Erick Perchez

80582912

CS 2302 Data Structures

5/8/2019

Lab 7

**Introduction:** In this lab we were to use our maze (lab6) to create a random maze and remove however many walls the user indicated. Then we are to draw the solution from start to end.

**Proposed solution:**

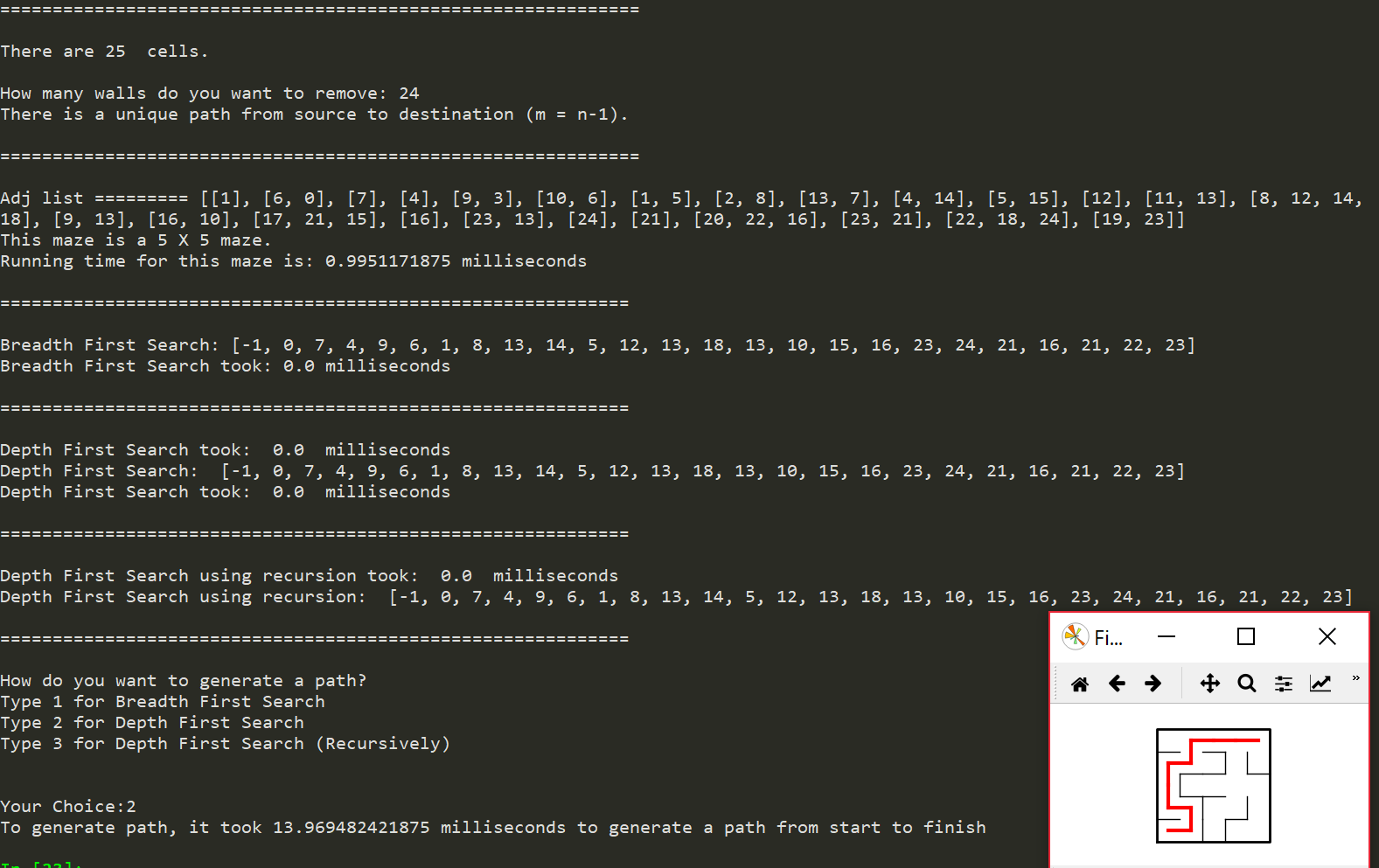
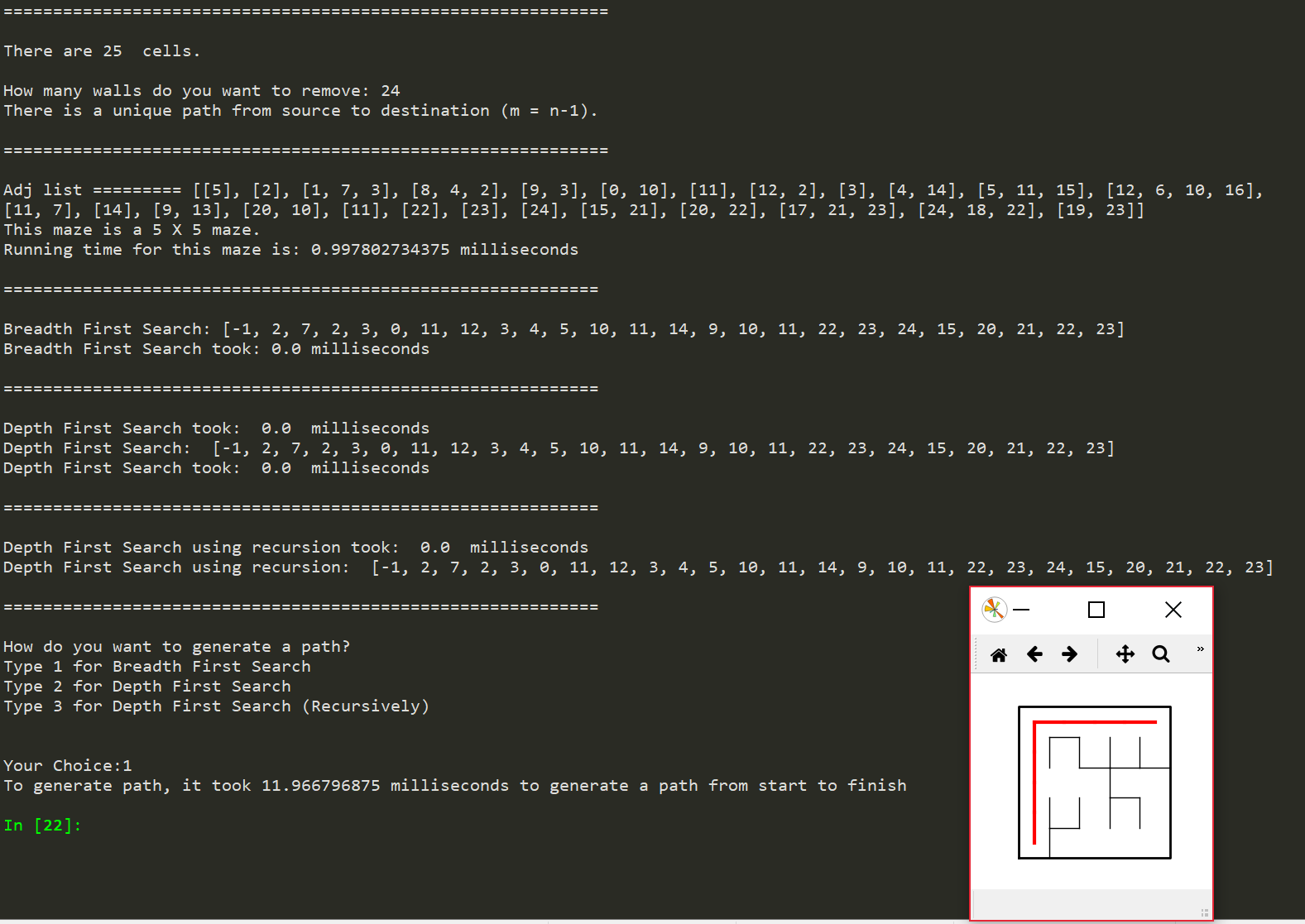
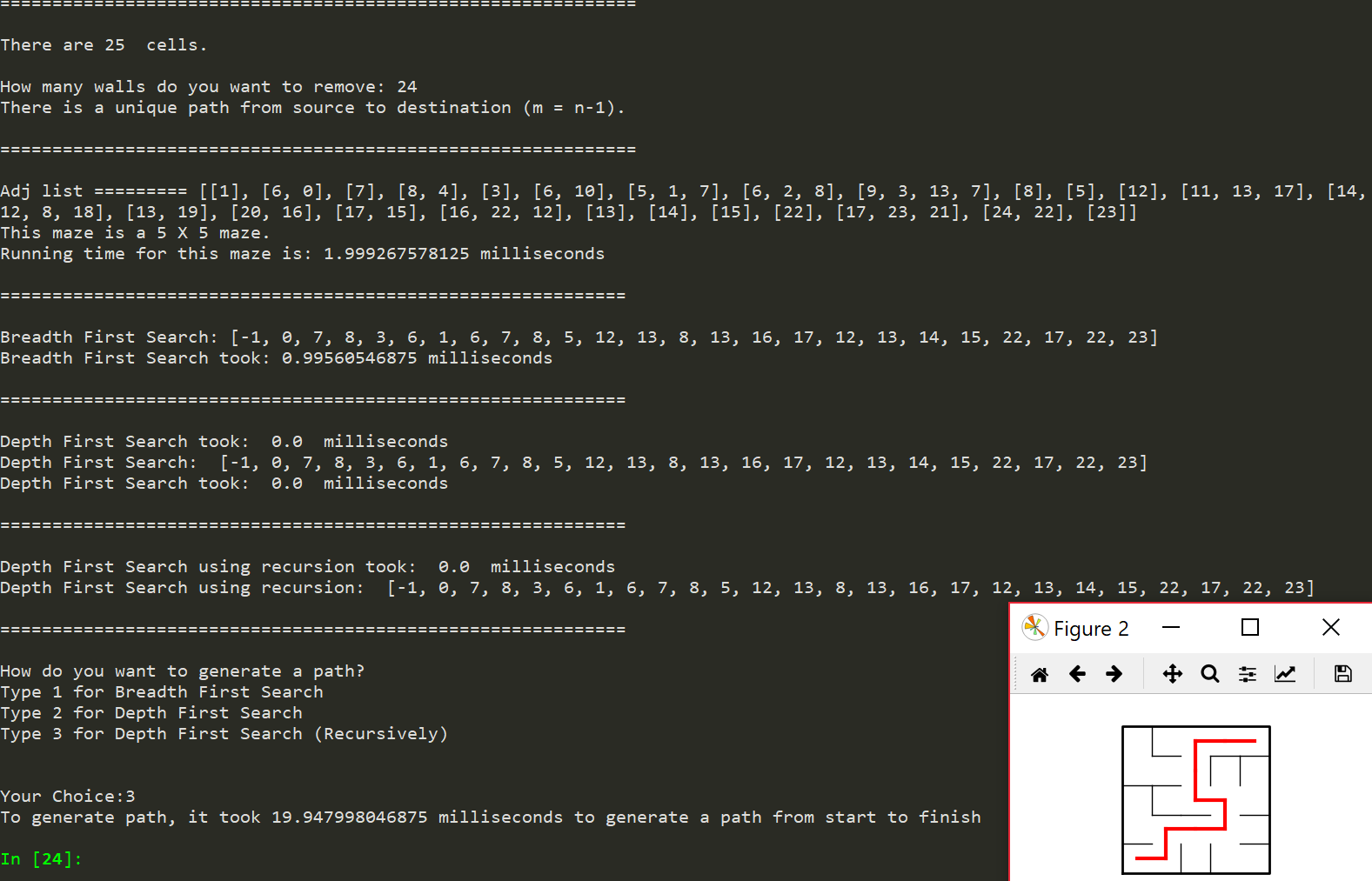
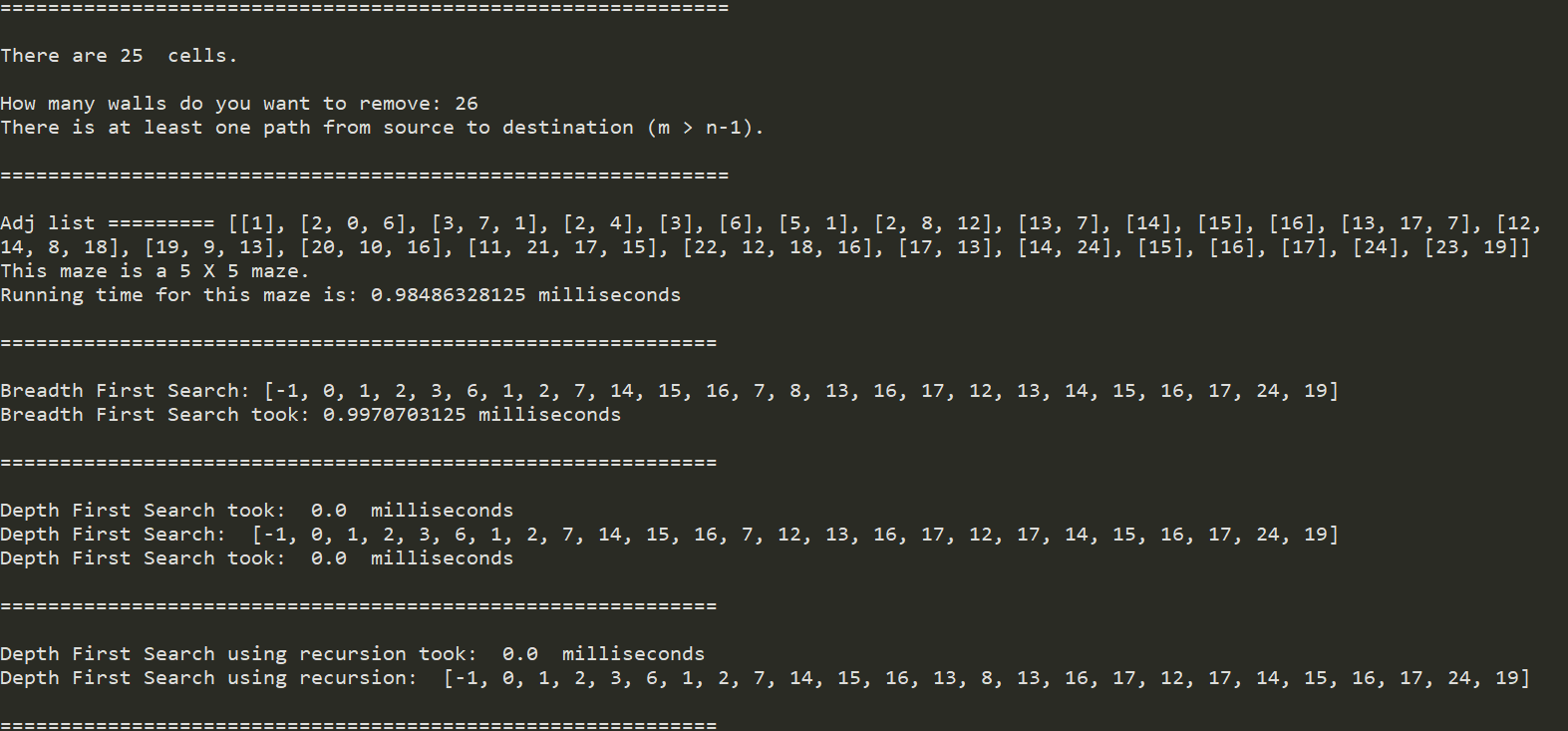
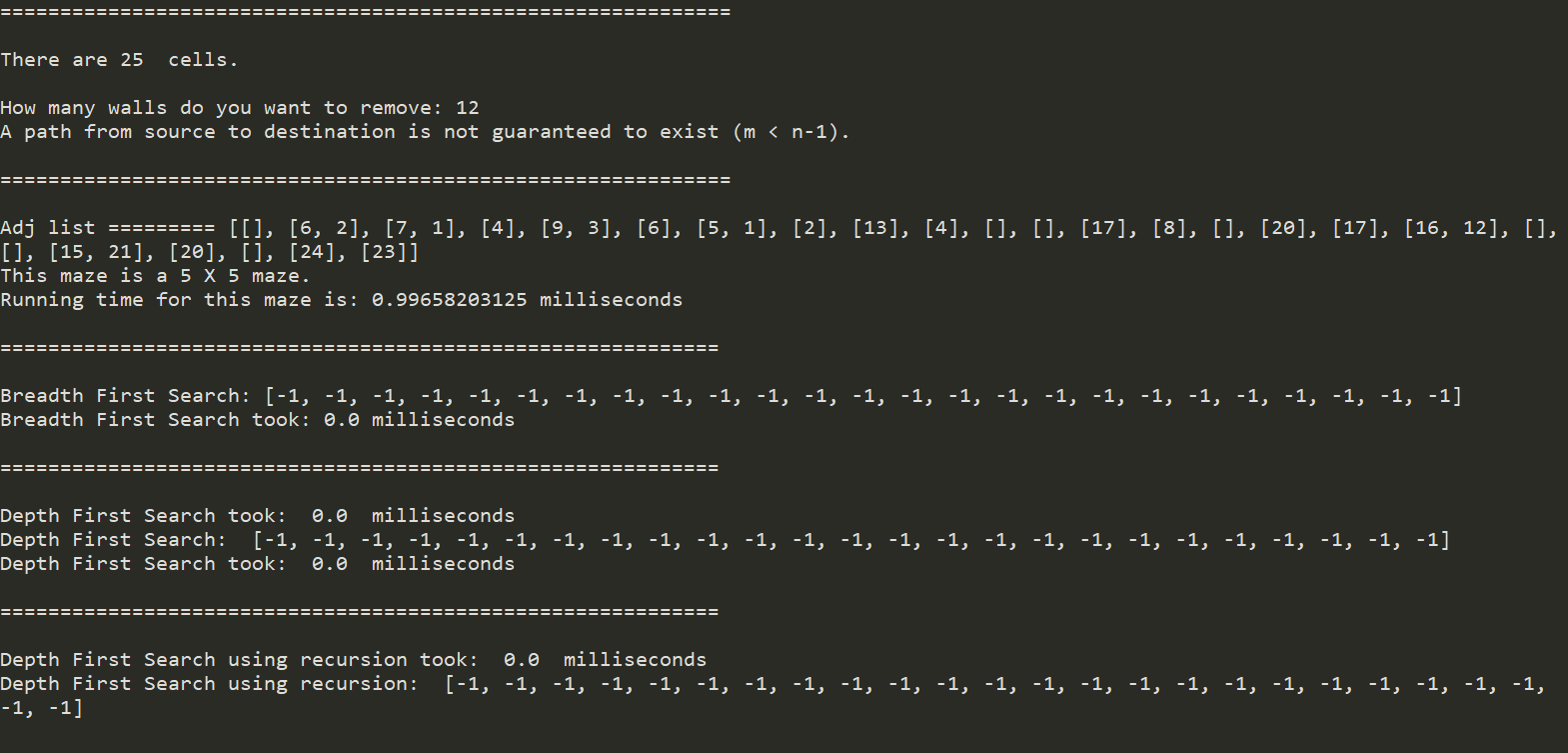
1. Display n, number of cells, and ask for m and display a message: To do this, I tell the user the number of cells, which is the columns times the rows. Then I ask how many walls they want to remove. I take the input and make three scenarios. If m (user input)< n-1, I print “A path from source to destination is not guaranteed to exist.” If m = n-1, I print “The is a unique path from source to destination.” If m > n-1, I print “There is at least one path from source to destination.” Running Time: O(1)
2. Write a method to build adjacency list representation for your maze: To do this, I create a temporary list in my method, “Union\_Maze” that creates a maze and removes walls depending on the number of walls the user wanted to remove, and use the list called popped(that is the list that contains the walls removed) and a new 2D list and send it to a method called get\_adj\_list. This method has a for loop that creates two temporary variables that store the elements, from the popped wall list, popped[i][0] and popped[j][0]. I then append them to the empty 2D list and return that list.
3. a) Implement Breadth-first search: To implement, I use the pseudo code were given and I convert it to python. I first create a list of Boolean False values of the length of the graph given. I then create a list of -1’s length of the graph. Then, from the imported queue function, I create a new queue. I then input the 0 which is the starting point, then I change visited[0] to True, as it was visited. I then created a while loop with condition, “while Q.empty() is False:” In the while loop I pop a value and save it to a variable. Then I create a for loop using “ for t in G[v], v being the popped element. I then check if t has been visited, if it hasn’t, I then change visited[t] to True, then I change prev[t] to be v as it was the cell before it. I then input t into the queue, ending the two loops. I then return the list, previous.

