**Introduction**

**Breadth First Search function:**

Since we were provided with pseudocode from the in-class exercise, I just thought of following the same code structure.

Therefore, I created the algorithm starting off with “visited” and “previous” as Boolean and integer arrays respectively. These allow to keep track of the information from each vertex visit. Then, creating a Queue and appending the source vertex was necessary to start searching the graph. A while loop is used to iterate through the graph while the queue is not empty and simultaneously pops the first in the list. A for loop is then used to traverse though other vertices that are connected to that current vertex, while changing the state of visit for each vertex, adding info to “prev” array and adding the nodes to the queue. Lastly it returns the “prev” array.

Running time and time complexity:

Experiment: Created maze with low number of removed walls (no solution for maze)

Result: No solution displayed in graph

Experiment: Changed size of the solvable maze from 32 cells to 225, 900 and to finally 3600 cells

Result: Time increased according to O complexity

*Big O complexity: O(|v|+|E|)*

**Depth First Search function (recursive):**

This algorithm is the smallest out of the other search algorithms. Its quite simple since it only has loop which calls the method itself recursively.

The function was created by setting the first visited graph node to true. Then, to go through the graph, a for loop is used to traverse all the vertices connected to the source vertex that have not been visited. Plus, sets the “prev” array’s vertex to the parent and recurses for every new vertex.

Running time and time complexity:

Experiment: Created maze with low number of removed walls (no solution for maze)

Result: No solution displayed in graph

Experiment: Changed size of the solvable maze from 32 cells to 225, 900 and to finally 3600 cells

Result: Time increased according to O complexity

*Big O complexity: O(|E|)*

**Depth First Search function (non-recursive):**

This algorithm is almost identical to the Breadth First Search except for the use of a different Abstract Data Type.

Similarly, to build the iterative version of this function “visited” ad “prev” lists are made to store Boolean and integer data respectively. Afterwards, a stack is initialized empty and the source is appended while setting the first node to visited. Then, a while loop is used to traverse through the graph if the stack is not empty. For every iteration, the stack is popped and used in a for loop to traverse every connected vertex to the source. In the for loop, every vertex is set as visited, the “prev” information is set to the parent index for every node connected to it and adds the new graph node to the stack. Lastly it returns the prev array.

Running time and time complexity:

Experiment: Created maze with low number of removed walls (no solution for maze)

Result: No solution displayed in graph

Experiment: Changed size of the solvable maze from 32 cells to 225, 900 and to finally 3600 cells

