Tyler Salas

1:30-2:50 Class

Dr. Fuentes

80596221

Lab 7 Report

Lab 7, the lab assigned this week, aimed to implement the use of breadth first search and depth first search. We were required to take a randomly generated maze that we have made in past labs, and from there implement the methods to create a path from the start to the finish. The start being the bottom left corner and end being top right. In addition we also had to display a few statistics on the created mazes.

The idea was to follow the psuedo-code provided by doctor Fuentes for the two algorithms and implement them into the python language, solving the randomly generated mazes we create.

For the first part, I display a few stats that follow the following parameters

A path from source to destination is not guaranteed when m < #cells-1

There is a unique path from source to destination when m = #cells-1

There is at least one path from source to destination when m > #cells-1

I get the variable cells by multiplying the number of rows by the number of columns in

the maze. What this does is return the number of cells and displays it in the variables place. A path won’t be guaranteed if m is less because not enough walls will be popped. If it is equal there will be a single path from source to destination and if it is greater there will be at least one path, if not more.

The creation of the adjacency list starts with the declaration of an empty list with n number of empty lists inside where n is decided by the number of rows by the number of columns, or the number of cells. From here, in the while loop that creates the maze rows and columns, whenever a wall is popped, the walls are joined. A function creates an edge by joining one element of the wall to the other at both points in the list. By the end of the while loop the correct adjacency list is created.

From here the three sorting methods are tested. Those three being breadth first search, depth first search, and depth first search with recursion.

Breadth First Search:

0 - 1 - 2 - 17 - 32 - 33 - 34 - 35 - 36 - 37 - 52 - 53 - 68 - 83 - 98 - 99 - 100 - 115 - 130 - 145 - 146 - 147 - 148 - 133 - 134 - 149

Runtime: 0.008007049560546875

Depth First Search:

0 - 1 - 2 - 17 - 32 - 33 - 34 - 35 - 36 - 37 - 52 - 53 - 68 - 83 - 98 - 99 - 100 - 115 - 130 - 145 - 146 - 147 - 148 - 133 - 134 - 149

Runtime: 0.0

Depth First Search With Recursion:

0 - 1 - 2 - 17 - 32 - 33 - 34 - 35 - 36 - 37 - 52 - 53 - 68 - 83 - 98 - 99 - 100 - 115 - 130 - 145 - 146 - 147 - 148 - 133 - 134 - 149

Runtime: 0.0020487308502197266

Breadth First Search starts by declaring different lists, one being visited the length of G all set to false, A list called prev all set to –1, an empty Q that will be used to keep track of items to check. So it repeats a loop as long as the Q is not empty. The element Q is popped and that point in G is evaluated, if it is false it is then set to true, u is added to prev and t is added to the first position of Q. From all this previous is returned and this will be the list needed to print the path.

Depth first does the same thing except when an item is added to Q it is added to the end of the list, making it a stack but following the same process.

Runtimes

10x15

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| Breadth First | 0.0020809173583982 | .00399684906005859 | 0.0039939880371093 |
| Depth First | 0.0039691925048822 | 0.0021159648895263 | 0.0010991096496582 |
| DF Rec | 0.0039865970611572 | 0.0040020942687988 | 0.0039460659027099 |

20x30

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| Breadth First | 0.0060386657714843 | 0.0039856433868403 | 0.0080258846282958 |
| Depth First | 0.0039782524108886 | 0.0060825347900390 | 0.0039930343627929 |
| DF Rec | 0.0123820304870605 | 0.0361778736114501 | 0.0040328502655029 |

From the data of the runtimes that was collected, we can see that the depth first search with recursion seemed to take longer than the depth first and breadth first. While the Breadth First and Depth First seemed to jump around in their efficiencies. Im sure this is due to the fact of that randomization of the mazes. The O runtime of both breadth first and depth first would be a O(n^2) due to the fact of there being two loops in the methods.

In conclusion it was interesting to witness the implementation and use of these two algorithms and it was a good study in the way of the methods.