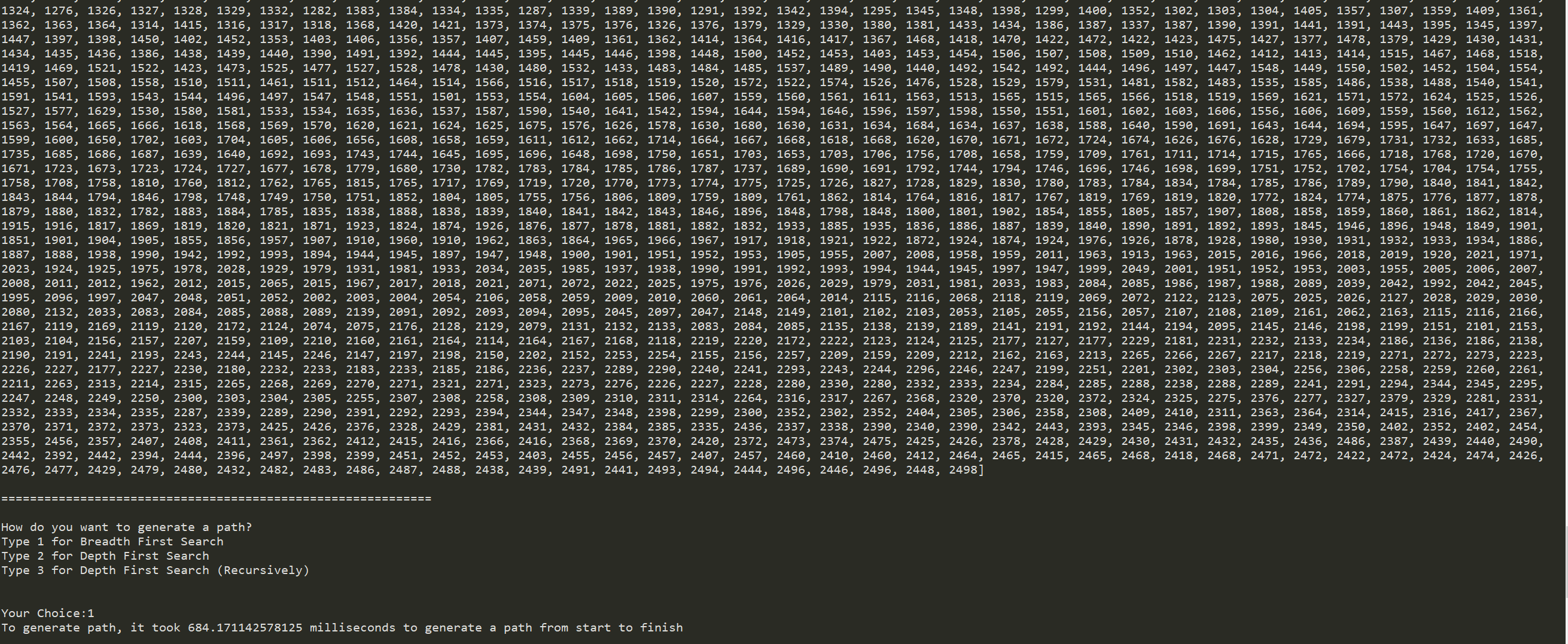
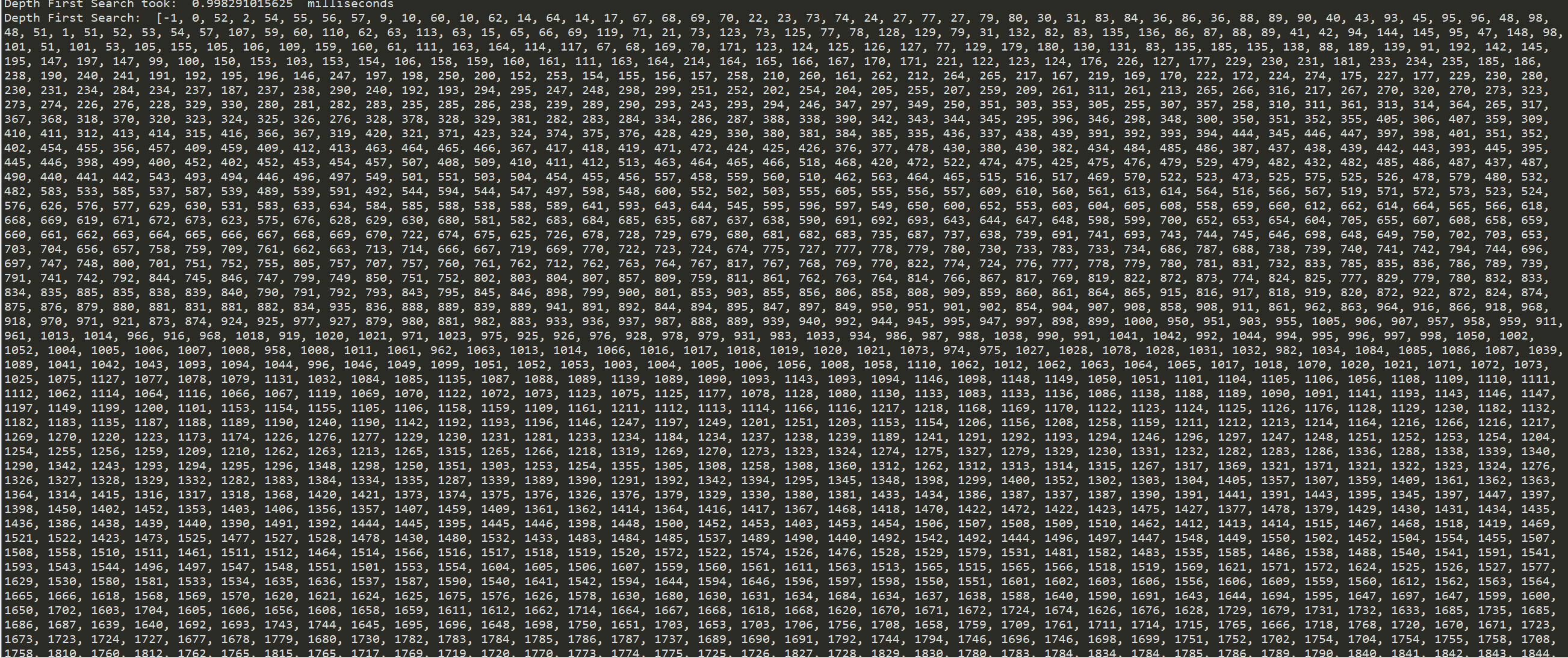
b) Depth-first search: TO start, I create a list of Boolean False length of the given graph. Then another list of -1’s of length of the graph. I then make a new list to represent a stack. I then append 0 to it as it’s the start, then change the visited value for 0 to True. I create a while loop with condition that says while s is not empty. I then pop a value and save it as v. then I create a for loop,” for t in G[v] and then check if its been visited. If not, I change the visited value to True, prev[t] is the popped value, then I append t to the stack.

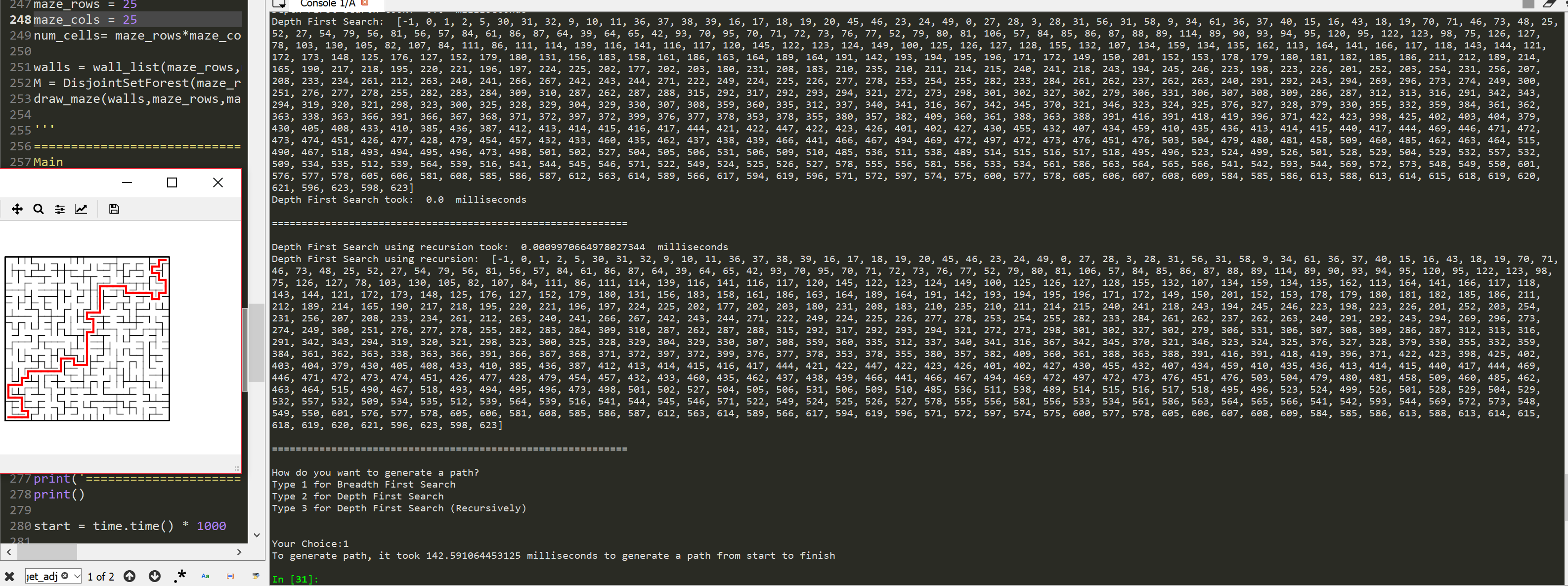
c) Recursive Depth-first search: To do this, the method receives two variables, G, graph, and source, being the source of where it comes from. I start by changing visited[source] to true as it has been visited, I then create a for loop that states “ for t in G[source]” that checks if that vertex/ cell has been visited, if it hasn’t, prev[t] becomes source, and the recursion call sends the values of G, and t to the recursive call.

