Lab 7 Report – Andres Silva - 80588336

This lab consisted of implementing the previously learned knowledge of graphs and different methods of searching through graphs to generate and solve a maze of any given size.

The program consist of a main body that prompts the user to define the size of the maze to be generated, the number of wall the user desires to remove, and to choose between Breath First Search (BFS) or Depth First Search (DFS), which is responsible of finding a path between the first cell and the last cell.

If the user chooses to remove less than half of the walls in the maze, there exists no path, so the program simply terminates. If the user removes exactly (rows x columns)-1 walls, a unique path exists as a solution. Similarly, if the user removes more than those walls, a path is guaranteed to exist, and there might be multiple paths that count as a solution.

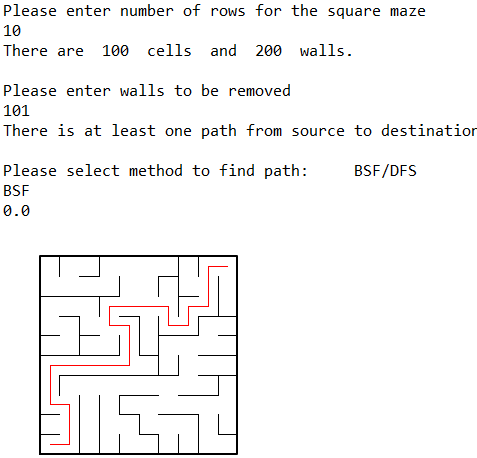
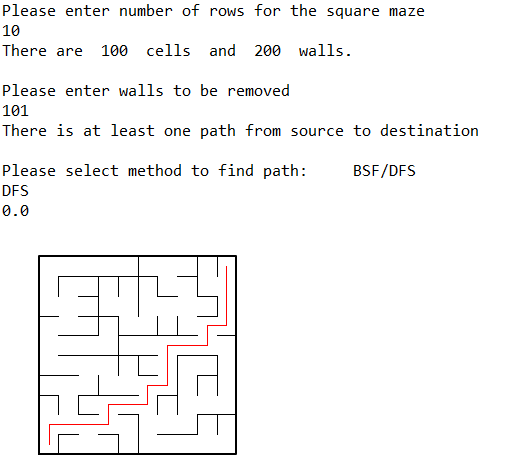
After all the user choices are defined, an edge list is generated by the “Maze\_normal” method, that does not utilize path compression. This edge list is generated by using the Disjoint Set Forest previously used in lab 6. After deleting more than walls than necessary (when the DSF is a single set, the method will not erase more walls), the method will keep deleting walls, but not joining the roots of the sets since they belong to the same set.

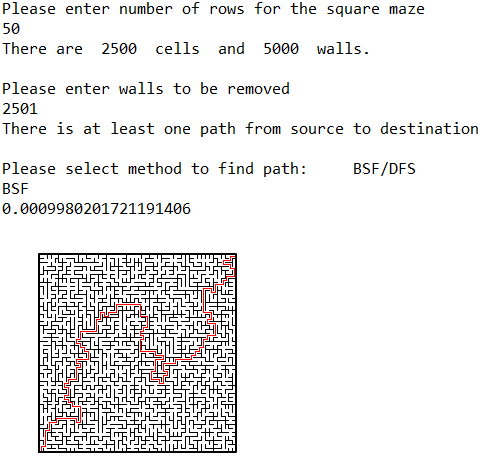
The BFS and DFS methods use and adjacency list (AL) to do their search, so another method that converts the edge list (EL) to an adjacency list was need. With the AL in place, the search can be performed and a prev array returned from the last cell to the first cell. The prev array is the solution to the maze and can be plotted by python plot almost the same way the walls are plotted (horizontal and vertical lines). The only problem is that the plot method takes an edge list, so another method to make an AL to EL was created, taking into consideration the format of [smaller index, larger index].

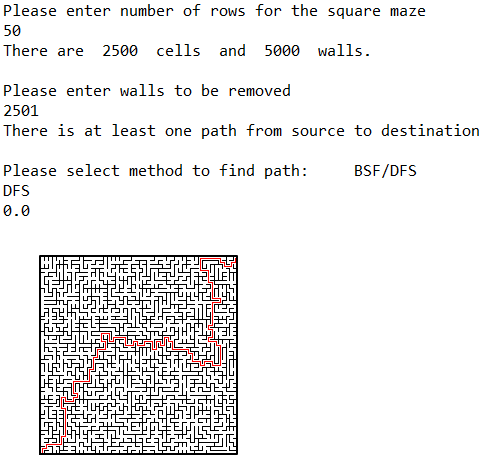
The plotting of the solution represented as an EL was achieved by modifying the plotting method, so that it plots lines from the position of where the numbers are displayed, and the color of the line also was changed to red.

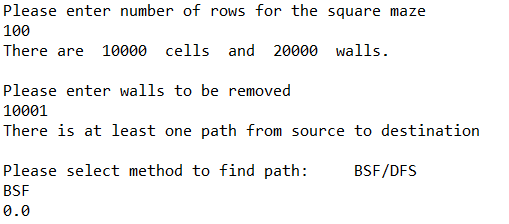
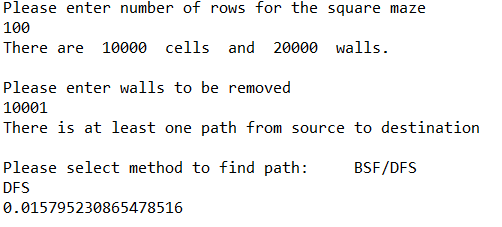
**Results**

The speed of both types of search had different but not consisted times in the test cases (n = 10, 50, 100). Larger numbers were not tested because of the large amount of time it takes the program to generate and delete the cells and walls of the maze.









