Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: April 15, 2019

Instructor: Olac Fuentes

Assignment: Lab 6 DSF

TA: Anindita Nath & Maliheh Zaragan

**Introduction:**

In python, you can use a data structure called a Disjoint Set Forest that will help keep track of a set of elements portioned throughout a number of subsets in a particular program. One example that uses a disjoint set forest is that of a maze used in puzzles, as it creates a path to be made for this maze so it can be entered one way and leave on another, as usual mazes do. The big difference is that I need to find a way to create the maze randomly by letting M (number of rows) and N (number of columns) must belong in separate sets so that it forms the new path. I must also do this using two methods, one using standard union, and the other using union by compression.

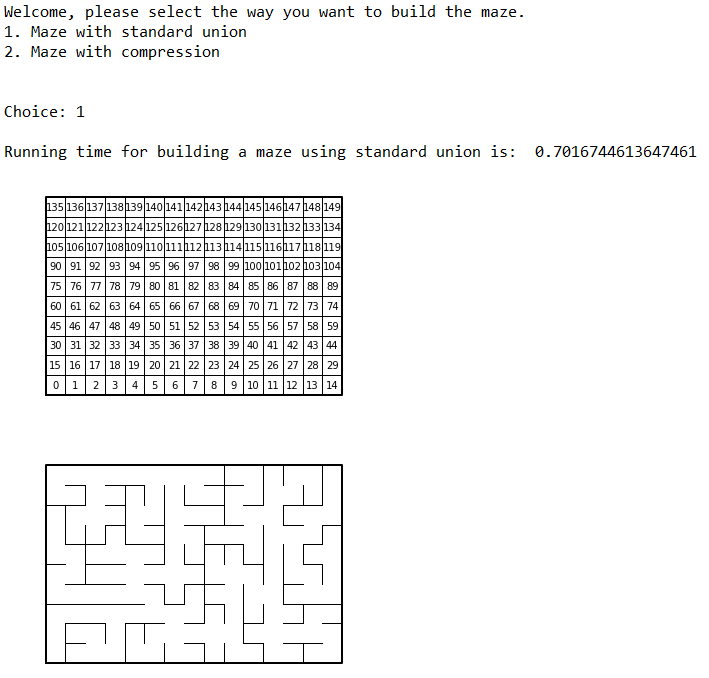
**Proposed solution design and implementation:**

1. Standard Union: For this method, I need to find a way to utilize the methods given to me by the professor to formulate a way to not only create a good representation of the maze, but to also find a distinct solution using random paths and using standard union. By definition, when using standard union, I should be able to compare two numbers so that I can compared them both, these two numbers are the values of the walls, when they are not equal to each other the value of the second number equals that of the first. My proposed solution is to find a way to fully utilize the method union to achieve this, so long as walls do not match each other in any way, that is, when there is value present, that value should not repeat itself at all anymore later on in the lab.
2. Compression: For this method, I would need to repeat the same steps as mentioned above, however I would only need to utilize the method found within the code given by the professor called union\_by\_size as that will guarantee that I find the value necessary to complete the same procedure of finding walls, but by methods of compression.

