Lab 7: Graphs

Manuel Ruvalcaba

CS2302 1:30-2:50

1. Introduction

The purpose of this program is to build a maze using a disjoint set forest with *n* cells, to account for the removal of less than *n*-1, exactly *n*-1, and more than *n-1* walls. Then we have to convert it into a graph and find the solution to the maze using various different searching algorithms.

2. Proposed Solution Design and Implementation

In order to build the mazes, I was provided two methods named *draw_maze* and *wall_list*. The *draw_maze* method draws the maze that is created in the program. The *wall_list* method creates and returns a list of walls that are in the maze.

I also created various methods named *DisjointSetForest*, *find*, *find_c*, *union*, and *union_by_size*.

The method named *DisjointSetForest* accepts an integer as the size for the disjoint set forest, creating and returning a numpy array of zeros of the given size. In this case, the size would be the maze rows multiplied by the maze columns.

The *find* method accepts an integer and a disjoint set forest, finding and returning the root of the forest containing the integer.

The *find_c* method accepts an integer and a disjoint set forest, finding and returning the root of the forest containing the integer, while compressing the path between the integer and the root.

The *union* method accepts a disjoint set forest and two integers, then finds their roots. If the roots are the same, it does nothing. Otherwise, it unifies the trees.

The *union_by_size* method accepts a disjoint set forest and two integers, then finds their roots. If the roots are the same, it does nothing. Otherwise, it makes the root of the smaller tree point to the root of the larger tree while compressing the path.

Another method I implemented is called *buildMazeSU*. What this method does, is it creates the maze using standard union, removing the amount of walls specified.

Another method I implemented is called *buildMazeUBS*. What this method does, is it creates the maze using union by size, removing the amount of walls specified.

I also have a method called *buildAdjList,* which accepts *walls,* a *maze,* and *maze_cols*. This method builds the adjacency list of the maze to be represented as a graph.

I have a method named *breadth_first_search* that accepts a graph represented by an adjacency list, and the starting index. This method then finds every path from a starting cell to an ending cell.

I have a method named *depth_first_search* that accepts a graph represented by an adjacency list, and the starting index. This method then finds every path from a starting cell to an ending cell.

I have a method named *depth_first_searchI* that works iteratively compared to the previous method and accepts a graph represented by an adjacency list, and the starting index. This method then finds every path from a starting cell to an ending cell.

I also have a method named *printPath* that accepts the *prev* list, given to us by the above methods. This method finds the path from the

first cell to the ending cell and puts it in a suitable format for drawing.

I also have a method named *draw_path* that accepts *walls, maze_rows, maze_cols,* and a *path*. This method draws a maze and the path in the maze.

3. Experimental Results

In order to test my code, I used mazes of different sizes. I then did tests involving mazes where a path was not guaranteed, a unique path was guaranteed, and at least one path existed.

My first maze size was 15 columns by 15 rows. I then did a test removing walls that would result in a path not guaranteed to exist. If the path doesn't exist, it would print, "There is no path." After that, I did a test removing walls to create a unique path, then finally a test removing walls that would result in at least one path.

```
IPython console
Console 1/A
There are 225 cells

How many walls should be removed? 200
A path from source to destination is not guaranteed to exist

Building Maze.
Maze Built.
Using Breadth First Search to find solution.
Time in microseconds to find solution:  59.11599987484806
There is no path

Building Maze.
Maze Built.
Using Depth First Search to find solution.
Time in microseconds to find solution:  8.224999874073546
There is no path

Building Maze.
Maze Built.
Using Iterative Depth First Search to find solution.
Time in microseconds to find solution:  123.88799996188027
There is no path
```

