Laila Romero

CS 2302 Data Structures

May 1, 2019

Lab 7

* Introduction

For this lab we were trying to create a randomized maze through mathplotlib and a disjoint set forest. We needed to take user input to how many walls(m) they would want to remove and make sure that all the cells were in the same set/pointed to the same root every time it was randomized with a number equal to or greater than the number of cells-1 (n-1). After this we were to print out the adjacency list of the maze. Then we were to implement breadth first and depth first to solve a path for the maze we created. We also needed to compare the run times for randomizing the mazes with different sizes.

• Proposed Solution Design and Implementation

To get make sure the cells of the maze were always in the same set, I made a disjoint set forest measuring the size of the maze columns \* maze rows. Then I made a method to count the number of sets in the disjoint set forest called NumSets which took the disjoint set forest as a parameter. It then kept a counter called num which increased by 1 every time the index of a cell was less than -1 (indicating it belonged to a different set) as it traversed through the forest. At the end it returned the number of sets in the forest.

Then I made a method called MazeNorm which took the number of rows, columns, the disjoint set forest, and the wall list, and the choice of the user as parameters. It then checked if the choice was less than or equal to cells-1 then it would enter a for loop to check for the number of sets in the disjoint set forest, while it is greater than one set it creates a randomized integer called d. It then checks if the roots of the cells are equal to each other using the find method in the dsf code. If the roots are not, then it uses the union method from the dsf code to give the cells the same root. It then randomly pops a wall between the two cells. It repeats this method as many times as the user input as long as it met the condition of the if statement.

The second if statement in the method then checks if the choice of the user if larger than the available walls. If so, then an error message will print and a maze will not be made.

The last conditional statement then checks if the choice of the user is larger than cells-1. If so, then it enters a while loop of while the choice is greater than 0, then a randomized integer d is created to randomly pop walls in the maze. It repeats this method as many times as the user input as long as it met the condition of the if statement.

To return the adjacency list, I used the method of MazeNorm and returned it in side that method. To begin with, I created an empty array called G. I then went into the method and after every randomized pop of walls in the maze I stored those values in a variable called poppedWalls. I then appended the values in poppedWalls to the empty array of G and returned G at the end of the method.

For the breadth first search, I created a method called BFS which took G and v (the starting point) as parameters. I then created a boolean array called visited with values of “False”. I then create an array called prev of length of g (the adjacency list) and fill it with -1’s. Then I created an empty queue and enqueue the value of v into the queue. Then I change make the value of visited[v] to be True instead of False because that cell has now been visited. It then enters a while loop which states that while the q is not empty a variable u will dequeue the value in the queue. Then it goes into a for loop stating if a number is in the array of G at index u, then it is to enter an if statement to check if the value is visited in the the visited array. If not, then the value in the visited array changes to true and is added to the prev array. Once it traverses through the entire adjacency list (G) then the method returns prev.

The recursive depth first search is very similar to the breadth first search but instead of initializing the variables visited and prev inside the method, you initialize them outside in the main method and call for them as global variables.

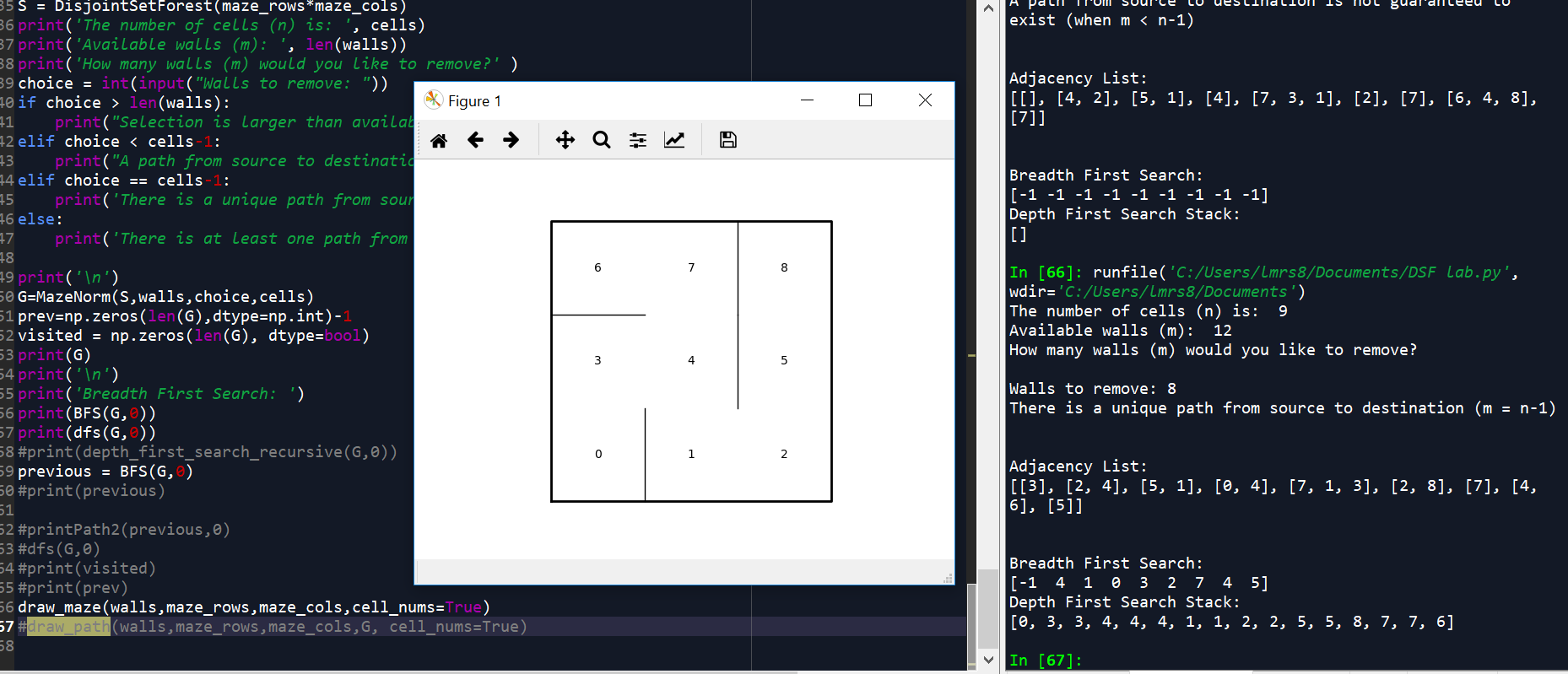
• Experimental results

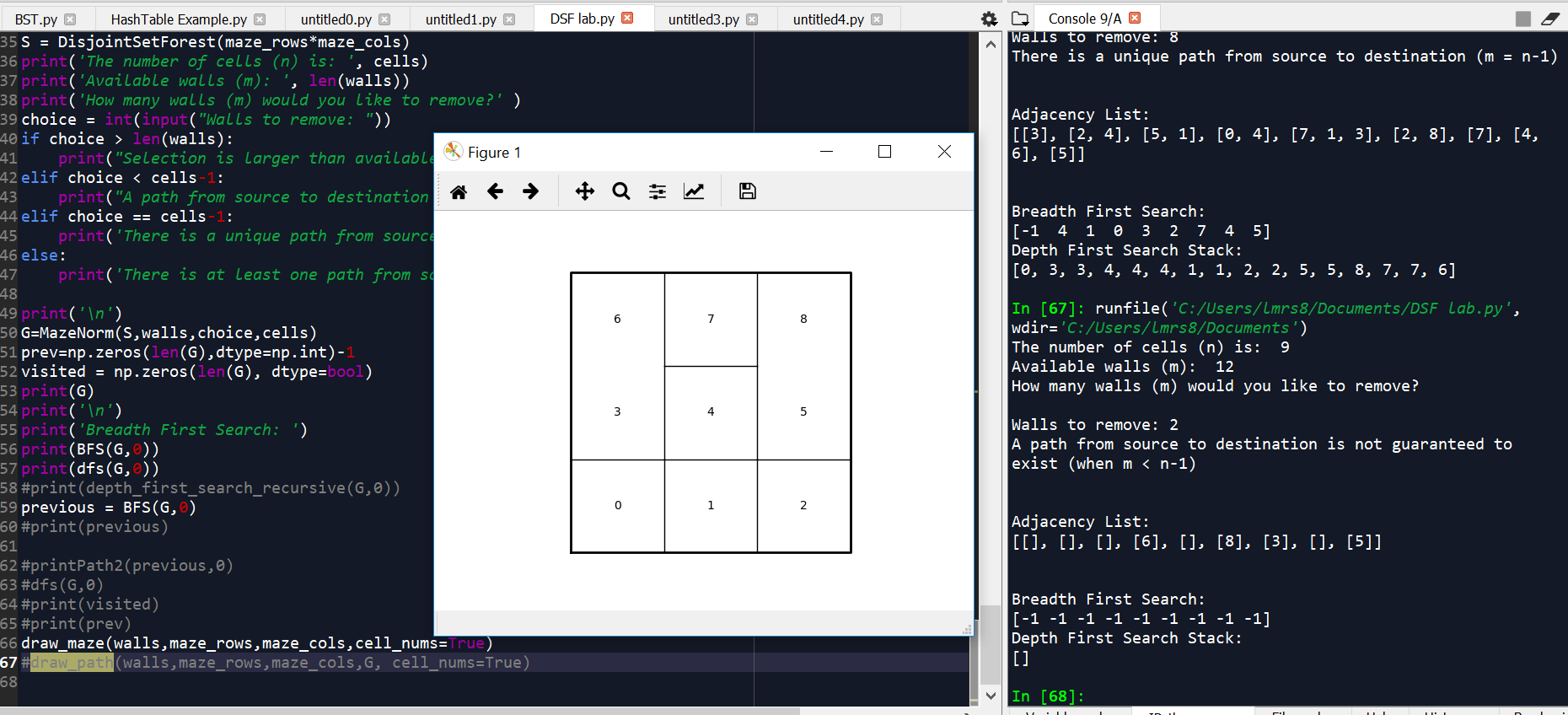
In order to test my program, I tried different sizes of mazes to make sure the method worked under all circumstances. I also compared the running times to make the same randomized maze with the MazeNorm and MazeComp and found out the MazeComp method was generally faster to produce the maze. The average times for each method to produce a 10 x 12 maze were:

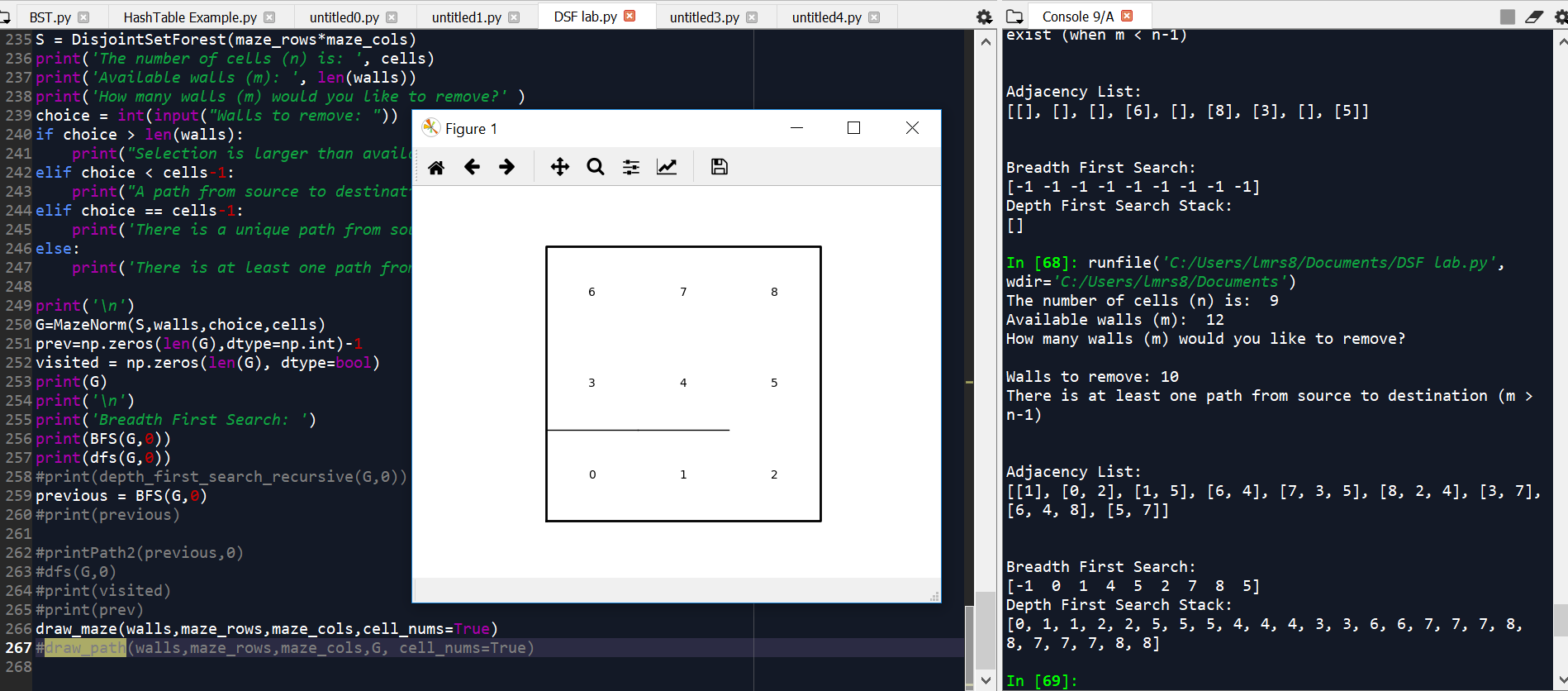
Running time for normal: 0.33629941940307617

Running time for compression: 0.28255653381347656

|  |  |
| --- | --- |
| **Method** | **Big O** |
| MazeNorm | O(n) |
| BFS | O(n) |
| Dfs | O(n) |
| dfsrec | O(n) |







• Conclusion

In conclusion, I learned how to create a maze with both union by size and union. I learned how to implement disjoint set forests and to make all the cells point to one root even when randomized. I also learned that the compression method ran faster than the regular method to create a maze.

• Appendix – Source codes

"""

Laila Romero

CS 2302

Professor Olac Fuentes

Lab 7

04/30/19

This code creates a randomized maze through a disjoint set forest and takes user

input to remove walls. It then generate the adjacency list of the maze

it also uses breadth first and depth first search to solve the maze

"""

import matplotlib.pyplot as plt

import numpy as np

import random

from scipy import interpolate

import queue

def BFS(G,v):

visited = np.zeros(len(G), dtype=bool)

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

q = queue.Queue()

q.put(v)

visited[v] = True

while not q.empty():

u = q.get()

for i in G[u]:

if not visited[i]:

visited[i]=True

prev[i]=u

q.put(i)

#print('Breadth First Solution: ')

return prev

def dfsrec(G, source):

global visited

global prev

visited[source]=True

for t in G[source]:

if not visited[t]:

prev[t]=source

dfs(G,t)

def dfs(G,v):

visited = []

stack = [v]

while stack != []:

current=stack.pop()

for i in G[current]:

if i not in visited:

stack.append(i)

visited.append(current)

print('Depth First Search Stack: ')

return visited

#def depth\_first\_search\_recursive(G, v, visited=None):

