Ericka Najera

30 April 2019

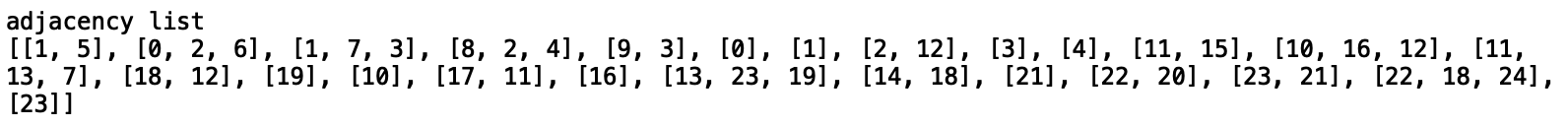
Dr. Fuentes

CS 2302 Lab 7

Lab 7 consisted of modifying lab 6 to find a solution to the maze I provided. The methods were to ask a user how many walls to remove and print a message indicating if the number of walls would be enough to find a path from point a to point b. The second method was to get the cells as vertices and indicate the edges between then and return an adjacency list. The following three methods was to create a solution to the maze using breadth first search, depth first search iterative and depth first search recursively.

In other to ask the user for how many walls to remove I ask the user for the input and they I would place three if statements to check if m > n-1, m<n-1 and m=n-1, and prints a message indicating if there is a path or not with the number of walls being removed. Then I changed by while loop from running until there was only one set to a for loop to run while the number of walls they want to take down. In order to receive a completely maze I have to input a large number. To create the maze, I have to get a random number then get the adjacent cells to it and choose one randomly. Then I check if there is a union between those two cells, if there isn’t then we do a union and pop the wall and append it to another list which would be the visited or edge list. From there I would get the adjacent cell I choose and make it the new cell to keep the while loop going, if somehow there are no more adjacent to that cell then we choose another cell, which at one point two different paths will union together create a complete maze.

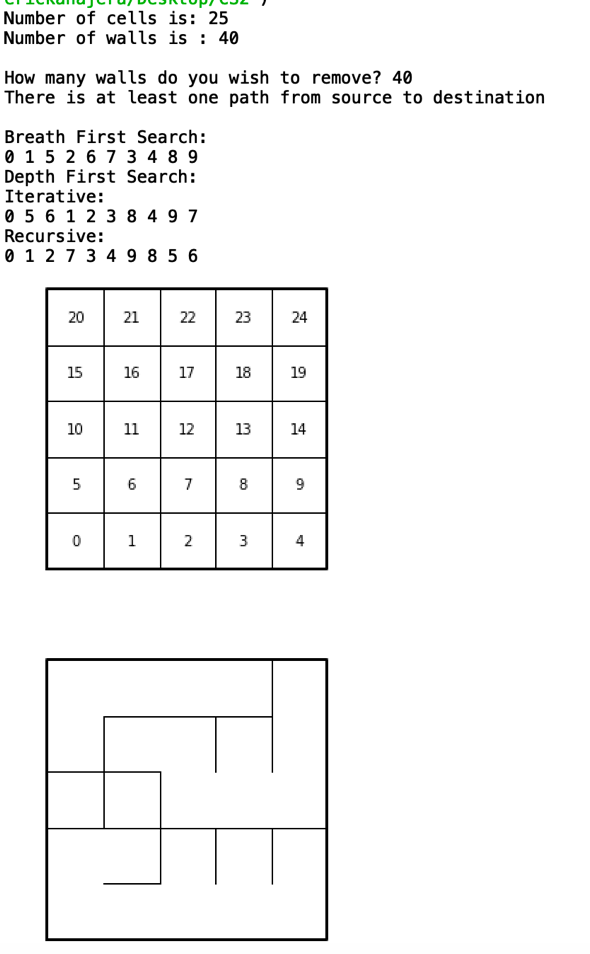
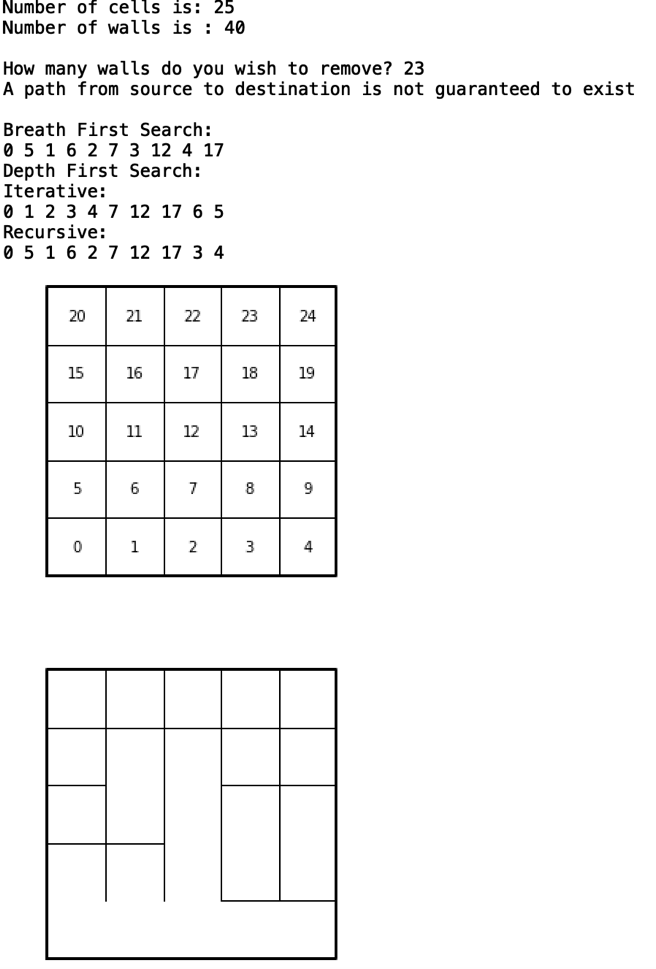
The adjacency list can be created by getting the popped walls from the maze and creating an adjacent list from there. The way I did it was to create a method that would look for the edges that contained cell n, then I would store them into an array. From there I would create another function that would go through a for loop for how many vertices or cells there are. Then I would call the find adjacent method to get that list for that specific index then I would simply append the list creating an adjacency list.

