**Author:** Julian Gonzalez

**Assignment:** Lab 6 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.

***Solutions***

The bulk of the code for this lab was given for us by Professor Fuentes. We essentially had to create two methods following the given pseudocode from the lab assignment

*Create full maze with all adjacent cells are separated by a wall*

*Assign each cell to a different set in a disjoint set forest S*

*While S has more than one set*

*Select a random wall w =[c1,c2]*

*If cells c1 and c2 belong to different sets, remove w and join c1’s set and c2’s set*

*otherwise do nothing*

*Display maze*

This was used to create the function that would do what is asked of us for the lab. We were asked to modify the given code to work with compressed disjoint set forest as well. The solution that I came up with for both functions is as follows. First a “wall list” would be created by using the given wall\_list() function. Afterwards a M and N (row and columns) would be created and all tree variables will be passed onto the function. Inside a Disjoint set forest would be created with the given rows and columns using the DisjointSetForest() function. Then the while loop would happen which would call a special function that would pass the created disjoint set forest and would check the number of sets (by checking the number of -1’s) in the DSF and return the count. Inside the while loop essentially what was written in pseudocode was done. The if statement utilized the given class code of find\_c to check the cells of c1 and c2 and while inside removing w required to use the built in .pop() function for lists to remove w and afterwards joining c1 and c2 was done by the given class code union() and union\_by\_size() which did regular union and compressed union. After that was done w was returned to be displayed with the draw\_maze() function again provided as class code.

***Experimental***

A close up of a logo

Description automatically generated The way I test my coded was using different Row and columns sizes to compare the runtimes of the size increasing and the difference of runtime between standard union and compressed union.

10x10

A close up of a piece of paper

Description automatically generatedA close up of a piece of paper

Description automatically generatedA close up of a logo

Description automatically generatedA close up of a logo

Description automatically generated

50x50

20x20

40x40

30x30

A close up of a map

Description automatically generatedA close up of a map

Description automatically generated

***Conclusion***

Overall this fun was relatively easy as most of the code was provided to us on the class webpage. Nut what we did was insightful and did help to further my knowledge of Disjoint set forest

Appendix:

1. # -\*- coding: utf-8 -\*-
2. """
3. Course: CS-2302 Data-Stuctures
4. Author: Julian Gonzalez
5. Assignment: Lab 6
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
16. ######################################################
17. **def** DisjointSetForest(size):
18. **return** np.zeros(size,dtype=np.int)-1
20. **def** dsfToSetList(S):
21. #Returns aa list containing the sets encoded in S
22. sets = [ [] **for** i **in** range(len(S)) ]
23. **for** i **in** range(len(S)):
24. sets[find(S,i)].append(i)
25. sets = [x **for** x **in** sets **if** x != []]
26. **return** sets
28. **def** find(S,i):
29. # Returns root of tree that i belongs to
30. **if** S[i]<0:
31. **return** i
32. **return** find(S,S[i])
34. **def** find\_c(S,i): #Find with path compression
35. **if** S[i]<0:
36. **return** i
37. r = find\_c(S,S[i])
38. S[i] = r
39. **return** r
41. **def** union(S,i,j):
42. # Joins i's tree and j's tree, if they are different
43. ri = find(S,i)
44. rj = find(S,j)
45. **if** ri!=rj:
46. S[rj] = ri
48. **def** union\_c(S,i,j):
49. # Joins i's tree and j's tree, if they are different
50. # Uses path compression
51. ri = find\_c(S,i)
52. rj = find\_c(S,j)
53. **if** ri!=rj:
54. S[rj] = ri
56. **def** union\_by\_size(S,i,j):
57. # if i is a root, S[i] = -number of elements in tree (set)
58. # Makes root of smaller tree point to root of larger tree
59. # Uses path compression
60. ri = find\_c(S,i)
61. rj = find\_c(S,j)
62. **if** ri!=rj:
63. **if** S[ri]>S[rj]: # j's tree is larger
64. S[rj] += S[ri]
65. S[ri] = rj
66. **else**:
67. S[ri] += S[rj]
68. S[rj] = ri
69. ####################################################
70. **def** draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=False):
71. fig, ax = plt.subplots()
72. **for** w **in** walls:
73. **if** w[1]-w[0] ==1: #vertical wall
74. x0 = (w[1]%maze\_cols)
75. x1 = x0
76. y0 = (w[1]//maze\_cols)
77. y1 = y0+1
78. **else**:#horizontal wall
79. x0 = (w[0]%maze\_cols)
80. x1 = x0+1
81. y0 = (w[1]//maze\_cols)
82. y1 = y0
83. ax.plot([x0,x1],[y0,y1],linewidth=1,color='k')
84. sx = maze\_cols
85. sy = maze\_rows
86. ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')
87. **if** cell\_nums:
88. **for** r **in** range(maze\_rows):
89. **for** c **in** range(maze\_cols):
90. cell = c + r\*maze\_cols
91. ax.text((c+.5),(r+.5), str(cell), size=10,
92. ha="center", va="center")
93. ax.axis('off')
94. ax.set\_aspect(1.0)
96. **def** wall\_list(maze\_rows, maze\_cols):
97. # Creates a list with all the walls in the maze
98. w =[]
99. **for** r **in** range(maze\_rows):
100. **for** c **in** range(maze\_cols):
101. cell = c + r\*maze\_cols
102. **if** c!=maze\_cols-1:
103. w.append([cell,cell+1])
104. **if** r!=maze\_rows-1:
105. w.append([cell,cell+maze\_cols])
106. **return** w
108. ############################################################
109. #Create full maze with all adjacent cells are separated by a wall
110. #Assign each cell to a different set in a disjoint set forest S
111. #While S has more than one set
112. #Select a random wall w =[c1,c2]
113. #If cells c1 and c2 belong to different sets, remove w and join c1’s set and c2’s set
114. #otherwise do nothing
115. #Display maze
116. ####################################################
117. #method that returns True if theres more than one set in the disjoint set forest
118. # and false otherwise
119. **def** checkSets(S):
120. c = 0
121. #counts number of sets
122. **for** i **in** range(len(S)):
123. **if** S[i] == -1:
124. c +=1
125. **if** c>1:
126. **return** True
127. **return** False
129. #Create full maze with all adjacent cells are separated by a wall
130. **def** buildMaze(w,row,cols):
132. S = DisjointSetForest(row\*cols)
133. #Assign each cell to a different set in a disjoint set forest S
134. **while** checkSets(S) == True:
135. randNum = random.randint(0,len(w)-1)
136. #Select a random wall w =[c1,c2]
137. randomWall = w[randNum]
138. c1,c2 = randomWall[0],randomWall[1]
140. **if** find\_c(S,c1) != find\_c(S,c2):#If cells c1 and c2 belong to different sets, remove w and join c1’s set and c2’s set
141. w.pop(randNum)
142. #            temp = []
143. #            for x in range(len(walls)):
144. #                if walls[x] != d:
145. #                    temp.append(walls[x])
146. #            walls=temp
147. union(S,c1,c2)
149. **return** w
151. #same as last but with path compression
152. **def** buildMazeC(w,M,N):
154. SC = DisjointSetForest(M\*N)
155. #Assign each cell to a different set in a disjoint set forest S
156. **while** checkSets(SC) == True:
157. #Select a random wall w =[c1,c2]
158. randNum = random.randint(0,len(w)-1)
159. randomWall = w[randNum]
160. c1,c2 = randomWall[0],randomWall[1]
162. **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
163. w.pop(randNum)
164. union\_by\_size(SC,c1,c2)
166. **return** w
168. M = 10 #rows
169. N = 10 #columns
170. standard = wall\_list(M,N)
171. compressed =  wall\_list(M,N)
173. start = time.time()
174. standardUnion = buildMaze(standard,M,N)
175. draw\_maze(standardUnion,M,N)
176. end = time.time()
177. **print**('Union standard:',end-start)
179. start2 = time.time()
180. compressedUnion = buildMazeC(compressed,M,N)
181. draw\_maze(compressedUnion,M,N)
182. end2= time.time()
183. **print**('Union compression:',end2-start2)

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