CS2302 - Data Structures

Spring 2019

Lab No.7 Report

Introduction

This lab will cover the use of Graphs and the different methods that can be used to traverse them such as Depth First Search and Breadth First Search.

In this project we will make use of stacks and queues to go through a maze, creating an algorithm that solves it. Concepts such as adjacency lists and edge lists will also be used when creating a method that solves the puzzle.

Design & Implementation

To create a randomly generated maze, DSF is the optimal data structure. The maze first starts as a grid of cells. Then we need to remove a wall from each cell until each cell is connected by one unique path.

As in the past lab the concept is the same, but we also need to take into account that the user could ask for a maze with more than one path. To do this, first a unique path must be established. After this has been done then any wall can be removed.

90	91	92	93	94	95	96	97	98	99
80	81	82	83	84	85	86	87	88	89
70	71	72	73	74	75	76	77	78	79
60	61	62	63	64	65	66	67	68	69
50	51	52	53	54	55	56	57	58	59
40	41	42	43	44	45	46	47	48	49
30	31	32	33	34	35	36	37	38	39
20	21	22	23	24	25	26	27	28	29
10	11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9

Figure 1: Starting grid for the maze.

$elif\ path$:

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

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

$counter\ +=\ 1$

Snippet of code showing what happens after a path has been established. We unite the DSF since we Will use it for the walls in the maze. We also add the poped wall into our edje list that Will be used later on.

Edge List to Adjacency List:

For this method we will receive an edge list (EL) and convert it into an adjacency list (AL). Simple enough every list inside the edge list is composed of 2 items. The edge list is ordered in the following manner:

$$[[0,1][1,2][2,3]]$$

To transform it into an adjacency list we just need to create a new list of lists with the length of the biggest number. Then we populate this list by inserting the first number in the EL into the index of the LA of the other number and vice versa. Giving us in the end:

$$[[1][0,2][2]]$$

Breadth First Search:

For this method we will use a queue. Each time we visit a node then we put that node into the queue and mark it as visited, so that we not add it again to the queue. We also need a prev array, in which the previous neighbor for the vertex represented by the index can be found. We continue adding the neighbors of the number into the queue. When we mark all the neighbors we then pop the queue for the next vertex from which we will take neighbors.

Once the method has found the prev value for the cell at $rows * columns - 1$ then we stop the loop we are using and return the array.

Depth First Search:

We will do the same as with Depth first search but instead of a queue we will use a stack. Once the prev element of the same cell has been found then we return the prev array.

Drawing the path:

To be able to draw the path from the vertex at the desired corners we need to use the prev array. Using the prev array we will need to create an edge list that starts in the last element and continues entering the number from the next value in the prev array.

After we have the new edge list of the path. It will tells us the path from finish to start by going into the prev values of each vertex, starting with the last one since it is were we want to go.

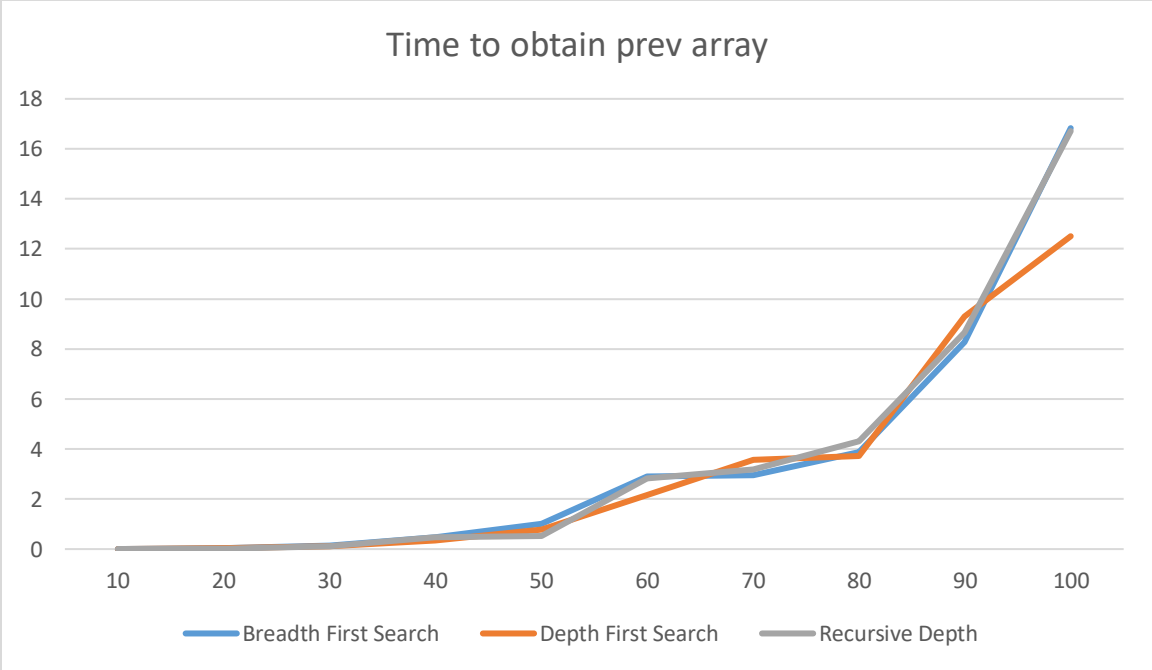
Then we just enter this edge list into a modified version of the maze draw. It is really simple, the vertical walls that would be made instead are going to be horizontal and the horizontal vertical, increasing and decreasing their size by their half.

