CS 2302 Data Structures

Spring 2019

MW 10:30-11:50 in CCSB 1.0202

LAB # 7

Claudio Angel Garcia

Instructor: Olac Fuentes

TA: Anindita Nath, Maliheh Zargaran

IA: Eduardo Lara

Peer leader: Erick Macik

Introduction

In this lab we are tasked to modify lab 6 by adding various other functions. First, we need to add print statements to the program in case of removing less than n-1 walls, removing more than n-1 walls, and when removing exactly n-1 walls. When asking the user how many walls they want to remove, assuming m is the input, If m < n − 1 then a path from source to destination is not guaranteed to exist, if m = n − 1 then there is a unique path from source to destination, if when m > n − 1 then there is at least one path from source to destination.

The second task is to build an adjacency list representation of the maze, when doing this we need to check If two cells u and v are contiguous and there is no wall separating them, then there must be an edge from u to v in the graph before being able to build the adjacency list.

The third task is to find a solution to the maze created by using different algorithms and methods, one method should find the solution using breadth-first search, the second one should use Depth-first search using a stack and the third one should use a recursive DFS.

Solution

For the first given instruction first I ask the user for the number of wall that they want to remove, then with that number we calculate in an if statement if the number of columns time then number of rows - 1 is equal to the number of walls then we print “There is a unique path from source to destination”, then elif the number of columns times the number of rows is greater than the number of the walls to remove then we print “There is a unique path from source to destination”, else of all of this would print “There is at least one path from source to destination.”

For building the adjacency list of the maze first we take an edge list then for the size of the list we will append the content of the list in [i][1] to [i][0] and vice versa, then return the adjacency list.

