CS3 Lab 6 Report

Daniela Flores

Intro

Lab 7 consisted of implementing more methods and modifications to the program written for lab 6. In lab 7 we had to draw a maze with the number of walls removed being whatever the user gave as input. We also had to do breadth first and depth first searches on a maze to find the solution of it and compare their running times. We also had to do the adjacency list representation of the maze.

Proposed Solution and Implementation

I first started this lab by modifying the method for drawing a maze with the disjoint set forest to be able to remove the desired number of walls from the user. An if statement checked if the number of walls wanting to be removed exceeded more of those that are necessary to draw a maze with one solution only. inside this if statement a while loop removes the number of walls that the user gave as input by getting a random wall, splitting this walls coordinates to get the adjacent cells, checking if the walls belong in the same set and uniting the cells if they do not belong in the same set. This while loop keeps iterating until a counter, which keeps track of the number of walls being removed, equals the number of walls given by the user. I also created the adjacency list representation of the maze in this method by initializing a list of lists and adding the cells into one another when uniting said cells, for example: list[x] = y and list[y] = x. at the end of the method I drew the maze and returned the adjacency list.

The next method I worked on was the breath first search to find the solution of a maze. Dr.Fuentes gave us the pseudocode for this method so I simply just translated it to python syntax. This method first started by initializing a list populated with 'False' and another one that was populated with negative ones. Both lists were the size of the adjacency list given as a parameter of this method. I then initialized a queue and appended the starting cell given as a parameter to it. A while loop then iterates until the queue is empty. Inside this while loop a variable stores the first element being dequeued, then a for loop iterates through all the elements of that certain list in the adjacency list. Inside this for loop all the elements in the visited list which are in the indices of the list of the adjacency list are set to True. After the for loop is done being executed the list 'prev' is returned.

The next method I worked on was the depth first search without recursion. This method is exactly like the breadth first search method but instead of using a queue, a stack is used. The depth first search method with recursion consists of a for loop and a nested if statement, inside this if statement the recursive call is made.

The last method I wrote was the one that printed the solution of the maze. This method was also provided by Dr.Fuentes as pseudocode, I just rewrote it to meet pythons syntax. This method consisted of an if statement that checks that the number being stored at a certain index is not –1. Inside this if statement the recursive call is made and a print statement to print the arrows that show which way the path is going. Outside this if statement another print statement is used to print the cells in the solution.

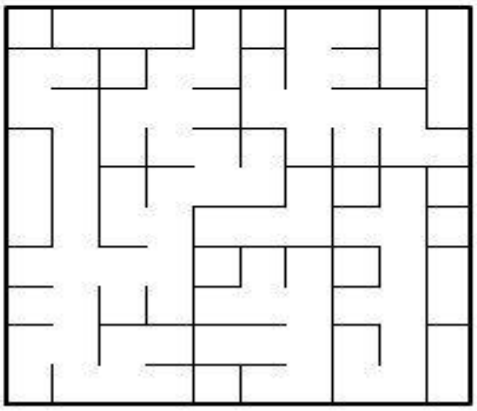
Experimental Results:

I experimented with my program by first checking if the program I wrote for lab 6 was useful for this lab, and it partially was I just had to modify it a bit. I also experimented with what would happen if the user

input a negative or a number bigger than the number of walls in the maze. I then took care of those cases with if statements. The following images is what my program output when run:

```
In [40]: runfile('C:/Users/flore/.spyder-py3/cs3Lab7.py',
wdir='C:/Users/flore/.spyder-py3')
number of cells: 100

type number of walls you want to remove
80
A path from source to destination is not guaranteed to
exist
```