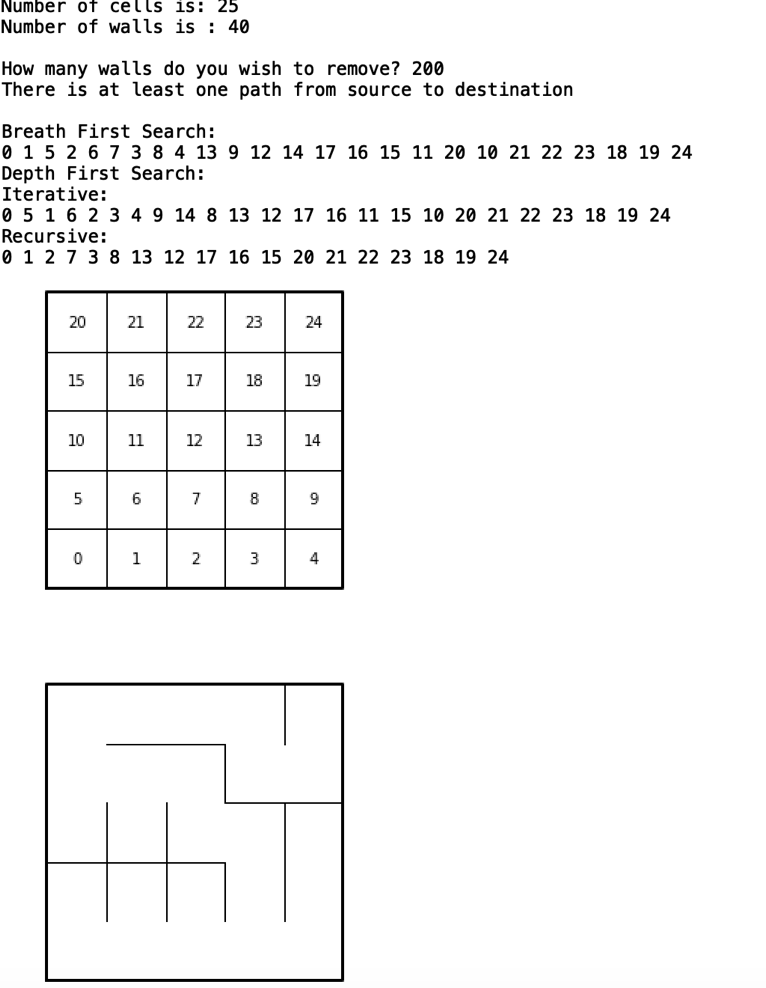


The function to get the breadth first search is to use a queue and every time you dequeue on number you print it. In order to avoid repetitions or a never-ending loop I would have to create a visited array in this case I use 0 if it was not visited and 1 if visited. Then I would create a queue. Since my maze would need to find a solution from the bottom left corner to the top right corner, my solution would go from cell zero to the last cell. I would create a while loop to run until the queue is not empty, in order for the queue to not be empty in the beginning I would enqueue the starting cell. Inside the while loop I will pop the first item of the queue and print it. Then if that cell was my goal, I would terminate my method if not then I would create a for loop go through the adjacent cells and check if those cells are visited. If they are visited, I would ignore them, if they are unvisited then I would enqueue them. My output would result in the printed solution to my maze.

The method to get the depth first search iteratively is to use a stack instead of queue and every time you pop a number you print it. In order to avoid repetitions or a never-ending loop I would have to create a visited array as well, in this case I use 0 if it was not visited and 1 if visited. Then I would create a stack. My solution would go from cell zero to the last cell of the maze. I would create a while loop to run until the stack is not empty, in order for the stack to not be empty in the beginning I would pop the last cell. Inside the while loop I will pop the first item of the stack and print it. Then if that cell was my goal cell, I would terminate my while loop, if not then I would create a for loop go through the adjacent cells and check if those cells are visited. If they are visited, I would ignore them, if they are unvisited then I would append them. My output would result in the printed solution to my maze using depth first search.

