Lab 7 report

Nicole Favela

CS2301

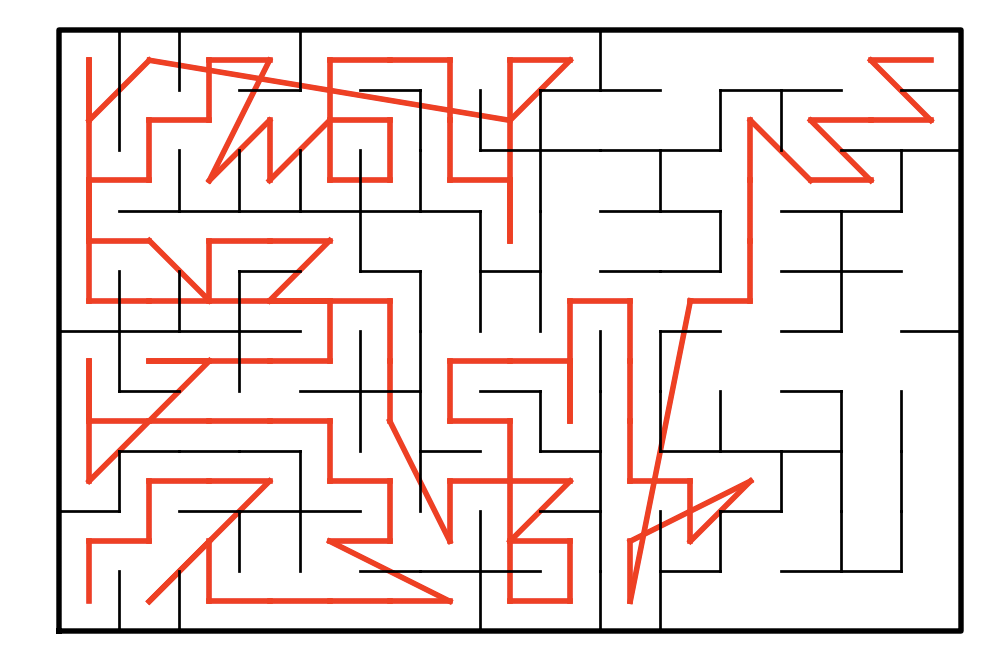
May 1, 2019

**Introduction:**

The purpose of this assignment is to use disjoint set forests to create a maze (similar to lab 6) and to then solve the maze using depth first search recursively, regular depth first search, and breadth first search. I used the code provided which draws the mazes which consists of cells (a two dimensional array) and then removes walls randomly. Unlike lab 6, this program is meant to ask the user for the number of walls they want to remove (m) as well as the method in which to build the maze (union by size or standard union). I was tasked with creating the search algorithms to solve the maze to draw the path created by their solutions.

**Proposed solution and implementation:**

To count the number of sets, I used a function called numSets which goes through all the elements in S (the array for the dsf) and counts the elements in the same set. I am using union by size and standard union in my functions that build the mazes, I am getting a random integer called d and using that to select a wall from an array of walls as in the previous lab. All mazes work for both union and union\_by\_size. C1 and c2 are elements in the dsf at positions 0 and 1 respectively. Inside the loop I am checking if c1 and c2 are in the set. If they are not in the set, I am joining them and popping a wall. This is the same as in lab 6. All my buildmaze functions are the same as in lab 6 except for an extra conditional I added a conditional to make sure that if element that are adjacent and are mirrored then the walls are popped at those points. In this lab, I used 2 different search algorithms to solve the maze. In depth first search (recursive version) I am using a flag to keep track of whether or not the vertex and destination (top right corner) match. My parameters are AL, vertex, destination (which is the end cell at the top right corner) and path (a list to store the vertices along the route). I am using a for loop to go through the adjacency list and check if i has been added to the path or not. If not, it recursively goes through the adjacency list (AL). From there, I am returning the path as a list. I had to add a conditional at the end to pop the last element if it is a dead end so the draw\_path function I created does not go that way. In depth first search (iterative), I did it exactly like my breadth first search, but instead of a queue I used a stack. In my breadth first search function (BFS), I have 3 parameters AL (adjacency list), v which was the start vertex, and target, the end destination vertex at the top right corner of every maze. I created a queue to store the vertexes visited and path to store what is popped from the queue. From there, I used last to store the last thing in path and then enter the first conditional in the while loop. Then I iterate through the list (called adj) and append to the queue what is seen in the path and check always if the target vertex is found, if not it moves to the next vertex and find what that is adjacent to. My breadth first search then returns “visit” if that target is found and “queue” if the same target vertex and start are entered. In order to draw the path, I had to convert my list of paths taken to an edge list which I did in Path\_to\_EL. In that function, I created an empty edge list and went to the length of the path created using a for loop that appended the nearest neighbors in an EL. My draw\_path function was altered from the given one draw\_maze. In draw\_maze, I created a global value equal to walls in order to draw the path and not alter the original. Early attempts to draw the path went through walls and zig zagged as shown below:



I fixed this by changing the list of paths to an edge list instead and then passed that as a parameter into my draw\_path function and popped unnecessary cells along the path. I used the edge list I created from the respective algorithms and went thought and the edge list path (EL) and drew the paths in red.

