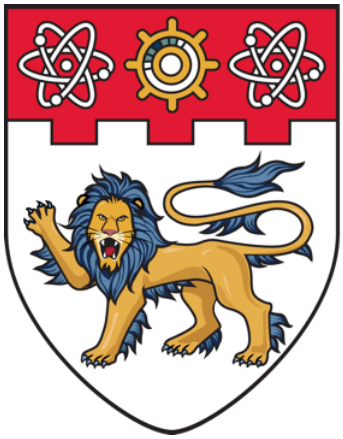

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4046 Intelligent Agents
Assignment 1

Marathe Ajinkya Avinash
U1522716K

Table of Contents

Objective:

The objective of the assignment is to find the optimal policy and utilities of the non-walled states in the grid using both value iteration and policy iteration. The grid has states with different rewards and the agents state sequence is infinite.

Value Iteration:

Value Iteration calculates the utilities of all the states using the Bellman equation and then chooses the policy (action of the agent) in the direction of highest utility amongst its neighbour states. The bellman equation is as given below.

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

Figure 0-1 Bellman Equation

The first step of value iteration is to initialise utilities of all states to 0. Utilities for each state are calculated for several iterations before the utilities converge.

Convergence in Value Iteration:

To find out the optimal number of iterations at which we should stop computing the utilities, we introduce a threshold epsilon. At first, the difference between the new utility and the old utility is calculated at each iteration for each state. When the maximum delta among all states is lesser than epsilon, then the value iteration process is stopped.

Hyperparameter optimization:

The optimal value of epsilon threshold was found manually by trial and error. As given in the question, the maximum number of iterations was taken to be 50. The epsilon value was initialised to 1. When the maximum delta reached epsilon, the number of iterations N was noted. The process was continued till the 50^{th} iteration. If the policy at 50^{th} iteration was different from the policy at N^{th} iteration, then the threshold epsilon is reduced. This process was followed to find out the threshold epsilon

Epsilon	Epsilon Convergence Iterations	Policy same as 50 iterations
1.0	6	No
0.9	12	No
0.8	24	No
0.7	37	Yes

Utilities after 37 Iterations:

31.06 0.00 29.99 29.49 28.95 30.17
30.23 28.50 29.46 30.20 0.00 28.39
29.57 28.95 28.06 29.60 30.36 29.78
28.93 28.49 28.01 28.10 29.75 30.51
28.35 0.00 0.00 0.00 28.18 29.80
27.72 27.16 26.61 26.94 28.10 29.17

Optimal Policy after 37 iterations:

N		N	W	E	N
N	W	N	N		N
N	W	N	N	N	W
N	W	W	N	N	E
N				N	N
N	W	W	E	E	N

N → NORTH
S → SOUTH
W → WEST
E → EAST
■ →WALL

Utility Plot:

States (0,0) and (5,5) are taken as examples to explain the behaviour of other states. As it can be seen in Figure 0-1 and Figure 0-3, for 37 iterations of value iteration, the graph of utilities against iterations is approximately a linear function. When the value iteration algorithm is run for 500 iterations (without applying convergence criteria), the graph is exponential and reaches convergence naturally as shown in Figure 0-2 and Figure 0-4. The utility graphs for all states have been shown in Figure 0-5. Utilities for walled states have not been calculated and thus there graphs are flat.

State(0,0):

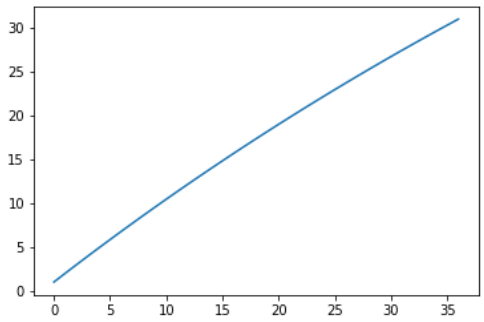

Figure 0-2 After 37 Iterations

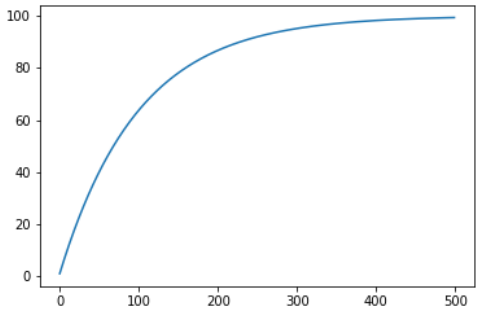

Figure 0-3 After 500 Iterations

State(5,5):

