**Lab Objective**

For this lab, I’m given a program that creates a maze, but in the program, it outputs 2 figures which are the maze before the removing of the walls and another figure of the maze with the removed walls. For this lab I am instructed into modifying a program that creates a mazeinto removing the walls of the maze randomly while still making a path from the 0 square in the original figure to the 149 square. Also, when doing this I need to use disjoint set forests and the functions that it contains.

**Proposed Solution**

For this part of the program, I first had to understand how the maze program worked before doing anything. After running the program various times and understanding the logic behind it, I found out that I only need to modify the remove function, because when I remove a wall, the wall being removed must be rand while still making a path to the 0 square to the 149 square.

**Implementation**

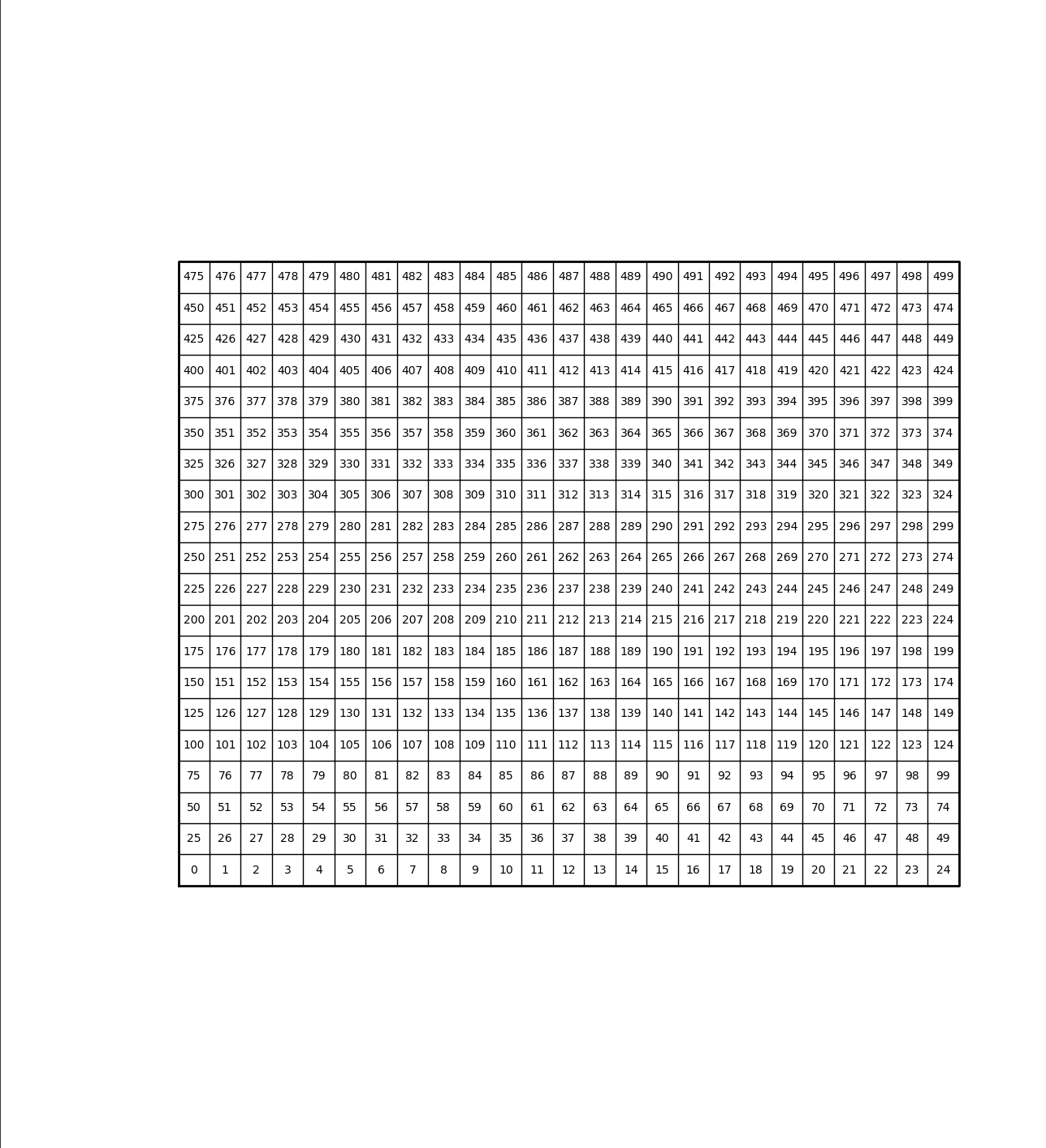
To begin my Implementation, I first needed to create a method where I am able to count the amount of sets I have in my disjoint set forest and to do that I need to pass the parameters of my forest to the method. In method to count the amount of sets, I first made a counter that starts at 0 which will count the amount of sets I have. Then I made a for loop that traverses the disjoint set forest by using the length of it. Inside the for loop I checked if my disjoint set forest at an index of “i” is less than 0, then it would add 1 to the count variable and outside the if and for loop I returned the count.