Result: Time increased according to O complexity

*Big O complexity: O(|E|)*

*Outputs:*

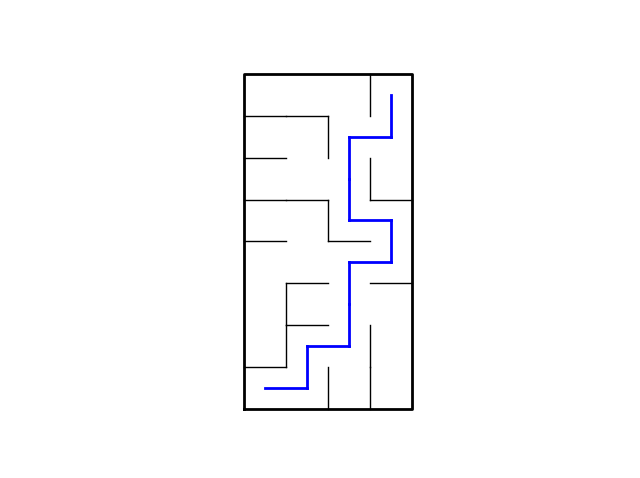


Figure 1: 8 by 4 maze path solution

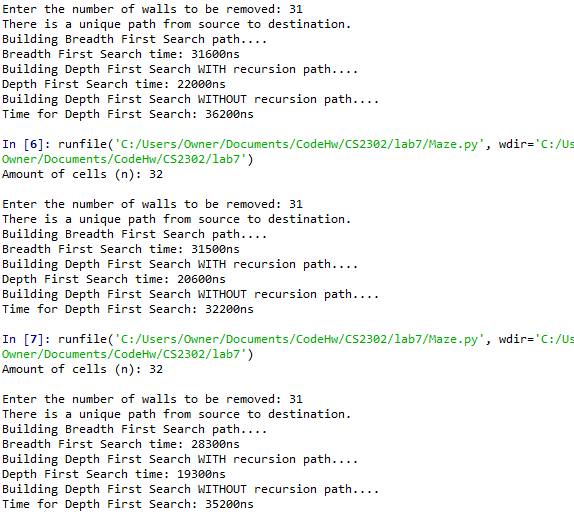


Figure 2: 8 by 4 search output

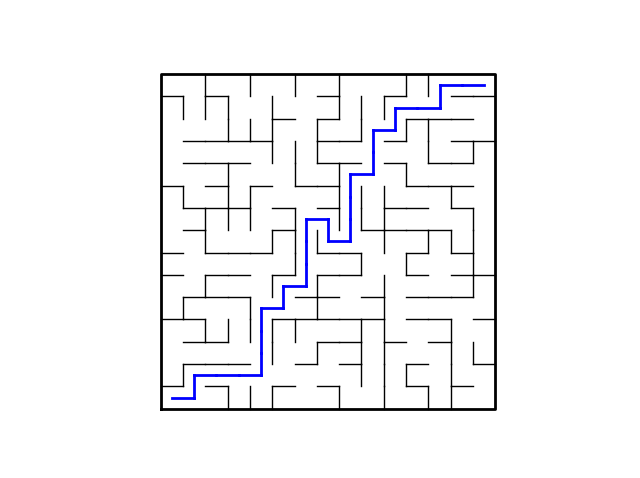


Figure 3: 15 by 15 maze path solution

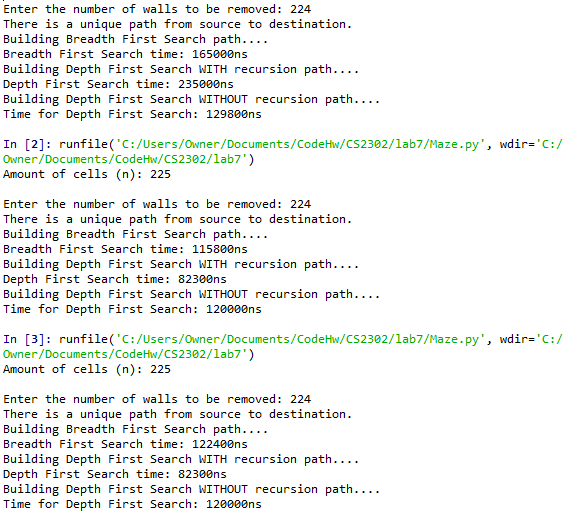


Figure 4: 15 by 15 search output

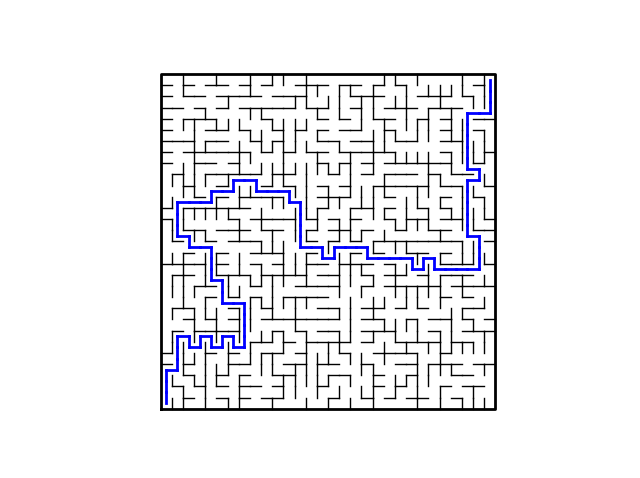


Figure 5: 30 by 30 maze path solution

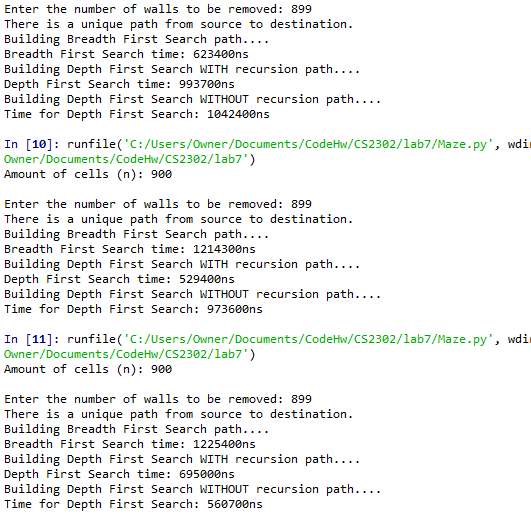


Figure 6: 30 by 30 search output

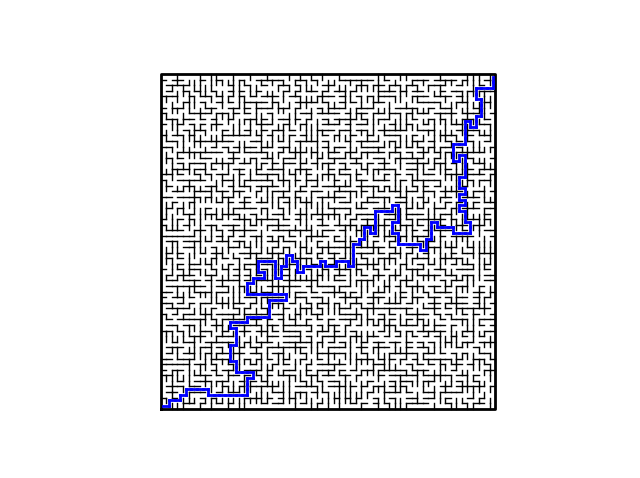


Figure 7: 60 by 60 maze path solution

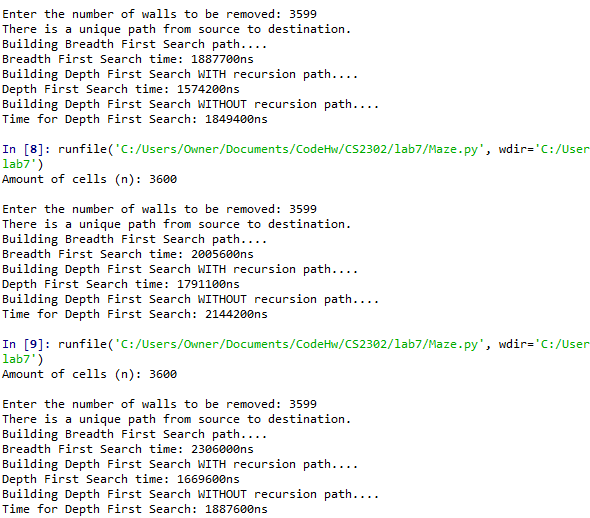


Figure 8: 60 by 60 search output

**Appendix**

*# Course: 2302-001*

*# Author: Esteban Retana*

*# Assignment: Solve the maze from lab 6(building maze with disjoint set forest)*

*# Instructor: Olac Fuentes*

*# TA: Mali and Dita*

*# Date of last modification:4/28/19*

*# Purpose: To be able to implement a maze solving algorithm such as Breadth First Search or Depth First Search and display its path*

*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 dsfToSetList(*S*):

*#Returns a 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*

