Course: CS 2302

Author: Kimberly Morales

Assignment: Lab 7

Instructor: Olac Fuentes

TA(s): Anindita Nath, Maliheh Zargaran

Date: 4/30/19

# Introduction

To generate a path from the bottom right to the top left cell as a maze solution with the use of path finding algorithms and disjoint set forests based on the previous lab 6 work done to generate the maze. This requires the use for the shortest path’s algorithms: breadth first search and depth first search. To implement this is by drawing a graph and using it to draw a path by first creating an adjacency list. To see the overall visual solution requires gathering this path and then drawing a line at each cell text until it reaches the final cell.

# Proposed Solution Design and Implementation

My overall approach was to just follow Professor Fuente’s pseudocode and understand the inner functions of the original lab 6 maze code. Then I had to understand how the path was drawn and how to make an adjacency list. I made sure to do the path finding first then went on to the visual aspects. When thinking of the cases, I made sure to modify the original check maze method and to adjust the number of walls knocked down.

**Draw Methods**

Check\_maze\_uc and check\_maze

Both methods add in an additional parameter called case, which oversees directing to a specific if branch and to knock down walls so to not cause an error with going out of bounds.

Cases 1 and 2 are similar in knocking down walls and is mostly identical to the original method except that it uses m, the number of walls to knock down, as the loop controller. Each time that there is a union between two cells, m is subtracted instead of the number of sets.

In case 3, the number of walls to knock down and the number of sets is considered. There is also the concern of knocking down all the walls it it exceeds it so there is a quick check to see if m is greater than or equal to the number of walls. If it is, then we just return the cleared maze since all walls will be knocked down. If m is not greater than the number of walls then we follow similar code to cases 1 and 2 except that when there is a union between two cells then we subtract one from m and the sets, the number of sets calculated earlier with numsets method. If the number of sets eventually equal to 1 then the wall is deleted, and m is subtracted by 1. This is to avoid going over bound.

Draw\_path

This method is a modification of draw\_maze based on lab 6 but instead only draws the path from the bottom right to the top left cell or from 0 to number of cells-1. This method skill establishes a square and walls before drawing anything. Then a list is used to collect the coordinates used to put the text in each cell. Then we traverse through the path found from any of the path finding algorithms and plot at the index of the path and the previous point visited. As long as the index is not the starting point then the previous point is assigned the value of the index and then traverses to the next point in the path. The color of this path should be red and will not draw if no path is found.

Draw\_graph\_maze

Another method that modifies the original draw\_maze and draw\_graph method from the graph’s python file. While the original method did draw the maze connections correctly, I preferred it to show the original structure of the maze, so I created this method to show it accurately as possible. When trying to find out the pattern on how the coordinates are drawn, I realized that it was the row that was often changing more often, and the column number often kept a consistent number. This minor addition made it work and it keeps is accurate no matter if the rows or columns are greater than each other.

**Lab Methods**

Maze\_adjlist

In order to find a path, a graph is needed, but to generate a graph, an adjacency list is needed. Since the original lab 6 only created a list of walls, it was important to convert this list to an adjacency list. I used the original wall list before any deletions occurred and the wall list after deletions. It is important to traverse through both lists because you do not want to add in any deleted walls since the graph is connection open connections between two cells. To append adjacent connections to the current cell, it is important to add the first and second elements since these connections occur for both cells. My earlier approach did not have this in mind and so whenever a cell had more than two connections, it would not draw a path. Another approach that I did that was similar to the final version was to delete deleted walls from the original list based on the deleted walls list. However, Python has issues when deleting elements while also traversing through a list, so this approach changed to have two loops with not deletions occurring to either list.

Bfs

Breadth first search like all the other algorithms used in this program are path finding. Its behavior is to first visit the current root node and then look at each of its neighbors before going to other nodes. While an interesting algorithm to program, I mostly followed Professor Fuente’s pseudocode on how it works programmatically and translated it to python. It was tricky to find a way to create a queue in python but then I realized that stacks and queues share a lot of characteristics. I realized that all I needed to do to make a queue was to create a list and just use append to add to the queue and to pop the queue was to use the list’s pop at the first element only. The rest of the code is like the pseudocode and to explain it simply, it loops through the queue and pops the current node and traverses through the graph’s current node’s connections and see if it has visited it. If it has not, then it is set to true and adds the current node to the previous list (meaning it has been visited) and then adds the next node.

