PAOLA TERRAZAS

CS2302

SOLVING A MAZE USING DIFFERENT SEARCH METHODS

DR. FUENTES

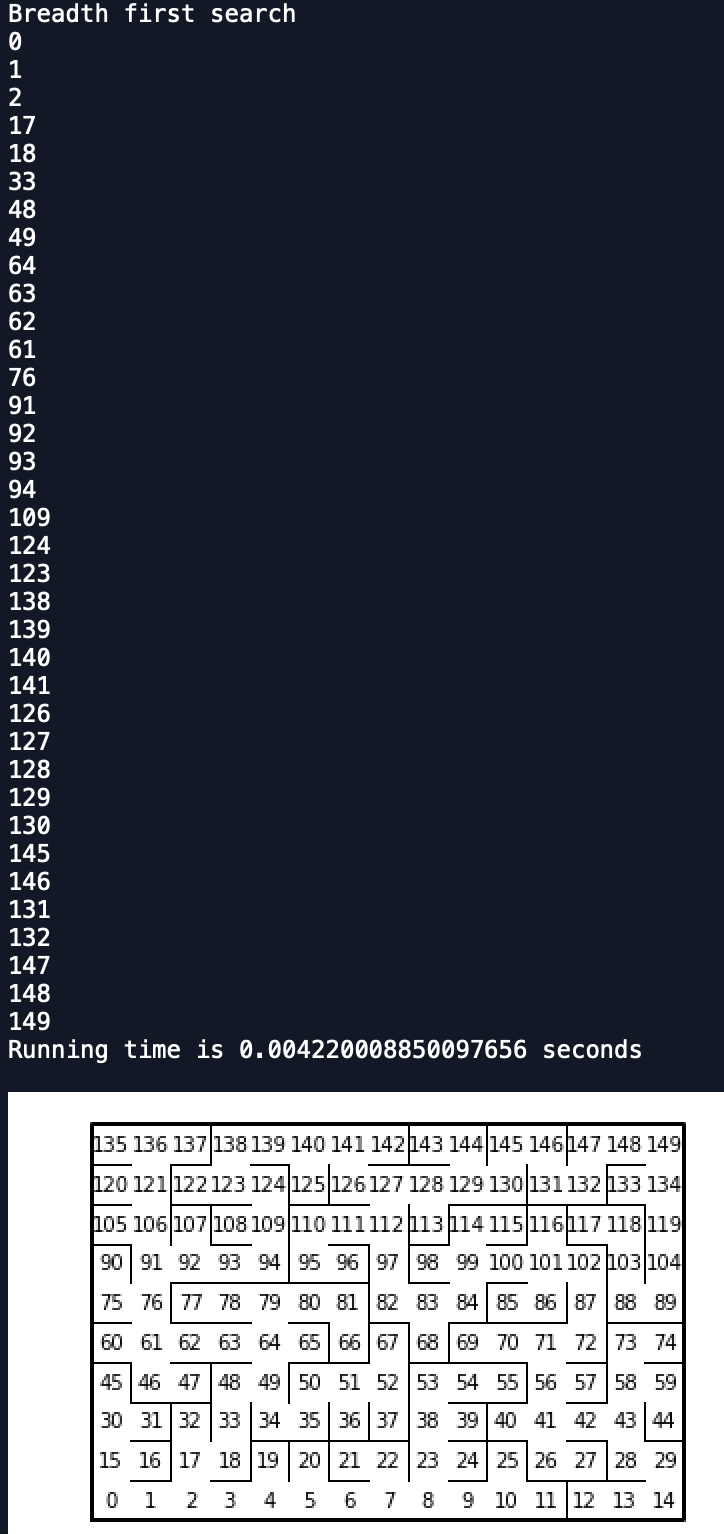
TA – ANINDITA NATH

5.1.2019

LAB 7 REPORT

In our seventh lab we were asked to modify our lab 6 to allow if removing less than n -1 walls, some cells will not be reachable from the start cells. Also, if you remove more than n-1 walls, you would have multiple paths from the source to the destination. We also needed to write a method to build the adjacency list representation of our maze as well as implement breadth-first search, depth-first search and depth-first search using recursion. This lab was very challenging for me. I was able to create the adjacency list, that part being very simple. Then, I went on to creating the breadth first search I found easiest. I used the code from my lab 6, using my normalMaze method. It begins by declaring q, an empty list. I also have a variable known as visited that is the length of G equal to False and my previous list all set to -1. There is a while loop that continues looping as long as the length of q is not 0 and the element q is popped and if the i in G is false then it is set to True because its visited and queue is added to previous and the i is then inserted in the first position of q, returning previous.

