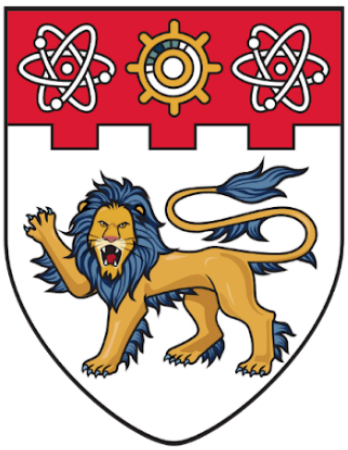


NANYANG TECHNOLOGICAL UNIVERSITY

SINGAPORE

CE/CZ4046: INTELLIGENT AGENTS

ASSIGNMENT 1: AGENT DECISION MAKING

AGARWAL ANUSHA(U2023105H)

School of Computer Science & Engineering

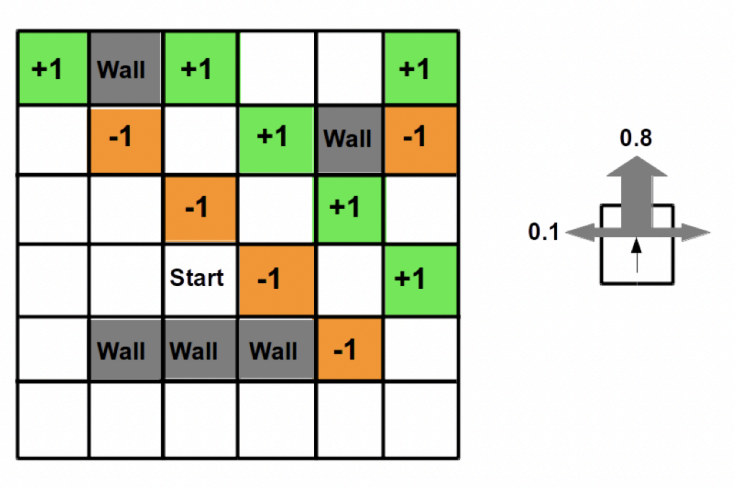
Nanyang Technological University

Table of Contents

1. INTRODUCTION **3**
2. PROPOSED SOLUTION FRAMEWORK **4**
2.1. FRAMEWORK DETAILS - 4
2.2. IMPLEMENTATION TOOLS - 7
3. SOLUTION IMPLEMENTATION DESCRIPTION **7**
3.1. GRID ENVIRONMENT INITIALIZATION 7
3.2. NEIGHBORHOOD OF CURRENT STATE 8
3.3. VALUE ITERATION 12
3.4. POLICY ITERATION 15
4. RESULT ANALYSIS **18**
4.1. VALUE ITERATION 18
4.2. POLICY ITERATION 26
5. BONUS QUESTION - **31**
5.1. COMPLICATED GRID SETUP: 31
5.2. LARGER GRID SETUP 37

1. INTRODUCTION

Agent decision making refers to the ability of an agent to make decisions based on available information to achieve intended goals. In this context, we are investigating a given grid environment to find the optimal policy (maximum reward) in any state.



Transition Probability in this grid environment suggests that the agent moves in the intended direction with a 0.8 probability. There is a 0.1 probability for the agent to digress to the right of the intended direction, and a 0.1 probability for the agent to digress to the left of the intended direction. If the movement in any direction leads to collision with the wall, the agent remains stationary.

Rewards are set as Green States with +1 reward, Brown States with -1 reward, and White States with -0.04 reward. Since there is no terminal state, the agent's state sequence is limitless.

To solve this challenge, we intend to find the optimal policy that results in maximum achievable reward in any given state. To achieve this, the agent must navigate the entire path and acquire knowledge about the optimal path. This algorithm is an example of how intelligent rational agents make decision in complicate environment

2. PROPOSED SOLUTION FRAMEWORK

2.1. FRAMEWORK DETAILS -

In the provided grid environment, Value Iteration and Policy Iteration algorithms are used to find the optimal policy in any given state.

The solution framework, along with details about utilized packages and Java classes is outlined below.

- Package: **entities** -

- **Constants.java**: This class contains predefined constant entities marked as final to avoid further changes such as
  - Reward of given state
  - Transition probability of actions
  - Discount Factor
  - Maximum Reward
  - Constant C
  - Upper bound on utility values
  - Epsilon for maximum allowable error
  - K for number of times the Bellman algorithm is executed
- **Policy.java**: This is an enum class defining possible directions (UP, DOWN, LEFT, RIGHT) and their corresponding symbols (↑,↓,←,→). Functions implemented in this class are:
  - getSymbol(): To get corresponding symbol to action
  - getDirection(): Get direction
- **State.java**: This class initialized states with some coordinates. Default initialization of states: Utility = 0, Policy = UP, StateType = WHITE. It defines setter-getter functions including
  - getStateType(): to get type of state (White, Brown, Green, Wall)
  - setStateType(): to set type of state and its corresponding utility.
  - getUtility(): to get utility of state
  - setUtility(): to set utility of state
  - getPolicy(): to get policy of state
  - setPolicy(): to set policy given a direction value
- **StateCoordinate.java**: This class defines number to directions with Up = 0, Down = 1, Left = 2, Right = 3. It defines coordinates of a state in the grid environment. It defines the following functions:
  - getCol(): Gets col corresponding to a state
  - getRow(): Gets row corresponding to a state
  - getNeighbours(): Find the neighboring coordinates of given state i.e. coordinates if the agent moves up/left/right. For this we define the coordinate offset.
- **StateType.java**: This is an enum class defining types of state (WHITE, BROWN, GREEN, WALL). It implements the following functions-
  - getReward(): gets the reward corresponding to each state (WHITE: -0.04, BROWN: -1.0, GREEN: +1.0, WALL: 0.0)
  - getSymbol(): gets character symbol corresponding to each state (WHITE: W, BROWN: B, GREEN: G, WALL: X)

❖ Package: **controller**-

- **DisplayController.java**: This is a background class not called directly. It is a print class to enable user-friendly display of different outputs. It implements the following functions-

        - printExperimentParameters(): To print the Algorithm Parameters such as R_Mac, Epsilon, C, Discount factor as set in Value Iteration.
        - print(): To print a consolidated grid world with state type, policy direction and reward.
        - printGridWorld(): To print grid world with state type symbols (W,B,G,X) in the grids
        - printPolicy(): To print a grid with suggested policy action in each state (↑,↓,←,→).
        - printStateUtilities(): To print a list of each state and its corresponding utility
        - printUtilityGrid(): To print utilities of each state in a grid format
        - frameTitle(): To print in the title of what kind output will be shown next.
    - **GridEnvironment.java**: This class is used to create the grid environment, taking data from the presetGridWorld text files. It also acts as the boundary class to call functions in DisplayController.java and NeighbourhoodStates.java. It implements the following functions -
        - importGridFromFile(): to import the grid world and define a maze environment using the presetGridWorld given in text file.
    - **NeighbourhoodStates.java**: This class is used to implement functions to find the neighborhood possible states given the current state (UP, LEFT, RIGHT) taking care if an agent runs to the wall and stays stationary-
        - getCell(): Takes as input the coordinate and the cells[][] array, and outputs the states at cell coordinates
        - getNeighboursOfCell(): Takes as input the current cell and the cells[][] array. It outputs the corresponding neighbors (UP,LEFT,RIGHT) with respect to current Policy
        - getNeighborsOfCell(): This is an overload function to previous function which returns corresponding neighbors given the current state, cell[][] and also the PolicyDirection.
- Package: **fileManager**-
    - **LogFileController.java**: This class finds the header states and entire data of utilities of each state spanning across iterations to be stored in the csv file using the following functions-
        - add(): Function to take as input the grid and keep adding the incoming List of utilities for corresponding states in each iteration
        - finalConvertToCSV(): this function writes the entire data into CSV file using wrtiteToCSV.writeToFile() function.
    - **writeToCSV**.java: This class writes the headers (states) and data (utility of each state in each iteration) to a CSV file. It implements the following method -

        - writeToFile(): Takes in input header and data to be written in csv format.
    - **UtilityPlots.ipynb**: This class is used to plot the utility of states across iterations. It implements the following functions-
        - processFilesInFolder(): It loops the csv files in given folder to run utility plot function on each file
        - plotUtility(): This function plots the utility of each state spanning across iterations
- Package: **main-**
    - ValueIteration.java: This Java class implements the Value Iteration Algorithm for the grid environment to find the optimal policy in each state using the following function -
        - runValueIteration(): Given the grid environment, run the value iteration algorithm on it
        - calculateUtilityChange(): Calculate the change in utility from previous policy and updated policy - used for convergence.
    - PolicyIteration.java: This Java class implements the Policy Iteration Algorithm for the grid environment to find the optimal policy in each state using the following function -
        - runPolicyIteration(): given the grid environment, run the policy iteration algorithm on it
        - policyEvaluation(): To evaluate policy by calculating and updating utilities given the policy actions
        - policyImprovement(): To check if there is a better action that gives bigger utility and return the best policy
- Folder: **logs-**
    - This folder contains csv output files with each state utilities spanning across iterations
- Folder: **presetGridWorld-**
    - This folder contains the text files with grid environment

2.2. IMPLEMENTATION TOOLS -

This algorithm is executed in Java with JVM version - 17.0.8.1. Platform used is Visual Studio Code. To visualize CSV as a dataframe and to plot utility of states across iterations, Jupyter Notebook has been used.

3. SOLUTION IMPLEMENTATION DESCRIPTION

3.1. GRID ENVIRONMENT INITIALIZATION

A new grid environment is initialized using the GridEnvironment.java class. Default constructor is used to define the grid states to initial values- a grid of predefined number of rows and columns, with each state set to default Utility = 0, Policy = UP, and stateType = WHITE.

Parameterized constructor *GridEnvironment(String filename)* takes in the text file with the grid environment. It calls the function *importGridFromFile(filename)* to set the grid world using the grid from the file. It defines the State type and utility of each state where

State Symbol	State Type	Reward
W	WHITE	-0.04
B	BROWN	-1.0
G	GREEN	+1.0
X	WALL	0.0

```java
public class GridEnvironment {
    State[][] cells;
    int numOfCol, numOfRow;

    /**
     * Initialise Grid with speicified numb of rows and columns. States are set to default values
     * Default Values: Utility = 0, Policy = UP, StateType = WHITE
     * @param col
     * @param row
     */
    public GridEnvironment() {
        if (numOfCol < 0 || numOfRow < 0)
            throw new IllegalArgumentException(s:"Column and Row must be a positive integer.");

        this.numOfCol = Constants.NUM_COL;
        this.numOfRow = Constants.NUM_ROW;
        this.cells = new State[this.numOfCol][this.numOfRow];

        for (int c = 0; c < numOfCol; c++) {
            for (int r = 0; r < numOfRow; r++) {
                StateCoordinate coordinates = new StateCoordinate(c, r);
                cells[c][r] = new State(coordinates);
            }
        }

    }

    //Paramterized constructor to extract states from the text file
    public GridEnvironment(String fileName) {
        this();
        this.importGridFromFile(fileName);
    }
```

Fig 3.1.1

```java
/**
 * Import the grid world states from text file
 * @param currentCell
 */
public void importGridFromFile(String fileName) {
    try {
        String filePath = new File(pathname:"").getAbsolutePath();
        Scanner s = new Scanner(new BufferedReader(new FileReader(filePath.concat("/presetGridWorlds/" + fileName))));

        while (s.hasNext()) {
            for (int r = 0; r < this.numOfRow; r++) {
                for (int c = 0; c < this.numOfCol; c++) {
                    char stateType = s.next().charAt(index:0);
                    cells[c][r].setStateType(stateType);
                }
            }
        }

        s.close();
    } catch (Exception e) {
        System.out.println(e);
    }
}
```

Fig 3.1.2 - importGridFromFile

3.2. NEIGHBORHOOD OF CURRENT STATE

Neighborhood of the current state needs to be defined to later be able to get expected utility using transition probabilities and utilities of each neighborhood state. As previously defined, the agent moves in the intended direction with a probability of 0.8 and moves to either left or right of intended direction with 0.1 probability.
We define two overloaded functions *getNeighbourhoodOfCell()* to return a neighborhood array with [Intended Position, Left Angle (L), Right Angle (R)]. It takes care that if there is a wall in the neighborhood (forward, left, or right) then the agent stays in the current state.