Dfs\_s

Depth first search works by first starting at a root node and then keeps traversing as far as possible and then backtracks. This method is depth first search with a stack and all I did was copy past the bfs method put changed q to s and made it pop the last element instead of the first element.

Dfs

Depth first search with recursion was tricky to mess with and requires the use of global variables. My initial method, while accurately following the original pseudocode, only returned one node. I knew this issue was the way that I was doing the recursive call and the use of no global variables. So, to fix the issue, I added in the visited and previous list as parameters since I was having trouble with the global variables. This kept track of changes and returned the list. On the way the method works, it traverses through the list based on the current node and see if a node has been visited and if it has not then it is added to the previous list and made true. Then the method is called recursively.

Gen\_path

This method was made for the draw\_path method and is based on the print\_path pseudocode except that it adds each element to the list rather than printing. It visits each node in the path and if the previous path at that node index is not empty then a recursive call is made and the previous list with the node index is returned in the call. At the end of the loop and recursion, the node is appended to the path list and is returned.

Main I/O

Main is in charge of the user input and displaying the maze, graph, and solved maze along with text results of the adjacency list, paths found for each of the path algorithms, and a message if a path is possible or not based on how big m or n is. N (the number of cells) is calculated from the user input of the product of the number of rows and number of columns. M (the number of walls to remove) is used for check\_maze methods and to find an appropriate message to display. Based on the length of the path, if the length is of 1 or less than that means that there was no path found so drawing the solution to the maze will not be drawn.

# Experimental Results

To find the running time for each path finding algorithm, I used the case that the number of walls knocked down is equal to the number of cells -1. While I did have the code for standard union and find, I used path compression so to not overstress my laptop and to keep consistent results. The number of rows and number of columns is equal so to make it a square maze and to keep a consistent total number.

Case: m = n-1 with Union by Size with Path Compression

Breadth First Search

|  |  |
| --- | --- |
| Input (N = Nrows X Ncols) | Time |
| Nrows = 5  Ncols = 5  N = 25 | 8.106699999643752e-05 |
| Nrows = 6  Ncols =6  N = 36 | 0.00012373300000945164 |
| Nrows = 7  Ncols = 7  N = 49 | 0.00012288000004900823 |
| Nrows = 8  Ncols = 8  N = 64 | 0.00011221399995520187 |

Depth First Search Recursion

|  |  |
| --- | --- |
| Input (N = Nrows X Ncols) | Time |
| Nrows = 5  Ncols = 5  N = 25 | 4.8639999988608906e-05 |
| Nrows = 6  Ncols =6  N = 36 | 5.12010000193186e-05 |
| Nrows = 7  Ncols = 7  N = 49 | 3.4987000049113703e-05 |
| Nrows = 8  Ncols = 8  N = 64 | 5.2906999940205424e-05 |

Depth First Search with Stack

|  |  |
| --- | --- |
| Input (N = Nrows X Ncols) | Time |
| Nrows = 5  Ncols = 5  N = 25 | 5.034699995576375e-05 |
| Nrows = 6  Ncols =6  N = 36 | 5.2479999908428e-05 |
| Nrows = 7  Ncols = 7  N = 49 | 3.669399995942513e-05 |
| Nrows = 8  Ncols = 8  N = 64 | 5.3759999900648836e-05 |

Screenshots

N=5

Maze Rows: 5

Maze Cols: 5

Num of walls to remove: 24

There is a unique path from source to destination

Adjacency List: [[1, 5], [0, 6], [7], [4, 8], [3], [0], [1, 7, 11], [2, 6], [3, 9, 13], [8], [15], [6, 12], [11, 13], [8, 12, 14, 18], [13, 19], [10, 20], [17, 21], [16, 18], [13, 17, 23], [14], [15, 21], [16, 20, 22], [21], [18, 24], [23]]

Breadth First Search:

[0, 1, 6, 11, 12, 13, 18, 23, 24]