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 draw\_maze(*walls*,*maze\_rows*,*maze\_cols*,*maze\_path*,*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")

*# Added maze path, displays line when its not empty*

*if* len(maze\_path) != 0:

*# Check each connection and draw path*

*for* i in range(len(maze\_path)-1):

*if* maze\_path[i+1]-maze\_path[i] == maze\_cols:

x0 = (maze\_path[i]%maze\_cols)+.5

x1 = x0

y0 = (maze\_path[i+1]//maze\_cols)-.5

y1 = y0+1

*elif* maze\_path[i+1]-maze\_path[i] == -maze\_cols:

x0 = (maze\_path[i+1]%maze\_cols)+.5

x1 = x0

y0 = (maze\_path[i]//maze\_cols)-.5

y1 = y0+1

*elif* maze\_path[i+1]-maze\_path[i] == -1:

x0 = (maze\_path[i+1]%maze\_cols)+.5

x1 = x0+1

y0 = (maze\_path[i]//maze\_cols)+.5

y1 = y0

*else*:

x0 = (maze\_path[i]%maze\_cols)+.5

x1 = x0+1

y0 = (maze\_path[i+1]//maze\_cols)+.5

y1 = y0

ax.plot([x0,x1],[y0,y1],*linewidth*=2,*color*='blue')

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 NumSets(*S*):

count =0

*for* i in range(len(S)):

*if* S[i]<0:

count += 1

*return* count

*#finds the path to a vertex v and appends it to a list*

def find\_path(*prev*,*v*,*a*):

*if* prev[v] != -1:

find\_path(prev,prev[v],a)

a.append(v)

