NANYANG TECHNOLOGICAL UNIVERSITY

CZ4046: Intelligent Agents

Assignment 1

HEW GUO WEI

U1622752K

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

HEW GUO WEI
U1622752K

Objective: To write a Java Program to apply Value Iteration and Policy Iteration to solve a MDP problem.

<u>Part1:</u>

Object-oriented programming is used in this assignment. First of all, a Cell object is created to represent each cell in the maze. Each cell has attributes like policy (direction), utility and state, reward. State allows the user to assign nature of the cell, such as Wall, empty, green, brown, which associates with different rewards each, to a particular cell,

```java
public class Cell {

    private double reward;
    private double utility;
    private double tempUtility;
    private State state;
    private Policy policy;
    private Policy tempPolicy;

    enum State{
        NONE, GREEN, BROWN, WALL
    }

    enum Policy{
        NONE, _UP_, DOWN, LEFT, RIGHT
    }

    public Cell(State state){

        this.state = state;

        switch(state){
            case NONE:
                this.reward = -0.04;
                break;
            case GREEN:
                this.reward = +1;
                break;
            case BROWN:
                this.reward = -1;
                break;
            case WALL:
                this.reward = 0;
                break;
        }

        policy = Policy.NONE;
        tempPolicy = Policy.NONE;
        utility= 0;
        tempUtility = 0;

    }
```

```java
public class Main {

    public static void main(String[] args) {
        final double DISCOUNT = 0.99;

        Cell[][] cells;
        cells = new Cell[8][8];

        cells[1][1] = new Cell(Cell.State.GREEN); //0,0
        cells[1][3] = new Cell(Cell.State.GREEN); //0.2
        cells[1][6] = new Cell(Cell.State.GREEN); //0.5
        cells[2][4] = new Cell(Cell.State.GREEN); //1,3
        cells[3][5] = new Cell(Cell.State.GREEN); //2,4
        cells[4][6] = new Cell(Cell.State.GREEN); //3,5

        cells[2][2] = new Cell(Cell.State.BROWN); //1,1
        cells[2][6] = new Cell(Cell.State.BROWN); //1,5
        cells[3][3] = new Cell(Cell.State.BROWN); //2,2
        cells[4][4] = new Cell(Cell.State.BROWN); //3,3
        cells[5][5] = new Cell(Cell.State.BROWN); //4,4

        cells[1][2] = new Cell(Cell.State.WALL); //0,1
        cells[2][5] = new Cell(Cell.State.WALL); //1,4
        cells[5][2] = new Cell(Cell.State.WALL); //4,1
        cells[5][3] = new Cell(Cell.State.WALL); //4,2
        cells[5][4] = new Cell(Cell.State.WALL); //4,3

        //build a walled-shell around the grid
        for(int i=0; i<8;i++){
            cells[0][i] = new Cell(Cell.State.WALL);
            cells[7][i] = new Cell(Cell.State.WALL);
            cells[i][0] = new Cell(Cell.State.WALL);
            cells[i][7] = new Cell(Cell.State.WALL);
        }

        //initialize all other empty cells
        for(int i=1; i<7; i++){
            for(int j=1; j<7; j++){
                if(cells[i][j] == null){
                    cells[i][j] = new Cell(Cell.State.NONE);
                }
            }
        }

        //initialize all other empty cells
        for(int i=1; i<7; i++){
            for(int j=1; j<7; j++){
                System.out.print("["+cells[i][j].getState()+"]");
            }
            System.out.println();
        }
```

```
[GREEN] [WALL] [GREEN] [NONE] [NONE] [GREEN]
[NONE] [BROWN] [NONE] [GREEN] [WALL] [BROWN]
[NONE] [NONE] [BROWN] [NONE] [GREEN] [NONE]
[NONE] [NONE] [NONE] [BROWN] [NONE] [GREEN]
[NONE] [WALL] [WALL] [WALL] [BROWN] [NONE]
[NONE] [NONE] [NONE] [NONE] [NONE] [NONE]
```

In the initialized stage, all the cells are initialized according to their states given. Then, an additional layer of wall is built around the maze. The intention is such that when we need to calculate the utilities of the out-of-bound cells, it would be considered as a walled cell.

