Lab 7

Fernando De Santiago

University of Texas at El Paso

**Introduction**

The purpose of this lab was to use breadth first search as queue, depth first search as stack, and depth first search as recursion to solve a maze if possible when told how many walls to delete and draw the solution if possible.

# Requirement 1

The first requirement was to show the user how many states there are and asking them to input how many walls they would like removed. Depending on how many walls they want removed it will lead to 1 of 3 possible options. The first option is if the number of walls it wants removed is less than the number of cells minus 1, then it will print that a path is not guaranteed to exist. The second is if the number of walls it wants removed is equal to the number of cells minus 1, and it says there is only one path. And the last option is if they remove more walls than cells minus 1, where it says there might be more than one possible path.

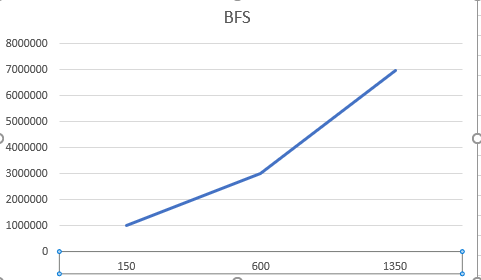
**Requirement 2**

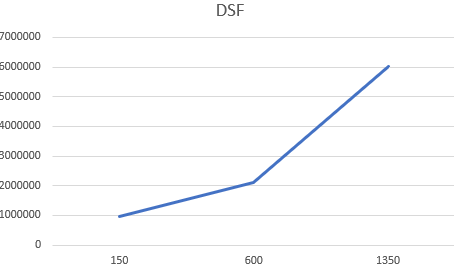
The second requirement was to create and adjacency list from the walls removed that will lead to the destination which is out upper right-hand corner. This was the easiest one to do cause we were given a model previously and so I decided to use that model.

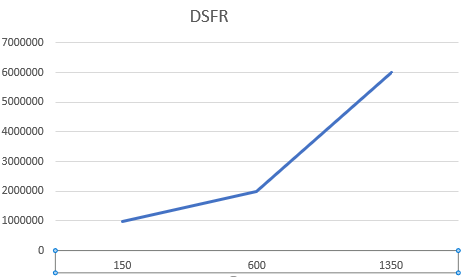
**Requirement 3**

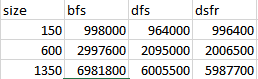
The last requirement had me building 3 methods which are Breadth first search using a queue, Depth first search using stack, and Depth first search using recursion. The first method is breadth first search which takes in 3 parameters: 1. Is Adjacency list, 2. Is the starting point and last is the goal. We start by initializing a queue that is a list of lists that uses the starting point and what is connected to it. I then initialize a empty set. We then create a while loop that states while the queue is not empty then it will do the following. It will pop the first value of the queue and insert it into 2 variables called vertex and path. We then reach out first conditional that if the vertex has not been in the set then it will check another conditional that checks if the vertex is the goal if it is then will the path. Inside the first conditional we then will add the vertex to the set and enter the last for loop that will go on while there is something in the graph[vertex] and inside it will append the next element and the path plus the next element into the queue. Depth first search is the same way only that you replace the queue with a stack. For the last method of depth first search recursive it takes in 3 to 4 parameters which are the adjacency list, starting point, goal and the path which if not defined is empty. We start by initializing path equal to path to [start]. I then created a variable called checker that is initialized to true. We then reach our first conditional which checks if the last item in the path is the same as the goal if it isn’t then it goes into a for loop that goes on while I is in adjacency of start. Then we have a another conditional that checks if the starting number is equal to the goal. If it is then it will turn the checker into false and return the path. Else it will go to the next conditional that check while I is not in path and checker is not false then path will be equal to the recursive call of the method and then inside this if it goes to another conditional that will check if the last thing in the path is the goal then it will return the path else it will pop the way it took if it backtracks and at the end will return the path.

