figure 5: First code snippet of `runValueIteration` function

In the `runValueIteration` function, `iteration` is initialised to 1 and `maxChangeInUtility` is initialised to 0. `threshold` is calculated so that the do-while loop (shown in Figure 5) terminates once the `maxChangeInUtility` crosses the given threshold.

```

private static void runValueIteration(Maze maze) {
    ...
    do {
        maxChangeInUtility = 0;

        for (int c = 0; c < Const.NUM_COL; c++) {
            for (int r = 0; r < Const.NUM_ROW; r++) {
                Cell curCell = maze.getCell(new Coordinates(c, r));

                if (curCell.getCellAttribute() == CellAttribute.WALL)
                    Continue;

                double changeInUtility = calculateUtility(curCell, maze);

                if (changeInUtility > maxChangeInUtility)
                    maxChangeInUtility = changeInUtility;
            }
        }
        iteration++;
    } while (maxChangeInUtility > threshold);

    output.finalise();
}

```

Figure 6: Second code snippet of runValueIteration function

As shown in Figure 6, at the beginning of each iteration, the do-while loop resets the maximum change in utility (`maxChangeInUtility`) to 0. `curCell` gets the current cell based on the current column and row of the maze, and checks if it's a wall or not.

If the `curCell` is a wall: Skip it.

If the `curCell` is NOT a wall: calculate the change in utility (`changeInUtility`) for the current cell based on its neighbours and update the `curCell`'s utility using the `calculateUtility` function (Figure 7).

If the current change in utility is greater than the current maximum change in utility, the maximum change in utility is updated to the current change in utility. This helps to keep track of the overall maximum change in utility across all iterations. The do-while loop continues its iterations until the maximum change in utility is less than the threshold.


```

private static double calculateUtility(Cell curCell, Maze maze) {
    double[] subUtilities = new double[Coordinates.ALL_DIRECTIONS];

    for (int direction = 0; direction < Coordinates.ALL_DIRECTIONS; direction++) {
        Cell[] neighbours = maze.getNeighboursOfCell(curCell, direction);
        double up = Const.PROBABILITY_UP * neighbours[0].getUtility();
        double left = Const.PROBABILITY_LEFT * neighbours[1].getUtility();
        double right = Const.PROBABILITY_RIGHT * neighbours[2].getUtility();

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

    int maximumUtility = 0;
    for (int utility = 1; utility < subUtilities.length; utility++) {
        if (subUtilities[utility] > subUtilities[maximumUtility]) {
            maximumUtility = utility;
        }
    }

    float curReward = curCell.getCellAttribute().getReward();
    double prevUtility = curCell.getUtility();
    double newUtility = curReward + Const.DISCOUNT_FACTOR *
subUtilities[maximumUtility];
    curCell.setUtility(newUtility);
    curCell.setAction(maximumUtility);

    return (Math.abs(prevUtility - newUtility));
}

```

Figure 7: Code snippet of calculateUtility function

The `calculateUtility` function in Figure 7 starts off by creating an array, `double[] subUtilities`, to store the calculated sub-utilities for each direction. It then loops through all possible directions for the cell and calculates the sub-utility value for each direction. This is done by summing up the weighted utility values of the three neighbours (i.e. UP, LEFT, RIGHT) in the given direction. The direction with the maximum sub-utility is determined and sets the utility and action of the current cell to the corresponding values.

The current reward and utility value of the cell is retrieved, and using the Bellman equation, `newUtility = curReward + Const.DISCOUNT_FACTOR * subUtilities[maximumUtility]`, it calculates the new utility value for the cell. With the `maximumUtility` and `newUtility` values obtained, the new utility value and the corresponding action (a.k.a policy) is set for the cell. Finally, the absolute difference between the old and new utility values of the cell is returned.

Overall, for every iteration, `calculateUtility` function is executed for every state in the maze and an action is set based on whether it maximises the expected utility of the subsequent state.

Plot of Optimal Policy

The following figure shows the final plot of the optimal policies of all states after convergence on the 687th iteration by using a C value of 0.1.