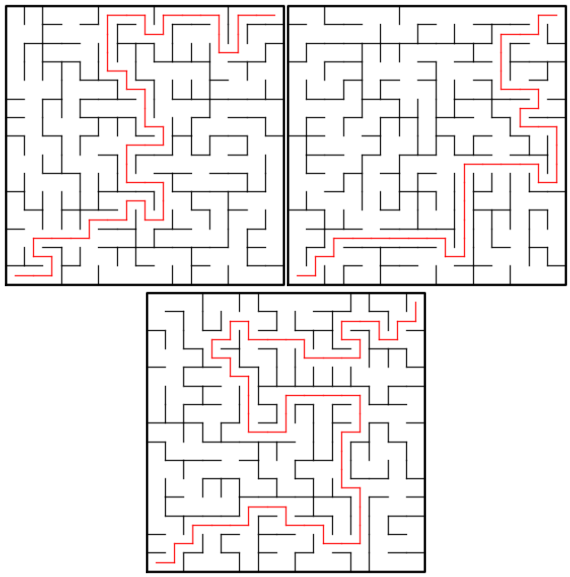
```
IPython console
Console 1/A
There are 225 cells

How many walls should be removed? 224
There is a unique path from source to destination

Building Maze.
Maze Built.
Using Breadth First Search to find solution.
Time in microseconds to find solution:  559.2939996859059

Building Maze.
Maze Built.
Using Depth First Search to find solution.
Time in microseconds to find solution:  359.8389998842322

Building Maze.
Maze Built.
Using Iterative Depth First Search to find solution.
Time in microseconds to find solution:  675.4710002496722
```



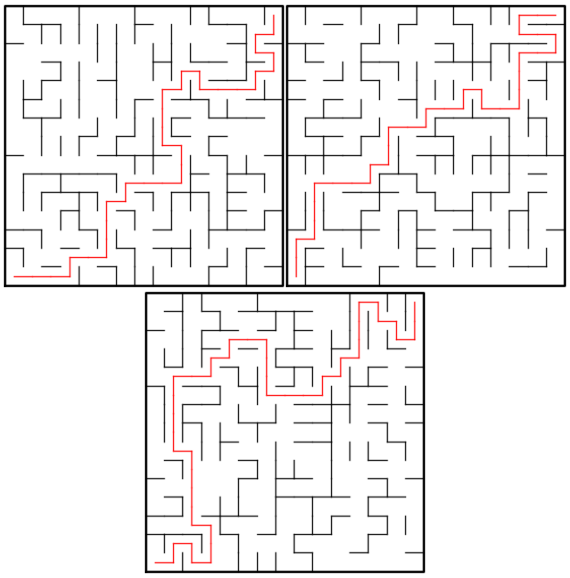
```
IPython console
Console 1/A
There are 225 cells

How many walls should be removed? 250
There is at least one path from source to destination

Building Maze.
Maze Built.
Using Breadth First Search to find solution.
Time in microseconds to find solution:  389.65400017332286

Building Maze.
Maze Built.
Using Depth First Search to find solution.
Time in microseconds to find solution:  331.56700010295026

Building Maze.
Maze Built.
Using Iterative Depth First Search to find solution.
Time in microseconds to find solution:  648.2250000772183
```



	Time in Microseconds to Find Path		
	Breadth First Search	Depth First Search Rec	Depth First Search Iterative
Path not guaranteed	59.115	8.224	123.888
Unique path	559.293	359.838	675.471
At least one path	389.654	331.567	648.225

My first maze size was 20 columns by 20 rows. I then did a test removing walls that would result in a path not guaranteed to exist. If the path doesn't exist, it would print, "There is no path." After that, I did a test removing walls to create a unique path, then finally a test removing walls that would result in at least one path.

IPython console

Console 1/A

```
There are 400 cells

How many walls should be removed? 350
A path from source to destination is not guaranteed to exist

Building Maze.
Maze Built.
Using Breadth First Search to find solution.
Time in microseconds to find solution:  101.26899996976135
There is no path

Building Maze.
Maze Built.
Using Depth First Search to find solution.
Time in microseconds to find solution:  16.96399976935936
There is no path

Building Maze.
Maze Built.
Using Iterative Depth First Search to find solution.
Time in microseconds to find solution:  182.49000004288973
There is no path
```

