Reinforcement Learning

Amr Mohamed Nasr (47)
Michael Raafat (57)
Mohammed Deifallah (59)

OVERVIEW

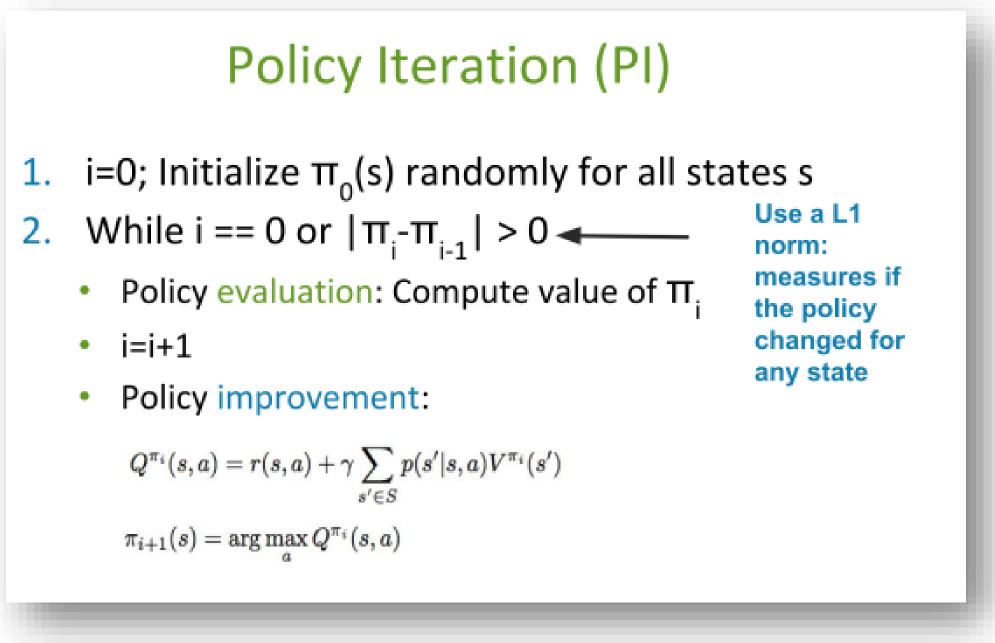
For this assignment, you will be making a ***Maze Solver***. Your program will generate a maze of size N×N. Also, you should generate barriers at random grid locations. Then you will try to learn the path out of the grid using **Policy Iteration** and **Value Iteration**.

ALGORITHMS USED

As mentioned above, the assignment has two main algorithms to be implemented and applied to reach The objective. Below are more details about them:

1) POLICY ITERATION

I. ALGORITHM

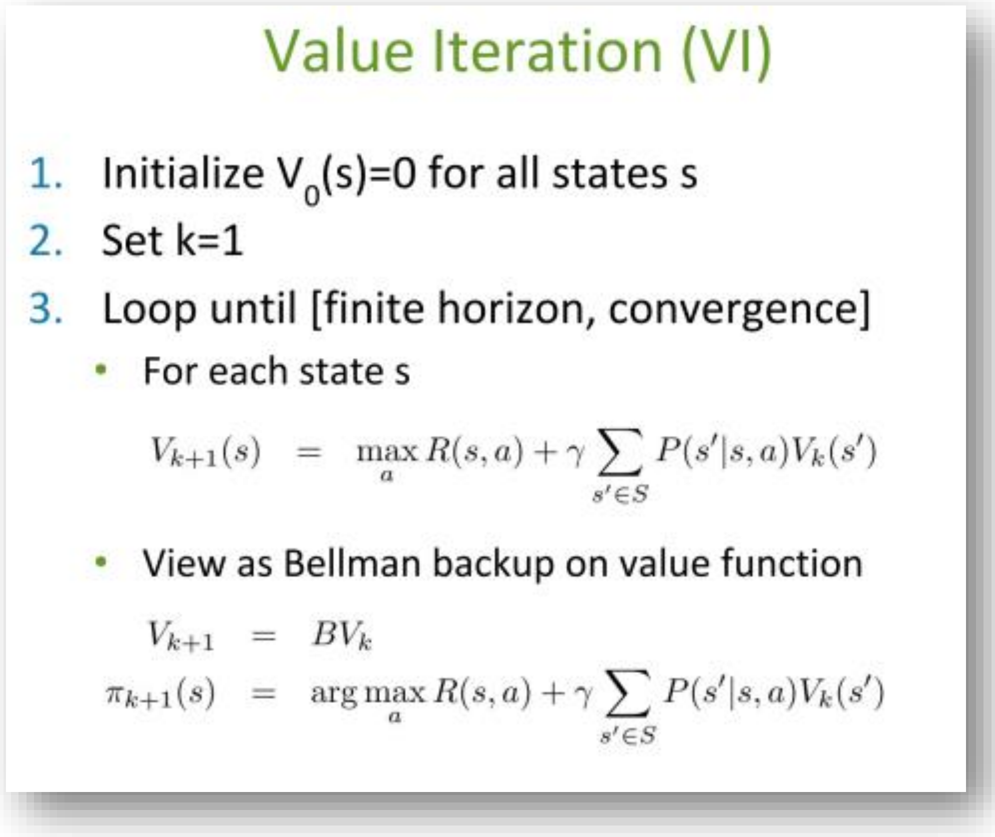


II. DATA STRUCTURE

It only uses 2D matrices to represent the maze cells with corresponding values and actions. Also, it uses a chronologically-ordered list to keep the steps for tracing issues.

2) VALUE ITERATION

I. ALGORITHM



II. DATA STRUCTURE

Again, it works with 2D matrices to represent the system.

CODE ORGANIZATION

The code is mainly divided into three main components:

1) ENVIRONMENT

1. CELL

```
private int r, c;
private Type cell_type;
private Subtype cell_subtype;
private List<Action> possible_actions;
```

It has the necessary attributes to represent each state.

Also, note that each cell has a type and subtype. The following figures show the possible categories of them:

```
public enum Type {
    START, OPEN, BARRIER, END;
}
```

```
public enum Subtype {
    MIDDLE, EDGE, CORNER;
}
```

2. GRID

It has a 2D matrix of **Cell**s and the size of the grid world.

Note that this code snippet show how the probability is being applied to decide the barrier cells.

```
private void initGrid(double r) {
    Random random = new Random();
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (grid[i][j].getType() == Type.OPEN) {
                double prob = random.nextDouble();
                if (prob < r) {
                    grid[i][j].setType(Type.BARRIER);
                }
            }
        }
    }
}
```

2) BRAIN

1. POLICY INTERFACE

```
boolean comparePolicy(Policy policy);

void setRandActions();

void setCells(Cell cell[][]);
void setActions(Action grid[][]);
void setValues(Double values[][]);

Cell[][] getGrid();
Double[][] getValues();
Action[][] getActions();
```

2. POLICY EVALUATION

```
while (diff > 1e-3 && num_loops < 1e9) {
    diff = 0.0d;
    num_loops++;
    Double[][] new_state_values = new Double[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            Cell cell = grid.getGrid()[i][j];
            Action action = actions[i][j];
            double maxVal = -Double.MAX_VALUE;
            if (action == Action.NONE) {
                maxVal = cell.getReward();
            } else {
                Point nextState = action.getTransition(cell);
                maxVal = cell.getReward() + gamma * state_values[nextState.x][nextState.y];
            }
            diff += Math.abs(state_values[i][j] - maxVal);
            new_state_values[i][j] = maxVal;
        }
    }
    state_values = new_state_values;
}
```