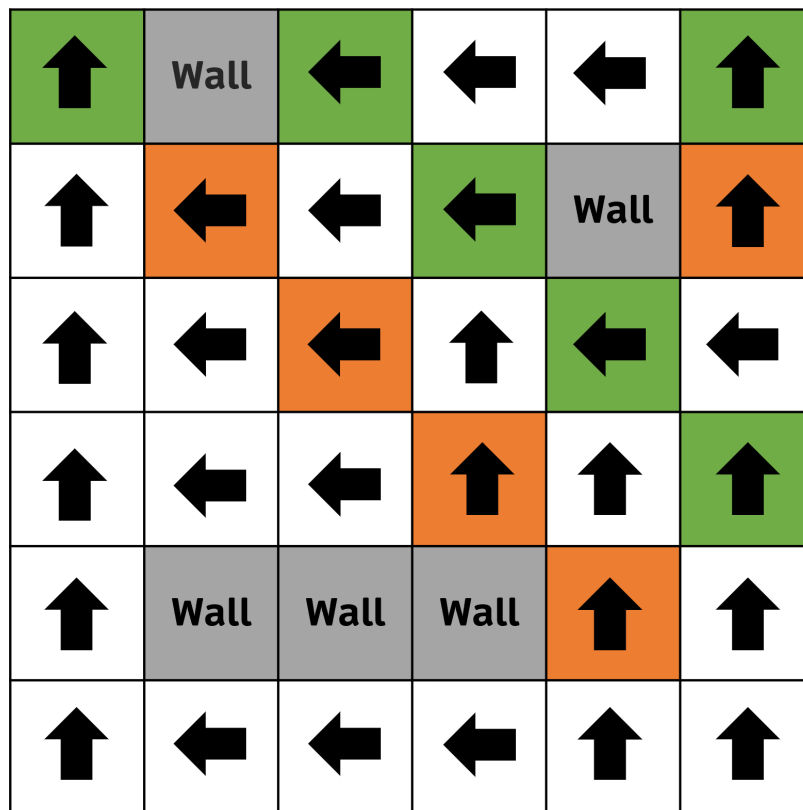


Figure 6: Plot of Optimal Policy for Value Iteration

Utilities of All States

The following table shows the final utilities of all states after convergence on the 687th iteration by using the ϵ value of 0.1.

Table 1: Utility of all states for Value Iteration on the 687th iteration

State/Wall	Value	State/Wall	Value	State/Wall	Value
State(0,0)	99.901	State(2,0)	94.950	State(4,0)	92.561
State(0,1)	98.295	State(2,1)	94.449	Wall(4,1)	0
State(0,2)	96.852	State(2,2)	93.200	State(4,2)	93.010
State(0,3)	95.458	State(2,3)	93.139	State(4,3)	91.723
State(0,4)	94.218	Wall(2,4)	0	State(4,4)	89.458
State(0,5)	92.844	State(2,5)	90.443	State(4,5)	88.480
Wall(1,0)	0	State(3,0)	93.780	State(5,0)	93.236
State(1,1)	95.786	State(3,1)	94.303	State(5,1)	90.826
State(1,2)	95.491	State(3,2)	93.083	State(5,2)	91.704
State(1,3)	94.358	State(3,3)	91.023	State(5,3)	91.798
Wall(1,4)	0	Wall(3,4)	0	State(5,4)	90.477
State(1,5)	91.636	State(3,5)	89.265	State(5,5)	89.209

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

Figure 7 shows the plot of utility estimates as a function of the number of iterations.

