Course: CS 2302

Author: Kimberly Morales

Assignment: Lab 6

Instructor: Olac Fuentes

TA(s): Anindita Nath, Maliheh Zargaran

Date: 4/17/19

# Introduction

Mazes are often composed of many simples’ paths so that there is at least one solution. When generating a random maze, it can easily be done by just picking a random wall and knocking it down. However, this does not guarantee a solution each time. To generate random mazes with simple paths it requires knocking down walls and to have each cell have a simple path to each cell. This requires using the disjoint set forest(dsf) class and utilizing the union and find methods. Then to see the difference of running times, use path compression and standard union.

# Proposed Solution Design and Implementation

When generating a random maze with any simple path between two cells, I realized that I would have to pick a random wall to knock down and to see if they belong to the same set. Knowing from what I learned from class, a disjoint set forest(dsf) could easily find if two cells had a path. I mostly saw the dsf as a link between cells, so I would have to find the coordinates of each cell and if they belong to different sets then that means there is no path between them. In this case, we delete the wall and combine them so that there is a path created. A wall is deleted so that it is shown graphically, and the union decreases the number of sets. My approach to union by size was similar but with functions changed to having find by path compression and union by size. When inputting how big the maze is, I made it a user input for easier testing.

Check\_maze

This method uses the standard union and find methods. A loop is used to count the number of sets until it is less than 1. The reasoning behind this is because a union decreases the amount of sets since it is combing them. Meanwhile within the loop, a random wall is picked with the python randomizer since we are trying to generate an interesting random maze. With the random wall chosen, the first and second elements, which are the cells encompassing of the random wall, then have their roots found. If these two cells end up being from different sets then that means that there is no path connecting them, so we delete a wall to show it graphically and union them to make them in the same set. Once there is one set left then we return the new list.

Check\_maze\_uc

This method uses union by size and find by path compression methods. A loop is used to count the number of sets until it is less than 1. The reasoning behind this is because a union decreases the amount of sets since it is combing them. Meanwhile within the loop, a random wall is picked with the python randomizer since we are trying to generate an interesting random maze. With the random wall chosen, the first and second elements, which are the cells encompassing of the random wall, then have their roots found. If these two cells end up being from different sets then that means that there is no path connecting them, so we delete a wall to show it graphically and union them to make them in the same set. Once there is one set left then we return the new list.

Num\_sets

Number of sets is a helper method to keep track of the number of sets within the set since a union decreases the amount of sets. We add to the number of sets count if an element is equal to the root.

Main I/O

User input is given for the maze columns and maze rows so to make it easier to test bigger inputs. Walls are created from wall list which creates walls based on the size of columns \* rows. Then an initial maze is drawn with cell numbers and with no walls deleted. Then a dsf is created from the size of rows \* columns. Then we use standard union check size function and check maze with union by size. We time how long it takes for each function and display it on the screen. Finally, the final maze is drawn from the edited walls list.

# Experimental Results

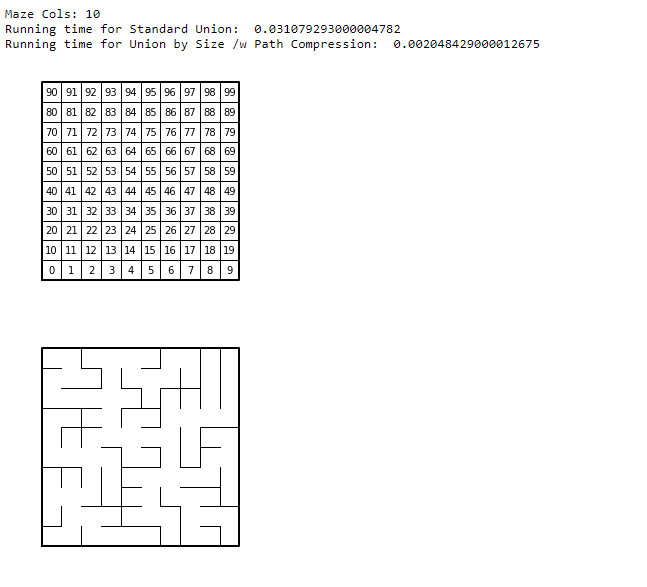
Running time calculations done with input sizes that create a square maze so to maintain consistency in times.

