**ALGORITHMS CSCI 406 PROJECT 4**

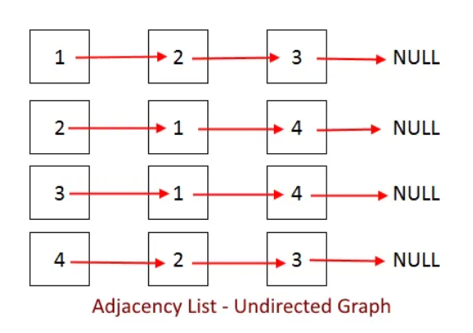
* 1. **Problem Modelling**

1. **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.

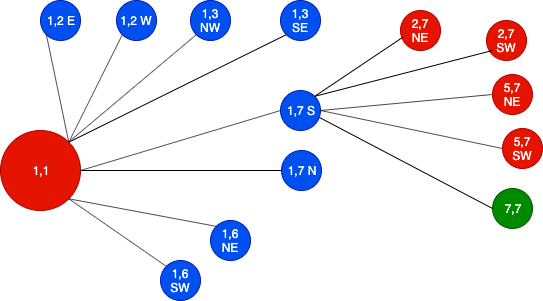


By doing this, I got a structure as shown on the left and as we mentioned in the lectures. So now, the structure I have is ready for any shortest path algorithm like BFS or DSF but I have chosen BFS for my implementation. The complex part for the traversal on the graph was to choose which direction to 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.

1. **Partial Graph Model**



1. **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.