3.2.1. getNeighbourhoodOfCell(State currentCell, State cells[][]) : takes in input the current cell and cells[][] grid. It returns an array with neighborhood cells with respect to the current cell, and the policy defined in that cell using getPolicy() and getDirection() functions on current state.

```java
/**
 * Get the corresponding neighbours (UP, LEFT, RIGHT) wrt current Policy
 * @param currentCell
 * @param cells
 * @return [Intended Position, Left Angle (L), Right Angle (R)]
 */
public static State[] getNeighboursOfCell(State currentCell, State cells[][]) {
    Policy currentPolicy = currentCell.getPolicy();
    StateCoordinate[] neighbourCoordinates = currentCell.getNeighbours(currentPolicy.getDirection());

    //Ensure the neighbour stateType is not a wall
    State[] neighbourCells = new State[neighbourCoordinates.length];
    for (int n = 0; n < neighbourCoordinates.length; n++) {
        State neighbourCell = getCell(neighbourCoordinates[n],cells);

        //If neighout a wall, stay in current state
        if (neighbourCell.getStateType() == StateType.WALL)
            neighbourCoordinates[n] = (StateCoordinate) currentCell;

        neighbourCells[n] = getCell(neighbourCoordinates[n],cells);
    }
    //Collect possible neighbourhood of current state
    return neighbourCells;
}
```

Fig 3.2.1.1 - getNeighboursOfCell

3.2.2. getNeighbourhoodOfCell(State currentcell, int direction, State cells[][]): takes in input the current cell, cells[][] grid as well as the direction policy. Here, the direction is explicitly given, and thereby neighborhood cells are calculated.

```java
/**
 * Get the corresponding neighbours (UP, LEFT, RIGHT) wrt given Policy Direction
 * @param currentCell
 * @param direction
 * @param cells
 * @return [Intended Position, Left Angle (L), Right Angle (R)]
 */
public static State[] getNeighboursOfCell(State currentCell, int direction, State cells[][]) {
    StateCoordinate[] neighbourCoordinates = currentCell.getNeighbours(direction);

    /* Make sure neighbour stateType is not a wall */
    State[] neighbourCells = new State[neighbourCoordinates.length];
    for (int n = 0; n < neighbourCoordinates.length; n++) {
        State neighbourCell = getCell(neighbourCoordinates[n],cells);

        //If neighout a wall, stay in current state
        if (neighbourCell.getStateType() == StateType.WALL)
            neighbourCoordinates[n] = (StateCoordinate) currentCell;

        neighbourCells[n] = getCell(neighbourCoordinates[n],cells);
    }
    //Collect possible neighbourhood of current state
    return neighbourCells;
}
```

Fig 3.2.2.1 - getNeighboursOfCell()

Both these functions use -

1. getCell() to find the cell of State datatype at given coordinates

```java
public static State getCell(StateCoordinate coordinate,State cells[][]) {
    int c = coordinate.getCol();
    int r = coordinate.getRow();

    return cells[c][r];
}
```

Fig 3.2.1 - getCell()

2. **getNeighbours()** function defined in StateCoordinate class. It uses offset logic the get coordinates of neighborhood cells. Eg. if agent is at location (col, row): (2,1) and the agent intends to move forward in right direction, to (3,1), then neighbors are calculated as explained below -

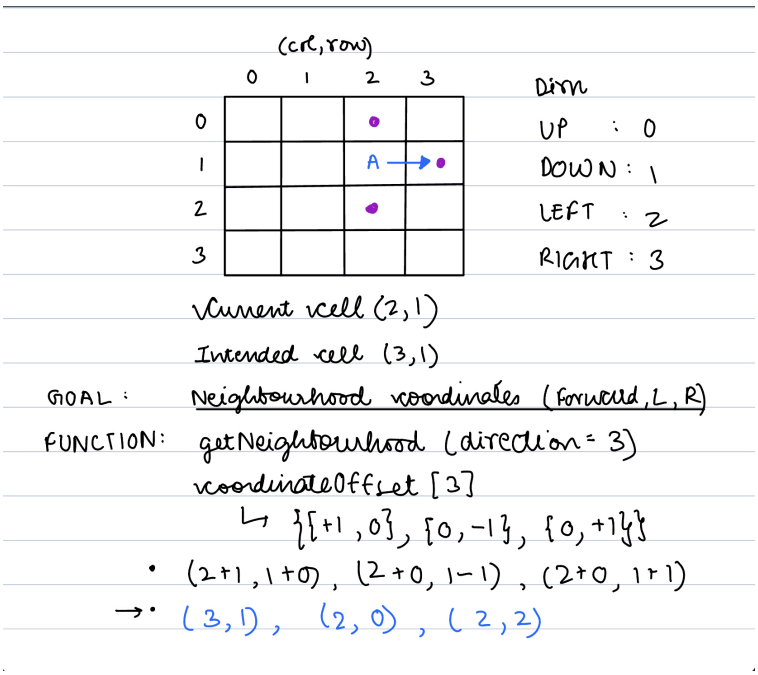


Fig 3.2.2

```java
/**
 * Find the neighbouring possible coordinates with respect to provided direction.
 * @return [Intended Position, Left Angle (L), Right Angle (R)]
 */
public StateCoordinate[] getNeighbours(int direction) {

    //Neighbouring Coordinates for possible movement direction
    //[UP (Forward), Left Dir (L), Right Dir (R)]
    StateCoordinate[] coordinates = new StateCoordinate[3];

    /*
     * Coordinate Offset: 4 x 3 x 2
     * 4x: [UP, DOWN, LEFT, RIGHT]
     * 3x: [UP (Forward), Left Dir (L), Right Dir (R)]
     * 2x: [col, row]
     */

    //Eg if Agent is at 2,1
    //Intended Action is to go up
    //Possible neighbours are -
    // Actually Moves up -> (2,0)
    // Moves Left -> (1,1)
    // Moves Right-> (3,1)

    int[][][] coordinateOffset = { { { 0, -1 }, { -1, 0 }, { +1, 0 } },
                                   { { 0, +1 }, { +1, 0 }, { -1, 0 } },
                                   { { -1, 0 }, { 0, +1 }, { 0, -1 } },
                                   { { +1, 0 }, { 0, -1 }, { 0, +1 } } };

    //UP: Forward
    try {
        coordinates[0] = new StateCoordinate(this.col + coordinateOffset[direction][0][0], this.row + coordinateOffset[direction][0][1]);
    } catch (IllegalArgumentException e) {
        coordinates[0] = this;
    }

    //LEFT: Left Angle
    try {
        coordinates[1] = new StateCoordinate(this.col + coordinateOffset[direction][1][0], this.row + coordinateOffset[direction][1][1]);
    } catch (IllegalArgumentException e) {
        coordinates[1] = this;
    }

    //RIGHT: Right Angle
    try {
        coordinates[2] = new StateCoordinate(this.col + coordinateOffset[direction][2][0], this.row + coordinateOffset[direction][2][1]);
    } catch (IllegalArgumentException e) {
        coordinates[2] = this;
    }

    return coordinates;
}
```

Fig 3.2.3 - getNeighbours()

3.3. VALUE ITERATION

The implementation of the Value Iteration algorithm can be found in the ValueIteration.java class.

In Value Iteration, Bellman equation is used to update the utility of each state till the algorithm converges. The utility of each state is given by:

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

The implementation of this algorithm can be summarized in the following steps -

1. Initialize the utility of each state as 0, and the policy as UP

2. Import grid environment
3. Set maximum error threshold using epsilon and discount factor

$$\boldsymbol{Threshold} = \frac{\epsilon(1-\gamma)}{\gamma}$$

$$\boldsymbol{Epsilon} = C * Rmax$$

4. Repeat until maximum change in utility is greater than threshold -
    a. For each state: Calculate maximum utility and set the policy to action that maximizes the expected utility
    b. Find the maximum change in current utility and updated utility

This is implemented using the following two functions -

1. runValueIteration(GridEnvironment grid):
   This function inputs the grid world and runs value iteration on it. This is done by defining the error threshold and iterating the policy till convergence (stop when maximum change in current utility and updated utility becomes less than error threshold value). Within each iteration, run through each state (except Wall) and set each state policy to maximum reward using *calculateUtilityChange()*. **In each iteration we get some policy for the entire gridWorld and we stop when this policy no longer changes.**

   Each iteration results are added to the log csv file using *logger.add(grid)* and once all iterations are completed this logger converts the data to CSV file with state headers and utilities of each state in each iteration.

```java
/**
 * Function to run value Iteration on Grid
 * @param grid
 */
private static void runValueIteration(GridEnvironment grid) {
    double threshold = Constants.EPSILON * ((1 - Constants.DISCOUNT_FACTOR) / Constants.DISCOUNT_FACTOR);
    LogFileController logger = new LogFileController(fileName:"ValueIteration", grid);

    //Display grid with State Type, Reward, and Initial Policy
    System.out.println(x:"Grid World Original :");
    grid.print();
    logger.add(grid);

    double maxChangeInUtility = 0;
    int iteration = 1;
    do {
        // Calculate Maximum change in utility
        maxChangeInUtility = 0;

        //Run loop for 1 iteration
        for (int c = 0; c < Constants.NUM_COL; c++) {
            for (int r = 0; r < Constants.NUM_ROW; r++) {
                State currentCell = grid.getCell(new StateCoordinate(c, r));

                //If current cell is a wall then skip
                if (currentCell.getStateType() == StateType.WALL)
                    continue;

                // Find change in utility
                double changeInUtility = calculateUtilityChange(currentCell, grid);
                if (changeInUtility > maxChangeInUtility)
                    maxChangeInUtility = changeInUtility;
            }
        }

        iteration++;
        // maze.print();
        logger.add(grid);
    } while (maxChangeInUtility > threshold);

    System.out.printf(format:"Total Iterations to Converge : %d\n", iteration-1);
    System.out.printf(format:"Change in utility from previous iteration: %5.3f\n", maxChangeInUtility);
    grid.printExperimentParamters(isValueIteration:true,threshold);
    grid.printPolicy();
    grid.printStateUtilities();
    grid.printUtilityGrid();
    logger.finalConvertToCSV();
}
```

Fig 3.3.1 - runValueIteration

2. calculateUtilityChange(State currentCell, GridEnvironment grid):
   In this function we calculate the change in utility between current state and updated state. This is implemented using 4 steps for current state -
   a. Find the expected utility of each possible action (here directions - UP, DOWN, LEFT, RIGHT) in that state.
      Expected Utility = $\Sigma P(s'|s,a)U[s']$
   b. Find the maximum possible utility
      The item with the maximum utility is identified using *maxUtilityIndex* so that action can be updated into each state (0 = UP, 1 = DOWN, 2 = LEFT, 3 = RIGHT)

c. Use bellman equation to calculate update utility using
U = Current Reward + Discount_Factor*(Max Expected utility)
Set current policy and utility to updated policy with maximum utility

d. Return the change between previous policy utility and current policy utility (best new utility)

```java
/**
 * Calculate the utility of the given State.
 * @param currentCell
 * @param grid
 * @return The difference previous Utility and new update Utility
 */
private static double calculateUtilityChange(State currentCell, GridEnvironment grid) {
    double[] subUtilities = new double[StateCoordinate.TOTAL_DIRECTIONS];

    //1. Find all possible utilities (i.e. 4 possible directions)
    for (int dir = 0; dir < StateCoordinate.TOTAL_DIRECTIONS; dir++) {
        //1.1 Add the expected utilities of neighbouring cells (i.e. UP, LEFT, RIGHT)
        State[] neighbours = grid.getNeighboursOfCell(currentCell, 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();

        //Sub state utility
        subUtilities[dir] = up + left + right;
    }

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

    //3. Set utility & policy of current cell
    float currentReward = currentCell.getStateType().getReward();
    double prevUtility = currentCell.getUtility();

    //Apply Simplified Belmman equation
    //Utility of a state = currentStateReward + Max Possible Discounted Future Reward
    double newUtility = currentReward + Constants.DISCOUNT_FACTOR * subUtilities[maxUtilityIndex];
    currentCell.setUtility(newUtility);
    currentCell.setPolicy(maxUtilityIndex);

    //4. Return the difference of prevUtility & newUtility
    // Converges when this new utility becomes very close to previous utility
    return (Math.abs(prevUtility - newUtility));
}
```

Fig 3.3.2 - calculateUtilityChange()

It is worth noting that the starting state of the agent has no impact on the determined optimal policy as there is no terminal state or predetermined time for finishing the task.
We thereby get the optimal policy using this algorithm which is discussed later in the Results section.

3.4. POLICY ITERATION

The implementation of Policy Iteration algorithm can be found in the *PolicyIteration.java* class.

In **Policy Iteration**, the algorithm alternates between **Policy Evaluation** and **Policy Improvement**
The Bellman update is given by -

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

This class is implemented using 3 functions as discussed below -

1. **runPolicyIteration(GridEnvironment grid):**

The **policy iteration** algorithm alternates the following two steps, beginning from some initial policy $\pi 0$ = UP (in our case):
This algorithm can be summarized in the following steps -

1. Initialize utility of each state to 0, and policy action as UP
2. Import the grid environment
3. Initialize a flag variable called *PolicyChanged* to false. This is to keep track of policy changes
4. Repeat until convergence (stop when *policyChanged* becomes False)
    a. Set policyChanged to false
    b. For each state: Perform Policy evaluation k times
    c. For each state: Perform Policy Improvement
        i. Return true if policy changed
    d. Update policyChanged to value returned from policy improvement
    e. Exit if policyChanged = false

```java
/**
 * This function is used to run Policy Iteration on the given grid world to find optimal Policyp
 * @param grid
 */
private static void runPolicyIteration(GridEnvironment grid) {
    //Flag to check if there is a change in policy (new and updated)
    boolean policyChanged;
    int iteration = 1;

    LogFileController logger = new LogFileController(fileName:"PolicyIteration", grid);

    //Display initial grid with State Type, Reward, and Initial Policy
    System.out.println(x:"Grid World Original :");
    grid.print();
    logger.add(grid);

    do {
        policyChanged = false;

        // 1. Policy Evaluation
        policyEvaluation(grid, Constants.K);

        // 2. Policy Improvement
        for (int c = 0; c < Constants.NUM_COL; c++) {
            for (int r = 0; r < Constants.NUM_ROW; r++) {
                State currentCell = grid.getCell(new StateCoordinate(c, r));

                // If current cell is a wall, skip
                if (currentCell.getStateType() == StateType.WALL)
                    continue;

                //Check if there is a change in policy after Policy Improvement
                boolean changed = policyImprovement(currentCell, grid);

                if (changed)
                    policyChanged = true;
            }
        }

        iteration++;
        logger.add(grid);
    } while (policyChanged); //Continue until policy converges

    System.out.printf(format:"Total Iterations to Converge : %d\n", iteration-1);
    grid.printExperimentParamters(isValueIteration:false,threshold:0);
    grid.printPolicy();
    grid.printStateUtilities();
    grid.printUtilityGrid();

    logger.finalConvertToCSV();
}
```

Fig 3.4.1 - Policy Iteration

2. **policyEvaluation(GridEnvironment grid, int k)**

Policy evaluation: given a policy πi , calculate Ui = U πi , the utility of each state if πi were to be executed. In this function we perform Bellman update repeatedly for K times.

