**Lab Objective**

For this lab, I’m improving the program from lab 6 and this time I need to use the graph functions for Lab 7. This time I first need to ask for the user inputs on how many walls they, should remove and based on that input I should be able to remove the number of walls and print out that if the removal of walls is greater than the number of walls (when m < n - 1) then it should print that the path may not exist. If the removal walls are equal to the number of walls(when m = n - 1) then I need to print that there is one path to the end of the maze. If the removed walls are less than the number of walls (when m > n - 1) then there should be at least one path from the source to destination. Also, I need to modify the remove function to remove the walls from the dsf from lab 6.

**Proposed Solution**

For this part of the program, I knew that there will be the need to modify to program from lab 6 in order to be used for lab 7. To begin I need to ask for the users input on the number of walls they want to remove and based on their input one of the 3 option will be printed out which depends on the number of walls removed. After the program gets the users input then, now I need to modify the remove method I have and adjust it to remove walls based on the number we need to remove.

**Implementation**

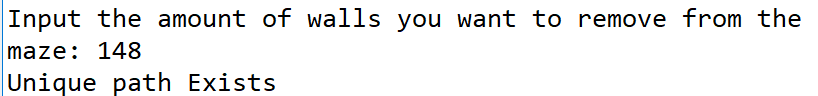
To begin my Implementation, I first asked the user what number of walls they wanted to remove from the graph, but since I’m asking for a number I need to convert their input which is a string into an integer. After converting the data type of the users input, I made a method and called it in the main where I pass the number of cells from the previous dsf from lab 6 and I pass the number of walls we want to remove.

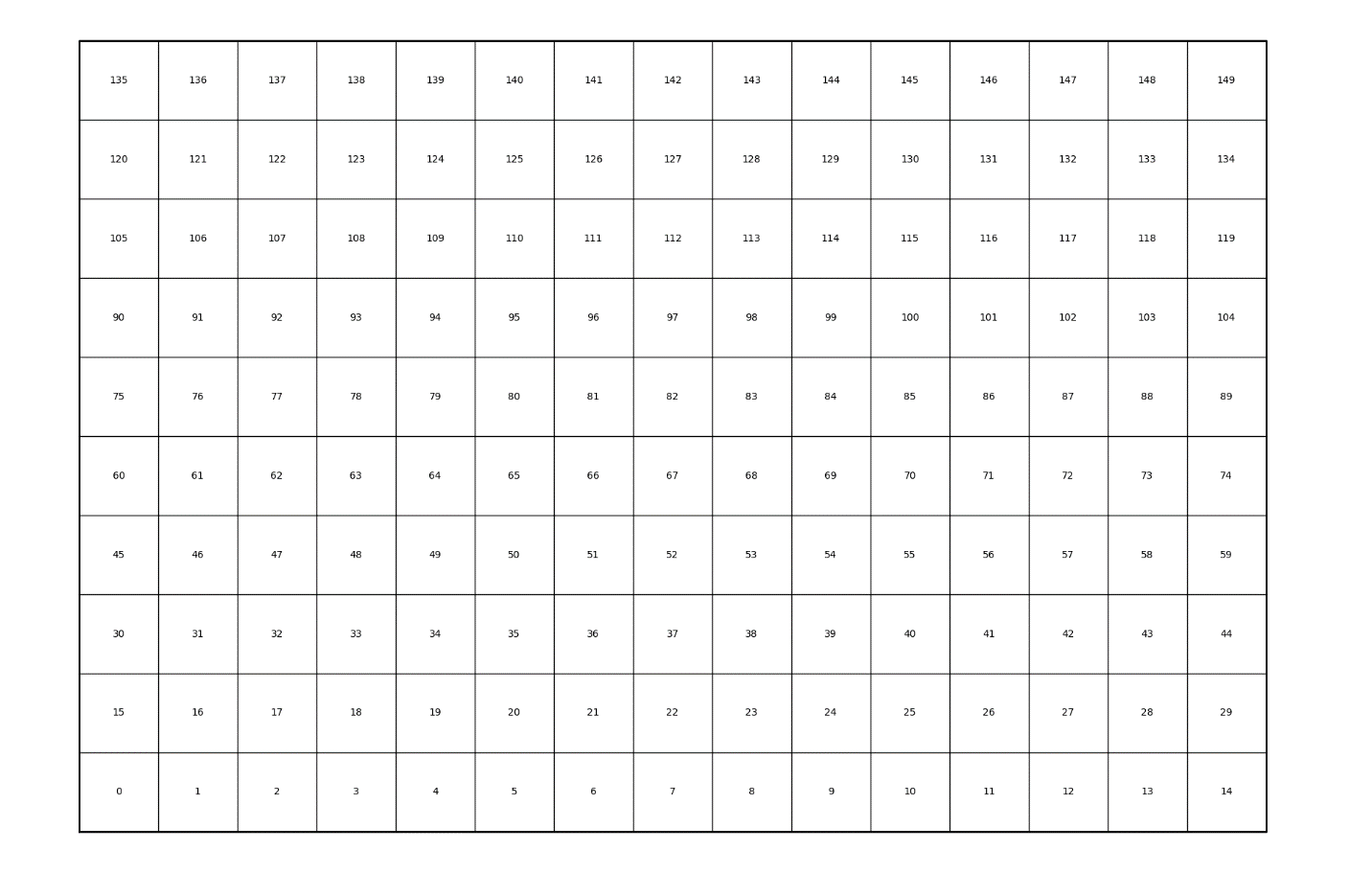
Inside the method, I created an if statement that if the number of walls to remove is less than the number of walls I have then the path should not exist. An elif was made that if the number of walls and removed walls are equal then there is a unique path. Another elif was made that if the number of walls to remove is greater than the number of walls we have then there is a at least one path to take.

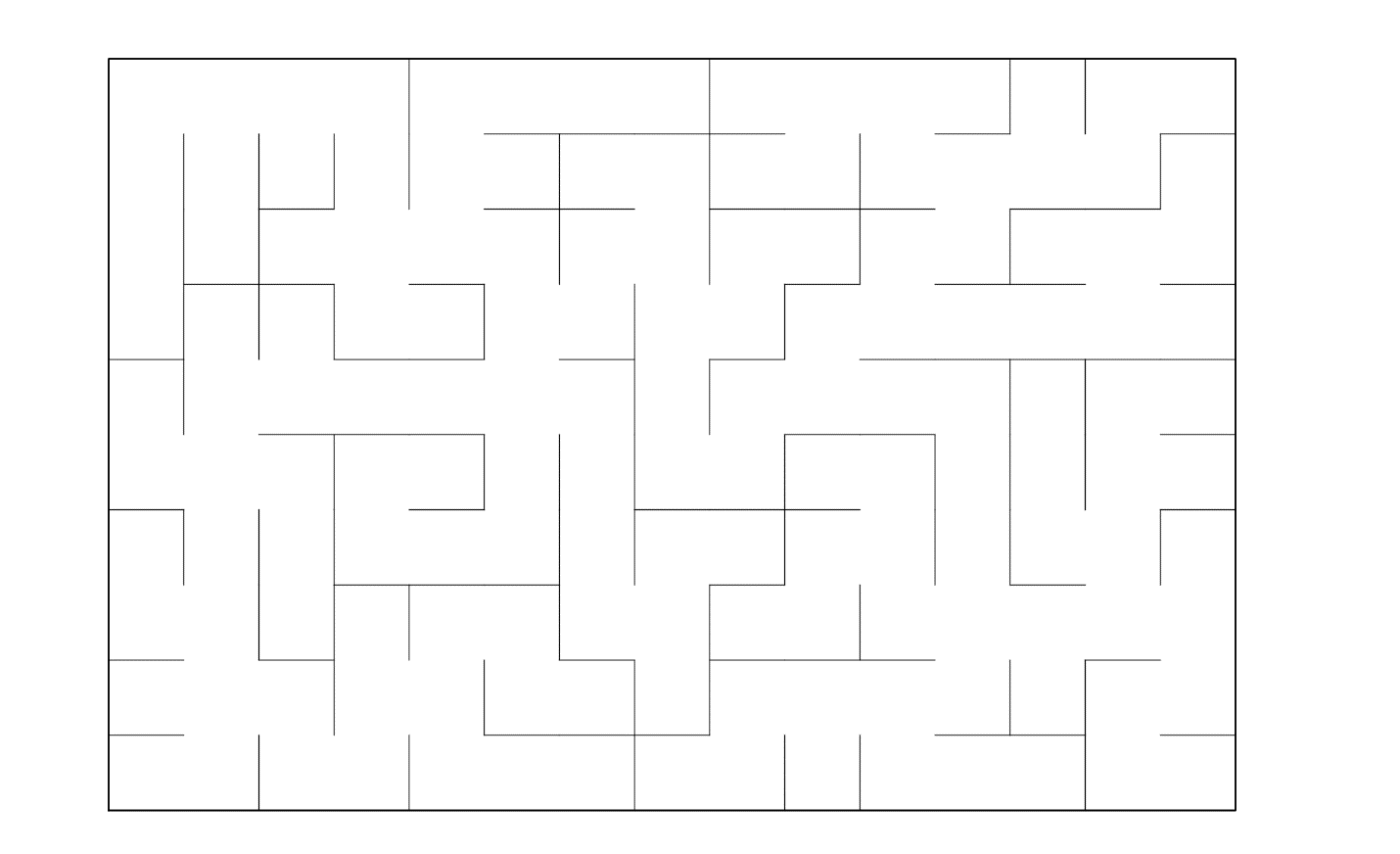
After making that method, I had to modify my remove method by moving the method into the main in order to be used later on in the program. In my remove function I changed the while loop to iterate until the users input to remove the walls is 0. The way the function exits out the while loop is by decreasing the number of walls we have to remove each time we go inside the if statement.

**Tracing**

To trace this program, I will use the maze rows of 10 and maze columns of 15. When the program begins the user will be asked to use input the number of walls they want to remove, and, in this case, I will input 148. After storing the number of walls to remove the code proceeds to use the dsf from lab 6 and pass the values of the rows and columns from the maze which gives us a total of 149 cells. After that the program calls the method to display the results of the users input and it passes the number of cells we have in the set/maze and passes the number of walls to remove. Since the user’s input is exact to the number of walls then it prints out that a path exists and continues to remove the number of walls that was inserted by the user and then draws the maze without the removed walls and the one with the removed walls.







**Question 2 objective**

For this part of the program I need to get the values of the maze and create it to an adjacency list.

**Proposed solution**

For this part I had two solutions, one of them was for get all the values from the dsf and append them to an empty list. However, that partially worked for this question, but it later proved to not work for the next part of the lab.

The second approach was to create a graph class in my lab which will contain the functions to create a graph. The functions I need is to be able to add an edge between the points that were removed walls.

**Implementation**

To implement the code, I’ve first made a class called Graph which will be using to get the graph functions. In the class I made a method that takes the self and vertices as parameters. In the method I declare self. Vertices to be equal to the vertices we pass, and we declare self. Graph to an empty list where we will use later. After the declarations were made, I needed to make a for loop where I traverse the number of vertices I have, and, in the loop, I need to append the list.

After making the method I then created another method that depended on the class. In this method I needed to add the edge/connection between the walls that were removed and to do that I needed to use the graph, the wall that was removed at the 0 index and the wall that was removed at the 1 index. Also, to get these walls I need to call the method in the removal function in my main in order to work. Back to my method, I needed to use the graph at the index of wall at 0 and append the wall at the first index.

**Tracing**

In this we will use size 10 of rows and size 15 of columns that will be used for the maze. To be able to use these dimensions I needed to make a variable to store the graph, but to make the graph I needed to pass those sizes by multiplying them and passing them to the graph class.

Once the graph is made I then go into the remove function that was described how it worked earlier. Inside the removal of the walls, inside the if statement and after the union call, the add edge call method is called while passing the new graph, the two walls that had a wall removed in between needed to have an edge connecting them, which is done in the method call. The output will be shown in the last question.

**Question 3**

For this part of the program I need to make two different codes in which both finds the path from the cell 0 to the final cell but one has to be breadth first search and the other one has to be depth first search. Also, the depth first search I need to make it with recursion

**Proposed solution**

For this part I had to follow the java code that was given during class, but I had to do some edits to it since I had made a graph class. For breadth first search(BFS) I had to pass the graph and the vertex we are starting at. Then I need to keep track if I visited a vertex in the graph and I also needed to create a queue since BFS depends on a queue. In BFS I need to keep track of what vertexes have been visited so I need to declare the vertexes false, because they have not been visited and until I actually visit them then the vertex should be considered true.

For the depth first search (DFS) I still need to same values to pass but in the method, I need to use a stack instead of a queue. In DFS I need to keep track of what vertexes have been visited so I need to declare the vertexes false, because they have not been visited and until I actually visit them then the vertex should be considered true. The difference between both the DFS and the DFS with recursion will be the way I traverse the graphs.

**Implementation**

To implement a BFS I passed the graph and the vertex we will start at; inside the method I needed to declare visited to false by the length of the graph to show that the vertex has not been visited. Then I had to make a queue variable that for now has an empty list and in the empty list I then have to append the vertex in the list. Then I have to declare that the visited at the index of the vertex is true. After that I need to traverse the queue with a while loop, in the loop I declared a variable v that will store the values that have been popped from the queue at the 0 position and then I print the v value at that index. After printing I need to make a for loop to traverse the graph at v where we popped the queue, inside the for loop there is an if statement that if I have visited the index is False then I need to append the value at that index and declare that that index has been visited.

To implement a DFS I passed the graph and the start node we will start at; inside the method I needed to declare visited to an empty list. Then I had to make a stack variable that for now has the start. After that I need to traverse the stack with a while loop, in the loop I declared a variable start node that will store the values that have been popped from the stack and then I need to append the start node to the visited list; after I need to print the start node that we currently have. After printing I need to make a for loop to traverse the graph by the start node where we popped the stack, inside the for loop I need to append the index to the stack and outside the loops I need to return the visited list.

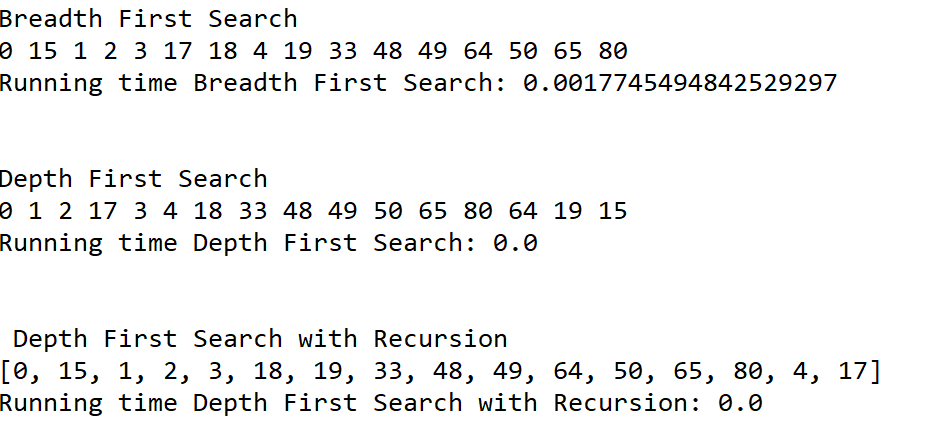
To implement a DFS with recursion I passed the graph ,the start node and visited to none; inside the method I needed to check that if visited is none the visited now has an empty list. Then I had to append the start node to the visited list. After that I need to traverse the graph with a for loop by the start node, inside the for loop I need to check that if the index has not been visited then I need to call the method pass the graph, the index, then the visited list. After the recursive call is done I need to return the visited lists.

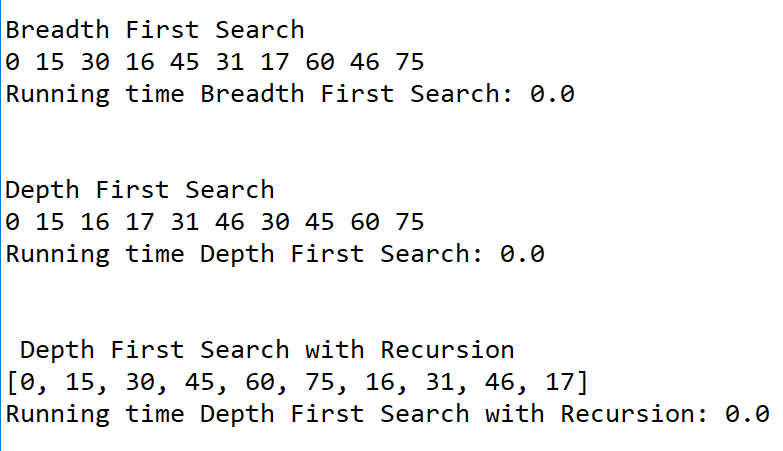
**Tracing**

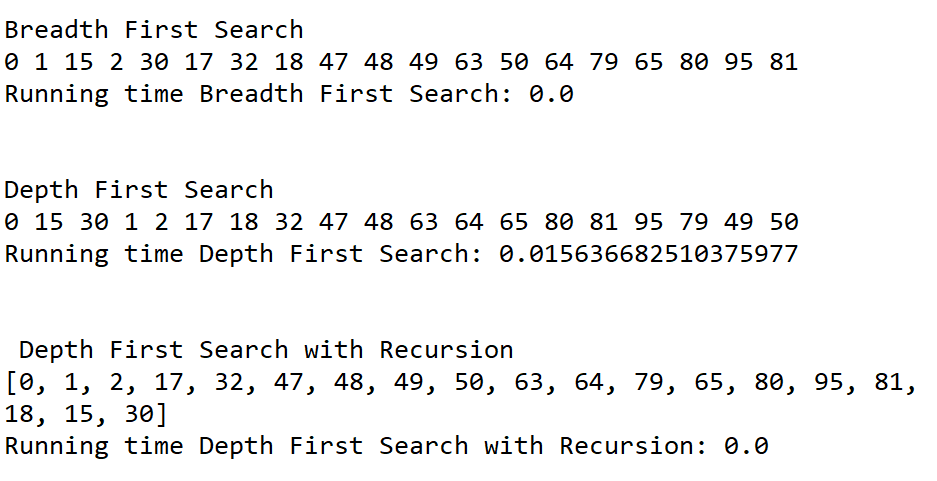
To trace this code, I will be using rows of size 10 and columns of size 15. In BFS the graph and the starting vertex which is 0 will be passed into the method the visited variable will be declared as false, we have a queue that is an empty list, and we append the vertex into the queue; Then the variable visited at the vertex will be declared as true. After the declarations the while loop will traverse the queue, and, in the loop, the variable v will store the values popped from the queue and the print the variable v. After that the for loop traverses my graph and checks that if visited at the current index is false then I append it index to the queue and visited is now true.

In DFS the graph and the starting vertex which is 0 will be passed into the method the visited variable will be declared as an empty list, we have a stack that is storing the start node. After the declarations the while loop will traverse the stack, and, in the loop, the variable start node will store the values popped from the stack, the value popped will be appended to the visited list, and the print the variable start node. After that the for loop traverses my graph by the start node variable and appends the index to the stack; outside the loops I return visited.

In DFS with recursion the graph, and the starting vertex which is 0, and visited which is false will be passed into the method. Inside the method theres an if statement that if visited is none the visited now has an empty list. Then the code appends the start node to the visited list. After that I need to traverse the graph with a for loop by the start node, inside the for loop I need to check that if the index has not been visited then I need to call the method pass the graph, the index, then the visited list. After the recursive call is done I need to return the visited lists.







**Academic dishonesty**

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act. Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

|  |
| --- |
| # -\*- coding: utf-8 -\*- |
|  | """ |
|  | Created on Thu Apr 25 07:57:25 2019 |
|  |  |
|  | @author: Alexis Navarro |
|  | CS 2302 |
|  | Olac Fuentes |
|  |  |
|  | Purpose: to find the path to solve the maze made from lab 6 and to use the graph functions. |
|  | """ |
|  |  |
|  | # -\*- coding: utf-8 -\*- |
|  |  |
|  |  |
|  |  |
|  | import matplotlib.pyplot as plt |
|  | import numpy as np |
|  | import random |
|  |  |
|  | #import datetime # need to use datetime instead of time because when using time, I would always get 0 for my running time when the size of my maze was too small |
|  | # datetime is more precise for smaller mazes when being tested |
|  | import time |
|  |  |
|  |  |
|  |  |
|  | #GIVEN FUNCTIONS (PROVIDED BY CS 2302) |
|  |  |
|  | def DisjointSetForest(size): |
|  | return np.zeros(size,dtype=np.int)-1 |
|  |  |
|  | def find(S,i): |
|  | # Returns root of tree that i belongs to |
|  | if S[i]<0: |
|  | return i |
|  | return find(S,S[i]) |
|  |  |
|  |  |
|  |  |
|  | def union(S,i,j): |
|  | # Joins i's tree and j's tree, if they are different |
|  | ri = find(S,i) |
|  | rj = find(S,j) |
|  | if ri!=rj: # Do nothing if i and j belong to the same set |
|  | S[rj] = ri # Make j's root point to i's root |
|  |  |
|  | def findC(S,i): |
|  | if S[i]<0: |
|  | return i |
|  | r = findC(S,S[i]) |
|  | S[i]=r |
|  | return r |
|  |  |
|  | #combines the two set by using their size as reference |
|  | def union\_by\_Size(S,i,j): |
|  | ri = findC(S,i) |
|  | rj = findC(S,j) |
|  | if ri!=rj: # Do nothing if i and j belong to the same set |
|  | if S[ri]>S[rj]: # j's tree is larger |
|  | S[rj] += S[ri] |
|  | S[ri] = rj |
|  | else: |
|  | S[ri] += S[rj] |
|  | S[rj] = ri |
|  |  |
|  | #------------------------------------------------------------------------------ |
|  | #MADE METHOD/FUNCTION REQUIRED TO ACCOMPLISH THE LAB |
|  |  |
|  | #method to count the amount of sets in the DSF |
|  | def setAmount(S): |
|  | count=0 |
|  | for i in range(len(S)): |
|  | if S[i]<0: |
|  | count +=1 |
|  | return count |
|  |  |
|  |  |
|  | def removeC(S,maze\_walls,numSets): |
|  | while numSets > 1: |
|  | w = random.choice(maze\_walls)# w gets the wall that was randomly selected |
|  | i=maze\_walls.index(w)#gets the position where we chose the wall to delete |
|  | if find(S,w[0]) != find(S,w[1]): |
|  | maze\_walls.pop(i) #deletes the wall |
|  | union\_by\_Size(S,w[0],w[1])# combines the walls after the deletion |
|  | numSets-=1 |
|  | return w |
|  | #------------------------------------------------------------------------------ |
|  | #METHOD TO DRAW THE MAZE (PROVIDED BY THE CS2302) |
|  | def draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=False): |
|  | fig, ax = plt.subplots() |
|  | for w in walls: |
|  | if w[1]-w[0] ==1: #vertical wall |
|  | x0 = (w[1]%maze\_cols) |
|  | x1 = x0 |
|  | y0 = (w[1]//maze\_cols) |
|  | y1 = y0+1 |
|  | else:#horizontal wall |
|  | x0 = (w[0]%maze\_cols) |
|  | x1 = x0+1 |
|  | y0 = (w[1]//maze\_cols) |
|  | y1 = y0 |
|  | ax.plot([x0,x1],[y0,y1],linewidth=1,color='k') |
|  | sx = maze\_cols |
|  | sy = maze\_rows |
|  | ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k') |
|  | if cell\_nums: |
|  | for r in range(maze\_rows): |
|  | for c in range(maze\_cols): |
|  | cell = c + r\*maze\_cols |
|  | ax.text((c+.5),(r+.5), str(cell), size=10, |
|  | ha="center", va="center") |
|  | ax.axis('off') |
|  | ax.set\_aspect(1.0) |
|  |  |
|  |  |
|  |  |
|  | def wall\_list(maze\_rows, maze\_cols): |
|  | # Creates a list with all the walls in the maze |
|  | w =[] |
|  | for r in range(maze\_rows): |
|  | for c in range(maze\_cols): |
|  | cell = c + r\*maze\_cols |
|  | if c!=maze\_cols-1: |
|  | w.append([cell,cell+1]) |
|  | if r!=maze\_rows-1: |
|  | w.append([cell,cell+maze\_cols]) |
|  | return w |
|  |  |
|  | #------------------------------------------------------------------------------ |
|  | #Lab 7 Code |
|  |  |
|  | #Method to find the cells in the maze |
|  | def cellsAmount(S): |
|  | count = 0 |
|  | if S is None: |
|  | return 0 |
|  | for i in S: |
|  | count+=1 |
|  | return count |
|  |  |
|  | #Method where n displays the number of cells and m is the amount of walls the user wants to remove |
|  | def display\_numCells(n,m): |
|  | if m < n-1: |
|  | print('Path does not Exist') |
|  | elif m == n-1: |
|  | print('Unique path Exists') |
|  | elif m > n-1: |
|  | print('There is atleast one path from the source to destination') |
|  |  |
|  | ''' |
|  | def graph\_toList(S,original,maze\_walls,numCells, duplicate=False): |
|  | adj=[] #empty adjacency List |
|  |  |
|  | for i in range(numCells): #traverse throught the amount of cells in maze\_walls |
|  | if original[i] in maze\_walls: |
|  | adj.append(original) |
|  | return adj |
|  | ''' |
|  |  |
|  | #made a class that will create the funcitions to make a graoh |
|  | class Graph: |
|  | def \_\_init\_\_(self,vertices): |
|  | self.vertices = vertices |
|  | self.graph = [] |
|  | for v in range(vertices): |
|  | self.graph.append([]) |
|  |  |
|  | # method to create the adjacency List its also a method that applies to this lab better |
|  | # unlike the graph to list method |
|  | def addEdge(G,v1,v2): |
|  | G.graph[v1].append(v2) |
|  |  |
|  | #method to use Breadth First Search |
|  | def Breadth\_FirstSearch(adjList,v): |
|  | visited= [False]\*(len(adjList.graph)) |
|  |  |
|  | Q=[] |
|  | Q.append(v) |
|  | visited[v]=True |
|  |  |
|  | while Q: |
|  | v = Q.pop(0) |
|  | print(v,end=" ") |
|  |  |
|  | for i in adjList.graph[v]: |
|  | if visited[i]==False: |
|  | Q.append(i) |
|  | visited[i]=True |
|  |  |
|  | #method to use depth First Search |
|  | def dfs(adjList, startNode): |
|  | visited=[] |
|  | stack = [startNode] |
|  | while stack: |
|  | startNode=stack.pop() |
|  | visited.append(startNode) |
|  | print(startNode,end=" ") |
|  |  |
|  | for i in adjList.graph[startNode]: |
|  | stack.append(i) |
|  | return visited |
|  |  |
|  | #method to use depth First Search with recursion |
|  | def dfs2(adjList,startNode, visited = None): |
|  | if visited is None: |
|  | visited = [] |
|  | visited.append(startNode) |
|  | for i in adjList.graph[startNode]: |
|  | if i not in visited: |
|  | dfs2(adjList,i,visited) |
|  | return visited |
|  |  |
|  |  |
|  | plt.close("all") |
|  | #------------------------------------------------------------------------------ |
|  | #MAIN |
|  |  |
|  | #size of rows and columns (Dimensions of the maze) |
|  | #Various sizes to test |
|  | maze\_rows = 10 # use datetime for these dimensions since they are smaller |
|  | maze\_cols = 15 |
|  |  |
|  | #maze\_rows = 20 # for bigger maze dimensions use time import |
|  | #maze\_cols = 25 |
|  |  |
|  | #maze\_rows = 40 |
|  | #maze\_cols = 45 |
|  |  |
|  | #maze\_rows = 2 |
|  | #maze\_cols = 4 |
|  | adjList=Graph(maze\_rows\*maze\_cols) |
|  |  |
|  | x=input('Input the amount of walls you want to remove from the maze: ') # ask for users input to remove the walls |
|  | numRemove=int(x) # convert the string input into a integer |
|  | numRemove+=1 |
|  |  |
|  | maze\_walls = wall\_list(maze\_rows,maze\_cols)#Gets the list of walls in the maze |
|  |  |
|  | draw\_maze(maze\_walls,maze\_rows,maze\_cols,cell\_nums=True) #calls the draw maze method and makes the complete maze without deletion |
|  |  |
|  | S = DisjointSetForest(maze\_rows\*maze\_cols)# makes the new DSF by combining the rows and columns |
|  |  |
|  |  |
|  | numCells=cellsAmount(S) # cells amount of the dsf |
|  |  |
|  | #numSets=setAmount(S) |
|  |  |
|  | display\_numCells(numCells,numRemove) # question 1 |
|  |  |
|  | # The remove walls method had to be moved to the main in order to be able to function with the search methods |
|  | while numRemove > 0: |
|  | w = random.choice(maze\_walls)# w gets the wall that was randomly selected |
|  | i=maze\_walls.index(w)#gets the position where we chose the wall to delete |
|  | if find(S,w[0]) != find(S,w[1]): |
|  | maze\_walls.pop(i) #deletes the wall |
|  | union(S,w[0],w[1])# combines the walls after the deletion |
|  | addEdge(adjList,w[0],w[1]) # used to create the adjacency List (question 2) |
|  | numRemove-=1 |
|  |  |
|  | draw\_maze(maze\_walls,maze\_rows,maze\_cols) |
|  |  |
|  | print(adjList.graph) # use this to print the graph with the adjacency List |
|  |  |
|  |  |
|  | #Question 3 |
|  | start\_Time1=time.time() |
|  | print('\nBreadth First Search') |
|  | Breadth\_FirstSearch(adjList,0) |
|  | end\_Time1=time.time() |
|  | print('\nRunning time Breadth First Search: %s '%(end\_Time1-start\_Time1)) |
|  |  |
|  |  |
|  |  |
|  | start\_Time2=time.time() |
|  | print('\n\nDepth First Search ') |
|  | dfs(adjList,0) |
|  | end\_Time2=time.time() |
|  | print('\nRunning time Depth First Search: %s '%(end\_Time2-start\_Time2)) |
|  |  |
|  |  |
|  |  |
|  | start\_Time3=time.time() |
|  | print('\n\n Depth First Search with Recursion') |
|  | print(dfs2(adjList,0)) |
|  | end\_Time3=time.time() |
|  | print('Running time Depth First Search with Recursion: %s '%(end\_Time3-start\_Time3)) |