1. **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. **Code**

1. **Cell Class:**
2. **import** java.util.LinkedList;
4. **public** **class** Cell {
5. **private** **int** row;
6. **private** **int** col;
7. **private** **char** color;
8. **private** **char** bfsColor;
9. **private** **char** circular;
10. **private** String direction;
11. **boolean** reversed;
12. **private** LinkedList<Cell> adj;
13. **private** Cell parent;
14. **private** **boolean** forward;
16. **public** Cell() {
17. };
19. **public** Cell(**int** row, **int** col, **char** color, **char** bfsColor, **char** circular, String direction, **boolean** reversed) {
20. **super**();
21. **this**.row = row;
22. **this**.col = col;
23. **this**.color = color;
24. **this**.bfsColor = bfsColor;
25. **this**.circular = circular;
26. **this**.direction = direction;
27. **this**.reversed = reversed;
28. **this**.adj = **new** LinkedList<Cell>();
29. **this**.forward = **true**;
30. }
32. **public** **void** printAdj() {
33. **for** (Cell cell : adj) {
34. System.out.print(cell.getRow());
35. System.out.print(" ");
36. System.out.print(cell.getCol());
37. System.out.print("   ");
38. }
39. System.out.println();
40. }
42. **public** **int** getRow() {
43. **return** row;
44. }
46. **public** **void** setRow(**int** row) {
47. **this**.row = row;
48. }
50. **public** **int** getCol() {
51. **return** col;
52. }
54. **public** **void** setCol(**int** col) {
55. **this**.col = col;
56. }
58. **public** **char** getColor() {
59. **return** color;
60. }
62. **public** **void** setColor(**char** color) {
63. **this**.color = color;
64. }
66. **public** **char** getBfsColor() {
67. **return** bfsColor;
68. }
70. **public** **void** setBfsColor(**char** bfsColor) {
71. **this**.bfsColor = bfsColor;
72. }
74. **public** **char** getCircular() {
75. **return** circular;
76. }
78. **public** **void** setCircular(**char** circular) {
79. **this**.circular = circular;
80. }
82. **public** String getDirection() {
83. **return** direction;
84. }
86. **public** **void** setDirection(String direction) {
87. **this**.direction = direction;
88. }
90. **public** **boolean** isReversed() {
91. **return** reversed;
92. }
94. **public** **void** setReversed(**boolean** reversed) {
95. **this**.reversed = reversed;
96. }
98. **public** LinkedList<Cell> getAdj() {
99. **return** adj;
100. }
102. **public** **void** setAdj(LinkedList<Cell> adj) {
103. **this**.adj = adj;
104. }
106. **public** Cell getParent() {
107. **return** parent;
108. }
110. **public** **void** setParent(Cell parent) {
111. **this**.parent = parent;
112. }
114. **public** **boolean** isForward() {
115. **return** forward;
116. }
118. **public** **void** setForward(**boolean** forward) {
119. **this**.forward = forward;
120. }
122. @Override
123. **public** String toString() {
124. **return** "Cell [row=" + row + ", col=" + col + ", color=" + color + ", bfsColor=" + bfsColor + ", circular="
125. + circular + ", direction=" + direction + ", reversed=" + reversed + "]";
126. }
128. }
129. **Graph Class:**
130. **import** java.util.LinkedList;
132. **public** **class** Graph {
134. **private** **int** row;
135. **private** **int** col;
137. **private** LinkedList<Cell> cells;
139. **public** Graph(**int** row, **int** col) {
140. **super**();
141. **this**.row = row;
142. **this**.col = col;
143. cells = **new** LinkedList<Cell>();
144. }
146. **public** **void** addCell(Cell cell) {
147. cells.add(cell);
148. }
150. **public** Cell getCell(**int** row, **int** col, String direction) {
151. **for** (Cell cell : cells) {
152. **if** (cell.getRow() == row && cell.getCol() == col
153. && cell.getDirection().equals(direction))
154. **return** cell;
155. }
157. **return** **null**;
158. }
160. **public** **void** printGraph() {
161. **for** (Cell cell : cells) {
162. System.out.println(cell.toString());
163. }
164. }
166. **public** **void** printAdj() {
167. **for** (Cell cell : cells) {
168. System.out.print(cell.getRow());
169. System.out.print(" ");
170. System.out.print(cell.getCol());
171. System.out.print(" ");
172. System.out.print(cell.getDirection());
173. System.out.print(" :");
174. cell.printAdj();
175. }
177. }
179. **public** **void** fillAdj() {