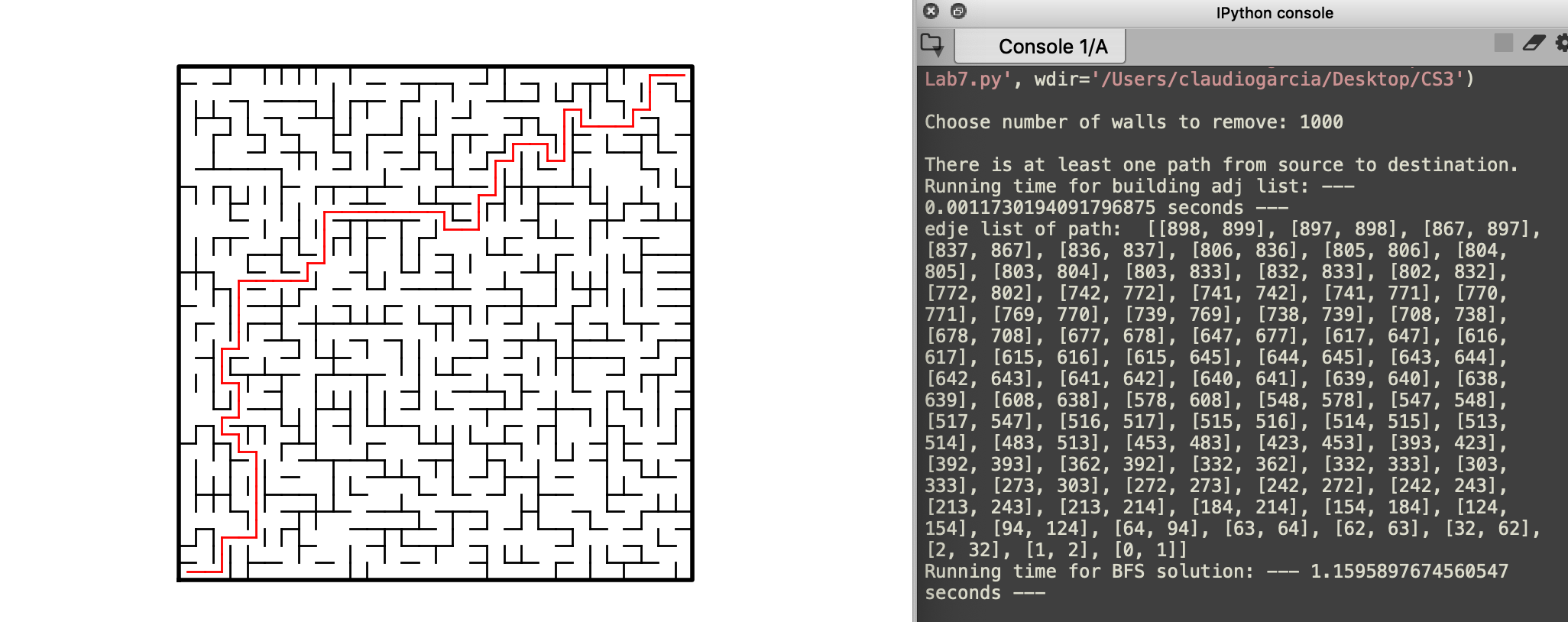
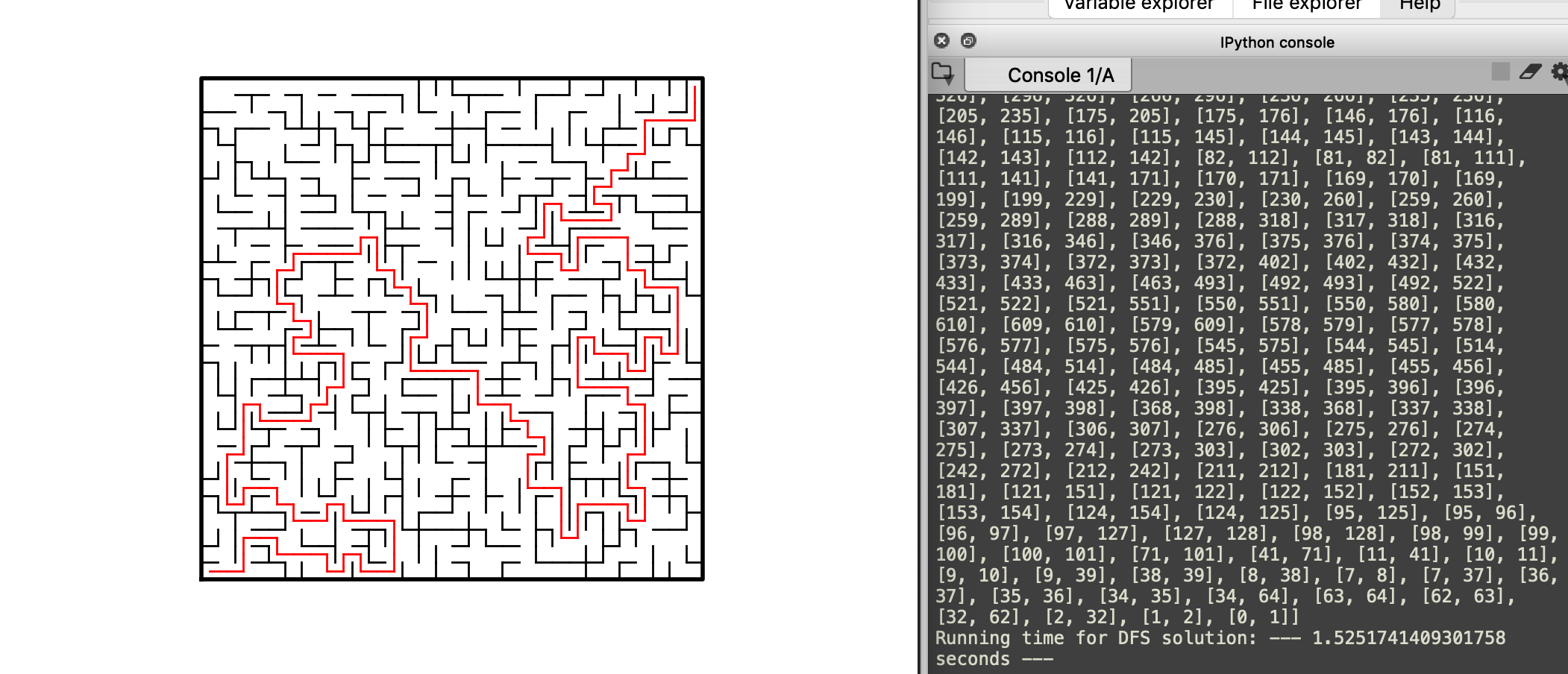
For the last problem we have to find the solution to the maze using BFS, DFS and the recursive of DFS. For the BFS we Initialize a previous array , a Q, and a visited variable. First we spend the first elect to the Q and check that has been visited in the visited variable. Then we initialize a while loop that ends if wanted vertex has been reached end, else a for loop that appends to queue if vertex has not been visited and checks that has been visited and after that we start the vertex prev[0] = -1 since because no vertex points to this one and return prev.