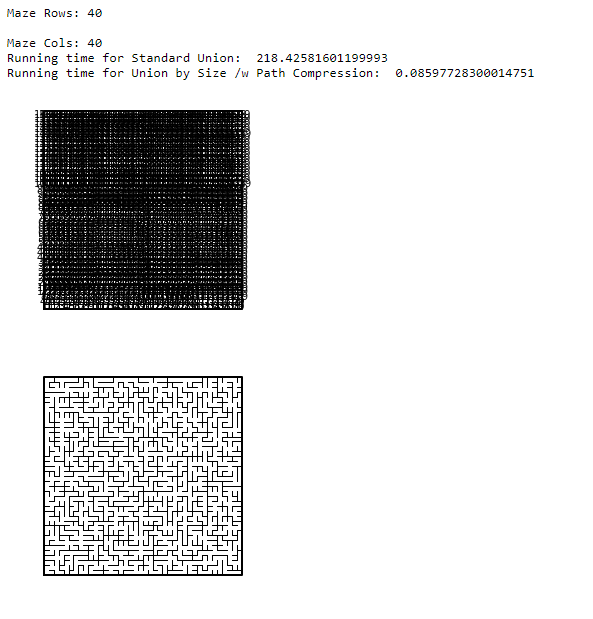
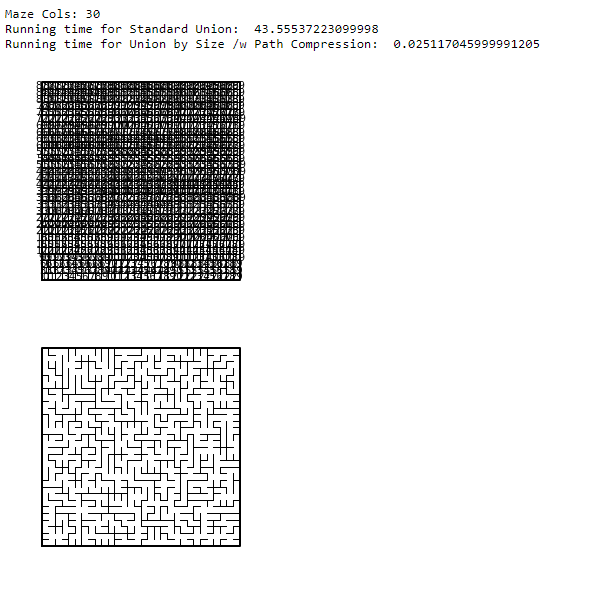
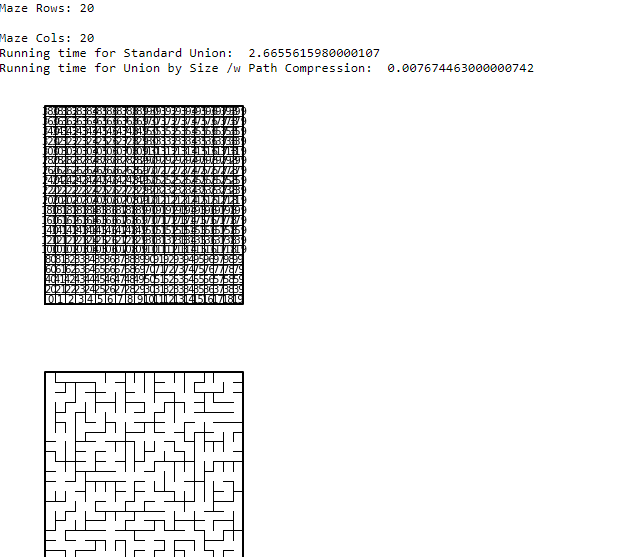
Standard Union

|  |  |
| --- | --- |
| Input(Nrows, Ncols) | Running Time |
| Rows =10  Cols =10 | 0.031079293000004782 |
| Rows = 20  Cols = 20 | 2.6655615980000107 |
| Rows = 30  Cols = 30 | 43.55537223099998 |
| Rows = 40  Cols = 40 | 218.42581601199993 |

