# Introduction

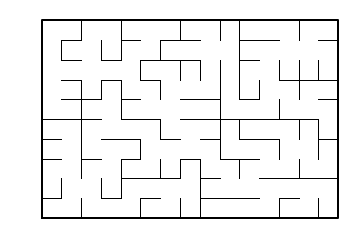
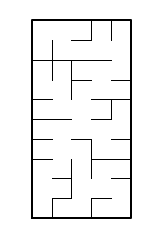
The problem we were given was to create a maze based on input of how many walls to remove from our maze. Similar to lab 6, except here we removed walls based on input, not until we only had 1 root. Along with finding a solution to creating a maze with user input, we also had to put the new cells without walls into an adjacency list. After getting an adjacency list, we solved the maze using depth first search, breadth first search (with stacks), and breadth first search with recursion.

# Proposed solution

First I had to find a way to redo the maze construction from lab 6. First thing I did was get the input from the user and created a new method called message that would display a message to the user based on how many walls they wanted to remove. Next I used the input in a while loop that checked as long as the input was greater than 1, this allowed us to iterate with like a counter. The rest of the building was similar to lab 6, we got a random wall, we checked if both cells were not in the same dsf, and if they weren’t we would union. After the union we updated our input by -1 to iterate and remove the amount of walls the user asked for. The next method we did was the adjacency list. For the adjacency list I used the original wall list, the new wall list with removed walls, and the number of cells. Next I created a new list for adjacency. Next we iterated through the original wall list and we checked if the original wall we iterated in was not in the new wall list, if the wall was not in the list we appended [i[0]].append[1] and [i[1]].append[0] because that would insert the edge into the list. The last 3 methods were copied from the pseudocode that was given in class and translated into python. I used queues, stacks, and recursion.

# Experimentation

Removed walls of different lengths to check if worked, created graphs of different sizes and printed arrays of last 3 methods to check if solutions were similar/same.



# Conclusion

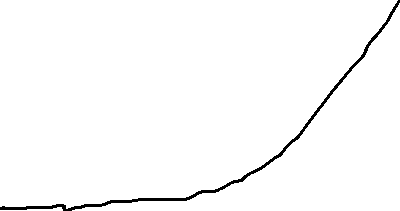
Learned how to solve mazes using depth first search and breadth first search. Also learned a new way to construct mazes. Had a bit of trouble with a couple parts like the adjacency list, but came up with the solution and also enjoyed creating the pseudocode into real code.

# Appendix

1. #Lab7 modify maze
2. # Programmed by Olac Fuentes
3. # Last modified April 29, 2019
4. #Cesar Lopez
5. #10:30 - 12:00
7. **import** matplotlib.pyplot as plt
8. **import** numpy as np
9. **import** random
11. **def** draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=False):
12. fig, ax = plt.subplots()
13. **for** w **in** walls:
14. **if** w[1]-w[0] ==1: #vertical wall
15. x0 = (w[1]%maze\_cols)
16. x1 = x0
17. y0 = (w[1]//maze\_cols)
18. y1 = y0+1
19. **else**:#horizontal wall
20. x0 = (w[0]%maze\_cols)
21. x1 = x0+1
22. y0 = (w[1]//maze\_cols)
23. y1 = y0
24. ax.plot([x0,x1],[y0,y1],linewidth=1,color='k')
25. sx = maze\_cols
26. sy = maze\_rows
27. ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')
28. **if** cell\_nums:
29. **for** r **in** range(maze\_rows):
30. **for** c **in** range(maze\_cols):
31. cell = c + r\*maze\_cols
32. ax.text((c+.5),(r+.5), str(cell), size=10,
33. ha="center", va="center")
34. ax.axis('off')
35. ax.set\_aspect(1.0)
37. **def** wall\_list(maze\_rows, maze\_cols):
38. # Creates a list with all the walls in the maze
39. w =[]
40. **for** r **in** range(maze\_rows):
41. **for** c **in** range(maze\_cols):
42. cell = c + r\*maze\_cols
43. **if** c!=maze\_cols-1:
44. w.append([cell,cell+1])
45. **if** r!=maze\_rows-1:
46. w.append([cell,cell+maze\_cols])
47. **return** w
49. **from** scipy **import** interpolate
51. **def** DisjointSetForest(size):
52. **return** np.zeros(size,dtype=np.int)-1
54. **def** dsfToSetList(S):
55. #Returns aa list containing the sets encoded in S
56. sets = [ [] **for** i **in** range(len(S)) ]
57. **for** i **in** range(len(S)):
58. sets[find(S,i)].append(i)
59. sets = [x **for** x **in** sets **if** x != []]
60. **return** sets
62. **def** find(S,i):
63. # Returns root of tree that i belongs to
64. **if** S[i]<0:
65. **return** i
66. **return** find(S,S[i])
68. **def** find\_c(S,i): #Find with path compression
69. **if** S[i]<0:
70. **return** i
71. r = find\_c(S,S[i])
72. S[i] = r
73. **return** r
75. **def** union(S,i,j):
76. # Joins i's tree and j's tree, if they are different
77. ri = find(S,i)
78. rj = find(S,j)
79. **if** ri!=rj:
80. S[rj] = ri
82. **def** union\_c(S,i,j):
83. # Joins i's tree and j's tree, if they are different
84. # Uses path compression
85. ri = find\_c(S,i)
86. rj = find\_c(S,j)
87. **if** ri!=rj:
88. S[rj] = ri
90. **def** union\_by\_size(S,i,j):
91. # if i is a root, S[i] = -number of elements in tree (set)
92. # Makes root of smaller tree point to root of larger tree
93. # Uses path compression
94. ri = find\_c(S,i)
95. rj = find\_c(S,j)
96. **if** ri!=rj:
97. **if** S[ri]>S[rj]: # j's tree is larger
98. S[rj] += S[ri]
99. S[ri] = rj
100. **else**:
101. S[ri] += S[rj]
102. S[rj] = ri

105. **def** draw\_dsf(S):
106. scale = 30
107. fig, ax = plt.subplots()
108. **for** i **in** range(len(S)):
109. **if** S[i]<0: # i is a root
110. ax.plot([i\*scale,i\*scale],[0,scale],linewidth=1,color='k')
111. ax.plot([i\*scale-1,i\*scale,i\*scale+1],[scale-2,scale,scale-2],linewidth=1,color='k')
112. **else**:
113. x = np.linspace(i\*scale,S[i]\*scale)
114. x0 = np.linspace(i\*scale,S[i]\*scale,num=5)
115. diff = np.abs(S[i]-i)
116. **if** diff == 1: #i and S[i] are neighbors; draw straight line
117. y0 = [0,0,0,0,0]
118. **else**:      #i and S[i] are not neighbors; draw arc
119. y0 = [0,-6\*diff,-8\*diff,-6\*diff,0]
120. f = interpolate.interp1d(x0, y0, kind='cubic')
121. y = f(x)
122. ax.plot(x,y,linewidth=1,color='k')
123. ax.plot([x0[2]+2\*np.sign(i-S[i]),x0[2],x0[2]+2\*np.sign(i-S[i])],[y0[2]-1,y0[2],y0[2]+1],linewidth=1,color='k')
124. ax.text(i\*scale,0, str(i), size=20,ha="center", va="center",
125. bbox=dict(facecolor='w',boxstyle="circle"))
126. ax.axis('off')
127. ax.set\_aspect(1.0)
129. #created to display message based on amount of walls wanted to remove
130. **def** message(m, n):
131. **if** m < n-1:
132. **print**("A path from source to destination is not guaranteed to exist when: walls removed (your input) < (cells/walls-1)")
133. **if** m == n-1:
134. **print**("There is a unique path from source to destination when: walls removed (your input) = (cells/walls-1)")
135. **if** m > n-1:
136. **print**("There is at least one path from source to destination when: walls removed (your input) > (cells/walls-1)")
138. #creates adjacent list, takes in original walls, new walls, and number of cells
139. **def** AdLi(ow, w, cells):
140. adj = [[]**for** i **in** range(cells)]#creates 3d list that holds walls as vertices instead of cells
141. **for** j **in** ow:
142. **if** j **not** **in** w:#when a wall is not found in the new walls, they are inserted as an edge
143. adj[j[0]].append(j[1])
144. adj[j[1]].append(j[0])
145. **return** adj
147. #used pseudocode given in class
148. **def** BFS(g, v):
149. visited = len(g)\*[False]
150. prev = len(g)\*[-1]
151. q = queue.Queue()
152. q.put(v)
153. visited[v] = True
154. **while** **not** q.empty():
155. u = q.get()
156. **for** t **in** g[u]:
157. **if** **not** visited[t]:
158. visited[t] = True
159. prev[t] = u
160. q.put(t)
161. **return** prev
163. #used pseudocode given in class
164. **def** dfsS(g, v):
165. s = []#a list in python works like a stack
166. visited = len(g)\*[False]
167. prev = len(g)\*[-1]
168. s.append(v)
169. visited[v] = True
170. **while** len(s) **is** **not** 0:
171. u = s.pop()
172. **for** t **in** g[u]:
173. **if** **not** visited[t]:
174. visited[t] = True
175. prev[t] = u
176. s.append(t)
177. **return** prev
179. #used pseudocode given in class
180. **def** dfsR(g, v, visited, prev):
181. visited[v] = True
182. **for** t **in** g[v]:
183. **if** **not** visited[t]:
184. prev[t] = v
185. dfsR(g, t, visited, prev)
186. **return** prev
187. #######################################
188. **import** queue
189. plt.close("all")
190. maze\_rows = 10
191. maze\_cols = 15
193. walls = wall\_list(maze\_rows,maze\_cols)
194. S = DisjointSetForest(maze\_rows\*maze\_cols)
195. draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True)
196. d = random.randint(0, len(walls)-1)
197. OrigW = walls.copy()
199. **print**("There is:",maze\_rows\*maze\_cols,"cells/walls")
200. m = (int)(input("How many walls would you like to remove?"))
201. **print**(m, "walls will be removed")
202. message(m, (maze\_rows\*maze\_cols))
204. **while**(m>0):
205. d = random.randint(0,len(walls)-1)
206. hold = walls[d]
207. **if** find(S, hold[0]) != find(S, hold[1]):
208. walls.pop(d)
209. union(S, hold[0], hold[1])
210. m -= 1
211. #hold of adjacent list
212. adj = AdLi(OrigW, walls, maze\_rows\*maze\_cols)
213. **print**(adj)
214. **print**("breadth",BFS(adj, 0))
215. #declared here instead of creating global variables, used for recursive method
216. #v = visited, and p = previous
217. v = len(adj)\*[False]
218. p = len(adj)\*[-1]
219. **print**("stack", dfsS(adj, 0))
220. **print**("recursive", dfsR(adj, 0, v, p))
221. draw\_maze(walls,maze\_rows,maze\_cols)

# Running Time



O(n)

I 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. C.L