Design:

There is a user interface that asks for inputs such as the size of the maze, the number of walls that are desired to be removed and what traverse method should be used by the program.

Running Times

Different tests were performed in order to obtain the running times of each type of union took to complete.



Outputs

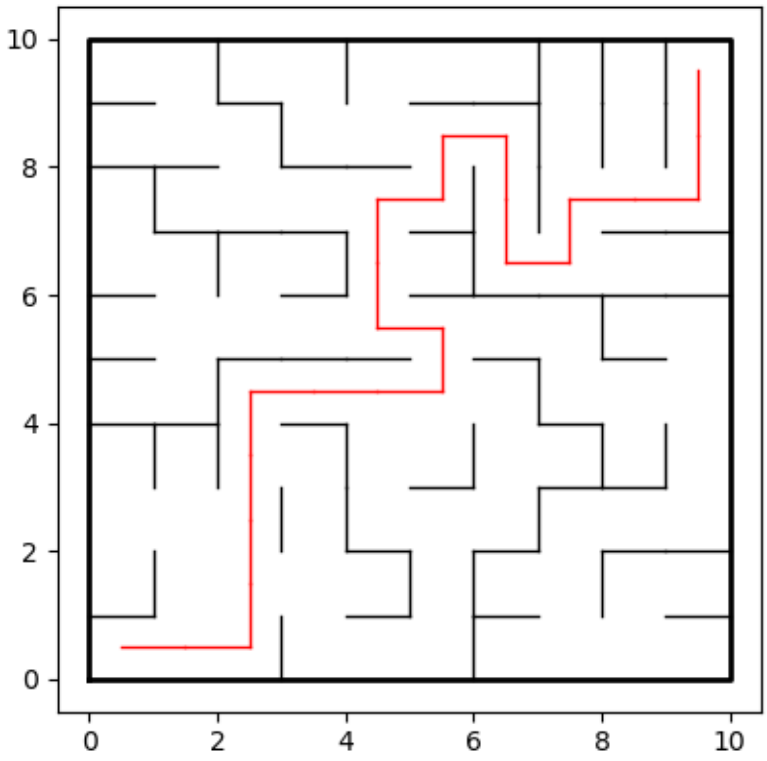
```
Choose the number of columns for the maze: 10

Choose the number of rows for the maze: 10

Choose number of walls to remove: 105

There is at least one path from source to destination.

Choose traversal method (1.BFS, 2.Depth First Search, 3. Recursive Depth Search): 2
time to obtain prev array:  0.0044193267822265625
prev array:  [-1  0  1  4  5 15  7 17  7  8 -1  1  2 12 -1 25 17 27  8 18 -1 22 12 -1
 34 24 25 28 29 39 -1 -1 22 32 44 34 26 36 48 49 -1 -1 32 42 43 44 36 57
 47 48 -1 -1 -1 54 55 45 55 56 59 49 -1 -1 -1 -1 54 64 76 66 67 -1 -1 -1
 -1 74 64 74 86 67 77 78 -1 -1 -1 -1 85 75 85 77 78 79 -1 -1 -1 -1 -1 -1
 -1 87 -1 89]
edje list of path:  [[89, 99], [79, 89], [78, 79], [77, 78], [67, 77], [66, 67], [66, 76], [76, 86],
[85, 86], [75, 85], [74, 75], [64, 74], [54, 64], [54, 55], [45, 55], [44, 45], [43, 44], [42, 43],
[32, 42], [22, 32], [12, 22], [2, 12], [1, 2], [0, 1]]
drawing maze...
```