IPython console

Console 1/A

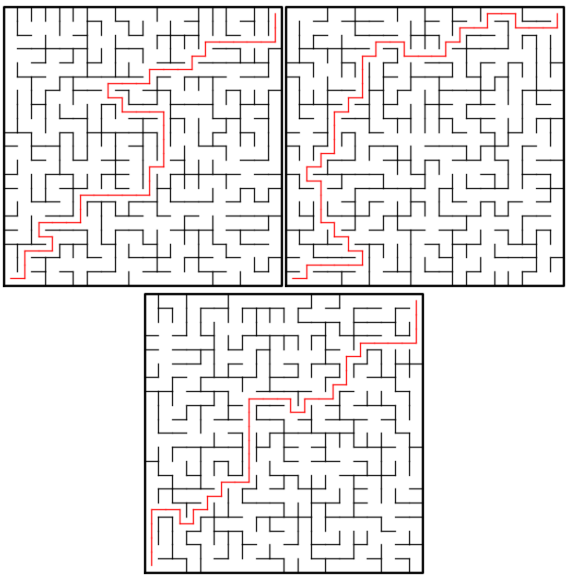
```
There are 400 cells

How many walls should be removed? 399
There is a unique path from source to destination

Building Maze.
Maze Built.
Using Breadth First Search to find solution.
Time in microseconds to find solution:  761.3179996042163

Building Maze.
Maze Built.
Using Depth First Search to find solution.
Time in microseconds to find solution:  414.8430002715031

Building Maze.
Maze Built.
Using Iterative Depth First Search to find solution.
Time in microseconds to find solution:  1438.3310003722727
```



IPython console

Console 1/A

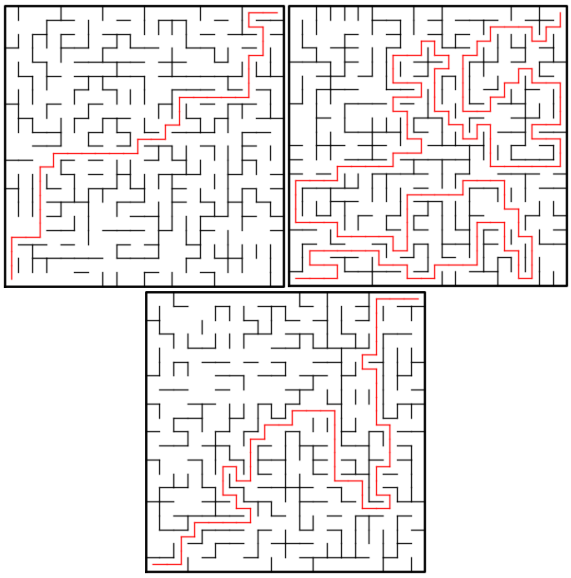
```
There are 400 cells

How many walls should be removed? 450
There is at least one path from source to destination

Building Maze.
Maze Built.
Using Breadth First Search to find solution.
Time in microseconds to find solution:  687.8080002934439

Building Maze.
Maze Built.
Using Depth First Search to find solution.
Time in microseconds to find solution:  767.4869998481881

Building Maze.
Maze Built.
Using Iterative Depth First Search to find solution.
Time in microseconds to find solution:  833.286000215594
```



	Time in Microseconds to Find Path		
	Breadth First Search	Depth First Search Rec	Depth First Search Iterative
Path not guaranteed	101.268	16.963	182.49
Unique path	761.317	414.843	1438.331
At least one path	687.808	767.486	833.286

It seems that the recursive version of depth first search, while not always providing the most direct path, was the fastest at finding a path.

4. Conclusion

With this project, I have learned how to find the solution to a maze using a graph. I guess I can create a website that makes mazes of a size requested by the user now.

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.