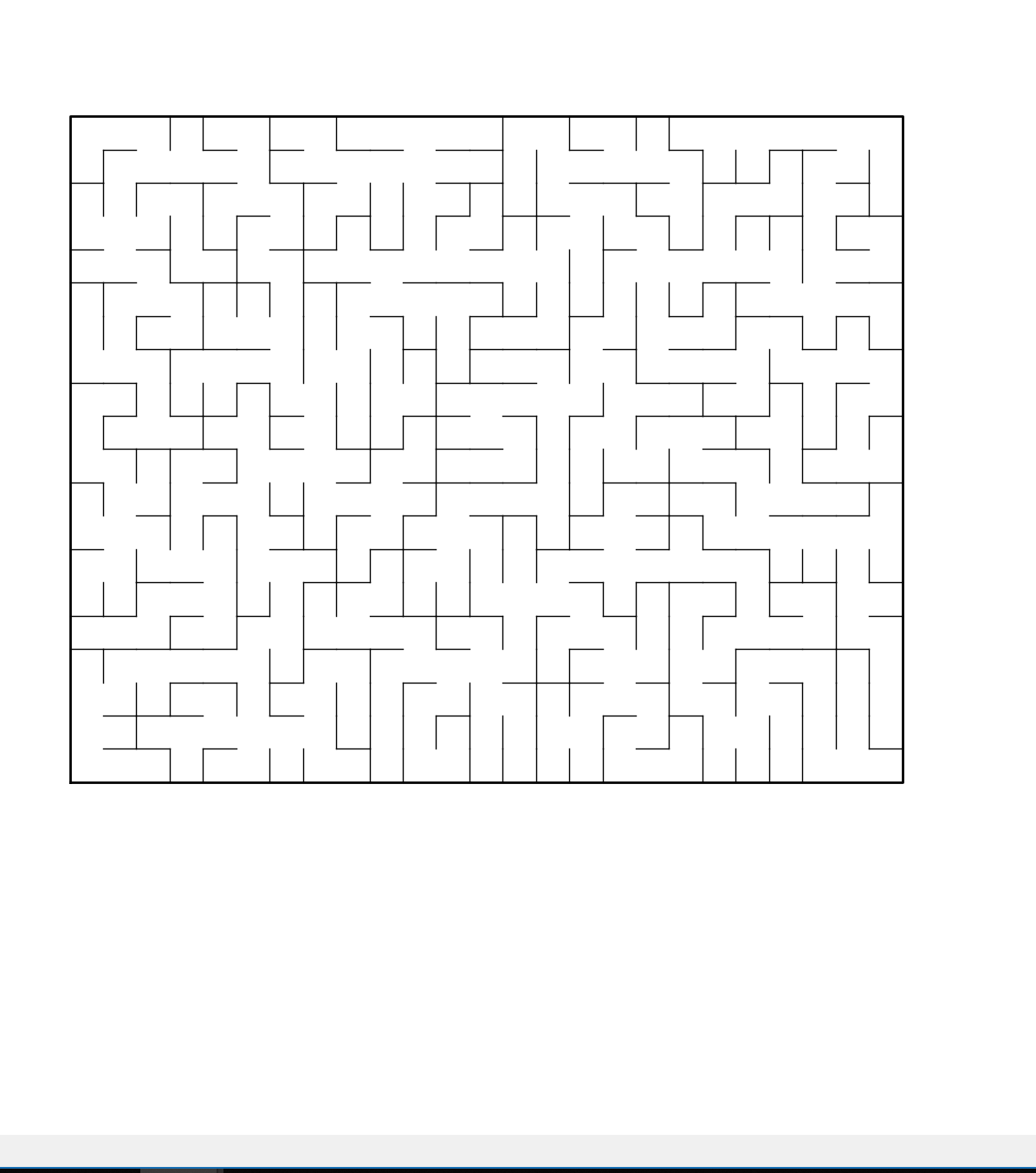
After making the amount of sets method, now I am able to create a new remove method, but I also need to make another remove method that will use path compression. When making the signature of the method, the parameters that I passed where the disjoint set forest, the maze\_walls variable (comes from the method wall\_list), and the number of sets. Inside the method, I first started by making a while that checks that the number of sets is greater than 1. Inside the while loop, I made a variable called w that will store the random selected wall from the maze. Then I need to make a variable to store the index of the maze\_walls that was selected to be removed randomly. After making the storage variables, I then needed to make an if statement that checks, if cells c1 and c2 belong to different sets, remove w and join c1’s set and c2’s set otherwise do nothing. If they’re not in the same set, then inside my if statement I need to pop the maze wall that was randomly selected then combine both sets by the use of the standard union function (when making the remove with compression, then I used the union by size method which combines both sets by path compression). After making my unions, I then need to be able to exit the while loop, and to do that I need to decrease the number of sets by 1; outside the loop I then returned w.