HEW GUO WEI
U1622752K

```java
public class Policies {

    //curr, up, down, left, right
    static public void valueIteration(Cell cell0, Cell cell1, Cell cell2, Cell cell3, Cell cell4, double DISCOUNT){

        double U1, U2, U3, U4;
        double UP, DOWN, LEFT, RIGHT;

        if(cell1.getState() == Cell.State.WALL){
            UP = cell0.getUtility();
        }else{
            UP = cell1.getUtility();
        }

        if(cell2.getState() == Cell.State.WALL){
            DOWN = cell0.getUtility();
        }else{
            DOWN = cell2.getUtility();
        }

        if(cell3.getState() == Cell.State.WALL){
            LEFT = cell0.getUtility();
        }else{
            LEFT = cell3.getUtility();
        }

        if(cell4.getState() == Cell.State.WALL){
            RIGHT = cell0.getUtility();
        }else{
            RIGHT = cell4.getUtility();
        }

        U1 = 0.8*UP + 0.1*LEFT + 0.1*RIGHT; U2 = 0.8*DOWN + 0.1*LEFT + 0.1*RIGHT;
        U3 = 0.8*LEFT + 0.1*UP + 0.1*DOWN;
        U4 = 0.8*RIGHT + 0.1*UP + 0.1*DOWN;

        double temp1 = Math.max(U1,U2);
        double temp2 = Math.max(U3,U4);
        double finaltemp = Math.max(temp1,temp2);

        //set policy
        if(finaltemp == U1){
            cell0.setTempPolicy(Cell.Policy._UP_);
        }

        if(finaltemp == U2){
            cell0.setTempPolicy(Cell.Policy.DOWN);
        }

        if(finaltemp == U3){
            cell0.setTempPolicy(Cell.Policy.LEFT);
        }
```

```java
        if(finaltemp == U3){
            cell0.setTempPolicy(Cell.Policy.LEFT);
        }

        if(finaltemp == U4){
            cell0.setTempPolicy(Cell.Policy.RIGHT);
        }

        //return utility
        double tempUtility = DISCOUNT*finaltemp + cell0.getReward();
        cell0.setTempUtility(tempUtility);

    }
```

In the case of value iteration, when we are calculating the utility of a particular cell, there is a function that takes in cells involved (the cell itself and the 4 other cells around it) as the parameters. Then, if one of the cells is wall, the utility of the main cell is taken instead. Since value iteration has a max operator, the function will return the maximum discounted utility among the 4 possible actions. It is important to note, that in one particular iteration, policy and utility values of all cells are constant, and only temporary policy, utility values are changing while these temp values will only be updated to the main values when a particular iteration has finished.

HEW GUO WEI
U1622752K

```
//VALUE ITERATION
//double error = 0.000001;
double maxerr = 0.00002;
int count1 = 0;
while(maxerr > 0.00001){
    //for(int x=0; x<4000; x++){
    maxerr = 0;
    for(int i=1; i<7; i++){
        for(int j=1; j<7; j++){
            if(cells[i][j].getState() != Cell.State.WALL){

                Policies.valueIteration(cells[i][j],cells[i-1][j],cells[i+1][j],cells[i][j-1], cells[i][j+1], DISCOUNT);

                //compare old utility with new utility
                double diff = cells[i][j].getTempUtility() - cells[i][j].getUtility();
                if(diff > maxerr){
                    maxerr = diff;
                }
            }
        }
    }

    //transfer temp utility value to the main utility value
    for(int i=1; i<7; i++){
        for(int j=1; j<7; j++){
            if(cells[i][j].getState() != Cell.State.WALL){
                cells[i][j].setUtility(cells[i][j].getTempUtility());
            }
        }
    }
    count1++;
}
```

During the actual run, an epsilon is set. In the while loop, the difference between the cell's temporary utility and the old utility will be recorded, and finally the maximum among these value differences will be returned and compared against the epsilon value. While loop stops when the max difference is finally smaller than the epsilon value. The 2nd for loop is simply to update the current utilities from the temporary utilities.