For the DFS we have to use a stack and we initialize prev and visited in the same way as BFS, then we append first element to stack S and check as True. Then we do a while loop that if True continue if the vertex has been reached end, else, for list[n] if visited is False then check that to true and append that to the stack and also check if Stack is empty then append the list[0][1] and return the previous array.

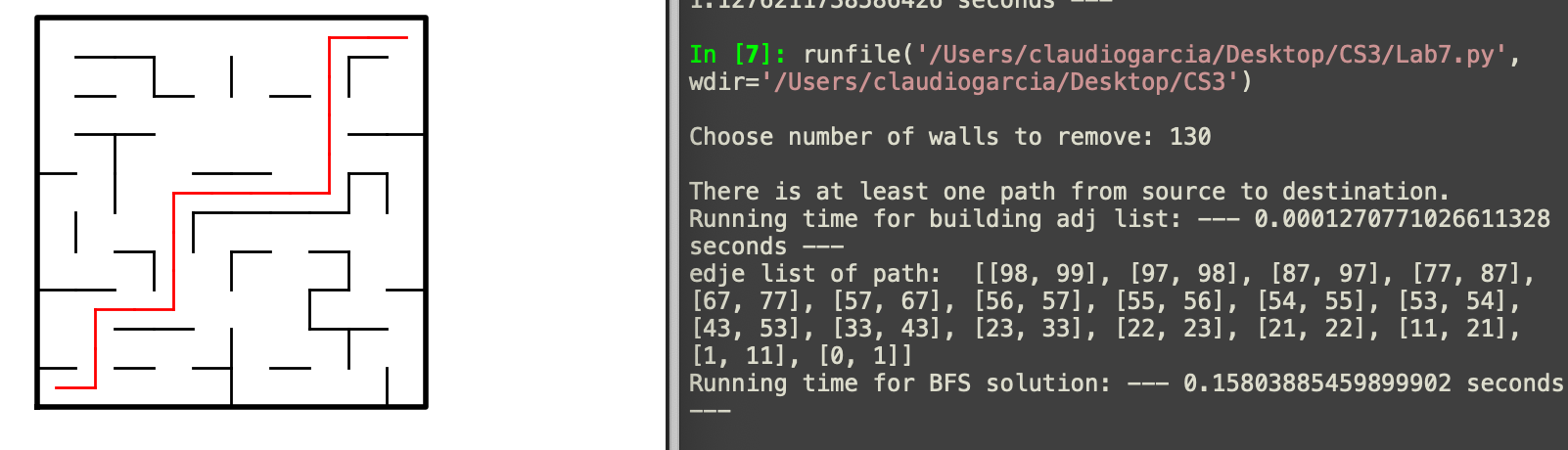
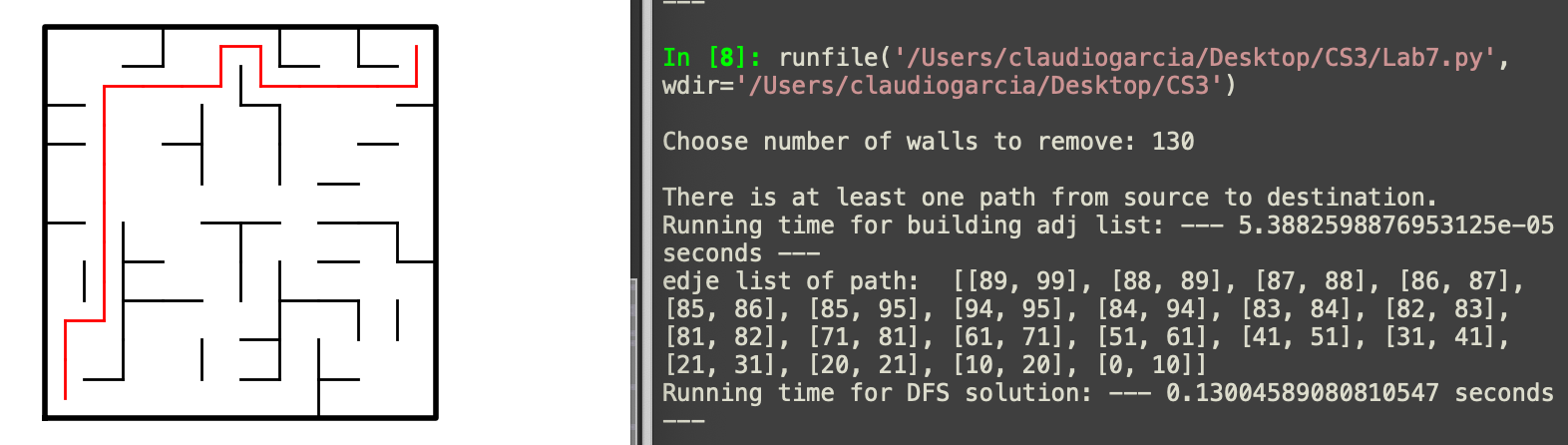
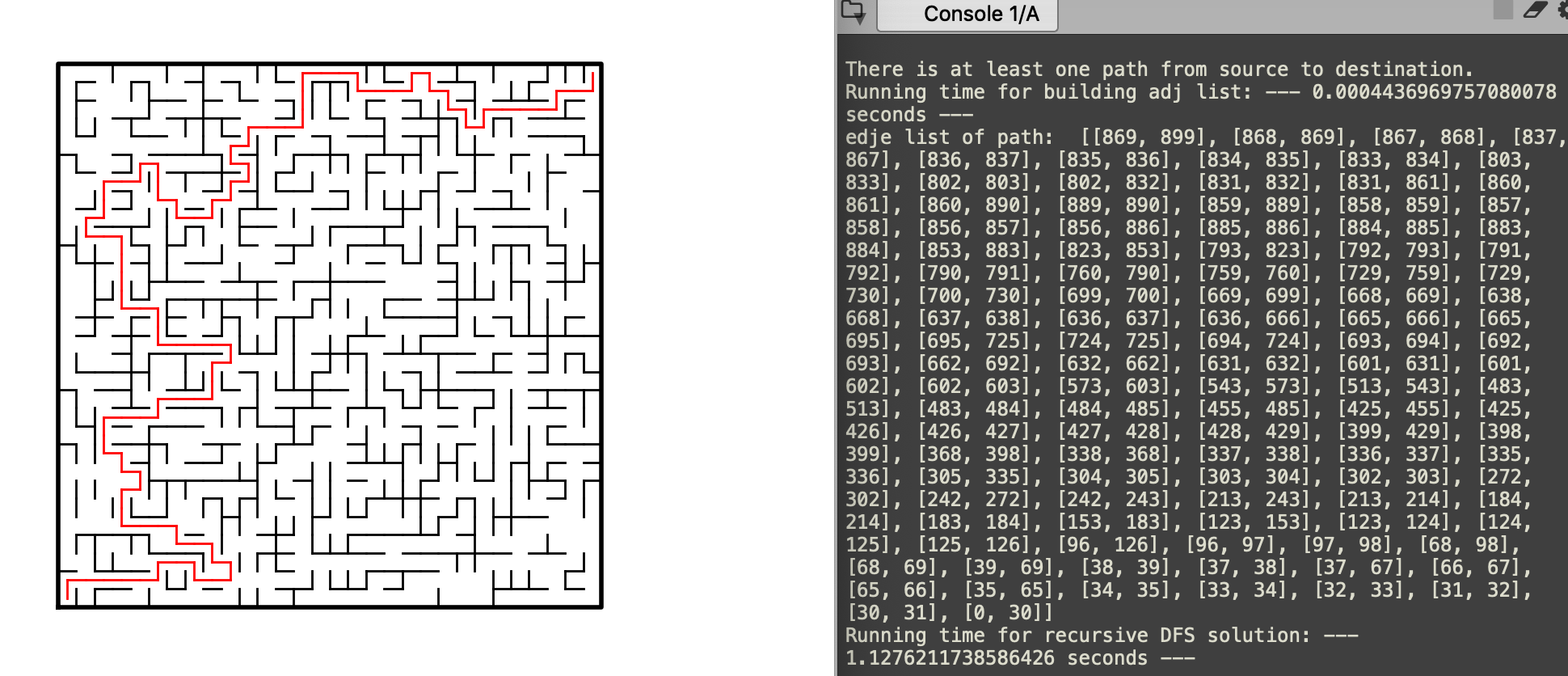
For the recursive method of DFS we need the list, an origin, the previous array and a visited variable, then for I in the list origin then if visited at the index i is false then previous[i] is equal to the origin and the we do the recursive call and return the previous array.