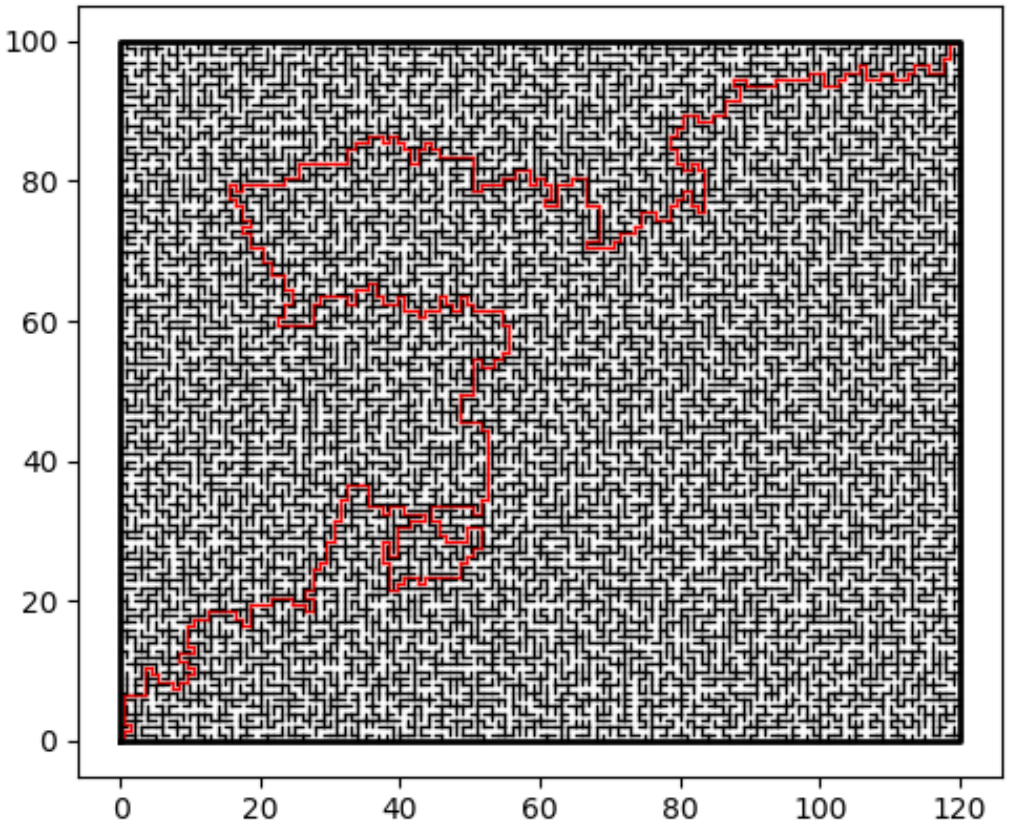
```
Choose the number of columns for the maze: 120

Choose the number of rows for the maze: 100

Choose number of walls to remove: 11999

There is a unique path from source to destination.

Choose traversal method (1.BFS, 2.Depth First Search, 3. Recursive Depth Search): 1
time to obtain prev array:  17.83202624320984
prev array:  [   -1    -1   122 ... 11877 11878 11998]
```