**Experimental results**:

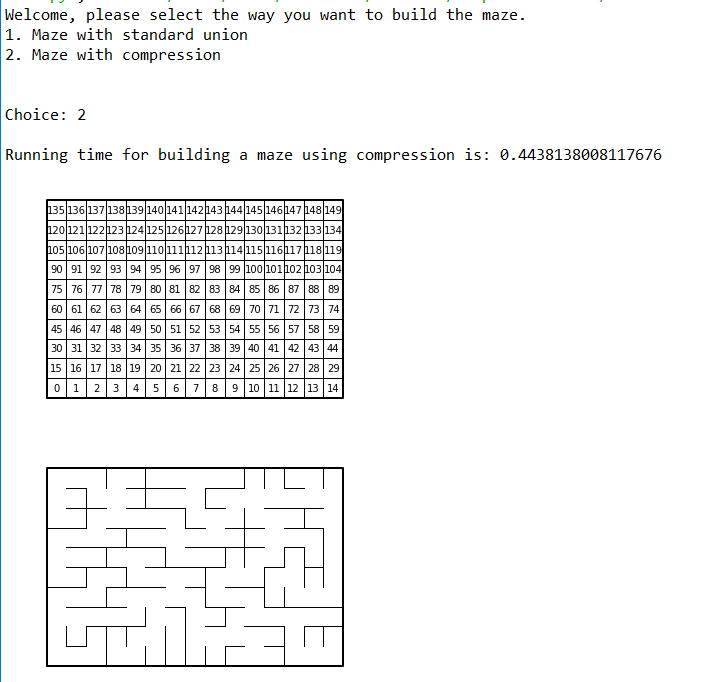
1. Standard Union: The way that I constructed my method was that it creates the grid values first so that my actual maze corresponds to that of the necessary space needed to that of the final product. Then checking that the totals sets set in my graph are greater than one, I start removing walls found throughout the maze, as well as making sure that two sets of walls are identical to each other. If true, then it will unite walls until the criteria has been given. Afterwards, this new maze will be sent to be drawn and printed so that the user sees what the end result was.

The results of creating a random maze using standard union is shown below:



1. Compression: The way I constructed the maze was almost identical to that of the standard union except that I used the method union\_c and union\_by\_size given by the professor. The way it works is that if it finds that the value of two numbers in the maze do not equal by using find\_c, then it will unite by size instead of the usual union given in the previous method. But other than that, everything else in the process has been untouched.

The results of creating a random maze using compression is shown below:



**Running times Table**

|  |  |  |
| --- | --- | --- |
| Attempt | Standard Union | Compression |
| 1 | 0.7016 sec | 0.4438 sec |
| 2 | 0.8092 sec | 0.5614 sec |
| 3 | 0.5066 sec | 0.5854 sec |

As seen above, the method with the overall best time was the compression method compared to that of the standard union method. This is partly due to the fact that the compression method uses union by size instead of union, which if applicable, makes the root of smaller tree point to root of larger tree until a path is constructed, whilst the standard union continues nonstop until it finds the root of the tree and returns the path of that given tree.

**Conclusions**:

With this lab, I was able to learn to code better using the Python language, including using algorithms to create and modify disjoint set forest to create a maze with a singular unique solution. I was also able to learn to solve different problems by using standard union and compression throughout my lab, including to determine the placement of each wall found throughout the mazes.

**Appendix :**

**lab6.py**

"""

Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: April 13, 2019

Instructor: Olac Fuentes

Assingment: Lab 6 Disjoint Set Forests

TA: Anindita Nath & Maliheh Zaragan

Purpose: to implement both standard union and compression techniques

to use a disjoint set forest to build a maze.

"""

#Imports various tools to help us calculate the hash tables to be used in this lab

import matplotlib.pyplot as plt

import numpy as np

import random

import time

#Method that creates a Disjoint Set Forest

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 totalSets(S):

# Gets the total number of sets

setNum = 0

#For i in range of the length of S, setNum is added by 1

for i in range(len(S)):

if S[i] < 0:

setNum += 1

#Returns setNum

return setNum

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 wallList(maze\_rows, maze\_cols):

# Creates a list with all the walls in the maze called w

w = []

#For i and c in range of the rows and columns

for r in range(maze\_rows):

for c in range(maze\_cols):

#C and r times the columns create the cell

cell = c + r\*maze\_cols

#If the value of c does not equal columns -1, it will append on column of cell + 1

if c != maze\_cols-1:

w.append([cell, cell+1])

#If the value of c does not equal rows -1, it will append on column of cell + columns

if r != maze\_rows-1:

w.append([cell, cell+maze\_cols])

#Returns w

return w

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

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

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]

S[ri] = rj

else:

S[ri] += S[rj]

S[rj] = ri

#Method that sets the requirements to build a standard union maze

def su\_maze():

#Draws the initial maze with only the number of cells so that the user knows whicch was the total amount

draw\_maze(walls, rows, columns, cellNums=True)

#While the total number of sets is greater than 1

while totalSets(S) > 1:

#Removes a random wall

remove\_wall = random.randint(0,len(walls)-1)

