Sergio Gramer – 88521512
Lab #7: Disjoint Set Forest Maze Pt. 2

In lab 7 we are tasked with continuing our previous lab (lab 6) where we created a maze from a disjoint set forest. This time around we are supposed to make the program let us know based on the number of walls being removed if:

A.) There was no path being able to be generated
B.) There was one unique path in the maze
C.) There was at least one unique path

After making the selection of number of walls we want to remove the program is supposed to display the message (above). After this we are supposed to build an adjacency list that is built randomly because the walls are removed randomly. Finally, we are tasked with implementing

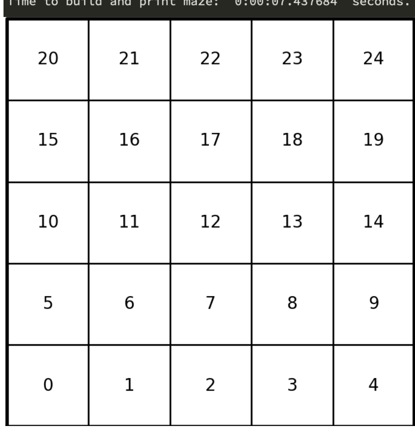
A.) Breadth-First Search
B.) Depth-First Search using a stack
C.) Depth-First Search using recursion

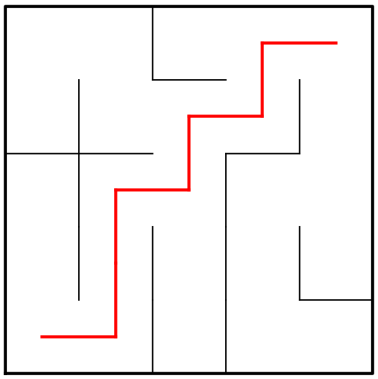
All of this is to be displayed at the end of the program by drawing the maze that we start off with and the final completed maze with a red line delineating the path we are taking from beginning to end.

To begin solving these tasks I started by using the same program I had created for Lab 6. I modified the algorithm that would create the graph to also take the walls that we were removing and placing them inside of another list. This was going to be used a later time to build the adjacency list. Once that was finished I included in the beginning of the program a section that would prompt the user to enter the number of walls based on the number of allotted squares inside of the maze. This would then trigger an if-else statement that would display if there was a guaranteed path (unique) to the end of the maze, more than one paths to the end, no paths to the finish or the user was attempting to remove too many walls that didn't exist. If the latter of the choices was taken the system would exit the program and the user would have to run again. I used the instructors pseudo-code that he provided for the breadth-first and depth-first search to create my algorithms. It was not difficult to follow what was supposed to be coded and is written almost to the point.

More than 1 unique path

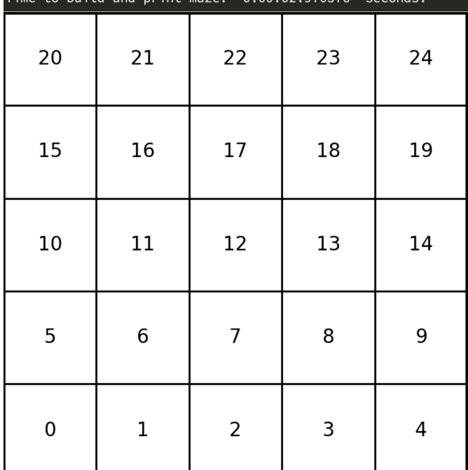
```
In [3]: runfile('/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab 7 Maze pt.2/lab7.py', wdir='/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab 7 Maze pt.2')
The Number of cells in this maze are:  25
Please enter the number of walls you would like to remove.

24
There is a unique path from source to destination.
Adjacency List:  [[1, 5], [0, 6], [7], [4, 8], [3], [0, 10], [1, 11], [2, 12], [3, 13], [14], [5], [6, 12], [7, 11, 17], [8, 14], [9, 13, 19], [20], [17, 21], [12, 16,
18], [17, 23], [14, 24], [15, 21], [16, 20], [23], [18, 22, 24], [19, 23]]
Disjoint Set Forest Maze:  [-1  7 15  7  3  0  1 22  3 18  0  6  6  3  9  0 15  1  3  9 15 16  2  9
 23]
Breadth First Search:  [-1, 0, 7, 8, 3, 0, 1, 12, 13, 14, 5, 6, 11, 14, 19, 20, 17, 12, 17, 24, 21, 16, 23, 18, 23]
Depth First Search:  [-1, 0, 7, 8, 3, 0, 1, 12, 13, 14, 5, 6, 11, 14, 19, 20, 17, 12, 17, 24, 21, 16, 23, 18, 23]
Depth First Search Recursively:  [-1, 0, 7, 8, 3, 0, 1, 12, 13, 14, 5, 6, 11, 14, 19, 20, 17, 12, 17, 24, 21, 16, 23, 18, 23]
Time to build and print maze:  0:00:07.437684  seconds.
```