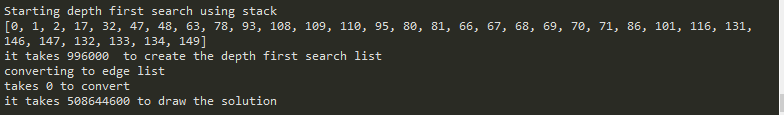
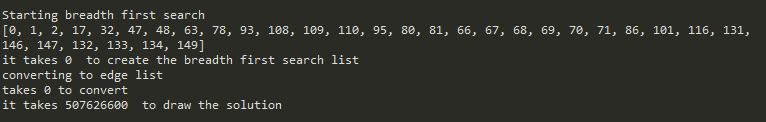
Graphs

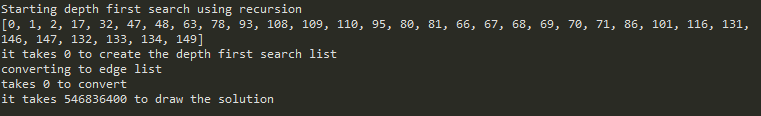


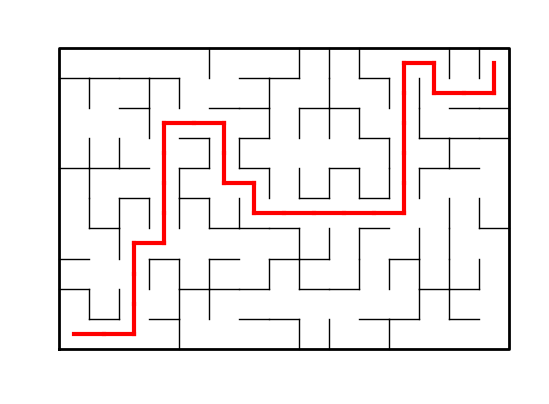












**Conclusion**

In conclusion I learned how to navigate an adjacency list to create a breadth first search and a depth first search. Personally I found that doing it recursively gave me the most amount of trouble due to the way the return was working and had to use activation records in order to figure out how to get out and delete unneeded walls. I also had a hard time drawing the solution until I saw that I needed to convert the path into an edge list and use that into a path.

