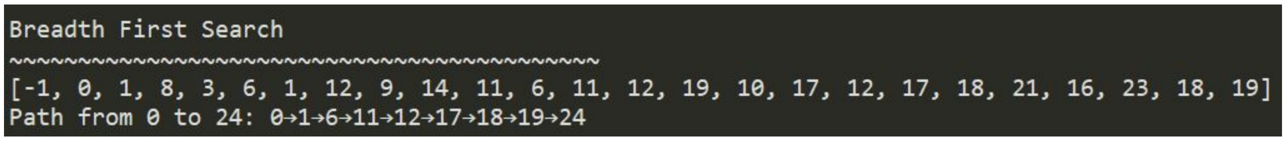
Author: *Nguyen, Abram*
Assignment: *Lab 7 Report*
Course: *CS 2302 - Data Structures*
Instructor: *Fuentes, Olac*
T.A.: *Nath, Anindita*

Introduction:

In this lab, I'm comparing three different path finding algorithms. I'm comparing breadth first search, depth first search using stacks, and depth first search using recursion. I will test these three algorithms on four significantly different sized mazes. The mazes were created using disjoint set forests.

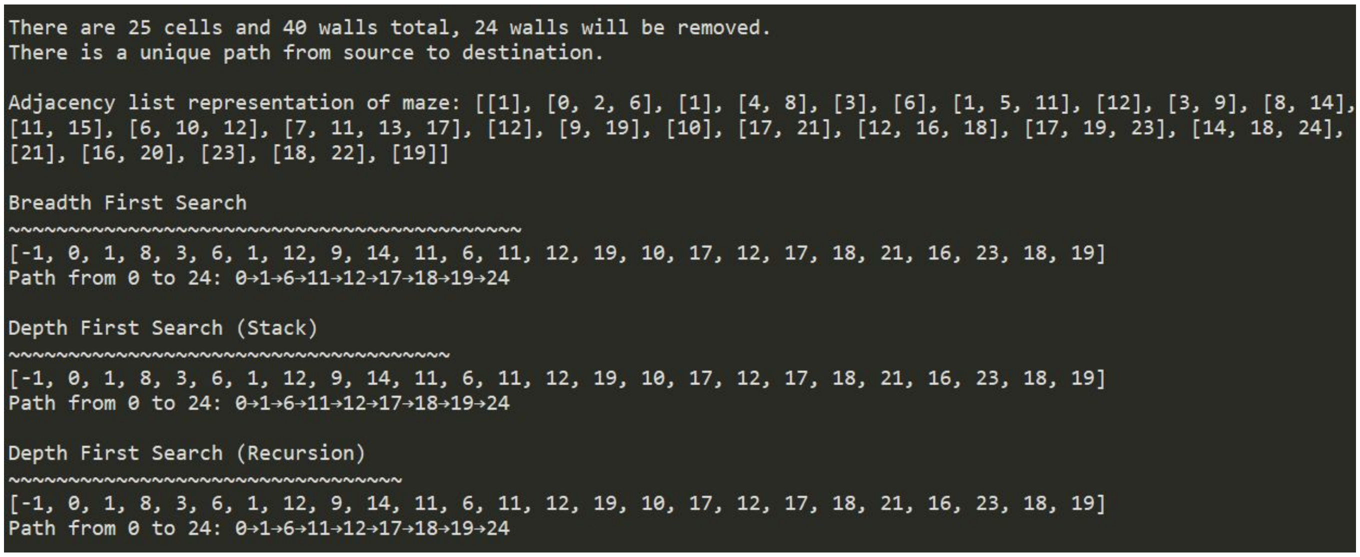
Proposed solution design and implementation:

Each of the three path finding algorithms used in the lab return a list of values. The list is similar to a disjoint set forest. It is a list that tells us what a given node's previously visited node is. For example, in the list below, cell (node) number 24 points to cell 19. Cell 19 points to 18, which points to 17, which points to 12, and so on. This will eventually lead us to cell 0, which contains -1, letting us know that this is the root of the set of nodes. By taking this path, we get the optimal path through the maze from the first to the last cell.



```
Breadth First Search
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[-1, 0, 1, 8, 3, 6, 1, 12, 9, 14, 11, 6, 11, 12, 19, 10, 17, 12, 17, 18, 21, 16, 23, 18, 19]
Path from 0 to 24: 0→1→6→11→12→17→18→19→24
```