**Academic Honesty Statement**

“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.”

-Andres Silva.

**Appendix**

'''

Created on Thu April 25 15:16:21 2019

CS 2302 - Andres Silva

> Teacher: Olac Fuentes

> TAs: Anindita Nath & Maliheh Zargaran

> Lab #7

> Create a maze of nxn size and plot a solution using graph implementations.

> LAST MODIFIED: APRIL 28th, 2019

'''

import matplotlib.pyplot as plt

import numpy as np

import random

from dsf import \*

import time

def draw\_maze(EL,walls,maze\_rows,maze\_cols,cell\_nums=False):

fig, ax = plt.subplots()

#Plot line from edge list 'EL'

for l in EL:

if l[1]-l[0] != 1: #vertical wall

lx0 = (l[1]%maze\_cols) +.5

lx1 = lx0

ly0 = (l[1]//maze\_cols) -.5

ly1 = ly0+1

else:#horizontal wall

lx0 = (l[0]%maze\_cols) +.5

lx1 = lx0+1

ly0 = (l[1]//maze\_cols) +.5

ly1 = ly0

ax.plot([lx0,lx1],[ly0,ly1],linewidth=1,color='r')

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: #If not last column

w.append([cell,cell+1]) # wall between adjacent columns

if r!=maze\_rows-1: #if not last row

w.append([cell,cell+maze\_cols]) # wall between adjacent rows

return w

def Maze\_normal(r,c,S,W,m):

i = 0

EL = []

while i < m and i != (r\*c)-1:

d = random.randint(0,len(W)-1) #random index

if find(S,W[d][0]) != find(S,W[d][1]): #If the roots are different,

union(S,W[d][0],W[d][1]) #Join the sets,

EL.append(W.pop(d)) #Delete wall

i += 1

if i == (r\*c)-1: #Handling the infinite loop, delete as many walls as possible, disregarding the DSF

while i < m or i <= len(W):

d = random.randint(0,len(W)-1) #random index

union(S,W[d][0],W[d][1]) #Join the sets,

EL.append(W.pop(d)) #Delete wall

i += 1

return EL

def EdgeList\_to\_AdjList(EL, size):

AL = []

for i in range(size):

AL.append([])

for i in range(len(EL)):

AL[EL[i][0]].append(EL[i][1])

AL[EL[i][1]].append(EL[i][0])

return AL

def BFS(AL,s):

prev = np.zeros(len(AL), dtype = int)-1 #Initialize the prev array

visited = [False] \* len(AL) #Keep track of the visited nodes

Q = [] #Create Queue

Q.append(AL[0][0]) #Append source to Queue and mark it as visited

visited[AL[0][0]] = True

while Q:

if prev[-1] >= 0: #If last node visited, loop is finished

break

s = Q.pop(0) #Get next element in Queue

for i in AL[s]: #For all adjacent vertex to current node,

if visited[i] == False: #Mark it as visited

visited[i] = True

prev[i] = s #Mark the prev array at the neightboor location as s

Q.append(i) #Enqueue the neightboor for future visit

prev[0] = -1 #Make the source node the root.

return prev #Return prev array

def DFS(AL,s):

prev = np.zeros(len(AL), dtype=int)-1 #Initialize prev array

visited = [False] \* len(AL) #Initialize visited array

Stack = [] #Create stack

Stack.append(AL[0][0]) #Push source node and mark it as visited

visited[AL[0][0]] = True

while Stack:

if prev[-1] >= 0: # Break loop at last vertex

break

s = Stack.pop() #retrieve next node to be visited

for i in AL[s]:

if visited[i] == False: #When visiting the neightboor, mark it as true

visited[i] = True

prev[i] = s #Keep track in the prev array

Stack.append(i) #Push neightboor

if Stack == []:

S.append(AL[0][1])

return prev

def prev\_to\_EL(prev):

EL = [] #Create edge list

i = len(prev)-1 #Start from last node

while prev[i] >= 0: #While not reaching the end

if i < prev[i]: #Decide direction of line

edge = [i,prev[i]] #Create an edge form current index to prev[i]

else:

edge = [prev[i],i]

i = prev[i] #Update index to content of array

EL.append(edge) #append to edge list

EL.append([0,i]) #Finally, create a node with 0 to last known index

return EL

"""

MAIN \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

"""

def main():

r = int(input("Please enter number of rows for the square maze\n"))

c = r

print("There are ", r\*c, " cells", " and ", (r\*c\*2)," walls.")

m = int(input("Please enter walls to be removed\n"))

if m < (r\*c) - 1:

print("A path from source to destination is not guaranteed to exist, terminating program...")

return

elif m == (r\*c) - 1:

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

else:

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

selection = input("Please select method to find path: BSF/DFS\n")

W = wall\_list(r,c)

S = DisjointSetForest(r \* c)

Edge\_List = Maze\_normal(r,c,S,W,m)

AL = EdgeList\_to\_AdjList(Edge\_List, r\*c)

if selection == "BSF":

start = time.time()

prev = BFS(AL,0)

path = prev\_to\_EL(prev)

end = time.time()

print(end - start)

elif selection == "DFS":

start = time.time()

prev = DFS(AL,0)

path = prev\_to\_EL(prev)

end = time.time()

print(end - start)

else:

print("Invalid input, exiting..")

return

# draw\_maze(path,W,r,c)

main()

#plt.close("all")