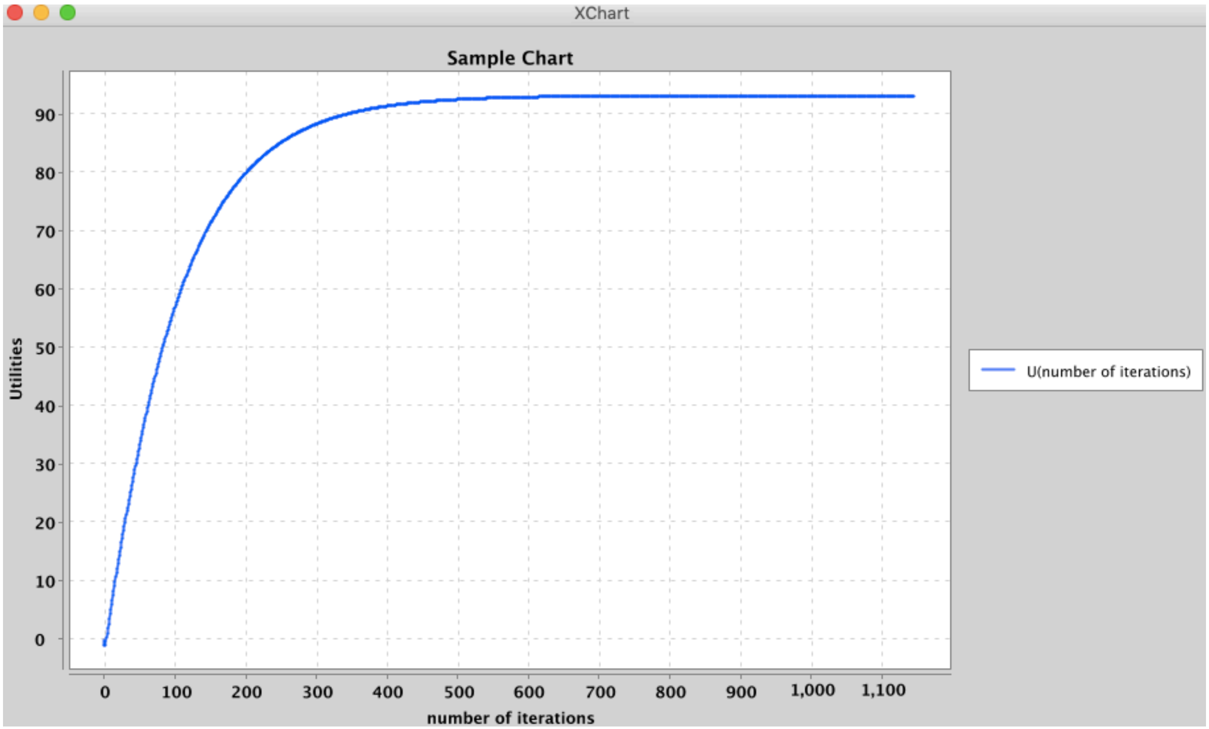
```
------VALUE ITERATION----

[100.00][.00][95.04][93.87][92.65][93.33]
[98.39][95.88][94.54][94.40][.00][90.92]
[96.95][95.59][93.29][93.18][93.10][91.79]
[95.55][94.45][93.23][91.11][91.81][91.89]
[94.31][.00][.00][.00][89.55][90.57]
[92.94][91.73][90.53][89.36][88.57][89.30]
number of value iteration: 1147

[_UP_][NONE][LEFT][LEFT][LEFT][_UP_]
[_UP_][LEFT][LEFT][LEFT][NONE][_UP_]
[_UP_][LEFT][LEFT][_UP_][LEFT][LEFT]
[_UP_][LEFT][LEFT][_UP_][_UP_][_UP_]
[_UP_][NONE][NONE][NONE][_UP_][_UP_]
[_UP_][LEFT][LEFT][LEFT][_UP_][_UP_]
```

The above values are results from 1147 iterations.