#CS2302

#Tyler Salas

#Lab7

#Dr.Fuentes

#Anindita Nath

#Create Mazes and evaluate with graphs

import matplotlib.pyplot as plt

import numpy as np

import random

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 find\_c(S,i):

if S[i] <= 0:

return i

s = i

while S[i] >= 0:

i = S[i]

root = i

while S[s] >= 0:

p = S[s]

S[s] = root

s = p

return root

def union(S,i,j):

# Joins i's tree and j's tree, if they are different

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 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)

fig.savefig('square.png')

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

def moreOneSet(S):

count = 0

for i in range(len(S)):

if S[i] < 0:

count += 1

if count > 1:

return True

return False

def printPath(prev,v):

if prev[v] != -1:

printPath(prev,prev[v])

print("-",end=' ')

print(v,end=' ')

#Adds an element to an adjacency list

def addToList(L,x,y):

L[x].append(y)

L[y].append(x)

def breadthFirstSearch(G,v):

visited = [False for i in range(len(G))]

prev = [-1 for i in range(len(G))]

Q = []

Q.insert(0,v)

visited[v] = True

while len(Q) > 0:

#print(Q)

u = Q.pop()

for t in G[u]:

if visited[t] == False:

visited[t] = True

prev[t] = u

Q.insert(0,t)

return prev

def depthFirstSearch(G,v):

visited = [False for i in range(len(G))]

prev = [-1 for i in range(len(G))]

Q = []

Q.append(v)

visited[v] = True

while len(Q) > 0:

#print(Q)

u = Q.pop()

for t in G[u]:

if visited[t] == False:

visited[t] = True

prev[t] = u

Q.append(t)

return prev

def depthFirstSearchRec(G,v,visited,prev,Q):

if not True in visited:

Q = []

Q.append(v)

visited[v] = True

if len(Q) > 0:

u = Q.pop()

for t in G[u]:

if visited[t] == False:

visited[t] = True

prev[t] = u

Q.append(t)

depthFirstSearchRec(G,v,visited,prev,Q)

plt.close("all")

maze\_rows = 35

maze\_cols = 45

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

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

aList = [[] for i in range(maze\_rows\*maze\_cols)]

'''

print("A path from source to destination is not guaranteed when m < ",(maze\_rows\*maze\_cols)-1)

print("There is a unique path from source to destination when m = ",(maze\_rows\*maze\_cols)-1)

print("There is at least one path from source to destination when m > ",(maze\_rows\*maze\_cols)-1)

m = input("How many walls shall be popped: ")

m = int(m)

'''

m = (maze\_rows\*maze\_cols)-1

count = 0

#Builds maze

while moreOneSet(S) and count < m:

c1 = random.randint(0,len(walls)-1)

if find(S,walls[c1][0]) != find(S,walls[c1][1]):

#Adds element to adjacency list representing graph

addToList(aList,walls[c1][0],walls[c1][1])

#uniting two components in the dsf

union(S,walls[c1][0],walls[c1][1])

#removing walls

walls.pop(c1)

#Adding to keep track of m

count += 1

#Breadth First Search

start = time.time()

print("Breadth First Search: ")

G = breadthFirstSearch(aList,0)

printPath(G,len(aList)-1)

end = time.time()

print()

print("Runtime: ",end-start)

print()

#Depth First Search

start = time.time()

print("Depth First Search: ")

G = depthFirstSearch(aList,0)

printPath(G,len(aList)-1)

end = time.time()

print()

print("Runtime: ",end-start)

print()

#Depth First Search With Recursion

start = time.time()

print("Depth First Search With Recursion: ")

visited = [False for i in range(len(aList))]

prev = [-1 for i in range(len(aList))]

Q = []

depthFirstSearchRec(aList,0,visited,prev,Q)

printPath(prev,len(aList)-1)

end = time.time()

print()

print("Runtime: ",end-start)

print()

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

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.



-Tyler Salas