Laboratory #6 Building a Maze

Ivan Gastelum

CS 2302 Data Structures

Professor Olac Fuentes

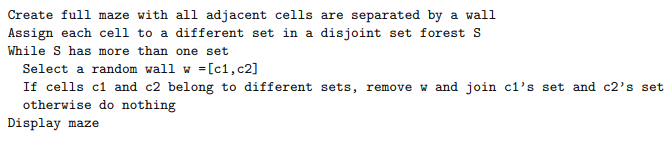
TA: Nath Anindita

April 17th 2019

**INTRODUCTION:**

This laboratory’s objective is to fix and create a maze that will always be possible to go across from point A to point B. This can be possible by using the concept of a disjoint set forest. A disjoint set forest is a number of sets in which each value of each set are joined and share the same root. A root is basically a value or key item that has no parent. Using this concept a matrix with closed cells or walls can be transformed into a maze by using the union functions that makes each item of one set be joined to another set and share the same root, this would eventually be represented as a path in the maze.

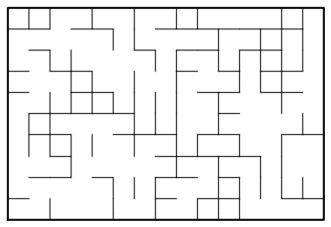
In order to reach the objective explained above, it was necessary to follow this pseudocode:



Instructions: To build a maze, you must do the following. Let M be the number of rows and N be the number of columns of your square maze. When all walls are present (see figure), each of the M ∗ N cells in the maze belongs to a different set. Thus you have M ∗ N sets in your disjoint set forest. When you remove a wall, if the cells that were separated by that wall belonged to different sets, you must unite these sets. This process is repeated until all cells belong to a single set; at that point you display the maze.

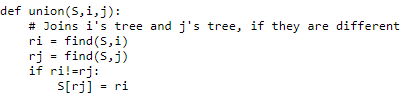
**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION:**

Before beginning to fix a maze, a code was given to us as a base that already draws the maze, prints the numbers of cells, and the walls. Everything is done through a list of walls where each list-wall gives the two cells adjacent to it; example: w = [c1,c2]. Inside the code, we were also provided with a preliminary code that loops randomly at each wall-list and removes it from the array. This results as this, in the following figure:



Although this looks like the objective has been reached, this is not a real maze as there are some cells that appear to be inaccessible. And as it was stated that the user should be able to point out any cell in the maze as end point and be connected to the start point.

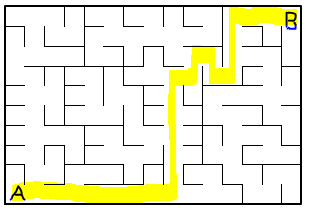
Now we can solve such problem by using the union method from a disjoint set forest:



In this case, i and j can be a set of cells that point to a parent root. But we have to create first of all, a disjoint set forest ‘S’ of same size as the walls sets, and initialize the set by having all items in -1, meaning that all will be roots.

A method to create a random maze every time the program is run can be done by a while loop that randomly pick a cell wall and joins the first cell to the second one if they don’t share the same set of one root. We first have to find the root of each cell, and with a conditional check if they are the same, if not, use the union function, and then repeat the process until we hit the while condition which can be that we have one whole set; at this point we can determine that all cells share a root, and every cell has access to some other random cell. See appendix for reference.

By doing this we obtain the desired output:



It can be shown if we select a point A and a point B as an example shown in the figure above, we can actually find a way to connect both.

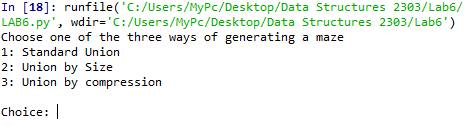
**EXPERIMENTAL RESULTS:**

My Python code was developed to have three different methods of creating the maze.

Methods of generating the maze:

* ***Standard Union function***  
  Joins the cells by the standard union method shown above. Simplest method.
* ***Union by Size***Joins the cells by checking which set is bigger, and based on this, the smaller set will join to the bigger one
* ***Union by compression***Joins the cells through compression, meaning that each item will be change to join directly to the root, instead of having multiple nodes that lead to a root.

My code begins showing this menu option:



Choosing each option, the output is the same, but their running times differ as follows:

**Running Time Complexities:**

|  |  |
| --- | --- |
| Standard Union | |
| Trials | Running Time (miliseconds) |
| 1 | 364.774 |
| 2 | 224.861 |
| 3 | 111.93 |
| 4 | 227.857 |
| 5 | 179.886 |
| 6 | 214.134 |
| 7 | 406.748 |
| 8 | 258.838 |
| 9 | 112.92 |
| 10 | 219.14 |
| average | 232.1088 |

|  |  |
| --- | --- |
| Union by Size | |
| Trials | Running Time (miliseconds) |
| 1 | 114.193 |
| 2 | 167.89 |
| 3 | 188.8 |
| 4 | 185.88 |
| 5 | 90.93 |
| 6 | 204.8 |
| 7 | 210.868 |
| 8 | 95.94 |
| 9 | 94.94 |
| 10 | 83.94 |
| average | 143.8181 |

|  |  |
| --- | --- |
| Union by Compression | |
| Trials | Running Time (miliseconds) |
| 1 | 161.899 |
| 2 | 97.93 |
| 3 | 152.905 |
| 4 | 95.941 |
| 5 | 101.934 |
| 6 | 130.918 |
| 7 | 97.938 |
| 8 | 146.909 |
| 9 | 158.9 |
| 10 | 145.9 |
| average | 129.1174 |

**CONCLUSIONS:**

Above we can see each Union functions, and their running time complexities done in 10 trials in order to check their variations. The values ranges from 80 to 300, and its most efficient method came to be Union by Compression by almost half the running time of the standard union.

**APPENDIX:**