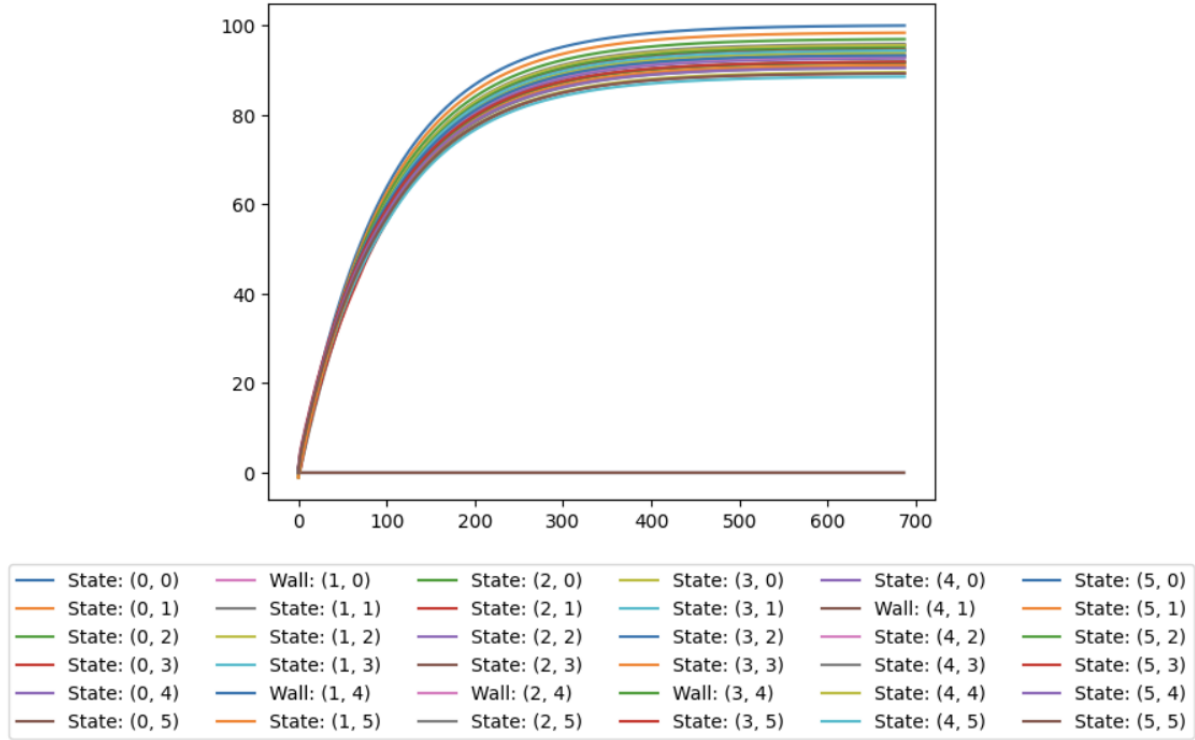


Figure 7: Plot of Utility Estimates as a function of the number of Iterations for Value Iteration

PART 1: POLICY ITERATION

Descriptions of Implemented Solutions

Overview

Policy Iteration: It iteratively evaluates and improves the policy until it converges to an optimal policy. The algorithm consists of two main steps: Policy Evaluation (1) and Policy Improvement (2).

- (1) In Policy Evaluation, the utility of each state in the maze is estimated using the Bellman Equation, which takes into account the expected utility of the successor states.
- (2) In Policy Improvement, the policy of each state is updated to the action that maximises the expected utility of the successor states. The algorithm iterates through each state in the maze (excluding walls) and checks if changing the policy of that state would improve the overall policy. If yes, the policy of that state is updated, and the algorithm continues to the next state.

The algorithm terminates when the policy no longer changes after (2), indicating that it has converged to an optimal policy.

```
public class PolicyIteration {  
    private final static int K = 75;  
  
    public static void main(String[] args) {  
        Maze maze = new Maze("maze.txt");  
        runPolicyIteration(maze);  
    }  
}
```

Figure 8: Declaration of necessary variables for Policy Iteration

Before Policy Iteration can be done on a given maze, the following variable is declared:

- `k` : the number of iterations to estimate the utility of the policy (see “How K is calculated” section below for details)

The `main` function loads the given maze and calls the `runPolicyIteration` function.

How K is calculated

In policy iteration, the value of k is a hyperparameter that determines the number of iterations to estimate the utility of the policy.

A larger k will generally lead to a more accurate approximation of the value function, but will also result in a longer computation time. Conversely, a smaller k will result in faster computation times, but may lead to a less accurate approximation of the value function.