Breadth first search 30x30 removing 1000 walls

Experimentation



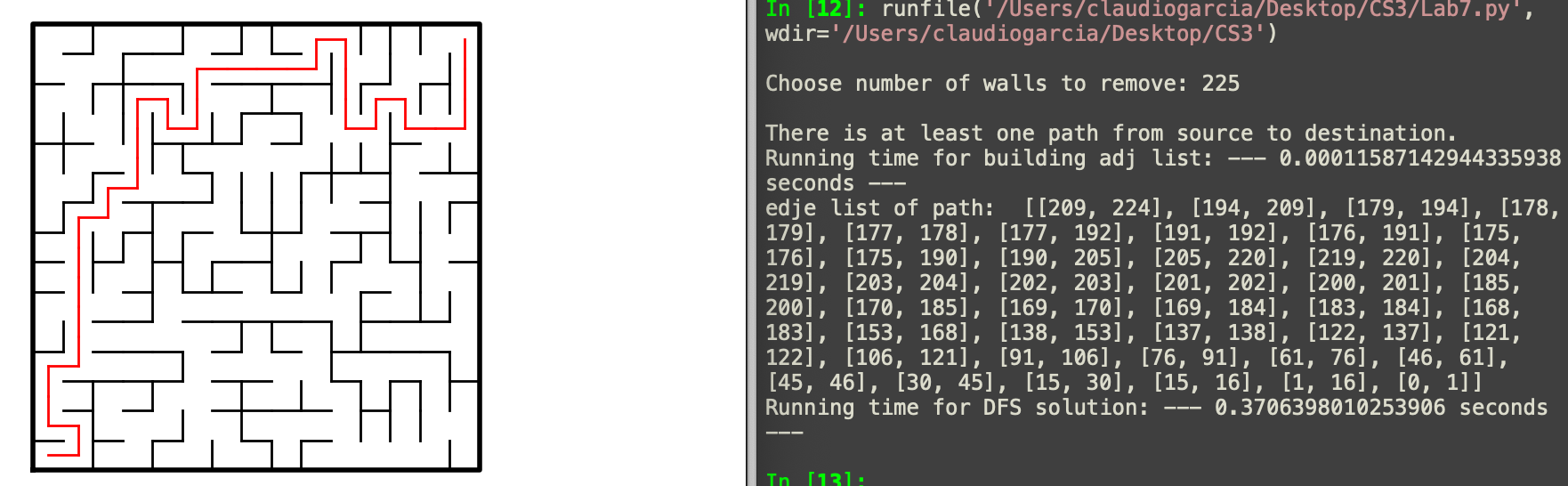
