Rigoberto Quiroz

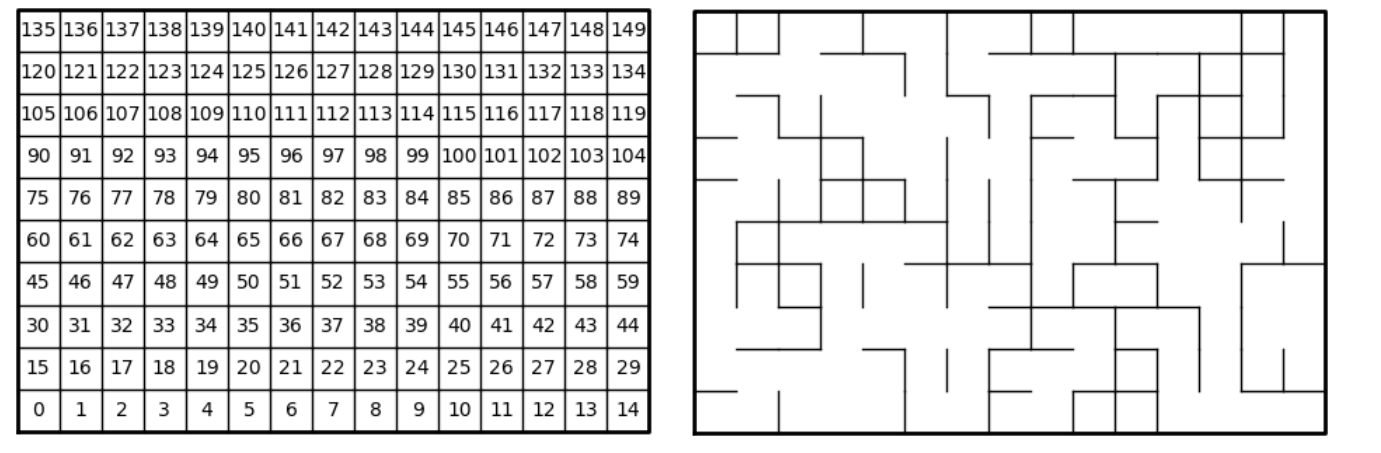
4/13/19

Lab6 Report

CS2302 1:30 PM – 2:50 PM

Description:

For this lab we had to create a maze using the Disjoint set data Structure. The way the initial maze was by created was by setting a x and y maze. Depending on the number of squares inside the cube, lists would be created. For example, a list would contain [1,2] [0,1], etc. This list contains regions that are next to each other. Once we have the list of all the regions we will start deleting the walls that are blocking each region, creating a maze that can get from point A to point B with a single path (no 2 paths can be used to reach the same point). Initial maze is shown below.



The way I ended up solving this lab was by creating a disjoint set data structure (created according to the number of rows and columns). Once created, we will check how many roots we have in our disjoint set, as long as we have more than 1 set we will select random list that contains each region that are next to each other. Once we have selected a random list we will create a union between point A and point B, deleting the wall that separates each region. Once deleted we will pop that list, so that we do not select it again. We will repeat this process until we only have 1 set in our disjoint set. In the case that we select an list that have the same root, we will skip that list and advance to the next list.

6.1 User Input:

This section of the code will ask the user which type of implementation they want to use in their Disjoint set, if they choose 1, the program will create a disjoint set using the standard Union, indexes in the list point to their parents and or roots. If the user chooses 2 then we will implement the compression mod, which each time we make a union, the indexes will point to their root, and since we are doing Union by size, our roots will contain the number of values elements that are in the same set.

6.2 Standard Union:

We will receive two indexes that are giving by the random selected list. We will check if they belong in the same set, by checking their roots, if they in different set, then our second index will point to our first index. If they are in the same set then we will return False, which will tell us to skip that list, and we will not remove the walls.

6.3 Standard Find:

This method will receive an index from the disjoint set, if the value that is inside the index is less then 0 we will return the index we received. Otherwise we will go to the element that that is inside the index, and check if its less than 0, if it is not the we will repeat the same process until we find a element that is less than 0 (means that they are the root of the set).

6.4 Number of Sets:

Since we want to end up with on single set in our disjoint set, we will check each element in our list, if we see a -1 then we will add to our counter. Once we have finished looking through the list we will return the number of sets we have found.

