Alperen UGUS
CWID: 10864101

11/25/19

ALGORITHMS CSCI 406 PROJECT 4

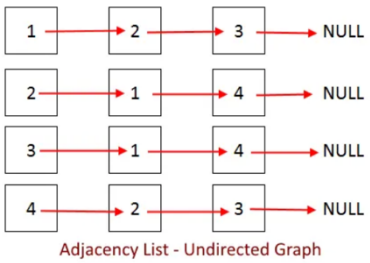
1.1 Problem Modelling

a. How to model the maze as a graph?

In order to be able to model this maze as a graph, first thing to do for me was to think about what information for each cell could be useful for Breadth First Algorithm. I decided to have 2 different nodes in the reverse direction for each cell in order to be able to move in the reverse direction if encountered to a circular cell while doing the BFS. I created a Cell class that holds row and column numbers, color, circularity, color for breadth first, direction, adjacency list, parent information.

To be able to store all the cells as nodes, I created a Graph class that has a linked list for cells. Also, I created a Maze class which is the driver one that has main method in it. I created a Graph object in the main method and started reading file. After reading each cells information from the file, I created a Cell object that holds the information and added it to the linked list in that Graph object.

Also, Graph class has a very important method which is the fundamental implementation for graphs. The "fillAdj()' method calculates all the adjacent cells to a cell using its color, direction, row and column values. I added all the adjacent cells to related cell object.



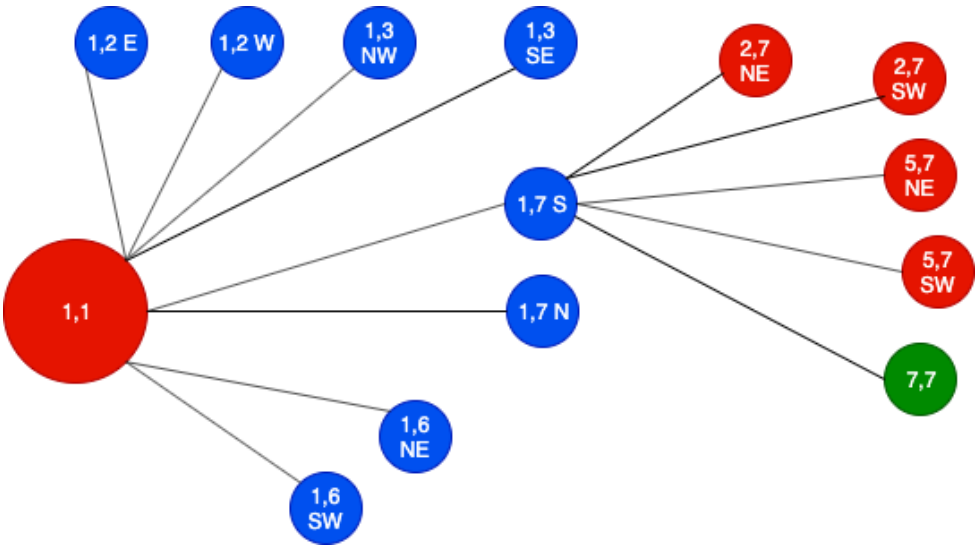
Adjacency List - Undirected Graph

By doing this, I got a structure as shown on the left and as we mentioned in the lectures. So now, the structure I have is ready for any shortest path algorithm like BFS or DSF but I have chosen BFS for my implementation. The complex part for the traversal on the graph was to choose which direction to go according to the circularity information for each cell and to track if any circular cell is encountered before. To deal with this, I have added "reversed" and "forward" flags for each Cells.

The second cells that I have created to be able to traverse back have a reversed flag as true. For the forward flag, I first assigned it true for each cell. Then before traversing in the graph, I checked circularity of the first cell and assigned forward true if it is not circular, and false if it is circular. Then for each cell to be added to the queue for BFS, I checked the previous cell's forward flag and current cell's circularity flag. Using these, I chose which direction to go.

Finally, I traversed the graph, while traversing I assigned each cell its parent. In the end, I used those parents to traceback the right path.

b. Partial Graph Model



c. Graph algorithm to solve the problem:

I have used Breadth First Search Algorithm to solve this problem since this is a verified algorithm to traverse graph data structures. It starts at a node and explores all neighbor nodes at each depth level for one node.

d. Why does this algorithm work?

I have modelled the maze as a graph. If there exists a path from the root node to the end node, then it will be traversed by BFS since BFS traverses all the nodes that are adjacent to a cell in each depth. By including all the cells' reversed direction nodes, I am covering all the possible moves within the maze. Therefore, thinking each level from the root to the end as depth levels, BFS will eventually discover the path in between these two if there is one.

1.2 Code

a. Cell Class:

```
1.  import java.util.LinkedList;
2.
3.  public class Cell {
4.      private int row;
5.      private int col;
6.      private char color;
7.      private char bfsColor;
8.      private char circular;
9.      private String direction;
10.     boolean reversed;
11.     private LinkedList<Cell> adj;
12.     private Cell parent;
13.     private boolean forward;
14.
15.     public Cell() {
16.     };
17.
18.     public Cell(int row, int col, char color, char bfsColor, char circular, String dire
    ction, boolean reversed) {
19.         super();
20.         this.row = row;
21.         this.col = col;
22.         this.color = color;
23.         this.bfsColor = bfsColor;
24.         this.circular = circular;
25.         this.direction = direction;
26.         this.reversed = reversed;
27.         this.adj = new LinkedList<Cell>();
28.         this.forward = true;
29.     }
30.
31.     public void printAdj() {
32.         for (Cell cell : adj) {
33.             System.out.print(cell.getRow());
34.             System.out.print(" ");
35.             System.out.print(cell.getCol());
36.             System.out.print("   ");
```

```
37.         }
38.         System.out.println();
39.     }
40.
41.     public int getRow() {
42.         return row;
43.     }
44.
45.     public void setRow(int row) {
46.         this.row = row;
47.     }
48.
49.     public int getCol() {
50.         return col;
51.     }
52.
53.     public void setCol(int col) {
54.         this.col = col;
55.     }
56.
57.     public char getColor() {
58.         return color;
59.     }
60.
61.     public void setColor(char color) {
62.         this.color = color;
63.     }
64.
65.     public char getBfsColor() {
66.         return bfsColor;
67.     }
68.
69.     public void setBfsColor(char bfsColor) {
70.         this.bfsColor = bfsColor;
71.     }
72.
73.     public char getCircular() {
74.         return circular;
75.     }
76.
77.     public void setCircular(char circular) {
78.         this.circular = circular;
79.     }
80.
81.     public String getDirection() {
82.         return direction;
83.     }
84.
85.     public void setDirection(String direction) {
86.         this.direction = direction;
87.     }
88.
89.     public boolean isReversed() {
90.         return reversed;
91.     }
92.
93.     public void setReversed(boolean reversed) {
94.         this.reversed = reversed;
95.     }
96.
97.     public LinkedList<Cell> getAdj() {
```

```
98.         return adj;
99.     }
100.
101.        public void setAdj(LinkedList<Cell> adj) {
102.            this.adj = adj;
103.        }
104.
105.        public Cell getParent() {
106.            return parent;
107.        }
108.
109.        public void setParent(Cell parent) {
110.            this.parent = parent;
111.        }
112.
113.        public boolean isForward() {
114.            return forward;
115.        }
116.
117.        public void setForward(boolean forward) {
118.            this.forward = forward;
119.        }
120.
121.        @Override
122.        public String toString() {
123.            return "Cell [row=" + row + ", col=" + col + ", color=" + color + ", bfs
    Color=" + bfsColor + ", circular="
124.                    + circular + ", direction=" + direction + ", reversed=" + revers
    ed + "]";
125.        }
126.
127.    }
```

b. Graph Class:

```
1.  import java.util.LinkedList;
2.
3.  public class Graph {
4.
5.      private int row;
6.      private int col;
7.
8.      private LinkedList<Cell> cells;
9.
10.     public Graph(int row, int col) {
11.         super();
12.         this.row = row;
13.         this.col = col;
14.         cells = new LinkedList<Cell>();
15.     }
16.
17.     public void addCell(Cell cell) {
18.         cells.add(cell);
19.     }
20.
21.     public Cell getCell(int row, int col, String direction) {
22.         for (Cell cell : cells) {
```

```
23.             if (cell.getRow() == row && cell.getCol() == col
24.                     && cell.getDirection().equals(direction))
25.                 return cell;
26.         }
27.
28.         return null;
29.     }
30.
31.     public void printGraph() {
32.         for (Cell cell : cells) {
33.             System.out.println(cell.toString());
34.         }
35.     }
36.
37.     public void printAdj() {
38.         for (Cell cell : cells) {
39.             System.out.print(cell.getRow());
40.             System.out.print(" ");
41.             System.out.print(cell.getCol());
42.             System.out.print(" ");
43.             System.out.print(cell.getDirection());
44.             System.out.print(" :");
45.             cell.printAdj();
46.         }
47.
48.     }
49.
50.     public void fillAdj() {
51.         for (Cell cell : cells) {
52.
53.             for (Cell subcell : cells) {
54.                 // If the row and column values and colors are not the same
55.                 if (!(cell.getRow() == subcell.getRow() && cell.getCol() == subcell.get
    Col())
56.                         && cell.getColor() != subcell.getColor()) {
57.
58.                     // Fill adjacencies according to directions
59.
60.                     if (cell.getDirection().equals("E")
61.                             && subcell.getCol() > cell.getCol()
62.                             && subcell.getRow() == cell.getRow()) {
63.                         cell.getAdj().add(subcell);
64.                     }
65.
66.                     if (cell.getDirection().equals("W")
67.                             && subcell.getCol() < cell.getCol()
68.                             && subcell.getRow() == cell.getRow()) {
69.                         cell.getAdj().add(subcell);
70.                     }
71.
72.                     if (cell.getDirection().equals("N")
73.                             && subcell.getRow() < cell.getRow()
74.                             && subcell.getCol() == cell.getCol()) {
75.                         cell.getAdj().add(subcell);
76.                     }
77.
78.                     if (cell.getDirection().equals("S")
79.                             && subcell.getRow() > cell.getRow()
80.                             && subcell.getCol() == cell.getCol()) {
81.                         cell.getAdj().add(subcell);
82.                     }
```

```
83.
84.                     if (cell.getDirection().equals("NE")
85.                             && subcell.getRow() < cell.getRow()
86.                             && subcell.getCol() > cell.getCol()
87.                             && (Math.abs(subcell.getCol() - cell.getCol())
88.                                     == Math.abs(subcell.getRow() - cell.getRow()))) {
89.                         cell.getAdj().add(subcell);
90.                     }
91.
92.                     if (cell.getDirection().equals("NW")
93.                             && subcell.getRow() < cell.getRow()
94.                             && subcell.getCol() < cell.getCol()
95.                             && (Math.abs(subcell.getCol() - cell.getCol())
96.                                     == Math.abs(subcell.getRow() - cell.getRow()))) {
97.                         cell.getAdj().add(subcell);
98.                     }
99.
100.                        if (cell.getDirection().equals("SE")
101.                                && subcell.getRow() > cell.getRow()
102.                                && subcell.getCol() > cell.getCol()
103.                                && (Math.abs(subcell.getCol() - cell.getCol())
104.                                        == Math.abs(subcell.getRow() - cell.getRow()
    ))) {
105.                            cell.getAdj().add(subcell);
106.                        }
107.
108.                        if (cell.getDirection().equals("SW")
109.                                && subcell.getRow() > cell.getRow()
110.                                && subcell.getCol() < cell.getCol()
111.                                && (Math.abs(subcell.getCol() - cell.getCol())
112.                                        == Math.abs(subcell.getRow() - cell.getRow()
    ))) {
113.                            cell.getAdj().add(subcell);
114.                        }
115.                    }
116.
117.                }
118.
119.            }
120.
121.        }
122.
123.        public int getRow() {
124.            return row;
125.        }
126.
127.        public void setRow(int row) {
128.            this.row = row;
129.        }
130.
131.        public int getCol() {
132.            return col;
133.        }
134.
135.        public void setCol(int col) {
136.            this.col = col;
137.        }
138.
139.        public LinkedList<Cell> getCells() {
140.            return cells;
141.        }
```

```
142.
143.            public void setCells(LinkedList<Cell> cells) {
144.                this.cells = cells;
145.            }
146.
147.        }
```

c. Maze Class:

```
1.  import java.io.File;
2.  import java.io.FileNotFoundException;
3.  import java.util.LinkedList;
4.  import java.util.ListIterator;
5.  import java.util.Queue;
6.  import java.util.Scanner;
7.
8.  public class Maze {
9.
10.     public static void main(String[] args) {
11.         Graph graph;
12.
13.         File file = new File("src/input.txt");
14.         Scanner scanner;
15.
16.         // Read maze from the file
17.         try {
18.             scanner = new Scanner(file);
19.             int row = scanner.nextInt();
20.             int col = scanner.nextInt();
21.
22.             // Initialize graph object
23.             graph = new Graph(row, col);
24.
25.             // Read all the maze cells from the file
26.             while (scanner.hasNext()) {
27.                 row = scanner.nextInt();
28.                 col = scanner.nextInt();
29.                 char color = scanner.next().charAt(0);
30.                 char circular = scanner.next().charAt(0);
31.                 String direction = scanner.next();
32.                 // Create 2 cells using the values from the file
33.                 Cell newCell1 = new Cell(row, col, color, 'W', circular, direction, fal
    se);
34.                 Cell newCell2 = null;
35.
36.                 // Create the second node for each cell in order to have the reversed o
    rdered
37.                 // nodes
38.                 if (direction.equals("E"))
39.                     newCell2 = new Cell(row, col, color, 'W', circular, "W", true);
40.                 else if (direction.equals("W"))
41.                     newCell2 = new Cell(row, col, color, 'W', circular, "E", true);
42.                 else if (direction.equals("N"))
43.                     newCell2 = new Cell(row, col, color, 'W', circular, "S", true);
44.                 else if (direction.equals("S"))
45.                     newCell2 = new Cell(row, col, color, 'W', circular, "N", true);
46.                 else if (direction.equals("NE"))
```

```
47.                     newCell2 = new Cell(row, col, color, 'W', circular, "SW", true);
48.                 else if (direction.equals("SW"))
49.                     newCell2 = new Cell(row, col, color, 'W', circular, "NE", true);
50.                 else if (direction.equals("NW"))
51.                     newCell2 = new Cell(row, col, color, 'W', circular, "SE", true);
52.                 else if (direction.equals("SE"))
53.                     newCell2 = new Cell(row, col, color, 'W', circular, "NW", true);
54.                 else
55.                     newCell2 = new Cell(row, col, color, 'W', circular, direction, fals
    e);
56.
57.                 // Add cells to the graph
58.                 graph.addCell(newCell1);
59.                 graph.addCell(newCell2);
60.             }
61.
62.             scanner.close();
63.
64.             // Calculate adjacency nodes for each node
65.             graph.fillAdj();
66.
67.             // graph.printAdj();
68.
69.             // BFS ALGORITHM
70.
71.             Queue<Cell> queue = new LinkedList<>();
72.             // Add the starting node to the queue
73.             queue.add(graph.getCell(1, 1, "E"));
74.             if (graph.getCell(1, 1, "E").getCircular() == 'C')
75.                 graph.getCell(1, 1, "E").setForward(false);
76.
77.             while (!queue.isEmpty()) {
78.                 Cell u = queue.peek();
79.
80.                 for (Cell cell : u.getAdj()) {
81.                     if (cell.getBfsColor() == 'W' || cell.getColor() == 'X') {
82.                         // Direction forward, upcoming cell not circular, not reversed

83.                         if (u.isForward() && cell.getCircular()
84.                                 == 'N' && !cell.isReversed()) {
85.                             cell.setBfsColor('G');
86.                             cell.setParent(u);
87.                             queue.add(cell);
88.                         }
89.                         // Direction forward, upcoming cell circular, reversed
90.                         if (u.isForward() && cell.getCircular()
91.                                 == 'C' && cell.isReversed()) {
92.                             cell.setBfsColor('G');
93.                             cell.setParent(u);