My results are as follows for breadth-first search:



The list that is printed is the path that was found using my printPath method. The running time is also printed. I attempted the depth-first search doing what the professor suggested in the lab description, using the same code from breadth-first search but the queue is replaced by a stack, but I was only getting more confused. Therefore, I attempted to write a new method from scratch. My method took G and v as parameters, then created and empty listed named visited like my other method and a stack with v. A while loop that continues if the stack is not empty, then it created a variable u, which would pop the item in the stack. After that is a for loop for i in every u in G, if i was not visited the stack would append that element, otherwise visited would append u. Lastly, it returns visited. When I attempted to run my code, it would result in error with my normalMaze method. I could not figure the issue. I also attempted to do the recursion depth-first search but I also had several issues.

But more importantly than getting my code to run correctly using my maze, I learned the importance and difference between depth-first search and breadth-first search. Although, my depth-first search method did not run, I can collect big 0 with the code I wrote for it. Both my method are constant O(n) considering it depends on the size.

#SOURCE CODE

#PAOLA TERRAZAS

#DR.FUENTES

import matplotlib.pyplot as plt

import numpy as np

import random

import time

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

def numOfSets(S):

n =0

for i in range(len(S)):

if S[i] <0:

n+=1

return n

def DisjointSetForest(size):

return np.zeros(size,dtype=np.int)-1

def find(S,i):

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 compMaze(row,col,S,walls):

while numOfSets(S)>1:

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

if find\_c(S,walls[c][0])!= find\_c(S,walls[c][1]):

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

walls.pop(c)

def adjList(l,v,y):

l[v].append(y)

l[y].append(v)

def union(S,i,j):

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj:

S[rj] = ri

def normalMaze(row,col,S,walls):

count = 0

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

while count < maze:

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

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

adjList(l,walls[c][0],walls[c][1])

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

walls.pop(c)

count +=1

def printPath(prev,v):

if prev[v] != -1:

printPath(prev,prev[v])

print(v)

def breadthFirst(G,v):

q = []

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

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

q.insert(0,v)

visited[v] = True

while len(q) is not 0:

queue = q.pop(0)

for i in G[queue]:

if visited[i] is False:

visited[i] = True

previous[i] = queue

q.insert(0,i)

return previous

def depthFirst(G,v):

visited = []

stack = [v]

while stack != []:

u = stack.pop()

for i in G[u]:

if i not in visited:

stack.append(i)

visited.append(u)

return visited

plt.close("all")

maze\_rows = 10

maze\_cols = 15

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

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

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

starttime = time.time()

print("Breadth first search ")

normalMaze(maze\_rows, maze\_cols,S,walls)

B = breadthFirst(l,0)

printPath(B,len(l)-1)

endtime = time.time()

print('Running time is', endtime-starttime, 'seconds')

#starttime = time.time()

#print("Depth first search ")

#normalMaze(maze\_rows, maze\_cols,S,walls)

#printPath(D,len(l)-1)

#endtime = time.time()

#print()

#print('Running time is', endtime-starttime, 'seconds')

#print()

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

Scholastic Dishonesty

Any student who commits an act of scholastic dishonesty is subject to discipline. Scholastic dishonesty includes, but not limited to cheating, plagiarism, collusion, the submission for credit of any work or materials that are attributable to another person.

* **Cheating**
  + Copying form the test paper of another student
  + Communicating with another student during a test
  + Giving or seeking aid from another student during a test
  + Possession and/or use of unauthorized materials during tests (i.e. Crib notes, class notes, books, etc)
  + Substituting for another person to take a test
  + Falsifying research data, reports, academic work offered for credit
* **Plagiarism**
  + Using someone’s work in your assignments without the proper citations
  + Submitting the same paper or assignment from a different course, without direct permission of instructors
* **Collusion**
  + Unauthorized collaboration with another person in preparing academic assignments

Sign \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Date: 4/16/2019