**Lab 7**

The code provides a solution to the random mazes by making use of an adjacency list that is provided by the method AL\_rep(walls, maze\_rows, maze\_cols) which takes the current wall list and creates an Adjacency List representation by comparing the maze list to a “blank” and filling in the spaces that were removed to create the maze. The methods Breath\_First\_Search() and Depth\_First\_Search() search for a path from the beginning point at vertex 0 to the end at the last vertex of the maze by using nested loops to traverse the adjacency list representation of the graph to check if a path connects them, then going back through it by storing the previous vertex that was visited. Depth\_First\_SearchR() uses recursion to check for possible paths, if the end is reached, then it will return the proper path.

**Experimental Lab Results**

|  |  |  |  |
| --- | --- | --- | --- |
| METHODS | INPUTS | | OUTPUTS |
| Breath\_First\_Search() | **Columns: 3**  **Rows: 3**  **Walls Removed: 10** |  | **[0, 1, 2, 5, 8]** |
| Depth\_First\_Search() | **[0, 3, 4, 5, 8]** |
| Depth\_First\_SearchR() | **[0, 1, 2, 5, 8]** |
| Breath\_First\_Search() | **Columns: 10**  **Rows: 10**  **Walls Removed: 105** |  | **[0, 1, 11, 12, 13, 14, 24, 25, 26, 36, 35, 45, 44, 54, 53, 63, 64, 74, 84, 85, 75, 76, 86, 87, 77, 78, 79, 89, 99]** |
| Depth\_First\_Search() | **[0, 1, 11, 12, 13, 14, 24, 25, 26, 36, 35, 45, 44, 54, 53, 63, 64, 74, 84, 85, 75, 76, 86, 87, 77, 78, 79, 89, 99]** |
| Depth\_First\_SearchR() | **[0, 1, 11, 12, 13, 14, 24, 25, 26, 36, 35, 45, 44, 54, 53, 63, 64, 74, 84, 85, 75, 76, 86, 87, 77, 78, 79, 89, 99]** |
| Breath\_First\_Search() | **Columns: 20**  **Rows: 10**  **Walls Removed: 190** |  | **Path to end of the maze not possible**  **None** |
| Depth\_First\_Search() | **Path to end of the maze not possible**  **None** |
| Depth\_First\_SearchR() | **“ “** |

**Time Complexity**

Where v is the number of vertices and e is the number of edges.

AL\_rep() = O(v2)

Breath\_First\_Search() = O( |v|+|e|)

Depth\_First\_Search() = O(|v|+|e|)

Depth\_First\_SearchR() = O(|v|+|e|)

**Conclusion**

A graph representation can reduce complex problems or ones that seem random in to a more understandable form that can be worked with to find a solution.