Time: 8.106699999643752e-05

Depth First Search:

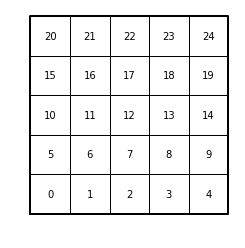
[0, 1, 6, 11, 12, 13, 18, 23, 24]

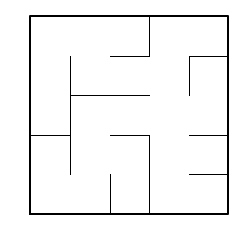
Time: 4.8639999988608906e-05

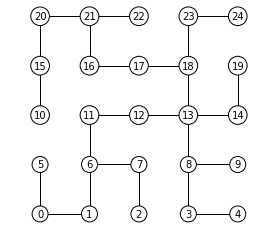
Depth First Search /w Stack:

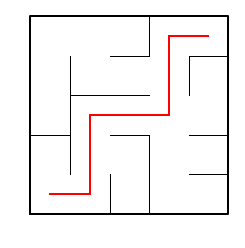
[0, 1, 6, 11, 12, 13, 18, 23, 24]

Time: 5.034699995576375e-05









In [**8**]:

N = 6

Maze Rows: 6

Maze Cols: 6

Num of walls to remove: 35

There is a unique path from source to destination

Adjacency List: [[1, 6], [0, 7], [3], [2, 4], [3, 5, 10], [4], [0], [1, 8, 13], [7, 9, 14], [8, 10, 15], [4, 9, 11, 16], [10, 17], [18], [7], [8, 20], [9], [10], [11], [12, 19], [18, 25], [14, 21], [20, 27], [23, 28], [22], [30], [19, 26, 31], [25, 32], [21, 28, 33], [22, 27, 29], [28], [24, 31], [25, 30], [26, 33], [27, 32, 34], [33, 35], [34]]

Breadth First Search:

[0, 1, 7, 8, 14, 20, 21, 27, 33, 34, 35]

Time: 0.00012373300000945164

Depth First Search:

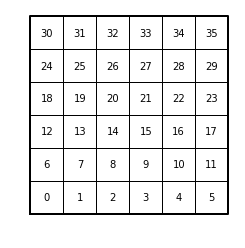
[0, 1, 7, 8, 14, 20, 21, 27, 33, 34, 35]

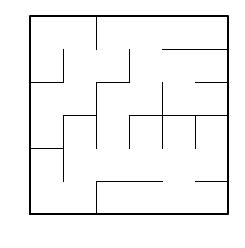
Time: 5.12010000193186e-05

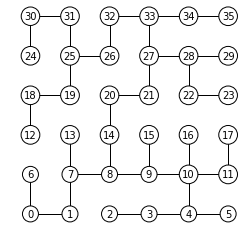
Depth First Search /w Stack:

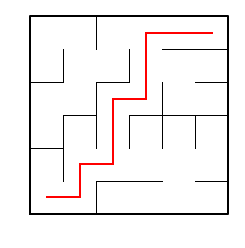
[0, 1, 7, 8, 14, 20, 21, 27, 33, 34, 35]

Time: 5.2479999908428e-05









In [**10**]:

N = 7

Maze Rows: 7

Maze Cols: 7

Num of walls to remove: 48

There is a unique path from source to destination

Adjacency List: [[7], [2, 8], [1], [4], [3, 5], [4, 6, 12], [5], [0, 8, 14], [1, 7, 9, 15], [8, 10], [9, 11, 17], [10, 18], [5, 13], [12, 20], [7, 21], [8, 16, 22], [15], [10], [11, 19, 25], [18], [13, 27], [14], [15, 23], [22, 30], [31], [18, 26], [25, 27, 33], [20, 26], [29], [28, 36], [23, 31, 37], [24, 30, 32, 38], [31], [26, 40], [41], [36], [29, 35, 37], [30, 36], [31, 39, 45], [38], [33, 41], [34, 40, 48], [43], [42, 44], [43, 45], [38, 44], [47], [46, 48], [41, 47]]

Breadth First Search:

[0, 7, 8, 9, 10, 11, 18, 25, 26, 33, 40, 41, 48]