#If the walls in S from 0 does not equal to the walls in S from 1, then it will unite terms

if find(S,(walls[remove\_wall])[0]) != find(S, (walls[remove\_wall])[1]):

union(S,(walls[remove\_wall])[0], (walls[remove\_wall])[1])

walls.pop(remove\_wall)

#Method that sets the requirements to build a compression maze

def c\_maze():

#Draws the initial maze with only the number of cells so that the user knows whicch was the total amount

draw\_maze(walls, rows, columns, cellNums=True)

#While the total number of sets is greater than 1

while totalSets(S) > 1:

#Removes a random wall

remove\_wall = random.randint(0,len(walls)-1)

#If the walls in S from 0 does not equal to the walls in S from 1, then it will unite by size

if find\_c(S,(walls[remove\_wall])[0]) != find\_c(S, (walls[remove\_wall])[1]):

union\_by\_size(S, (walls[remove\_wall])[0], (walls[remove\_wall])[1])

walls.pop(remove\_wall)

#Method that draws the maze from either union or compression methods

def draw\_maze(walls, rows, columns, cellNums=False):

# Plots the maze

fig, ax = plt.subplots()

#For i in walls

for i in walls:

# If the amount of i[1] - i[0] equals 1, it will create a vertical wall

if i[1]-i[0] == 1:

x0 = (i[1] % columns)

x1 = x0

y0 = (i[1] // columns)

y1 = y0 + 1

# If the amount of i[1] - i[0] equals 1, it will create a horizontal wall

else:

x0 = (i[0] % columns)

x1 = x0+1

y0 = (i[1] // columns)

y1 = y0

#Used to plot the frame

ax.plot([x0, x1], [y0, y1], linewidth=1, color='k')

#Sets a new variable called sx and sy that ues the values of the rows and columns to plot the lines

sx = columns

sy = rows

ax.plot([0, 0, sx, sx, 0], [0, sy, sy, 0, 0], linewidth=2, color='k')

#If the number of cells is true

if cellNums:

#For r in range of the number of rows

for r in range(rows):

#For c in range of the number of columns

for c in range(columns):

#The value of cell increases by c plus r times the number of columns

cell = c + r\*columns

ax.text((c+.5), (r+.5), str(cell), size=10,ha="center", va="center")

#Sets the axis to be off

ax.axis('off')

#Sets the axis aspect ratio to 1

ax.set\_aspect(1.0)

#Sets the total number of rows and columns that the maze contains inside of the maze

rows = 10

columns = 15

#Creates a new disjoint set forest based on the number of columns and rows

S = DisjointSetForest(rows \* columns)

#Finds the Number of sets defined in the maze

Num\_Of\_Sets = totalSets(S)

#Finds the number of walls inside of the maze

walls = wallList(rows,columns)

#Asks the user to choose a binary search tree or hash table with chaining

print("Welcome, please select the way you want to build the maze.")

print("1. Maze with standard union")

print("2. Maze with compression")

print()

#Sets a variable 'x' to be made for the input section of the program

x = int(input('Choice: '))

print()

sets = dsfToSetList(S)

#If the user selects 1, the program will build a maze using standard union

if x == 1:

#Starts a timer to be used later for running time comparisons

start1 = time.time()

su\_maze()

#Draws the mazed made by standard union

draw\_maze(walls, rows, columns)

elapsed\_time\_union = time.time() - start1

#Prints the running time for the su\_maze method

print('Running time for building a maze using standard union is: ', elapsed\_time\_union)

#If the user selects 2, the program will build a maze using compression

elif x == 2:

#Starts a timer to be used later for running time comparisons

start2 = time.time()

c\_maze()

#Draws the mazed made by standard union

draw\_maze(walls, rows, columns)

elapsed\_time\_compression = time.time() - start2

#Prints the running time for the c\_maze method

print('Running time for building a maze using compression is:', elapsed\_time\_compression)

#If the user inputs anything other than 1 or two, the following error message will be displayed

else:

print('Incorrect input! Try again.')

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 provide inappropriate assistance to any student in the class.