This function performs K iterations for each state. It calculates and updates the utilities of all states based on the current policy using Bellman Iteration.

```java
/**
 * 1. Policy Evaluation
 * Calculates utilities for a given Policy.
 * @param grid
 * @return
 */
private static void policyEvaluation(GridEnvironment grid, int k) {
    //k = Constant K: number of times Bellman update is executed to get next utility estimate
    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
                State currentCell = grid.getCell(new StateCoordinate(c, r));

                //If current state is a Wall, then skip
                if (currentCell.getStateType() == StateType.WALL)
                    continue;

                //2. Add the expected utilities of neighbouring cells(i.e. UP, LEFT, RIGHT)  based on current Policy
                State[] neighbours = grid.getNeighboursOfCell(currentCell);
                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 futureEstimateUtility = up+left+right;

                //3. Update Utility
                float reward = currentCell.getStateType().getReward();
                //Apply Simplified Belmman equation
                //Utility of a state = currentStateReward + Max Possible Discounted Future Reward
                currentCell.setUtility(reward + Constants.DISCOUNT_FACTOR * futureEstimateUtility);
            }
        }
    }
}
```

Fig 3.4.2 - Policy Evaluation

3. **policyImprovement(State currentCell, GridEnvironment grid)**

Policy improvement: Calculate a new maximum expected utility policy π based on Ui
Iterate through the possible directions (UP,DOWN, LEFT, RIGHT) and store it in the *maxUtilities* array. The action with maximum possible utility is identified in maxUtil (0 = UP, 1= DOWN, 2= LEFT. 3 = RIGHT).

New maximum expected utility is compared against the current utility. If max expected utility is greater than current utility, the policy is updated and the function returns boolean value true, indicating a policy update has occurred. Otherwise, the function returns false, if there is no change in policy (convergence).

```java
/**
 * 2. Policy Improvement
 * @param currentCell
 * @param grid
 * @return True if change in policy (current vs updated)
 */
private static boolean policyImprovement(State currentCell, GridEnvironment grid) {

    //1. Find the maximum possible sub-utility
    double[] maxUtility = new double[StateCoordinate.TOTAL_DIRECTIONS];
    //Calculate utility of possible actions(direction) and get maximum to get best policy
    for (int dir = 0; dir < StateCoordinate.TOTAL_DIRECTIONS; dir++) {
        State[] neighbours = grid.getNeighboursOfCell(currentCell, 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();

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

    int maxUtil = 0;
    for (int x = 1; x < maxUtility.length; x++) {
        if (maxUtility[x] > maxUtility[maxUtil])
            maxUtil = x;
    }

    // 2. Get the Current sub-utility
    State[] neighbours = grid.getNeighboursOfCell(currentCell);
    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. Policy improovement to best (max) possible utility
    if (maxUtility[maxUtil] > currSubUtility) {
        //Update policy to policy with max utility
        currentCell.setPolicy(maxUtil);
        return true;
    } else {
        return false;
    }
}
```

Fig 3.4.3 - Policy Improvement

4. RESULT ANALYSIS

4.1. VALUE ITERATION

Results are obtained using different C values, where C is a constant that defines the maximum allowed error. It helps define the convergence threshold.
Epsilon = C*Rmax

Threshold = Epsilon*(1-DiscountFactor)/DiscountFactor

Intuitively, the number of iterations to converge should decrease as we increase C value. The lower the C value, the more optimal the solution. Here we vary C values as [0.01,0.1,1,10,30,50,80]

Fig 4.1.1 C = 0.01 (Converges in 916 iterations)

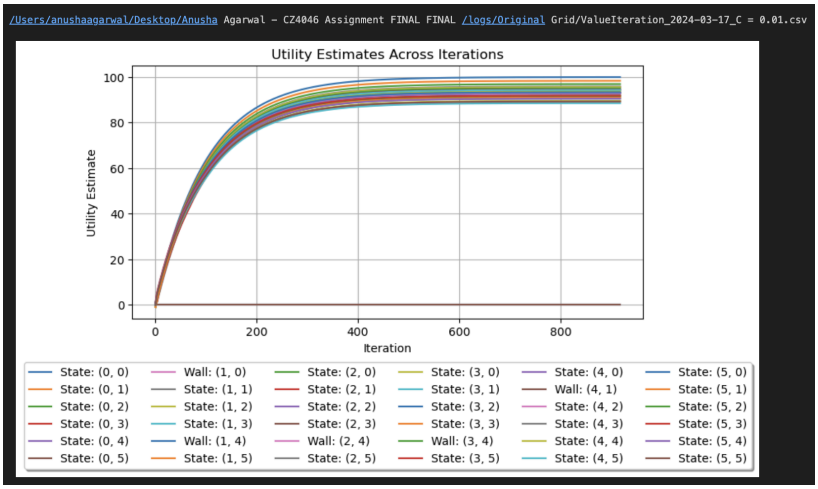
C=0.01

```
Total Iterations to Converge : 916
Change in utility from previous iteration: 0.000

********************
* Experiment Setup *
********************

Discount Factor         :       0.99
Max Reward(Rmax)        :       1.0
Constant 'c'            :       0.01
Epsilon Value(c * Rmax) :       0.01
Utility Upper Bound     :       100.00
Convergence Threshold   :       0.00010
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 99.990
(0, 1): 98.383
(0, 2): 96.939
(0, 3): 95.544
(0, 4): 94.303
(0, 5): 92.928
(1, 0): WALL
(1, 1): 95.873
(1, 2): 95.577
(1, 3): 94.443
(1, 4): WALL
(1, 5): 91.719
(2, 0): 95.036
(2, 1): 94.535
(2, 2): 93.285
(2, 3): 93.223
(2, 4): WALL
(2, 5): 90.526
(3, 0): 93.865
(3, 1): 94.388
(3, 2): 93.167
(3, 3): 91.106
(3, 4): WALL
(3, 5): 89.347
(4, 0): 92.645
(4, 1): WALL
(4, 2): 93.093
(4, 3): 91.805
(4, 4): 89.539
(4, 5): 88.560
(5, 0): 93.319
(5, 1): 90.908
(5, 2): 91.785
(5, 3): 91.879
(5, 4): 90.558
(5, 5): 89.289
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ←    |   ←    |   ←    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
*****************************
* Optimal Policy Utilities *
*****************************

|--------|--------|--------|--------|--------|--------|
| 99.990 |  Wall  | 95.036 | 93.865 | 92.645 | 93.319 |
|--------|--------|--------|--------|--------|--------|
| 98.383 | 95.873 | 94.535 | 94.388 |  Wall  | 90.908 |
|--------|--------|--------|--------|--------|--------|
| 96.939 | 95.577 | 93.285 | 93.167 | 93.093 | 91.785 |
|--------|--------|--------|--------|--------|--------|
| 95.544 | 94.443 | 93.223 | 91.106 | 91.805 | 91.879 |
|--------|--------|--------|--------|--------|--------|
| 94.303 |  Wall  |  Wall  |  Wall  | 89.539 | 90.558 |
|--------|--------|--------|--------|--------|--------|
| 92.928 | 91.719 | 90.526 | 89.347 | 88.560 | 89.289 |
|--------|--------|--------|--------|--------|--------|
```



4.1.2 C = 0.1(Converges in 687 iterations)

C=0.1

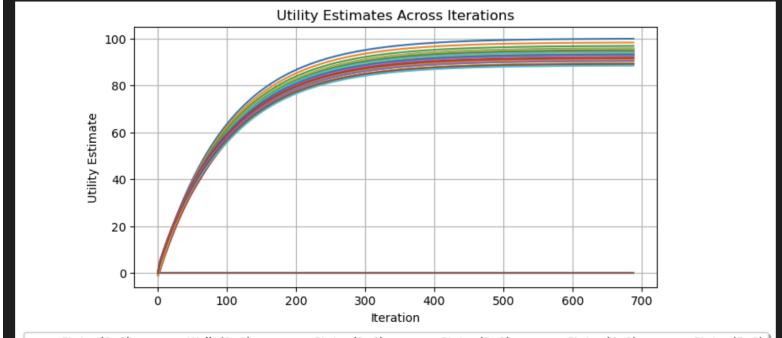
```
Total Iterations to Converge : 687
Change in utility from previous iteration: 0.001

********************
* Experiment Setup *
********************

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

```
**********
* Optimal Policy *
**********

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ←    |   ←    |   ←    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
**********
* Utilities of States:  *
**********

(0, 0): 99.900
(0, 1): 98.295
(0, 2): 96.851
(0, 3): 95.458
(0, 4): 94.217
(0, 5): 92.843
(1, 0): WALL
(1, 1): 95.786
(1, 2): 95.490
(1, 3): 94.357
(1, 4): WALL
(1, 5): 91.636
(2, 0): 94.949
(2, 1): 94.449
(2, 2): 93.199
(2, 3): 93.138
(2, 4): WALL
(2, 5): 90.443
(3, 0): 93.780
(3, 1): 94.303
(3, 2): 93.082
(3, 3): 91.022
(3, 4): WALL
(3, 5): 89.265
(4, 0): 92.560
(4, 1): WALL
(4, 2): 93.010
(4, 3): 91.723
(4, 4): 89.458
(4, 5): 88.479
(5, 0): 93.235
(5, 1): 90.826
(5, 2): 91.703
(5, 3): 91.797
(5, 4): 90.477
(5, 5): 89.209
```

```
**********
* Optimal Policy Utilities *
**********

|--------|--------|--------|--------|--------|--------|
| 99.900 |  Wall  | 94.949 | 93.780 | 92.560 | 93.235 |
|--------|--------|--------|--------|--------|--------|
| 98.295 | 95.786 | 94.449 | 94.303 |  Wall  | 90.826 |
|--------|--------|--------|--------|--------|--------|
| 96.851 | 95.490 | 93.199 | 93.082 | 93.010 | 91.703 |
|--------|--------|--------|--------|--------|--------|
| 95.458 | 94.357 | 93.138 | 91.022 | 91.723 | 91.797 |
|--------|--------|--------|--------|--------|--------|
| 94.217 |  Wall  |  Wall  |  Wall  | 89.458 | 90.477 |
|--------|--------|--------|--------|--------|--------|
| 92.843 | 91.636 | 90.443 | 89.265 | 88.479 | 89.209 |
|--------|--------|--------|--------|--------|--------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal - CZ4046 Assignment FINAL FINAL /logs/Original Grid/ValueIteration_2024-03-17_C = 0.1.csv



4.1.3 C = 1(Converges in 458 iterations)

C = 1

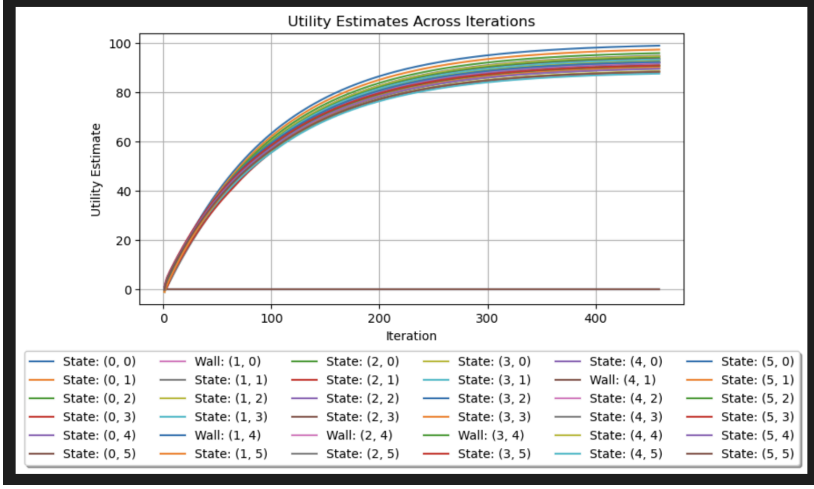
```
Total Iterations to Converge : 458
Change in utility from previous iteration: 0.010

********************
* Experiment Setup *
********************

Discount Factor         :       0.99
Max Reward(Rmax)        :       1.0
Constant 'c'            :       1.0
Epsilon Value(c * Rmax) :       1.0
Utility Upper Bound     :       100.00
Convergence Threshold   :       0.01010
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 99.008
(0, 1): 97.412
(0, 2): 95.979
(0, 3): 94.595
(0, 4): 93.363
(0, 5): 91.999
(1, 0): WALL
(1, 1): 94.913
(1, 2): 94.627
(1, 3): 93.503
(1, 4): WALL
(1, 5): 90.799
(2, 0): 94.088
(2, 1): 93.587
(2, 2): 92.346
(2, 3): 92.293
(2, 4): WALL
(2, 5): 89.615
(3, 0): 92.927
(3, 1): 93.452
(3, 2): 92.241
(3, 3): 90.190
(3, 4): WALL
(3, 5): 88.446
(4, 0): 91.716
(4, 1): WALL
(4, 2): 92.177
(4, 3): 90.900
(4, 4): 88.644
(4, 5): 87.674
(5, 0): 92.399
(5, 1): 89.998
(5, 2): 90.881
(5, 3): 90.983
(5, 4): 89.671
(5, 5): 88.411
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ←    |   ←    |   ←    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
*****************************
* Optimal Policy Utilities *
*****************************

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 99.008 |  Wall  | 94.088 | 92.927 | 91.716 | 92.399 |
|--------|--------|--------|--------|--------|--------|
| 97.412 | 94.913 | 93.587 | 93.452 |  Wall  | 89.998 |
|--------|--------|--------|--------|--------|--------|
| 95.979 | 94.627 | 92.346 | 92.241 | 92.177 | 90.881 |
|--------|--------|--------|--------|--------|--------|
| 94.595 | 93.503 | 92.293 | 90.190 | 90.900 | 90.983 |
|--------|--------|--------|--------|--------|--------|
| 93.363 |  Wall  |  Wall  |  Wall  | 88.644 | 89.671 |
|--------|--------|--------|--------|--------|--------|
| 91.999 | 90.799 | 89.615 | 88.446 | 87.674 | 88.411 |
|--------|--------|--------|--------|--------|--------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal – CZ4046 Assignment FINAL FINAL /logs/Original Grid/ValueIteration_2024-03-17_C = 1.0.csv



4.1.4 C = 10(Converges in 229 iterations)

C = 10

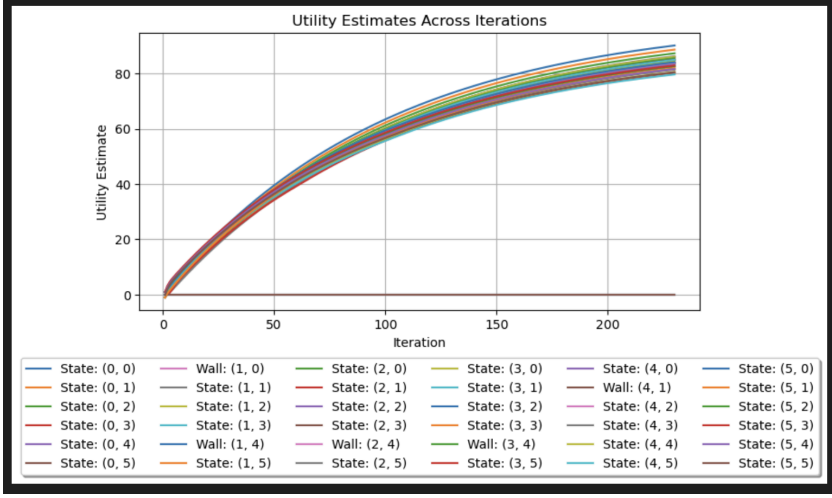
```
Total Iterations to Converge : 229