180. **for** (Cell cell : cells) {
182. **for** (Cell subcell : cells) {
183. // If the row and column values and colors are not the same
184. **if** (!(cell.getRow() == subcell.getRow() && cell.getCol() == subcell.getCol())
185. && cell.getColor() != subcell.getColor()) {
187. // Fill adjacencies according to directions
189. **if** (cell.getDirection().equals("E")
190. && subcell.getCol() > cell.getCol()
191. && subcell.getRow() == cell.getRow()) {
192. cell.getAdj().add(subcell);
193. }
195. **if** (cell.getDirection().equals("W")
196. && subcell.getCol() < cell.getCol()
197. && subcell.getRow() == cell.getRow()) {
198. cell.getAdj().add(subcell);
199. }
201. **if** (cell.getDirection().equals("N")
202. && subcell.getRow() < cell.getRow()
203. && subcell.getCol() == cell.getCol()) {
204. cell.getAdj().add(subcell);
205. }
207. **if** (cell.getDirection().equals("S")
208. && subcell.getRow() > cell.getRow()
209. && subcell.getCol() == cell.getCol()) {
210. cell.getAdj().add(subcell);
211. }
213. **if** (cell.getDirection().equals("NE")
214. && subcell.getRow() < cell.getRow()
215. && subcell.getCol() > cell.getCol()
216. && (Math.abs(subcell.getCol() - cell.getCol())
217. == Math.abs(subcell.getRow() - cell.getRow()))) {
218. cell.getAdj().add(subcell);
219. }
221. **if** (cell.getDirection().equals("NW")
222. && subcell.getRow() < cell.getRow()
223. && subcell.getCol() < cell.getCol()
224. && (Math.abs(subcell.getCol() - cell.getCol())
225. == Math.abs(subcell.getRow() - cell.getRow()))) {
226. cell.getAdj().add(subcell);
227. }
229. **if** (cell.getDirection().equals("SE")
230. && subcell.getRow() > cell.getRow()
231. && subcell.getCol() > cell.getCol()
232. && (Math.abs(subcell.getCol() - cell.getCol())
233. == Math.abs(subcell.getRow() - cell.getRow()))) {
234. cell.getAdj().add(subcell);
235. }
237. **if** (cell.getDirection().equals("SW")
238. && subcell.getRow() > cell.getRow()
239. && subcell.getCol() < cell.getCol()
240. && (Math.abs(subcell.getCol() - cell.getCol())
241. == Math.abs(subcell.getRow() - cell.getRow()))) {
242. cell.getAdj().add(subcell);
243. }
244. }
246. }
248. }
250. }
252. **public** **int** getRow() {
253. **return** row;
254. }
256. **public** **void** setRow(**int** row) {
257. **this**.row = row;
258. }
260. **public** **int** getCol() {
261. **return** col;
262. }
264. **public** **void** setCol(**int** col) {
265. **this**.col = col;
266. }
268. **public** LinkedList<Cell> getCells() {
269. **return** cells;
270. }
272. **public** **void** setCells(LinkedList<Cell> cells) {
273. **this**.cells = cells;
274. }
276. }
277. **Maze Class:**
278. **import** java.io.File;
279. **import** java.io.FileNotFoundException;
280. **import** java.util.LinkedList;
281. **import** java.util.ListIterator;
282. **import** java.util.Queue;
283. **import** java.util.Scanner;
285. **public** **class** Maze {
287. **public** **static** **void** main(String[] args) {
288. Graph graph;
290. File file = **new** File("src/input.txt");
291. Scanner scanner;
293. // Read maze from the file
294. **try** {
295. scanner = **new** Scanner(file);
296. **int** row = scanner.nextInt();
297. **int** col = scanner.nextInt();
299. // Initialize graph object
300. graph = **new** Graph(row, col);
302. // Read all the maze cells from the file
303. **while** (scanner.hasNext()) {
304. row = scanner.nextInt();
305. col = scanner.nextInt();
306. **char** color = scanner.next().charAt(0);
307. **char** circular = scanner.next().charAt(0);
308. String direction = scanner.next();
309. // Create 2 cells using the values from the file
310. Cell newCell1 = **new** Cell(row, col, color, 'W', circular, direction, **false**);
311. Cell newCell2 = **null**;
313. // Create the second node for each cell in order to have the reversed ordered
314. // nodes
315. **if** (direction.equals("E"))
316. newCell2 = **new** Cell(row, col, color, 'W', circular, "W", **true**);
317. **else** **if** (direction.equals("W"))
318. newCell2 = **new** Cell(row, col, color, 'W', circular, "E", **true**);
319. **else** **if** (direction.equals("N"))
320. newCell2 = **new** Cell(row, col, color, 'W', circular, "S", **true**);
321. **else** **if** (direction.equals("S"))
322. newCell2 = **new** Cell(row, col, color, 'W', circular, "N", **true**);