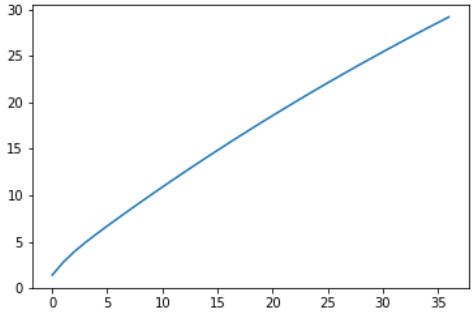

Figure 0-4 After 37 Iterations

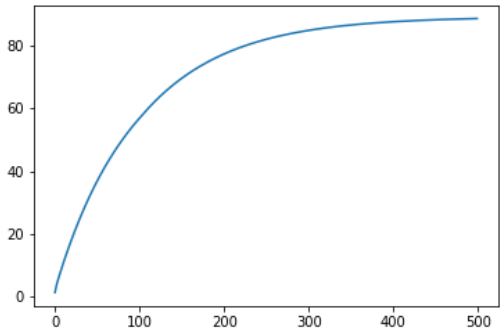

Figure 0-5 After 500 Iterations

All States (6x6):

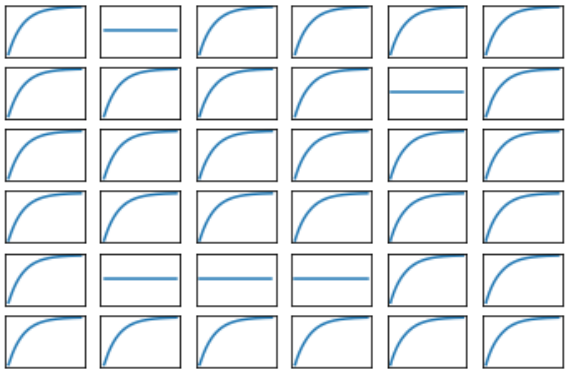

Figure 0-6 After 500 Iterations

Code Explanation for Value Iteration:

For each iteration of the algorithm, the utility for each state is calculated and the deltas are stored in an array. After each iteration, the maximum value from the deltas is compared against the epsilon to check for convergence. Utility values for each state for each iteration are written into a text file to draw graphs. Graphs have been drawn using python. The method computeOptimalUtilityAndAction() has been explained in the appendix

```java
public static void valueIteration(Grid grid) {
    // convergence value
    double EPSILON = 0.7;

    for (int iterations = 1; iterations <= 50; iterations++) {
        // array for storing all the delta values in an interation
        ArrayList<Double> delta = new ArrayList<>();
        for (int i = 0; i < grid.getNoRows(); i++) {
            for (int j = 0; j < grid.getNoCols(); j++) {
                // computing utility using bellman equation
                double new_utility = UtilityFunctions.computeOptimalUtilityAndAction(grid, i, j, algorithm: "value");
                // computing delta
                delta.add(Math.abs(new_utility - grid.gridWorld[i][j].getUtility()));
                // setting new utility
                grid.gridWorld[i][j].setUtility(new_utility);
                // writing utility values into the file for this iteration
                try {

                    FileWriter fstream = new FileWriter( fileName: "file"+Integer.toString(i)+Integer.toString(j)+".txt", append: true);
                    BufferedWriter out = new BufferedWriter(fstream);
                    out.write( str: Double.toString(grid.gridWorld[i][j].getUtility())+"\n");
                    out.close();

                }
                catch(IOException ioe){


                }
            }
        }
        // Checking convergence condition
        if (Collections.max(delta) <= EPSILON) {
            System.out.println("Iterations==" + iterations);
            break;
        }

    }
```