4 different k values were experimented with, and Table 2 shows the results of the number of iterations it took before the policy converged for each of the k values.

As the iterations to converge plateaus at $k = 75$, it will be used as the value of k in the current implementation of Policy Iteration.

Table 2: Iterations to Converge for 4 k values

k	Iterations to Converge
25	12
50	7
75	6
100	6

Code Implementation for Policy Iteration

```
private static void runPolicyIteration(Maze maze) {  
    boolean changeOrNot;  
    int iteration = 1;  
    createOutput output = new createOutput("PolicyIteration", maze);  
    ...  
}
```

Figure 9: First code snippet of `runPolicyIteration` function

In the `runPolicyIteration` function, `changeOrNot` boolean variable is instantiated to track whether any policy changes were made during an iteration so that the do-while loop (shown in Figure 10) terminates when there are no policy changes. `iteration` is initialised to 1.

```

private static void runPolicyIteration(Maze maze) {
    ...
    do {
        changeOrNot = false;
        policyEvaluation(maze, K);

        for (int c = 0; c < Const.NUM_COL; c++) {
            for (int r = 0; r < Const.NUM_ROW; r++) {
                Cell currCell = maze.getCell(new Coordinates(c, r));

                if (currCell.getCellAttribute() == CellAttribute.WALL){
                    continue;
                }

                boolean changed = policyImprovement(currCell, maze);

                if (changed){
                    changeOrNot = true;
                }
            }
        }
        iteration++;
    } while (changeOrNot);

    output.finalise();
}

```

Figure 10: Second code snippet of runPolicyIteration function

As shown in Figure 10, at the beginning of each iteration, the do-while loop resets the `changeOrNot` variable to `false`. The `policyEvaluation` function (see figure 11) is called with the maze and the number of iterations `k` as its input.

It then loops over all cells in the maze and performs policy improvement for each non-wall cell. Calling the `policyImprovement` function (see figure 12) evaluates the optimal action for the cell and updates the cell's action accordingly, and if it returns `true`, indicates that the optimal action for the cell has changed. At the end of the loop,

if `changeOrNot` is still **true**: the policy iteration loop continues.

If `changeOrNot` is **false**: the loop terminates and the final policy has been determined.

```

private static void policyEvaluation(Maze maze, int k) {
    for (int i = 0; i < k; i++) {
        for (int column = 0; column < Const.NUM_COL; column++) {
            for (int row = 0; row < Const.NUM_ROW; row++) {
                Cell curCell = maze.getCell(new Coordinates(column, row));

                if (curCell.getCellAttribute() == CellAttribute.WALL)
                    continue;

                Cell[] neighbours = maze.getNeighboursOfCell(curCell);
                double up = Const.PROBABILITY_UP * neighbours[0].getUtility();
                double left = Const.PROBABILITY_LEFT * neighbours[1].getUtility();
                double right = Const.PROBABILITY_RIGHT * neighbours[2].getUtility();

                float reward = curCell.getCellAttribute().getReward();
                curCell.setUtility(reward + Const.DISCOUNT_FACTOR * (up + left + right));
            }
        }
    }
}

```

Figure 11: Code snippet of policyEvaluation function

The goal of policy evaluation is to update the utilities of each state in the environment given the current policy. The Bellman equation for the utility of a state s is: $U(s) = R(s) + \gamma * \sum (P(s, a, s') * U(s'))$, where $U(s)$ is the utility of state s , $R(s)$ is the reward for being in state s , γ is the discount factor 0.99, $P(s, a, s')$ is the probability of transitioning from state s to state s' when taking action a and $\sum (P(s, a, s') * U(s'))$ is the expected utility of the next state s' given the current state s and action a .

In figure 11, the outermost for loop iterates k times over all states and updates the utility of each state based on the current policy. In the inner for loop, `curCell` gets the current cell based on the current column and row of the maze, and checks if it's a wall or not.

If the `curCell` is a wall: Skip it.

If the `curCell` is NOT a wall: the algorithm sums up the expected utilities of its neighbours (UP, LEFT, RIGHT) based on the current policy.