********************
* Experiment Setup *
********************

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

```
**************************
* Utilities of States:  *
**************************

(0, 0): 90.089
(0, 1): 88.596
(0, 2): 87.262
(0, 3): 85.976
(0, 4): 84.831
(0, 5): 83.562
(1, 0): WALL
(1, 1): 86.197
(1, 2): 86.008
(1, 3): 84.971
(1, 4): WALL
(1, 5): 82.447
(2, 0): 85.480
(2, 1): 84.979
(2, 2): 83.822
(2, 3): 83.847
(2, 4): WALL
(2, 5): 81.346
(3, 0): 84.407
(3, 1): 84.950
(3, 2): 83.832
(3, 3): 81.872
(3, 4): WALL
(3, 5): 80.260
(4, 0): 83.282
(4, 1): WALL
(4, 2): 83.864
(4, 3): 82.679
(4, 4): 80.515
(4, 5): 79.628
(5, 0): 84.049
(5, 1): 81.731
(5, 2): 82.665
(5, 3): 82.849
(5, 4): 81.619
(5, 5): 80.440
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ←    |   ←    |   ←    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
*****************************
* Optimal Policy Utilities *
*****************************

|--------|--------|--------|--------|--------|--------|
| 90.089 |  Wall  | 85.480 | 84.407 | 83.282 | 84.049 |
|--------|--------|--------|--------|--------|--------|
| 88.596 | 86.197 | 84.979 | 84.950 |  Wall  | 81.731 |
|--------|--------|--------|--------|--------|--------|
| 87.262 | 86.008 | 83.822 | 83.832 | 83.864 | 82.665 |
|--------|--------|--------|--------|--------|--------|
| 85.976 | 84.971 | 83.847 | 81.872 | 82.679 | 82.849 |
|--------|--------|--------|--------|--------|--------|
| 84.831 |  Wall  |  Wall  |  Wall  | 80.515 | 81.619 |
|--------|--------|--------|--------|--------|--------|
| 83.562 | 82.447 | 81.346 | 80.260 | 79.628 | 80.440 |
|--------|--------|--------|--------|--------|--------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal – CZ4046 Assignment FINAL FINAL /logs/Original Grid/ValueIteration_2024-03-17_C = 10.0.csv



4.1.5 C = 30(Converges in 119 iterations)

C=30

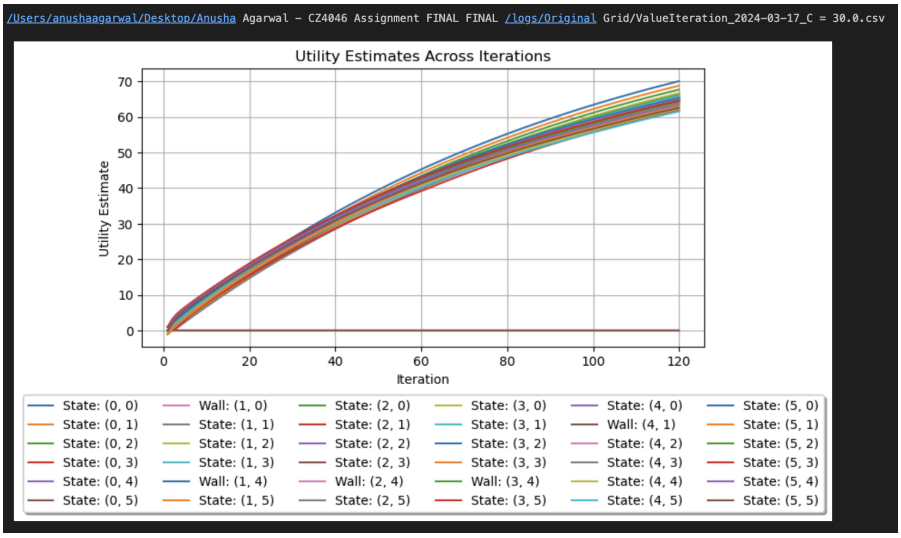
```
Total Iterations to Converge : 119

********************
* Experiment Setup *
********************

Discount Factor         :       0.99
Max Reward(Rmax)        :       1.0
Constant 'c'            :       30.0
Epsilon Value(c * Rmax) :       30.0
Utility Upper Bound     :       100.00
Convergence Threshold   :       0.30303
```

```
**********
* Optimal Policy *
**********

|---------|---------|---------|---------|---------|---------|
|    ↑    |  Wall   |    ←    |    ←    |    ←    |    ↑    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |    ←    |    ←    |    ←    |  Wall   |    ↑    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |    ←    |    ←    |    ↑    |    ←    |    ←    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |    ←    |    ←    |    ↑    |    ↑    |    ↑    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |  Wall   |  Wall   |  Wall   |    ↑    |    ↑    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |    ←    |    ←    |    ←    |    ↑    |    ↑    |
|---------|---------|---------|---------|---------|---------|
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 70.062
(0, 1): 68.796
(0, 2): 67.688
(0, 3): 66.622
(0, 4): 65.670
(0, 5): 64.617
(1, 0): WALL
(1, 1): 66.623
(1, 2): 66.651
(1, 3): 65.810
(1, 4): WALL
(1, 5): 63.691
(2, 0): 66.170
(2, 1): 65.651
(2, 2): 64.682
(2, 3): 64.880
(2, 4): WALL
(2, 5): 62.778
(3, 0): 65.294
(3, 1): 65.862
(3, 2): 64.953
(3, 3): 63.196
(3, 4): WALL
(3, 5): 61.877
(4, 0): 64.360
(4, 1): WALL
(4, 2): 65.200
(4, 3): 64.222
(4, 4): 62.265
(4, 5): 61.565
(5, 0): 65.364
(5, 1): 63.234
(5, 2): 64.226
(5, 3): 64.592
(5, 4): 63.546
(5, 5): 62.549
```

```
**************************
* Optimal Policy Utilities *
**************************

|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 70.062  |  Wall   | 66.170  | 65.294  | 64.360  | 65.364  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 68.796  | 66.623  | 65.651  | 65.862  |  Wall   | 63.234  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 67.688  | 66.651  | 64.682  | 64.953  | 65.200  | 64.226  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 66.622  | 65.810  | 64.880  | 63.196  | 64.222  | 64.592  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 65.670  |  Wall   |  Wall   |  Wall   | 62.265  | 63.546  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
|         |         |         |         |         |         |
| 64.617  | 63.691  | 62.778  | 61.877  | 61.565  | 62.549  |
|         |         |         |         |         |         |
|---------|---------|---------|---------|---------|---------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal – CZ4046 Assignment FINAL FINAL /logs/Original Grid/ValueIteration_2024-03-17_C = 30.0.csv



4.1.6 C = 50(Converges in 68 iterations)

C=50

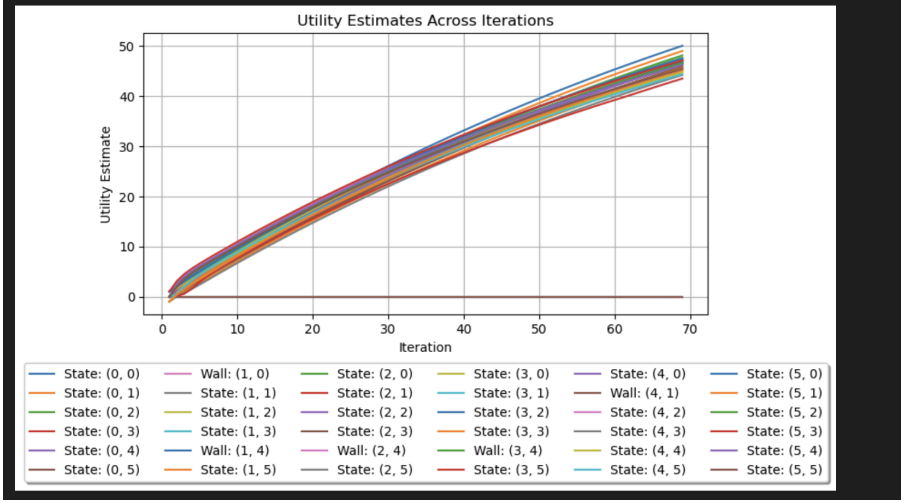
```
Total Iterations to Converge : 68

********************
* Experiment Setup *
********************

Discount Factor         :       0.99
Max Reward(Rmax)        :       1.0
Constant 'c'            :       50.0
Epsilon Value(c * Rmax) :       50.0
Utility Upper Bound     :       100.00
Convergence Threshold   :       0.50505
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 50.016
(0, 1): 48.979
(0, 2): 48.096
(0, 3): 47.251
(0, 4): 46.492
(0, 5): 45.655
(1, 0): WALL
(1, 1): 47.031
(1, 2): 47.277
(1, 3): 46.632
(1, 4): WALL
(1, 5): 44.919
(2, 0): 47.612
(2, 1): 46.979
(2, 2): 45.591
(2, 3): 45.903
(2, 4): WALL
(2, 5): 44.193
(3, 0): 46.923
(3, 1): 47.468
(3, 2): 46.703
(3, 3): 45.091
(3, 4): WALL
(3, 5): 43.478
(4, 0): 46.279
(4, 1): WALL
(4, 2): 47.269
(4, 3): 46.478
(4, 4): 44.727
(4, 5): 44.351
(5, 0): 47.483
(5, 1): 45.533
(5, 2): 46.522
(5, 3): 47.076
(5, 4): 46.203
(5, 5): 45.396
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ←    |   ←    |   →    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ↑    |   ↑    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   →    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |   →    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
****************************
* Optimal Policy Utilities *
****************************

|--------|--------|--------|--------|--------|--------|
| 50.016 |  Wall  | 47.612 | 46.923 | 46.279 | 47.483 |
|--------|--------|--------|--------|--------|--------|
| 48.979 | 47.031 | 46.979 | 47.468 |  Wall  | 45.533 |
|--------|--------|--------|--------|--------|--------|
| 48.096 | 47.277 | 45.591 | 46.703 | 47.269 | 46.522 |
|--------|--------|--------|--------|--------|--------|
| 47.251 | 46.632 | 45.903 | 45.091 | 46.478 | 47.076 |
|--------|--------|--------|--------|--------|--------|
| 46.492 |  Wall  |  Wall  |  Wall  | 44.727 | 46.203 |
|--------|--------|--------|--------|--------|--------|
| 45.655 | 44.919 | 44.193 | 43.478 | 44.351 | 45.396 |
|--------|--------|--------|--------|--------|--------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal - CZ4046 Assignment FINAL FINAL /logs/Original Grid/ValueIteration_2024-03-17_C = 50.0.csv



4.1.7 C = 80((Converges in 22 iterations))

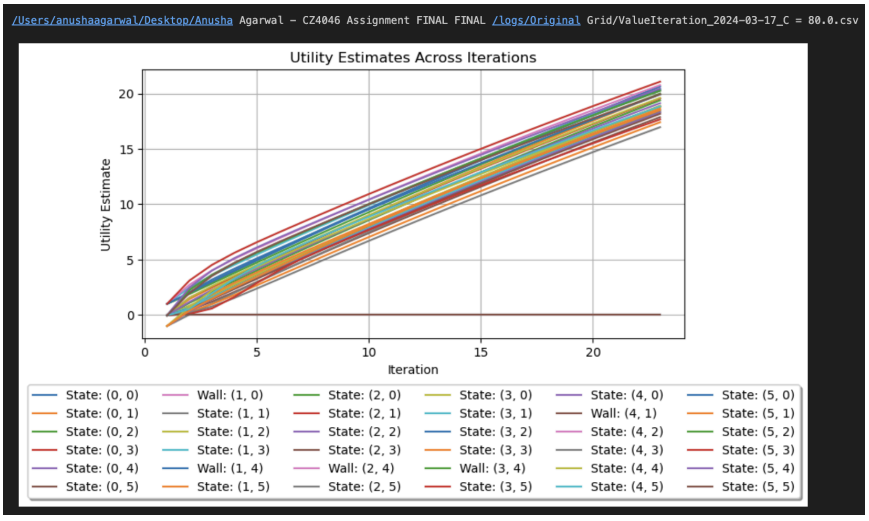
C=80