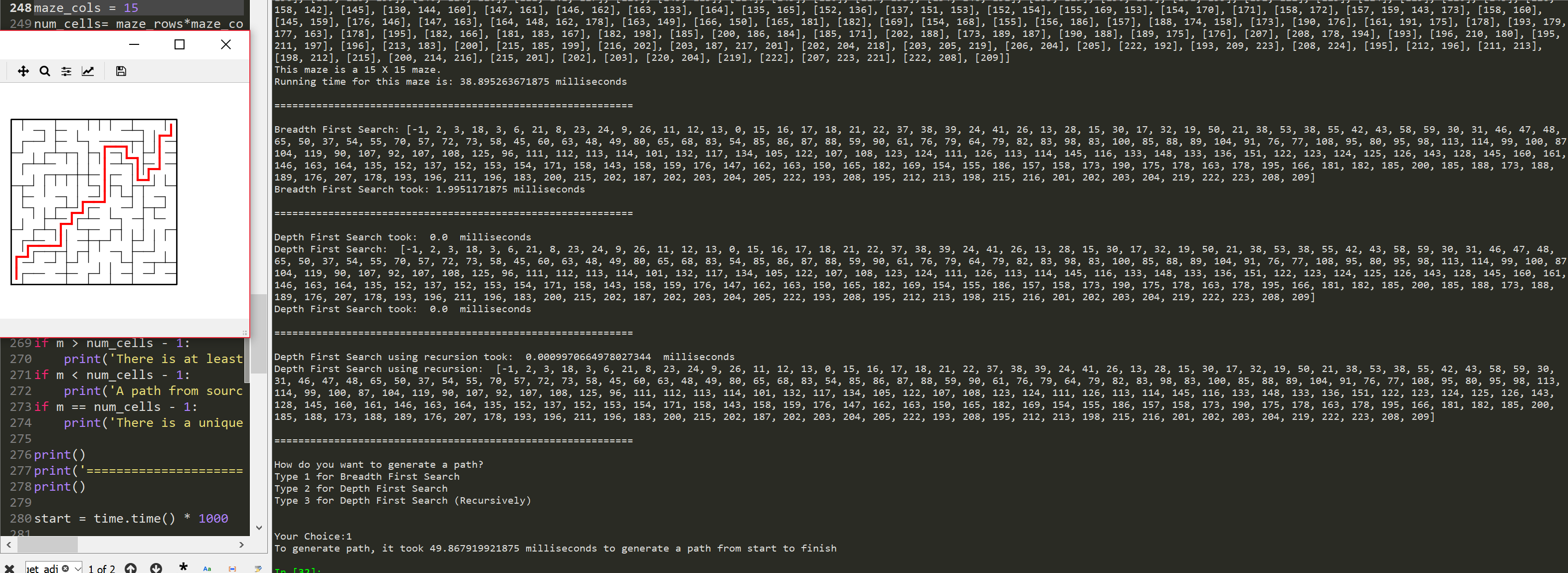
I then return the list prev that contains the previous cells.

