Joaquin Hidalgo

CS 2302 MW @1:30pm

Lab 07

**BFS vs. DFS vs. DFS (recursion)**

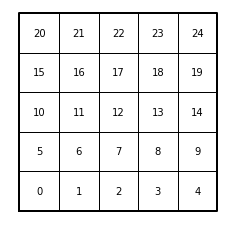
**Introduction:** The problem at hand was to draw a maze and pop walls of two cells chosen at random if and only if they are a part of two different sets. Once wall is popped, then you union by size each cell until you receive a perfect maze and display. Once obtained the finished, perfect maze, now we use 3 different methods to traverse the maze in order to arrive at the top right of the maze from the bottom left side of the maze and compare running times to figure out which is the best one.

**Proposed solution design and implementation:** First I ask the user input ahead of time to declare size of rows and columns of the maze the number of walls needed to be removed. After this, if the number of cells minus 1 equals the user input to remove that many walls, then ask if you want to preform BFS, DFS (with iteration or recursion). At this point, I start the timer to compare running times later. Once I create the perfect maze to traverse, I start with choice 1 which is BFS. BFS is checking every possible pathway from the starting point to every single cell. I assume this will be the fastest way to traverse the whole maze due to the fact you do not commit to one single way rather explore all possible ways using a queue to keep count of the needed pathways to traverse. As I pop off cells which are the walls, there are two numbers to each cell and create a connection between the two cells and append these cells to make an, adjacency list. Then have an array of Booleans all set to false at the beginning to keep track of which pathways/cells have been visited, once visited, make corresponding cell True. Return an array of “prev” so that way we can know where each cell has come from which is then passed to a method called, GetPath which is given the adjacency list and the ending point. This creates a list of a direct path from top-right cell to cell 0 (bottom left).  
For the DFS, I traverse the Maze almost the same way as a BFS but instead of using a queue I use a stack to keep track of each cell that is neighboring the open path way maze. Since I use a stack, here we are committing to one pathway and if this pathway leads us to a null, we then go back, trace our steps and continue to traverse the next possible way to travel which I think will make this program run slower.

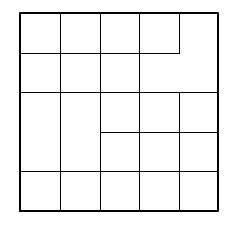
Lastly, I use DFS recursion to traverse the maze, by using global variables that include a Boolean array of visited cells and a ‘prev’ array that keep track where each index of the array tells where the pathway come from. I then pass the prev array after executing the DFS recurrion method and pass adjacency list of the maze to a method called, GetPath which returns the direct path to the top right corner of the maze from the bottom left.

**Experimental results:**

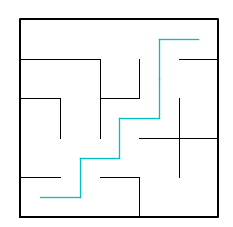
**Maze representation w/ cells**

****

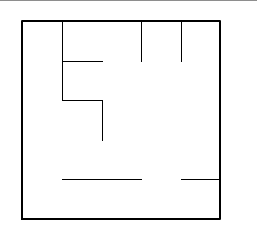
**When removing walls is less than (n-1)**

****

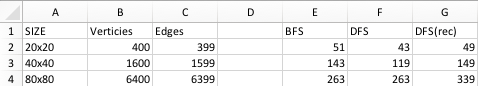
**When removing walls is equal to (n-1) ‘PERFECT MAZE’**

****

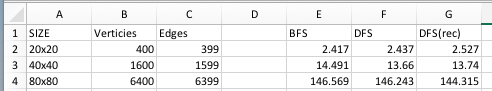
**When removing walls is greater than (n-1) ‘Multiple solution MAZE’**