# if visited is None:

# visited = set()

# visited.add(v)

# for next in G[v] - visited:

# depth\_first\_search\_recursive(G, next, visited)

# return visited

def printPath(prev,v):

if prev[v] != -1:

printPath(prev,prev[v])

print('-')

print(v)

def printPath2(prev,v):

q = queue.Queue()

q.put(v)

while not q.empty():

u = q.get()

for i in prev[u]:

if prev[v] != -1:

prev[i]=u

q.put(i)

print('-')

#print('Breadth First Solution: ')

#def printPath(prev):

# path =[]

# for i in range(len(prev)-1):

# path[i]= prev[-1]

# path[i+1]=prev[prev[-1]]

# return path

def DisjointSetForest(size):

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

def draw\_dsf(S):

scale = 10000

fig, ax = plt.subplots()

for i in range(len(S)):

if S[i]<0: # i is a root

ax.plot([i\*scale,i\*scale],[0,scale],linewidth=1,color='k')

ax.plot([i\*scale-1,i\*scale,i\*scale+1],[scale-2,scale,scale-2],linewidth=1,color='k')

else:

x = np.linspace(i\*scale,S[i]\*scale)

x0 = np.linspace(i\*scale,S[i]\*scale,num=5)

diff = np.abs(S[i]-i)

if diff == 1: #i and S[i] are neighbors; draw straight line

y0 = [0,0,0,0,0]

else: #i and S[i] are not neighbors; draw arc

y0 = [0,-6\*diff,-8\*diff,-6\*diff,0]

f = interpolate.interp1d(x0, y0, kind='cubic')

y = f(x)

ax.plot(x,y,linewidth=1,color='k')

ax.plot([x0[2]+2\*np.sign(i-S[i]),x0[2],x0[2]+2\*np.sign(i-S[i])],[y0[2]-1,y0[2],y0[2]+1],linewidth=1,color='k')

ax.text(i\*scale,0, str(i), size=20,ha="center", va="center",

bbox=dict(facecolor='w',boxstyle="circle"))

ax.axis('off')

ax.set\_aspect(1.0)

def draw\_maze(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)

def draw\_path(walls,maze\_rows,maze\_cols,G, cell\_nums):

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")

for i in range(len(G)):

if G[i] != -1:

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 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 find(S,i):

# Returns root of tree that i belongs to

if S[i]<0:

return i

return find(S,S[i])

def union(S,i,j):

ri = find(S,i)

rj = find(S,j)

if ri!=rj:

S[rj] = ri

def NumSets(S):

num =0

for i in range(len(S)):

if S[i] <0:

num+=1

return num

def MazeNorm(S,walls,choice,cells):

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

if choice<=cells-1:

while NumSets(S)>1 and choice > 0:

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

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

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

poppedWalls=walls.pop(d)

choice -=1

G[poppedWalls[1]].append(poppedWalls[0])

G[poppedWalls[0]].append(poppedWalls[1])

elif choice> len(walls):

print("Not possible")

return None

elif choice > cells-1:

while choice>0:

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

poppedWalls=walls.pop(d)

G[poppedWalls[1]].append(poppedWalls[0])

G[poppedWalls[0]].append(poppedWalls[1])

choice -=1

print('Adjacency List: ')

return G

plt.close("all")

maze\_rows = 3

maze\_cols = 3

cells = maze\_rows\*maze\_cols

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

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

print('The number of cells (n) is: ', cells)

print('Available walls (m): ', len(walls))

print('How many walls (m) would you like to remove?' )

choice = int(input("Walls to remove: "))

if choice > len(walls):

print("Selection is larger than available walls")

elif choice < cells-1:

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

elif choice == cells-1:

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

else:

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

print('\n')

G=MazeNorm(S,walls,choice,cells)

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

visited = np.zeros(len(G), dtype=bool)

print(G)

print('\n')

print('Breadth First Search: ')

print(BFS(G,0))

print(dfs(G,0))

#print(depth\_first\_search\_recursive(G,0))

previous = BFS(G,0)

#print(previous)

#printPath2(previous,0)

#dfs(G,0)

#print(visited)

#print(prev)

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

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

“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.”

