**Introduction**

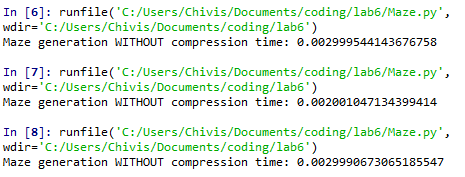
**RemoveWalls ():**

Proposed solution and implementation:

Like the given algorithm to remove wall randomly, except we use the regular union without compression function.

Made a function that take two parameters, the disjoint forest and the array of walls from the maze. First the number of sets from the disjoint forest is obtained to determine how many times to use the union function with the rest of the walls. A for loop is used to run through the number of sets left to unify. Inside the for loop, we obtain the index of the random walls chosen to be removed and compare if both cells are the same. If both cells are the same, then nothing is done. But if not, the randomly picked boxes are unified.

*Console output:*



Running time and time complexity:

1st test: 0.0029995s

2nd test: 0.002001s

3rd test: 0.0029991s

Time complexity: O(n)

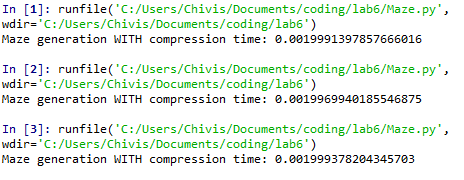
**RemoveWalls\_c ():**

Proposed solution and implementation:

The function should have the same process for removing walls, such as picking random walls that are going to be unified to create a path. Knowing the number of times to loop through the removing walls algorithm, etc.

Created a function that obtains the number of sets that must be unified to know the amount of times to remove walls in the maze. A for loop is used to iterate through the amount sets left to unify. Inside the for loop, an index from randomly picked walls is picked to remove. If both indices from the randomly picked cells differ, then they are removed/unified by using the function *union\_by\_size*.

*Console output:*



Running time and time complexity:

1st test: 0.0019991s

2nd test: 0.001997s

3rd test: 0.0019994s

Time complexity: O(n)

**Both functions:**

**Appendix**

|  |  |
| --- | --- |
|  | # Author: Esteban Retana |
|  | # Assignment: Use disjoint forest to build a maze |
|  | # Instructor: Olac Fuentes |
|  | # TA: Eduardo Lara |
|  | # Date of last modification:4/16/19 |
|  | # Purpose: To be able to create a good random maze with disjoint forests |
|  | import matplotlib.pyplot as plt |
|  | import numpy as np |
|  | import random |
|  | import time |
|  |  |
|  | 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 |
|  |  |
|  | 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 |
|  |  |
|  | def NumSets(S): |
|  | count =0 |
|  | for i in range(len(S)): |
|  | if S[i]<0: |
|  | count += 1 |
|  | return count |
|  |  |
|  | # Builds maze by removing walls without compression |
|  | def RemoveWalls(S,walls): |
|  | # Gets total number of sets in forest |
|  | sets = NumSets(S) |
|  | r = [] |
|  | for i in range(sets): |
|  | r = random.choice(walls) |
|  | i = walls.index(r) |
|  | # Checks if there is a pathalready |
|  | if find(S,r[0]) != find(S,r[1]): |
|  | # Eliminated random walls |
|  | walls.pop(i) |
|  | union(S,r[0],r[1]) |
|  |  |
|  | # Builds maze by removing walls with compression |
|  | def RemoveWalls\_c(S,maze\_walls): |
|  | # Gest total number of sets in forest |
|  | sets = NumSets(S) |
|  | r = [] |
|  | for i in range(sets): |
|  | r = random.choice(walls) |
|  | i = walls.index(r) |
|  | # Checks if there is a path already |
|  | if find(S,r[0]) != find(S,r[1]): |
|  | # Eliminated random walls |
|  | walls.pop(i) |
|  | union\_by\_size(S,r[0],r[1]) |
|  |  |
|  |  |
|  | plt.close("all") |
|  | maze\_rows = 10 |
|  | maze\_cols = 15 |
|  |  |
|  | walls = wall\_list(maze\_rows,maze\_cols) |
|  |  |
|  | # Displays original maze with cell numbers |
|  | draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True) |
|  |  |
|  | # Creates dfs with dimensions |
|  | fullSize = maze\_rows \* maze\_cols |
|  | S = DisjointSetForest(fullSize) |
|  | # ============================================================================= |
|  | # Maze generation without compression |
|  | # ============================================================================= |
|  | start = time.time() |
|  |  |
|  | RemoveWalls(S,walls) |
|  |  |
|  | end = time.time() |
|  |  |
|  | removeWallTime = end - start |
|  |  |
|  | draw\_maze(walls,maze\_rows,maze\_cols) |
|  | print('Maze generation WITHOUT compression time:', removeWallTime) |
|  |  |
|  | # ============================================================================= |
|  | # Maze generation with compression |
|  | # Uncomment this section and comment out the maze generation without compression to find running times |
|  | # ============================================================================= |
|  | #start2 = time.time() |
|  | # |
|  | #RemoveWalls\_c(S,walls) |
|  | # |
|  | #end2 = time.time() |
|  | # |
|  | #removeWallTime2 = end2 - start2 |
|  | # |
|  | #draw\_maze(walls,maze\_rows,maze\_cols) |
|  | #print('Maze generation WITH compression time:', removeWallTime2) |

Academic dishonesty

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.