FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION

OF HIGHER EDUCATION

ITMO UNIVERSITY

Report

on the practical task No. 5

“Algorithms on graphs. Introduction to graphs and basic algorithms on graphs ”

Performed by

Cheng-Yuan and Ma

Academic group

J4133c

Petr Chunaev *St.*

Petersburg 2022

Language : python

Goal :The use of different representations of graphs and basic algorithms on graphs (Depth-first search and Breadth-first search)

Problem :

I. Generate a random adjacency matrix for a simple undirected unweighted graph with 100 vertices and 200 edges (note that the matrix should be symmetric and contain only 0s and 1s as elements). Transfer the matrix into an adjacency list. Visualize the graph and print several rows of the adjacency matrix and the adjacency list. Which purposes is each representation more convenient for?

II. Use Depth-first search to find connected components of the graph and Breadth- first search to find a shortest path between two random vertices. Analyse the results obtained.

III. Describe the data structures and design techniques used within the algorithms.

Theory:

adjacency list represents :
a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

adjacency matrix :
a dense way of describing the finite graph structure. It is the 2D matrix that is used to map the association between the graph nodes. If **n** number of vertices, thsen the adjacency matrix of that graph is **n x n**

Application

1. **Adjacency Matrix:** Adjacency matrix is used where information about each and every possible edge is required for the proper working of an algorithm like : Floyd-Warshall Algorithm where shortest path from each vertex to each every other vertex is calculated (if it exists). It can also be used in DFS (Depth First Search) and BFS (Breadth First Search) but list is more efficient there. Sometimes it is also used in network flows.

2. **Adjacency List:** Adjacency List is a space efficient method for graph representation and can replace adjacency matrix almost everywhere if algorithm doesn't require it explicitly. It is used in places like: BFS, DFS, Dijkstra's Algorithm etc.

The **Depth–first search (DFS)** algorithm starts at the root of the tree (or some arbitrary node for a graph) and explored as far as possible along each branch before **backtracking**.

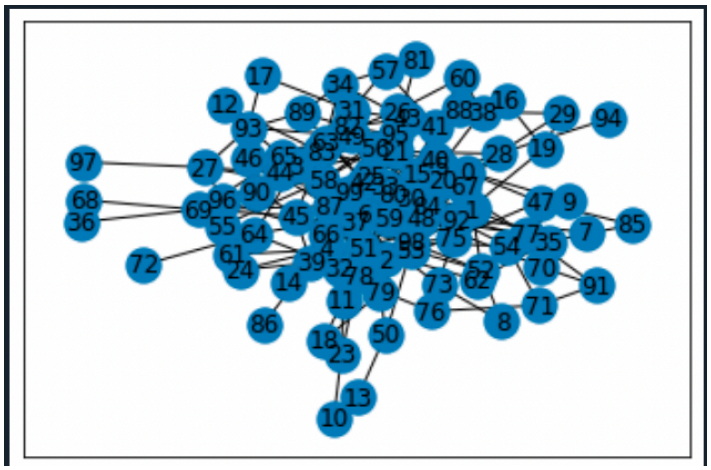
The **Breadth–first search (BFS)** algorithm also starts at the root of the tree (or some arbitrary node of a graph), but unlike DFS, it explores the neighbor nodes first, before moving to the next-level neighbors. In other words, BFS explores vertices in the order of their distance from the source vertex, where distance is the minimum length of a path from the source vertex to the node.

Results:

I. II.

These are graphs with 100 nodes and 200 edges

!!! Due to massive number in matrix , it is hard to show adjacency matrix clearly , so I made another graph with fewer nodes and edges below this graph !!!