**Appendix**

**# -\*- coding: utf-8 -\*-**

**"""**

**Created on Fri Apr 26 09:28:42 2019**

**@author: Fernando**

**"""**

**#CS2302**

**#Fernando De Santiago**

**#LAB7**

**#Olac Fuentes, Anindita Nath and Maliheh Zargaran**

**#last edited 4/30/19 19:29:00 PM**

**#Section M/W 10:30-11:50**

**#purpose: To create a maze and display a message if it's possible to be solved.**

**#wether the number of walls removed give a correct maze. and to create 3**

**#algorithms to search and create a path from bottom left corner to upper right.**

**# 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 matplotlib.pyplot as plt**

**import numpy as np**

**import random**

**import time**

**def ELConverter(G):#used to convert adjacency list to edge list**

**if G is None:**

**return None**

**g = []**

**for source in range(1,len(G)):**

**g.append([G[source-1],G[source]])**

**return g**

**def DisjointSetForest(size):#creates sets \* size and fills with -1**

**return np.zeros(size,dtype=np.int,order='C')-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 find\_c(S,i):**

**#Find with path compression**

**if S[i]<0:**

**return i**

**r= find\_c(S,S[i])**

**S[i]=r**

**return r**

**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: # Do nothing if i and j belong to the same set**

**S[rj] = ri # Make j's root point to i's root**

**def NumSets(S):#counts all the sets in the set S**

**count=0**

**for i in range(len(S)):**

**if S[i]<0:**

**count+=1**

**return count**

**def Draw\_Solution(Edge,walls2,maze\_rows,maze\_cols):**

**fig, ax = plt.subplots()**

**# draw edgelist path**

**for w in Edge:**

**if w[1]-w[0] ==1: #new x**

**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:#new y**

**x0 = (w[0]%maze\_cols)+.5**

**y0 = (w[0]//maze\_cols)+.5**

**x1 = (w[1]%maze\_cols)+.5**

**y1 = (w[1]//maze\_cols)+.5**

**ax.plot([x0,x1],[y0,y1],linewidth=3,color='r')**

**sx = maze\_cols**

**sy = maze\_rows**

**#draw inside of 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')**

**#Draw the rim of the maze**

**ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')**

**ax.axis('off')**

**ax.set\_aspect(1.0)**

**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# Make j's root point to i's root**

**def draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=False):#draws the base maze**

**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**

**plt.close("all")**

**maze\_rows = 10**

**maze\_cols = 15**

**S=DisjointSetForest(maze\_rows\*maze\_cols)**

**size=maze\_rows\*maze\_cols**

**walls = wall\_list(maze\_rows,maze\_cols)**

**n=NumSets(S)**

**def buildMaze1(m,n,G,duplicate=False):#Creates maze using regular find and union**

**count=0**

**while count<m and NumSets(S)>1:**

**count+=1**

**d = random.randint(0,len(walls)-1)**

**C1=walls[d]**

**C2=C1[0]**

**print(C2)**

**C3=C1[1]**

**print(C3)**

**if find(S,C2)!=find(S,C3):**

**if C2<C3 and C3 not in G[C2]:**

**G[C2].append(C3)**

**if duplicate:**

**G[C3].append(C2)**

**(walls.pop(d))**

**union(S,C2,C3)**

**global walls\_2**

**walls\_2=walls**

**return G**

**def buildMaze2(m,n,G,duplicate=False):#creates maze using union\_by\_size**

**count=0**

**while count!=m and NumSets(S)>1:**

**d = random.randint(0,len(walls)-1)**

**C1=walls[d]**

**C2=C1[0]**

**C3=C1[1]**

**if find\_c(S,C2)!=find\_c(S,C3):**

**if C2<C3 and C3 not in G[C2]:**

**G[C2].append(C3)**

**G[C3].append(C2)**

**if duplicate is not False:**

**walls.pop(d)**

**union\_by\_size(S,C2,C3)**

**count+=1**

**global walls3**

**walls3=walls**

**return G**

**def breadth\_first\_search(G,start, end):#used to find shortest path using BFS**

**queue=[(start,[start])]#using queue**

**visited=set()#creates an empty set**

**while queue:**

**(vertex,path)=queue.pop(0)#pop the first number**

**if vertex not in visited:#if vertex has not been visited**

**if vertex==end:#if vertex is the end then return the path**

**return path**

**visited.add(vertex)#adds vertex to the set**

**for next in G[vertex]:#for any number in G[vertex] then append**

**queue.append((next,path+[next]))**

**def depth\_first\_search\_stack(G,S,E):#used to find shortest path using DFS and**

**#stacks**

**stack = [(S,[S])]#creating stack**

**visited=set() #creates a empty set**

**while stack:**

**(vertex,path)=stack.pop()#pop the last number**

**if vertex not in visited:#if vertex has not been visited**

**if vertex==E:#if vertex is equal to the end return the path**

**return path**

**visited.add(vertex)#adds vertex to the set**

**for next in G[vertex]:#for any number in G[vertex] then append**

**stack.append((next,path+[next]))**

**def depth\_first\_search\_recursive(G,S,E,path=[]):#used to find shortest path**

**#using DFS recursively**

**path=path+[S]#adds to the list the path and the vertex S**

**checker=True#used to check**

**if path[-1]!