94.                             cell.setForward(!u.isForward());
95.                             queue.add(cell);
96.                         }
97.                         // Direction reversed, upcoming cell not circular, reversed
98.                         if (!u.isForward() && cell.getCircular()
99.                                 == 'N' && cell.isReversed()) {
100.                            cell.setBfsColor('G');
101.                            cell.setParent(u);
102.                            queue.add(cell);
103.                            cell.setForward(u.isForward());
104.                        }
105.                        // Direction reversed, upcoming circular, not reversed
```

```
106.                                 if (!u.isForward() && cell.getCircular()
107.                                         == 'C' && !cell.isReversed()) {
108.                                     cell.setBfsColor('G');
109.                                     cell.setParent(u);
110.                                     queue.add(cell);
111.                                     cell.setForward(!u.isForward());
112.                                 }
113.                                 // Final Cell
114.                                 if (cell.getDirection().equals("X")) {
115.                                     cell.setBfsColor('G');
116.                                     cell.setParent(u);
117.                                     queue.add(cell);
118.                                 }
119.
120.                             }
121.                         }
122.
123.         //              PRINT QUEUE
124.         //              System.out.print(queue.peek().getRow());
125.         //              System.out.print(" ");
126.         //              System.out.print(queue.peek().getCol());
127.         //              System.out.print(" ");
128.         //              System.out.print(queue.peek().getDirection());
129.         //              System.out.print(" ");
130.         //              System.out.println(queue.peek().isForward());
131.
132.                         // Remove the processed cell from the queue
133.                         queue.remove();
134.                         u.setBfsColor('B');
135.                     }
136.
137.                     Cell cell = graph.getCell(row, col, "X");
138.                     if (cell.getParent() == null) {
139.                         System.out.println("No path found!");
140.                     }
141.
142.                     LinkedList<Cell> reverseResult = new LinkedList<Cell>();
143.                     LinkedList<Cell> result = new LinkedList<Cell>();
144.
145.                     // Traceback using parents
146.                     while (cell.getParent() != null) {
147.                         reverseResult.add(cell);
148.                         // System.out.println(cell.toString());
149.                         cell = cell.getParent();
150.
151.                         if (cell.getParent() == null) {
152.                             // System.out.println(cell.toString());
153.                             reverseResult.add(cell);
154.                         }
155.                     }
156.
157.                     // Reverse the linked list so that we can have the ordered path
158.                     ListIterator listIterator = reverseResult.listIterator(reverseResult
    .size());
159.                     while (listIterator.hasPrevious()) {
160.         //              System.out.println(listIterator.previous());
161.                         result.add((Cell) listIterator.previous());
162.                     }
163.
164.                     // PRINT THE RESULTING PATH
165.                     for (Cell i : result) {
```

```
166.                     System.out.print("(" + i.getRow() + "," + i.getCol() + ") ");
167.                 }
168.
169.             } catch (FileNotFoundException e) {
170.                 // TODO Auto-generated catch block
171.                 e.printStackTrace();
172.             }
173.
174.         }
175.
176.     }
```

1.3 Results

The output of the program for the maze:

(1,1) (1,6) (5,2) (6,2) (7,2) (2,2) (4,2) (2,4) (6,4) (6,7)
(2,7) (6,3) (7,4) (5,6) (4,5) (5,5) (2,5) (3,5) (6,5) (4,3)
(4,5) (3,5) (1,5) (5,5) (6,5) (7,6) (2,1) (4,3) (4,1) (7,1)
(4,4) (1,7) (7,7)