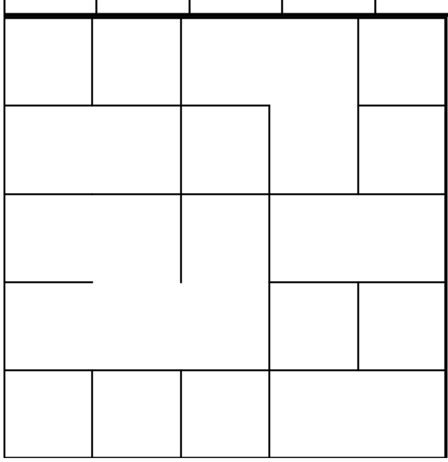




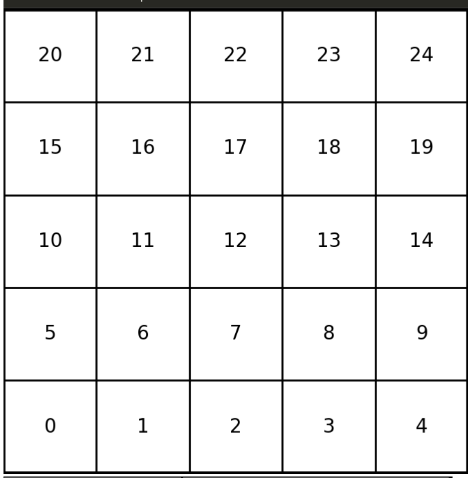
No unique path.