6.5 Union by Size:

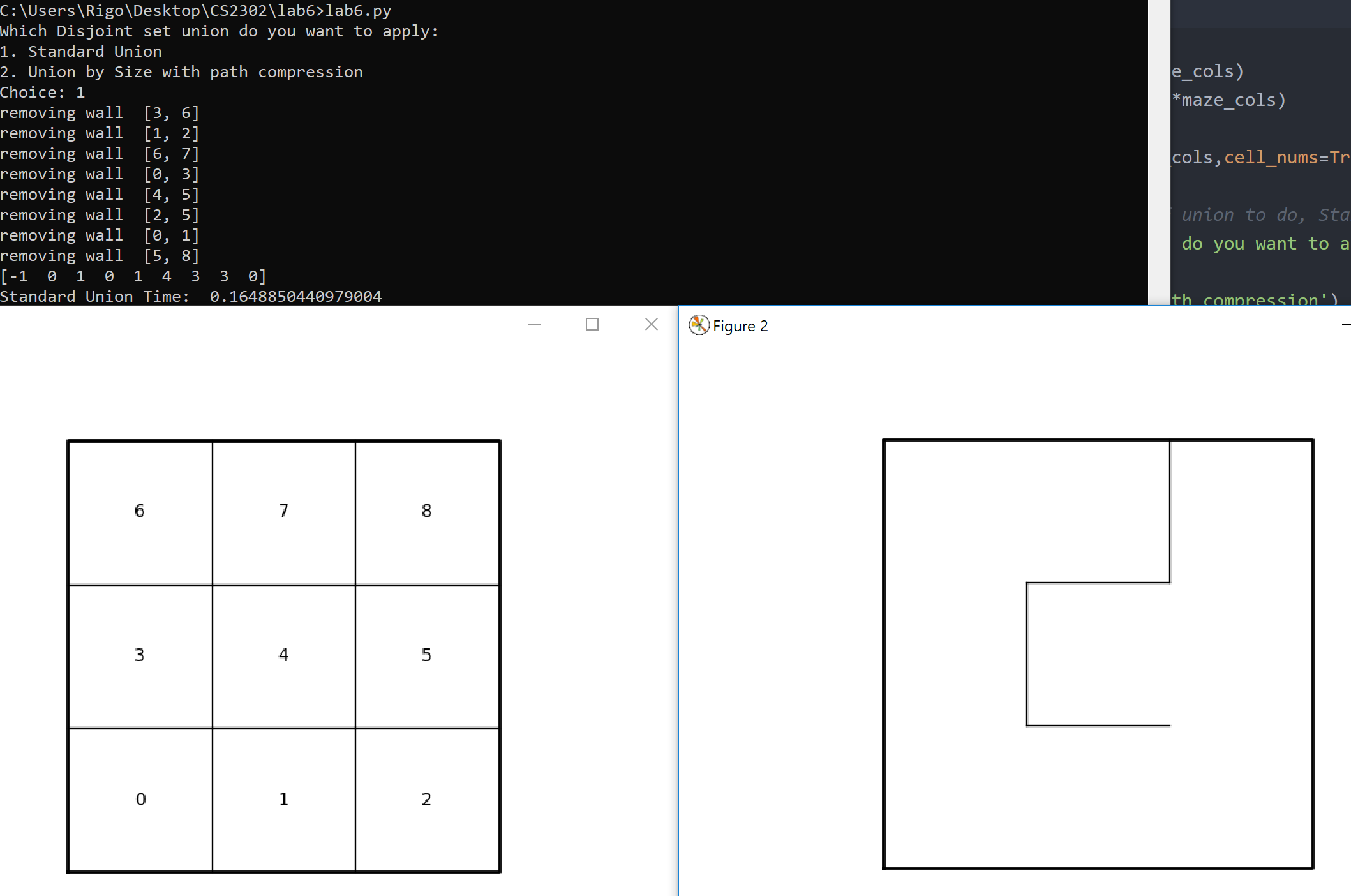
This method will work very similar to the standard union, the difference would be that once we know that both values are not in the set, we will check which root has a smaller value, which ever root has the higher length we will join the smaller set with the greater length. Then we will add the value of the smaller root to the bigger root, updating its length.

6.6 Compression Find:

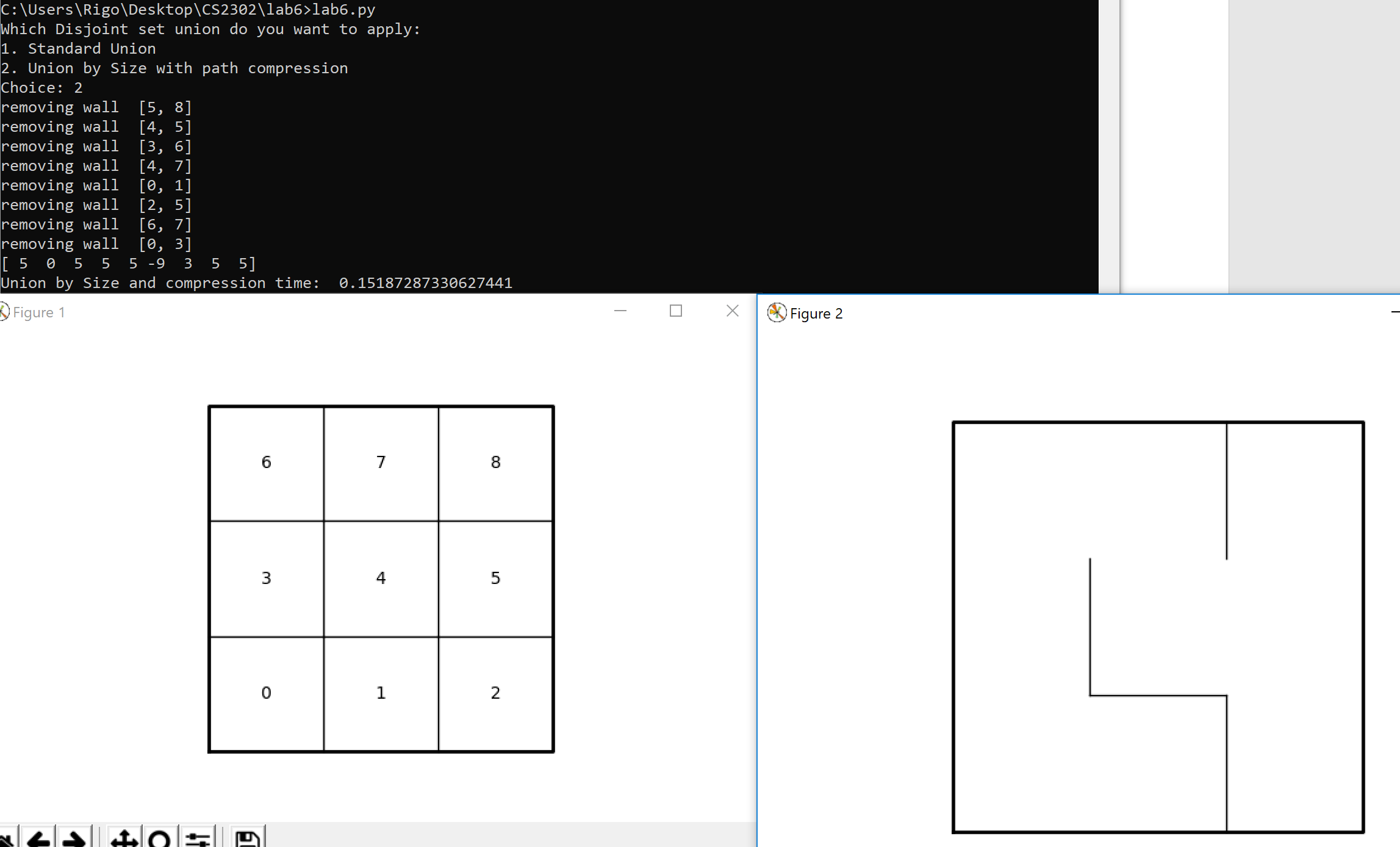
This method will work in a similar way to our standard find, we will find the root, if the index has a parent that is not its root then we will change its parent to its root. Since we are doing this in a recursive manner we will do this process for all the elements that are leading to values that are not their roots.

Input: 3x3:

Output (Standard):