To get the adjacency list representation of the maze, I compared a list of walls that still contained all of its original walls and a list of walls of the maze I was using. I would add every wall that did not exist in the list of walls of the maze to the adjacency list.



```
There are 25 cells and 40 walls total, 24 walls will be removed.
There is a unique path from source to destination.

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

Breadth First Search
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[-1, 0, 1, 8, 3, 6, 1, 12, 9, 14, 11, 6, 11, 12, 19, 10, 17, 12, 17, 18, 21, 16, 23, 18, 19]
Path from 0 to 24: 0→1→6→11→12→17→18→19→24

Depth First Search (Stack)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[-1, 0, 1, 8, 3, 6, 1, 12, 9, 14, 11, 6, 11, 12, 19, 10, 17, 12, 17, 18, 21, 16, 23, 18, 19]
Path from 0 to 24: 0→1→6→11→12→17→18→19→24

Depth First Search (Recursion)
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
[-1, 0, 1, 8, 3, 6, 1, 12, 9, 14, 11, 6, 11, 12, 19, 10, 17, 12, 17, 18, 21, 16, 23, 18, 19]
Path from 0 to 24: 0→1→6→11→12→17→18→19→24
```

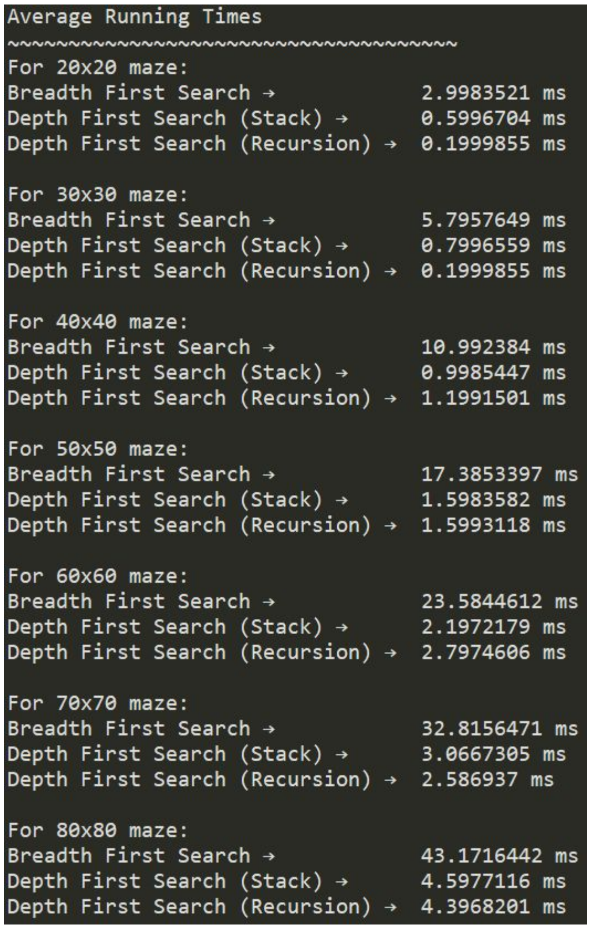
1) In the breadth first search algorithm, lists of already visited nodes and a path are being kept track of using a queue. The nodes being visited are put into a queue, so nodes of similar distance from the root are being explored almost simultaneously. Every possible path is being explored alongside one another, every vertex is explored once and the optimal path from the root to the last node (bottom-left position of the maze to the upper-right position of the maze) can be found. Breadth first search is usually better when searching for nodes that are more likely to be closer to the root. In this case, the end of the maze is usually farther away from the starting point.
2) The depth first search algorithm that uses a stack looks extremely similar to breadth first search, which uses a queue. It keeps track of already visited nodes and a path that's being taken. In depth first search, an arbitrary node adjacent to the root (starting node) is chosen and that path is explored as deeply as possible. Once a new node is visited, it is added to a stack. If a wall is hit or, in other words, no nodes can be explored down the path, the algorithm will back track and explore other possible paths. This means that a node will be removed from the stack and it's unvisited adjacent nodes will be explored as well. Every node will be visited once. Depth first search is usually better when searching for nodes that are closer to the leaves of the graph. In this case, it would suit the problem better because we're looking for the end of a maze.
3) The depth first search algorithm that uses recursion uses global lists of previously visited nodes and a path that will be used to find the end of a maze. The algorithm is still depth first search, it's just implemented in a different way than the previous algorithm. If a node that is found has not been visited, it is recorded as visited and added to a path of nodes, like in the previously discussed path finding algorithms. The recursive call is made when an unvisited node is found.