323. **else** **if** (direction.equals("NE"))
324. newCell2 = **new** Cell(row, col, color, 'W', circular, "SW", **true**);
325. **else** **if** (direction.equals("SW"))
326. newCell2 = **new** Cell(row, col, color, 'W', circular, "NE", **true**);
327. **else** **if** (direction.equals("NW"))
328. newCell2 = **new** Cell(row, col, color, 'W', circular, "SE", **true**);
329. **else** **if** (direction.equals("SE"))
330. newCell2 = **new** Cell(row, col, color, 'W', circular, "NW", **true**);
331. **else**
332. newCell2 = **new** Cell(row, col, color, 'W', circular, direction, **false**);
334. // Add cells to the graph
335. graph.addCell(newCell1);
336. graph.addCell(newCell2);
337. }
339. scanner.close();
341. // Calculate adjacency nodes for each node
342. graph.fillAdj();
344. // graph.printAdj();
346. // BFS ALGORITHM
348. Queue<Cell> queue = **new** LinkedList<>();
349. // Add the starting node to the queue
350. queue.add(graph.getCell(1, 1, "E"));
351. **if** (graph.getCell(1, 1, "E").getCircular() == 'C')
352. graph.getCell(1, 1, "E").setForward(**false**);
354. **while** (!queue.isEmpty()) {
355. Cell u = queue.peek();
357. **for** (Cell cell : u.getAdj()) {
358. **if** (cell.getBfsColor() == 'W' || cell.getColor() == 'X') {
359. // Direction forward, upcoming cell not circular, not reversed
360. **if** (u.isForward() && cell.getCircular()
361. == 'N' && !cell.isReversed()) {
362. cell.setBfsColor('G');
363. cell.setParent(u);
364. queue.add(cell);
365. }
366. // Direction forward, upcoming cell circular, reversed
367. **if** (u.isForward() && cell.getCircular()
368. == 'C' && cell.isReversed()) {
369. cell.setBfsColor('G');
370. cell.setParent(u);
371. cell.setForward(!u.isForward());
372. queue.add(cell);
373. }
374. // Direction reversed, upcoming cell not circular, reversed
375. **if** (!u.isForward() && cell.getCircular()
376. == 'N' && cell.isReversed()) {
377. cell.setBfsColor('G');
378. cell.setParent(u);
379. queue.add(cell);
380. cell.setForward(u.isForward());
381. }
382. // Direction reversed, upcoming circular, not reversed
383. **if** (!u.isForward() && cell.getCircular()
384. == 'C' && !cell.isReversed()) {
385. cell.setBfsColor('G');
386. cell.setParent(u);
387. queue.add(cell);
388. cell.setForward(!u.isForward());
389. }
390. // Final Cell
391. **if** (cell.getDirection().equals("X")) {
392. cell.setBfsColor('G');
393. cell.setParent(u);
394. queue.add(cell);
395. }
397. }
398. }
400. //              PRINT QUEUE
401. //              System.out.print(queue.peek().getRow());
402. //              System.out.print(" ");
403. //              System.out.print(queue.peek().getCol());
404. //              System.out.print(" ");
405. //              System.out.print(queue.peek().getDirection());
406. //              System.out.print(" ");
407. //              System.out.println(queue.peek().isForward());
409. // Remove the processed cell from the queue
410. queue.remove();
411. u.setBfsColor('B');
412. }
414. Cell cell = graph.getCell(row, col, "X");
415. **if** (cell.getParent() == **null**) {
416. System.out.println("No path found!");
417. }
419. LinkedList<Cell> reverseResult = **new** LinkedList<Cell>();
420. LinkedList<Cell> result = **new** LinkedList<Cell>();
422. // Traceback using parents
423. **while** (cell.getParent() != **null**) {
424. reverseResult.add(cell);
425. // System.out.println(cell.toString());
426. cell = cell.getParent();
428. **if** (cell.getParent() == **null**) {
429. // System.out.println(cell.toString());
430. reverseResult.add(cell);
431. }
432. }
434. // Reverse the linked list so that we can have the ordered path
435. ListIterator listIterator = reverseResult.listIterator(reverseResult.size());
436. **while** (listIterator.hasPrevious()) {
437. //              System.out.println(listIterator.previous());
438. result.add((Cell) listIterator.previous());
439. }
441. // PRINT THE RESULTING PATH
442. **for** (Cell i : result) {
443. System.out.print("(" + i.getRow() + "," + i.getCol() + ") ");
444. }
446. } **catch** (FileNotFoundException e) {
447. // TODO Auto-generated catch block
448. e.printStackTrace();
449. }
451. }
453. }
     1. **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)