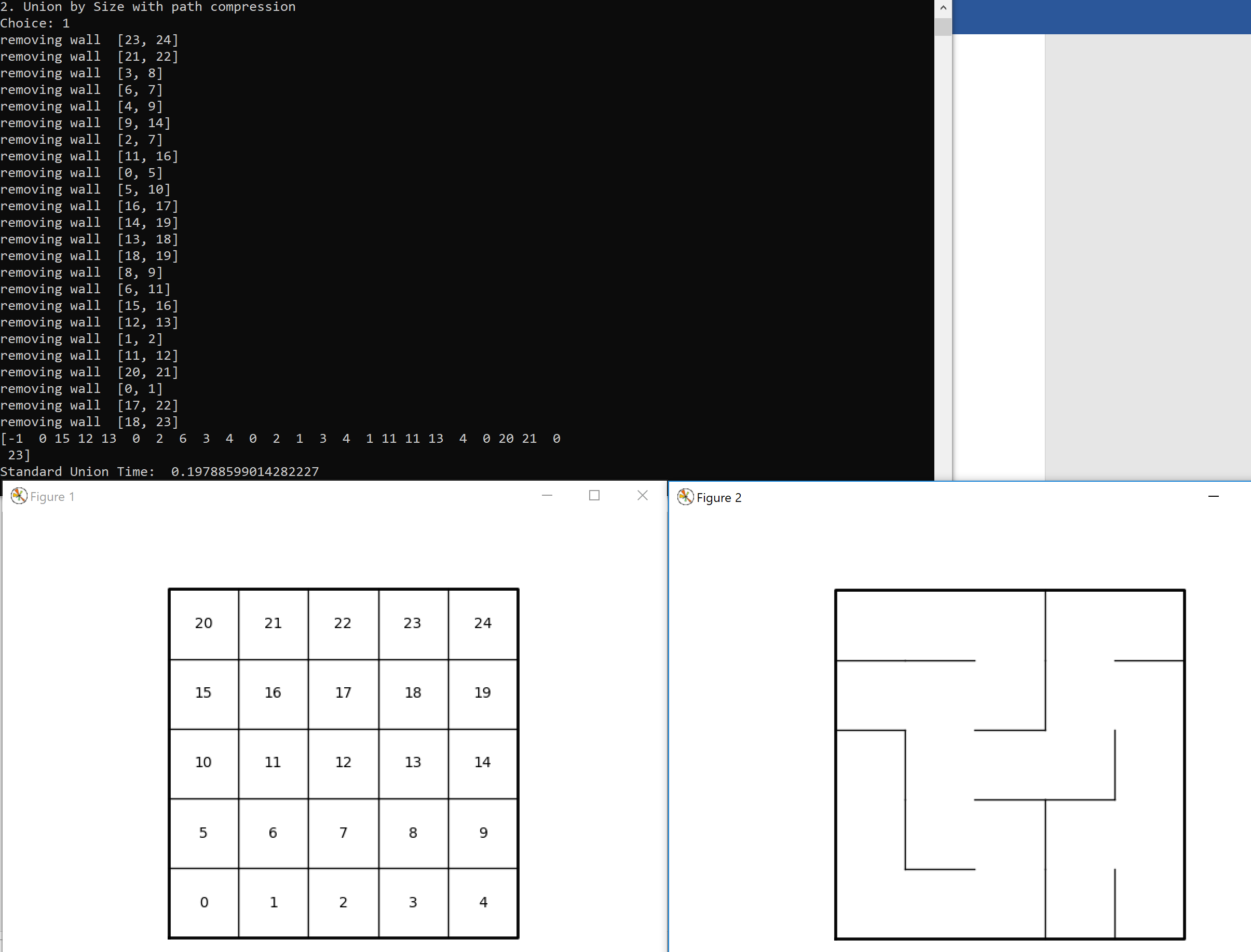


Output (Size and compression):

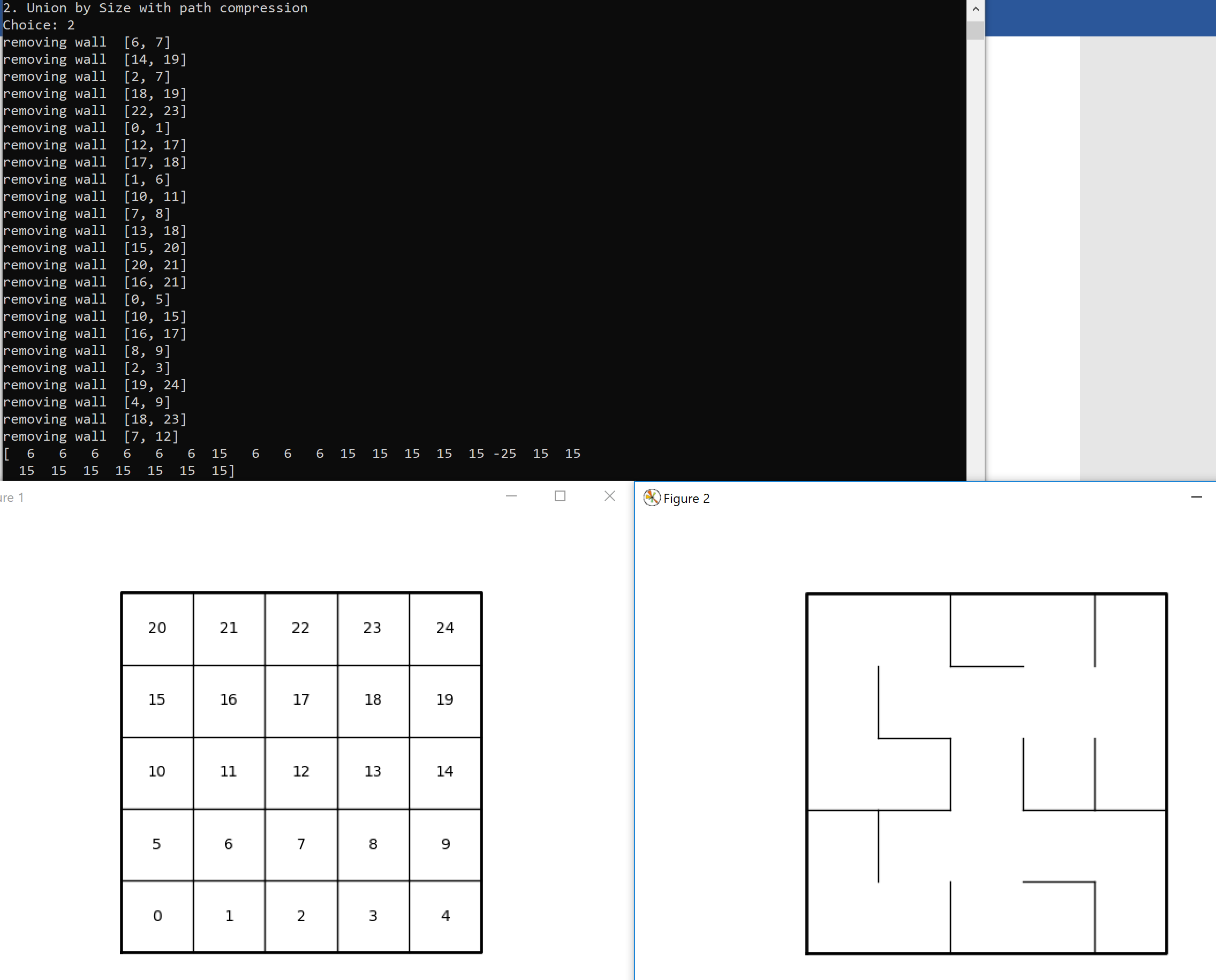


Input: 5x5:

Output (Standard):