Policy Iteration:

In policy Iteration algorithm, all the states are initialised with a policy. In this case, all the states have been initialised with West as the initial action for the policy. The policy iteration algorithm has two steps-

Policy Evaluation:

Given the policy, the utility of each state is calculated if the given action (policy) is executed. This is done using a modified form of the Bellman Equation where only the intended action of the current policy is considered instead of the maximum of all intended actions. The equation is as shown in figure 0-7.

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s') \ .$$

Figure 0-7 Bellman equation for policy evaluation

For each iteration of policy evaluation, the utilities are updated for a number of iterations till they converge (are lesser than or equal to epsilon). Here the epsilon is taken as 0.7

Policy Improvement:

After every policy evaluation, the Bellman equation is used to calculate the maximum using all intended actions. If the policy we get using this method is same as the existing policy for all states, then the policy iteration algorithm is considered to have converged.

Utilities after 3 Iterations:

33.90 0.00 32.47 31.95 31.37 32.59
33.05 31.28 31.94 32.63 0.00 30.78
32.35 31.71 30.55 32.00 32.67 32.07
31.68 31.22 30.68 30.50 32.04 32.75
31.08 0.00 0.00 0.00 30.44 32.02
30.41 29.83 29.26 29.12 30.29 31.37

Policy Evaluation Iteration:

For each iteration of policy iteration , the algorithm took 5,32 and 2 iterations of policy evaluation respectively.

Optimal Policy for 3 Iterations:

N		N	W	E	N
N	W	N	N		N
N	W	N	N	N	W
N	W	W	N	N	N
N				N	N
N	W	W	E	E	N

Utility Plot:

State(0,0):

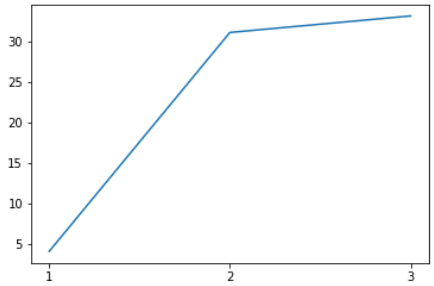

Figure 0-8 after 3 iterations

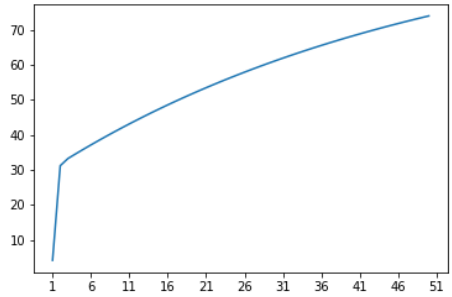

Figure 0-9 After 50 iterations

State(5,5):

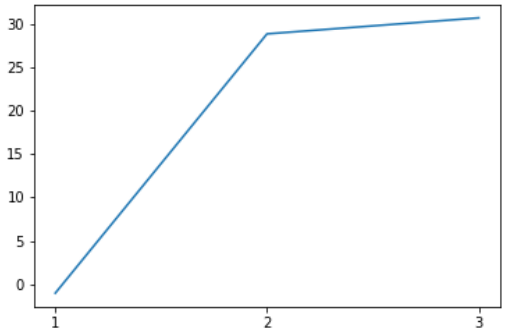

Figure 0-10 After 3 Iterations

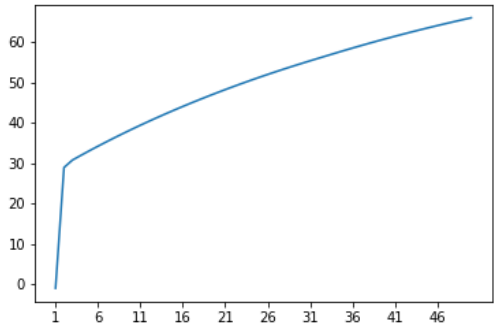

Figure 0-11 After 50 Iterations

All States:

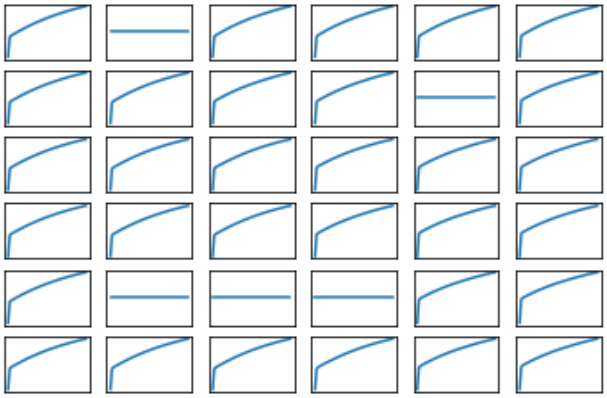

Figure 0-12 After 50 iterations

Code for Policy Iteration:

```
public static void policyIteration(Grid grid) {
    // policy iterations
    for (int iterations = 1; iterations <= 50; iterations++) {
        // convergence criteria to check change in policy
        boolean convergence = true;
        // convergence value for policy evaluation
        double convergence_value = 0.7;
        // Policy Evaluation iterations
        for (int policyEvalIterations = 1; policyEvalIterations < 50; policyEvalIterations++) {
            ArrayList<Double> delta = new ArrayList<>();
            for (int i = 0; i < grid.getNoRows(); i++) {
                for (int j = 0; j < grid.getNoCols(); j++) {
                    // Calculating utilities for the given action
                    double currentUtility = grid.gridWorld[i][j].getUtility();
                    double newUtility = UtilityFunctions.computeOptimalUtilityAndAction(grid, i, j, algorithm: "policy");
                    grid.gridWorld[i][j].setUtility(newUtility);
                    delta.add(Math.abs(currentUtility - newUtility));
                }
            }
            // checking for policy evalaution convergence
            if (Collections.max(delta) <= convergence_value) {
                System.out.println(policyEvalIterations);
                break;
            }
        }
        // writing utilities  to file after policy evaluation
        for (int i = 0; i < grid.getNoRows(); i++) {
            for (int j = 0; j < grid.getNoCols(); j++) {
                try {
                    FileWriter fstream = new FileWriter( fileName: "file"+Integer.toString(i)+Integer.toString(j)+".txt", append: true);
                    BufferedWriter out = new BufferedWriter(fstream);
                    out.write( str: Double.toString(grid.gridWorld[i][j].getUtility())+"\n");
                    out.close();
                }
                catch(IOException ioe){
                }
                String currentPolicy = grid.gridWorld[i][j].getAction();
                grid.gridWorld[i][j].setUtility(UtilityFunctions.computeOptimalUtilityAndAction(grid,i,j, algorithm: "value"));
                String newPolicy=grid.gridWorld[i][j].getAction();
                if (currentPolicy!=newPolicy) convergence=false;
```

```
                }
                }
            // checking for policy iteration convergence
            if (convergence == true) {
                System.out.println(" Policy Iterations==" + iterations);
                break;
            }
        }
    }
}
```

Bonus question:

The grid world was made more complex by making it a rectangle of higher dimension. The dimensions chosen were (12x6). To create this grid the existing grid was doubled while making some minor changes to make it difficult

Grid:

```
g,wl,g,w,w,g
w,b,wl,b,wl,b
w,w,b,w,g,w
w,w,w,g,w,g
w,wl,wl,wl,b,w
w,w,b,w,w,b
wl,wl,g,w,w,g
w,b,w,g,wl,b
w,w,wl,wl,g,w
w,w,w,b,w,g
w,wl,wl,wl,b,w
w,w,wl,b,b,wl
```