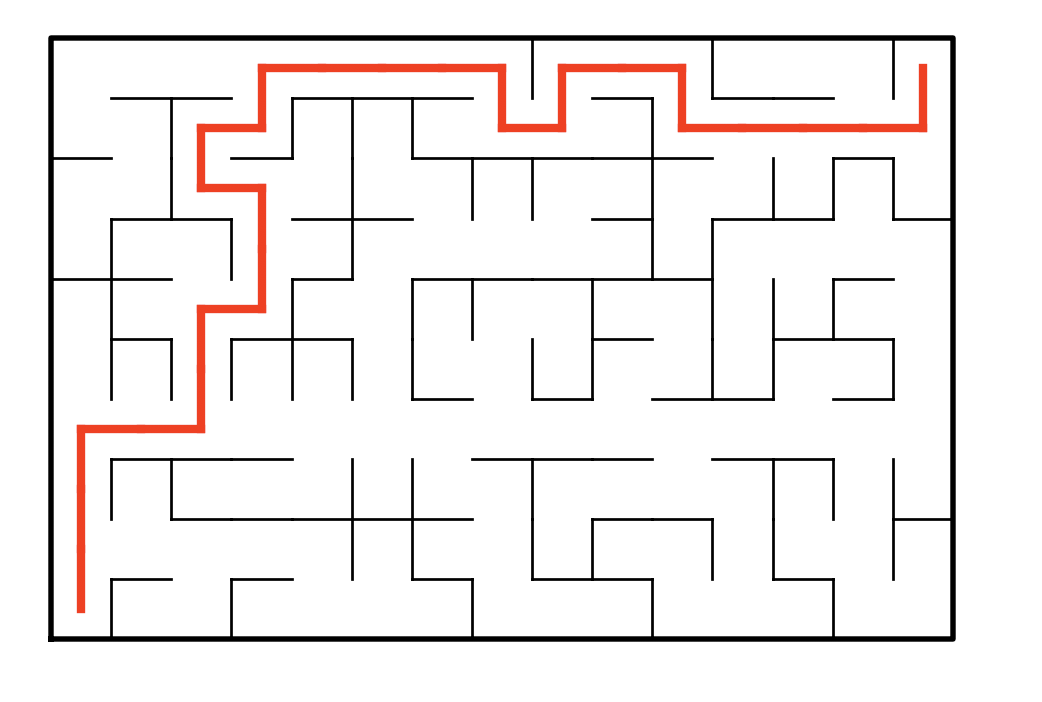
**Runtimes and samples outputs:**

For my sample runtimes I tested my methods each with varying sizes rows and columns. For my mazes I used size = 150, n = 1350, and n = 12150 by altering the rows and columns by a factor of 3 each time. I timed the runtimes of breadth first search and depth first search functions from start to finish in order to compare their runtimes. In order to get a good estimate of the runtimes, I used a for loop to run the algorithms 10 times each and then took the average. I also created mazes and tested the algorithms for both union by size (when the user selects 1) and standard union if the user selects 2. The times are recorded in seconds using time.time\_ns(). For all the runtime operations, choice 1 was selected (union\_by\_size) for simplicity.

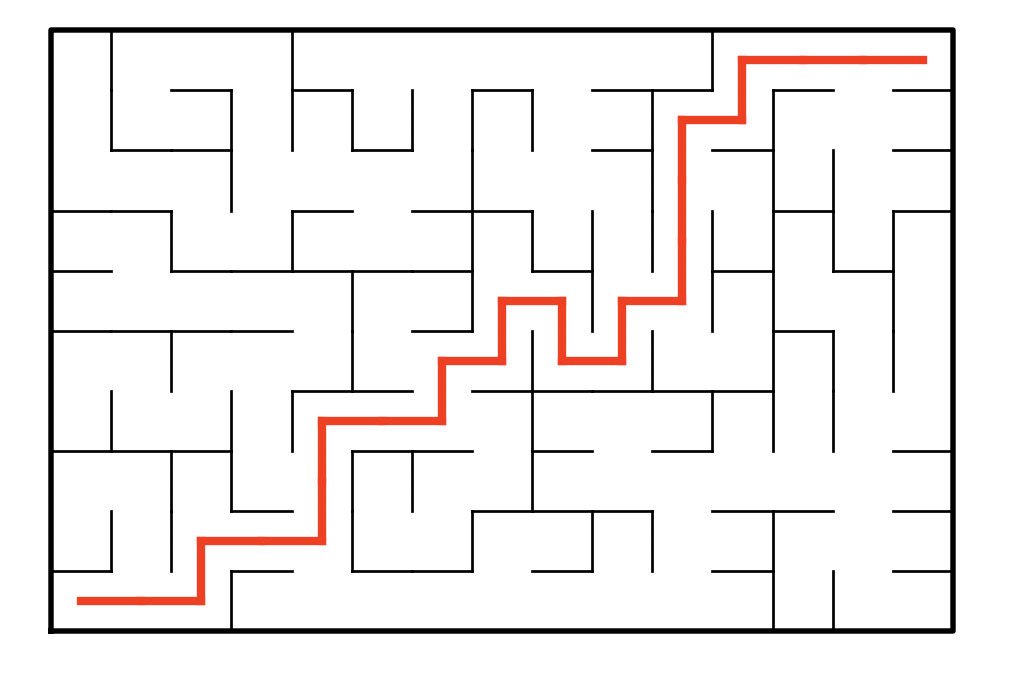
The sample console output for a maze of size 150 solved with breadth first search is shown below with case 3 selected (m>n-1):