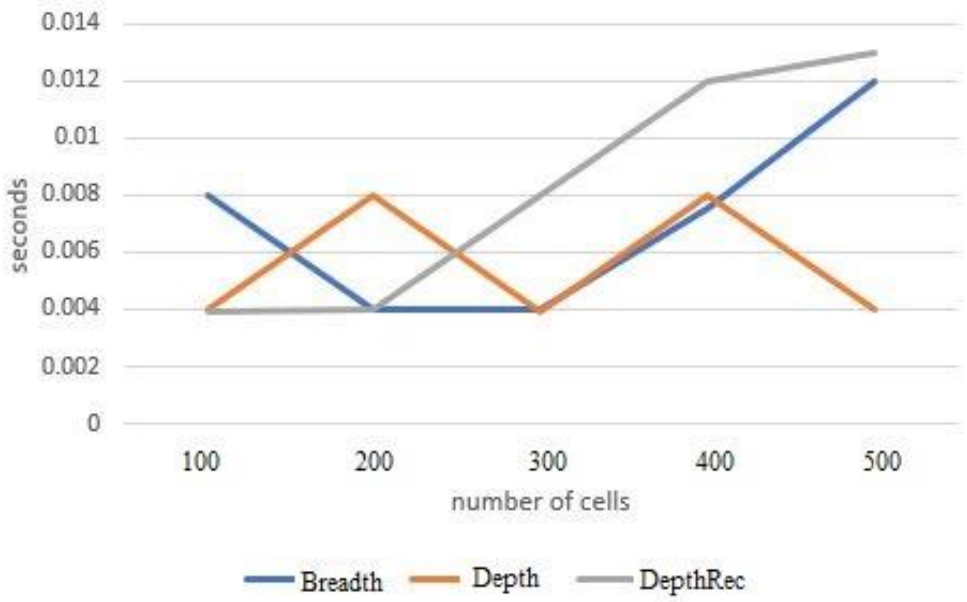


```
adjacency List: [[1], [2, 11, 0], [1], [4], [3, 14, 5],
[4], [16], [17], [18, 9], [19, 8], [11], [12, 10, 21, 1],
[11], [23], [24, 15, 4], [14, 25, 16], [26, 15, 6], [7, 18,
27], [17, 8], [9], [21], [20, 31, 11], [32, 23], [33, 13,
22], [14, 34], [15], [16, 27], [28, 17, 26, 37], [27],
[39], [31],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

```
depth first iteravely
0  ->  1  ->  11  ->  21  ->  31  ->  32  ->  22  ->  23  -
>  33  ->  34  ->  44  ->  45  ->  55  ->  56  ->  57  ->
58  ->  59  ->  69  ->  79  ->  78  ->  88  ->  98  ->  99
depth first recursively
0  ->  1  ->  11  ->  21  ->  31  ->  32  ->  22  ->  23  -
>  33  ->  34  ->  44  ->  45  ->  55  ->  56  ->  57  ->
58  ->  59  ->  69  ->  79  ->  78  ->  88  ->  98  ->  99
```

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

Running Times:



Conclusion:

What I learned from this lab was how Breadth and Depth first searches work. I also learned more about disjoint set forests as that was used to draw the maze with one solution. I also learned on how to create an adjacency list with a maze.

Appendix:

```
#Course:CS2302
#aAuthor:Daniela Flores
#Lab7
#Instructor:Olac Fuentes
#T.A: Dita Nath
#date of last Mod: 4/30/19
#purpose of program: draw a maze with the help of a disjoint set forest to better
#understand this data structure,draw maze with the help of user input, do an
#adjacency list based on the maze,and do various types of searches to solve the maze
import matplotlib.pyplot as plt
import numpy as np
import random
import time
from collections import deque
#method that draws maze, provided by Dr.Fuentes
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 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
#############################################################
#method that craetes a DSF with given size
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: # Do nothing if i and j belong to the same set
        S[rj] = ri  # Make j's root point to i's root

########################################################
#union by compression
def union_by_size(S,i,j):
    ri = findC(S,i)
    rj = findC(S,j)
    if ri!=rj:
        if S[ri] > S[rj]:
            S[rj] += S[ri]
            S[ri] = rj
        else:
            S[ri]+=S[rj]
            S[rj] = ri


#finds the root using path compression
def findC(S,i):
```

```
    if S[i] < 0:
        return i
    r = findC(S,S[i])
    S[i] = r
    return r


#######################################################
#method that returns True if theres more than one set in the disjoint set forest
# and false otherwise
def numOfSets(forest):
    count = 0
    #counts number of sets
    for i in range(len(forest)):
        if forest[i] == -1:
            count = count +1
    if count>1:
        return True
    else:
        return False

#method that will keep popping walls if the adjacent cells are not in the same set
#draws maze with standard union
def wallsAsforests(walls,row,cols,bool):
    wList = [ [] for i in range(row*cols)]
    counter = 0
    forest = DisjointSetForest(row*cols)
    #while the disjoint set forest has more than 1 set this loop will keep iterating
    while numOfSets(forest) == True:
```

```
        #gets random integer
        d = random.randint(0,len(walls)-1)
        #stores random wall
        randomWall = walls[d]
        #splits the wall's cordinate to get adjacent cells
        cellX = randomWall[0]
        cellY = randomWall[1]
        #checks if cells are not in same set
        if findC(forest,cellY) != findC(forest,cellX):
            #gets both cells to same set
            union(forest,cellX,cellY)
            ########################
            #creates adjacency list
            wList[cellX].append(cellY)
            wList[cellY].append(cellX)
            counter= counter+1
            #removes wall
            walls.pop(d)
    #draws finished maze
    draw_maze(walls,row,cols,cell_nums = bool)
    #returns adjacency representation of maze
    return wList
#modified version of method above to remove the # of walls provided by user
def wallsAsForest7(walls,row,cols,wallsP):
    counter = 0
    forest = DisjointSetForest(row*cols)
    #if statement checks if number of walls being removed is bigger than
    #the cumber of cells
    if len(forest)<wallsP:
```

