Robert Marc

80487972

CS 2302

MW 1:30-3:00

Lab 5 Report

Introduction

The purpose of this lab was to modify the code from lab 6 and implement search algorithms to find paths that can solve the maze.

Proposed Solutions

While building the maze, an adjacency list is built alongside the wall list. This adjacency list is then passed through the search algorithms and an array that when traversed in reverse, contains the path for the maze.

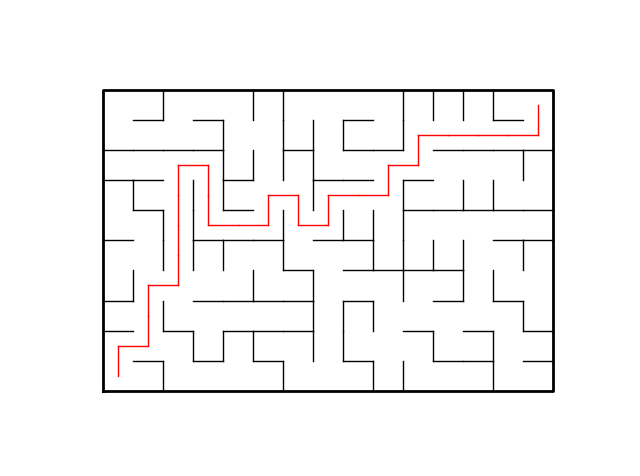
Experimental Results

For N cells:

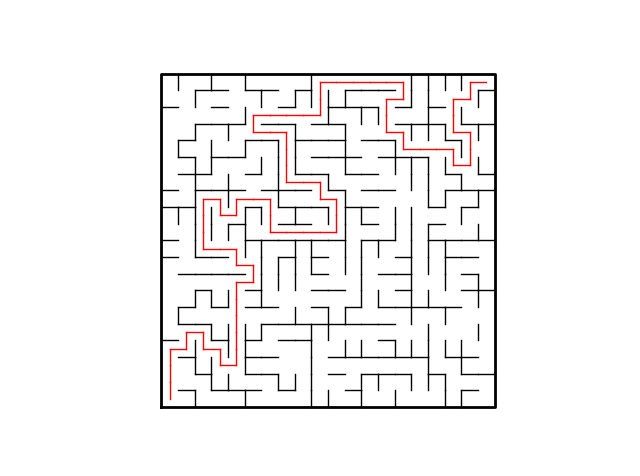
Average Number of Operations for a given N across 10 tests

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N = | 100 | 225 | 400 | 624 |
| breadthFirst | 85.3 | 180.3 | 369.8 | 496.4 |
| depthFirst | 71.9 | 121.1 | 276.7 | 411.4 |
| depthFirstR | 99 | 224 | 399 | 624 |

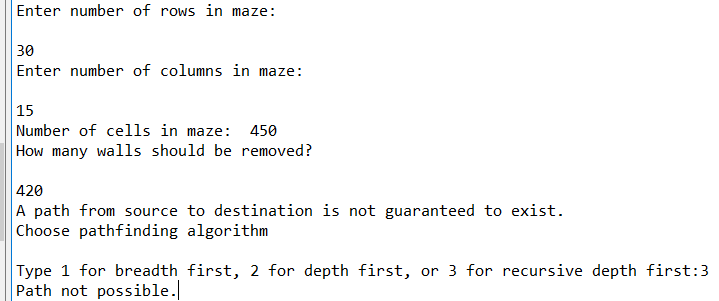
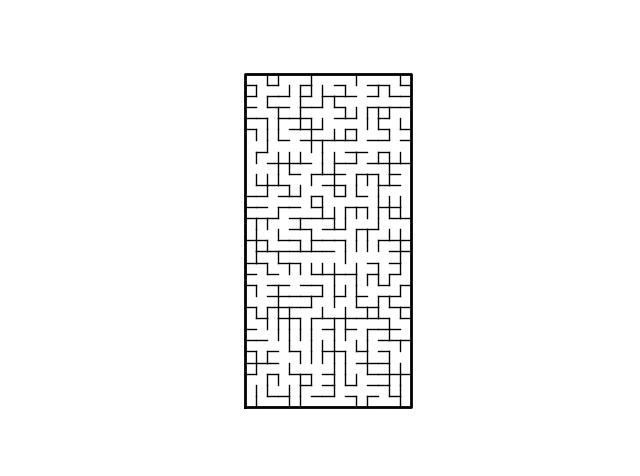
Outputs



10x15 – breadthFirst search used, 149 walls removed, 150 cells total



20x20 – depthFirst search used, 415 walls removed, 400 cells total



30x15 – depthFirstR used, 420 walls removed, 450 cells total, path from 0 to 449 not possible

Conclusions

The hardest part of this lab was getting the method to draw the correct path. The searches were easy enough to figure out, though I wonder if I implemented the recursive depth first search incorrectly. It always takes N-1 comparisons to find a correct path. Other than that, I implemented a method that checks that a path is possible from beginning to end, and stops the program from running the drawPath method if a path does not exist. All of the searches completed to fast for time.time() to properly measure the elapsed time.

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.”



Appendix

# 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

"""

@Course: CS2302 MW 1:30-2:50 pm

@Author: Robert Marc, 80487972

@Assignment: Lab 7

@Instructor: Dr. Olac Fuentes

@TAs: Anindita Nath and Maliheh Zargaran

@Date of Last Modification: 4/29/19 @11:50 PM

@Purpose: To modify lab 6 code and implement pathfinding search algorithms

to solve the mazes constructed

"""

import matplotlib.pyplot as plt

import numpy as np

import random

import time

from scipy import interpolate

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

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 DisjointSetForest(size):

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

def dsfToSetList(S):

#Returns aa list containing the sets encoded in S

sets = [ [] for i in range(len(S)) ]

for i in range(len(S)):

sets[find(S,i)].append(i)

sets = [x for x in sets if x != []]

return sets

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): #Find with path compression

if S[i]<0:

return i

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

S[i] = r

return r

def union(S,i,j):

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

"""

Modified to return True if a union is made, or False if nothing is changed

"""

ri = find(S,i)

rj = find(S,j)

if ri!=rj:

S[rj] = ri

return True

return False

def union\_c(S,i,j):

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

# Uses path compression

"""

Modified to return True if a union is made, or False if nothing is changed

"""

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj:

S[rj] = ri

return True

return False

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

"""

Modified to return True if a union is made, or False if nothing is changed

"""

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

return True

else:

S[ri] += S[rj]

S[rj] = ri

return True

return False

def draw\_dsf(S):

scale = 30

fig, ax = plt.subplots()

for i in range(len(S)):

if S[i]<0: # i is a root

ax.plot([i\*scale,i\*scale],[0,scale],linewidth=1,color='k')

ax.plot([i\*scale-1,i\*scale,i\*scale+1],[scale-2,scale,scale-2],linewidth=1,color='k')

else:

x = np.linspace(i\*scale,S[i]\*scale)

x0 = np.linspace(i\*scale,S[i]\*scale,num=5)

diff = np.abs(S[i]-i)

if diff == 1: #i and S[i] are neighbors; draw straight line

y0 = [0,0,0,0,0]

else: #i and S[i] are not neighbors; draw arc

y0 = [0,-6\*diff,-8\*diff,-6\*diff,0]

f = interpolate.interp1d(x0, y0, kind='cubic')

y = f(x)

ax.plot(x,y,linewidth=1,color='k')

ax.plot([x0[2]+2\*np.sign(i-S[i]),x0[2],x0[2]+2\*np.sign(i-S[i])],[y0[2]-1,y0[2],y0[2]+1],linewidth=1,color='k')

ax.text(i\*scale,0, str(i), size=20,ha="center", va="center",

bbox=dict(facecolor='w',boxstyle="circle"))

ax.axis('off')

ax.set\_aspect(1.0)

def maze(remove):

"""

Modified maze construction using union by size with path compression

Now builds an adjacency list alongside the DSF and wall list operations

"""

adjList = []

for i in range(rows\*cols):

adjList.append([])

while remove > 0 and len(dsfToSetList(mazeDSF)) > 1:

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

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

if union\_by\_size(mazeDSF,walls[d][0],walls[d][1]):

remove -= 1

adjList[walls[d][0]].append(walls[d][1])

adjList[walls[d][1]].append(walls[d][0])

walls.pop(d)

while remove > 0:

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

remove -= 1

adjList[walls[d][0]].append(walls[d][1])

adjList[walls[d][1]].append(walls[d][0])

walls.pop(d)

return adjList

def printPath(path,v):

"""

Prints the appropriate path to take to solve the maze

"""

if path[v] != -1:

printPath(path,path[v])

print(' -> ',end='')

print(v,end='')

def checkPath(path,v):

if path[v] != -1:

return checkPath(path,path[v])

if v == 0:

return True

def drawPath(path,rows,cols,v,ax):

"""

Draws the appropriate path to take to solve the maze

"""

if path[v] != -1 and path[path[v]] != -1:

#print("path[v]=",path[v])

#print("path[path[v]]=",path[path[v]])

#print("path[v]-path[path[v]]=",path[v]-path[path[v]])

#print()

drawPath(path,rows,cols,path[v],ax)

if path[v] - path[path[v]] == 1: #horizontal path going from right to left

x0 = (path[v]%cols)-0.5

x1 = x0+1

y0 = (path[v]//cols)+0.5

y1 = y0

elif path[v] - path[path[v]] == -1: #horizontal path going from left to right

x0 = (path[path[v]]%cols)-0.5

x1 = x0+1

y0 = (path[path[v]]//cols)+0.5

y1 = y0

elif path[v] - path[path[v]] > 1: #vertical path going from top to bottom

x0 = (path[v]%cols)+0.5

x1 = x0

y0 = (path[v]//cols)-0.5

y1 = y0+1

else: #vertical path going from bottom to top

x0 = (path[path[v]]%cols)+0.5

x1 = x0

y0 = (path[path[v]]//cols)-0.5

y1 = y0+1

ax.plot([x0,x1],[y0,y1],linewidth=1,color='red')

if v == len(path)-1:

if v - path[v] == 1: #horizontal path

x0 = (v%cols)-0.5

x1 = x0+1

y0 = (v//cols)+0.5

y1 = y0

else: #vertical path

x0 = (v%cols)+0.5

x1 = x0

y0 = (v//cols)-0.5

y1 = y0+1

ax.plot([x0,x1],[y0,y1],linewidth=1,color='red')

def breadthFirst(adjList):

global comps

"""

Breadth first search, visits a cell, then all of its neighbers, and so on

until all cells have been searched

"""

visited = np.full(len(adjList),False,dtype=bool)

path = np.zeros(len(adjList),dtype=int)-1

q = []

goal = len(adjList)-1

q.append(0)

visited[0] = True

while len(q) > 0:

v = q.pop(0)

if v == goal:

return path

for adj in adjList[v]:

if not visited[adj]:

comps += 1

visited[adj] = True

path[adj] = v

q.append(adj)

return path

def depthFirst(adjList):

"""

Depth first search, visits a cell and the first of its neighbors, then

continues to the farthest cell it can reach, then backs up and searches

to the next farthest cell it can reach

"""

global comps

visited = np.full(len(adjList),False,dtype=bool)

path = np.zeros(len(adjList),dtype=int)-1

s = []

goal = len(adjList)-1

s.append(0)

visited[0] = True

while len(s) > 0:

v = s.pop()

if v == goal:

return path

for adj in adjList[v]:

if not visited[adj]:

comps += 1

visited[adj] = True

path[adj] = v

s.append(adj)

return path

def depthFirstR(adjList,v):

"""

Recursive depth first search, same operation as normal depth first search,

but instead of using a stack, it moves through the first adjacency in

each cell before returning to move through to the next one

"""

global visited

global prev

global comps

visited[v] = True

for adj in adjList[v]:

if not visited[adj]:

comps += 1

prev[adj] = v

depthFirstR(adjList,adj)

return prev

plt.close("all")

fig, ax = plt.subplots()

"""

Queries for input on maze construction

"""

print("Enter number of rows in maze: ")

rows = int(input())

print("Enter number of columns in maze: ")

cols = int(input())

cells = rows\*cols

print("Number of cells in maze: ",cells)

print("How many walls should be removed?")

remove = int(input())

"""

Prints information on if there is a possible path in the maze.

"""

if remove < cells-1:

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

elif remove == cells-1:

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

else:

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

"""

Maze construction and drawing

"""

walls = wall\_list(rows,cols)

mazeDSF = DisjointSetForest(rows\*cols)

adjList = maze(remove)

draw\_maze(ax,walls,rows,cols)

"""

Queries for input on which pathfinding algorithm to use to solve the maze

"""

comps = 0

print("Choose pathfinding algorithm")

choice = input("Type 1 for breadth first, 2 for depth first, or 3 for recursive depth first:")

if choice== "1":

start = time.time()

path = breadthFirst(adjList)

elapsed = time.time() - start

if checkPath(path,len(path)-1):

printPath(path,len(path)-1)

drawPath(path,rows,cols,len(path)-1,ax)

print()

else:

print("Path not possible.")

elif choice == "2":

start = time.time()

path = depthFirst(adjList)

elapsed = time.time() - start

if checkPath(path,len(path)-1):

printPath(path,len(adjList)-1)

drawPath(path,rows,cols,len(adjList)-1,ax)

print()

else:

print("Path not possible.")

elif choice == "3":

visited = np.full(len(adjList),False,dtype=bool)

prev = np.zeros(len(adjList),dtype=int)-1

start = time.time()

path = depthFirstR(adjList,0)

elapsed = time.time() - start

if checkPath(path,len(path)-1):

printPath(path,len(adjList)-1)

drawPath(path,rows,cols,len(adjList)-1,ax)

print()

else:

print("Path not possible.")

else:

print("Input not recognized, please try again.")

print("Elapsed time: ",elapsed)

print("Comparisons: ",comps)