HEW GUO WEI
U1622752K



This above graph is plotted with utilities of a particular maze cell againts the number of value iteration performed. There is a total of 1147 points. As shown above, the value converges as the number of iteration increases.

```
    xData[count1] = count1;
    yData[count1] = cells[3][3].getUtility();
}

// Create Chart
XYChart chart = QuickChart.getChart( chartTitle: "Sample Chart",  xTitle: "number of iterations",  yTitle: "Utilities",  seriesName: "U(number of iterations)", xData, yData);
//display chart in separate windows
new SwingWrapper(chart).displayChart();
```

An external library Xchart is used to plot the graph.

HEW GUO WEI
U1622752K

There are 2 parts to Policy Iteration; the first is policy evaluation, while the second is policy iteration.

```
static public void policyEvaluation(Cell cell0, Cell cell1, Cell cell2, Cell cell3, Cell cell4, double DISCOUNT){

    double U1, U2, U3, U4;
    double UP, DOWN, LEFT, RIGHT;

    if(cell1.getState() == Cell.State.WALL){
        UP = cell0.getUtility();
    }else{
        UP = cell1.getUtility();
    }

    if(cell2.getState() == Cell.State.WALL){
        DOWN = cell0.getUtility();
    }else{
        DOWN = cell2.getUtility();
    }

    if(cell3.getState() == Cell.State.WALL){
        LEFT = cell0.getUtility();
    }else{
        LEFT = cell3.getUtility();
    }

    if(cell4.getState() == Cell.State.WALL){
        RIGHT = cell0.getUtility();
    }else{
        RIGHT = cell4.getUtility();
    }

    U1 = 0.8*UP + 0.1*LEFT + 0.1*RIGHT;
    U2 = 0.8*DOWN + 0.1*LEFT + 0.1*RIGHT;
    U3 = 0.8*LEFT + 0.1*UP + 0.1*DOWN;
    U4 = 0.8*RIGHT + 0.1*UP + 0.1*DOWN;

    switch(cell0.getPolicy()){
        case _UP_:
            cell0.setTempUtility(DISCOUNT*U1 + cell0.getReward());
            break;
        case DOWN:
            cell0.setTempUtility(DISCOUNT*U2 + cell0.getReward());
            break;
        case LEFT:
            cell0.setTempUtility(DISCOUNT*U3 + cell0.getReward());
            break;
        case RIGHT:
            cell0.setTempUtility(DISCOUNT*U4 + cell0.getReward());
            break;
    }
}
```

Similarly, policy evaluation takes 5 cells as parameters and based on the main cell's current policy, it calculates the the utility and returns that as the utility of that cell.

HEW GUO WEI
U1622752K

```java
static public void policyIteration(Cell cell0, Cell cell1, Cell cell2, Cell cell3, Cell cell4, double DISCOUNT){

    double U1, U2, U3, U4;
    double UP, DOWN, LEFT, RIGHT;

    if(cell1.getState() == Cell.State.WALL){
        UP = cell0.getUtility();
    }else{
        UP = cell1.getUtility();
    }

    if(cell2.getState() == Cell.State.WALL){
        DOWN = cell0.getUtility();
    }else{
        DOWN = cell2.getUtility();
    }

    if(cell3.getState() == Cell.State.WALL){
        LEFT = cell0.getUtility();
    }else{
        LEFT = cell3.getUtility();
    }

    if(cell4.getState() == Cell.State.WALL){
        RIGHT = cell0.getUtility();
    }else{
        RIGHT = cell4.getUtility();
    }
    U1 = 0.8*UP + 0.1*LEFT + 0.1*RIGHT;
    U2 = 0.8*DOWN + 0.1*LEFT + 0.1*RIGHT;
    U3 = 0.8*LEFT + 0.1*UP + 0.1*DOWN;
    U4 = 0.8*RIGHT + 0.1*UP + 0.1*DOWN;

    double temp1 = Math.max(U1,U2);
    double temp2 = Math.max(U3,U4);
    double finaltemp = Math.max(temp1,temp2);

    //set policy
    if(finaltemp == U1){
        cell0.setTempPolicy(Cell.Policy._UP_);
    }

    else if(finaltemp == U2){
        cell0.setTempPolicy(Cell.Policy.DOWN);
    }

    else if(finaltemp == U3){
        cell0.setTempPolicy(Cell.Policy.LEFT);
    }

    else if(finaltemp == U4){
        cell0.setTempPolicy(Cell.Policy.RIGHT);
    }
```

