Laboratory #7 Solving the Maze

Ivan Gastelum

Data Structures 2302

Professor Olac Fuentes

TA: Nath Anindita

**INTRODUCTION:**

From the previous laboratory where we need to create a maze out of a disjoint set forest, now are using the concept of breadth first search and depth first search to solve go from a start to an end. From previous lab #6, the way to build a maze where all cell were accessible, the number of walls removed ‘n’ – 1 must have equal the number of total cells in the maze. If we were to have n-1<m, not all cells will be accessible, and on the other side if n-1>m, we would have a maze that has multiple ways of going to some destinations. Thus, this laboratory starts by demonstrating this truth in question No. 1, then question 2 we will use our graph knowledge to create an adjacency list of the maze. 3rd question we implement the breadth first search and depth first search to reach to the last cell of the maze.

The following are the instructions of the laboratory:

1. Modify your maze-building program to allow for both cases mentioned above. Your program should display n, the number of cells, and ask the user for m, the number of walls to remove, then display a message indicating one of the following:

(a) A path from source to destination is not guaranteed to exist (when m < n − 1)

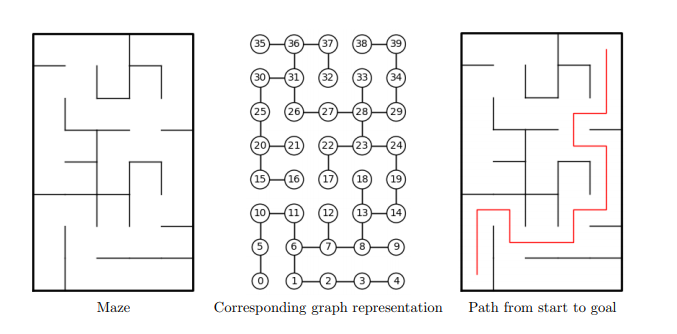
(b) The is a unique path from source to destination (when m = n − 1)

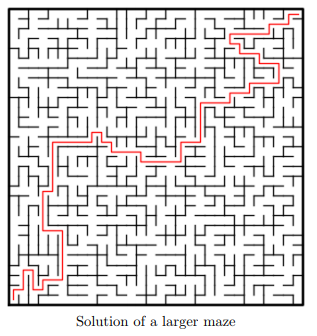
(c) There is at least one path from source to destination (when m > n − 1)

2. Write a method to build the adjacency list representation of your maze. Cells in the maze should be represented by vertices in the graph. If two cells u and v are contiguous and there is no wall separating them, then there must be an edge from u to v in the graph. The example below shows a maze and the corresponding graph representation.

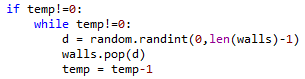
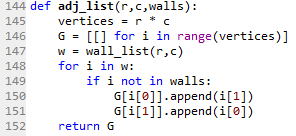
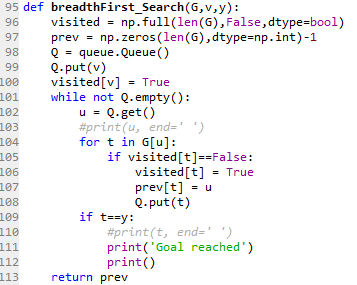
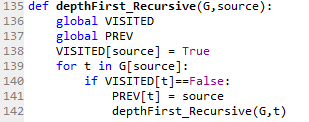
3. Implement the following algorithms to solve the maze you created, assuming the starting position is bottom-left corner and the goal position is the top-right corner. (a) Breadth-first search. (b) Depth-first search using a stack. This is identical to breadth-first search but the queue is replaced by a stack. (c) Depth-first search using recursion.

4. As usual, write a report describing your results. Display the paths found by your algorithms and compare their running times for different maze sizes.





**PROPOSED DESIGN SOLUTION AND IMPLEMENTATION:**

1. The modification of the maze code to demonstrate the three conditions is as follows: I asked the user to choose from two options, ‘1. By selecting number of walls to remove’, and ‘Creating a standard maze’. If the user select option 1, then the user can type the number of walls he wants to remove, and with a while loop with two conditions: as long as m is not 0 and the dsf is greater than 1, we will be randomly removing walls in the correct standard way as the previous code from lab 6. Now whether m reaches 0 before having a dsf of 1, this will mean that the user has chosen to remove walls less than the number of cells, if not, I created another condition so that the maze can randomly remove more walls even if the maze has been completely accessible.  
     
   
2. In order to create an adjacency list we first have to understand the concept of what an adjacency list is, and what its values represent: An adjacency list is a list of lists by which they represent the source vertex, and the values inside each list represent the destination vertex, meaning if the second sub-list of an adjacency list has the value 2, it means that vertex 1 points to 2, because the second sub-list carries the index 1 followed by 0. So, in order to facilitate and translate the walls list, I created another wall list that’s completely new, and compared with the modified walls list after building the maze. By doing this we can append all those walls that are not in the modified walls list, and append the second values which comes to be the destination at each respective list. Code as shown below:  
     
   
3. Implementing codes for breadth first search and depth first search was a challenge:  
   (a) Breadth First Search: follows a specific algorithm that can be translated into a Python code, we first create a Boolean list that checks when a cell has been visited, and an empty prev list with -1 numbers, and last we create a Queue to be putting all values as we traverse through each cell of the maze. At the end, we return the prev list which has all previous cell indexes at each index:  
     
     
   (b) Depth First Search: Usually is the same concept but using a Stack instead of a Queue  
   (c) Deph First Search Recursively: constantly calls the cell that’s in the source of the adjacency list, and taking the visited and prev lists from previous as a global variables, to not lose track of the values. Code shown below:  
     
    

**EXPERIMENTAL RESULTS:**

Here is a Table showing the running time complexities for the three methods used, in milliseconds at different maze sizes in rows and columns.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Running Time Complexity (miliseconds) | | |
| Maze Size | Breadth First Search | Depth First Search | Recursion DFS |
| 10,15 | 4.99 | 1.001 | 1.001 |
| 20,30 | 24.986 | 3.996 | 3.95 |
| 25,35 | 17.987 | 48.972 | 48.987 |
| 40, 60 | 50.97 | 18.98 | 19 |
| 60, 80 | 95.941 | 31.982 | 31.982 |

**CONCLUSION:**

In conclusion, breadth first search according to my results, it is the least efficient way to solve the maze as it came out to have the longest running time, compared to Depth first search in the two ways it can be encoded. For Depth First Search using a Stack and using recursion, both times seems to be very identical.

**APPENDIX:**