5. Appendix

```
'''
Course: CS2302 MW 1:30-2:50
Author: Manuel A. Ruvalcaba
Assignment: Lab #7: Graphs
Instructor: Dr. Olac Fuentes
TA: Anindita Nath, Maliheh Zargaran
Date of Last Modification: April 29, 2019
Purpose of the Program: The purpose of this program is to build a maze using a
                        disjoint set forest and then using graph representations
                        to find the solution to the maze with three different
                        searching algorithms(Breadth first search, depth first
                        search and an iterative version of depth first search).
'''
import matplotlib.pyplot as plt
import numpy as np
import random
import timeit
def draw_maze(walls,maze_rows,maze_cols,cell_nums=False):
    #draws the 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)

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 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):
    # Returns root of tree that i belongs to
    if S[i]<0:
        return i
    root = find_c(S,S[i])
    S[i] = root
    return root


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):
    # Joins i's tree and j's tree, if they are different
    ri = find_c(S,i)
    rj = find_c(S,j)
    if ri!=rj: # Do nothing if i and j belong to the same set
        if S[rj] < S[ri]:
            S[rj]+=S[ri]
            S[ri] = rj # Make i's root point to j's root
        else:
            S[ri]+=S[rj]
            S[rj] = ri # Make j's root point to i's root


def buildMazeSU(S,walls,remove):
    #builds the maze using standard union
    sets = len(S)-1
    j = 0
    while j < sets and j < remove: #removes walls until the amount of walls to be
removed is achieved
```

```
        d = random.randint(0,len(walls)-1)
        a = find(S,walls[d][0])
        b = find(S,walls[d][1])
        if a != b: #checks if the two cells are not part of the same set
            j += 1
            union(S,walls[d][0],walls[d][1])
            walls.pop(d)
    while j >= sets and j < remove: #removes the walls after the perfect maze has
been built
        d = random.randint(0,len(walls)-1)
        a = find(S,walls[d][0])
        b = find(S,walls[d][1])
        j += 1
        union(S,walls[d][0],walls[d][1])
        walls.pop(d)
    print(j)


def buildMazeUBS(S,walls,remove):
    #builds the maze using union by size with path compression
    sets = len(S)-1
    j = 0
    while j < sets and j < remove: #removes walls until there is only one set using
union by size and path compression
        d = random.randint(0,len(walls)-1)
        a = find_c(S,walls[d][0])
        b = find_c(S,walls[d][1])
        if a != b: #checks if the two cells are not part of the same set
            j += 1
            union_by_size(S,walls[d][0],walls[d][1])
            walls.pop(d)
    while j >= sets and j < remove: #removes the walls after the perfect maze has
been built
        d = random.randint(0,len(walls)-1)
        a = find_c(S,walls[d][0])
```

```
        b = find_c(S,walls[d][1])
        j += 1
        union_by_size(S,walls[d][0],walls[d][1])
        walls.pop(d)

def buildAdjList(walls,S,b):
    #builds the adjacency list of the maze to be a graph
    G =  [ [] for j in range(len(S)) ]
    for i in range(len(S)-1):
        if len(S)-i > maze_cols and i % maze_cols != maze_cols-1:
            if [i,i+1] not in walls:
                G[i].append(i+1)
                G[i+1].append(i)
            if [i,i+b] not in walls:
                G[i].append(i+maze_cols)
                G[i+maze_cols].append(i)
        elif len(S)-i > maze_cols:
            if [i,i+b] not in walls:
                G[i].append(i+maze_cols)
                G[i+maze_cols].append(i)
        elif i % maze_cols != maze_cols-1:
            if [i,i+1] not in walls:
                G[i].append(i+1)
                G[i+1].append(i)
    return G

def breadth_first_search(G,v):
    #Finds different paths using breadth first search
    visited = [False for j in range(len(G))]
    prev = [-1 for i in range(len(G))]
    Q = []
    Q.append(v)
    visited[v] = True
```

```
    while len(Q) is not 0:
        u = Q[0]
        Q.pop(0)
        for t in G[u]:
            if visited[t] == False:
                visited[t] = True
                prev[t] = u
                Q.append(t)
    return prev

def depth_first_search(G,source):
    #Finds different paths using depth first search recursively
    global visited
    global prev
    visited[source] = True