­­­

**Experimental Results:** 

Now, different sized mazes having a unique path.





**Conclusion:** I learned a lot of different ways to involve different algorithms in my code. It was tough but it seemed to work

**Appendix:**

'''

Course: CS2302

Author: Erick Perchez

Assignment: Lab 7

Instructor: Dr. Fuentes

TA: Andita Nath

Date: 05/06/2019

Purpose: To modify a maze and show a path using differrent algorithms

'''

import matplotlib.pyplot as plt

import numpy as np

import time

import random

import queue

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

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

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

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj:

if S[ri]>S[rj]:

S[rj] += S[ri]

S[ri] = rj

else:

S[ri] += S[rj]

S[rj] = ri

def NumSets(S):

count =0

for i in S:

if i < 0:

count += 1

return count

'''

=============================================================================

'''

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

# Draws a maze

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)

return ax

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

# Creates the adj list fora graph

def get\_adj\_list(p, A):

for i in range(len(p)):

temp0 = p[i][0]

temp1 = p[i][1]

A[temp0].append(temp1)

A[temp1].append(temp0)

return A

# Check's for a path recursively backwards from end to start by checking

#a path exists

def path(plot, prev, vertex, x, y):

# Only element at previous[0] should equal -1 if a path exists

# If a -1 is found before it, it means no path exists.

if prev[vertex] != -1:

# path is ploted in red from removed wall

if vertex == (prev[vertex] + maze\_cols):

x1 = x

y1 = y - 1

path(plot, prev, prev[vertex], x1, y1)

plot.plot([x1, x], [y1, y], linewidth = 3, color = 'r')

if vertex = (prev[vertex] - maze\_cols):

x1 = x

y1 = y + 1

path(plot, prev, prev[vertex], x1, y1)

plot.plot([x1, x], [y1, y], linewidth = 3, color = 'r')

if vertex == (prev[vertex] + 1):

x1 = x - 1

y1 = y

path(plot, prev, prev[vertex], x1, y1)

plot.plot([x1, x], [y1, y], linewidth = 3, color = 'r')

if vertex == (prev[vertex] - 1):

x1 = x + 1

y1 = y

path(plot, prev, prev[vertex], x1, y1)

plot.plot([x1, x],[y1, y], linewidth = 3, color = 'r')