<u>Experimental results:</u>

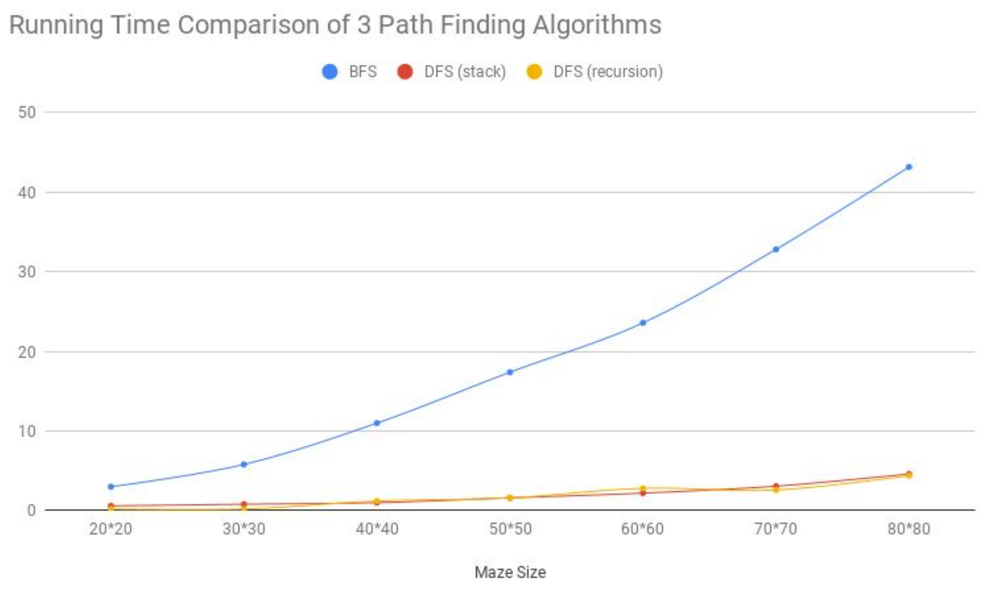
I modified my code so that a unique path from the root to the end of the maze always exists. I did it this way to simplify the code and focus more on the experiments with the three path finding algorithms.

I created 5 mazes of 7 different maze sizes, the sizes being 20, 30, 40, 50, 60, 70 and 80. The total number of cells in the size 20x20 maze is 400, for reference. I took the average run time to find a path from the source to the destination for each of the three path finding algorithms. Below are the results of the running times and the mazes I created.

	Average Times(ms)		
Maze Size	BFS	DFS (stack)	DFS (recursion)
20*20	2.9983521	0.5996704	0.1999855
30*30	5.7957649	0.7996559	0.1999855
40*40	10.992384	0.9985447	1.1991501
50*50	17.3853397	1.5983582	1.5993118
60*60	23.5844612	2.1972179	2.7974606
70*70	32.8156471	3.0667305	2.586937
80*80	43.1716442	4.5977116	4.3968201



```
Average Running Times
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
For 20x20 maze:
Breadth First Search →          2.9983521 ms
Depth First Search (Stack) →    0.5996704 ms
Depth First Search (Recursion) → 0.1999855 ms

For 30x30 maze:
Breadth First Search →          5.7957649 ms
Depth First Search (Stack) →    0.7996559 ms
Depth First Search (Recursion) → 0.1999855 ms

For 40x40 maze:
Breadth First Search →          10.992384 ms
Depth First Search (Stack) →    0.9985447 ms
Depth First Search (Recursion) → 1.1991501 ms

For 50x50 maze:
Breadth First Search →          17.3853397 ms
Depth First Search (Stack) →    1.5983582 ms
Depth First Search (Recursion) → 1.5993118 ms

For 60x60 maze:
Breadth First Search →          23.5844612 ms
Depth First Search (Stack) →    2.1972179 ms
Depth First Search (Recursion) → 2.7974606 ms

For 70x70 maze:
Breadth First Search →          32.8156471 ms
Depth First Search (Stack) →    3.0667305 ms
Depth First Search (Recursion) → 2.586937 ms

For 80x80 maze:
Breadth First Search →          43.1716442 ms
Depth First Search (Stack) →    4.5977116 ms
Depth First Search (Recursion) → 4.3968201 ms
```