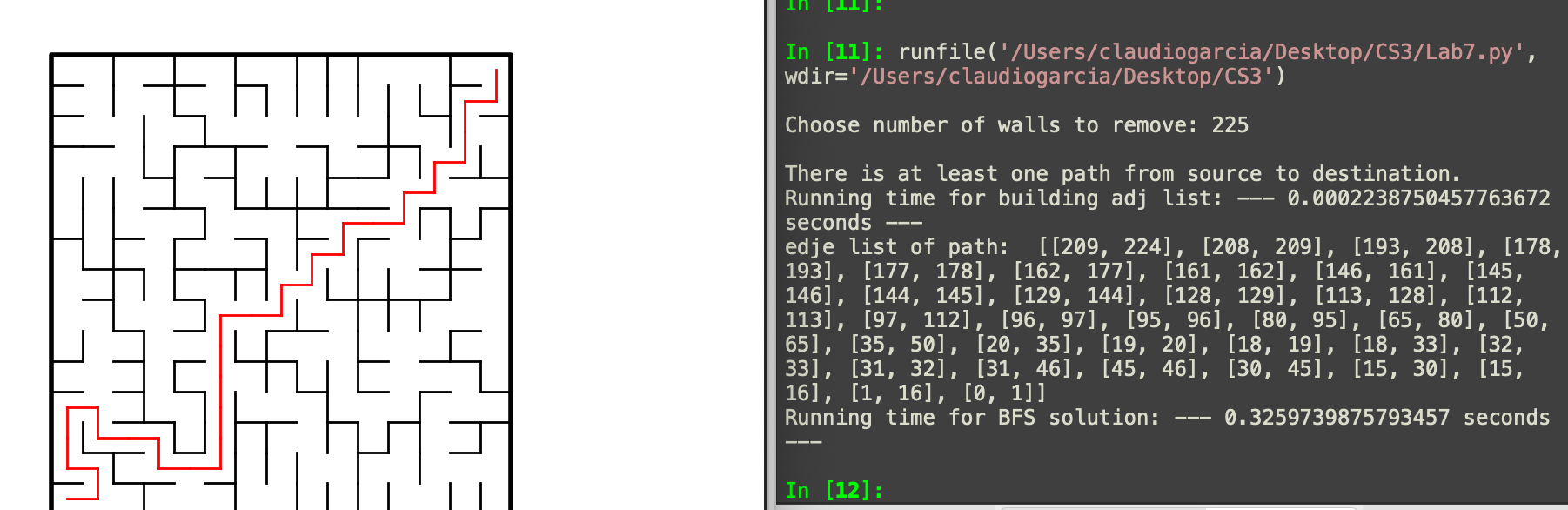
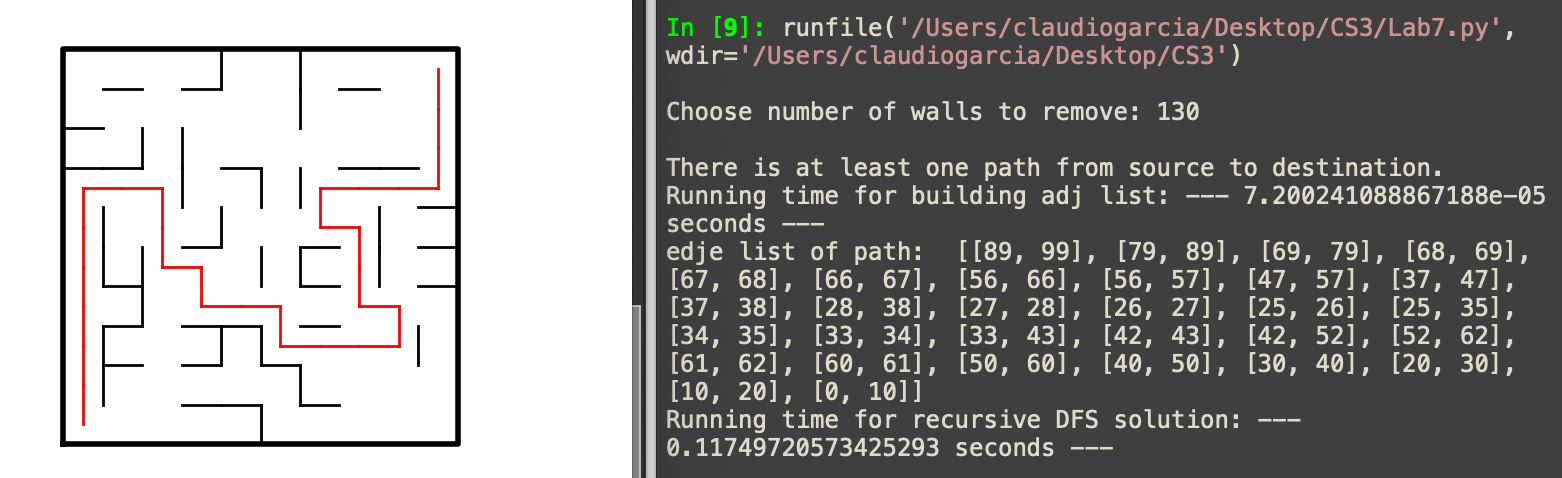
Depth first search 30x30 removing 1000 walls



Recursive Breadth first search 30x30 removing 1000 walls

Breadth first search 10x10 removing 1000 walls

Breadth first search 10x10 removing 1000 walls

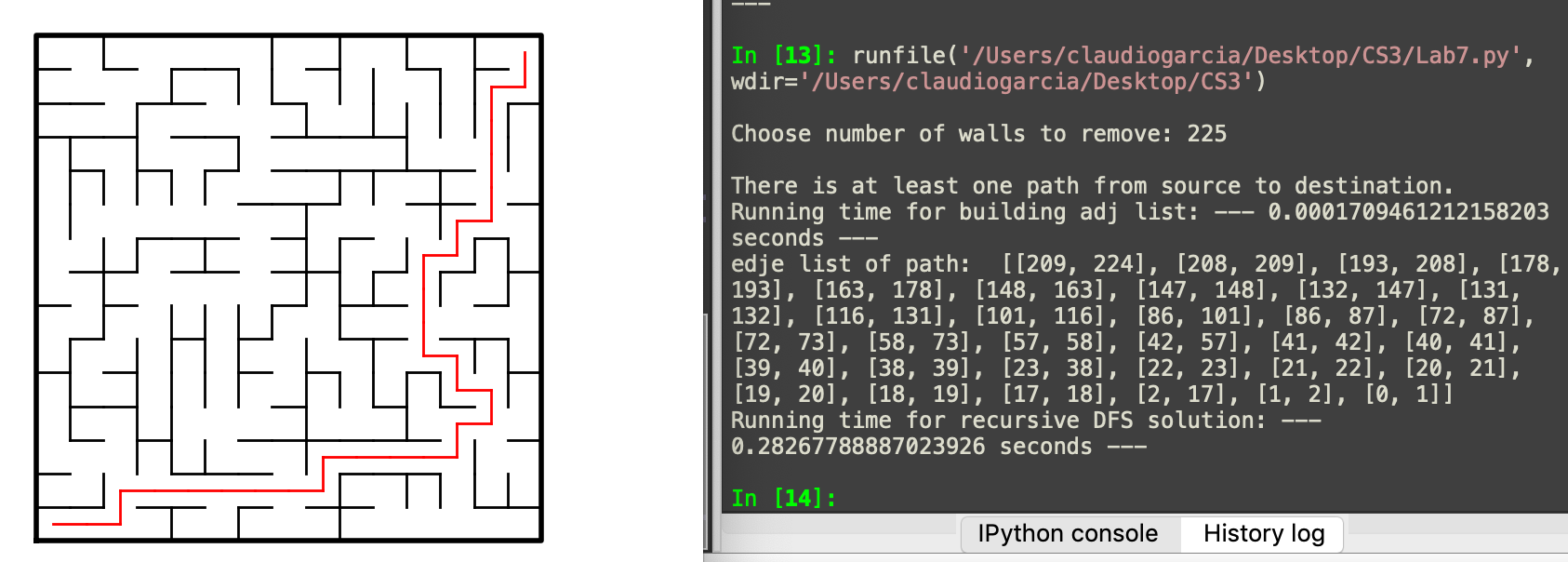


Recursive Depth first search 10x10 removing 1000 walls

Breadth first search 15x15 removing 225 walls

Depth first search 15x15 removing 225 walls

Conclusion

With this lab I learned more about disjoint sets forests and how to implement them in the creation of mazes. Also I learned about more algorithms that can be used to solve the maze, these algorithms are breadth first search and depth first search.

Recursive depth first search 15x15 removing 225 walls

Appendix

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 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: # Do nothing if i and j belong to the same set

S[rj] = ri # Make j's root point to i's root

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 setAmount(s): #sets in the disjoint set forests

c=0

for i in range(len(s)):

if s[i]<0:

c+=1

return c

def removeComp(s, walls, numSets): #remove random parts of the wall

while numSets>1:

w = random.choice(walls) #random wall selection

i = walls.index(w) #position of wall

if find(s, w[0]) != find(s, w[1]): ##if root of w[0] is not the same as w[1]

walls.pop(i) #wall removal

union\_by\_size(s, w[0], w[1]) #wall union after the removal

numSets -=1

return w

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

fig, ax = plt.subplots()

for p in path:

if p[1]-p[0] != 1:

# Vertical Path

px0 = (p[1]%maze\_cols)+.5

px1 = px0

py0 = (p[1]//maze\_cols)-.5

py1 = py0+1

else:

# Horizontal Path

px0 = (p[0]%maze\_cols)+.5

px1 = px0+1

py0 = (p[1]//maze\_cols)+.5

py1 = py0

ax.plot([px0,px1],[py0,py1],linewidth=1,color='r')

for w in walls:

if w[1]-w[0] == 1: # Vertical Wall position

x0 = (w[1]%maze\_cols)

x1 = x0

y0 = (w[1]//maze\_cols)

y1 = y0+1

else: # Horizontal Wall postion

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('on')

ax.set\_aspect(1.0)