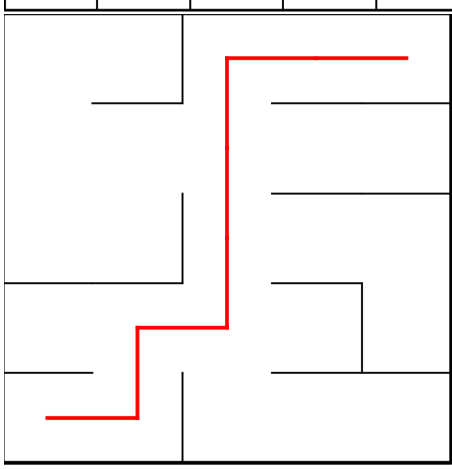
```
In [4]: runfile('/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab 7 Maze pt.2/lab7.py', wdir='/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab 7 Maze pt.2')
The Number of cells in this maze are:  25
Please enter the number of walls you would like to remove.

10
A path from source to destination is not guaranteed to exist.
Adjacency List:  [[], [], [], [4], [3], [6], [5, 7, 11], [6, 12], [], [], [11], [6, 10], [7], [14], [13], [16], [15], [], [23], [], [], [], [23], [18, 22], []]
Disjoint Set Forest Maze:  [-1 -1 -1 -1  3 10  5  6 -1 -1 -1  5  5 -1 13 -1 15 -1 -1 -1 -1 -1 18 22
 -1]
Breadth First Search:  [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
Depth First Search:  [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
Depth First Search Recursively:  [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
Time to build and print maze:  0:00:02.970376  seconds.
```





```
In [5]: runfile('/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab 7 Maze pt.2/lab7.py', wdir='/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab 7 Maze pt.2')
The Number of cells in this maze are:  25
Please enter the number of walls you would like to remove.

25
There is at least one path from source to destination.
Adjacency List:  [[1], [0, 6], [3, 7], [2, 4], [3], [6], [1, 5, 7], [2, 6, 8, 12], [7], [14], [11, 15], [10, 16], [7, 13, 17], [12, 14], [9, 13], [10, 16, 20], [11, 15,
17], [12, 16, 18, 22], [17, 19], [18], [15, 21], [20], [17, 23], [22, 24], [23]]
Disjoint Set Forest Maze:  [-1  5  0  2  3  0  1  2  0 10  0 12  9 12 13 10 11 11 17 17 15  0 17 22
  9]
Breadth First Search:  [-1, 0, 7, 2, 3, 6, 1, 6, 7, 14, 11, 16, 7, 12, 13, 16, 17, 12, 17, 18, 15, 20, 17, 22, 23]
Depth First Search:  [-1, 0, 7, 2, 3, 6, 1, 6, 7, 14, 15, 16, 7, 12, 13, 16, 17, 12, 17, 18, 15, 20, 17, 22, 23]
Depth First Search Recursively:  [-1, 0, 7, 2, 3, 6, 1, 6, 7, 14, 11, 16, 7, 12, 13, 10, 17, 12, 17, 18, 15, 20, 17, 22, 23]
Time to build and print maze:  0:00:02.856881  seconds.
```