The reward of `curCell` is obtained and the utility of the current state is updated using the Bellman equation.


```

private static boolean policyImprovement(Cell currCell, Maze maze) {
    double[] maxSubUtility = new double[Coordinates.ALL_DIRECTIONS];
    for (int direction = 0; direction < Coordinates.ALL_DIRECTIONS; direction++) {
        Cell[] neighbouringCells = maze.getNeighboursOfCell(currCell, direction);
        double up = Const.PROBABILITY_UP * neighbouringCells[0].getUtility();
        double left = Const.PROBABILITY_LEFT * neighbouringCells[1].getUtility();
        double right = Const.PROBABILITY_RIGHT * neighbouringCells[2].getUtility();
        maxSubUtility[direction] = up + left + right;
    }
    Cell[] neighbours = maze.getNeighboursOfCell(currCell);
    double up = Const.PROBABILITY_UP * neighbours[0].getUtility();
    double left = Const.PROBABILITY_LEFT * neighbours[1].getUtility();
    double right = Const.PROBABILITY_RIGHT * neighbours[2].getUtility();
    int maxSubUtilityDirection = 0;
    for (int m = 1; m < maxSubUtility.length; m++) {
        if (maxSubUtility[m] > maxSubUtility[maxSubUtilityDirection])
            maxSubUtilityDirection = m;
    }
    double currSubUtility = up + left + right;
    if (maxSubUtility[maxSubUtilityDirection] > currSubUtility) {
        currCell.setAction(maxSubUtilityDirection);
        return true;
    } else {
        return false;
    }
}

```

Figure 12: Code snippet of policyImprovement function

The goal of policy improvement is to improve the policy until we reach an optimal policy that maximises the expected total reward for the agent. We update the policy of each non-wall cell in the maze based on the results of policy evaluation. This process continues until the policy is converged and no further changes are required.

The `policyImprovement` function in Figure 12 starts off by creating an array, `double[] maxSubUtility`, to store the maximum possible sub-utility for each direction. The for loop iterates through each direction and calculates the sub-utility for that direction based on the current policy. The neighbouring cells of `currCell` are stored in the array `neighbours` and the sub-utility for the current policy is calculated based on the neighbouring cells and their utilities. The direction with the highest sub-utility value is stored in the `maxSubUtilityDirection` variable. It is compared with `currSubUtility`, which holds the sub-utility value for the current policy, to check if the sub-utility value for the new policy is greater than the sub-utility value for the current policy. If yes, the action for `currCell` is updated to the direction with the highest sub-utility and returns `true`, which indicates that the optimal action for the cell has changed.

Plot of Optimal Policy

The following figure shows the final plot of the optimal policies of all states after convergence on the 6th iteration by using a k value of 75.

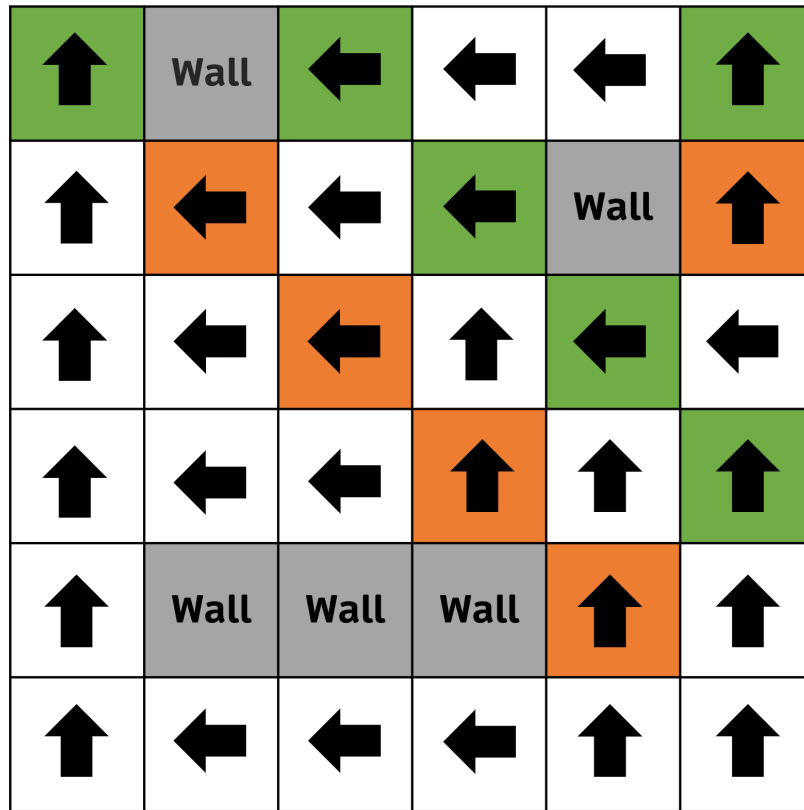


Figure 13: Plot of Optimal Policy for Policy Iteration

Utilities of All States

The following table shows the final utilities of all states after convergence on the 6th iteration by using a k value of 75.

Table 3: Utility of all states for Policy Iteration on the 6th iteration

State/Wall	Value	State/Wall	Value	State/Wall	Value
State(0,0)	98.925	State(2,0)	94.008	State(4,0)	91.638
State(0,1)	97.331	State(2,1)	93.507	Wall(4,1)	0
State(0,2)	95.898	State(2,2)	92.267	State(4,2)	92.100
State(0,3)	94.515	State(2,3)	92.215	State(4,3)	90.824
State(0,4)	93.284	Wall(2,4)	0	State(4,4)	88.569
State(0,5)	91.921	State(2,5)	89.539	State(4,5)	87.599
Wall(1,0)	0	State(3,0)	92.848	State(5,0)	92.322
State(1,1)	94.832	State(3,1)	93.373	State(5,1)	89.922
State(1,2)	94.548	State(3,2)	92.163	State(5,2)	90.805
State(1,3)	93.424	State(3,3)	90.113	State(5,3)	90.908
Wall(1,4)	0	Wall(3,4)	0	State(5,4)	89.596
State(1,5)	90.722	State(3,5)	88.370	State(5,5)	88.337

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

Figure 7 shows the plot of utility estimates as a function of the number of iterations.