In order to get the depth first search recursively, I would do the same thing just adjust a few things. My parameters would be the start point, an empty list to store the solution, the adjacency list, and the visited array to get if it is 0 or 1. In the begging of my method I would make sure my start cell or the cell it is currently on gets marked as visited. Then instead of printing, I would append it to the maze list. Just like the other function I would do a traverse through the adjacent cells of the current cell and send them recursively if there aren’t visited. Somehow, I couldn’t terminate the method after my goal was reached because of the other recursive calls, but I decided to store the solution in a list that way I can traverse that list and print until my goal is reached. That way the rest of the depth first search is not shown after I reach my goal.





Running Times:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | size |  |  |
|  | 25 | 64 | 100 | 500 | 1000 |
| program | ﻿38140000 | ﻿72887000 | ﻿127019000 | ﻿596107000 | ﻿1446004000 |
| breadth first search | ﻿887000 | ﻿1830000 | ﻿3404000 | ﻿3701000 | ﻿44137000 |
| dfs iterative | ﻿524000 | ﻿1321000 | ﻿905000 | ﻿5287000 | ﻿60030000 |
| dfs recursive | ﻿641000 | ﻿1015000 | ﻿1442000 | ﻿7515000 | ﻿57665000 |

Breadth first search: O(V + E)

Depth first search:

Iterative: O(V + E)

Recursive: O(V + E)

Appendix:

﻿ # Ericka Najera Lab 7 MW 10:30-11:50

#Professor Fuentes CS2302

#Lab 7 implement Breath first search and depth first search

#Implementation of disjoint set forest

# Programmed by Olac Fuentes

# Last modified March 28, 2019

import matplotlib.pyplot as plt

import numpy as np

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): #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

ri = find(S,i)

rj = find(S,j)

if ri!=rj:

S[rj] = ri

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:

S[rj] = ri

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

def NumSets(S):

sets = 0

count = np.zeros(len(S),dtype = int)

for i in range(len(S)):

if S[i]<0:

sets+=1

count[find(S,i)] +=1

return sets

######################################################################

# 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 random

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 findAdjacent(walls,n):

#returns a list of the adjacent walls to n

adjacent = []

#searches the walls and creates a list with the adjacent cells to that index

for i in walls:

if i[0]==n:

adjacent.append(i[1])

if i[1]==n:

adjacent.append(i[0])

return adjacent

def adjacentList(walls,maze\_rows,maze\_cols ):