These are adjacency list : from 0 to 99

```
the adjacency list :
0 -> 9 -> 26 -> 43 -> 53 -> 67 -> 84
1 -> 5 -> 15 -> 35 -> 42 -> 52 -> 84
2 -> 11 -> 48 -> 75 -> 87
3 -> 49 -> 64 -> 95
4 -> 24 -> 66 -> 80 -> 98
5 -> 1 -> 33 -> 38 -> 87
6 -> 11 -> 20 -> 25 -> 64 -> 65 -> 67 -> 92
7 -> 92
8 -> 52 -> 73
9 -> 0 -> 54
10 -> 23
11 -> 2 -> 6 -> 18 -> 23 -> 32
12 -> 83 -> 93
13 -> 50
14 -> 51 -> 66
15 -> 1 -> 21 -> 75 -> 83 -> 87 -> 88 -> 92 -> 95
16 -> 19 -> 41 -> 43
17 -> 31 -> 93
18 -> 11 -> 79
19 -> 16 -> 29 -> 92
20 -> 6 -> 41 -> 49 -> 77 -> 84
21 -> 15 -> 28 -> 44
23 -> 10 -> 11 -> 78
24 -> 4 -> 39 -> 55 -> 61
25 -> 6 -> 37 -> 80 -> 82 -> 84
26 -> 0 -> 60 -> 65
27 -> 44 -> 93 -> 96 -> 97 -> 99
28 -> 21 -> 29 -> 67 -> 80 -> 84 -> 94
29 -> 19 -> 28 -> 38
30 -> 31 -> 54 -> 66 -> 98
31 -> 17 -> 30 -> 43 -> 57 -> 63
32 -> 11 -> 37 -> 59
33 -> 5 -> 44 -> 48 -> 67
34 -> 43 -> 57 -> 63 -> 89
35 -> 1 -> 62 -> 85 -> 92
36 -> 69
37 -> 25 -> 32 -> 42 -> 73 -> 83 -> 99
38 -> 5 -> 29 -> 41 -> 60 -> 88 -> 95
39 -> 24 -> 45 -> 58 -> 76 -> 86 -> 87
40 -> 47 -> 63 -> 99
41 -> 16 -> 20 -> 38 -> 49 -> 57 -> 59 -> 60
42 -> 1 -> 37 -> 53 -> 63 -> 89
```

```
45 -> 39 -> 63 -> 66 -> 69 -> 98
46 -> 63 -> 87
47 -> 40 -> 75 -> 85
48 -> 2 -> 33 -> 66 -> 70 -> 77 -> 87 -> 95 -> 99
49 -> 3 -> 20 -> 41 -> 56 -> 93
50 -> 13 -> 53 -> 79
51 -> 14 -> 59 -> 61 -> 62 -> 75 -> 79 -> 84 -> 90 -> 92
52 -> 1 -> 8 -> 53 -> 98
53 -> 0 -> 42 -> 50 -> 52 -> 66
54 -> 9 -> 30 -> 71 -> 98
55 -> 24 -> 58
56 -> 49 -> 58 -> 82 -> 84
57 -> 31 -> 34 -> 41
58 -> 39 -> 43 -> 55 -> 56 -> 80 -> 82
59 -> 32 -> 41 -> 51 -> 78 -> 80
60 -> 26 -> 38 -> 41
61 -> 24 -> 51 -> 69
62 -> 35 -> 51
63 -> 31 -> 34 -> 40 -> 42 -> 45 -> 46
64 -> 3 -> 6 -> 69 -> 72 -> 78
65 -> 6 -> 26 -> 69 -> 93
66 -> 4 -> 14 -> 30 -> 45 -> 48 -> 53 -> 96
67 -> 0 -> 6 -> 28 -> 33
68 -> 69
69 -> 36 -> 45 -> 61 -> 64 -> 65 -> 68 -> 87
70 -> 48 -> 91
71 -> 54 -> 76 -> 91
72 -> 64
73 -> 8 -> 37 -> 75 -> 76
75 -> 2 -> 15 -> 47 -> 51 -> 73
76 -> 39 -> 71 -> 73
77 -> 20 -> 48 -> 91
78 -> 23 -> 59 -> 64 -> 92
79 -> 18 -> 50 -> 51 -> 84
80 -> 4 -> 25 -> 28 -> 58 -> 59 -> 99
81 -> 95
82 -> 25 -> 43 -> 56 -> 58 -> 89 -> 95
83 -> 12 -> 15 -> 37 -> 95
84 -> 0 -> 1 -> 20 -> 25 -> 28 -> 51 -> 56 -> 79
85 -> 35 -> 47
86 -> 39
87 -> 2 -> 5 -> 15 -> 39 -> 46 -> 48 -> 69 -> 99
88 -> 15 -> 38 -> 43
```

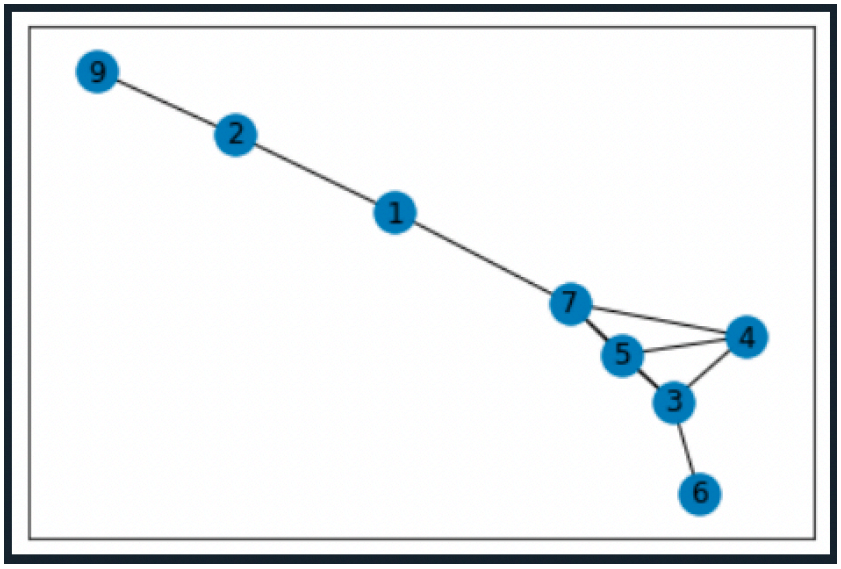
After using breadth-first search to find route from 51 to 11 : {51 -> 59 -> 32 -> 11}