```
Total Iterations to Converge : 22

********************
* Experiment Setup *
********************

Discount Factor         :       0.99
Max Reward(Rmax)        :       1.0
Constant 'c'            :       80.0
Epsilon Value(c * Rmax) :       80.0
Utility Upper Bound     :       100.00
Convergence Threshold   :       0.80808
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 20.638
(0, 1): 19.937
(0, 2): 19.384
(0, 3): 18.861
(0, 4): 18.386
(0, 5): 17.864
(1, 0): WALL
(1, 1): 18.319
(1, 2): 18.884
(1, 3): 18.525
(1, 4): WALL
(1, 5): 17.406
(2, 0): 19.992
(2, 1): 19.553
(2, 2): 18.253
(2, 3): 18.162
(2, 4): WALL
(2, 5): 16.956
(3, 0): 19.605
(3, 1): 20.409
(3, 2): 19.912
(3, 3): 18.500
(3, 4): WALL
(3, 5): 17.663
(4, 0): 19.125
(4, 1): WALL
(4, 2): 20.753
(4, 3): 20.241
(4, 4): 18.772
(4, 5): 18.831
(5, 0): 20.324
(5, 1): 18.637
(5, 2): 20.255
(5, 3): 21.067
(5, 4): 20.458
(5, 5): 19.929
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ↑    |   ←    |   ←    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ↑    |   ↑    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ↑    |   ↑    |   ↑    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   →    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   →    |   →    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
*****************************
* Optimal Policy Utilities *
*****************************

|--------|--------|--------|--------|--------|--------|
| 20.638 |  Wall  | 19.992 | 19.605 | 19.125 | 20.324 |
|--------|--------|--------|--------|--------|--------|
| 19.937 | 18.319 | 19.553 | 20.409 |  Wall  | 18.637 |
|--------|--------|--------|--------|--------|--------|
| 19.384 | 18.884 | 18.253 | 19.912 | 20.753 | 20.255 |
|--------|--------|--------|--------|--------|--------|
| 18.861 | 18.525 | 18.162 | 18.500 | 20.241 | 21.067 |
|--------|--------|--------|--------|--------|--------|
| 18.386 |  Wall  |  Wall  |  Wall  | 18.772 | 20.458 |
|--------|--------|--------|--------|--------|--------|
| 17.864 | 17.406 | 16.956 | 17.663 | 18.831 | 19.929 |
|--------|--------|--------|--------|--------|--------|
```



Therefore we see, as estimated, number of iterations decrease as we increase C value, however it affects the accuracy of the policy as in C=50,C=80. Therefore can use C = 1 for the most optimal solution, which takes 458 iterations, almost 1/2th of the number of iterations taken when C=0.01. We can keep a smaller C as well since our maze is small but it leads to wastage of system computations. We can go as high as C=30 (converges in 119 iterations) , however, C=1 seems like a safe parameter.

4.2. POLICY ITERATION

Results are obtained by varying K which indicates the number of times Bellman update is performed in Policy Evaluation Step. Here we vary K=[40,20,10,1]

4.2.1. K=40(Converges in 9 iterations)

K = 40

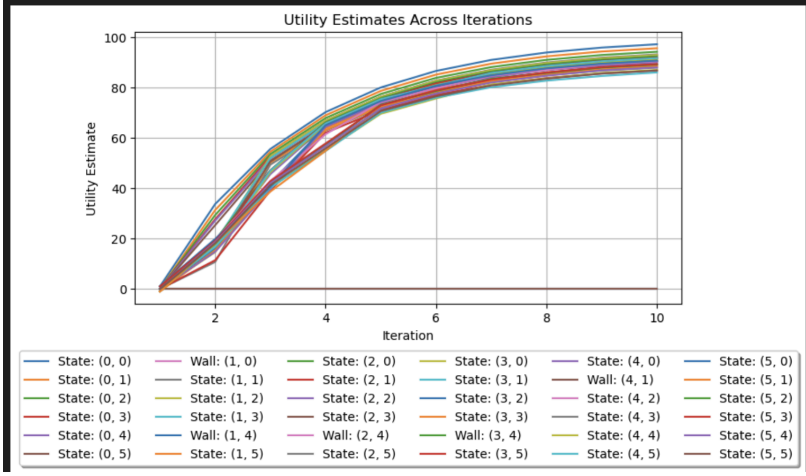
```
Total Iterations to Converge : 9

********************
* Experiment Setup *
********************

Discount        :       0.99
k               :       40
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 97.343
(0, 1): 95.767
(0, 2): 94.352
(0, 3): 92.986
(0, 4): 91.771
(0, 5): 90.424
(1, 0): WALL
(1, 1): 93.286
(1, 2): 93.019
(1, 3): 91.911
(1, 4): WALL
(1, 5): 89.241
(2, 0): 92.481
(2, 1): 91.981
(2, 2): 90.755
(2, 3): 90.716
(2, 4): WALL
(2, 5): 88.072
(3, 0): 91.337
(3, 1): 91.865
(3, 2): 90.671
(3, 3): 88.637
(3, 4): WALL
(3, 5): 86.918
(4, 0): 90.142
(4, 1): WALL
(4, 2): 90.626
(4, 3): 89.365
(4, 4): 87.127
(4, 5): 86.172
(5, 0): 90.841
(5, 1): 88.455
(5, 2): 89.347
(5, 3): 89.465
(5, 4): 88.168
(5, 5): 86.923
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ←    |   ←    |   ←    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
****************************
* Optimal Policy Utilities *
****************************

|--------|--------|--------|--------|--------|--------|
| 97.343 |  Wall  | 92.481 | 91.337 | 90.142 | 90.841 |
|--------|--------|--------|--------|--------|--------|
| 95.767 | 93.286 | 91.981 | 91.865 |  Wall  | 88.455 |
|--------|--------|--------|--------|--------|--------|
| 94.352 | 93.019 | 90.755 | 90.671 | 90.626 | 89.347 |
|--------|--------|--------|--------|--------|--------|
| 92.986 | 91.911 | 90.716 | 88.637 | 89.365 | 89.465 |
|--------|--------|--------|--------|--------|--------|
| 91.771 |  Wall  |  Wall  |  Wall  | 87.127 | 88.168 |
|--------|--------|--------|--------|--------|--------|
| 90.424 | 89.241 | 88.072 | 86.918 | 86.172 | 86.923 |
|--------|--------|--------|--------|--------|--------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal – CZ4046 Assignment FINAL FINAL /logs/PolicyIteration/Original Grid/PolicyIteration_2024-03-17_K = 40.csv



4.2.2. K=20(Converges in 16 iterations)

K = 20

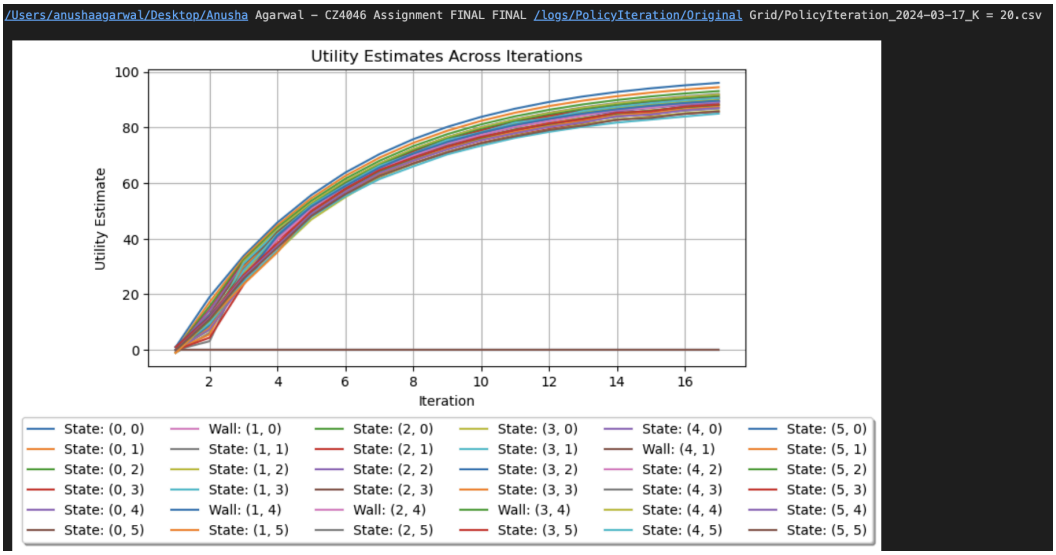
```
Total Iterations to Converge : 16

********************
* Experiment Setup *
********************

Discount        :       0.99
k               :       20
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 96.029
(0, 1): 94.467
(0, 2): 93.067
(0, 3): 91.716
(0, 4): 90.513
(0, 5): 89.181
(1, 0): WALL
(1, 1): 92.002
(1, 2): 91.748
(1, 3): 90.653
(1, 4): WALL
(1, 5): 88.010
(2, 0): 91.212
(2, 1): 90.712
(2, 2): 89.499
(2, 3): 89.472
(2, 4): WALL
(2, 5): 86.853
(3, 0): 90.081
(3, 1): 90.612
(3, 2): 89.432
(3, 3): 87.411
(3, 4): WALL
(3, 5): 85.711
(4, 0): 88.899
(4, 1): WALL
(4, 2): 89.401
(4, 3): 88.154
(4, 4): 85.929
(4, 5): 84.986
(5, 0): 89.610
(5, 1): 87.237
(5, 2): 88.137
(5, 3): 88.266
(5, 4): 86.981
(5, 5): 85.748
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ←    |   ←    |   ←    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
****************************
* Optimal Policy Utilities *
****************************

|--------|--------|--------|--------|--------|--------|
| 96.029 |  Wall  | 91.212 | 90.081 | 88.899 | 89.610 |
|--------|--------|--------|--------|--------|--------|
| 94.467 | 92.002 | 90.712 | 90.612 |  Wall  | 87.237 |
|--------|--------|--------|--------|--------|--------|
| 93.067 | 91.748 | 89.499 | 89.432 | 89.401 | 88.137 |
|--------|--------|--------|--------|--------|--------|
| 91.716 | 90.653 | 89.472 | 87.411 | 88.154 | 88.266 |
|--------|--------|--------|--------|--------|--------|
| 90.513 |  Wall  |  Wall  |  Wall  | 85.929 | 86.981 |
|--------|--------|--------|--------|--------|--------|
| 89.181 | 88.010 | 86.853 | 85.711 | 84.986 | 85.748 |
|--------|--------|--------|--------|--------|--------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal – CZ4046 Assignment FINAL FINAL /logs/PolicyIteration/Original Grid/PolicyIteration_2024-03-17_K = 20.csv



4.2.3. K=10(Converges in 27 iterations)

K = 10

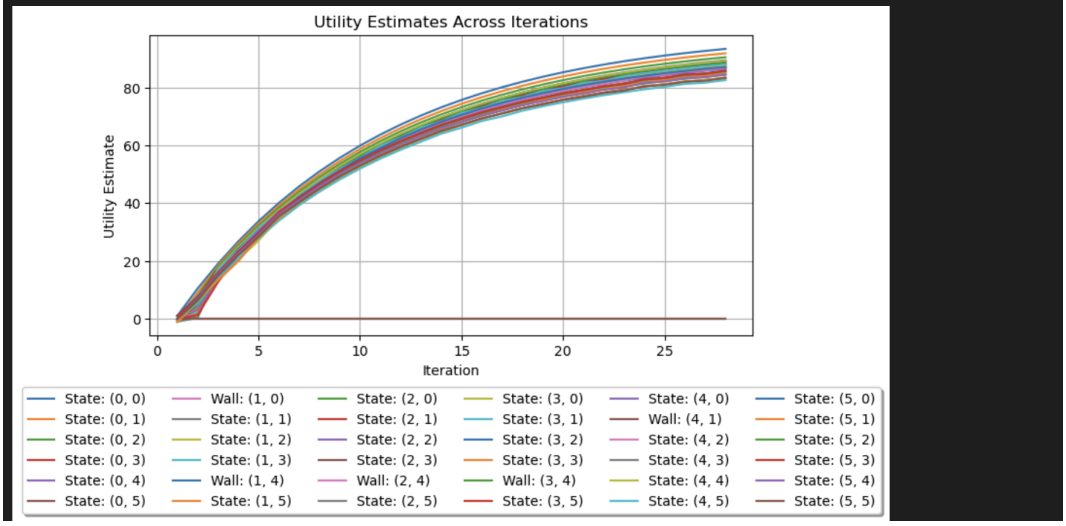
```
Total Iterations to Converge : 27

********************
* Experiment Setup *
********************

Discount       :       0.99
k              :       10
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 93.436
(0, 1): 91.904
(0, 2): 90.533
(0, 3): 89.211
(0, 4): 88.033
(0, 5): 86.728
(1, 0): WALL
(1, 1): 89.468
(1, 2): 89.243
(1, 3): 88.173
(1, 4): WALL
(1, 5): 85.582
(2, 0): 88.710
(2, 1): 88.210
(2, 2): 87.021
(2, 3): 87.016
(2, 4): WALL
(2, 5): 84.450
(3, 0): 87.605
(3, 1): 88.140
(3, 2): 86.988
(3, 3): 84.993
(3, 4): WALL
(3, 5): 83.332
(4, 0): 86.447
(4, 1): WALL
(4, 2): 86.984
(4, 3): 85.764
(4, 4): 83.566
(4, 5): 82.648
(5, 0): 87.183
(5, 1): 84.834
(5, 2): 85.748
(5, 3): 85.901
(5, 4): 84.640
(5, 5): 83.431
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ←    |   ←    |   ←    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ↑    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |  Wall  |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |   ←    |   ←    |   ←    |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
****************************
* Optimal Policy Utilities *
****************************

|--------|--------|--------|--------|--------|--------|
| 93.436 |  Wall  | 88.710 | 87.605 | 86.447 | 87.183 |
|--------|--------|--------|--------|--------|--------|
| 91.904 | 89.468 | 88.210 | 88.140 |  Wall  | 84.834 |
|--------|--------|--------|--------|--------|--------|
| 90.533 | 89.243 | 87.021 | 86.988 | 86.984 | 85.748 |
|--------|--------|--------|--------|--------|--------|
| 89.211 | 88.173 | 87.016 | 84.993 | 85.764 | 85.901 |
|--------|--------|--------|--------|--------|--------|
| 88.033 |  Wall  |  Wall  |  Wall  | 83.566 | 84.640 |
|--------|--------|--------|--------|--------|--------|
| 86.728 | 85.582 | 84.450 | 83.332 | 82.648 | 83.431 |
|--------|--------|--------|--------|--------|--------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal - CZ4046 Assignment FINAL FINAL /logs/PolicyIteration/Original Grid/PolicyIteration_2024-03-17_K = 10.csv



4.2.4. K=1(Converges in 39 iterations)

K = 1

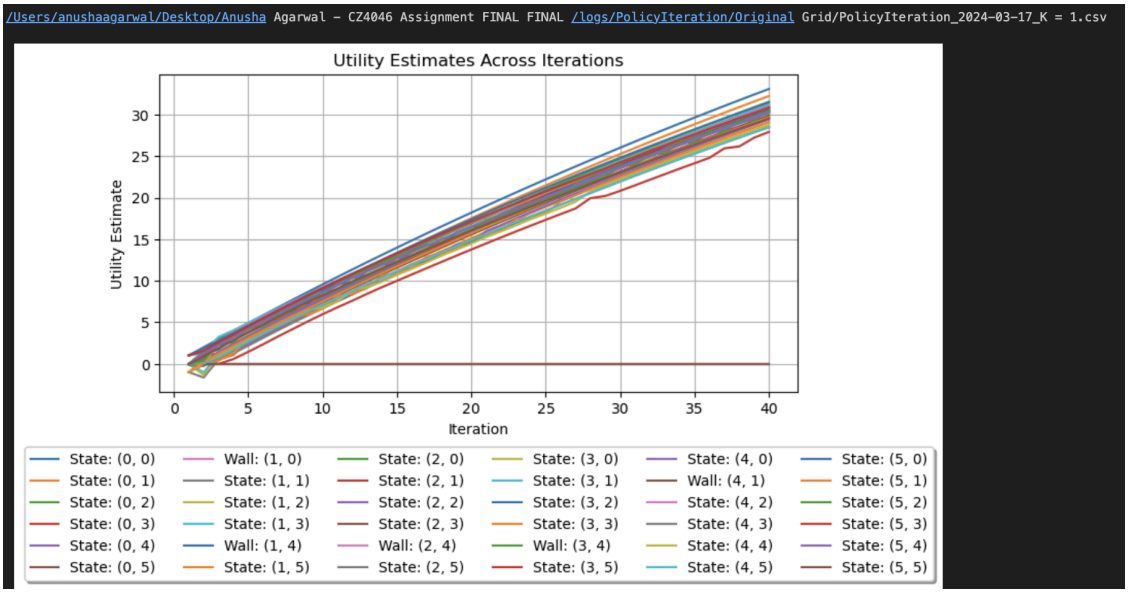
```
Total Iterations to Converge : 39

********************
* Experiment Setup *
********************

Discount        :       0.99
k               :       1
```

```
*************************
* Utilities of States:  *
*************************

(0, 0): 33.102
(0, 1): 32.259
(0, 2): 31.566
(0, 3): 30.906
(0, 4): 30.311
(0, 5): 29.655
(1, 0): WALL
(1, 1): 30.500
(1, 2): 30.930
(1, 3): 30.450
(1, 4): WALL
(1, 5): 29.079
(2, 0): 31.178
(2, 1): 30.696
(2, 2): 29.431
(2, 3): 29.885
(2, 4): WALL
(2, 5): 28.512
(3, 0): 30.647
(3, 1): 31.184
(3, 2): 30.579
(3, 3): 29.125
(3, 4): WALL
(3, 5): 27.933
(4, 0): 30.256
(4, 1): WALL
(4, 2): 30.989
(4, 3): 30.378
(4, 4): 28.772
(4, 5): 28.471
(5, 0): 31.480
(5, 1): 29.686
(5, 2): 30.433
(5, 3): 30.867
(5, 4): 30.180
(5, 5): 29.544
```

```
******************
* Optimal Policy *
******************

|---------|---------|---------|---------|---------|---------|
|    ↑    |  Wall   |    ←    |    ←    |    →    |    ↑    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |    ←    |    ↑    |    →    |  Wall   |    ↑    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |    ←    |    ←    |    ↑    |    ↑    |    ←    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |    ←    |    ←    |    ↑    |    ↑    |    →    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |  Wall   |  Wall   |  Wall   |    ↑    |    ↑    |
|---------|---------|---------|---------|---------|---------|
|    ↑    |    ←    |    ←    |    ←    |    →    |    ↑    |
|---------|---------|---------|---------|---------|---------|
```

```
*****************************
* Optimal Policy Utilities *
*****************************

|---------|---------|---------|---------|---------|---------|
| 33.102  |  Wall   | 31.178  | 30.647  | 30.256  | 31.480  |
|---------|---------|---------|---------|---------|---------|
| 32.259  | 30.500  | 30.696  | 31.184  |  Wall   | 29.686  |
|---------|---------|---------|---------|---------|---------|
| 31.566  | 30.930  | 29.431  | 30.579  | 30.989  | 30.433  |
|---------|---------|---------|---------|---------|---------|
| 30.906  | 30.450  | 29.885  | 29.125  | 30.378  | 30.867  |
|---------|---------|---------|---------|---------|---------|
| 30.311  |  Wall   |  Wall   |  Wall   | 28.772  | 30.180  |
|---------|---------|---------|---------|---------|---------|
| 29.655  | 29.079  | 28.512  | 27.933  | 28.471  | 29.544  |
|---------|---------|---------|---------|---------|---------|
```

/Users/anushaagarwal/Desktop/Anusha Agarwal – CZ4046 Assignment FINAL FINAL /logs/PolicyIteration/Original Grid/PolicyIteration_2024-03-17_K = 1.csv



Here we see, K=1 accuracy sufferers.As K increases, number of bellman updates increase, thereby increasing policy evaluation and state expected utilities, which is why the Policy Iteration algorithm converges in less iterations. Therefore we keep K=20 for optimal solution.

As seen in this experiment, Policy Iteration has a better performance than value iteration.

5. BONUS QUESTION -

5.1. COMPLICATED GRID SETUP:

In this section, two complicated grid worlds are set up to analyze the performance of value iteration and policy iteration on different maze environments. Difference in number of iteration, optimal utilities, policies are explored. Take C=1 and K=40.

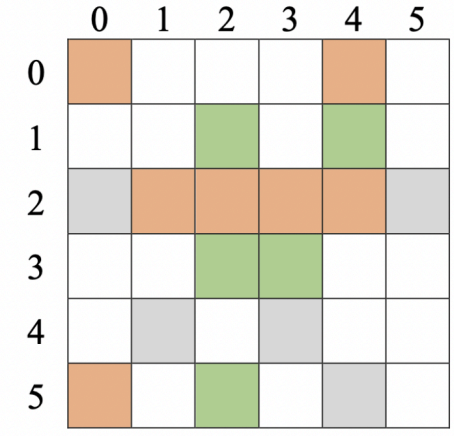


Fig 5.1.1

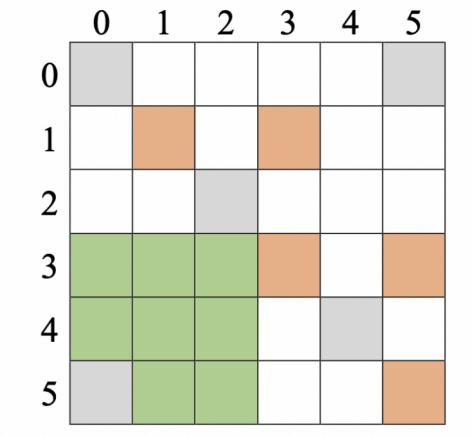


Fig 5.1.2

Complicated Grid Design 1 is created by increasing the BROWN states(-1.0 reward) and a line of all BROWN states and WALLs. Without modifying the epsilon values, Complicated Maze Design 1 converged at 360 iteration when using Value Iteration (Original Grid converged in 358) and 10 iteration when using Policy iteration(Original Grid converged in 9). This suggests that increasing the number of brown states complicates maze a bit but not much affect.

Complicated Grid Design 2 is created by surrounding the GREEN States with either a wall or brown states.

Complicated Maze Design 2 converged at 270 iterations when using Value Iteration (Original Grid converged in 358) and 12 iteration when using Policy iteration(Original Grid converged in 9). This suggests that tweaking in such a way also does not necessarily complicate/increase convergence iterations.

5.1.1. Complicated Grid Design 1 -

Value Iteration - (Converges in 360)

Complicated Maze 1 :
C = 1

```
Total Iterations to Converge : 360

********************
* Experiment Setup *
********************

Discount Factor         :       0.99
Max Reward(Rmax)        :       1.0
Constant 'c'            :       1.0
Epsilon Value(c * Rmax) :       1.0
Utility Upper Bound     :       100.00
Convergence Threshold   :       0.01010
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): 75.484
(0, 1): 77.235
(0, 2): WALL
(0, 3): 79.507
(0, 4): 78.463
(0, 5): 77.716
(1, 0): 77.494
(1, 1): 78.490
(1, 2): 78.573
(1, 3): 80.704
(1, 4): WALL
(1, 5): 79.878
(2, 0): 78.411
(2, 1): 79.657
(2, 2): 79.657
(2, 3): 82.051
(2, 4): 80.976
(2, 5): 80.949
(3, 0): 77.768
(3, 1): 78.862
(3, 2): 79.822
(3, 3): 82.270
(3, 4): WALL
(3, 5): 79.887
(4, 0): 76.721
(4, 1): 78.841
(4, 2): 78.664
(4, 3): 80.786
(4, 4): 79.612
(4, 5): WALL
(5, 0): 76.659
(5, 1): 77.681
(5, 2): WALL
(5, 3): 79.612
(5, 4): 78.683
(5, 5): 77.649
```

```
*****************
* Optimal Policy *
*****************

|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   ↓    |   ↓    |   ↓    |   ↓    |
|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   ↓    |   ↓    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|  Wall  |   ↓    |   ↓    |   ↓    |   ↓    |  Wall  |
|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   →    |   ↓    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ↑    |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   ↑    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
*****************************
* Optimal Policy Utilities *
*****************************

|--------|--------|--------|--------|--------|--------|
| 75.484 | 77.494 | 78.411 | 77.768 | 76.721 | 76.659 |
|--------|--------|--------|--------|--------|--------|
| 77.235 | 78.490 | 79.657 | 78.862 | 78.841 | 77.681 |
|--------|--------|--------|--------|--------|--------|
|  Wall  | 78.573 | 79.657 | 79.822 | 78.664 |  Wall  |
|--------|--------|--------|--------|--------|--------|
| 79.507 | 80.704 | 82.051 | 82.270 | 80.786 | 79.612 |
|--------|--------|--------|--------|--------|--------|
| 78.463 |  Wall  | 80.976 |  Wall  | 79.612 | 78.683 |
|--------|--------|--------|--------|--------|--------|
| 77.716 | 79.878 | 80.949 | 79.887 |  Wall  | 77.649 |
|--------|--------|--------|--------|--------|--------|
```

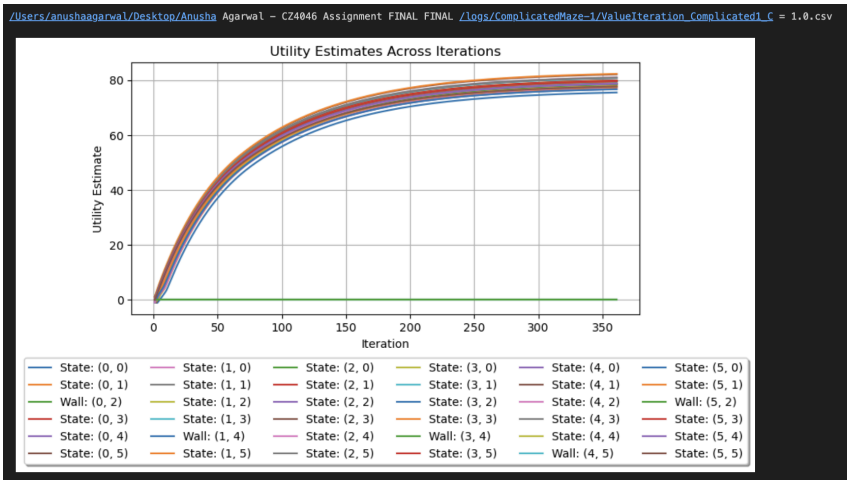


Fig 5.1.1.1

Policy Iteration - (Converges in 10)