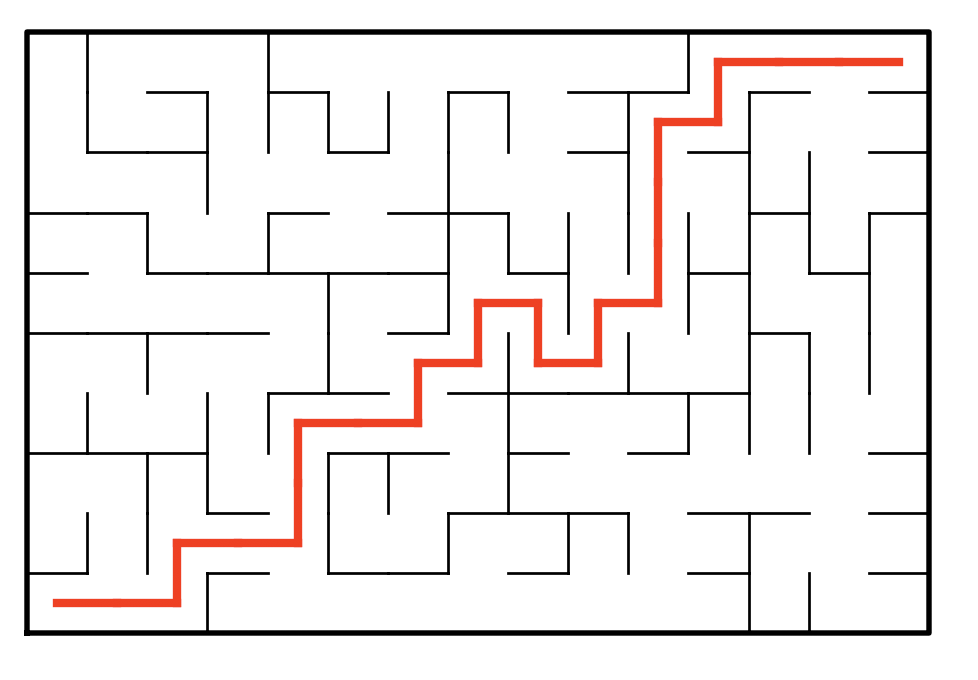
The following maze is produced using the same size using case 2 (m == n-1) using breadth first search.



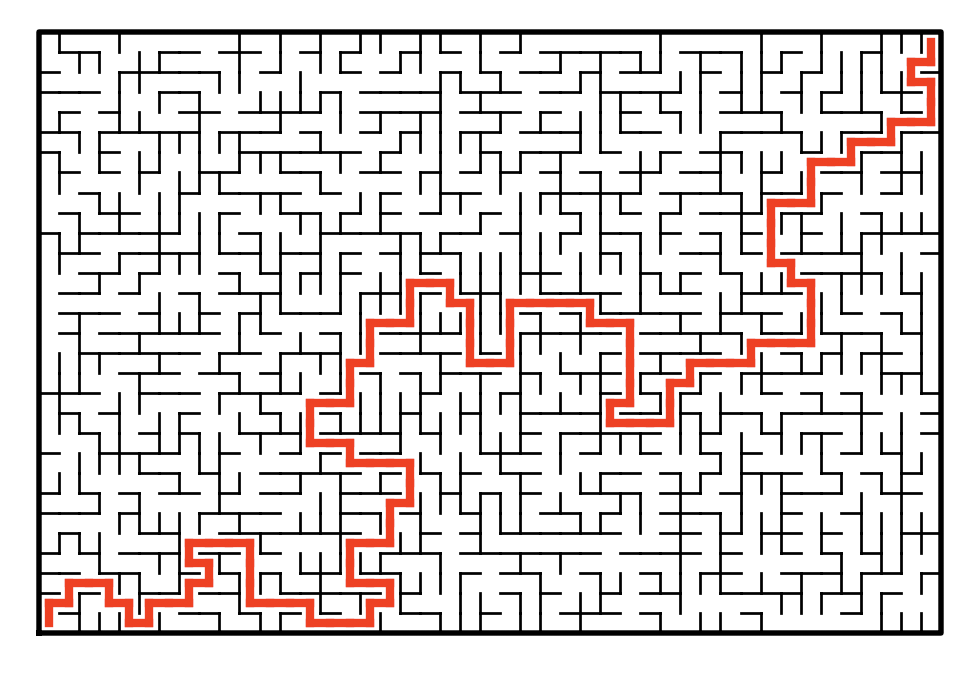
The following maze is produced by depth first search (iterative) with m>n-1:



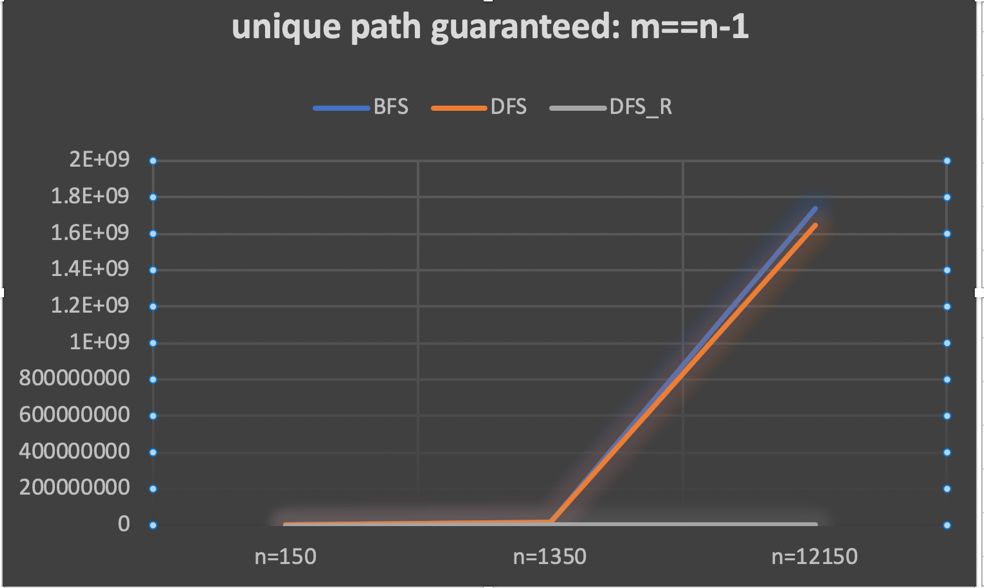
The following is produced by depth first search (recursive) with m>n-1:



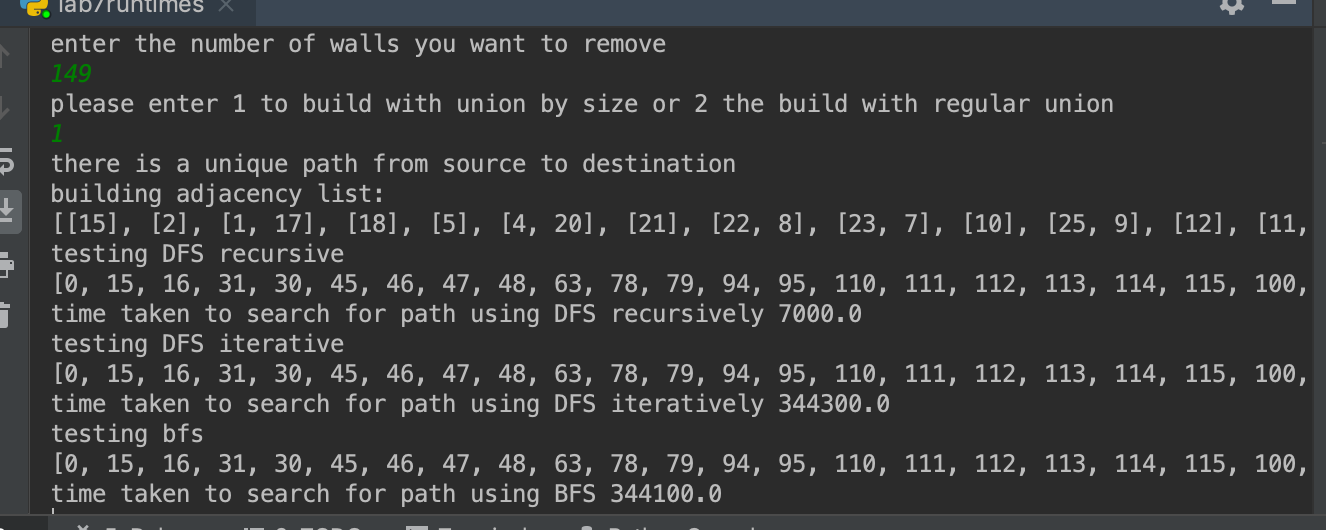
For all maze runtimes, I recorded the time taken to find a path in all 3 cases: m<n-1 (path not guaranteed), m==n-1 (unique path guaranteed), and m>n-1 (at least one path exists). The following are maze is produced by the removing 1349 walls with size = 1350 (unique path guaranteed):



The following is a graph produced when a unique source is guaranteed with varying maze sizes. For consistency, I used only union by size and m==n-1 because I was getting extremely variable runtimes when a path was not found. As shown in the graph, the algorithms almost completely overlap until n=1350, although the scale makes it appear as though they do, they are distinct numbers for the most part.



Sample console outputs:



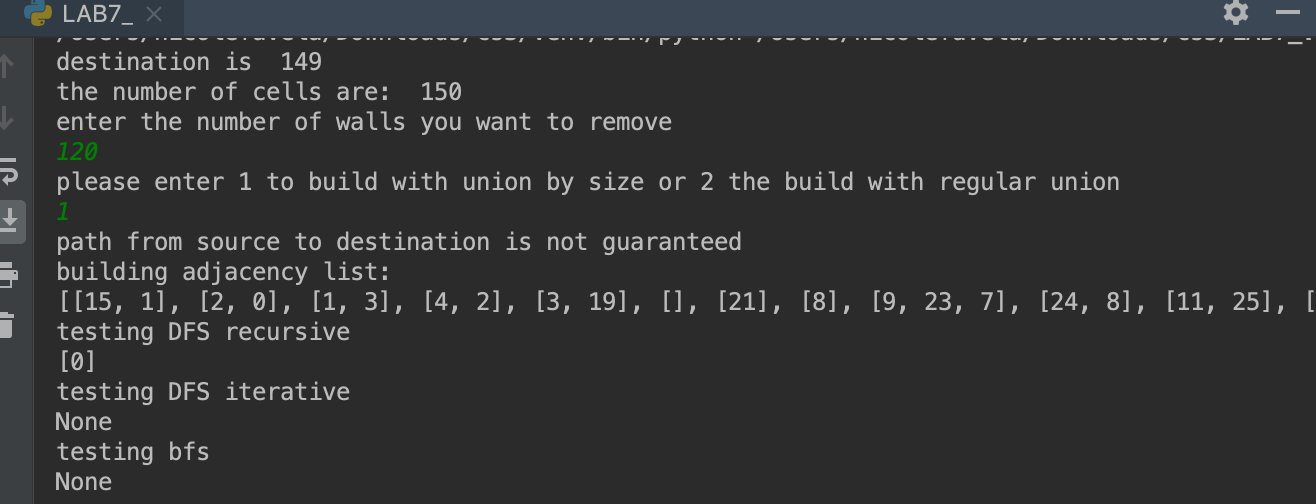
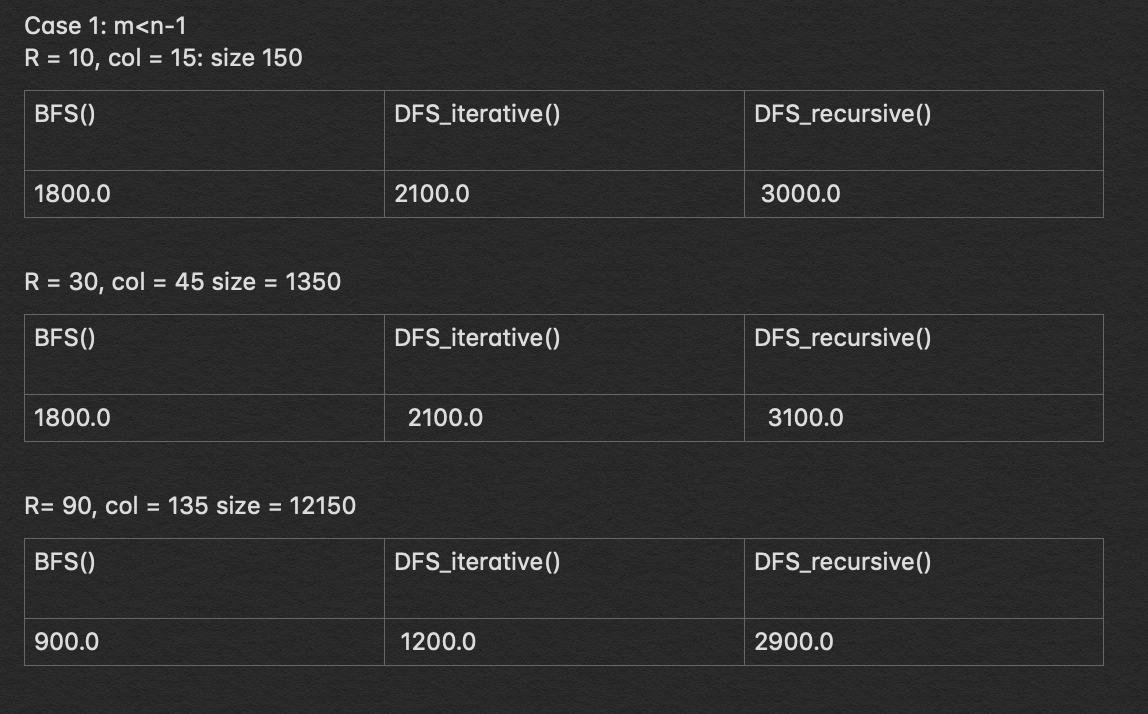
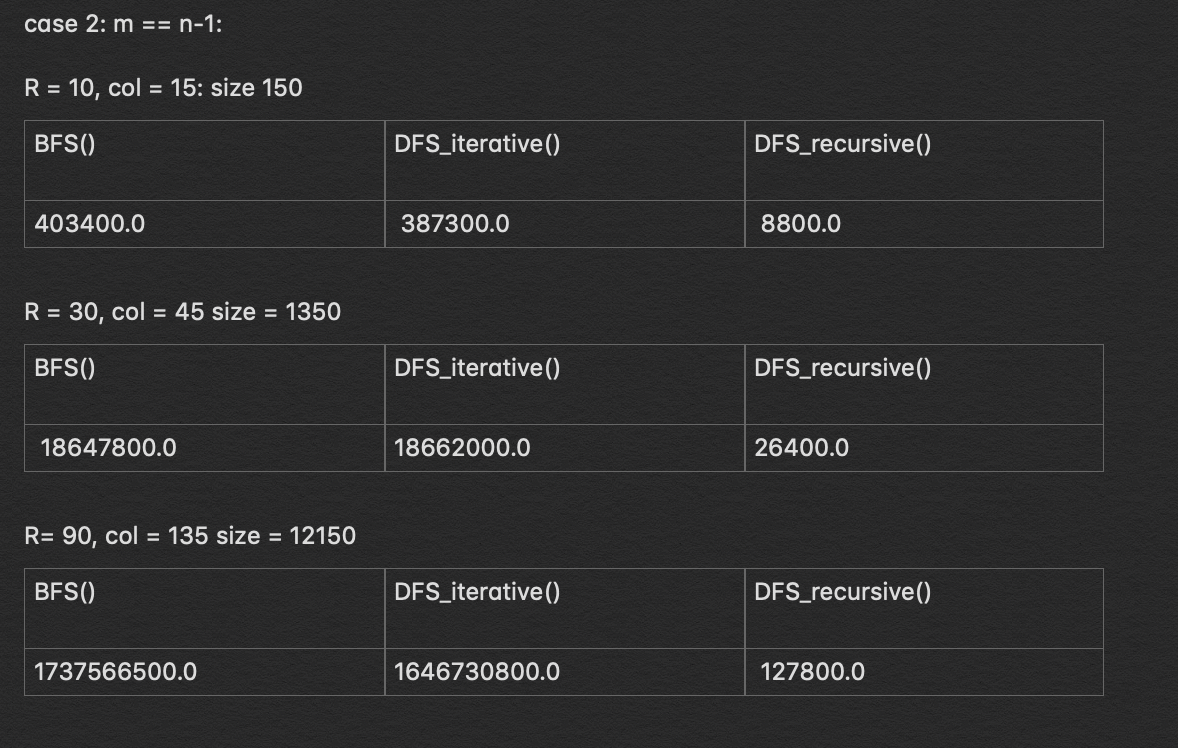
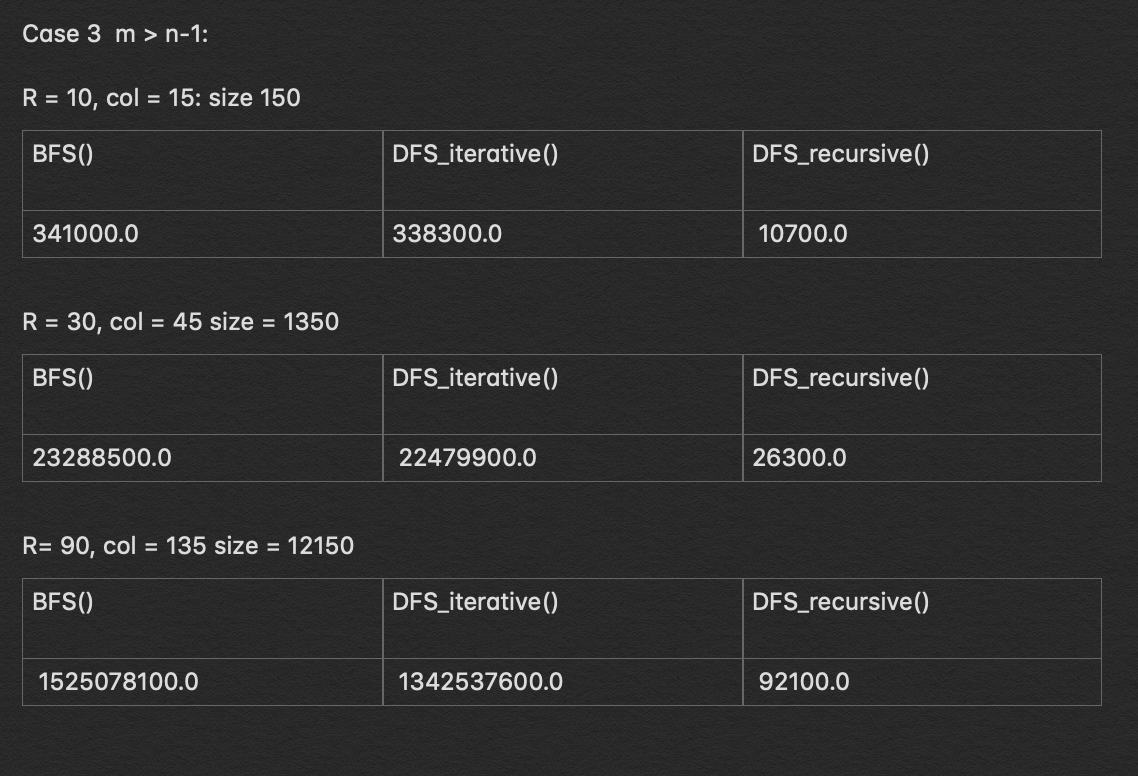




Table of selected values with times recorded as an average in nanoseconds:







**Conclusion:**

This project taught me how to use the graph search algorithms breadth first search and depth first search to solve a maze. The most challenging function to design and draw was the recursive version of depth first search. Due to the recursive nature the algorithm does when it reaches a dead end, I had first ended up with very convoluted and messy paths draws. This also taught me how to use an altered version of the draw graph provided in order to draw the path taken shown in red.

**Appendix source (source code):**

#CS2302  
#Nicole Favela  
#last modified: April 30, 2019  
#Lab7  
#purpose: to build a maze and solve it using breadth first search and depth first search algorithms and show the paths created  
#instructor: Olac Fuentes  
#TAs: Anindita Nath and Maliheh Zargaran  
  
# Starting point for program to build and draw a maze  
# Modify program using disjoint set forest to ensure there is exactly one  
# simple path joiniung any two cells  
# Programmed by Olac Fuentes  
# Last modified March 28, 2019  
  
import random  
from matplotlib import pyplot as plt  
import numpy as np  
import math  
import queue  
import time  
  
  
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\_by\_size(S,i,j):  
 # if i is a root, S[i] = -number of elements in tree (set)  
 # Makes root of smaller tree point to root of larger tree  
 # Uses path compression  
 ri = find\_c(S,i)  
 rj = find\_c(S,j)  
 if ri!=rj:  
 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  
  
  
#Find with path compression  
def find\_c(S,i):  
 if S[i]<0:  
 return i  
 r = find\_c(S,S[i])  
 S[i] = r  
 return r  
  
#counts the number of sets in the dsf  
def numSets(S):  
 count = 0  
 for i in range (len(S)):  
 if S[i] <0 or S[i] == i:  
 count+=1  
 return count  
  
#joins sets  
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  
  
#union by compression  
def union\_c(S,i,j):  
 # Joins i's tree and j's tree, if they are different  
 # Uses path compression  
 ri = find\_c(S,i)  
 rj = find\_c(S,j)  
 if ri!=rj:  
 S[rj] = ri  
  
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  
  
#builds maze with union by size  
def buildMaze\_with\_union\_by\_size(cells,G,mirror=False):  
 count = 0  
 while cells!=count and numSets(S)>1:  
 #d is a random integer in range of walls  
 d = random.randint(0, len(walls) - 1)  
  
 #wall is an list location in walls at d  
 wall = walls[d]  
 #grabs coordinates for cells  
 c1 = wall[0]  
 c2 = wall[1]  
 #if not in same set  
 if find\_c(S,c1)!= find\_c(S,c2):  
 #puts adjaceny list in order  
 if c1<c2 and c2 not in G[c1]:  
 G[c1].append(c2)  
 G[c2].append(c1)  
 if mirror is not False:  
 walls.pop(d)#remove wall  
 union\_by\_size(S,c1,c2) #make part of same set  
 count+=1  
 d+=1  
 #returns the walls remaining  
 #used to draw later  
 global walls2  
 walls2=walls  
 return G  
  
#builds maze with standard union  
def buildMaze(cells,G,mirror=False):  
 count = 0  
  
 while cells!=count and numSets(S)>1:  
 #d is a random integer in range of walls  
 d = random.randint(0, len(walls) - 1)  
 #wall is an list location in walls at d  
 wall = walls[d]  
 #grabs coordinates for cells  
 c1 = wall[0]  
 c2 = wall[1]  
 #if not in same set  
 if find(S,c1)!= find(S,c2):  
 # puts adjaceny list in order  
 if c1 < c2 and c2 not in G[c1]:  
 G[c1].append(c2)  
 if mirror:  
 G[c2].append(c1)  
 walls.pop(d)#remove wall  
 union(S, c1, c2) # make part of same set  
 count += 1  
 d += 1  
 global walls2  
 walls2 = walls  
 return G  
  