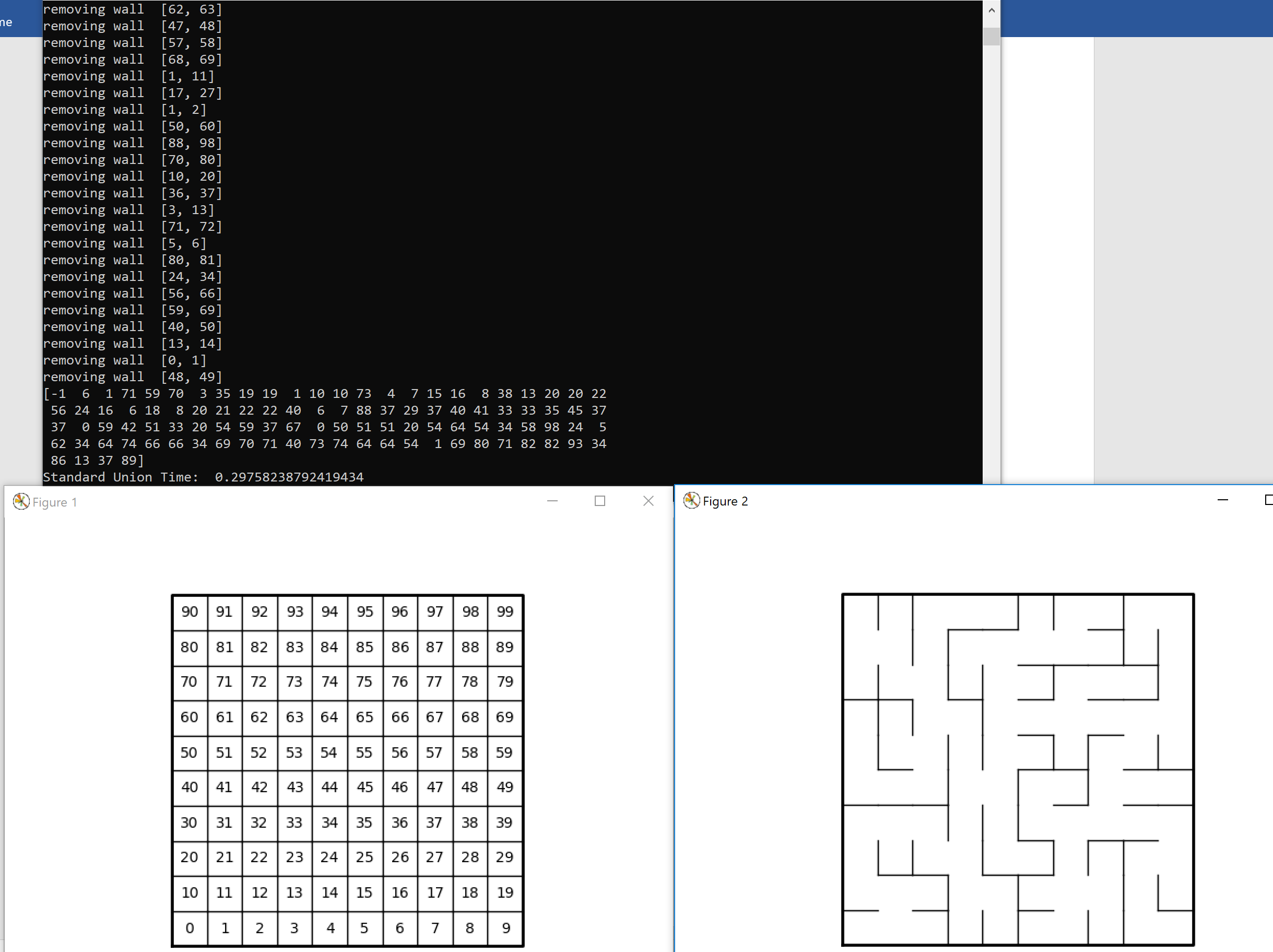


Output (Size and compression):

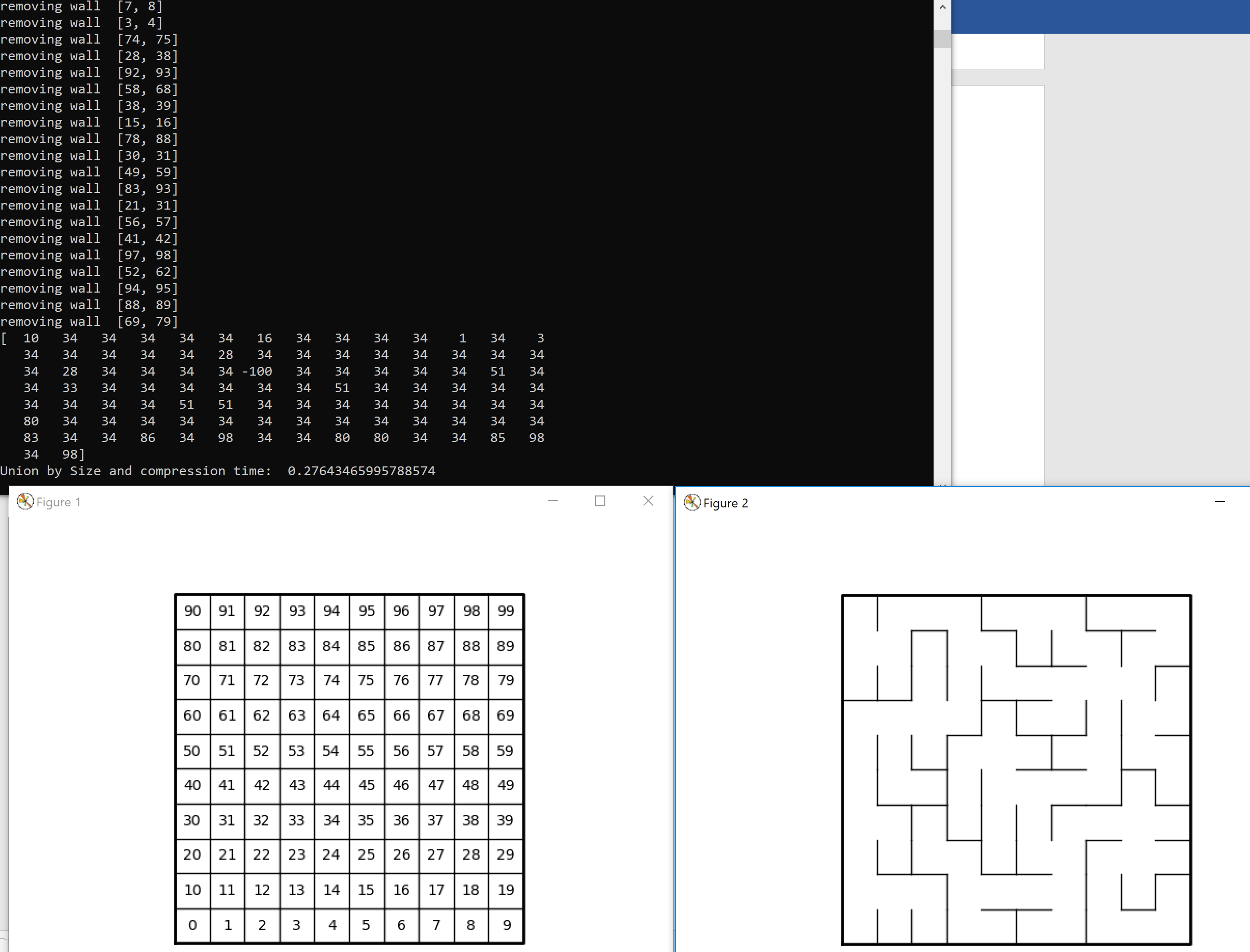


Input: 10x10:

Output (Standard):

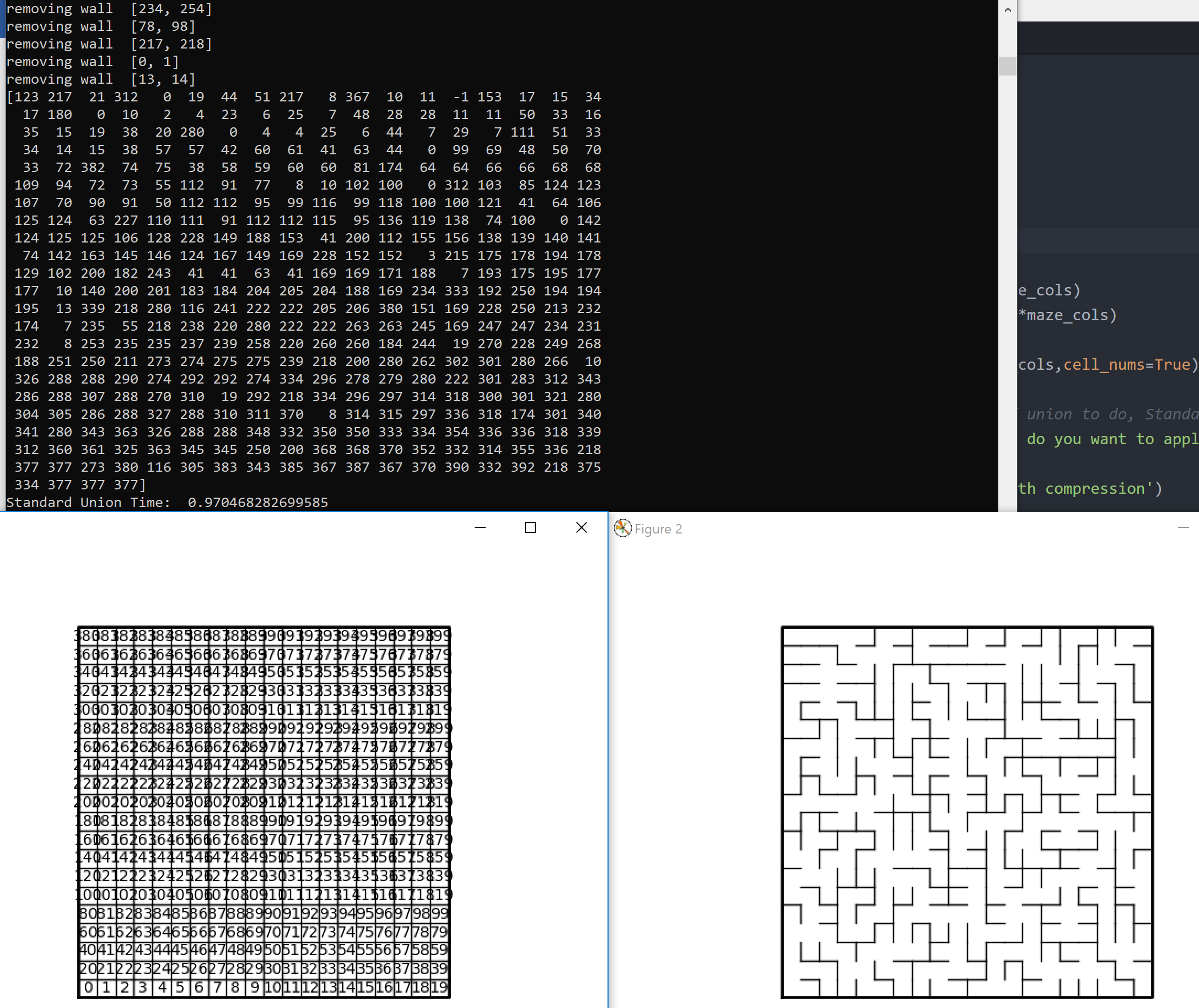


Output (Size and compression):

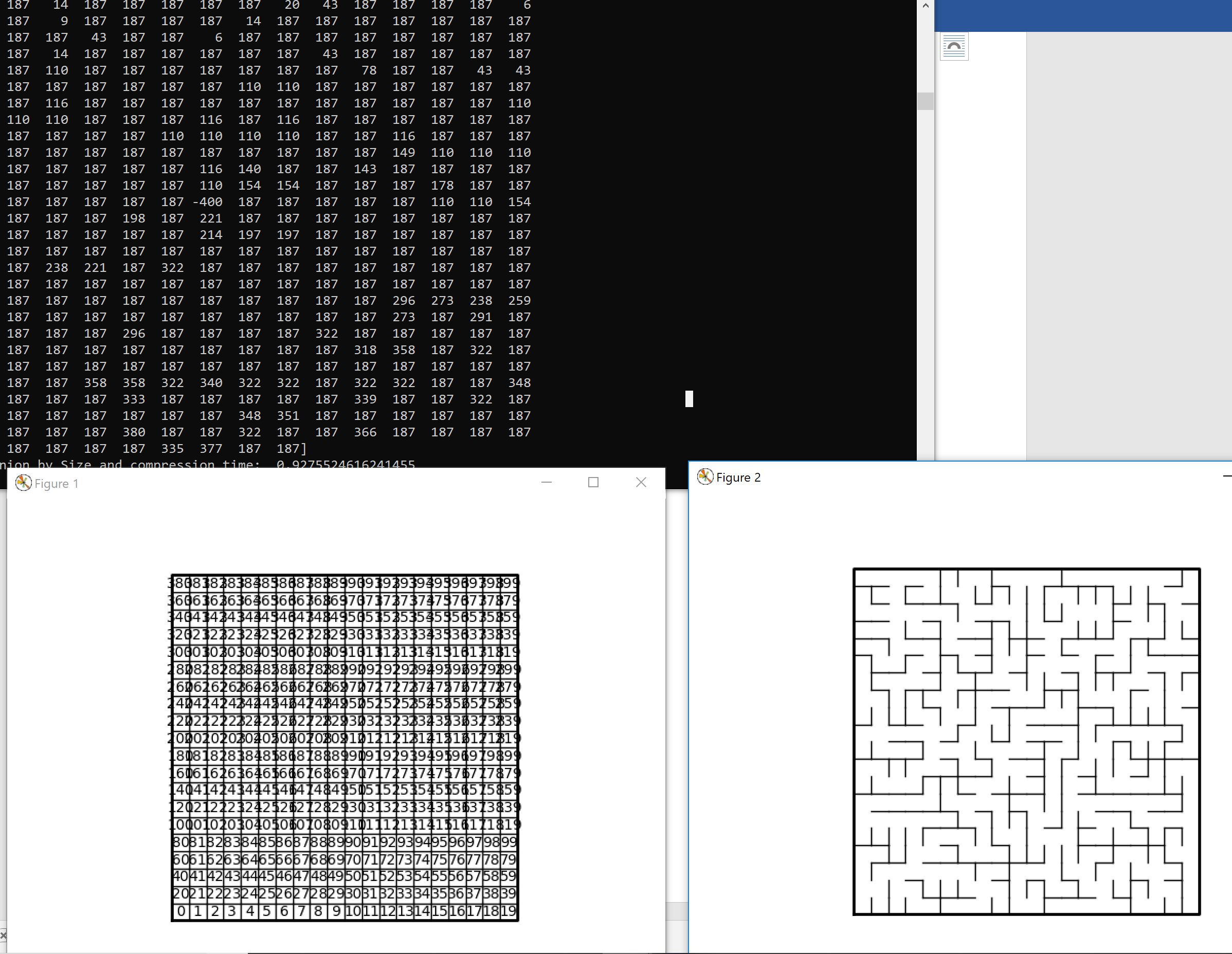


Input: 20x20:

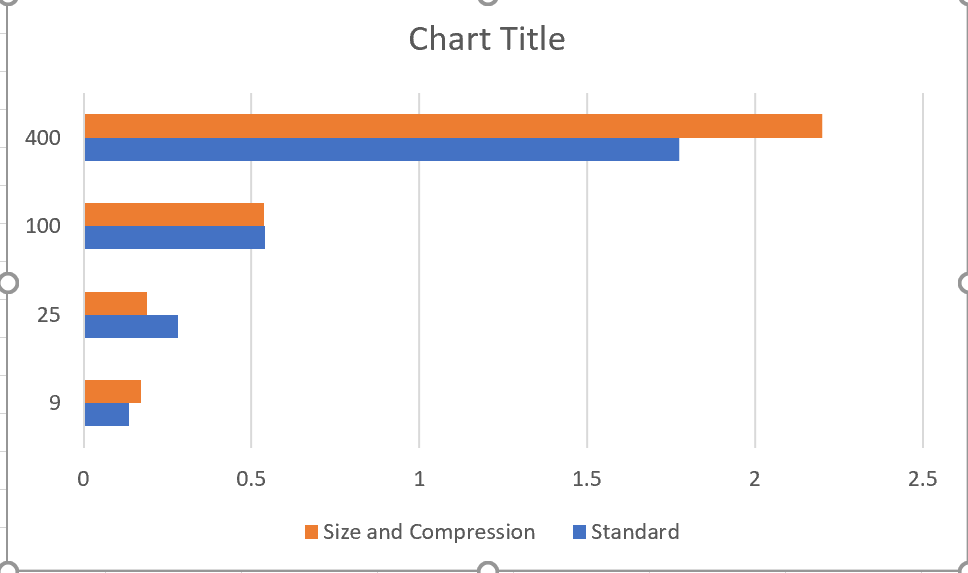
Output (Standard):



Output (Size and compression):



Running Times:



Summary:

What I learned in this lab was a deeper understanding of standard and how to apply a Union by size and compression data structure. I was also able to understand the running times of both ways of applying this data structure. I found it really fun how each maze ended up differently from the other mazes and how certain selection of list may make the disjoint set take longer to construct.

Appendix:

# Author: Rigoberto Quiroz

# Section: 1:30 PM - 2:50 PM

# This program will create a create maze with x rows and y columns. Then we will

# create lists that contain values that are next to each other. Then we will create

# a Disjoint set and set a while loop that will check the number of sets until

# we have 1 remaining set (contains all sets). Loop will unionize the values in

# our lists, and remove the walls that separate them. Creating pathways that will

# connect points A to B, creating only one path. Once that is done we are going

#to draw the maze, along with printing its Disjoint Set.

import matplotlib.pyplot as plt

import numpy as np

import random

import time

# Creates Disjoint Set according to the rows and columns of our maze

def DisjointSetForest(size):

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

def union(S,i,j):

# roots of i and j

ri = find(S,i)

rj = find(S,j)

# Do not belong in the same set

if ri !=rj:

# Union

S[rj] = ri

return

# Belong in the same set

return False

def find(S,i):

# Finds the root of our index i

if S[i] < 0:

return i

return find(S,S[i])

# Creates the max according to our x and y values

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

# Counts the number of Sets we have in our Disjoint Set

def numOfSets(S):

count = 0

# Count + 1, if value is a root

for key in S:

if key <= -1:

count +=1

return count

def find\_c(S,i):

# Finds root of i

if S[i] < 0:

return i

# Sets all values in a set to its root, without going to other values in

# the set

r = find\_c(S,S[i])

S[i] = r

return r

def unionBySize(S,i,j):

# Finds root of i and j

ri = find\_c(S,i)

rj = find\_c(S,j)

# do not belong in the same set

if ri != rj:

# Unionzing Set by size (Larger set)

# Root j is gretaer than root i

if -(S[ri]) < -(S[rj]):

# Keeps track of length of sets

S[rj] += S[ri]

S[ri] = rj

return

else:

# root i is greater than root j

S[ri] += S[rj]

S[rj] = ri

return

# Belong to the same set

return False

plt.close("all")

# Creates Maze specs, x and y

maze\_rows = 3

maze\_cols = 3

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

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

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

# User can choose which type of union to do, Standard, or compression

print('Which Disjoint set union do you want to apply:')

print('1. Standard Union')

print('2. Union by Size with path compression')

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

# Standard Union

if choice == 1:

standardStart = time.time()

# While we have more than 1 set

while numOfSets(S) != 1:

# Select a random list and unionize them

d = random.randint(0,len(walls)-1)

if union(S,walls[d][0],walls[d][1]) != False:

print('removing wall ',walls[d])

# remove them from lists

walls.pop(d)

# Final Disjoint Set, and maze drawing

print(S)

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

standardEnd = time.time()

print('Standard Union Time: ', standardEnd - standardStart)

plt.show()

exit(0)

# union by size and compression

if choice == 2:

compressionStart = time.time()

# While we have more than one set

while numOfSets(S) != 1:

# random list

d = random.randint(0,len(walls)-1)

# unionize items from random list

if unionBySize(S,walls[d][0],walls[d][1]) != False:

# remove from list

print('removing wall ',walls[d])

walls.pop(d)

#Final Disjoint Set and maze

print(S)

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

compressionEnd = time.time()

print('Union by Size and compression time: ', compressionEnd - compressionStart)

plt.show()

exit(0)

else:

# User selects something else than 1 or 2

print('Invalid input')

exit(0)