Value Iteration (37 Iterations):

Utilities:

31.06 0.00 31.06 30.23 29.63 30.57
30.23 28.50 0.00 28.44 0.00 28.78
29.57 28.95 27.27 28.28 29.28 28.87
28.93 28.49 27.95 29.15 28.91 29.64
28.35 0.00 0.00 0.00 27.34 28.94
27.72 27.16 27.12 27.95 27.57 27.29
0.00 0.00 29.36 29.23 28.61 28.99
25.57 26.81 29.30 30.45 0.00 27.23
25.17 26.21 0.00 0.00 29.71 29.07
24.76 25.63 25.44 26.61 29.01 29.78
24.24 0.00 0.00 0.00 27.44 29.08
23.67 23.17 0.00 23.15 25.49 0.00

Policy:

N * W W W N
N W * N * N
N W W E N W
N W E S N E
N * * * N N
N W S S W N
* * S S W W
E E E S * N
N N * * N W
N N E E N E
N * * * N N
N W * E N *

‘*’ stands for wall. The utilities and policies of wall are not calculated

Policy Iteration (11 Iterations):

Utilities:

52.89 0.00 61.90 60.72 59.82 60.45
52.04 51.77 0.00 58.62 0.00 58.36
52.94 54.07 55.30 57.58 58.05 57.59
53.85 55.21 56.51 57.84 57.37 58.00
53.04 0.00 0.00 0.00 55.50 57.00
52.14 51.35 51.51 53.46 54.62 54.96
0.00 0.00 53.63 53.61 54.25 55.34
49.79 51.03 53.53 54.68 0.00 53.32
49.15 50.17 0.00 0.00 53.90 53.20
48.49 49.35 49.37 50.54 52.93 53.65
47.75 0.00 0.00 0.00 51.12 52.69
46.92 46.19 0.00 46.57 48.90 0.00

Policy:

N * W W W N
N S * N * N
E S E N N N
E E E N N N
N * * * N N
N W S E N N
* * S S E N
E E E E * N
N N * * N W
N N E E N N
N * * * N N
N W * E N *

Conclusions:

The value iterations and policy iterations take longer to converge when the maze gets larger in size. In this case, the maze might be unsolvable.
The maze might also be unsolvable if the convergence value is too low as the delta may not reach that value.
The maze might also be unsolvable if the grid initialised has no definitive pathway due to the random arrangement of walls and states.

Appendix A:

Code Skeleton:

State Class:

An object of class state stands for a single state in the grid world and stores the utility, reward, the position of the state in the grid and other information about the state.

Grid Class:

The object of the Grid class is a grid world i.e. a 2D array of states. The initialiseGridStates() method takes in the config file and initialises the grid with the states

```java
// Read text file and initialise grid
public void initialiseGridStates(){
    try {
        int row_no=0;
        BufferedReader reader = new BufferedReader(new FileReader(this.configPath));
        String line=reader.readLine();
        while (line!=null){

            String[] parts=line.split( regex: ",");

            for(int col_no=0;col_no<this.getNoCols();col_no++){
                //grid_states[row_no][col_no]=parts[col_no];
                String state=parts[col_no];
                switch (state) {
                    case "wl":
                        gridWorld[row_no][col_no] = new State(row_no, col_no,  reward: 0,  utility: 0,  isWall: true);
                        break;
                    default:
                        gridWorld[row_no][col_no] = new State(row_no, col_no, stateRewards.get(state),  utility: 0,  isWall: false);
                        break;

                }

            }
            line=reader.readLine();
            ++row_no;
        }
    }
    catch(FileNotFoundException fnp){

    }
    catch (IOException ioe) {

    }



}
```