#creates a 2d array with a the adjacent walls of that index list

adj = [ ]

#for each index it finds the list of that cell in the findadjacent method

for i in range(maze\_rows\*maze\_cols):

adj.append(findAdjacent(walls,i))

return adj

def findWall(walls,c,r):

#return the index of the wall that unites c and r

for i in range(len(walls)):

if walls[i]==[c,r] or walls[i] == [r,c]:

return i

return None

def maze(S,walls,adjacents):

#asks the user for how many walls to remove

#displays the approriate term for n-1 walls

print('Number of cells is:',len(S))

print('Number of walls is :',len(walls))

m = int(input("How many walls do you wish to remove? "))

type(m)

if m <(len(walls)-1):

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

if m ==(len(walls)-1):

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

if m>(len(walls)-1):

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

#selets a random cell

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

#creates a visited array to store walls as they are popped which is an edge list

visited =[]

#originally this runs until there one cell

#this is modified to remove how many walls the user wants

for i in range(m):

#selects a random cell that is adjacent to the current cell

r = random.randint(0,len(adjacents[d])-1)

#checks for the union between them

if find(S,d)!=find(S,adjacents[d][r]):

#makes a union

union(S,d,adjacents[d][r])

#append the wall to the visted

visited.append(walls.pop(findWall(walls,d,adjacents[d][r])))

#the adjacent cell becomes the new cell

d = r

else:

#if no other cells are available it gets another random cell

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

#returns the edge list or visited cells

return visited

def breathFirstSearch(adj):

# start with an list of zeros

visited = np.zeros(len(adj),dtype = int)

# create a queue

queue = []

#the search will start at cell zero

cell = 0

#append the cell to the queue and mark as visted with a one

queue.append(cell)

visited[cell] = 1

while queue:

# dequeue and print it

cell = queue.pop(0)

print (cell, end = " ")

#if the cell is the top right corner it has reached the end of the maze

if cell == len(adj)-1:

break

# get all adjacent the cell and enqueue

#mark as visited

for i in adj[cell]:

if visited[i] == 0:

queue.append(i)

visited[i]=1

def DFSiterative(adj):

#create a list all zeros

visited = np.zeros(len(adj),dtype = int)

#search starts at zero

start = 0

#creates a stack

stack = []

#append to the stack and mark as visited

stack.append(start)

visited[0]=1

#run until stack is empty

while stack:

#pops from stack and prints

cell = stack.pop( )

print(cell,end=' ')

#if the cell is the top right corner it has reached the end of the maze

if cell == len(adj)-1:

break

# get all adjacent the cell and enqueue

#mark as visited

for i in adj[cell]:

if visited[i] == 0:

stack.append(i)

visited[i]=1

def DFSRecursive(start,visited,adj,maze):

#marks the starting cell as visited

visited[start] = 1

#appends to a list to record path instead of printing it

maze.append(start)

#runs and sends recursively for all adjacent walls

for i in adj[start]:

if visited[i]==0:

DFSRecursive(i,visited,adj,maze)

def printdfs(maze,adj):

#gets the path and prints until is reaches the goal which is the top right corner

for i in maze:

print(i,end=' ')

if i == len(adj)-1:

break

plt.close("all")

maze\_rows = 10

maze\_cols = 10

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

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

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

adj = adjacentList(walls,maze\_rows,maze\_cols)

m = maze(S,walls,adj)

adjacentlist=(adjacentList(m,maze\_rows,maze\_cols))

print()

print('adjacency list')

#print(adjacentList(m,maze\_rows,maze\_cols))

print()

print('Breath First Search:')

breathFirstSearch(adjacentlist)

print()

print('Depth First Search:')

print('Iterative:')

DFSiterative(adjacentlist)

visited = np.zeros(len(adjacentlist),dtype = int)

dfs=[]

print()

print('Recursive:')

DFSRecursive(0,visited,adjacentlist,dfs)

printdfs(dfs,adjacentlist)

draw\_maze(walls,maze\_rows,maze\_cols)

I Ericka Najera certify that this project is entirely my own work. I wrote, debugged, and teste 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.