Conclusion:

This lab helped me to learn more about the breadth first search and depth first search path finding algorithms. I think it's interesting that the only difference between my second and first algorithm was the use of a stack and a queue. I feel that I also understand graphs better now than I did before. This lab was mostly interesting to me because I like mazes and seeing my code come out to a finished product. The use of the breadth first search and depth first search algorithms to find a solution to the mazes are even more interesting.

Appendix:

```
"""
        Author:             Nguyen, Abram
        Assignment:         Lab 7
        Course:             CS 2302 - Data Structures
        Instructor:         Fuentes, Olac
        T.A.:               Nath, Anindita
        Last modified:

        Purpose of program: The purpose of this program is to demonstrate different
                               path finding algorithms by solving a maze created using
                               a disjoint set forest. The algorithms implemented are
                               breadth first search, depth first search using stacks,
                               and depth first search using recursion.
        """
        import matplotlib.pyplot as plt
        import numpy as np
        import random
        import queue


        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


        def find_c(S,i):
           if S[i]<0:
               return i
           r = find_c(S,S[i])
           S[i] = r
           return S[i]

        def union_by_size(S, i, j):
           ri = find_c(S, i)
           rj = find_c(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
```

```
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:
               w.append([cell,cell+1])
           if r!=maze_rows-1:
               w.append([cell,cell+maze_cols])
   return w


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 user_input():
   m = input("How many walls should be removed?\n")
   return int(m)


def numSets(S):
   c = 0
   for n in S:
       if n < 0:
           c += 1
   return c


def make_maze(size, walls):
   m = user_input() # how many walls to remove from maze
   n = size*size
   S = DisjointSetForest(n)
   print(f"\nThere are {n} cells and {len(walls)} walls total, {m} walls will be removed.")
   # w = size*size + size*(size-2)

   if m > n-1:
       sets = numSets(S)
       print("There is at least one path from source to destination.\n")
       if m > len(walls): # if all walls are going to be removed
           return []
       # remove walls...
       while m > 0:
           #Select a random wall w =[c1,c2]
           w = random.randint(0, len(walls)-1)
           #If cells c1 and c2 belong to different sets...
           if find_c(S, walls[w][0]) != find_c(S, walls[w][1]):
               #remove w and join c1’s set and c2’s set
               union_by_size(S, walls[w][0], walls[w][1])
               walls.pop(w)
               m -= 1
               sets -= 1
```

```
            if sets == 1: # avoid infinite loop
                walls.pop(w)
                m -= 1
        return walls

    elif m < n-1:
        print("A path from source to destination is not guaranteed to exist.\n")
    elif m == n-1:
        print("There is a unique path from source to destination.\n")

    while m > 0:
        #Select a random wall w =[c1,c2]
        w = random.randint(0, len(walls)-1)
        #If cells c1 and c2 belong to different sets...
        if find_c(S, walls[w][0]) != find_c(S, walls[w][1]):
            #remove w and join c1’s set and c2’s set
            union_by_size(S, walls[w][0], walls[w][1])
            walls.pop(w)
            m -= 1
    return walls

def walls_to_adj_list(walls, size, walls_og):
    G = []
    for g in range(size*size):
        G.append([])
    for w in walls_og:
        if w not in walls: # if there is a wall, cells aren't adjacent
            G[w[0]].append(w[1]) # insert adjacency
            G[w[1]].append(w[0]) # in undirected graph
    return G


# referenced from given graph search pseudocode
def BFS(G, v):
    visited = [False]*len(G)
    visited[v] = True
    prev = [-1]*len(G)
    Q = queue.Queue() # https://docs.python.org/2/library/queue.html#queue-objects
    Q.put(v) # enqueue v
    while not Q.empty():