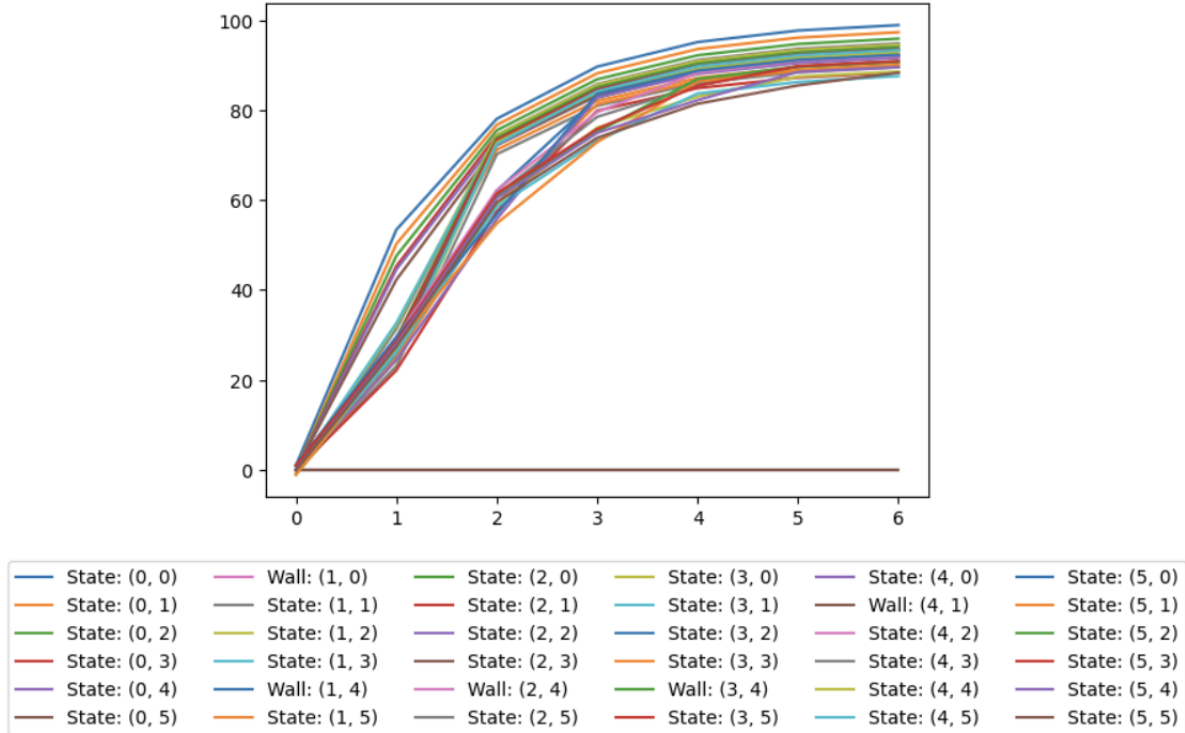


Figure 13: Plot of Utility Estimates as a function of the number of Iterations for Policy Iteration

Observations

- The final plot of the optimal policies of all states after convergence for both Value Iteration and Policy Iteration is the same.
- As compared to the Value Iteration algorithm which converged at the 687th iteration, the Policy Iteration algorithm was able to achieve convergence much earlier at the 6th iteration.

PART 2: BONUS QUESTION

Data Preparation

Increasing the dimensions of the maze is one of the ways to increase the complexity of the maze environment. This will result in the number of states and the number of possible actions in the environment to increase, implicitly implying that there will be longer calculation time and larger memory required to run both Value Iteration and Policy Iteration.

Creation of Mazes

Mazes of different sizes (i.e. 6x6, 12x12, 24x24, 48x48) are randomly generated using the Python code in figure 14, and then placed into individual .txt files for testing.

```
import random
import string

list = ["X", "G", "W", "B"]

for x in range(48): // change range value to 6, 12, 24 or 48
    for y in range(48): // change range value to 6, 12, 24 or 48
        print( random.choice(list)+' ',end='')
    print()
```

Figure 14: Python code to generate mazes

Figure 15 shows the different mazes of different sizes that were randomly generated, 5 for each size variation. Value Iteration and Policy Iteration are both used to run on the mazes and the iterations needed to converge for both are recorded.

6 x 6 Mazes	12 x 12 Mazes	24 x 24 Mazes	48 x 48 Mazes
complicated-maze-6x6-1.txt	complicated-maze-12x12-1.txt	complicated-maze-24x24-1.txt	complicated-maze-48x48-1.txt
complicated-maze-6x6-2.txt	complicated-maze-12x12-2.txt	complicated-maze-24x24-2.txt	complicated-maze-48x48-2.txt
complicated-maze-6x6-3.txt	complicated-maze-12x12-3.txt	complicated-maze-24x24-3.txt	complicated-maze-48x48-3.txt
complicated-maze-6x6-4.txt	complicated-maze-12x12-4.txt	complicated-maze-24x24-4.txt	complicated-maze-48x48-4.txt
complicated-maze-6x6-5.txt	complicated-maze-12x12-5.txt	complicated-maze-24x24-5.txt	complicated-maze-48x48-5.txt

Figure 15: txt files of different mazes of different sizes to be tested

Figure 16 shows the source code changes that were made in the `const` class designed in Part 1 to accommodate the larger maze dimensions. `NUM_COL` and `NUM_ROW` values change depending on which maze will be used during a run on both algorithms.

```
public class Const {
    public final static int NUM_COL = 24; // change value to 6, 12, 24 or 48
    public final static int NUM_ROW = 24; // change value to 6, 12, 24 or 48
    ...
}
```

Figure 16: Changes to `const` class

Approach & Experimentation

Experimentation was first done by running the implemented Value Iteration and Policy Iteration algorithms on individually designed complicated mazes. However, no matter how complicated (e.g. surrounding green states with either walls or brown states to make it harder to obtain the optimal policy) the maze is, the number of iterations do not exceed 687.