****

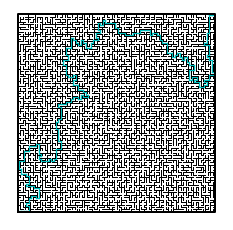
**Length of Path (Solution)**



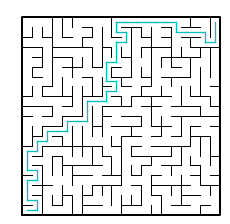
**Running times**

****

**80x80 Maze w/ solution**

****

**20x20 Maze w/ solution**

****

**Conclusion:** I learned how to make and draw a line of a solution which is the path from the bottom left corner of the maze to the top right. Here I used different methods to traverse the maze in order to get an adjacency list of all the possible solutions of the maze. Then I got the directed single path of the solution and represented it in the maze. BFS searches all multiples paths at different depths at a time and DFS commits to one path and it backtracks itself until arrived at the solution. I believe BFS is a better algorithm to traverse a maze but definitely takes up more space. I thought that DFS would be slower and have the recursive method of itself to be even slower than DFS iterative. But once comparing lengths of the path solution and comparing running times of each method, I **did NOT see a difference** when comparing DFS, BFS and DFS (recursion).

**Code:**

﻿"""

By: Joaquin Hidalgo

Lab 07

Created on Sat 20 Apr 4:20:00 2019

Instructor: Olac Fuentes

TA: Dita and Mali

Last modified Fri 26 Apr 14:12:37

Use a disjoint forrest set to create a VxV matrix of rows and columns.

If two cells are apart of different sets, union them and pop walls to create

a perfect maze, a maze with not enough popped walls and multiple solution maze.

Then, preform BFS, DFS, or DFS(recurssion) to find a soltion of the maze

starting at point 0 to the top right cell (n-1) and compare running times

"""

from collections import deque #queue

import matplotlib.pyplot as plt #draw lines

import numpy as np

import random

import time

#Create and set disjoint forest by size

def DisjointSetForest(size):

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

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\_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):

count =0

for i in range(len(S)):

if S[i]<0:

count += 1

return count

# Creates a list with all the walls in the maze

def wall\_list(maze\_rows, maze\_cols):

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

#Get path to solution from (0 to n-1) cell

def GetPath(path, source):

global path\_of\_search

if path[source] != -1:

GetPath(path, path[source])

path\_of\_search.append(source)

#Converts Path to solution to an EDGE LIST to DRAW LINES later...

def Path\_to\_EdgeList(li):

res = []

for i in range(1,len(li)):

res.append([li[i-1],li[i]])

return res

#take in an adjceny list and starting point

def Bfs\_iter(adj, source):

global visted

global prev

# iniztilze queue

Q = deque()

Q.append(source)

visted[source] = True

while(len(Q) != 0 ):

current = Q.popleft()

for i in adj[current]:

if (visted[i] == False):

visted[i] = True

prev[i] = current

Q.append(i)

def Dfs\_rec(adj, source):

global visted

global prev

visted[source] = True

for i in adj[source]:

if(visted[i] == False):

prev[i] = source

Dfs\_rec(adj, i)

def Dfs\_iter(adj, source):

global visted

global prev

#initilize stack

stack = []

stack.append(source)

visted[source] = True

while(len(stack) != 0 ):

current = stack.pop()

for i in adj[current]:

if (visted[i] == False):

visted[i] = True

prev[i] = current

stack.append(i)

def Draw\_maze\_cells(walls,maze\_rows,maze\_cols,cell\_nums=False):

fig, ax = plt.subplots()

# draw lines that are INSIDE the maze

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') # connect points

sx = maze\_cols

sy = maze\_rows

#draw the OUTER edge of Maze

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 Draw\_maze\_and\_path(path2,walls2,maze\_rows,maze\_cols):

fig, ax = plt.subplots()

# draw PATH

for w in path2:

if w[1]-w[0] ==1: #new horzintal

x0 = (w[0]%maze\_cols)+.5#good

y0 = (w[0]//maze\_cols)+.5#

x1 = (w[1]%maze\_cols)+.5#good

y1 = (w[1]//maze\_cols)+.5#

else:#new verticallll

x0 = (w[0]%maze\_cols)+.5#good

y0 = (w[0]//maze\_cols)+.5#

x1 = (w[1]%maze\_cols)+.5#good

y1 = (w[1]//maze\_cols)+.5#

ax.plot([x0,x1],[y0,y1],linewidth=1.3,color='c')

sx = maze\_cols

sy = maze\_rows

#draw INSIDE maze walls

for w in walls2:

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')

#Draw OUTER edge of maze

ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')

ax.axis('off')

ax.set\_aspect(1.0)

def draw\_solution\_maze\_only(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)

# Main

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

if \_\_name\_\_ == '\_\_main\_\_':

# USER-INPUT CHOICES

m\_rows= int(input('enter number of rows: '))

m\_cols = int(input('enter number of columns: '))

n = m\_rows\*m\_cols

print('\nThe number of cells: ',n)

m = int(input('Enter the number of walls to remove: '))

if (m == (n-1)):

alt\_choice =int(input('1. BFS 2. DFS 3. DFS(recursive): '))

startTime = time.time()

plt.close("all")

#Allocate space for variables

all\_path\_adj\_list = [ [] for i in range(n) ] #Adjency list of all possible paths

visted = np.zeros(n,dtype=bool) #visited nodes in graph

prev = np.zeros(n,dtype=np.int)-1 #path taken of chosen method

counter = 0 #keeps track of how many walls are Removed

S = DisjointSetForest(n) #Dsf keeps track of which cells are in the same set

path\_of\_search = []

#Case for PERFECT MAZE

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

if m == n-1:

print('\nThe is a unique path from source to destination (when m = n − 1)')

#Creates LIST that is all the walls that CAN BE popped

deafult\_wallList = wall\_list(m\_rows,m\_cols)

#Draw default maze with cells and numbers

Draw\_maze\_cells(deafult\_wallList,m\_rows,m\_cols,cell\_nums=True)

#Keep popping walls off 'deafult\_wallList' untill desired 'm' == counter

while True:

d = random.randint(0,len(deafult\_wallList)-1) # pick random wall

a,b = deafult\_wallList[d] #obtain the num of the two cells

#Find be compression

root1 = find\_c(S,a)

root2 = find\_c(S,b)

#Union by Size if two cells are in different Sets

if root1 != root2:

union\_by\_size(S,a,b)

#Popping wall creates a pathway/space in maze to travel

connection = deafult\_wallList.pop(d)

all\_path\_adj\_list[connection[0]].append(connection[1]) #create adj list of WHOLE PATH

all\_path\_adj\_list[connection[1]].append(connection[0]) #create adj list of WHOLE PATH

counter += 1 # we popped a wall

#Reached desired amount of walls popped

if counter == m:

break

remaining\_walls = deafult\_wallList #Store all remaining walls

print('\nAdjencency representation of maze:')

print(all\_path\_adj\_list)

#Bread First Search

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

if alt\_choice == 1:

print('\nBread First Search!!!\n')

Bfs\_iter(all\_path\_adj\_list, 0) #Pass all possible pathways and starting point to start Search

#Prev is now... BFS array... After method 'Bfs\_iter'

GetPath(prev, n-1)

#path\_of\_search is now the adj list of to the Maze solution

print('Path from cell 0 to ', n-1)

print(path\_of\_search)

edge\_list\_path = Path\_to\_EdgeList(path\_of\_search) #Connection of lines of the pathway DRAWN SOLUTION

#Draw finished maze w/ Path

Draw\_maze\_and\_path(edge\_list\_path,remaining\_walls,m\_rows,m\_cols)

endTime = time.time()

print('\nRunning time: ', endTime - startTime)

print('\nlength of path: ', len(path\_of\_search))

#Depth First Search

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

elif alt\_choice == 2:

print('\nDepth First Search!!!\n')

Dfs\_iter(all\_path\_adj\_list, 0) #Pass all possible pathways and starting point to start Search

#Prev is now... DFS array... After method 'Dfs\_iter'

GetPath(prev, n-1)