=E:#if the past item in the path is not the n-1**

**for i in G[S]:**

**if S==E:#vertex equals the n-1**

**checker = False#changes vertex to False**

**path2=[]#creates empty path**

**path2=path+[S]#does the same as path**

**return path2#returns path 2**

**if i not in path and checker is not False:**

**path= depth\_first\_search\_recursive(G,i,E,path)#recursive call**

**if path[-1]==E:#if last item in path is n-1 then return path**

**return path**

**else:**

**path.pop(-1)#pop to remove all unneeded items that don't**

**#lead to the path**

**return path**

**#for i in range(len(walls)//2): #Remove 1/2 of the walls**

**# d = random.randint(0,len(walls)-1)**

**# print('removing wall ',walls[d])**

**# walls.pop(d)**

**print('there are',n,'cells')**

**print('Enter number of walls you wish to remove.')**

**while True:**

**try:**

**m = int(input())**

**break**

**except:**

**print("That's not a valid option!")**

**if m<n-1:**

**print("A path from source to destination is not guaranteed to exist.")**

**elif m==n-1:**

**print("There is a unique path from source to destination.")**

**else:**

**print("There is at least one path from source to destination.")**

**draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True)**

**start=time.time()**

**G = [ [] for i in range(n) ]**

**print()**

**F=(buildMaze1(m,n,G,True))**

**print("Starting breadth first search")**

**start1=time.time()**

**print(breadth\_first\_search(F,0,n-1))**

**end1=time.time()**

**print("it takes",end1-start1," to create the breadth first search list")**

**print("converting to edge list")**

**start3=time.time()**

**ELB=ELConverter(breadth\_first\_search(F,0,n-1))**

**end3=time.time()**

**print("takes", end3-start3,"to convert")**

**if ELB is not None:**

**start2=time.time()**

**Draw\_Solution(ELB,walls\_2,maze\_rows,maze\_cols)**

**end2=time.time()**

**print("it takes",end2-start2," to draw the solution")**

**print()**

**print("Starting depth first search using stack")**

**start4=time.time()**

**print(depth\_first\_search\_stack(F,0,n-1))**

**end4=time.time()**

**print("it takes",end4-start4," to create the depth first search list")**

**print("converting to edge list")**

**start5=time.time()**

**ELDFSS=ELConverter(depth\_first\_search\_stack(F,0,n-1))**

**end5=time.time()**

**print("takes",end5-start5,"to convert")**

**if ELDFSS is not None:**

**start6=time.time()**

**Draw\_Solution(ELDFSS,walls\_2,maze\_rows,maze\_cols)**

**end6=time.time()**

**print("it takes",end6-start6,"to draw the solution")**

**print()**

**print("Starting depth first search using recursion")**

**start7=time.time()**

**print(depth\_first\_search\_recursive(F,0,n-1))**

**end7=time.time()**

**print("it takes",end7-start7,"to create the depth first search list")**

**print("converting to edge list")**

**start8=time.time()**

**ELDFSR=ELConverter(depth\_first\_search\_recursive(F,0,n-1))**

**end8=time.time()**

**print("takes",end8-start8,"to convert")**

**if ELDFSR is not None:**

**start9=time.time()**

**Draw\_Solution(ELDFSR,walls\_2,maze\_rows,maze\_cols)**

**end9=time.time()**

**print("it takes",end9-start9,"to draw the solution")**

**print()**

**end=time.time()**

**print("overall time to run all methods using regular union",end-start)**

**#**

**#**

**S=DisjointSetForest(maze\_rows\*maze\_cols)**

**walls = wall\_list(maze\_rows,maze\_cols)**

**#**

**start=time.time()**

**G = [ [] for i in range(n) ]**

**print()**

**F=(buildMaze2(m,n,G,True))**

**print("Starting breadth first search")**

**start1=time.time()**

**print(breadth\_first\_search(F,0,n-1))**

**end1=time.time()**

**print("it takes",end1-start1," to create the breadth first search list")**

**print("converting to edge list")**

**start3=time.time()**

**ELB=ELConverter(breadth\_first\_search(F,0,n-1))**

**end3=time.time()**

**print("takes", end3-start3,"to convert")**

**if ELB is not None:**

**start2=time.time()**

**Draw\_Solution(ELB,walls3,maze\_rows,maze\_cols)**

**end2=time.time()**

**print("it takes",end2-start2," to draw the solution")**

**print()**

**print("Starting depth first search using stack")**

**start4=time.time()**

**print(depth\_first\_search\_stack(F,0,n-1))**

**end4=time.time()**

**print("it takes",end4-start4," to create the depth first search list")**

**print("converting to edge list")**

**start5=time.time()**

**ELDFSS=ELConverter(depth\_first\_search\_stack(F,0,n-1))**

**end5=time.time()**

**print("takes",end5-start5,"to convert")**

**if ELDFSS is not None:**

**start6=time.time()**

**Draw\_Solution(ELDFSS,walls3,maze\_rows,maze\_cols)**

**end6=time.time()**

**print("it takes",end6-start6,"to draw the solution")**

**print()**

**print("Starting depth first search using recursion")**

**start7=time.time()**

**print(depth\_first\_search\_recursive(F,0,n-1))**

**end7=time.time()**

**print("it takes",end7-start7,"to create the depth first search list")**

**print("converting to edge list")**

**start8=time.time()**

**ELDFSR=ELConverter(depth\_first\_search\_recursive(F,0,n-1))**

**end8=time.time()**

**print("takes",end8-start8,"to convert")**

**if ELDFSR is not None:**

**start9=time.time()**

**Draw\_Solution(ELDFSR,walls3,maze\_rows,maze\_cols)**

**end9=time.time()**

**print("it takes",end9-start9,"to draw the solution")**

**print()**

**end=time.time()**

**print("overall time to run all methods using union by size",end-start)**

**“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.”**

**Fernando De Santiago**