    for t in G[source]:
        if visited[t] == False:
            prev[t] = source
            depth_first_search(G,t)

def depth_first_searchI(G,v):
    #Finds different paths using depth first search iteratively
    visited = [False for j in range(len(G))]
    prev = [-1 for i in range(len(G))]
    Q = []
    Q.append(v)
    visited[v] = True
    while len(Q) is not 0:
        u = Q[len(Q)-1]
        Q.pop(len(Q)-1)
        for t in G[u]:
            if visited[t] == False:
                visited[t] = True
```

```
                prev[t] = u
                Q.append(t)
    return prev

def getPath(prev, v):
    #gets the path to a vertex v
    a = []
    return printPath(prev,v,a)

def printPath(prev,v,a):
    #finds the path to a vertex v and appends it to a list
    if prev[v] is not -1:
        printPath(prev,prev[v],a)
    a.append(v)
    return a

def draw_path(walls,maze_rows,maze_cols,path):
    #draws the path that is recieved as a parameter
    fig, ax = plt.subplots()
    for w in walls:
        if w[1]-w[0] ==1:
            x0 = (w[1]%maze_cols)
            x1 = x0
            y0 = (w[1]//maze_cols)
            y1 = y0+1
        else:
            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')
    for a in range(len(path)-1):
        if path[a+1]-path[a]== maze_cols:
            x0 = (path[a]%maze_cols)+.5
            x1 = x0
            y0 = (path[a+1]//maze_cols)-.5
            y1 = y0+1
        elif path[a+1]-path[a]== -maze_cols:
            x0 = (path[a+1]%maze_cols)+.5
            x1 = x0
            y0 = (path[a]//maze_cols)-.5
            y1 = y0+1
        elif path[a+1]-path[a] == -1:
            x0 = (path[a+1]%maze_cols)+.5
            x1 = x0+1
            y0 = (path[a]//maze_cols)+.5
            y1 = y0
        else:
            x0 = (path[a]%maze_cols)+.5
            x1 = x0+1
            y0 = (path[a+1]//maze_cols)+.5
            y1 = y0
        ax.plot([x0,x1],[y0,y1],linewidth=1,color='red')
    ax.axis('off')
    ax.set_aspect(1.0)

#This is where you test the code

plt.close("all")
maze_rows = 20
maze_cols = 20
S = DisjointSetForest(maze_rows*maze_cols)
print("There are",len(S),"cells")
```

```
choice = int(input("How many walls should be removed? "))
if choice < len(S)-1:
    print("A path from source to destination is not guaranteed to exist")
elif choice == len(S)-1:
    print("There is a unique path from source to destination")
else:
    print("There is at least one path from source to destination")
print()
remove = choice
walls = wall_list(maze_rows,maze_cols)

draw_maze(walls,maze_rows,maze_cols,cell_nums=True)
print('Building Maze.')
buildMazeUBS(S,walls,remove)
print('Maze Built.')
draw_maze(walls,maze_rows,maze_cols)
G = buildAdjList(walls,S,maze_cols)
print('Using Breadth First Search to find solution.')
start = timeit.default_timer()
path = breadth_first_search(G,0)
stop = timeit.default_timer()
print('Time in microseconds to find solution: ', (stop - start)*1000000)
pathCells = getPath(path,len(G)-1)
if pathCells[0] == len(G)-1:
    print("There is no path")
else:
    draw_path(walls,maze_rows,maze_cols, pathCells)
print()

S = DisjointSetForest(maze_rows*maze_cols)
remove = choice
walls = wall_list(maze_rows,maze_cols)
```

```
print('Building Maze.')
buildMazeUBS(S,walls,remove)
print('Maze Built.')
draw_maze(walls,maze_rows,maze_cols)
G = buildAdjList(walls,S,maze_cols)
visited = [False for j in range(len(G))]
prev = [-1 for i in range(len(G))]
print('Using Depth First Search to find solution.')
start = timeit.default_timer()
depth_first_search(G,0)
stop = timeit.default_timer()
print('Time in microseconds to find solution: ', (stop - start)*1000000)
pathCells = getPath(prev,len(G)-1)
if pathCells[0] == len(G)-1:
    print("There is no path")
else:
    draw_path(walls,maze_rows,maze_cols, pathCells)
print()

S = DisjointSetForest(maze_rows*maze_cols)
remove = choice
walls = wall_list(maze_rows,maze_cols)

print('Building Maze.')
buildMazeUBS(S,walls,remove)
print('Maze Built.')
draw_maze(walls,maze_rows,maze_cols)
G = buildAdjList(walls,S,maze_cols)
print('Using Iterative Depth First Search to find solution.')
start = timeit.default_timer()
path = depth_first_searchI(G,0)
stop = timeit.default_timer()
print('Time in microseconds to find solution: ', (stop - start)*1000000)
```

```
pathCells = getPath(path,len(G)-1)
if pathCells[0] == len(G)-1:
    print("There is no path")
else:
    draw_path(walls,maze_rows,maze_cols, pathCells)
```