After using depth-first search , Total components are 3 : {[22],[74],[rest of nodes]}

```
81 -> 95
82 -> 25 -> 43 -> 56 -> 58 -> 89 -> 95
83 -> 12 -> 15 -> 37 -> 95
84 -> 0 -> 1 -> 20 -> 25 -> 28 -> 51 -> 56 -> 79
85 -> 35 -> 47
86 -> 39
87 -> 2 -> 5 -> 15 -> 39 -> 46 -> 48 -> 69 -> 99
88 -> 15 -> 38 -> 43
89 -> 34 -> 42 -> 82 -> 93
90 -> 51 -> 93
91 -> 70 -> 71 -> 77
92 -> 6 -> 7 -> 15 -> 19 -> 35 -> 51 -> 78 -> 99
93 -> 12 -> 17 -> 27 -> 49 -> 65 -> 89 -> 90
94 -> 28
95 -> 3 -> 15 -> 38 -> 48 -> 81 -> 82 -> 83
96 -> 27 -> 44 -> 66
97 -> 27
98 -> 4 -> 30 -> 45 -> 52 -> 54
99 -> 27 -> 37 -> 40 -> 48 -> 80 -> 87 -> 92

using Breadth-first search to find min route from 51 to 11 and found :
[51, 59, 32, 11]
The connected components are after using Depth-first search :
[[0, 43, 34, 89, 42, 63, 45, 69, 64, 78, 23, 10, 11, 2, 48, 66, 14, 51, 62, 35, 1, 15, 83, 12, 93, 49, 3, 95, 82, 25, 80, 58, 39, 86, 76, 71, 54, 98, 30, 31, 17, 57, 41, 38,
5, 33, 67, 6, 92, 99, 27, 44, 21, 28, 94, 84, 20, 77, 91, 70, 56, 79, 18, 50, 53, 52, 8, 73, 75, 47, 40, 85, 37, 32, 59, 13, 29, 19, 16, 96, 97, 87, 46, 7, 65, 26, 60, 88, 4,
24, 55, 61, 9, 81, 90, 72, 68, 36], [22], [74]]
there is total 3 components
```

These are graph with 10 nodes and 10 edges (for clear understanding of how outcome looks like)



Min route from 9 to 5 is {9,2,1,7,5}

Total is 3 components : {[0] , [8] , [rest of nodes]}

```
the adjacency matrix :
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 1, 1, 1, 1, 0, 0]
[0, 0, 0, 1, 0, 1, 0, 1, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 1, 1, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

edges are : [[7, 1], [2, 1], [5, 4], [6, 3], [5, 3], [7, 5], [4, 3], [7, 3], [7, 4], [9, 2]]
Adjacency List:
defaultdict(<class 'list'>, {1: [2, 7], 2: [1, 9], 3: [4, 5, 6, 7], 4: [3, 5, 7], 5: [3, 4, 7], 6: [3], 7: [1, 3, 4, 5], 9: [2]})
the adjacency list :
1 -> 2 -> 7
2 -> 1 -> 9
3 -> 4 -> 5 -> 6 -> 7
4 -> 3 -> 5 -> 7
5 -> 3 -> 4 -> 7
6 -> 3
7 -> 1 -> 3 -> 4 -> 5
9 -> 2

using Breadth-first search to find min route from 9 to 5 and found :
[9, 2, 1, 7, 5]
The connected components are after using Depth-first search :
[[0], [1, 7, 5, 4, 3, 6, 2, 9], [8]]
there is total 3 components
```

III.

I used numpy array and list as mainly structure

Firstly I created random matrix using random implementation in python

And converted them into adjacency list , so I can use it with and DFS BFS to find shortest part and connected components

DFS :

STEP 1: Take an array consisting of all the vertices. That is the size of array = number of **vertices.**

STEP 2: Assign each vertex value in the array as -1

STEP 3: Start from each vertex and apply DFS to them if and only if vertex value=-1 Else move to the next vertex.

STEP 4: At last count, the number of times DFS moves was initiated from the parent loop. That many are the components and each component is the vertices traversed during DFS.

BFS :

STEP 1: Take an Empty Queue.

STEP 2: Select a starting node (visiting a node) and insert it into the Queue.

STEP 3: Provided that the Queue is not empty, extract the node from the Queue and insert its child nodes (exploring a node) into the Queue.

STEP 4: Print the extracted node.

Conclusion:

It is high efficient to use either adjacency matrix and adjacency list , especially when it comes to use such traverse algorithm as DFS and BFS , it is extremely convenient to find if there is connected node . And also very easy to recognize output for observer .

Appendix

https://raw.githubusercontent.com/MaChengYuan/task5/main/task5.py