Therefore, an alternative experiment is conducted with multiple mazes of different dimensions to investigate the relationship between complexity and convergence.

With the txt files mentioned in figure 15, each maze file that ends with “-1.txt” will be labelled as M1, maze files that end with “-2.txt” will be labelled as M2, and so on.

6 x 6 Mazes Outputs

Another five 6 x 6 mazes were generated, to investigate any change in the number of iterations to converge as compared to the preset maze in Part 1.

Table 4 shows the number of iterations to converge for Value Iteration and Policy Iteration for the 5 generated 6 x 6 mazes.

Table 4: Value Iteration and Policy Iteration Output by 6x6 Mazes

Maze Size	Iterations to Converge									
	Value Iteration					Policy Iteration				
6 x 6	M1	M2	M3	M4	M5	M1	M2	M3	M4	M5
	687	678	414	677	568	9	6	5	5	7

12 x 12 Mazes Outputs

Table 5 shows the number of iterations to converge for Value Iteration and Policy Iteration for the 5 generated 12 x 12 mazes.

Table 5: Value Iteration and Policy Iteration Output by 12x12 Mazes

Maze Size	Iterations to Converge									
	Value Iteration					Policy Iteration				
12 x 12	M1	M2	M3	M4	M5	M1	M2	M3	M4	M5
	687	687	672	612	687	11	11	11	10	10

24 x 24 Mazes Outputs

Table 6 shows the number of iterations to converge for Value Iteration and Policy Iteration for the 5 generated 24 x 24 mazes.

Table 6: Value Iteration and Policy Iteration Output by 24x24 Mazes

Maze Size	Iterations to Converge									
	Value Iteration					Policy Iteration				
24 x 24	M1	M2	M3	M4	M5	M1	M2	M3	M4	M5
	687	659	672	687	687	11	32	11	13	12

48 x 48 Mazes Outputs

Table 7 shows the number of iterations to converge for Value Iteration and Policy Iteration for the 5 generated 48 x 48 mazes.

Table 7: Value Iteration and Policy Iteration Output by 48x48 Mazes

Maze Size	Iterations to Converge									
	Value Iteration					Policy Iteration				
48 x 48	M1	M2	M3	M4	M5	M1	M2	M3	M4	M5
	687	687	687	687	687	41	32	38	40	37

Results

The outputs from Tables 5 - 7 have been summarised in Table 8 as ranges of the number of iterations to converge for both Value Iteration and Policy Iteration for each maze size variation.

Table 8: Number of iterations to converge for both Value Iteration and Policy Iteration

Maze Dimensions	Iterations to Converge	
	Value Iteration	Policy Iteration
6 x 6	414 - 687	5 - 9
12 x 12	612 - 687	10 - 11
24 x 24	659 - 687	11 - 32
48 x 48	687	32 - 41

Observations for Value Iteration

Based on the values obtained, it can be observed that the number of iterations to converge for value iteration does not exceed 687 as the complexity increases, with the 48x48 mazes outputting a constant number of iterations to converge. It can be concluded that the complexity slightly affects the lower bound for convergence as it increases, with no change observed for the upper bound.

Observations for Policy Iteration

Based on the values obtained, it can be observed that the number of iterations to converge for policy iteration steadily increases. It can be concluded that for Policy Iteration, convergence gradually increases as complexity increases.

How does the number of states and the complexity of the environment affect convergence?

As the number of states and the complexity of the environment increases, the convergence time of the algorithm also increases. This is due to the fact that more complex environments have more possible states and actions, and therefore increasing the search space for the algorithm to find an optimal policy. As a result, the algorithm needs more time to explore the state space and converge to an optimal policy.

How complex can you make the environment and still be able to learn the right policy?

The ability to learn the right policy in Value Iteration and Policy Iteration depends on the properties and structure of the environment, such as the dimensions of the maze in GridWorld, as well as the specific algorithm parameters used, such as the c value in Value Iteration and the k value in Policy Iteration. Generally, if the agent has enough time and computational resources to learn and the environment is well-structured, it should be able to learn the optimal policy even in complex environments.