Appendix: Source Code

```
############################################################################
#                                                                          #
#                    """ GIVEN """                                         #
#                                                                          #
############################################################################

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)
    return ax

def path(plot, prev, vert, edgex, edgey) : #we used our axplot to generate the path(Red) from beginning to end if it exists
    if prev[vert] != -1: #when prev does not equal -1 it will continue.
        if (vert - 1) == prev[vert] : #if the prev vertice is equal to vert -1
            x1 = edgex - 1
            y1 = edgey
            path(plot, prev, prev[vert], x1, y1)
            plot.plot([x1, edgex], [y1, edgey], linewidth = 2, color = 'r')
        if (vert + 1) == prev[vert] :
            x1 = edgex + 1
            y1 = edgey
            path(plot, prev, prev[vert], x1, y1)
            plot.plot([x1, edgex],[y1, edgey], linewidth = 2, color = 'r')
        if (vert - maze_cols) == prev[vert] :
            x1 = edgex
            y1 = edgey - 1
            path(plot, prev, prev[vert], x1, y1)
            plot.plot([x1, edgex], [y1, edgey], linewidth = 2, color = 'r')
        if  (vert + maze_cols) == prev[vert] :
            x1 = edgex
            y1 = edgey + 1
            path(plot, prev, prev[vert], x1, y1)
            plot.plot([x1, edgex], [y1, edgey], linewidth = 2, color = 'r')


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
# Implementation of disjoint set forest
# Programmed by Olac Fuentes
# Last modified March 28, 2019

def DisjointSetForest(size):
    return np.zeros(size,dtype=np.int)-1


def dsfToSetList(S):
    #Returns aa list containing the sets encoded in S
    sets = [ [] for i in range(len(S)) ]
    for i in range(len(S)):
        sets[find(S,i)].append(i)
    sets = [x for x in sets if x != []]
    return sets

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): #Find with path compression
    if S[i]<0:
        return i
    r = find_c(S,S[i])
    S[i] = r
    return r

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
        return True
    return False

def union_c(S,i,j):
    # Joins i's tree and j's tree, if they are different
    # Uses path compression
    ri = find_c(S,i)
    rj = find_c(S,j)
    if ri!=rj:
        S[rj] = ri #uses true or false to return for whether the method
        return True #should execute or skip
    return False

def union_by_size(S,i,j):
    # if i is a root, S[i] = -number of elements in tree (set)
    # Makes root of smaller tree point to root of larger tree
    # Uses path compression
    ri = find_c(S,i)
    rj = find_c(S,j)
    if ri!=rj:
        if S[ri]>S[rj]: # j's tree is larger
            S[rj] += S[ri] #uses true or false to return for whether the method
            S[ri] = rj     #should execute or skip
        else:
            S[ri] += S[rj]
            S[rj] = ri

def NumSets(S):
    count =0
    for i in S:
        if i < 0:
            count += 1
    return count
```

```
################################################################################
#                                                    #
#                  """LAB 6 """                           #
#                                                    #
################################################################################

def MazeStandardUnion(S) : #will use the standard union method to create the maze
   while NumSets(S) > 1 : #will execute so long as the number of sets is greater than 1
      d = random.randint(0, len(walls)-1) # d is the random integer we create to remove walls based on that integer
      if union(S, walls[d][0], walls[d][1]) is True : #uses the true or false statement inside of the union
         walls.pop(d)                            #to decide if we execute

def MazeCompression(S) : #will use the Union_c with compression to create the maze
   while NumSets(S) > 1 :
      d = random.randint(0, len(walls)-1)
      if union_c(S, walls[d][0], walls[d][1]) is True :
         walls.pop(d)

################################################################################
#                                                    #
#                  """ LAB 7 """                          #

def MazeWalls(S, m) : #will use the Union_c with compression to create the maze
   wall_pop = [] #this list is used to keep track of the walls that we are popping
               #for the purpose of creating an adjacency list
   while m > 0 :
      d = random.randint(0, len(walls)-1)
      if m > len(walls)-1 : #m is the variable input by the user, if M is greater than the number of walls. jump out
         print('Number of walls you want to remove is greater than the number of walls that exist.')
         return

      if NumSets(S) == 1 : #when there is 1 set in the disjoint set forest start removing walls without union
         wall_pop.append(walls.pop(d))
         m = m - 1

      elif union(S, walls[d][0], walls[d][1]) is True: #keep removing walls with a union to create a set in the dsf
         wall_pop.append(walls.pop(d))
         m = m - 1

   return wall_pop #returns the list to use for creating the adjacency list

#def MazeSearch(M, i) : #
#   for j in M :
#      if j == i :
#         j += 1
#
#      if find(M, i) == find(M, j) :
#         return True
#
#   return False

def MazeAdjacencyList(M, wall_pop) : #used to create an adjacency list