Policy iteration simply returns the policy with the highest utility.

HEW GUO WEI
U1622752K

```java
//POLICY ITERATION

boolean unchanged = false;
int count2 = 0;

while(unchanged == false){

    double maxerr2 = 0.00002;
    int count3 = 0;
    while(maxerr2 > 0.00001) {

        maxerr2 = 0;
        for(int i=1; i<7; i++){
            for(int j=1; j<7; j++){
                if(cells[i][j].getState() != Cell.State.WALL) {

                    Policies.policyEvaluation(cells[i][j], cells[i - 1][j], cells[i + 1][j], cells[i][j - 1], cells[i][j + 1], DISCOUNT);

                    double diff = cells[i][j].getTempUtility() - cells[i][j].getUtility();
                    if (diff > maxerr2) {
                        maxerr2 = diff;
                    }
                }
            }
        }

        for(int i=1; i<7; i++){
            for(int j=1; j<7; j++){
                if(cells[i][j].getState() != Cell.State.WALL){

                    //transfer temp utility value to the main utility value
                    cells[i][j].setUtility(cells[i][j].getTempUtility());

                }
            }
        }

        count3++;
    }

    System.out.println("number of fixed-policy evaluation per iteration: " + count3);
    unchanged = true;

    for(int i=1; i<7; i++){
        for(int j=1; j<7; j++){
            if(cells[i][j].getState() != Cell.State.WALL) {

                Policies.policyIteration(cells[i][j],cells[i-1][j],cells[i+1][j],cells[i][j-1], cells[i][j+1], DISCOUNT);

                //compare new policy with old policy
                if(cells[i][j].getTempPolicy() != cells[i][j].getPolicy()){
                    cells[i][j].setPolicy(cells[i][j].getTempPolicy());
                    unchanged = false;
                }
            }
        }
    }
    count2++;
}

System.out.println();

//display policy in grid
for(int i=1; i<7; i++){
    for(int j=1; j<7; j++){
        System.out.print("[" + cells[i][j].getPolicy()+"]");
    }
    System.out.println();
}

System.out.println("number of policy iteration: " + count2);
```