Union by Size with Path Compression

|  |  |
| --- | --- |
| Input(Nrows, Ncols) | Running Time |
| Rows =10  Cols =10 | 0.002048429000012675 |
| Rows =20  Cols =20 | 0.007674463000000742 |
| Rows = 30  Cols = 30 | 0.025117045999991205 |
| Rows = 40  Cols = 40 | 0.08597728300014751 |

Screenshots



# Conclusions

To conclude, I never thought of a maze to be a dsf since I always assumed that dsf’s were merely abstract and used to partition. A maze is easily a set since it has specific paths between each cell. It was fun to generate mazes this way and it is interesting to see how there is still a solution for each of these mazes generated. I also saw the big difference in running times for the two different union operations since standard union would often make the program take forever to generate a maze. This was easily seen by the chart generated from the running times. When it comes to efficiency then I would use union by size if I had a more intensive project.

# Appendix

"""

Course: CS 2302 [MW 1:30-2:50]

Author: Kimberly Morales

Assignment: Lab 6

Instructor: Olac Fuentes

TA(s): Anindita Nath , Maliheh Zargaran

Date: 4/11/2019

Date of last modification: 4/11/2019

Purpose of program:

To generate random mazes by knocking down walls and to have each cell have a simple path to each cell.

This requires using the disjoint set forest(dsf) class and utilizing the union and find methods.

Then to see the difference running times, use path compression and standard union.

"""

import matplotlib.pyplot as plt

from scipy import interpolate

import numpy as np

import random

import timeit

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

#DISJOINT SET FOREST CLASS

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

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

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

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

#MAZE METHODS

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

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

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

#LAB METHODS

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

#Checks to see if two cells belong to different sets and removes a wall

#Uses standard union

def check\_maze(S,w,mc,mr):

while num\_sets(S) > 1:

dw = random.randint(0,len(w)-1) #dw: wall to remove

c1 = find(S,w[dw][0]) #c1: cell 1

c2 = find(S,w[dw][1]) #c2: cell 2

#If the two cells are from different cells than remove a wall so to allow a path and combine them

if c1 != c2:

del w[dw]

union(S,c1,c2)

return w

#Checks to see if each cell has a simple path to another cell

#Uses union by size and path compression

def check\_maze\_uc(S,w,mc,mr):

while num\_sets(S) > 1:

dw = random.randint(0,len(w)-1)

c1 = find\_c(S,w[dw][0])

c2 = find\_c(S,w[dw][1])

if c1 != c2:

del w[dw]

union\_by\_size(S,c1,c2)

return w

#Counts the total number of sets

def num\_sets(S):

count = 0

#Checks each set to see if they have similar parents

for i in range(len(S)):

ri = find(S,i)

#If the element equal to the root then it is in the same set

if i == ri:

count+=1

return count

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

#MAIN

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

plt.close("all")

maze\_rows = int(input('Maze Rows: '))

maze\_cols = int(input('Maze Cols: '))

walls = wall\_list(maze\_rows,maze\_cols)

#Draws initial maze with numbers

draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True)

#Create dsf with the size of the maze(rows \* cols)

S = DisjointSetForest(maze\_rows\*maze\_cols)

#Time creation of maze with standard union

start = timeit.default\_timer()

check\_maze(S,walls,maze\_cols,maze\_rows)

stop = timeit.default\_timer()

print('Running time for Standard Union: ', stop - start)

#Time creation of maze with union by size

startC = timeit.default\_timer()

check\_maze\_uc(S,walls,maze\_cols,maze\_rows)

stopC = timeit.default\_timer()

print('Running time for Union by Size /w Path Compression: ', stopC - startC)

#Draw final maze

draw\_maze(walls,maze\_rows,maze\_cols)

# Academic Honesty

“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.”

Name: Kimberly Morales



Signature: