Rigoberto Quiroz

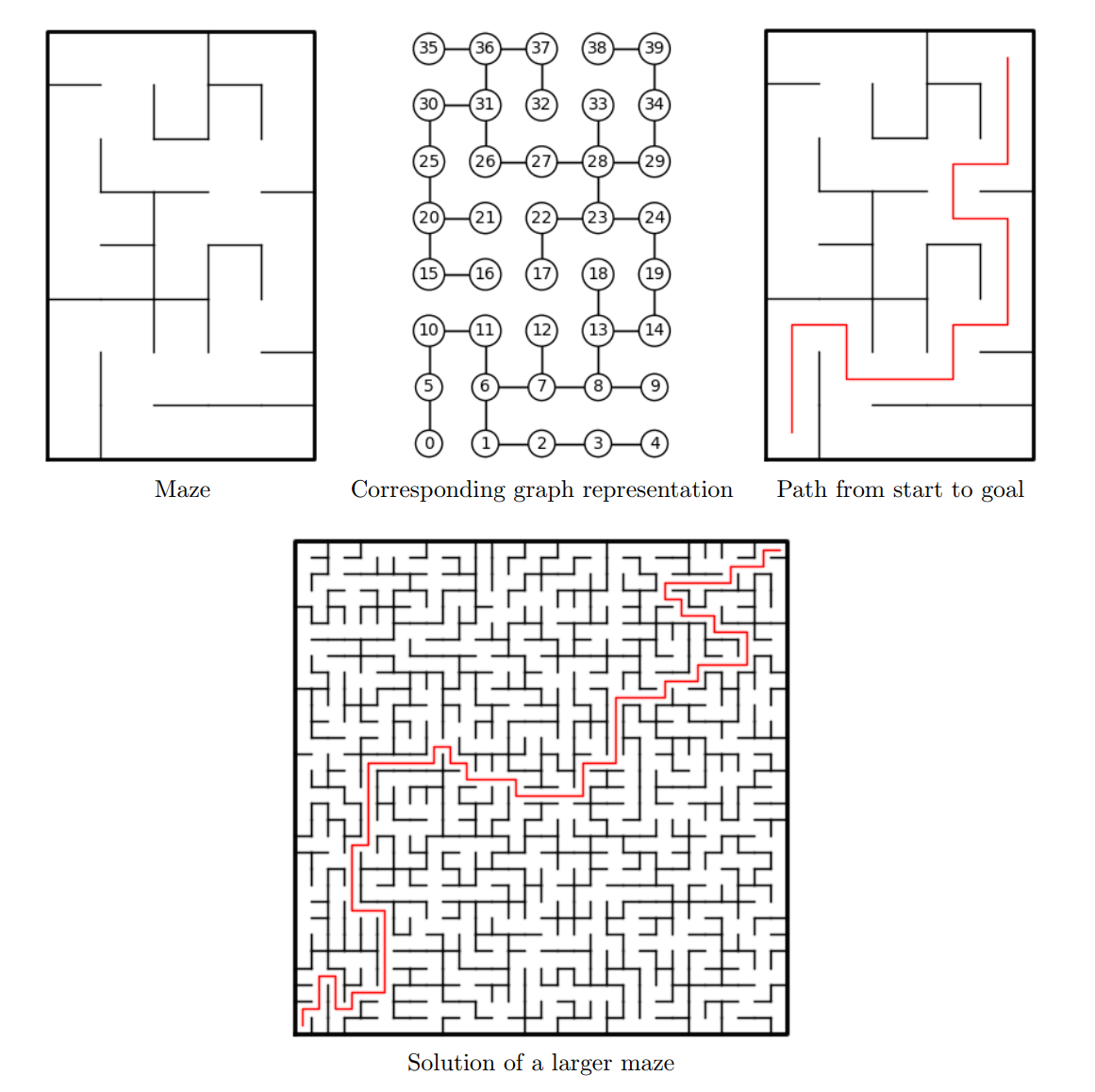
4/29/19

CS2302 1:30 PM – 2:50 PM

Lab7 Report

Description:

For this lab I had to modify out previous lab 6 and certain sections and add some searching algorithms. In this lab we must remove a certain number of walls, depending on what the user wants. The number of walls we have in our maze, and how many we remove affect if there are zero or more than 1 path to each vertex. We must print out if in our maze we have a possible path from point A to B, if we have a path then we will print it out.



The way I was able to solve this lab was when removing the walls from our maze, I stored the set of lists into a list. creating a 2D array, or a list of lists. Once we have removed all the walls, our list we contain all removed walls. Then I created a adjacent list from the list of lists, having repetition. We have to have repetition in our adjacent list since it might create a directed graph, making some type of mazes or graphs not reachable even though a path exits. Once the adjacent list was created, then the searching algorithms will take the adjacent list and find try to find paths to our destination. Since we are using different types of algorithms, we might find different paths from our point of origin to our destination.

**7.1 Paths to Destination:**

For this section of the code we must know if a path will exist from point A to point B. Depending on the number of cells in our maze we might have a path. If M (Number of walls removed) is greater than the N (number of cells) – 1, then they might be at least one path to our destination. There might be more paths than 1. Otherwise, if M is equal to the N – 1, then that means that there will always be one path from point A to point B. If M is less than N -1 then a path from A to B might not exists. Since we know measurements of our maze, we can predict if a path exists or not.

**7.2 Adjacent List:**

For this section of the code, when we are removing the walls, we have a list that will store all the walls we removed. In this method, we will send the list of removed walls, and the total number of cells in out maze. In the method we will create a list with length of cells, and we will have a loop that will take each sub list, taking the first value as the index, and storing the second value, we will also do the same for the second value. Make the second value our index and store the first value in it. I did this because at some points we might not be able to find a path even though a path does exist. Once done we will return the adjacent list.

**7.3 Searching Graph Algorithms:**

**A) Breadth-First Search:**

For this method we send the adjacent list and a point from origin. In this method we will create several list, visited, previous and queues. Our visited list will keep track of vertices we have touched. Previous list will keep track of vertices that came before the current vertex. Queue will tell the algorithm which vertex we will visit next. In this method we will have a loop, as long as we have elements in our queue list we will delete that element from our list and check if the element we just pop, have connecting vertices which we have not visited before. If we haven’t visited those vertices, then we will enter that value in our previous list and mark it as visited. We will do this until we no longer have elements in our queue.

**B) Depth-First Search (Stacks):**

This method will work exactly the same as Breadth-First Search. The only difference this method will have is that instead of using a stack. The same process will work exactly the same. Our stack will tell us which vertex we will advance to, possibly creating a different path from point A to B.

**C) Depth-First Search (Recursive):**

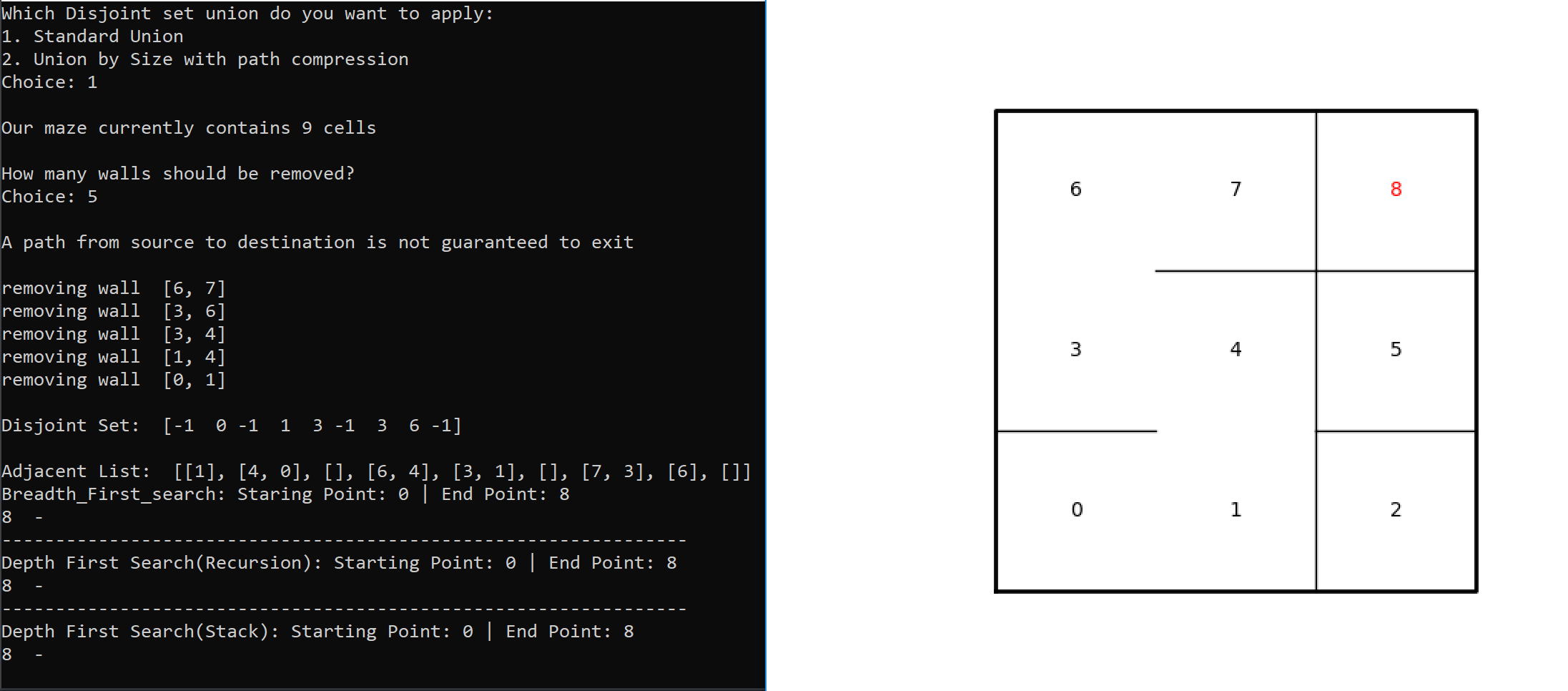
For this method, since we are doing it recursively we will define visited and previous as global variables. Sending graph and a point of origin. Since we are starting with our origin point we will mark it as visited, then we will check if that vertex has other connected vertices, if it does then we will go to that vertex, we will advance using recursion. We will repeat this process until we visited all the vertices.

**Test Cases:**

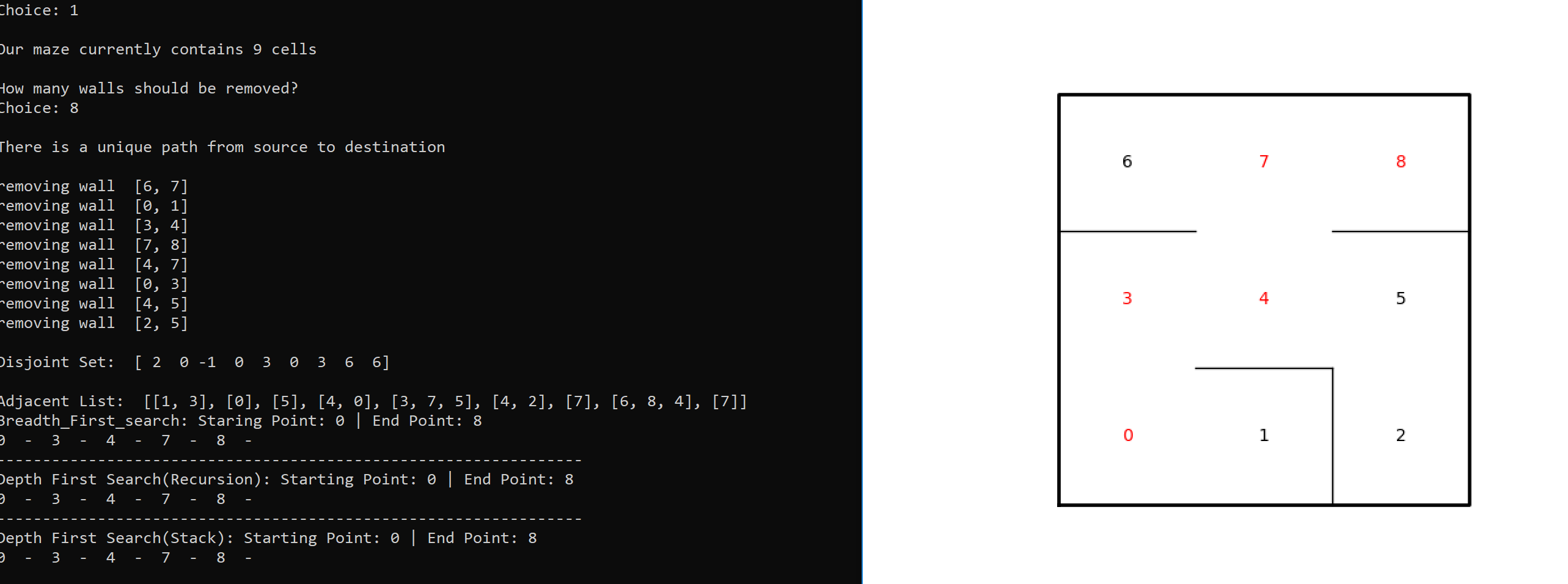
Standard Union:

Maze 3x3

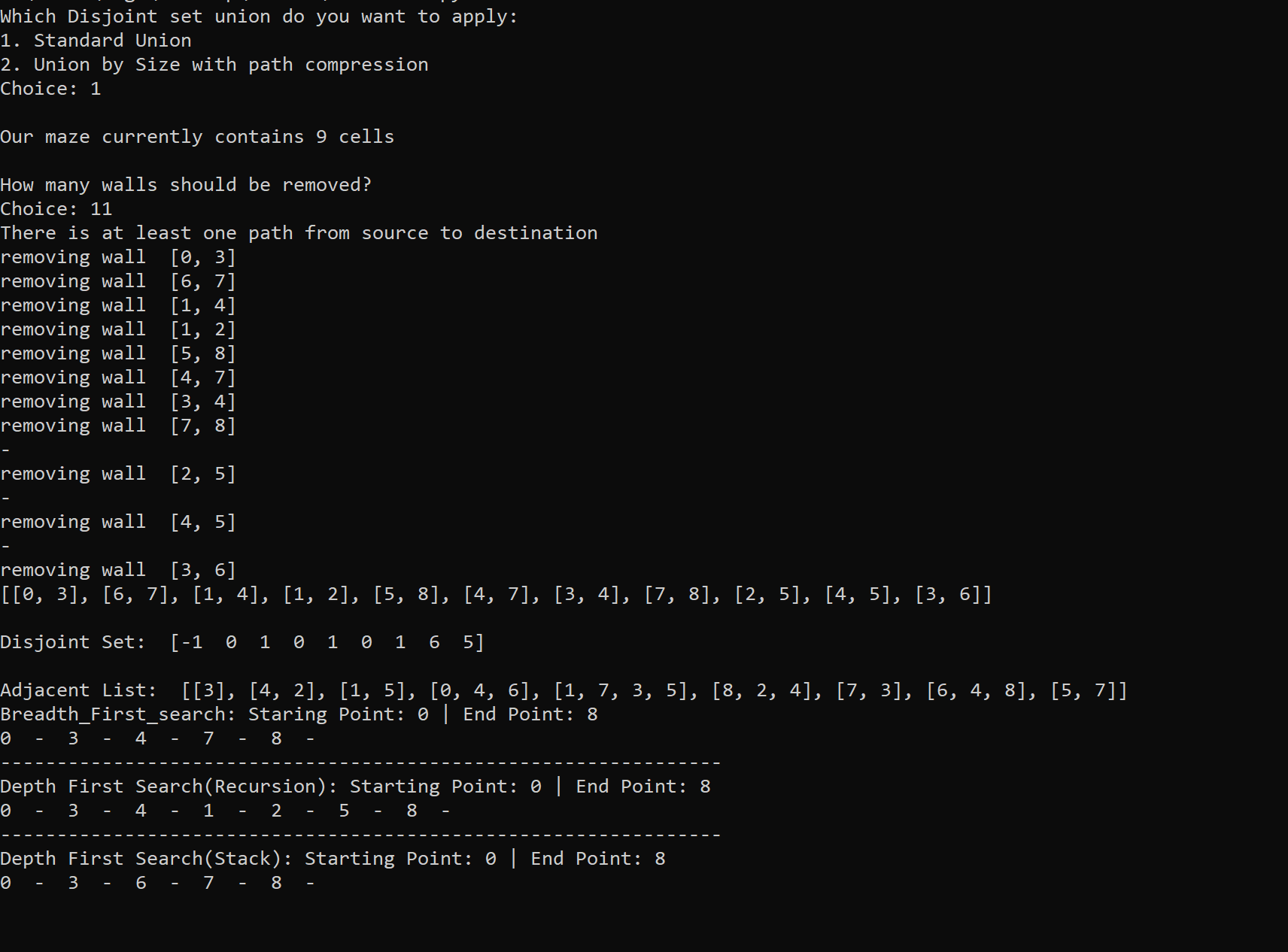
Removed Walls: 5



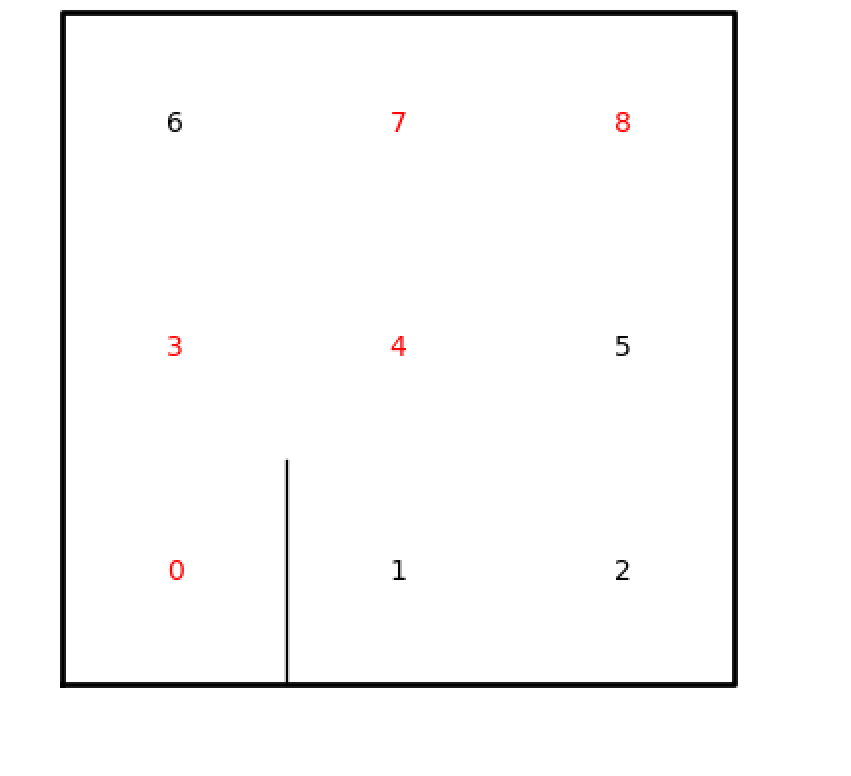
Removed Walls 8



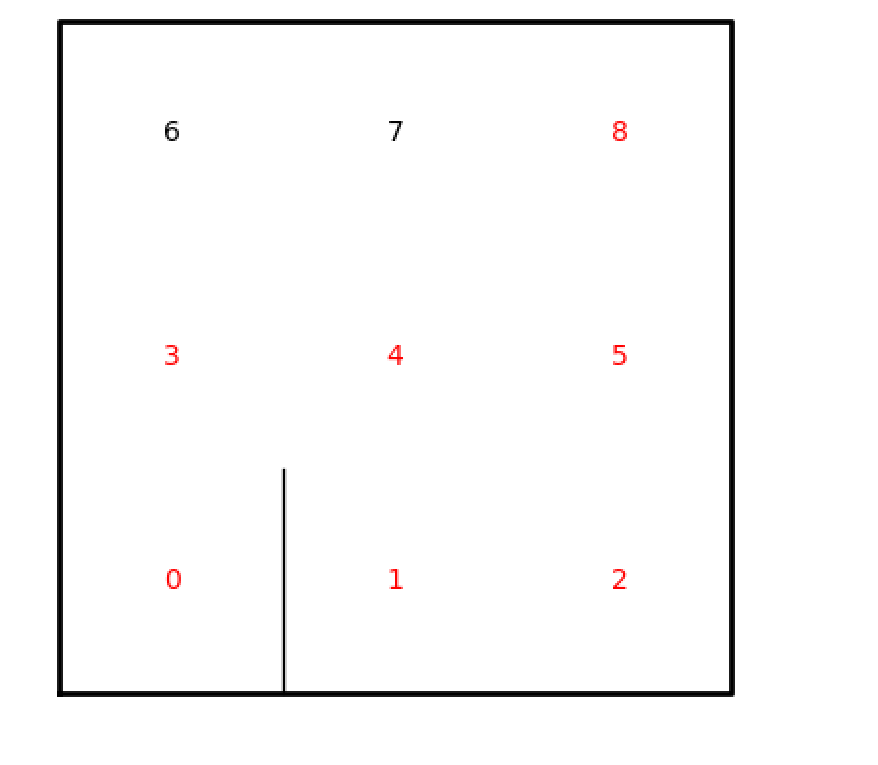
Removed Walls 11



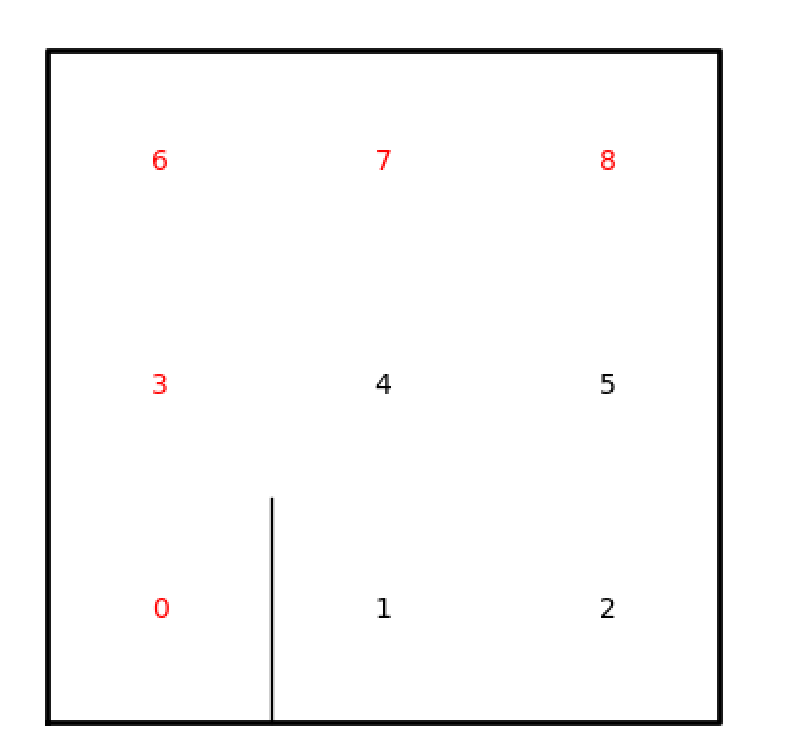
Breadth-First Search



Depth-First Search (Recursive)