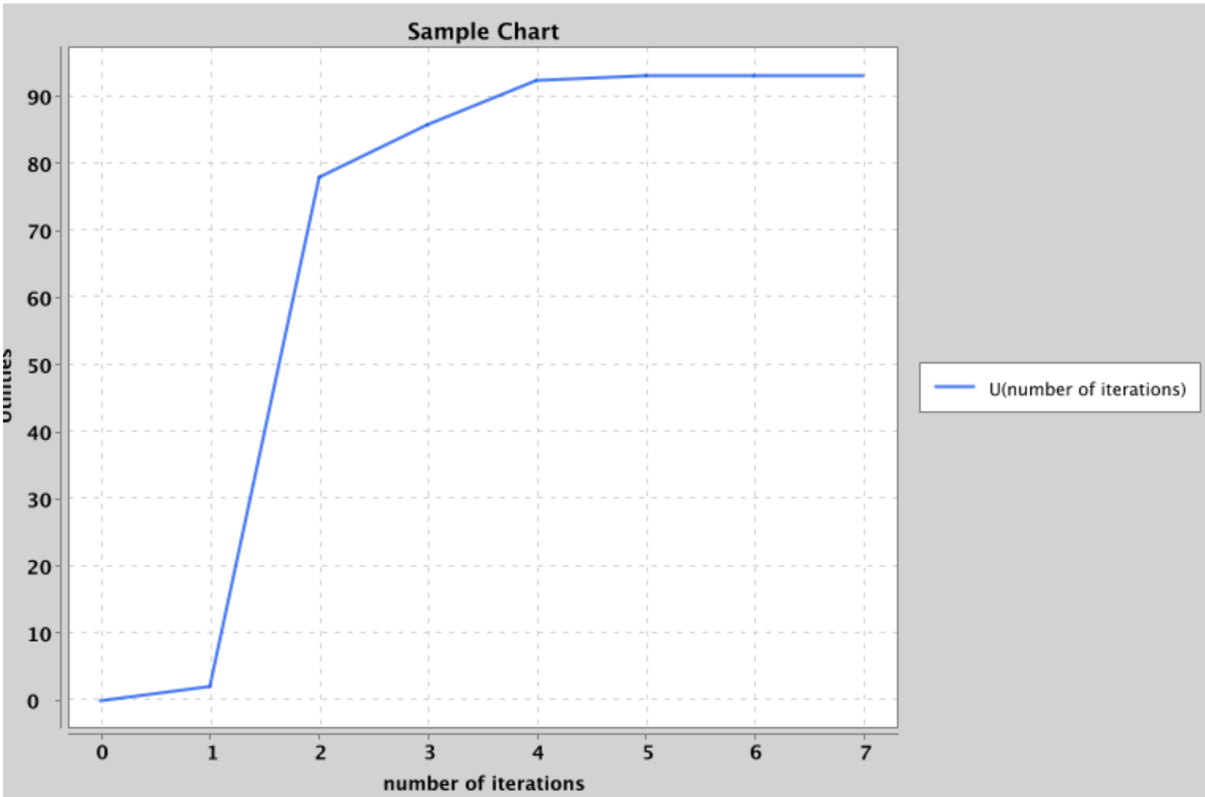
In the actual run, for every policy pre-set, policy evaluation is repeated until the current utilities are stable with the help of an epsilon value. Only then, policy iteration is run. Policy iteration will return the new policy if there is any change. When there are absolutely no changes of policies in one particular policy iteration, the while loop stops.

HEW GUO WEI
U1622752K

```
number of fixed-policy evaluation per iteration: 351
number of fixed-policy evaluation per iteration: 1150
number of fixed-policy evaluation per iteration: 876
number of fixed-policy evaluation per iteration: 163
number of fixed-policy evaluation per iteration: 100
number of fixed-policy evaluation per iteration: 16
number of fixed-policy evaluation per iteration: 16

[_UP_][NONE][LEFT][LEFT][LEFT][_UP_]
[_UP_][LEFT][LEFT][LEFT][NONE][_UP_]
[_UP_][LEFT][LEFT][_UP_][LEFT][LEFT]
[_UP_][LEFT][LEFT][_UP_][_UP_][_UP_]
[_UP_][NONE][NONE][NONE][_UP_][_UP_]
[_UP_][LEFT][LEFT][LEFT][_UP_][_UP_]
number of policy iteration: 7
```

In the above result, the total number of policy iteration is 7, and is consistant with the result from value iteration. As shown above, in each policy iteration, there is a high number of fixed policy evaluations to return stable fixed-policy utilities before a policy iteration is run.



The above graph is plotted with the utility of a particular maze cell against the number of policy iterations performed. As shown above, the utility value converges as the number of iteration increases. However, the graph is not as smooth as there is a high number of fixed-policy evaluations performed in between each policy iteration.