        u = Q.get() # dequeue
        for t in G[u]:
            if not visited[t]:
                visited[t] = True
                prev[t] = u
                Q.put(t) # enqueue t
    return prev


# referenced from given graph search pseudocode
def DFS_stack(G, v):
    visited = [False]*len(G)
    visited[v] = True
    prev = [-1]*len(G)
    S = []
    S.append(v) # add v to stack
    while len(S) > 0:
        u = S.pop() # take from top of stack
        for t in G[u]:
            if not visited[t]:
                visited[t] = True
                prev[t] = u
                S.append(t) # add t to stack
    return prev


# referenced from given graph search pseudocode
def DFS_rec(G, v):
    visited[v] = True
    for t in G[v]:
        if visited[t] is False:
            prev[t] = v
            DFS_rec(G, t)
    return prev


# referenced from given graph search pseudocode
def printPath(prev, v):
    if prev[v] != -1:
        printPath(prev, prev[v])
        print("→", end="")
    print(v, end="")

# draw a line that leads from cell 0 to the last cell in a maze
# https://matplotlib.org/2.0.2/api/_as_gen/matplotlib.axes.Axes.plot.html
def draw_path(walls, maze_rows, maze_cols, G, 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')


    B = BFS(G, 0)
    i = len(B)-1
    while B[i] != -1:
        # print(i, B[i])
        if i - B[i] == 1: # draw horizontal line
            x0 = i%maze_cols + .5
            x1 = x0 - 1
            y0 = B[i]//maze_rows + .5
            y1 = y0
        else: # draw vertical line
            x0 = i%maze_cols + .5
            x1 = x0
            y0 = B[i]//maze_rows + .5
            y1 = y0 - 1
        # print("FROM", x0, y0, "TO", x1, y1)
        ax.plot([x0,x1],[y0,y1],linewidth=1,color='r')
        i = B[i]
    """
    B = BFS(G, 0)
    print(B)
    path = []
    i = len(B)-1
    while B[i] != -1:
        path.append([i, B[i]])
        i = B[i]

    for i in range(len(path)):
        if path[i][0] - path[i][1] == 1: # draw horizontal line
            x0 = path[i][0]//maze_rows + 0.5
            x1 = x0 - 1
            y0 = path[i][0]//maze_cols + 0.5
            y1 = y0
        else:
            x0 = path[i][]
     """
    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)

"""
################################################################################
################################################################################
################################################################################
"""


plt.close("all")
```

```
# determine maze size, num of cells = size*size
size = int(input("What should be the size of the maze?\n"))


# form walls
walls_original = wall_list(size, size)
walls = wall_list(size, size)


# construct a maze
maze = make_maze(size, walls)


# draw cells w/ numbers and maze result
draw_maze(walls_original, size, size, cell_nums=True)
draw_maze(maze, size, size, cell_nums=False)


# print adj list representation of maze
G = walls_to_adj_list(walls, size, walls_original)
print("Adjacency list representation of maze:", G)


# draw path of maze solution, found using BFS
draw_path(walls, size, size, G)


# using path finding algorithms to find the shortest path from bottom-left to top-right of maze
global visited
global prev
print()
print("Breadth First Search\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
B = BFS(G, 0)
# print(B) # cell[i] is the cell number previously taken on path from 0 to size*size-1
print(f"Path from 0 to {size*size-1}:", end=" ")
printPath(B, size*size-1)
print("\n")


print("Depth First Search (Stack)\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
D = DFS_stack(G, 0)
# print(D) # cell[i] is the cell number previously taken on path from 0 to size*size-1
print(f"Path from 0 to {size*size-1}:", end=" ")
printPath(D, size*size-1)
print("\n")


visited = [False]*len(G)
prev = [-1]*len(G)
print("Depth First Search (Recursion)\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
R = DFS_rec(G, 0)
# print(R) # cell[i] is the cell number previously taken on path from 0 to size*size-1
print(f"Path from 0 to {size*size-1}:", end=" ")
printPath(R, size*size-1)
print("\n")
###############################################################################
# End of program
```

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.

- Abram Nguyen