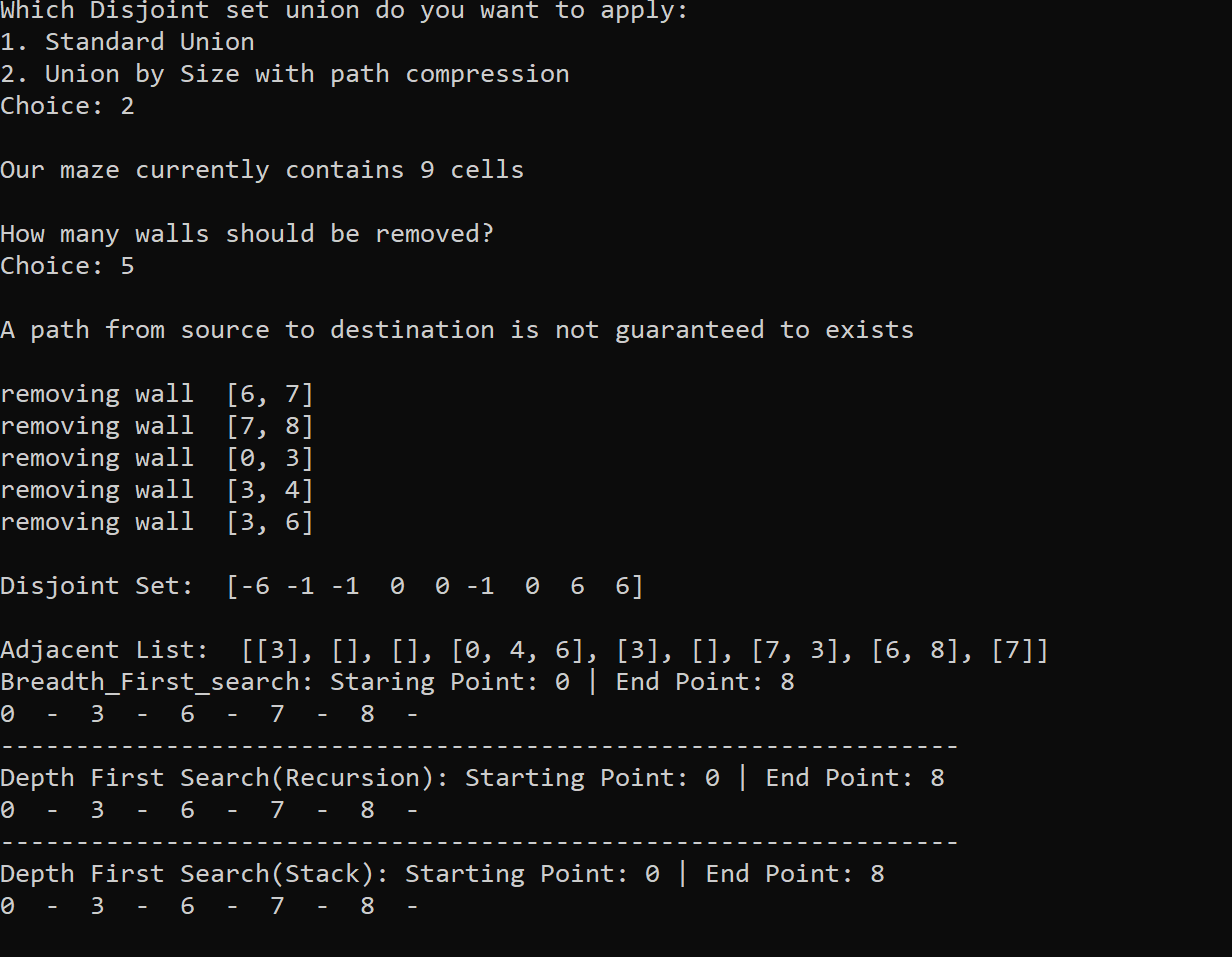
Depth-First Search (Stacks)

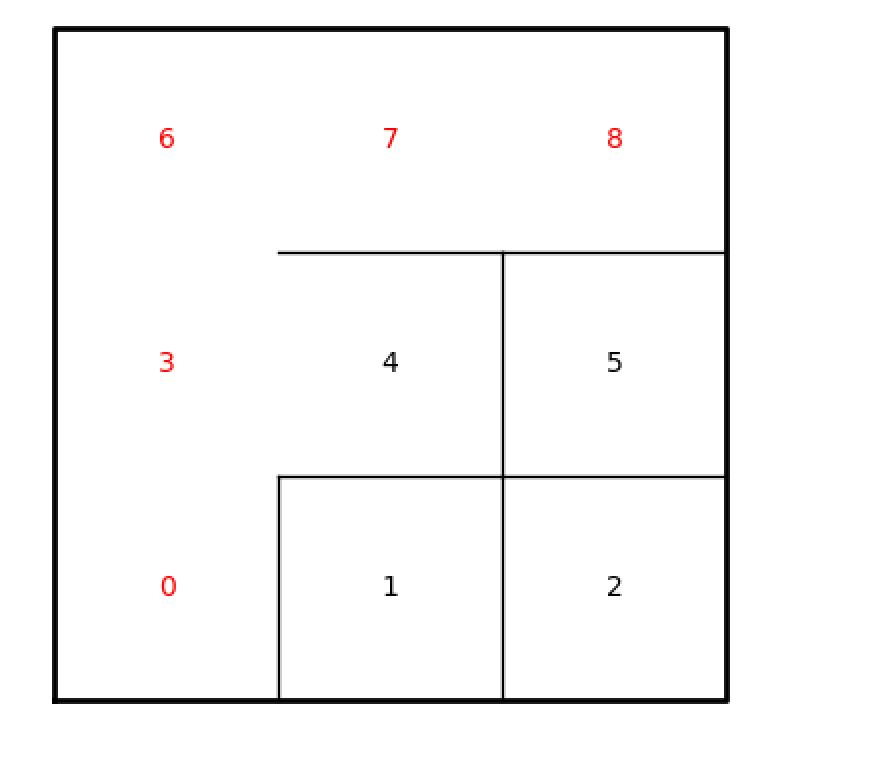


Union by Compression:

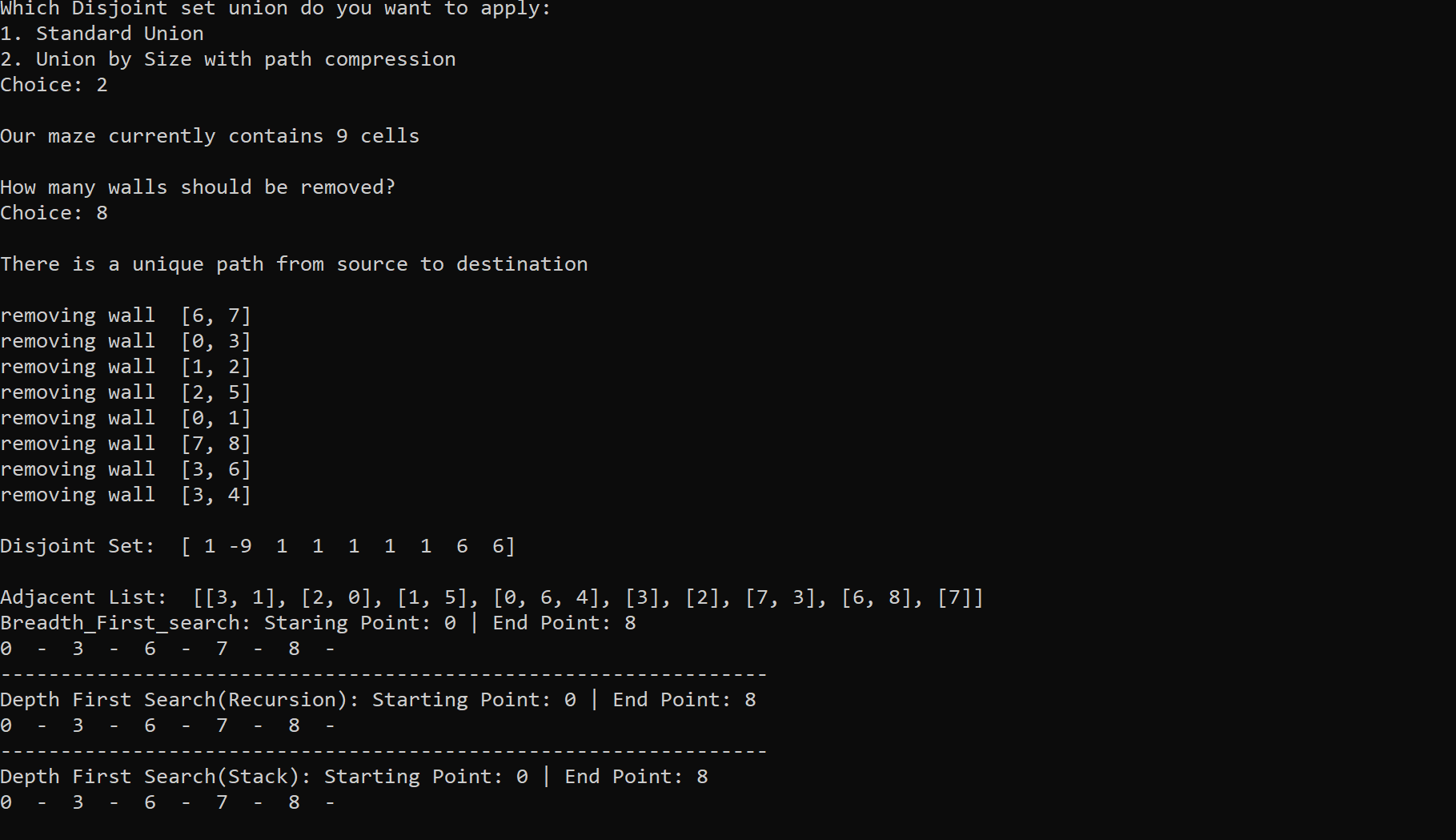
Maze 3x3

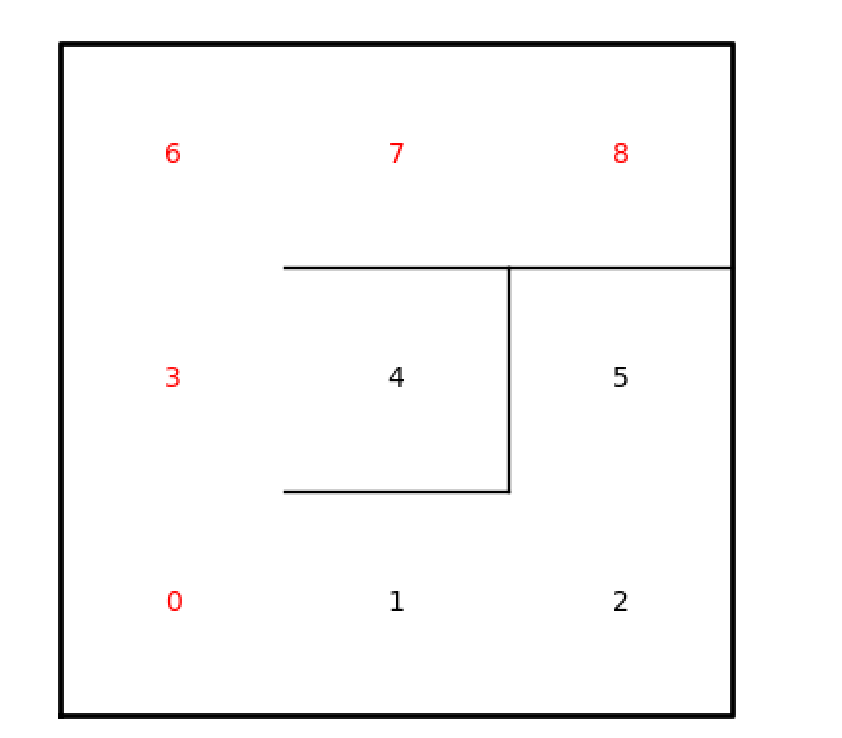
Removed Walls 5





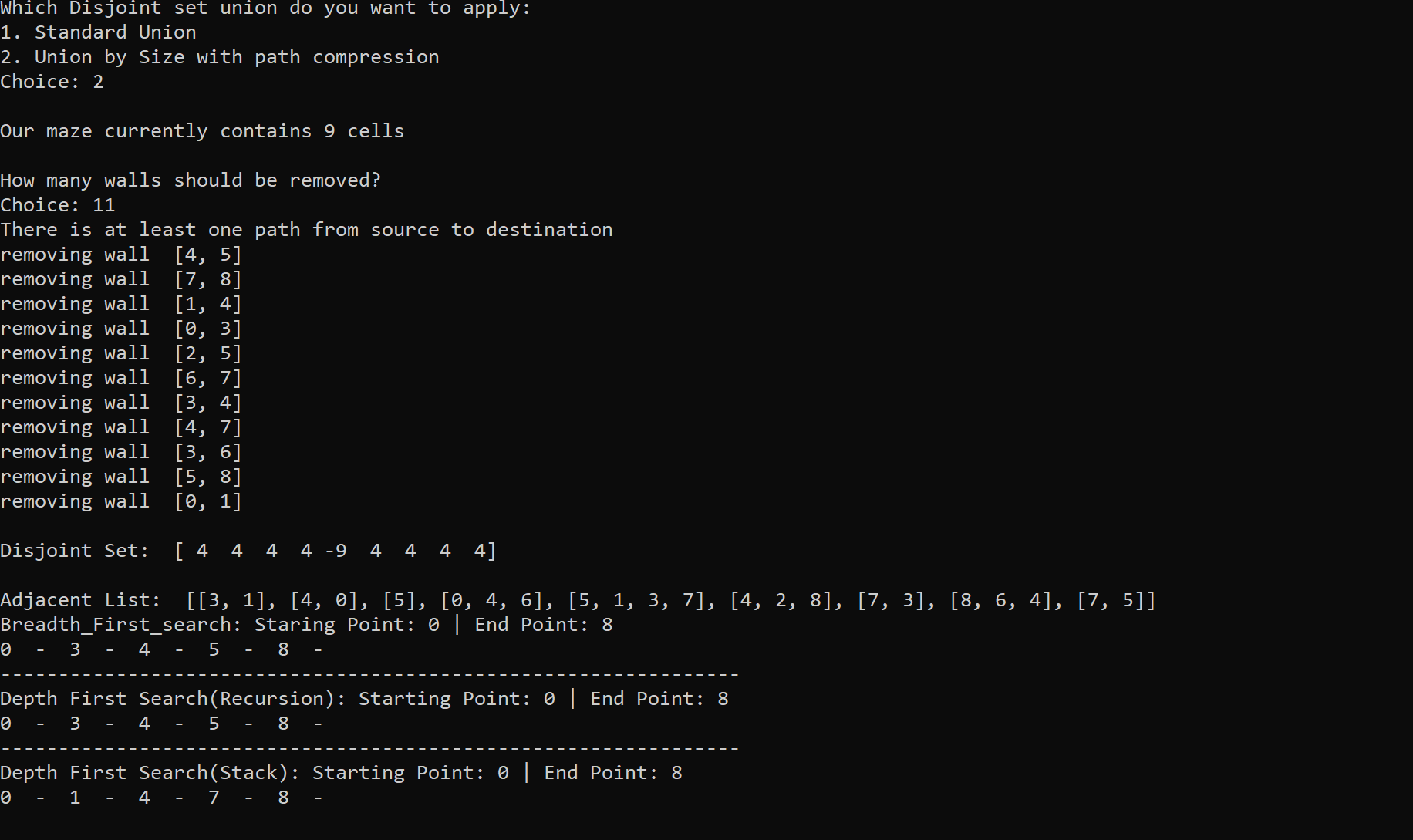
Removed Walls 8