Time: 0.00012288000004900823

Depth First Search:

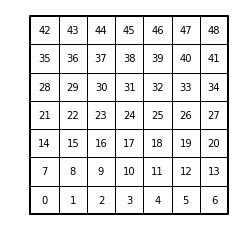
[0, 7, 8, 9, 10, 11, 18, 25, 26, 33, 40, 41, 48]

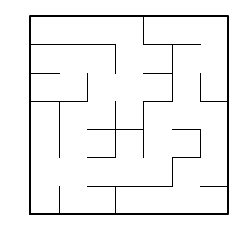
Time: 3.4987000049113703e-05

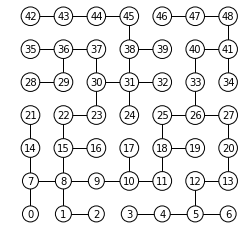
Depth First Search /w Stack:

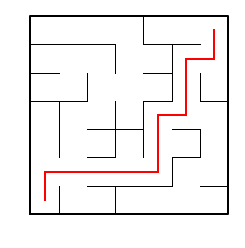
[0, 7, 8, 9, 10, 11, 18, 25, 26, 33, 40, 41, 48]

Time: 3.669399995942513e-05









In [**12**]:

N = 8

Maze Rows: 8

Maze Cols: 8

Num of walls to remove: 63

There is a unique path from source to destination

Adjacency List: [[1], [0, 2, 9], [1], [11], [12], [6], [5, 7, 14], [6], [16], [1, 10], [9, 18], [3, 12], [4, 11, 20], [14, 21], [6, 13, 15], [14, 23], [8, 24], [18, 25], [10, 17, 19], [18, 27], [12, 21, 28], [13, 20], [30], [15, 31], [16, 32], [17, 33], [27, 34], [19, 26, 28], [20, 27], [30, 37], [22, 29, 31], [23, 30, 39], [24, 33], [25, 32, 41], [26, 35, 42], [34, 36, 43], [35], [29, 45], [39], [31, 38], [41, 48], [33, 40], [34, 50], [35, 44], [43, 52], [37, 46], [45, 47], [46], [40, 56], [50, 57], [42, 49, 51], [50], [44, 53, 60], [52, 54, 61], [53, 55], [54, 63], [48], [49, 58], [57], [60], [52, 59], [53, 62], [61], [55]]

Breadth First Search:

[0, 1, 9, 10, 18, 19, 27, 26, 34, 35, 43, 44, 52, 53, 54, 55, 63]

Time: 0.00011221399995520187

Depth First Search:

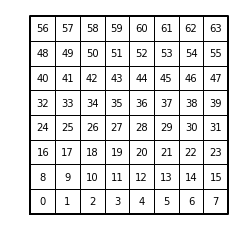
[0, 1, 9, 10, 18, 19, 27, 26, 34, 35, 43, 44, 52, 53, 54, 55, 63]

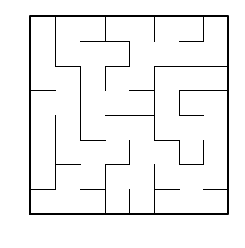
Time: 5.2906999940205424e-05

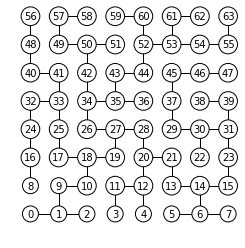
Depth First Search /w Stack:

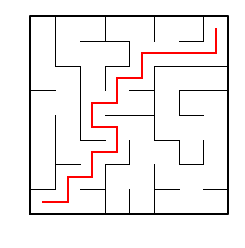
[0, 1, 9, 10, 18, 19, 27, 26, 34, 35, 43, 44, 52, 53, 54, 55, 63]

Time: 5.3759999900648836e-05









In [**14**]:

# Conclusions

Overall, I did a lot of the preliminary work from lab 6 and all I had to do was transfer this knowledge to a graph and knowing on how draw\_maze worked with it’s coordinates. Making the adjacency list was a difficult part for me since I did not realize that trying to delete elements in a list while also traversing through it would through off the original count of the loop and would skip over elements in the traversal. After fixing the error in this method, I was quick to just implement the rest and to test the several cases of drawing a maze. I enjoyed drawing the path in the maze since it looked fun to show how the path finding algorithm could be used to help solve hard puzzles.