#Traverses adjacency list using depth-first search

def depth\_first\_search(G):

start = time.time()\*1000

#creates a list of boolean

visited = [False] \* len(G)

#creates a list of length G

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

#a list is used as a stack

S = []

#appends 0 as its the start then changes to True

S.append(0)

visited[0] = True

#Traversal starts from 0

while S: #not empty

#vertex popped from S

v = S.pop()

for t in G[v]:

#checks if its visited

if not visited[t]:

visited[t] = True

prev[t] = v

S.append(t)

stop = time.time()\*1000

print('Depth First Search took: ', stop-start, ' milliseconds')

return prev

# Traverses(w/ recursion) adjacency list using depth-first search

def depth\_first\_search\_recursion(G, source):

#source starts at 0

visited[source] = True

for t in G[source]:

#checks if visited

if not visited[t]:

#then source is appended to prev list

prev[t] = source

depth\_first\_search\_recursion(G, t)

return prev

# Travereses adjacency list with breadth-first search

def breadth\_first\_search(G):

visited = [False] \* len(G)

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

#creates queue from function

Q = queue.Queue()

#0 is the start, so its put first and changesd to True on visited

Q.put(0)

visited[0] = True

while Q.empty() is False:

#vertex gets popped then saved to v

v = Q.get()

for t in G[v]:

#checks if visited then added to prev list

if not visited[t]:

visited[t] = True

prev[t] = v

Q.put(t)

return prev

# creates a maze and removes walls depending on users input

def Union\_Maze(M, w, m):

popped = []

while m > 0:

#D is the wall that gets removed at random

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

if NumSets(M) == 1:

popped.append(walls.pop(d))

m -= 1

elif union\_c(M, walls[d][0], walls[d][1]) is True:

popped.append(walls.pop(d))

m -=1

temp\_list = []

#creates a temporary list

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

temp\_list.append([])

adj\_list = get\_adj\_list(popped, temp\_list)

print('Adj list =========',adj\_list)

return adj\_list

plt.close("all")

maze\_rows = 15

maze\_cols = 15

num\_cells= maze\_rows\*maze\_cols

walls = wall\_list(maze\_rows,maze\_cols) # Creates the walls for the maze

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

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

'''

===============================================================================

Main

'''

print('=============================================================')

print()

print('There are', maze\_rows\*maze\_cols,' cells.')

m = int(input('How many walls do you want to remove: '))

while m < 0 or m > 3000000: #max number of walls in a 15X20

m = int(input('Invalid input, try a different number:'))

if m > len(walls) - 1:

print('The number of walls you want to remove exeeds the number of walls that are present.')

if m > num\_cells - 1:

print('There is at least one path from source to destination (m > n-1).')

if m < num\_cells - 1:

print('A path from source to destination is not guaranteed to exist (m < n-1).')

if m == num\_cells - 1:

print('There is a unique path from source to destination (m = n-1).')

print()

print('=============================================================')

print()

start = time.time() \* 1000

G = Union\_Maze(M, walls, m) # Creates maze

end = time.time() \* 1000

print('This maze is a', maze\_rows, 'X', maze\_cols, 'maze.')

print('Running time for this maze is:', (end-start), 'milliseconds')

print()

print('============================================================')

print()

#variable created to be used in depth\_first\_search\_recursion

visited = [False] \* len(G)

#variable created to be used in depth\_first\_search\_recursion

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

start = time.time() \* 1000

bfs = breadth\_first\_search(G)

end = time.time() \* 1000

print('Breadth First Search:', bfs)

print('Breadth First Search took:', end-start, 'milliseconds')

print()

print('============================================================')

print()

start = time.time() \* 1000

dfs = depth\_first\_search(G)

end = time.time() \* 1000

print('Depth First Search: ', dfs)

print('Depth First Search took: ', end-start, ' milliseconds')

print()

print('============================================================')

print()

start = time.time()

dfsr = depth\_first\_search\_recursion(G, 0)

end = time.time()

print('Depth First Search using recursion took: ', end-start, ' milliseconds')

print('Depth First Search using recursion: ', dfsr)

print()

print('============================================================')

print()

print('How do you want to generate a path?')

print('Type 1 for Breadth First Search')

print('Type 2 for Depth First Search')

print('Type 3 for Depth First Search (Recursively)')

print()

ans = int(input('Your Choice:'))

#sends the path to generate a path from start to end

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

start = time.tie()\*1000

#1 generates path with BFS

if ans == 1:

path(plot,bfs,(num\_cells)-1,maze\_cols-.5,maze\_rows-.5)

#2 generates a path with DFS

elif ans== 2:

path(plot,dfs,(num\_cells)-1,maze\_cols-.5,maze\_rows-.5)

#3 generates path with DFS recursive

else:

path(plot,dfsr,(num\_cells)-1,maze\_cols-.5,maze\_rows-.5)

end = time.time()\*1000

print('To generate path, it took', end-start, 'milliseconds to generate a path from start to finish')

plt.show()

“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”