   G = [] #this will be the list that our adjacency list goes into
   for i in range(maze_rows * maze_cols) : #we are creating a list of size rows* cols
      G.append([])                    #full of empty lists ([])

   for i in range(len(wall_pop)) : #we are now populating the empty lists with the list of popped walls we generated on the 
last method
      fi = wall_pop[i][0] #by popping from this list and saving the variable we ensure proper insertion of items
      se = wall_pop[i][1]
      G[fi].append(se) #we now append what was in the second item into the index of the first
      G[se].append(fi) #we now append what was in the first item into the index of the second
      #this is done to ensure that we populate the correct indexes with the correct numbers

   for j in range(len(G)) : #we sort the individual lists inside of our adjancency list for readabililty
      G[j].sort()

   return G
```

```
def BreadthFirstSearch(adj_list) :
    vis = [False] * len(adj_list) #generating an array with False inside the size of the adjacency list
    prev = [-1] * len(adj_list) #Generating an array with -1 to see what was our previous -1 means there was no previous
    Q = queue.Queue() #creating our Queue
    Q.put(0) #we put 0 in our q because we always begin with this and to jump into our while loop
    vis[0] = True #the list with false now at 0 is true because we have visited this according to our q

    while Q.empty() is False : #while our q is not empty we get the next item in the list and save it as v
        v = Q.get()
        for i in adj_list[v] : #for every variable inside of index v in our adjacency list we
            if not vis[i] :   #check to see if it has been visited if not we visit and put true
                vis[i] = True
                prev[i] = v  #now our previous list will show which number was the previous
                Q.put(i) #and put the items of i inside of the Q

    return prev

def DepthFirstSearch(adj_list) : #same comments as BreathFirstSearch method except
    vis = [False] * len(adj_list)
    prev = [-1] * len(adj_list)
    st = []     #we are using a list as a stack
    st.append(0)
    vis[0] = True

    while st :
        v = st.pop()
        for i in adj_list[v] :
            if not vis[i] :
                vis[i] = True
                prev[i] = v
                st.append(i)

    return prev

def DepthFirstSearchRecursively(adj_list, s) :
    vis[s] = True #same concept as the first depthfirstsearch except recursively
    #we created the visitied (vis) list outside of the method. populated the Source(s) with true
    for c in adj_list[s] : #for every element of the source(s) inside of the adjacency list
        if not vis[c] : #is has not been visited
            prev[c] = s #populate our list previous(prev) with the items (this list was also generated outside of the method)
            DepthFirstSearchRecursively(adj_list, c) #recursively call the method thus shortening our problem

    return prev #once done. return the list prev populated

def in_degreeAL(G, v) :
    count = 0
    if G == [] :
        print('List is empty')
        return
    if v > len(G) :
        print(v, ' is not a valid vertice.')
        return
    for i in range(len(G)) :
        for j in G[i] :
            if j == v:
                count += 1
    return count

def count_edges(G) :
    co = 0
    if G == [] :
        print('List is empty')
        return co
    for i in range(len(G)) :
        if G[i] is not [] :
            co += len(G[i])
    return co

def reverse_edges(G) :
    if G == [] :
```

```
        print('Nothing to reverse')
        return
    for i in range(len(G)) :
        if len(G[i]) < 2 :
            print('This edge list is incorrect, one edge only has 1 parameter.')
            return
        G[i] = [G[i][1], G[i][0]]
    G.sort()
    print(G)


###############################################################################
#                                                        #
#                 """ MAIN """                                  #                                                     #
plt.close("all")
maze_rows = 5
maze_cols = 5

walls = wall_list(maze_rows,maze_cols)
draw_maze(walls,maze_rows,maze_cols,cell_nums=True)
TimeStart = datetime.now()
M = DisjointSetForest(maze_rows * maze_cols)
print('The Number of cells in this maze are: ', maze_rows * maze_cols)

print('Please enter the number of walls you would like to remove.')
m = int(input())
if m > ((maze_rows * maze_cols) - 1) :
    print('There is at least one path from source to destination.')
if m == ((maze_rows * maze_cols) - 1) :
    print('There is a unique path from source to destination.')
if m < ((maze_rows * maze_cols) - 1) :
    print('A path from source to destination is not guaranteed to exist.')
wall_pop = MazeWalls(M, m)
adj_list = MazeAdjacencyList(M, wall_pop)
print('Adjacency List: ', adj_list)
az = draw_maze(walls,maze_rows,maze_cols)
plt.show()
print('Disjoint Set Forest Maze: ', M)
bsf = BreadthFirstSearch(adj_list)
print('Breadth First Search: ', bsf)
dfs = DepthFirstSearch(adj_list)
print('Depth First Search: ', dfs)
vis = [False] * len(adj_list)
prev = [-1] * len(adj_list)
dfsr = DepthFirstSearchRecursively(adj_list, 0)
print('Depth First Search Recursively: ', dfsr)
path(az, bsf, (maze_rows * maze_cols)-1, maze_cols-.5, maze_rows-.5)
TimeEnd = datetime.now()
print('Time to build and print maze: ', TimeEnd - TimeStart, ' seconds.')
```

ACADEMIC CERTIFICATION

I "Sergio Gramer" 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.

Signed: 04/15/2019

Sergio Gramer