The intialisePossibleActions() method checks the possible actions for all states depending on their position (walls, edges etc)

```
// initialise possible actions for each state depending on position in the grid
// Possible actions are those if a movement can be made without colliding with maze boundary or wall and position can be changed
private void initialisePossibleActions(){
    for(int i = 0; i< this.noRows; i++) {
        for (int j = 0; j < this.noCols; j++) {
            // no possible actions for wall states
            if (gridWorld[i][j].isWall()) continue;
            // If state is not in the first row and does not have wall immediately above
            if((i!=0) && !(gridWorld[i-1][j].isWall())) gridWorld[i][j].setMoveNorth(true);
            // If state is not in the last row and does not have wall immediately below
            if((i!= this.noRows -1)&&!(gridWorld[i+1][j].isWall())) gridWorld[i][j].setMoveSouth(true);
            // If state is not in the leftmost column and there is no wall immediately to the west
            if((j!=0) && !(gridWorld[i][j-1].isWall())) gridWorld[i][j].setMoveWest(true);
            // If state is not in the rightmost column and there is no wall to immediately to the east
            if((j!= this.noCols -1)&&!(gridWorld[i][j+1].isWall())) gridWorld[i][j].setMoveEast(true);


        }
    }

}
```

Utility Functions:

The UtilityFunctions class consists of utility functions for value and policy iterations. The function computeOptimalUtilityAndAction() computes the bellman equation values for both policy and value iteration depending on the method parameter “algorithm”

```
public static double computeOptimalUtilityAndAction(Grid grid,int i, int j,String algorithm){
    State state=grid.gridWorld[i][j];
    if (state.isWall()) return 0 ;
    double utility=state.getReward();
    double max=0;
    String action="";
    double sumN,sumS,sumW,sumE;
    double north_utility,south_utility,west_utility,east_utility;
    // If it is possible for the agent to move in that direction,get the utilities of neighbouring states
    if (state.isMoveNorth()){
        north_utility=grid.gridWorld[i-1][j].getUtility();
    }
    else north_utility=grid.gridWorld[i][j].getUtility();
    if (state.isMoveSouth()){
        south_utility=grid.gridWorld[i+1][j].getUtility();
    }
    else south_utility=grid.gridWorld[i][j].getUtility();
    if (state.isMoveWest()){
        west_utility=grid.gridWorld[i][j-1].getUtility();
    }
    else west_utility=grid.gridWorld[i][j].getUtility();
    if (state.isMoveEast()){
        east_utility=grid.gridWorld[i][j+1].getUtility();
    }
    else east_utility=grid.gridWorld[i][j].getUtility();
    // calculate the sum in intended directions
    sumN=INTENDED_PROBABILITY*north_utility+OTHER_PROBABILITY*(west_utility+east_utility);
    sumS=INTENDED_PROBABILITY*south_utility+OTHER_PROBABILITY*(west_utility+east_utility);
    sumW=INTENDED_PROBABILITY*west_utility+OTHER_PROBABILITY*(north_utility+south_utility);
    sumE=INTENDED_PROBABILITY*east_utility+OTHER_PROBABILITY*(north_utility+south_utility);
    // For Value Iterations and policy Evaluation
    if(algorithm=="value"){
        // computing the max value
        max=Math.max(Math.max(sumN,sumS),Math.max(sumE,sumW));
        // Assigning action to the state
        if(max==sumN) action="N";
        else if(max==sumS) action="S";
        else if(max==sumW) action="W";
        else action="E";
        // computing utility
        utility+=DISCOUNT_FACTOR*max;
```

```
        // Setting action
        state.setAction(action);

    }
    // For policy Iteration just calculating in the given direction
    else if(algorithm=="policy"){
        switch (state.getAction()){
            case (State.North):
                utility+=DISCOUNT_FACTOR*sumN;
                break;
            case (State.South):
                utility+=DISCOUNT_FACTOR*sumS;
                break;
            case (State.West):
                utility+=DISCOUNT_FACTOR*sumW;
                break;
            case (State.East):
                utility+=DISCOUNT_FACTOR*sumE;
                break;

        }

    }



    // return utility
    return utility;

}
```

End