Complicated Maze 1 :
K = 40

```
Total Iterations to Converge : 10

**********
* Experiment Setup *
**********

Discount       :       0.99
k              :       40
```

```
**********
* Utilities of States:  *
**********

(0, 0): 74.230
(0, 1): 75.984
(0, 2): WALL
(0, 3): 78.260
(0, 4): 77.228
(0, 5): 76.493
(1, 0): 76.243
(1, 1): 77.242
(1, 2): 77.325
(1, 3): 79.460
(1, 4): WALL
(1, 5): 78.658
(2, 0): 77.163
(2, 1): 78.411
(2, 2): 78.413
(2, 3): 80.809
(2, 4): 79.745
(2, 5): 79.732
(3, 0): 76.524
(3, 1): 77.619
(3, 2): 78.580
(3, 3): 81.028
(3, 4): WALL
(3, 5): 78.682
(4, 0): 75.490
(4, 1): 77.611
(4, 2): 77.433
(4, 3): 79.559
(4, 4): 78.398
(4, 5): WALL
(5, 0): 75.439
(5, 1): 76.464
(5, 2): WALL
(5, 3): 78.398
(5, 4): 77.480
(5, 5): 76.458
```

```
**********
* Optimal Policy *
**********

|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   ↓    |   ↓    |   ↓    |   ↓    |
|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   ↓    |   ↓    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|  Wall  |   ↓    |   ↓    |   ↓    |   ↓    |  Wall  |
|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   →    |   ↓    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↑    |  Wall  |   ↑    |  Wall  |   ↑    |   ↑    |
|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   ↑    |   ←    |  Wall  |   ↑    |
|--------|--------|--------|--------|--------|--------|
```

```
**********
* Optimal Policy Utilities *
**********

|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 74.230 | 76.243 | 77.163 | 76.524 | 75.490 | 75.439 |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 75.984 | 77.242 | 78.411 | 77.619 | 77.611 | 76.464 |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
|  Wall  | 77.325 | 78.413 | 78.580 | 77.433 |  Wall  |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 78.260 | 79.460 | 80.809 | 81.028 | 79.559 | 78.398 |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 77.228 |  Wall  | 79.745 |  Wall  | 78.398 | 77.480 |
|--------|--------|--------|--------|--------|--------|
|        |        |        |        |        |        |
| 76.493 | 78.658 | 79.732 | 78.682 |  Wall  | 76.458 |
|--------|--------|--------|--------|--------|--------|
```

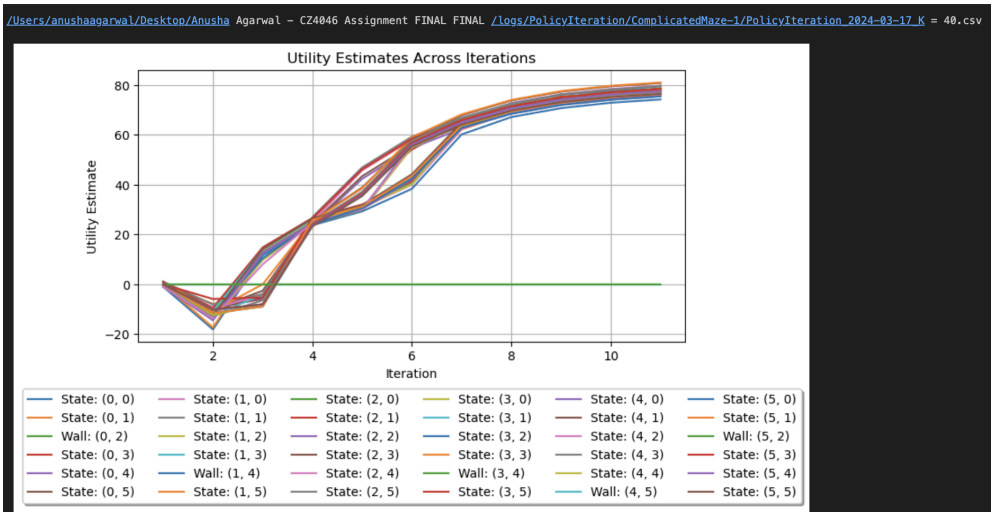


Fig 5.1.1.2

5.1.2. Complicated Grid Design 2 -

Value Iteration - (Converges in 270)

Complicated Maze 2:

C = 1

```
Total Iterations to Converge : 270

********************
* Experiment Setup *
********************

Discount Factor         :       0.99
Max Reward(Rmax)        :       1.0
Constant 'c'            :       1.0
Epsilon Value(c * Rmax) :       1.0
Utility Upper Bound     :       100.00
Convergence Threshold   :       0.01010
```

```
**********
* Optimal Policy *
**********

| Wall | ↓ | ← | ← | ↓ | Wall |
| ↓ | ↓ | ← | ↓ | ↓ | ← |
| ↓ | ↓ | Wall | ↓ | ↓ | ← |
| ↓ | ↓ | ← | ← | ← | ← |
| → | → | ← | ← | Wall | ↓ |
| Wall | → | ← | ← | ← | ← |
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): WALL
(0, 1): 96.786
(0, 2): 98.208
(0, 3): 99.510
(0, 4): 99.515
(0, 5): WALL
(1, 0): 94.305
(1, 1): 95.695
(1, 2): 98.213
(1, 3): 99.515
(1, 4): 99.520
(1, 5): 99.524
(2, 0): 93.213
(2, 1): 94.315
(2, 2): WALL
(2, 3): 99.520
(2, 4): 99.525
(2, 5): 99.529
(3, 0): 92.133
(3, 1): 93.222
(3, 2): 95.567
(3, 3): 96.993
(3, 4): 98.109
(3, 5): 98.223
(4, 0): 91.797
(4, 1): 92.976
(4, 2): 94.315
(4, 3): 95.576
(4, 4): WALL
(4, 5): 96.946
(5, 0): WALL
(5, 1): 91.897
(5, 2): 92.973
(5, 3): 93.117
(5, 4): 93.099
(5, 5): 94.336
```

```
**************************
* Optimal Policy Utilities *
**************************

| Wall | 94.305 | 93.213 | 92.133 | 91.797 | Wall |
| 96.786 | 95.695 | 94.315 | 93.222 | 92.976 | 91.897 |
| 98.208 | 98.213 | Wall | 95.567 | 94.315 | 92.973 |
| 99.510 | 99.515 | 99.520 | 96.993 | 95.576 | 93.117 |
| 99.515 | 99.520 | 99.525 | 98.109 | Wall | 93.099 |
| Wall | 99.524 | 99.529 | 98.223 | 96.946 | 94.336 |
```

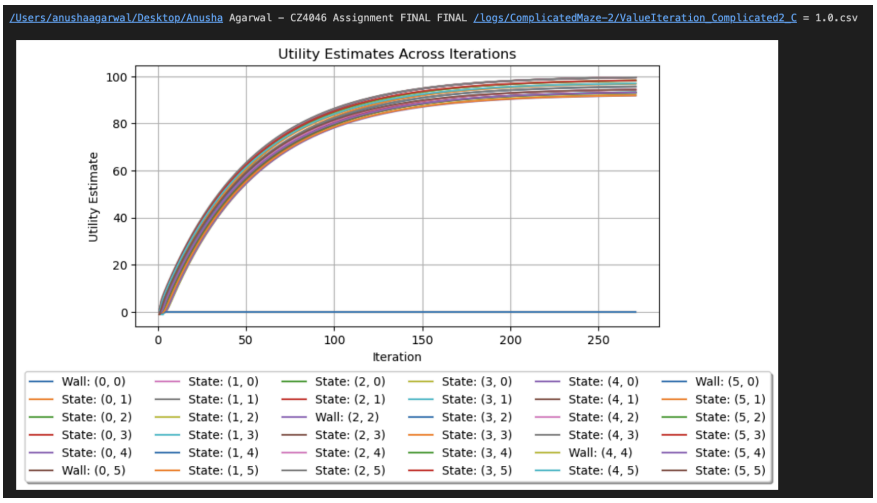


Fig 5.1.2.1

Policy Iteration - (Converges in 12)

Complicated Maze 2:
K=40

```
Total Iterations to Converge : 12

**********************
* Experiment Setup *
**********************

Discount        :       0.99
k               :       40
```

```
**************************
* Utilities of States:  *
**************************

(0, 0): WALL
(0, 1): 97.031
(0, 2): 98.449
(0, 3): 99.749
(0, 4): 99.751
(0, 5): WALL
(1, 0): 94.549
(1, 1): 95.937
(1, 2): 98.452
(1, 3): 99.751
(1, 4): 99.754
(1, 5): 99.755
(2, 0): 93.455
(2, 1): 94.554
(2, 2): WALL
(2, 3): 99.754
(2, 4): 99.756
(2, 5): 99.757
(3, 0): 92.373
(3, 1): 93.459
(3, 2): 95.801
(3, 3): 97.224
(3, 4): 98.338
(3, 5): 98.449
(4, 0): 92.033
(4, 1): 93.209
(4, 2): 94.545
(4, 3): 95.806
(4, 4): WALL
(4, 5): 97.171
(5, 0): WALL
(5, 1): 92.128
(5, 2): 93.201
(5, 3): 93.344
(5, 4): 93.325
(5, 5): 94.559
```

```
******************
* Optimal Policy *
******************

|--------|--------|--------|--------|--------|--------|
|  Wall  |   ↓    |   ←    |   ←    |   ↓    |  Wall  |
|--------|--------|--------|--------|--------|--------|
|   ↓    |   ↓    |   ←    |   ↓    |   ↓    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↓    |   ↓    |  Wall  |   ↓    |   ↓    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   ↓    |   ↓    |   ←    |   ←    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
|   →    |   →    |   ←    |   ←    |  Wall  |   ↓    |
|--------|--------|--------|--------|--------|--------|
|  Wall  |   →    |   ←    |   ←    |   ←    |   ←    |
|--------|--------|--------|--------|--------|--------|
```

```
****************************
* Optimal Policy Utilities *
****************************

|--------|--------|--------|--------|--------|--------|
|  Wall  | 94.549 | 93.455 | 92.373 | 92.033 |  Wall  |
|--------|--------|--------|--------|--------|--------|
| 97.031 | 95.937 | 94.554 | 93.459 | 93.209 | 92.128 |
|--------|--------|--------|--------|--------|--------|
| 98.449 | 98.452 |  Wall  | 95.801 | 94.545 | 93.201 |
|--------|--------|--------|--------|--------|--------|
| 99.749 | 99.751 | 99.754 | 97.224 | 95.806 | 93.344 |
|--------|--------|--------|--------|--------|--------|
| 99.751 | 99.754 | 99.756 | 98.338 |  Wall  | 93.325 |
|--------|--------|--------|--------|--------|--------|
|  Wall  | 99.755 | 99.757 | 98.449 | 97.171 | 94.559 |
|--------|--------|--------|--------|--------|--------|
```

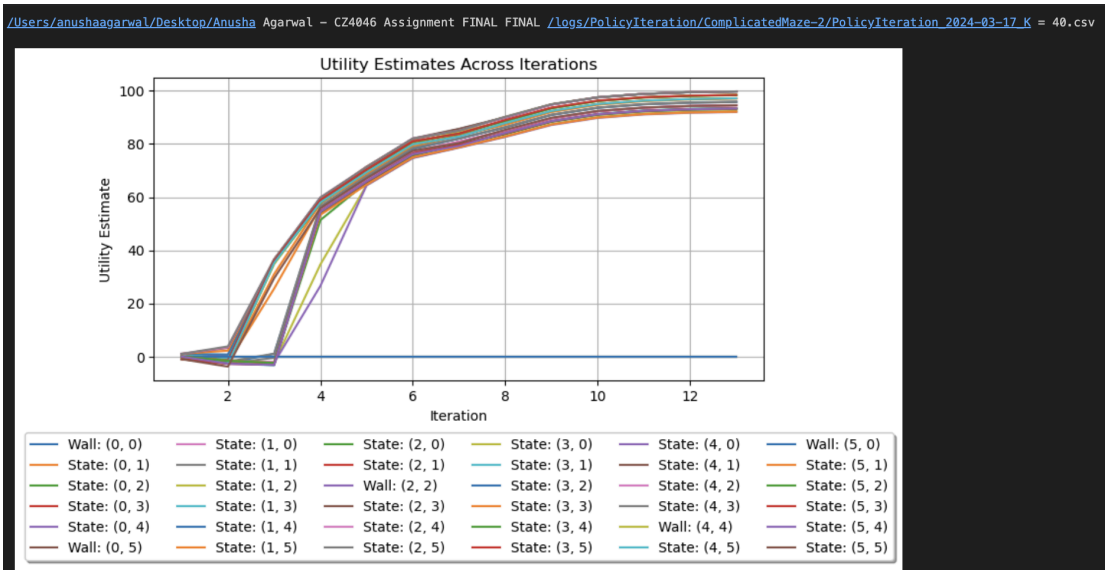
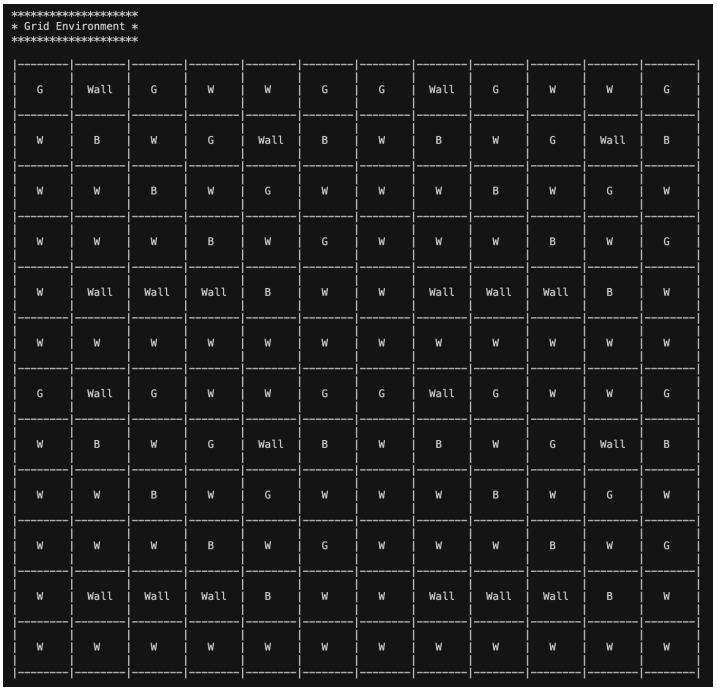


Fig 5.1.2.2

5.2. LARGER GRID SETUP

The original grid is scaled 2 times and 4 times creating a 24x24 grid and we see performance as below, where C is set to 1 and K is set to 40.