# Appendix

"""

Course: CS 2302 [MW 1:30-2:50]

Author: Kimberly Morales

Assignment: Lab 7

Instructor: Olac Fuentes

TA(s): Anindita Nath , Maliheh Zargaran

Date: 4/28/2019

Date of last modification: 4/28/2019

Purpose of program:

To generate a path from the bottom right to the top left cell as a maze solution.

This requires the use for the shortest paths algorithms: breadth first search and depth first search.

To implement this is by drawing a graph and using it to draw a path and know how to use an adjacency list.

"""

import matplotlib.pyplot as plt

from scipy import interpolate

import numpy as np

import random

import timeit

import math

import dsf

import graphs

from collections import deque

#################################################################################################

#MAZE METHODS

#################################################################################################

#Checks to see if two cells belong to different sets and removes a wall

#Uses standard union

def check\_maze(S,w,mc,mr,m,case):

if case == 1 or case == 2:

while m != 0:

dw = random.randint(0,len(w)-1) #dw: wall to remove

c1 = dsf.find\_c(S,w[dw][0]) #c1: cell 1

c2 = dsf.find\_c(S,w[dw][1]) #c2: cell 2

#If the two cells are from different cells than remove a wall so to allow a path and combine them

if c1 != c2:

del w[dw]

dsf.union\_by\_size(S,c1,c2)

m -= 1

return w

else:

sets = num\_sets(S)

if m == len(w) or m > len(w):

return w.clear()

while m != 0:

dw = random.randint(0,len(w)-1)

c1 = dsf.find\_c(S,w[dw][0])

c2 = dsf.find\_c(S,w[dw][1])

if c1 != c2:

del w[dw]

dsf.union\_by\_size(S,c1,c2)

m -= 1

sets -= 1

if sets == 1:

del w[dw]

m -= 1

return w

#Checks to see if each cell has a simple path to another cell

#Uses union by size and path compression

def check\_maze\_uc(S,w,mc,mr,m,case):

if case == 1 or case == 2:

while m != 0:

dw = random.randint(0,len(w)-1) #dw: wall to remove

c1 = dsf.find\_c(S,w[dw][0]) #c1: cell 1

c2 = dsf.find\_c(S,w[dw][1]) #c2: cell 2

#If the two cells are from different cells than remove a wall so to allow a path and combine them

if c1 != c2:

del w[dw]

dsf.union\_by\_size(S,c1,c2)

m -= 1

return w

else:

#Checks sets and number of walls to delete

sets = num\_sets(S)

#If there are more m than walls than clear the maze

if m == len(w) or m > len(w):

return w.clear()

while m != 0:

dw = random.randint(0,len(w)-1)

c1 = dsf.find\_c(S,w[dw][0])

c2 = dsf.find\_c(S,w[dw][1])

if c1 != c2:

del w[dw]

dsf.union\_by\_size(S,c1,c2)

m -= 1

sets -= 1

#If there is one more set then delete the last wall

if sets == 1:

del w[dw]

m -= 1

return w

#Counts the total number of sets

def num\_sets(S):

count = 0

#Checks each set to see if they have similar parents

for i in range(len(S)):

ri = dsf.find(S,i)

#If the element equal to the root then it is in the same set

if i == ri:

count+=1

return count

#################################################################################################

#DRAW METHODS

#################################################################################################

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)

#Draws shortest path based on the path algorithms

def draw\_path(v, maze\_rows,maze\_cols):

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

ct = []

ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')

#Add in coordinates used for the text

for r in range(maze\_rows):

for c in range(maze\_cols):

cell = c + r\*maze\_cols

ct.append([c+.5,r+.5])

p = 0 #p: previous coordinate traveled to

#Traverse through each point in the path

for i in v:

ax.plot((ct[p][0],ct[i][0]),(ct[p][1],ct[i][1]), linewidth=2,color='r')

#So that the path does not repeat and to point previous to the already plotted

if i != 0:

p = i

ax.axis('off')

ax.set\_aspect(1.0)