3. POLICIY IMPROVEMENT

The following code snippet shows the application of **Greedy Policy** to improve the current policy:

```java
Policy getGreedyPolicy(Double[][] vals, Grid grid, Double gamma) {
    Policy pol = new PolicyImp(grid.getGrid());
    pol.setValues(vals);
    Action actions[][] = new Action[grid.getSize()][grid.getSize()];
    int n = grid.getSize();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            actions[i][j] = Action.NONE;
            Cell cell = grid.getGrid()[i][j];
            double maxVal = -Double.MAX_VALUE;
            for (int k = 0; k < cell.getPossibleActions().size(); k++) {
                Point nextState = cell.getPossibleActions().get(k).getTransition(cell);
                double val = cell.getReward() + gamma * vals[nextState.x][nextState.y];
                if (val > maxVal) {
                    maxVal = val;
                    actions[i][j] = cell.getPossibleActions().get(k);
                }
            }
        }
    }
    pol.setActions(actions);
    return pol;
}
```

4. VALUE INTERFACE

```java
public interface ValueIteration {

    void value_iteration(Grid grid);
    List<Double[][]> getValueIterations();
    Policy getOptimalPolicy();
}
```

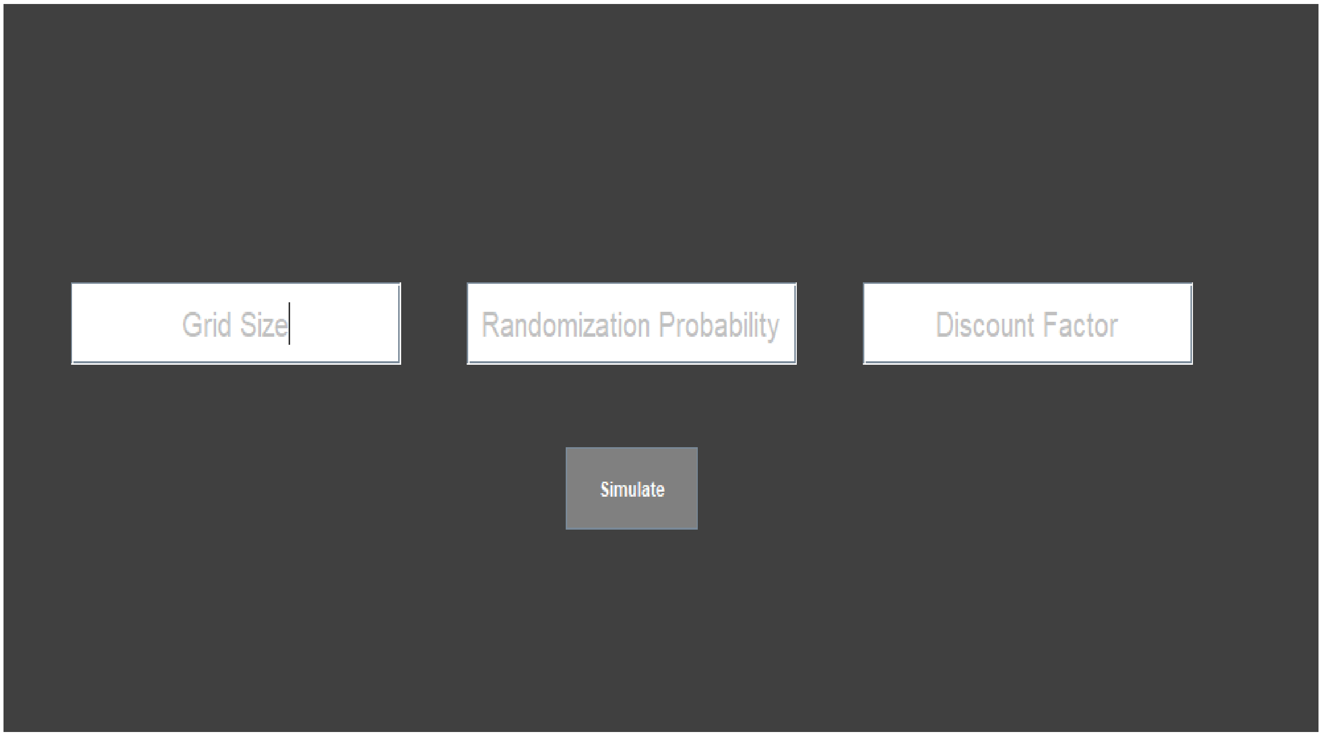
3) GUI

It’s a separate package for our visualization.