5.2.1. Grid Scaled x2



Value Iteration - (Converges in 455)

Original grid scaled x2.
C = 1

```
Total Iterations to Converge : 455

********************
* Experiment Setup *
********************

Discount Factor          :       0.99
Max Reward(Rmax)         :       1.0
Constant 'c'             :       1.0
Epsilon Value(c * Rmax)  :       1.0
Utility Upper Bound      :       100.00
Convergence Threshold    :       0.01010
```

↑	Wall	↑	→	→	→	↑	Wall	←	←	←	↑
↑	←	←	↑	Wall	↑	↑	←	←	←	Wall	↑
↑	←	←	↑	↑	↑	↑	←	←	↑	←	←
↑	←	←	↑	↑	↑	↑	←	←	↑	↑	↑
↑	Wall	Wall	Wall	↑	↑	↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	→	↑	↑	←	←	←	↑	↑
↑	Wall	↑	←	→	↑	↑	Wall	←	←	←	↑
↑	←	↑	↑	Wall	↑	↑	←	↑	↑	Wall	↑
↑	←	↑	↑	↑	↑	↑	←	↑	↑	↑	←
↑	←	←	↑	↑	↑	↑	←	←	↑	↑	→
↑	Wall	Wall	Wall	↑	↑	↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	→	↑	↑	←	←	←	→	↑

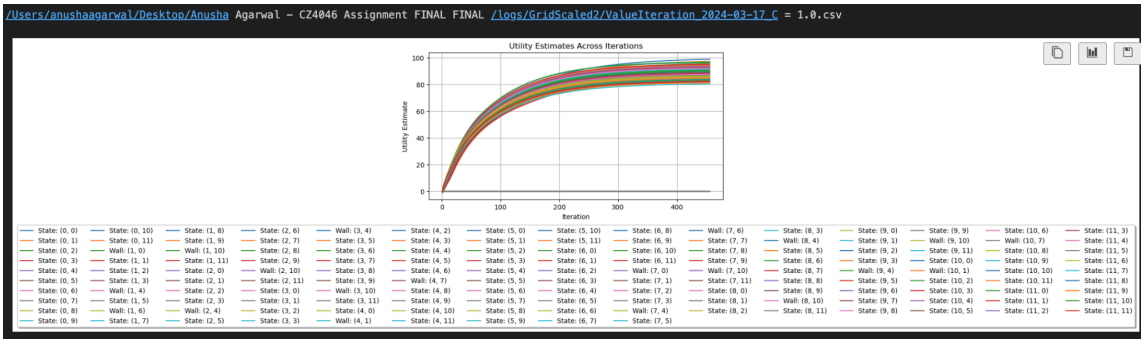


Fig 5.2.1.1

Policy Iteration - (Converges in 12)

```
Total Iterations to Converge : 12

********************
* Experiment Setup *
********************

Discount        :       0.99
k               :       40
```

```
******************
* Optimal Policy *
******************
```

↑	Wall	↑	→	→	→	↑	Wall	←	←	←	↑
↑	←	←	↑	Wall	↑	↑	←	←	←	Wall	↑
↑	←	←	↑	↑	↑	↑	←	←	↑	←	←
↑	←	←	↑	↑	↑	↑	←	←	↑	↑	↑
↑	Wall	Wall	Wall	↑	↑	↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	→	↑	↑	←	←	←	↑	↑
↑	Wall	↑	←	→	↑	↑	Wall	←	←	←	↑
↑	←	↑	↑	Wall	↑	↑	←	↑	↑	Wall	↑
↑	←	↑	↑	↑	↑	↑	←	↑	↑	↑	←
↑	←	←	↑	↑	↑	↑	←	←	↑	↑	→
↑	Wall	Wall	Wall	↑	↑	↑	Wall	Wall	Wall	↑	↑
↑	←	←	←	→	↑	↑	←	←	←	→	↑

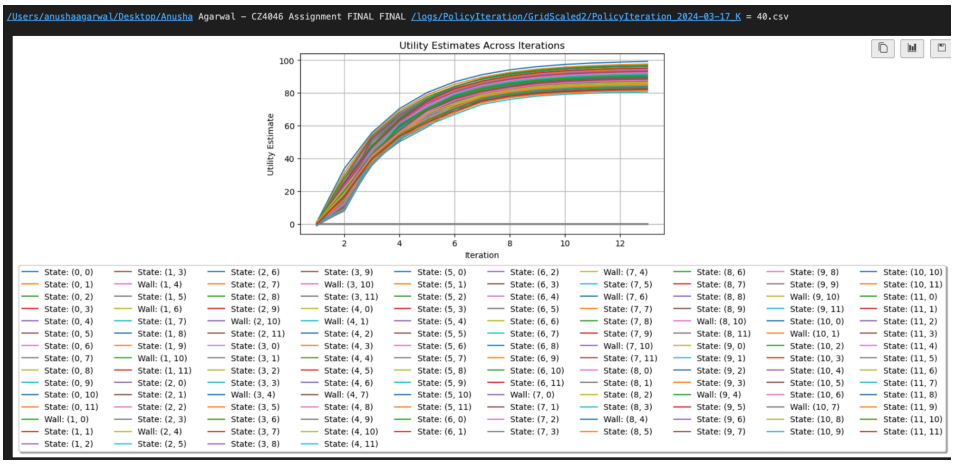


Fig 5.2.1.2

5.2.2. Grid Scaled x4 -

Value Iteration -

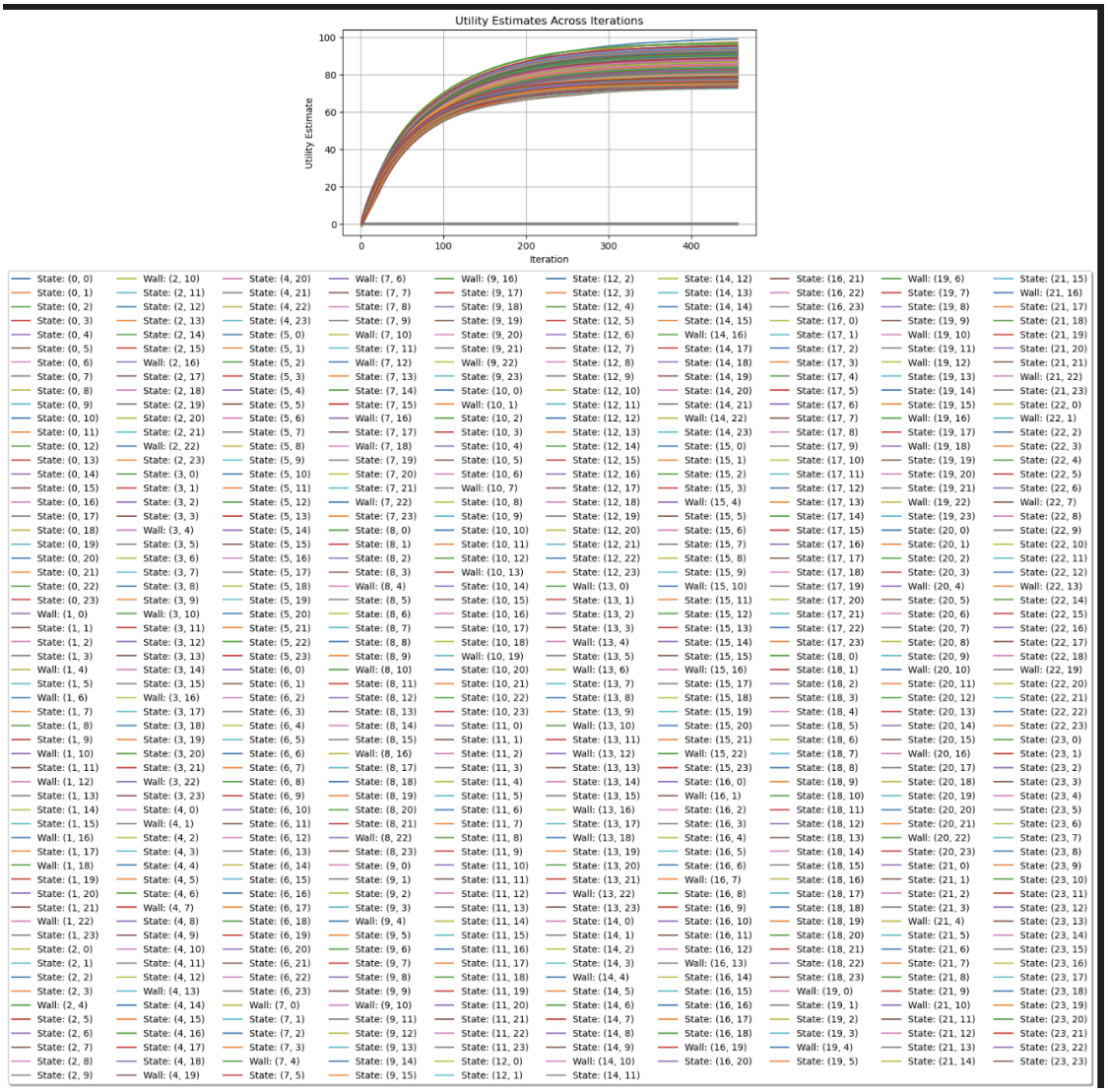


Fig 5.2.2.1

Policy Iteration -

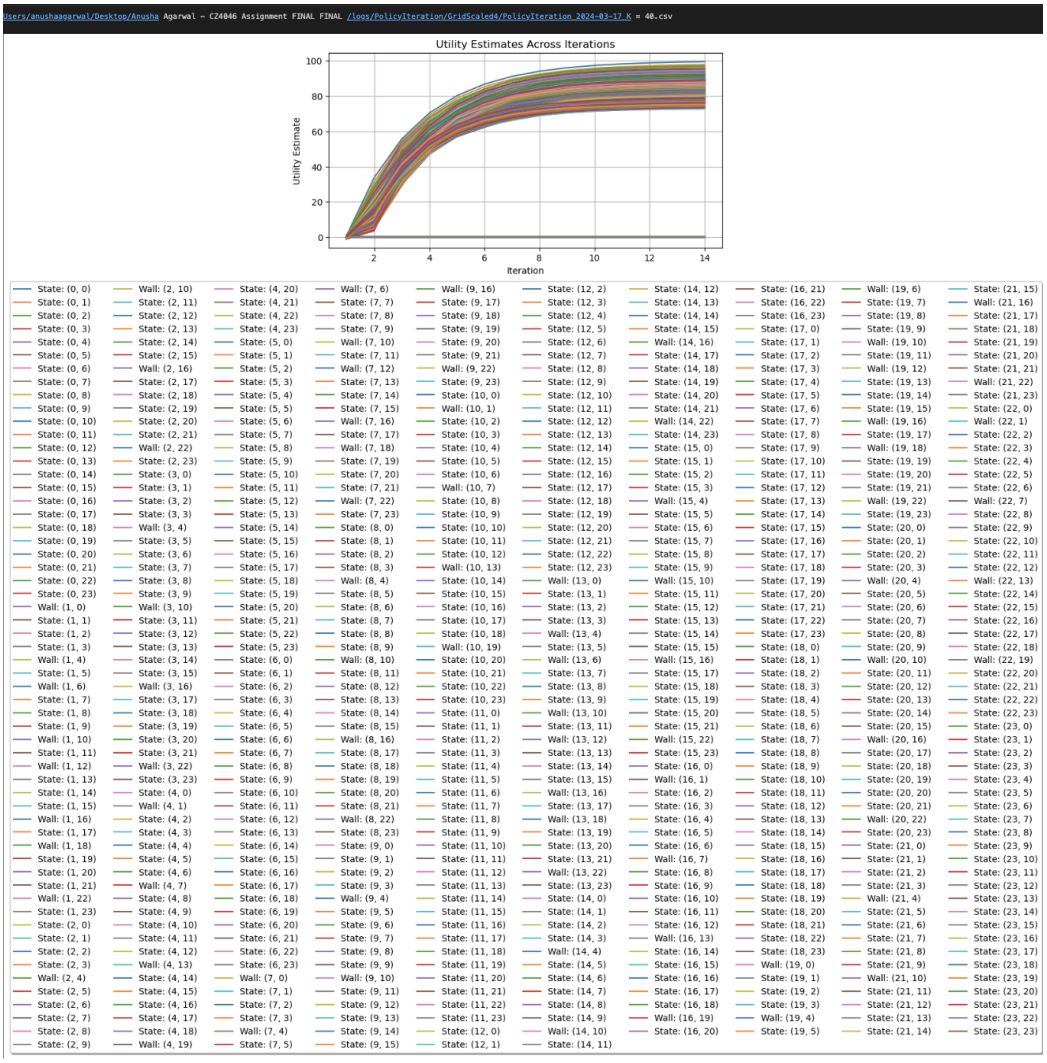


Fig 5.2.2.2

This maze was scaled 2, 4, 100 times (System crashed on 100). Therefore, increasing the maze size has a considerable impact on the number of iterations taken to find optimal policy for both algorithms thus creating a complicated maze environment where agents need to explore a bigger grid to find optimal solutions. As a result, larger the space complexity, higher the time complexity of convergence in finding the best policy. Moreover, with this the computational cost increases along with the data required for learning.