**Author:** Julian Gonzalez

**Assignment:** Lab 7 Report

**Course:** CS 2302 - Data Structures 10:30-11:50

**Instructor:** Fuentes, Olac

**T.A.:** Nath, Anindita

***Introduction***

In this assignment we used a disjoint set forest to build a maze. The maze should contain a collection of cells separated by walls in such a way that there is exactly one simple path separating any two cells. We also use search algorithms to find a path from 0 cell to the last cell in the maze

***Solutions***

For making the maze I used the same code that I had created for lab 6. I modified it to work with the extra parameter of deleting only a user defined number of walls. This was done with while loop keeping it as a counter against the user defined walls to delete.

***Experimental***

The way I test my coded was using different number of cells top be removed in a 6 by 5 maze. First, I removed 15 walls, then 20 walls, 29, and finally 35. The next part that I tackled was creating an adjacency list with our new walls and the old walls created. I then created a while loop that would travers the old walls list and while doing so would check if w in the old walls list was not the same as in the new wall list. It this case was true a new list would append items from the second index of the list into the first of the L list. I also did this in reverse, but it would be used only if the graph was undirected. Then it finally came to do the three search algorithms. All where done with relative ease as the pseudocode for them was given to us in class. The only one that was not was depth first search iteratively, but this was essentially the same as breadth search but with stacks, so it was not hard to translate. Overall, I got all the methods to work except the recursive version of depth first search even when following the pseudocode, I believe this has to do with the way the program is done recursively.

15 walls removed

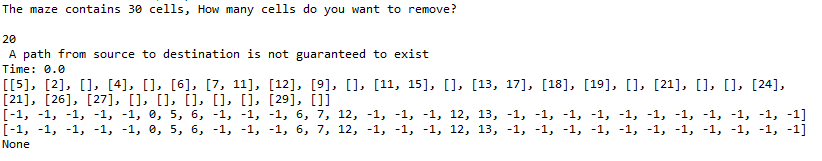
A screenshot of a social media post

Description automatically generated

A close up of a device

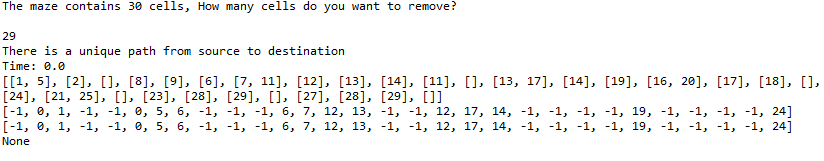
Description automatically generated

A close up of a device

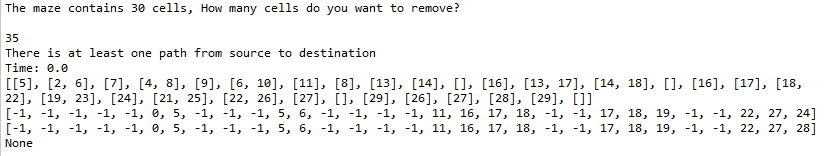
Description automatically generated

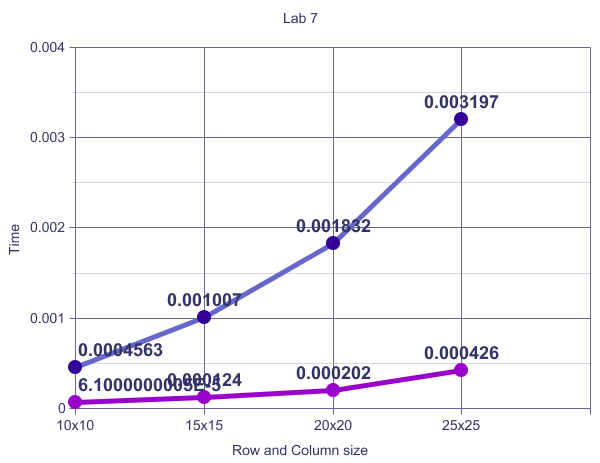
29 walls removed

20 walls removed

A close up of a device

Description automatically generated

A close up of a device

Description automatically generated

35 walls removed

Figure BLUE BREADTH SEARCH // PRUPLE DEPTH SEARCH

***Conclusion***

I learned how to make an adjacency list from our given wall list and how to work the two different search algorithms breadth and depth search.

Appendix:

1. # -\*- coding: utf-8 -\*-
2. """
3. Course: CS-2302 Data-Stuctures
4. Author: Julian Gonzalez
5. Assignment: Lab 7
6. Intstuctor: Olac Fuentes
7. T.A's: Anindita Nath, Maliheh Zargaran
9. """
11. **import** matplotlib.pyplot as plt
12. **import** numpy as np
13. **import** random
14. **import** time
15. **import** queue
17. ######################################################
18. **def** DisjointSetForest(size):
19. **return** np.zeros(size,dtype=np.int)-1
21. **def** union\_c(S,i,j):
22. # Joins i's tree and j's tree, if they are different
23. # Uses path compression
24. ri = find\_c(S,i)
25. rj = find\_c(S,j)
26. **if** ri!=rj:
27. S[rj] = ri
29. **def** find\_c(S,i): #Find with path compression
30. **if** S[i]<0:
31. **return** i
32. r = find\_c(S,S[i])
33. S[i] = r
34. **return** r
36. **def** draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=False):
37. fig, ax = plt.subplots()
38. **for** w **in** walls:
39. **if** w[1]-w[0] ==1: #vertical wall
40. x0 = (w[1]%maze\_cols)
41. x1 = x0
42. y0 = (w[1]//maze\_cols)
43. y1 = y0+1
44. **else**:#horizontal wall
45. x0 = (w[0]%maze\_cols)
46. x1 = x0+1
47. y0 = (w[1]//maze\_cols)
48. y1 = y0
49. ax.plot([x0,x1],[y0,y1],linewidth=1,color='k')
50. sx = maze\_cols
51. sy = maze\_rows
52. ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')
53. **if** cell\_nums:
54. **for** r **in** range(maze\_rows):
55. **for** c **in** range(maze\_cols):
56. cell = c + r\*maze\_cols
57. ax.text((c+.5),(r+.5), str(cell), size=10,
58. ha="center", va="center")
59. ax.axis('off')
60. ax.set\_aspect(1.0)
62. **def** wall\_list(maze\_rows, maze\_cols):
63. # Creates a list with all the walls in the maze
64. w =[]
65. **for** r **in** range(maze\_rows):
66. **for** c **in** range(maze\_cols):
67. cell = c + r\*maze\_cols
68. **if** c!=maze\_cols-1:
69. w.append([cell,cell+1])
70. **if** r!=maze\_rows-1:
71. w.append([cell,cell+maze\_cols])
72. **return** w
74. ######################################################
75. #Implementation of Search psuedocode given in class
76. **def** breadthFirstSearch(G,v): #used class pseudocode
77. visited = [False] \* len(G)#initialize visited False with length of G
78. prev = [-1] \* len(G)#initialize prev -1 with length of G
79. #print(visited,prev)
80. Q = queue.Queue()#may need to fix
81. Q.put(v)
82. visited[v] = True
83. **while** **not** Q.empty():
84. u = Q.get()
85. **for** t **in** G[u]:
86. **if** **not** visited[t]:
87. visited[t] = True
88. prev[t] = u
89. Q.put(t)
90. **return** prev
92. **def** depthFirstSearchS(G,v): #same as breadthFirstSearch but using list as a stack
93. visited = [False] \* len(G)#initialize visited False with length of G
94. prev = [-1] \* len(G)#initialize prev -1 with length of G
95. S = [] #create stack "list"
96. S.append(v)
97. visited[v] = True
98. **while** len(S) != 0:
99. u = S.pop()
100. **for** t **in** G[u]:
101. **if** **not** visited[t]:
102. visited[t] = True
103. prev[t] = u
104. S.append(t)
105. **return** prev
107. **def** depthFirstSearchR(G,source):#used class pseudocode
108. visited = [False] \* len(G)#initialize visited False with length of G
109. prev = [-1] \* len(G)#initialize prev -1 with length of G
110. visited[source] = True
111. **for** t **in** G[source]:
112. **if** **not** visited[t]:
113. prev[t]=source
114. depthFirstSearchR(G,t) ##dont know if need to return anything?
115. #return prev
116. ##################################################
118. **def** ModbuildMazeC(SC,w,numCells,numRemove):#modified version of BuildMazeC used in lab6
119. counter = 0 #used to remove user specifc amount of walls
120. **while** counter <  numRemove:
121. #Assign each cell to a different set in a disjoint set forest S
122. #Select a random wall w =[c1,c2]
123. randNum = random.randint(0,len(w)-1)
124. randomWall = w[randNum]
125. c1,c2 = randomWall[0],randomWall[1]
126. **if** counter < (numCells -1 ):
127. **if** find\_c(SC,c1) != find\_c(SC,c2):#If cells c1 and c2 belong to different sets, remove w and join c1’s set and c2’s set
128. w.pop(randNum)
129. union\_c(SC,c1,c2)
130. counter += 1
131. **else**:
132. w.pop(randNum)
133. counter +=1
135. **return** w

138. **def** adjMaze(newWalls,walls,cells):
139. L = [[] **for** i **in** range(cells)]
140. **for** w **in** walls:
141. **if** w **not** **in** newWalls: #checks for wall
142. L[w[0]].append(w[1]) # insert adjacency one way
143. #L[w[1]].append(w[0]) # used when want to see all edges if not used sometimes cannot find path to end
144. **return** L
146. ####################################################
148. row = 100
149. col = 100
150. numCells = row \* col#number of total cells
152. wallList = wall\_list(row,col)
153. #draw\_maze(wallList,row,col,cell\_nums=True)#cell\_nums=True
155. dsf = DisjointSetForest(numCells)
157. **print**("The maze contains %d cells, How many cells do you want to remove?" % (numCells))
158. numRemove = int(input())
160. **if** numRemove < numCells-1:
161. **print**(" A path from source to destination is not guaranteed to exist")
162. **if** numRemove == numCells-1:
163. **print**("There is a unique path from source to destination")
164. **if** numRemove > numCells-1:
165. **print**("There is at least one path from source to destination")
167. start = time.time()
168. tempWalls = wallList.copy()
170. newWalls= ModbuildMazeC(dsf,tempWalls,numCells,numRemove)
172. end = time.time()
173. **print**('Time:',end-start)
174. draw\_maze(newWalls,row,col,cell\_nums=True)

177. adjList = adjMaze(newWalls,wallList,numCells)#may not work depending on generated graph
178. **print**(adjList)
180. **print**(breadthFirstSearch(adjList,0))#0 for bottom left corner
181. **print**(depthFirstSearchS(adjList,0))
182. **print**(depthFirstSearchR(adjList,0))

I Julian Gonzalez certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.

* Julian Gonzalez