Appendix

```
"""
CS2303 - Data Structures
Jesus A. Hernandez - 80629917
Lab#7 - Graphs
Instructor - Dr. Olac Fuentes
TA - Anindita Nath & Maliheh Zargaran
Last Modified on May 02, 2019
"""
import matplotlib.pyplot as plt
import numpy as np
import random
import time

"""
Disjoint Set Forest -------------------------------------------------------
"""
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 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:
```

```
        S[rj] = ri


"""
Maze Functions ------------------------------------------------------------
"""
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)

# Creates a list with all the walls in the maze
def wall_list(maze_rows, maze_cols):
    w =[]
    for r in range(maze_rows):
        for c in range(maze_cols):
            cell = c + r*maze_cols
            if c!=maze_cols-1: # If not last column
                w.append([cell,cell+1]) # wall between adjacent columns
            if r!=maze_rows-1: #if not last row
                w.append([cell,cell+maze_cols]) # wall between adjacent rows
    return w

# Creates an edje list
def Maze_User(m,S,W):
    edge_list = []
    counter = 0
    path = False
    while counter < m:
        d = random.randint(0,len(W)-1) # Random index
```

```
        if find(S,W[d][0]) != find(S,W[d][1]): # Check if the roots are different
            union(S,W[d][0],W[d][1]) # Join the sets
            edge_list.append(W.pop(d)) # Delete wall
            counter += 1
            if counter >= len(S)-1:
                path = True
        elif path: # Once there is a path start removing any wall
            union(S,W[d][0],W[d][1]) # Join the sets
            edge_list.append(W.pop(d)) # Delete wall
            counter += 1
    return edge_list


"""
Convert Edge List to Adj List ----------------------------------------------------
Adj list will be used in methods for traversing the graph.
"""
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


"""
Find Path using Breadth-First Search ----------------------------------------
Use a Queue to find a path to the last cell. Return the prev array generated.
"""
def findPathBFS(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 of the graph to the queue
    visited[adj[0][0]] = True # Mark the vertex as visited

    while Q:
        if prev[len(adj)-1] >= 0: # Since the wanted vertex has been reached end
            break
        n = Q.pop(0)
        for j in adj[n]:
            if visited[j] == False: # Add to the queue if vertex hasn't been visited
                visited[j] = True
                prev[j] = n
                Q.append(j)
    prev[0] = -1 # Mark the starting vertex as -1 since no vertex points to this
    return prev

"""
Find Path using Depth First Search ------------------------------------------
Use a stack to find the path to the last cell. Some adjustments compared to
the queue had to be made.
"""
def findPathDepth(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:
        if prev[len(adj)-1] >= 0: # Since the wanted vertex has been reached end
```

```
            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

"""
Find Path using Recursive DFS ------------------------------------------------
"""
def recursiveDFS(adj,origin,visited,prev):
    visited[origin] = True
    for i in adj[origin]:
        if visited[i] == False:
            prev[i] = origin
            recursiveDFS(adj, i, visited, prev)
    return prev



"""
Prev Array to Edje List ------------------------------------------------------
To draw the path in the maze we first need to convert the prev array into an
edje list.
"""
def prev_edje(prev):
    E = []
    i = len(prev)-1 # Start at the end
    while True: # Continue until reaching the start point
```

```
        if prev[i] == 0 or prev[i] < 0: # Base case, exit when reached
            E.append([0,i])
            break
        elif i < prev[i]: # Edjes must be in the order of (small,big)
            E.append([i,prev[i]])
        else:
            E.append([prev[i],i])
        i = prev[i]
    return E # Return edje list



"""
Main ----------------------------------------------------------------------
"""
plt.close("all")

c = int(input("Choose the number of columns for the maze: "))
r = int(input("Choose the number of rows for the maze: "))
W = wall_list(r,c)
S = DisjointSetForest(c*r)

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

void = False
if (c*r)-1 == m:
    print("\nThere is a unique path from source to destination.")
elif (c*r) > m:
    print("\nA path from source to destination is not guaranteed to exist.")
    void = True
else:
    print("\nThere is at least one path from source to destination.")
```

```
selection = 0
if void:
    print("There is no path since maze has no solution.")
else:
    selection = int(input("Choose traversal method (1.BFS, 2.Depth First Search, 3. Recursive Depth Search): "))


start = time.time()
edge_list = Maze_User(m,S,W) # Obtain edje list from randomly created maze


adj_list = edge_list_to_adj_list(edge_list,c*r) # Obtain adj list from edje list


if selection == 1:
    prev = findPathBFS(adj_list) # Function ends when goal has been reached to shorten time
    end = time.time()
    print("time to obtain prev array: ",end-start)
    print("prev array: ",prev) # Array won't be completed sometimes due to above
    path = (prev_edje(prev))
    print("edje list of path: ",path)
    print("drawing maze...")
    draw_maze_path(W,path,r,c)


if selection == 2:
    prev = findPathDepth(adj_list) # Function ends when goal has been reached to shorten time
    end = time.time()
    print("time to obtain prev array: ",end-start)
    print("prev array: ",prev) # Array won't be completed sometimes due to above
    path = (prev_edje(prev))
    print("edje list of path: ",path)
    print("drawing maze...")
    draw_maze_path(W,path,r,c)
```

```
if selection == 3:
    visited = [False]*len(adj_list)
    p = np.zeros(len(adj_list),dtype=int)-1
    prev = recursiveDFS(adj_list,0,visited,p)
    end = time.time()
    print("time to obtain prev array: ",end-start)
    print("prev array: ",prev)
    path = (prev_edje(prev))
    print("edje list of path: ",path)
    start = time.time()
    print("drawing maze...")
    draw_maze_path(W,path,r,c)
    end = time.time()
    print("time to draw maze: ",end-start)
```

I certify that this essay is original work prepared by me, Jesus A Hernandez.