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 Maze\_User(m,S,W):

edge\_list = []

counter = 0

path = False

while counter < m:

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

# if roots are different

if find(S,W[d][0]) != find(S,W[d][1]):

# Join sets

union(S,W[d][0],W[d][1])

# Delete wall

edge\_list.append(W.pop(d))

counter += 1

if counter >= len(S)-1:

path = True

elif path: # if there a path start removing any walls

union(S,W[d][0],W[d][1])

edge\_list.append(W.pop(d)) # Wall removal

counter += 1

return edge\_list

def edge\_list\_to\_adj\_list(G,size):

adj\_list = [[] for i in range(size)]

for i in range(len(G)):

adj\_list[G[i][0]].append(G[i][1])

adj\_list[G[i][1]].append(G[i][0])

return adj\_list

############################ LAB 7 #####################################

def BreadthFirstSearch(adj):

prev = np.zeros(len(adj), dtype=np.int)-1 # Initialize prev array

visited = [False]\*len(adj) # No vertex has been visited

Q = []

Q.append(adj[0][0]) # Insert the first element to the queue

visited[adj[0][0]] = True #Vertex has been visited

while Q:

if prev[len(adj)-1] >= 0: #if wanted vertex has been reached end

break

n = Q.pop(0)

for j in adj[n]:

if visited[j] == False: # append to queue if vertex has not been visited

visited[j] = True

prev[j] = n

Q.append(j)

#starting vertex = -1 since bc no vertex points to this one

prev[0] = -1

return prev

def DepthFirstSearch(adj):

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

visited = [False]\*len(adj)

S = []

S.append(adj[0][0])

visited[adj[0][0]] = True

visited[0] = True

prev[adj[0][0]] = 0

while True:

# continue if the vertex has been reached end

if prev[len(adj)-1] >= 0:

break

n = S.pop()

for j in adj[n]:

if visited[j] == False:

visited[j] = True

prev[j] = n

S.append(j)

if S == []:

S.append(adj[0][1])

return prev

def DepthFirst\_recursive(adj,origin,visited,prev):

visited[origin] = True

for i in adj[origin]:

if visited[i] == False:

prev[i] = origin

DepthFirst\_recursive(adj, i, visited, prev)

return prev

def prevEdge(prev):

E = []

i = len(prev)-1 # i equal to the end

while True:

if prev[i] == 0 or prev[i] < 0: # Base case when prev is equal to 0 or less

E.append([0,i])

break

elif i < prev[i]: # Edges in the order of small,big

E.append([i,prev[i]])

else:

E.append([prev[i],i])

i = prev[i]

return E # Return the edge list

plt.close("all")

maze\_rows = 15

maze\_cols = 15

m = int(input("Choose number of walls to remove: "))

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

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

if (maze\_cols\*maze\_rows)-1 == m:

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

elif (maze\_cols\*maze\_rows) > m:

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

else:

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

edge\_list = Maze\_User(m,S,W) #Edge list from randomly created maze

start\_time = time.time()

adj\_list = edge\_list\_to\_adj\_list(edge\_list,maze\_cols\*maze\_rows) # convert edge list to adj list

print('Running time for building adj list: '"--- %s seconds ---" % (time.time() - start\_time))

#start\_time = time.time()

#prev = BreadthFirstSearch(adj\_list) # Function ends when goal has been reached to shorten time

#path = (prevEdge(prev))

#print("edje list of path: ",path)

#draw\_maze\_path(W,path,maze\_rows,maze\_cols)

#print('Running time for BFS solution: '"--- %s seconds ---" % (time.time() - start\_time))

#start\_time = time.time()

#prev = DepthFirstSearch(adj\_list)

#path = (prevEdge(prev))

#print("edje list of path: ",path)

#draw\_maze\_path(W,path,maze\_rows,maze\_cols)

#print('Running time for DFS solution: '"--- %s seconds ---" % (time.time() - start\_time))

#start\_time = time.time()

#visited = [False]\*len(adj\_list)

#p = np.zeros(len(adj\_list),dtype=int)-1

#prev = DepthFirst\_recursive(adj\_list,0,visited,p)

#path = (prevEdge(prev))

#print("edje list of path: ",path)

#draw\_maze\_path(W,path,maze\_rows,maze\_cols)

#print('Running time for recursive DFS solution: '"--- %s seconds ---" % (time.time() - start\_time))

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.

* Claudio Garcia