**Source Code**

**Created on Fri Apr 29 10:09:45 2019**

**CS 2302 - Data Structures**

**Instructor:Olac Fuentes**

**Lab 7, Disjoint Set Forests**

**Creates a random labyrinth by making use of the Forest's characteristics, and makes**

**use of the breadth first and depth first search algorithms to find a solution to the maze.**

**TA:Anindita Nath**

**@author: Hugo Chavez**

**"""**

**import time**

**# Starting point for program to build and draw a maze**

**# Modify program using disjoint set forest to ensure there is exactly one**

**# simple path joiniung any two cells**

**# Programmed by Olac Fuentes**

**# Last modified March 28, 2019**

**import math**

**import matplotlib.pyplot as plt**

**import numpy as np**

**import random**

**#-------------------------------------**

**class Queue:**

**def \_\_init\_\_(self):**

**self.items = []**

**def isEmpty(self):**

**return self.items == []**

**def enqueue(self, item):**

**self.items.insert(0,item)**

**def dequeue(self):**

**return self.items.pop()**

**def size(self):**

**return len(self.items)**

**def peek(self):**

**return self.items**

**class Stack:**

**def \_\_init\_\_(self):**

**self.items = []**

**def isEmpty(self):**

**return self.items == []**

**def push(self, item):**

**self.items.append(item)**

**def pop(self):**

**return self.items.pop()**

**def peek(self):**

**return self.items[len(self.items)-1]**

**def size(self):**

**return len(self.items)**

**#-------------------------------------**

**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 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**

**# returns a True value if the roots 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**

**return True**

**return False**

**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\_c(S,i,j):**

**# Joins i's tree and j's tree, if they are different**

**# Uses path compression**

**# returns a True value if the roots are different**

**ri = find\_c(S,i)**

**rj = find\_c(S,j)**

**if ri!=rj:**

**S[rj] = ri**

**return True**

**return False**

**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**

**# returns a True value if the roots are different**

**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**

**return True**

**return False**

**def one\_Forest(S):**

**#checks if the Forest has more than one root**

**count = 0**

**for i in range(len(S)):**

**if S[i] < 0:**

**count +=1**

**if count == 1:**

**return True**

**return False**

**#-------------------------------------**

**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 AL\_rep(walls, row,col):**

**AL = [ [] for i in range(row\*col) ]**

**walls1 = wall\_list(row,col)**

**count = len(walls)**

**x = 0**

**while count != 0:**

**i=0**

**while i < len(walls1):**

**if walls[x] == walls1[i]:**

**walls1.pop(i)**

**i+=1**

**x+=1**

**count -= 1**

**for i in range(row\*col):**

**for j in range(len(walls1)):**

**if i == walls1[j][0]:**

**AL[i].append(walls1[j][1])**

**AL[walls1[j][1]].append(i)**

**return AL**

**def Breadth\_First\_Search(AL,end):**

**q = Queue()**

**prev = np.zeros(len(AL),dtype=np.int)-1**

**visited = np.zeros(len(AL),dtype=bool)**

**q.enqueue(0)**

**visited[0] = True**

**while not q.isEmpty():**

**smt = int(q.dequeue())**

**for i in range(len(AL[smt])):**

**if not visited[AL[smt][i]]:**

**visited[AL[smt][i]] = True**

**prev[AL[smt][i]] = smt**

**q.enqueue(AL[smt][i])**

**currentV = end**

**lis=[]**

**if visited[end]:**

**while prev[currentV] != -1:**

**lis.append(currentV)**

**currentV = prev[currentV]**

**lis.append(0)**

**lis.reverse()**

**return lis**

**print('Path to end of the maze not possible')**

**def Depth\_First\_Search(AL,end):**

**s = Stack()**

**prev = np.zeros(len(AL),dtype=np.int)-1**

**visited = np.zeros(len(AL),dtype=bool)**

**s.push(0)**

**visited[0] = True**

**while not s.isEmpty():**

**current = int(s.pop())**

**for i in range(len(AL[current])):**

**if not visited[AL[current][i]]:**

**visited[AL[current][i]] = True**

**prev[AL[current][i]] = current**

**s.push(AL[current][i])**

**currentV = end**

**lis=[]**

**if visited[end]:**

**while prev[currentV] != -1:**

**lis.append(currentV)**

**currentV = prev[currentV]**

**lis.append(0)**

**lis.reverse()**

**return lis**

**print('Path to end of the maze not possible')**

**def Depth\_First\_SearchR(path,AL,current,end,visited):**

**if current == end:**

**path.append(end)**

**return path**

**if current not in visited:**

**visited.append(current)**

**for i in range(len(AL[current])):**

**p = []**

**p.append(current)**

**pat = Depth\_First\_SearchR(path+p,AL,AL[current][i],end,visited)**

**if pat:**

**return pat**

**return ''**

**#-------------------------------------**

**plt.close("all")**

**maze\_rows = 20**

**maze\_cols = 10**

**print('The number of rows', maze\_rows,', the number of columns',maze\_cols,'and the number of cells:', maze\_rows\*maze\_cols)**

**m = int(input('Enter the number of walls to remove:'))**

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

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

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

**count = 0**

**if m == ((maze\_rows\*maze\_cols)-1):**

**print('There is a unique path from source to destination (when m = n-1)')**

**# Builds maze with standard union**

**while not one\_Forest(forest): #!= 1:**

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

**if union(forest, walls[d][0],walls[d][1]):**

**walls.pop(d)**

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

**#-------------------------------------**

**elif m >= ((maze\_rows\*maze\_cols)-1):**

**print('There is at least one path from source to destination (when m > n-1)')**

**# Builds maze with standard union**

**while m != count: #!= 1:**

**while not one\_Forest(forest):**

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

**if union(forest, walls[d][0],walls[d][1]):**

**walls.pop(d)**

**count+=1**

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

**walls.pop(d)**

**count+=1**

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

**#-------------------------------------**

**elif m < (maze\_rows\*maze\_cols):**

**print('A path from source to destination is not guaranteed to exist (when m < n-1)')**

**# Builds maze with standard union**

**while m != count: #!= 1:**

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

**if union(forest, walls[d][0],walls[d][1]):**

**walls.pop(d)**

**count +=1**

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

**maze\_end = (maze\_rows\*maze\_cols)-1**

**AL = AL\_rep(walls,maze\_rows,maze\_cols)**

**print('Adjacency List Representation:')**

**print(AL)**

**print()**

**print('BFD path:')**

**start = time.time\_ns**

**print(Breadth\_First\_Search(AL,maze\_end))**

**end = time.time\_ns()**

**#z =float(end-start)**

**#print('time:',z)**

**print()**

**print('DFS path:')**

**start = time.time\_ns()**

**print(Depth\_First\_Search(AL,maze\_end))**

**end = time.time\_ns()**

**#z =float(end-start)**

**#print('time:', z)**

**print()**

**print('DFS-R path:')**

**start = time.time\_ns()**

**x = []**

**print(Depth\_First\_SearchR(x,AL,0,maze\_end,visited=[]))**

**end = time.time\_ns()**

**#z =float(end-start)**

**#print('time:', z)**

**print()**

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.

\_\_\_\_\_\_\_\_\_\_\_\_Hugo Chavez\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_