#path\_of\_search is now the adj list of to the Maze solution

print('Path from cell 0 to ', n-1)

print(path\_of\_search)

edge\_list\_path = Path\_to\_EdgeList(path\_of\_search) #Connection of lines of the pathway DRAWN SOLUTION

#Draw finished maze w/ Path

Draw\_maze\_and\_path(edge\_list\_path,remaining\_walls,m\_rows,m\_cols)

endTime = time.time()

print('\nRunning time: ', endTime - startTime)

print('\nlength of path: ', len(path\_of\_search))

#Depth First Search (Recurssion)

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

elif alt\_choice ==3:

print('\nDepth First Search (Recurssion)!!!\n')

Dfs\_rec(all\_path\_adj\_list, 0) #Pass all possible pathways and starting point to start Search

#Prev is now... DFS array... After method 'Dfs\_rec'

GetPath(prev, n-1)

#path\_of\_search is now the adj list of to the Maze solution

print('Path from cell 0 to ', n-1)

print(path\_of\_search)

edge\_list\_path = Path\_to\_EdgeList(path\_of\_search) #Connection of lines of the pathway DRAWN SOLUTION

#Draw finished maze w/ Path

Draw\_maze\_and\_path(edge\_list\_path,remaining\_walls,m\_rows,m\_cols)

endTime = time.time()

print('\nRunning time: ', endTime - startTime)

print('\nlength of path: ', len(path\_of\_search))

else:

print('Invalid input, please pay attention!!')

#Case for NOT ENOUGH walls poped from maze

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

elif m< n-1:

print('\nA path from source to destination is not guaranteed to exist (when m < n − 1)')

#Creates LIST that is all the walls that CAN BE popped

deafult\_wallList = wall\_list(m\_rows,m\_cols)

#Draw default maze with cells and numbers

Draw\_maze\_cells(deafult\_wallList,m\_rows,m\_cols,cell\_nums=True)

#Keep popping walls off 'deafult\_wallList' untill desired 'm' == counter

while True:

d = random.randint(0,len(deafult\_wallList)-1) # pick random wall

a,b = deafult\_wallList[d] #obtain the num of the two cells

#Find be compression

root1 = find\_c(S,a)

root2 = find\_c(S,b)

#Union by Size if two cells are in different Sets

if root1 != root2:

union\_by\_size(S,a,b)

deafult\_wallList.pop(d)

counter += 1

if counter == m:

break

remaining\_walls = deafult\_wallList #Store all remaining walls

#Draw finished maze

draw\_solution\_maze\_only(remaining\_walls,m\_rows,m\_cols)

endTime = time.time()

print('\nRunning time: ', endTime - startTime)

#Case for MULTIPLE PATH MAZE

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

else:

print('\nThere is at least one path from source to destination (when m > n − 1)')

#Creates LIST that is all the walls that CAN BE popped

deafult\_wallList = wall\_list(m\_rows,m\_cols)

#Draw default maze with cells and numbers

Draw\_maze\_cells(deafult\_wallList,m\_rows,m\_cols,cell\_nums=True)

#Keep popping walls off 'deafult\_wallList' untill desired 'm' == counter

while True:

d = random.randint(0,len(deafult\_wallList)-1) # pick random wall

a,b = deafult\_wallList[d] #obtain the num of the two cells

#Find be compression

root1 = find\_c(S,a)

root2 = find\_c(S,b)

#Union by Size if two cells are in different Sets

if root1 != root2:

union\_by\_size(S,a,b)

deafult\_wallList.pop(d)

counter += 1

if counter == n-1:

break

number\_of\_walls\_to\_pop = m - (n-1)

for i in range(number\_of\_walls\_to\_pop):

d = random.randint(0,len(deafult\_wallList)-1)

deafult\_wallList.pop(d)

remaining\_walls = deafult\_wallList #Store all remaining walls to display

#Draw finished maze

draw\_solution\_maze\_only(remaining\_walls,m\_rows,m\_cols)

endTime = time.time()

print('\nRunning time: ', endTime - startTime)

**Standards of Conduct and Academic Dishonesty**

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.

* Joaquin Hidalgo