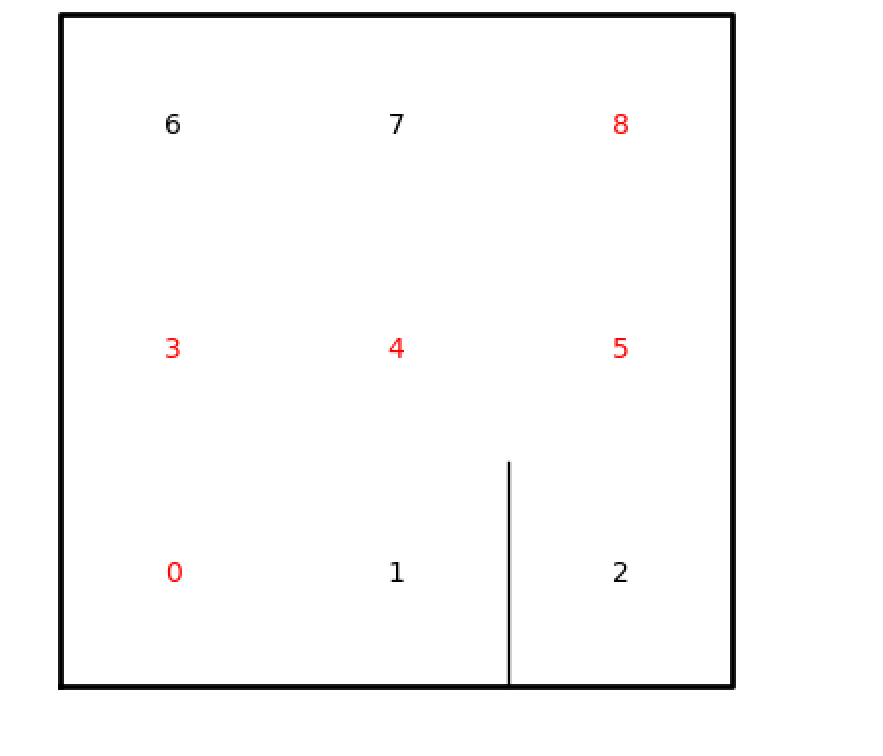




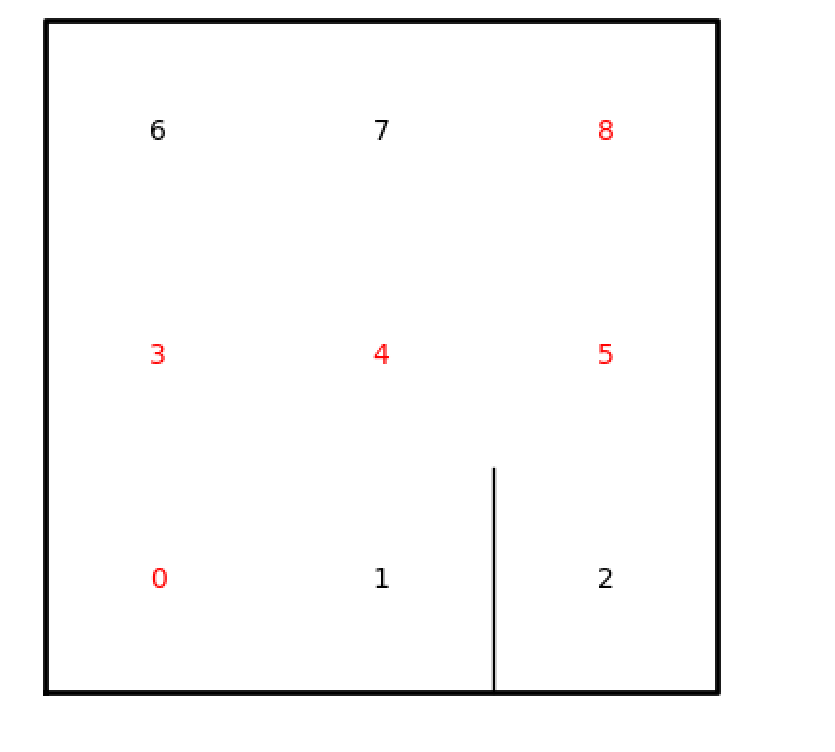
Removed Walls 11

Breadth-First Search

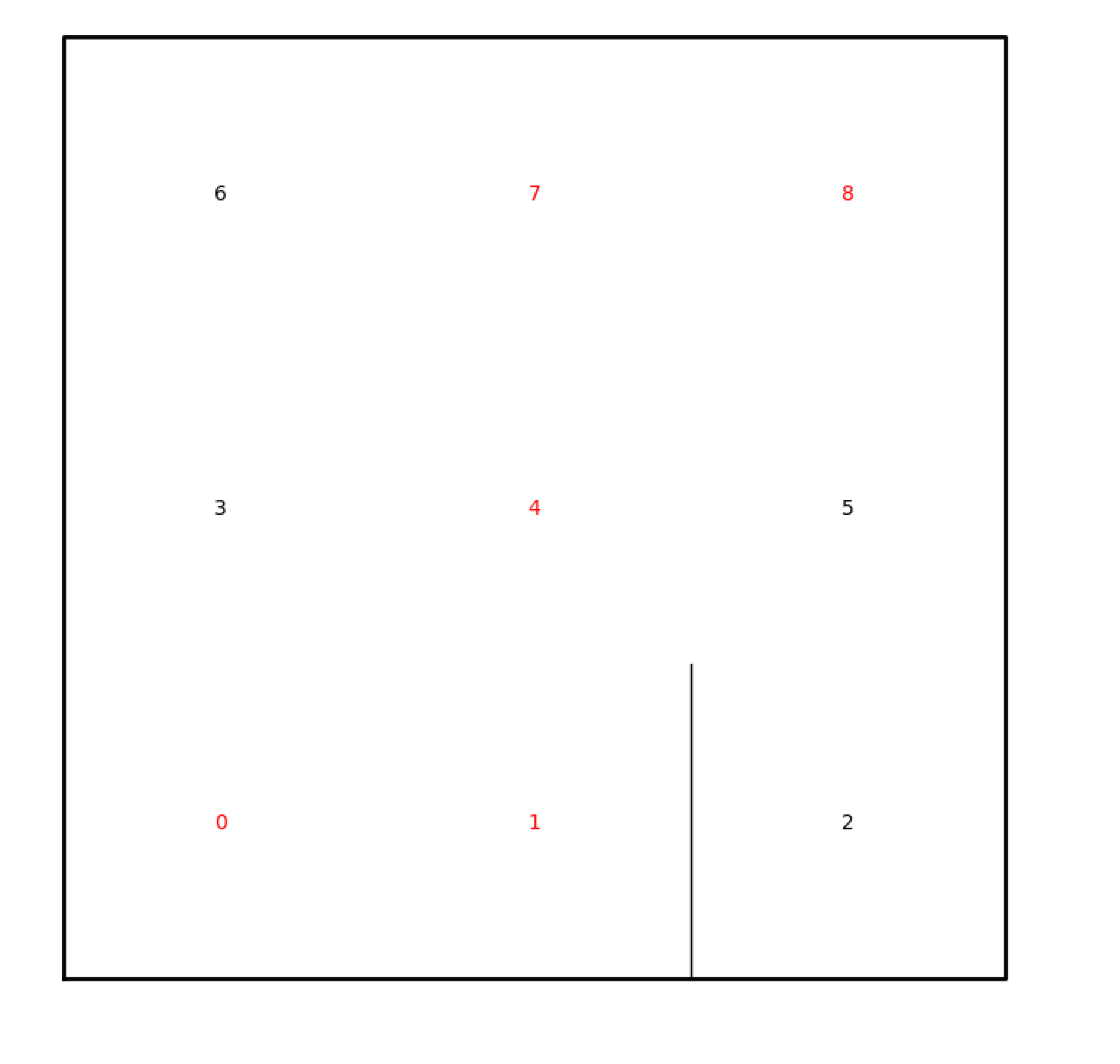




Depth-First Search (Recursive)