```
    while counter !=wallsP:
      #gets random integer
      d = random.randint(0,len(walls)-1)
      #stores random wall
      randomWall = walls[d]
      #splits the wall's cordinate to get adjacent cells
      cellX = randomWall[0]
      cellY = randomWall[1]
      #checks if all cells are already in one set
      if numOfSets(forest) == False:
        #for loop removes additional walls
        #to meet the desired number of walls the user
        #wants removed
        for i in range(wallsP-counter):
          d = random.randint(0,len(walls)-1)
          counter = counter+1
          walls.pop(d)
        break
      if findC(forest,cellY) != findC(forest,cellX):
        #gets both cells to same set
        union(forest,cellX,cellY)
        counter= counter+1
        #removes wall
        walls.pop(d)
else:
  while counter !=wallsP:
    #gets random integer
    d = random.randint(0,len(walls)-1)
    #stores random wall
```

```
            randomWall = walls[d]
            #splits the wall's cordinate to get adjacent cells
            cellX = randomWall[0]
            cellY = randomWall[1]
            #checks if cells are not in same set
            if findC(forest,cellY) != findC(forest,cellX):
                #gets both cells to same set
                union(forest,cellX,cellY)
                counter= counter+1
                #removes wall
                walls.pop(d)
    #draws finished maze
    draw_maze(walls,row,cols)

#method that does breath first search to solve maze
def breadthFirst(G,v):
    visited = []
    visited = [False for i in range(len(G))]
    prev = []
    prev = [-1 for i in range(len(G))]
    Q = deque([])
    Q.append(v)
    visited[v] = True
    while Q:
        #popleft removes first element in Q
        u = Q.popleft()
        for t in G[u]:
            if not visited[t]:
                visited[t] = True
```

```
            prev[t] = u
            Q.append(t)
    #returns solution of this search
    return prev

#this method does the iterative verison of depth first search to solve maze
def ItDepthFirst(G,v):
    visited = []
    visited = [False for i in range(len(G))]
    prev = []
    prev = [-1 for i in range(len(G))]
    Q = []
    Q.append(v)
    visited[v] = True
    while Q:
        #gets last element
        u = Q.pop()
        for t in G[u]:
            if not visited[t]:
                visited[t] = True
                prev[t] = u
                Q.append(t)
    #returns solution of this search
    return prev

#this method does the recursive depth first search to solve maze
def depthFirstR(G,source):
    visitedD[source] = True
    for t in G[source]:
```

```
        if not visitedD[t]:
            prevD[t] = source
            depthFirstR(G,t)
    #returns solution of this search
    return prevD
#this method prints the path solution given the list
def printPath(prev,v):
    if prev[v] !=-1:
        printPath(prev,prev[v])
        print(" -> ", end=' ')
    print(v,end=' ')



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


maze_rows = 10
maze_cols = 10
cells = maze_rows*maze_cols
walls = wall_list(maze_rows,maze_cols)
print('lenwalls',len(walls))
print('number of cells:',cells)
#aks user how many walls it wants removed and stores answer
userInput = int(input('type number of walls you want to remove\n'))
#checks if users input is valid
if userInput>=0 and userInput<=len(walls):
    if userInput <cells-1:
        print('A path from source to destination is not guaranteed to exist')
    if userInput ==cells-1:
```

```
   print(' There is a unique path from source to destination')
if  userInput >cells-1:
   print('There is at least one path from source to destination')
plt.close("all")
#draws maze with x num of walls rempoved, x being the users input
wallsAsForest7(walls,maze_rows,maze_cols,userInput,)
plt.show()
walls2 = wall_list(maze_rows,maze_cols)
plt.close("all")
#stores adjacency list representation of maze
G = wallsAsforests(walls2,maze_rows,maze_cols,True)
print('adjacency List:',G)
########################
#BREADTH FIRST SEARCH
print('breadth first')
bFirst = breadthFirst(G,0)
printPath(bFirst,len(G)-1)
print('')
#########################
#DEPTH FIRST SEARCH
global visitedD
visitedD = [False for i in range(len(G))]
global prevD
prevD = [-1 for i in range(len(G))]
print('depth first iteravely')
dFirst = ItDepthFirst(G,0)
printPath(dFirst,len(G)-1)
print('')
#########################
```

```
    #DEPTH FIRST SEARCH USING RECURSION


    print('depth first recursively')
    dFirstR = depthFirstR(G,0)
    printPath(dFirstR,len(G)-1)
    print('')
    plt.show()
    #start = time.time()
    #elapsedTime = time.time() - start
    #print('standard:',elapsedTime)
else:
    print('invalid number of cells')
```

Contract:

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 inappropiate assistance to any student in the class.

X