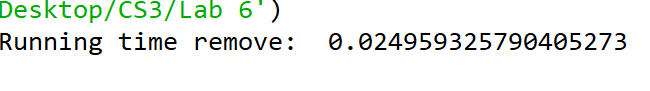
I also need to compare the runtimes of the remove method that uses the standard union function with the other remove method that uses the union by size method which has path compression. When comparing the two methods I needed to implement separate time imports before I called the methods, then I had to call the time import again after the calling of the methods. Before calling the methods, I get the start time and after calling the method I get the end time. To show the runtimes I subtracted the end time minus the start time, also if the runtime shows up as 0, its because the dimensions of the maze are too small.

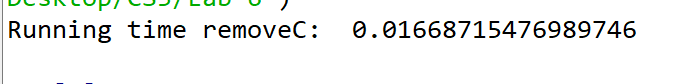
**Tracing**

To trace this program, I’m going to be using the dimensions for my maze rows to be the size 20 and the size of the maze columns will be of size 25. When starting this, the variable declared as maze\_walls will call the method wall\_list while passing the maze rows and maze columns into the method. In the wall list method, I pass both the maze rows and maze column, inside there is an empty list called w which will be used later. After there is a for loop that traverses the rows of the maze and then a second loop that traverses the for loop that traverses the columns in the maze. Inside both for loops the variable cell will store the index of the maze columns and add it to the product of the index of the rows by the maze column. Still inside the for loops, there is an if statement that if the index of the columns is not equal to the final index of the maze columns then the list w appends the cell variable and the increment of cell plus one. In another if statement, there is an if statement that if the index of the rows is not equal to the final index of the maze rows then the list w appends the cell variable and the cell variable plus the maze columns; once we exit the loop then we return w for the list of walls that will be used.

Now that maze\_walls has the list of walls then, the code will call the draw\_maze method which passes the maze walls, maze rows, maze columns; and what that method does is that it basically draws the maze from using the wall list and the dimensions given. After drawing the maze, I then made a variable called “S” that will store the new disjoint set forest which uses the product of the size of rows by the size of the columns. Now with the disjoint set forest made, I then made a variable called numSets that will store the total amount of sets in “S”. Assuming the method works, I then proceed to starting the start time to count for the standard remove method and when calling the remove function, I pass the disjoint set forest, maze\_walls, and the numSets. In the remove method, I iterate through the set by using the number of sets until its less than 1. In the while loop the variable w will be storing all the walls that where removed by random and then the “i” variable will use the index of maze walls and will point at the wall selected. Once “i” has the position of which wall will be removed, the next if statement will call the find method which passes the DSF and the index of w at the 0 index and then it checks if its not equal to the find of the DSF at the index 1. If those two sets are not in the same DSF then the if, pops the maze wall that is at the I position and the combines both sets with the standard union function in order to close the missing part of the sets. Then we decrease the number of sets by one and once we reach 0 for the number of sets then the method returns w that has the walls after the removal. After the method end then the end time ends counting and with the subtraction of end time – start time should return the final runtime. Also the same process happens with the remove method with path compression only that it should take less time to complete the method because path compression will basically use union by size which compares the size of the two sets and the combines directly to the largest set, which removing with compression should be quicker than the standard compression.