EXTRA WORK

1) We provide a GUI application, implemented in **Java Swing**, that enables the user to try our solver for different purposes:
   1- Measure the time performance.
   2- Try the various effects of parameters.

Here's a screenshot of the application intro:



2) We provide a separate implementation of DFS function that guarantees path existence between the start and end cells.

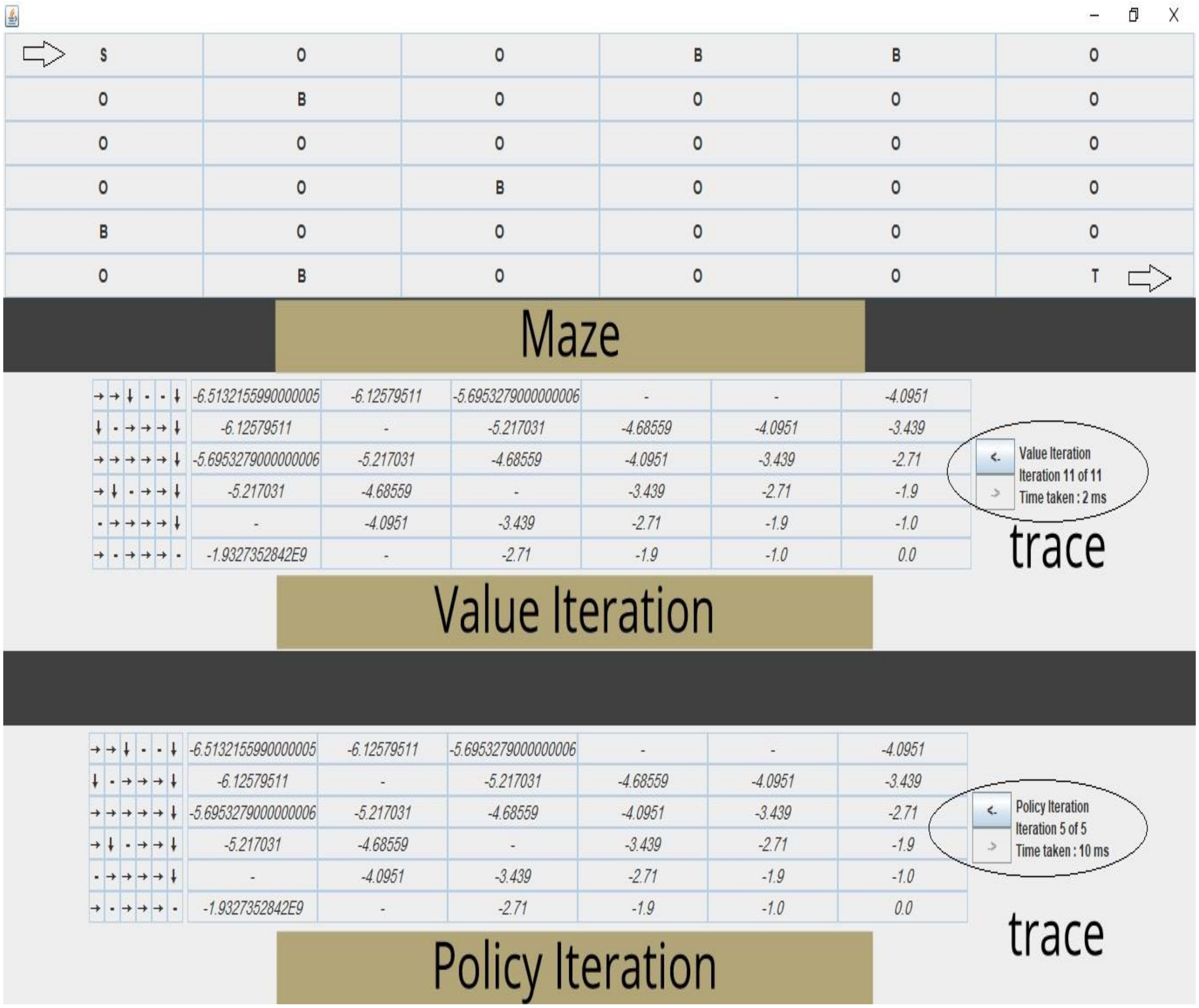
```java
public static boolean pathExists(Cell grid[][]) {
    List<Point> stack = new ArrayList<Point>();
    stack.add(grid[0][0].getPosition());
    Set<Point> visited = new HashSet<Point>();
    while(!stack.isEmpty()) {
        Point point = stack.remove(stack.size() - 1);
        visited.add(grid[point.x][point.y].getPosition());
        if (grid[point.x][point.y].getType() == Type.END) {
            return true;
        }
        // searching for valid neighbors cells
        List<Action> actions = grid[point.x][point.y].getPossibleActions();
        for (int i = 0; i < actions.size(); i++) {
            Action action = actions.get(i);
            if (!visited.contains(action.getTransition(grid[point.x][point.y]))) {
                stack.add(action.getTransition(grid[point.x][point.y]));
            }
        }

    }
    return false;

}
```