#breadth first search  
def BFS(AL,v,target):  
 #stores visitied vertices  
 visited=[]  
 #queue  
 queue=[[v]]  
 #vertex v is start  
 #target is last cell in top right corner  
 if v == target:  
 return queue  
 while queue:  
 path=queue.pop(0)  
 last=path[-1]  
 #if last element in path has not been visited, add to list  
 if last not in visited:  
 adj=AL[last]  
 #goes through adj and creates visit that is len of path  
 for i in adj:  
 visit=list(path)  
 #appends to visit  
 visit.append(i)  
 queue.append(visit)  
 #if target element reached, just append  
 if i ==target:  
 return visit  
 #adds to visited items  
 visited.append(last)  
  
#recursive version of DFS  
def DFS\_recursive(AL, vertex,destination,path=[]):  
 # used to keep track of whether or not the vertex is found  
 flag = True  
 # adds vertex to path  
 path += [vertex]  
 # goes through length of adjacency list  
 for i in AL[vertex]:  
 #if destination found at vertex return path  
 if vertex == destination:  
 flag = False  
 return path  
 #recursively calls if i not in path and flag is true  
 if not i in path and flag:  
 path = DFS\_recursive(AL, i, destination, path)  
 # checks if the last item in the path is the destination  
 if path[-1] == destination:  
 return path  
 else:  
 #if doesn't lead to path  
 path.pop(-1)  
 return path  
  
#iterative version of depth first search  
#uses stack  
def DFS\_iterative(AL, start,destination):  
 visited = []  
 stack = [[start]]  
 if start == destination:  
 return stack  
 #while stack not empty  
 while stack:  
 path = stack.pop(0)  
 #last is the last item in path  
 last = path[-1]  
 #same as BFS only with stack  
 if last not in visited:  
 adj = AL[last]  
 for i in adj:  
 visit = list(path)  
 visit.append(i)  
 stack.append(visit)  
 if i == destination:  
 return visit  
 visited.append(last)  
  
#changes list to edge list for maze  
def Path\_to\_EL(path):  
 #list of edges  
 if path is None:  
 return None  
 EL = []  
 #goes through path created by algorithms  
 for i in range (1,len(path)):  
 #appends path[0],path[1]...  
 EL.append([path[i-1], path[i]])  
 return EL  
  
#draws maze path  
def draw\_path(EL,walls2,maze\_rows,maze\_cols):  
 fig, ax = plt.subplots()  
 #draws path  
 for w in EL:  
 if w[1] - w[0] == 1: #horizontal  
 x0 = (w[0] % maze\_cols) + .5  
 y0 = (w[0] // maze\_cols) + .5  
 x1 = (w[1] % maze\_cols) + .5  
 y1 = (w[1] // maze\_cols) + .5  
 else: #vertical  
 x0 = (w[0] % maze\_cols) + .5  
 y0 = (w[0] // maze\_cols) + .5  
 x1 = (w[1] % maze\_cols) + .5  
 y1 = (w[1] // maze\_cols) + .5  
 #plots in red with linewidth 3  
 ax.plot([x0, x1], [y0, y1], linewidth=3, color='r')  
 sx = maze\_cols  
 sy = maze\_rows  
  
 #draws the maze  
 for w in walls2:  
 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')  
  
 ax.plot([0, 0, sx, sx, 0], [0, sy, sy, 0, 0], linewidth=2, color='k')  
  
 ax.axis('off')  
 ax.set\_aspect(1.0)  
  
plt.close("all")  
  
maze\_rows = 10  
maze\_cols = 15  
  
#creates dsf of 150  
S = DisjointSetForest(maze\_rows\*maze\_cols)  
#creates array of walls  
walls = wall\_list(maze\_rows,maze\_cols)  
#draws empty maze  
draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True)  
  
n = len(S)  
#last index in top right  
destination = n-1  
print('destination is ', destination)  
print('the number of cells are: ', n)  
  
print('enter the number of walls you want to remove')  
m = int(input())  
  
print('please enter 1 to build with union by size or 2 the build with regular union')  
choice = int(input())  
  
if m < n-1:  
 print('path from source to destination is not guaranteed')  
 if choice == 1:  
 print('building adjacency list:')  
 G = [[] for i in range(n)]  
 # M is the adjacency list returned  
 M = buildMaze\_with\_union\_by\_size(m, G,True)  
 print(M)  
 draw\_maze(walls, maze\_rows, maze\_cols)  
  
  
 print('testing DFS recursive')  
  
 print(DFS\_recursive(M, 0,destination))  
 EL\_path\_from\_DFS = Path\_to\_EL(DFS\_recursive(M, 0, destination))  
 # removes last line  
 EL\_path\_from\_DFS.pop()  
 #added to  
 if EL\_path\_from\_DFS is not None:  
 draw\_path(EL\_path\_from\_DFS, walls, maze\_rows, maze\_cols)  
  
 print('testing DFS iterative')  
 print(DFS\_iterative(M, 0,destination))  
 EL = Path\_to\_EL(DFS\_iterative(M, 0, destination))  
 if EL is not None:  
 draw\_path(EL, walls, maze\_rows, maze\_cols)  
  
 print('testing bfs')  
 print(BFS(M, 0,destination))  
 EL\_path\_from\_BFS = Path\_to\_EL(BFS(M, 0, destination))  
 if EL\_path\_from\_BFS is not None:  
 draw\_path(EL\_path\_from\_BFS, walls2, maze\_rows, maze\_cols)  
  
 plt.show()  
  
 elif choice == 2:  
 print('building adjacency list:')  
 G = [[] for i in range(n)]  
 M = buildMaze(m, G,True)  
 print(M)  
 draw\_maze(walls, maze\_rows, maze\_cols)  
  
 print('testing DFS recursive')  
  
 print(DFS\_recursive(M, 0, destination))  
 EL\_path\_from\_DFS = Path\_to\_EL(DFS\_recursive(M, 0, destination))  
 #removes last line  
 EL\_path\_from\_DFS.pop()  
 if EL\_path\_from\_DFS is not None:  
 draw\_path(EL\_path\_from\_DFS, walls2, maze\_rows, maze\_cols)  
  
 print('testing DFS iterative')  
 print(DFS\_iterative(M, 0, destination))  
 EL = Path\_to\_EL(DFS\_iterative(M, 0, destination))  
 if EL is not None:  
 draw\_path(EL, walls2, maze\_rows, maze\_cols)  
  
 print('testing bfs')  
 print(BFS(M, 0, destination))  
 EL\_path\_from\_BFS = Path\_to\_EL(BFS(M, 0, destination))  
 if EL\_path\_from\_BFS is not None:  
 draw\_path(EL\_path\_from\_BFS, walls2, maze\_rows, maze\_cols)  
 plt.show()  
  
if m == n-1:  
 print('there is a unique path from source to destination')  
 if choice == 1:  
 print('building adjacency list:')  
 G = [[] for i in range(n)]  
 M = buildMaze\_with\_union\_by\_size(m, G,True)  
 print(M)  
 draw\_maze(walls, maze\_rows, maze\_cols)  
  
  
 print('testing DFS recursive')  
 print(DFS\_recursive(M,0,destination))  
 EL\_path\_from\_DFS = Path\_to\_EL(DFS\_recursive(M,0,destination))  
 EL\_path\_from\_DFS.pop()  
 draw\_path(EL\_path\_from\_DFS, walls2, maze\_rows, maze\_cols)  
  
 print('testing DFS iterative')  
 print(DFS\_iterative(M, 0,destination))  
 EL = Path\_to\_EL(DFS\_iterative(M, 0,destination))  
 draw\_path(EL, walls2, maze\_rows, maze\_cols)  
  
 print('testing bfs')  
 print(BFS(M, 0,destination))  
 EL\_path\_from\_BFS = Path\_to\_EL(BFS(M, 0,destination))  
 draw\_path(EL\_path\_from\_BFS , walls2, maze\_rows, maze\_cols)  
  
 plt.show()  
 elif choice == 2:  
 print('building adjacency list:')  
 G = [[] for i in range(n)]  
 print(buildMaze(m, G,True))  
 draw\_maze(walls, maze\_rows, maze\_cols)  
 M = buildMaze(m, G)  
  
 print('testing DFS recursive')  
  
 print(DFS\_recursive(M, 0, destination))  
 EL\_path\_from\_DFS = Path\_to\_EL(DFS\_recursive(M, 0, destination))  
 EL\_path\_from\_DFS.pop()  
 draw\_path(EL\_path\_from\_DFS, walls2, maze\_rows, maze\_cols)  
  
 print('testing DFS iterative')  
 print(DFS\_iterative(M, 0, destination))  
 #draws the path depth first search takes  
 EL = Path\_to\_EL(DFS\_iterative(M, 0, destination))  
 draw\_path(EL, walls2, maze\_rows, maze\_cols)  
  
 print('testing bfs')  
 print(BFS(M, 0, destination))  
 #draws the path breadth first search takes  
 EL\_path\_from\_BFS = Path\_to\_EL(BFS(M, 0, destination))  
 draw\_path(EL\_path\_from\_BFS, walls2, maze\_rows, maze\_cols)  
  
 plt.show()  
if m > n-1:  
 print('there is at least one path from source to destination')  
 if choice == 1:  
 print('building adjacency list:')  
 G = [[] for i in range(n)]  
 print(buildMaze\_with\_union\_by\_size(m, G,True))  
 draw\_maze(walls, maze\_rows, maze\_cols)  
  
 M = buildMaze\_with\_union\_by\_size(m, G)  
  
  
 print('testing DFS recursive')  
 print(DFS\_recursive(M, 0,destination))  
 EL\_path\_from\_DFS = Path\_to\_EL(DFS\_recursive(M, 0, destination))  
 EL\_path\_from\_DFS.pop()  
 draw\_path(EL\_path\_from\_DFS, walls2, maze\_rows, maze\_cols)  
  
 print('testing DFS iterative')  
 print(DFS\_iterative(M, 0,destination))  
 EL = Path\_to\_EL(DFS\_iterative(M, 0, destination))  
 draw\_path(EL, walls2, maze\_rows, maze\_cols)  
  
 print('testing bfs')  
 print(BFS(M, 0,destination))  
 EL\_path\_from\_BFS = Path\_to\_EL(BFS(M, 0, destination))  
 draw\_path(EL\_path\_from\_BFS, walls2, maze\_rows, maze\_cols)  
  
 plt.show()  
 elif choice == 2:  
 print('building adjacency list:')  
 G = [[] for i in range(n)]  
 print(buildMaze(m, G,True))  
 draw\_maze(walls, maze\_rows, maze\_cols)  
 M = buildMaze(m, G)  
  
 print('testing DFS recursive')  
  
 print(DFS\_recursive(M, 0, destination))  
 EL\_path\_from\_DFS = Path\_to\_EL(DFS\_recursive(M, 0, destination))  
 EL\_path\_from\_DFS.pop()  
 draw\_path(EL\_path\_from\_DFS, walls2, maze\_rows, maze\_cols)  
  
 print('testing DFS iterative')  
 print(DFS\_iterative(M, 0, destination))  
 EL = Path\_to\_EL(DFS\_iterative(M, 0, destination))  
 draw\_path(EL, walls2, maze\_rows, maze\_cols)  
  
 print('testing bfs')  
 print(BFS(M, 0, destination))  
 EL\_path\_from\_BFS = Path\_to\_EL(BFS(M, 0, destination))  
 draw\_path(EL\_path\_from\_BFS, walls2, maze\_rows, maze\_cols)  
  
 plt.show()  
  
plt.show()

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.”

-Nicole Favela