Depth-First Search (Stack)



**Summary:**

What I learned from this lab was how different number of walls removed may affect if a path exist or not, as well as how useful graph algorithms such as adjacent list are in finding paths from point A to B if it exists. I was also able to learn how different graph searching algorithms function and how they might have different paths from point A to point B. With these searching algorithms I was also able to tell which took the longer path and which took the shorter path.

**Running Times:**

**Appendix:**

# Author: Rigoberto Quiroz

# Section: 1:30 PM - 2:50 PM

# This program will create a create maze with x rows and y columns. Then we will

# create lists that contain values that are next to each other. Then we will create

# a Disjoint set and set a while loop that will check the number of sets until

# we have 1 remaining set (contains all sets). Loop will unionize the values in

# our lists, and remove the walls that separate them. Creating pathways that will

# connect points A to B, creating only one path. Once that is done we are going

#to draw the maze, along with printing its Disjoint Set.

import matplotlib.pyplot as plt

import numpy as np

import random

import time

import graphs

import dsf

# Creates Disjoint Set according to the rows and columns of our maze

def DisjointSetForest(size):

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

def union(S,i,j):

# roots of i and j

ri = find(S,i)

rj = find(S,j)

# Do not belong in the same set

if ri !=rj:

# Union

S[rj] = ri

return

# Belong in the same set

return False

def find(S,i):

# Finds the root of our index i

if S[i] < 0:

return i

return find(S,S[i])

# Creates the max according to our x and y values

def draw\_maze(walls,prevL,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

#print([x0,x1],[y0,y1])

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')

i = 0

if cell\_nums:

for r in range(maze\_rows):

for c in range(maze\_cols):

cell = c + r\*maze\_cols

#print(cell)

if cell in prevL:

ax.text((c+.5),(r+.5), str(cell), size=10,

ha="center", va="center", color='r')

else:

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

# Counts the number of Sets we have in our Disjoint Set

def numOfSets(S):

count = 0

# Count + 1, if value is a root

for key in S:

if key <= -1:

count +=1

return count

def find\_c(S,i):

# Finds root of i

if S[i] < 0:

return i

# Sets all values in a set to its root, without going to other values in

# the set

r = find\_c(S,S[i])

S[i] = r

return r

def unionBySize(S,i,j):

# Finds root of i and j

ri = find\_c(S,i)

rj = find\_c(S,j)

# do not belong in the same set

if ri != rj:

# Unionzing Set by size (Larger set)

# Root j is gretaer than root i

if -(S[ri]) < -(S[rj]):

# Keeps track of length of sets

S[rj] += S[ri]

S[ri] = rj

return

else:

# root i is greater than root j

S[ri] += S[rj]

S[rj] = ri

return

# Belong to the same set

return False

def adjList(V, cells):

# Creates a list of size cells

adjL = [[] for i in range(cells)]

# Using the list that stored our removed cells, we are going to place those

# cells into their respective index.

for k in V:

adjL[k[0]].append(k[1])

adjL[k[1]].append(k[0])

# returns adjacent list

return adjL

def path(prev, origin):

# Prints the path from our point of origin to our destination

if prev[origin] != -1:

path(prev, prev[origin])

L.append(origin)

print(origin, ' - ', end=' ')

def breadth\_First\_search(G, origin):

# Creates list if places we have visited

visited = [False for i in range(len(G))]

# Creates list of vertices which we have visited

prev = [-1 for i in range(len(G))]

# How we are going to traverse the graph

queue = []

# starting point

queue.append(origin)

# we have visited our starting point

visited[origin] = True

while len(queue) is not 0:

# pop our starting point and move to vertices that have connections to our

# current vertices

u = queue.pop(0)

for t in G[u]:

if visited[t] is False:

visited[t] = True

prev[t] = u

queue.append(t)

return prev

def depthFirstSearchS(G, origin):

# visited list

visited = [False for i in range(len(G))]

# previous list

prev = [-1 for i in range(len(G))]

# stack, way in which we are going tp traverse the graph

stack = []

stack.append(origin)

visited[origin] = True

while len(stack) is not 0:

# pop our last item in the list and follow path until we have visited all

# vertercies

u = stack.pop()

for t in G[u]:

if visited[t] is False:

visited[t] = True

prev[t] = u

stack.append(t)

return prev

def depthFirstSearchR(G, origin):

# traverse our graph as long we have items in our graph that we have not visited

visited[origin] = True

for t in G[origin]:

if not visited[t]:

prev[t] = origin

depthFirstSearchR(G, t)

return prev

plt.close("all")

# Creates Maze specs, x and y

maze\_rows = 3

maze\_cols = 3

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

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

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

# User can choose which type of union to do, Standard, or compression

print('Which Disjoint set union do you want to apply:')

print('1. Standard Union')

print('2. Union by Size with path compression')

choice = int(input('Choice: '))

# Standard Union

if choice == 1:

# Number of cells we have in our maze

cells = maze\_rows \* maze\_cols

print('\nOur maze currently contains', cells, 'cells')

print('\nHow many walls should be removed?')

remWalls = int(input('Choice: '))

# number of remove walls in less than 0

if remWalls < 0:

print('you cannot remove negative walls')

exit(0)

# number of removed walls is less than cells -1

if remWalls < cells-1:

print('\nA path from source to destination is not guaranteed to exit\n')

vertices = []

# while we still have number of walls to be removed

while remWalls != 0:

# Select a random list and unionize them

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

if union(S,walls[d][0],walls[d][1]) != False:

print('removing wall ',walls[d])

# remove them from lists

# add it to a vertices list

vertices.append(walls[d])

walls.pop(d)

# decrease amount of remove walls

remWalls -= 1

# Final Disjoint Set, and maze drawing

print('\nDisjoint Set: ',S)

# Adjacent List

adL = adjList(vertices, cells)

print('\nAdjacent List: ',adL)

# Searching path using Breadth\_First\_search

print('Breadth\_First\_search: Staring Point: 0 | End Point:', cells-1)

L = []

path(breadth\_First\_search(adL,0),cells-1)

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

print()

print('----------------------------------------------------------------')

# Searching path using Depth First Search Recursive

print('Depth First Search(Recursion): Starting Point: 0 | End Point:', cells-1)

visited = [False for i in range(cells)]

prev = [-1 for i in range(cells)]

path(depthFirstSearchR(adL,0),cells-1)

print()

print('----------------------------------------------------------------')

# Searching path using Depth First Search (Stacks)

print('Depth First Search(Stack): Starting Point: 0 | End Point:', cells-1)

path(depthFirstSearchS(adL,0),cells-1)

print()

# Draws maze

plt.show()

# When the number of remove walls is equal to number of cells -1

if remWalls == cells-1:

print('\nThere is a unique path from source to destination\n')

vertices = []

# While we have more than 1 set

while remWalls != 0:

# Select a random list and unionize them

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

if union(S,walls[d][0],walls[d][1]) != False:

print('removing wall ',walls[d])