SAMPLE RUNS

The following sample run has a maze of size 6×6 with probability of .2 and discount factor of .9



S	0	0	B	B	0
0	B	0	0	0	0
0	0	0	0	0	0
0	0	B	0	0	0
B	0	0	0	0	0
0	B	0	0	0	T

→	→	↓	·	·	↓	-6.5132155990000005	-6.12579511	-5.6953279000000006	-	-	-4.0951
↓	·	→	→	→	↓	-6.12579511	-	-5.217031	-4.68559	-4.0951	-3.439
→	→	→	→	→	↓	-5.6953279000000006	-5.217031	-4.68559	-4.0951	-3.439	-2.71
→	↓	·	→	→	↓	-5.217031	-4.68559	-	-3.439	-2.71	-1.9
·	→	→	→	→	↓	-	-4.0951	-3.439	-2.71	-1.9	-1.0
→	·	→	→	→	·	-1.9327352842E9	-	-2.71	-1.9	-1.0	0.0

→	→	↓	·	·	↓	-6.5132155990000005	-6.12579511	-5.6953279000000006	-	-	-4.0951
↓	·	→	→	→	↓	-6.12579511	-	-5.217031	-4.68559	-4.0951	-3.439
→	→	→	→	→	↓	-5.6953279000000006	-5.217031	-4.68559	-4.0951	-3.439	-2.71
→	↓	·	→	→	↓	-5.217031	-4.68559	-	-3.439	-2.71	-1.9
·	→	→	→	→	↓	-	-4.0951	-3.439	-2.71	-1.9	-1.0
→	·	→	→	→	·	-1.9327352842E9	-	-2.71	-1.9	-1.0	0.0

ASSUMPTIONS & EXPLANATIONS

- The following table shows the corresponding rewards for each type of cell:

Cell Type	Reward
Start	-1
Middle	-1
Barrier	-2147483648 (minimum integer in Java)
End	0

- We only permit actions that have effect. In other words, each corner cell has only 2 actions, each edge cell has 3 actions and any other middle cell has the 4 default actions.
- The program has 3 inputs:
  1. N: The grid world size.
  2. P: the probability of barriers. $0 \leq P \leq 1$
  3. γ: The discount factor $0 \leq \gamma < 1$

     Note that the discount factor can't be 1 to prevent possible infinite loops in **Policy Iteration**.
- The algorithms terminate if any of the following conditions holds:
  1. The difference between two consecutive iterations is less than .001
  2. The number of iterations has reached 10^9
- The recommended configuration of the maze to really measure the performance is: **{N: 8, P: 0.2, γ: .9}**