**Academic dishonesty**

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act. Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

**APPENDIX**

# -\*- coding: utf-8 -\*-"""Created on Wed Apr 10 22:34:58 2019@author: Alexis NavarroCS 2302Olac FuentesPurpose:The purpose of this lab is to be able to work with disjoint set forests while applying it to create a maze. I have to be able to remove random walls in order to create a path from the square 0 to the square 149 (Shown in figure 1). To do this I need to get the amount of sets which will be used to remove a wall, make a disjoint set forest by using the rows and columns, Then I need to make the remove function after all the other parts are made. """ import matplotlib.pyplot as pltimport numpy as npimport random#import datetime # need to use datetime instead of time because when using time, I would always get 0 for my running time when the size of my maze was too small # datetime is more precise for smaller mazes when being testedimport time #GIVEN FUNCTIONS (PROVIDED BY CS 2302) 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 findC(S,i): if S[i]<0: return i r = findC(S,S[i]) S[i]=r return r #combines the two set by using their size as referencedef union\_by\_Size(S,i,j): ri = findC(S,i) rj = findC(S,j) if ri!=rj: # Do nothing if i and j belong to the same set if S[ri]>S[rj]: # j's tree is larger S[rj] += S[ri] S[ri] = rj else: S[ri] += S[rj] S[rj] = ri #------------------------------------------------------------------------------#MADE METHOD/FUNCTION REQUIRED TO ACCOMPLISH THE LAB #method to count the amount of sets in the DSFdef setAmount(S): count=0 for i in range(len(S)): if S[i]<0: count +=1 return count #method to delete random parts of the walldef remove(S,maze\_walls,numSets): while numSets > 1: w = random.choice(maze\_walls)# w gets the wall that was randomly selected i=maze\_walls.index(w)#gets the position where we chose the wall to delete if find(S,w[0]) != find(S,w[1]): maze\_walls.pop(i) #deletes the wall union(S,w[0],w[1])# combines the walls after the deletion numSets-=1 return w def removeC(S,maze\_walls,numSets): while numSets > 1: w = random.choice(maze\_walls)# w gets the wall that was randomly selected i=maze\_walls.index(w)#gets the position where we chose the wall to delete if find(S,w[0]) != find(S,w[1]): maze\_walls.pop(i) #deletes the wall union\_by\_Size(S,w[0],w[1])# combines the walls after the deletion numSets-=1 return w#------------------------------------------------------------------------------#METHOD TO DRAW THE MAZE (PROVIDED BY THE CS2302)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 wplt.close("all") #------------------------------------------------------------------------------#MAIN #size of rows and columns (Dimensions of the maze)#Various sizes to test#maze\_rows = 10 use datetime for these dimensions since they are smaller#maze\_cols = 15 maze\_rows = 20 # for bigger maze dimensions use time importmaze\_cols = 25 #maze\_rows = 40#maze\_cols = 45 maze\_walls = wall\_list(maze\_rows,maze\_cols)#Gets the list of walls in the maze draw\_maze(maze\_walls,maze\_rows,maze\_cols,cell\_nums=True) #calls the draw maze method and makes the complete maze without deletion S = DisjointSetForest(maze\_rows\*maze\_cols)# makes the new DSF by combining the rows and columns numSets=setAmount(S) # gets the amount of sets in the maze #This part will give the running time for the remove function that uses standard Union (Only one can be uncommented at a time, if not the program will crash)start\_Time=time.time()remove(S,maze\_walls,numSets)# calls the method to remove parts of the wall with regular unionend\_Time=time.time()print('Running time remove: ',(end\_Time-start\_Time)) #Uncomment this in order to get the running time with the remove function that uses path compression (Only one can be uncommented at a time, if not the program will crash)'''start\_Time2=time.time()removeC(S,maze\_walls,numSets)end\_Time2=time.time()print('Running time removeC: ',(end\_Time2-start\_Time2))''' draw\_maze(maze\_walls,maze\_rows,maze\_cols)#draws the walls after the deletion of random maze walls