# remove them from lists

# add to lisd

vertices.append(walls[d])

walls.pop(d)

remWalls -= 1

# Final Disjoint Set, and maze drawing

print('\nDisjoint Set: ',S)

# Adjacent List

adL = adjList(vertices, cells)

print('\nAdjacent List: ',adL)

#Breadth\_First\_search

print('Breadth\_First\_search: Staring Point: 0 | End Point:', cells-1)

L = []

path(breadth\_First\_search(adL,0),cells-1)

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

print()

print('----------------------------------------------------------------')

# Depth First Search (Recursive)

print('Depth First Search(Recursion): Starting Point: 0 | End Point:', cells-1)

visited = [False for i in range(cells)]

prev = [-1 for i in range(cells)]

path(depthFirstSearchR(adL,0),cells-1)

print()

print('----------------------------------------------------------------')

# Depth First Search (Stacks)

print('Depth First Search(Stack): Starting Point: 0 | End Point:', cells-1)

path(depthFirstSearchS(adL,0),cells-1)

print()

plt.show()

# Number of remove walls is greater than cells-1

if remWalls > cells-1:

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

# if we have more walls to remove than the number of cells, set it to

# the max amount of cells

if remWalls > (maze\_rows\*(maze\_rows-1))+(maze\_cols\*(maze\_cols-1)):

remWalls = (maze\_rows\*(maze\_rows-1))+(maze\_cols\*(maze\_cols-1))

vertices = []

# Keeps track of items we have removed

count = 0

# removes walls and adds them to their disjoint set

while remWalls > 0:

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

# Select a random list and unionize them

if union(S,walls[d][0],walls[d][1]) is not False:

print('removing wall ',walls[d])

vertices.append(walls[d])

walls.pop(d)

remWalls -= 1

count += 1

# once we reach our n point we will break, I did this becasue the loop

# will go infinte times

if count == cells-1:

break

# finish up the remaining walls, and add them to their sets

while remWalls > 0:

print('-')

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

print('removing wall ',walls[d])

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

vertices.append(walls[d])

walls.pop(d)

remWalls -= 1

# Final Disjoint Set, Breadth\_First\_search, depthFirstSearchR, depthFirstSearchS,

print(vertices)

print('\nDisjoint Set: ',S)

adL = adjList(vertices, cells)

print('\nAdjacent List: ',adL)

print('Breadth\_First\_search: Staring Point: 0 | End Point:', cells-1)

L = []

path(breadth\_First\_search(adL,0),cells-1)

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

print()

print('----------------------------------------------------------------')

print('Depth First Search(Recursion): Starting Point: 0 | End Point:', cells-1)

visited = [False for i in range(cells)]

prev = [-1 for i in range(cells)]

L = []

path(depthFirstSearchR(adL,0),cells-1)

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

print()

print('----------------------------------------------------------------')

print('Depth First Search(Stack): Starting Point: 0 | End Point:', cells-1)

L = []

path(depthFirstSearchS(adL,0),cells-1)

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

print()

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

plt.show()

# union by size and compression

if choice == 2:

# number of cells

cells = maze\_rows \* maze\_cols

print('\nOur maze currently contains', cells, 'cells')

print('\nHow many walls should be removed?')

remWalls = int(input('Choice: '))

# if we remove negative number of cells

if remWalls < 0:

print('you cannot remove negative walls')

exit(0)

if remWalls < cells-1:

print('\nA path from source to destination is not guaranteed to exists\n')

# stores vertices we have linked ot removed

vertices = []

while remWalls != 0:

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

# unionize items from random list

if unionBySize(S,walls[d][0],walls[d][1]) != False:

# remove from list

print('removing wall ',walls[d])

vertices.append(walls[d])

walls.pop(d)

remWalls -= 1

# prints out information: Disjoint set, Breadth\_First\_search, depthFirstSearchR

# depthFirstSearchS, and draws maze

print('\nDisjoint Set: ',S)

adL = adjList(vertices, cells)

print('\nAdjacent List: ',adL)

print('Breadth\_First\_search: Staring Point: 0 | End Point:', cells-1)

L = []

path(breadth\_First\_search(adL,0),cells-1)

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

print()

print('----------------------------------------------------------------')

print('Depth First Search(Recursion): Starting Point: 0 | End Point:', cells-1)

visited = [False for i in range(cells)]

prev = [-1 for i in range(cells)]

path(depthFirstSearchR(adL,0),cells-1)

print()

print('----------------------------------------------------------------')

print('Depth First Search(Stack): Starting Point: 0 | End Point:', cells-1)

path(depthFirstSearchS(adL,0),cells-1)

print()

plt.show()

# exactly one path per vertices

if remWalls == cells-1:

print('\nThere is a unique path from source to destination\n')

vertices = []

# stores walls we have removed

while remWalls != 0:

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

# unionize items from random list

if unionBySize(S,walls[d][0],walls[d][1]) != False:

# remove from list

print('removing wall ',walls[d])

vertices.append(walls[d])

walls.pop(d)

remWalls -= 1

# prints out information: Disjoint set, Breadth\_First\_search, depthFirstSearchR

# depthFirstSearchS, and draws maze

print('\nDisjoint Set: ',S)

adL = adjList(vertices, cells)

print('\nAdjacent List: ',adL)

#graphs.draw\_graph(adL)

print('Breadth\_First\_search: Staring Point: 0 | End Point:', cells-1)

L = []

path(breadth\_First\_search(adL,0),cells-1)

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

print()

print('----------------------------------------------------------------')

print('Depth First Search(Recursion): Starting Point: 0 | End Point:', cells-1)

visited = [False for i in range(cells)]

prev = [-1 for i in range(cells)]

path(depthFirstSearchR(adL,0),cells-1)

print()

print('----------------------------------------------------------------')

print('Depth First Search(Stack): Starting Point: 0 | End Point:', cells-1)

path(depthFirstSearchS(adL,0),cells-1)

print()

plt.show()

if remWalls > cells-1:

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

# While we have more than 1 set

# if we remove more walls than the number of walls we have in our maze

if remWalls > (maze\_rows\*(maze\_rows-1))+(maze\_cols\*(maze\_cols-1)):

remWalls = (maze\_rows\*(maze\_rows-1))+(maze\_cols\*(maze\_cols-1))

vertices = []

# Stores removed walls

count = 0

# keeps count of when to break from loop

while remWalls != 0:

# Select a random list and unionize them

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

if unionBySize(S,walls[d][0],walls[d][1]) != False:

print('removing wall ',walls[d])

vertices.append(walls[d])

walls.pop(d)

remWalls -= 1

count += 1

# when we have n-1 walls removed, we will break from loop, otherwise

# we will loop infinte times

if count == cells -1:

break

# removes and adds remaining walls into their sets

while remWalls != 0:

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

print('removing wall ',walls[d])

unionBySize(S,walls[d][0],walls[d][1])

vertices.append(walls[d])

walls.pop(d)

remWalls -= 1

# prints out information: Disjoint set, Breadth\_First\_search, depthFirstSearchR

# depthFirstSearchS, and draws maze

print('\nDisjoint Set: ',S)

adL = adjList(vertices, cells)

print('\nAdjacent List: ',adL)

print('Breadth\_First\_search: Staring Point: 0 | End Point:', cells-1)

L = []

path(breadth\_First\_search(adL,0),cells-1)

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

print()

print('----------------------------------------------------------------')

print('Depth First Search(Recursion): Starting Point: 0 | End Point:', cells-1)

visited = [False for i in range(cells)]

prev = [-1 for i in range(cells)]

L = []

path(depthFirstSearchR(adL,0),cells-1)

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

print()

print('----------------------------------------------------------------')

print('Depth First Search(Stack): Starting Point: 0 | End Point:', cells-1)

L = []

path(depthFirstSearchS(adL,0),cells-1)

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

print()

plt.show()

else:

# User selects something else than 1 or 2

exit(0)