*return* a

*# Builds maze by removing walls with compression*

def RemoveWalls\_c(*S*,*Adj\_list*,*m*,*walls*):

r = []

w = len(walls)

numCells = NumSets(S)

Message(m,numCells)

*for* i in range(m):

*# Chooses random wall to remove*

r = random.choice(walls)

f = walls.index(r)

*# Starts to remove random walls*

*if* i < numCells-1:

*while* True:

*if* find(S,r[0]) != find(S,r[1]):

*# Adjencency list build*

build\_adj\_list(Adj\_list,r[0],r[1])

*# Eliminates random walls*

walls.pop(f)

union\_by\_size(S,r[0],r[1])

*break*

*else*:

r = random.choice(walls)

f = walls.index(r)

*else*:

*# Adjencency list build*

build\_adj\_list(Adj\_list,r[0],r[1])

*# Eliminates random walls*

walls.pop(f)

union\_by\_size(S,r[0],r[1])

*if* i == w-1:

print("Reached maximum number of walls to remove.")

*return*

*#Function to display messages based on m*

def Message(*m*,*walls*):

*if* m < walls-1:

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

*elif* m == walls-1:

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

*else*:

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

*# Creates adjencency list*

def build\_adj\_list(*Adj\_list*,*s1*,*s2*):

Adj\_list[s1].append(s2)

Adj\_list[s2].append(s1)

*# Searches for solution using Breadth First Search*

def BFS(*G*,*v*):

visited = [False] \* len(G)

prev = [-1] \* len(G)

Q = []

Q.append(v)

visited[v] = True

*# Does loop while the Q is not empty*

*while* len(Q) != 0:

*# Obtains first value*

u = Q.pop(0)

*# Visits every vertex from index obtained*

*for* t in G[u]:

*if* visited[t] == False:

visited[t] = True

prev[t] = u

Q.append(t)

*return* prev

*# Searches for solution using Depth First Search*

def DFS(*G*,*source*):

visited[source] = True

*# Visits every vertex from source*

*for* t in G[source]:

*if* not visited[t]:

prev[t] = source

DFS(G,t)

*# Searches for solution using Depth First Search*

def DFS\_iter(*G*,*source*):

visited = [False] \* len(G)

prev = [-1] \* len(G)

Q = []

Q.append(source)

visited[source] = True

*# Does loop while the Q is not empty*

*while* len(Q) != 0:

*# Obtains first value*

u = Q.pop(-1)

*# Visits every vertex from index obtained*

*for* t in G[u]:

*if* visited[t] == False:

visited[t] = True

prev[t] = u

Q.append(t)

*return* prev

plt.close("all")

maze\_rows = 15

maze\_cols = 15

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

*# Displays original maze with cell numbers*

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

*# Creates dfs with dimensions*

n = maze\_rows \* maze\_cols

S = DisjointSetForest(n)

print("Amount of cells (n):",n)

m = int(input("Enter the number of walls to be removed: "))

*# Adjancency list build*

Adj\_list = [[] *for* i in range(n)]

RemoveWalls\_c(S,Adj\_list,m,walls)

*#Breadth first search*

print("Building Breadth First Search path....")

start = time.perf\_counter\_ns()

path = BFS(Adj\_list,0)

end = time.perf\_counter\_ns()

print("Breadth First Search time:",end-start,*end*="ns\n")

maze\_path = find\_path(path,len(Adj\_list)-1,[])

draw\_maze(walls, maze\_rows, maze\_cols,maze\_path,*cell\_nums*=False)

*#Depth First Search*

visited = [False] \* len(Adj\_list)

prev = [-1] \* len(Adj\_list)

print("Building Depth First Search WITH recursion path....")

start = time.perf\_counter\_ns()

DFS(Adj\_list,0)

end = time.perf\_counter\_ns()

print("Depth First Search time:",end-start,*end*="ns\n")

maze\_path = find\_path(prev,len(Adj\_list)-1,[])

draw\_maze(walls, maze\_rows, maze\_cols,maze\_path,*cell\_nums*=False)

*#Depth First Search iterably*

print("Building Depth First Search WITHOUT recursion path....")

start = time.perf\_counter\_ns()

path = BFS(Adj\_list,0)

end = time.perf\_counter\_ns()

print("Time for Depth First Search:",end-start,*end*="ns\n")

maze\_path = find\_path(path,len(Adj\_list)-1,[])

draw\_maze(walls, maze\_rows, maze\_cols,maze\_path,*cell\_nums*=False)

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.