#Draws graphical representation of the maze based on the structure

def draw\_graph\_maze(G,r,c):

fig, ax = plt.subplots()

n = len(G)

r = 30

coords = []

#These are in charge of maintaining the graph structure

numr = 0 #Row num

numc = 0 #Col num

for i in range(r):

for j in range(c):

coords.append([numr,numc])

numr += 1

numr = 0

numc += 1

for i in range(n):

for dest in G[i]:

ax.plot([coords[i][0],coords[dest][0]],[coords[i][1],coords[dest][1]],

linewidth=1,color='k')

for i in range(n):

ax.text(coords[i][0],coords[i][1],str(i), size=10,ha="center", va="center",

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

ax.set\_aspect(1.0)

ax.axis('off')

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

#################################################################################################

#LAB METHODS

#################################################################################################

def maze\_adjlist(wo,w,cells):

#Create empty list of size of cells so to generate a graph

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

#Traverse through original list with no deleted walls

for g in wo:

#Check if the current wall was not deleted and if not then add it to the graph

if g not in w:

#Add adjacent connections

a[g[0]].append(g[1])

a[g[1]].append(g[0])

return a

#Breadth first search

def bfs(G,v):

q = []

q.append(v)

visited[v] = True

while(len(q) is not 0):

u = q.pop(0) #Pop the first element

for t in G[u]:

if(visited[t] is False):

visited[t] = True

prev[t] = u

q.append(t)

return prev

#Depth first search with a stack

#Very similar to bfs but replaced with a stack

def dfs\_s(G, v):

s = []

s.append(v)

visited[v] = True

while len(s) != 0:

u = s.pop() #Only pop the back element

for t in G[u]:

if(visited[t] is False):

visited[t] = True

prev[t] = u

s.append(t)

return prev

#Depth first search with recursion

#Uses global variables so that it doesn't forget

def dfs\_rec(G, v, visited,prev):

visited[v] = True

for t in G[v]:

if (visited[t] is False):

prev[t] = v

dfs\_rec(G,t,visited,prev)

return prev

#Prints path traversal

def print\_path(prev, v):

if prev[v] != -1:

print\_path(prev,prev[v])

print('-',end='')

print(v,end='')

#Uses print path code but instead returns an array

def gen\_path(prev, v, path):

if prev[v] != -1:

gen\_path(prev,prev[v],path)

path.append(v)

return path

#################################################################################################

#MAIN

#################################################################################################

if \_\_name\_\_ == "\_\_main\_\_":

plt.close("all")

maze\_size = 0 #maze\_size: Size of maze (rows \* cols)

maze\_rows = int(input('Maze Rows: '))

maze\_cols = int(input('Maze Cols: '))

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

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

n = maze\_rows \* maze\_cols #num of cells n

m = int(input('Num of walls to remove: ')) #num of walls to remove m

#Cases so to appropriately draw the maze

if m < (n-1):

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

case = 1

elif m == (n-1):

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

case = 2

else:

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

case = 3

#Draws initial maze with numbers

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

#Create dsf with the size of the maze(rows \* cols)

S = dsf.DisjointSetForest(n)

#Create valid maze based on the previous case

check\_maze\_uc(S,walls,maze\_cols,maze\_rows,m,case)

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

#Make an adjacency list

G = maze\_adjlist(walls\_org,walls,n)

print('Adjacency List: ',G)

#Draw the grapical representation of the maze

draw\_graph\_maze(G,maze\_rows,maze\_cols)

#Create global variables for the algorithms

global visited

global prev

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

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

#BFS

print('Breadth First Search: ')

global v

v = gen\_path(bfs(G,0),n-1,[])

print(v)

#DFS REC

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

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

print('\nDepth First Search: ')

v = gen\_path(dfs\_rec(G,0,visited,prev),n-1,[])

print(v)

#DFS

print('\nDepth First Search /w Stack: ')

v = gen\_path(dfs\_s(G,0),n-1,[])

print(v)

#Draw the path

if len(v) != 1:

draw\_path(v,maze\_rows,maze\_cols)

# Academic Honesty

“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.”

Name: Kimberly Morales



Signature:

