

FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION  
OF HIGHER EDUCATION  
ITMO UNIVERSITY

Report  
On the practical task number 5  
“Algorithms on graphs. Introduction to graphs and basic algorithms on  
graphs”

Performed by:  
Patrik Marek Stronski  
ICU number: 333976  
J4133C  
Accepted By:  
Dr Petr Chunaev

St. Petersburg  
2021

## Goal

The goal of the task is to explore two types of graph traversal approaches – breadth-first and depth-first search.

## Formulation of the problem

The task is to create a graph with 100 nodes and 200 edges. The graph needs to be unweighted and not directed. This graph needs to be showed both using adjacency matrix and adjacency list notations.

On top of that, there are two additional tasks:

- Find the list of components using the depth-first search(DFS)
- Find the shortest distance between two vertices using breadth-first search (BFS)

## Brief theoretical part

Graph is composed of nodes (vertices) connected with edges. There are numerous types of graphs: directed/undirected, weighted/unweighted. Directed graphs, are when the connections between vertices are only one directional. It means that the edge  $(v_1, v_2)$  connects two vertices, from  $v_1$  to  $v_2$ , but not the other way. Of course, there can be also edge  $(v_2, v_1)$ , but it is a separate edge. When the graph is undirected, the vertices are either connected (bidirectionally) or not. When it comes to weighted graphs – each edge has a numerical weight assigned. This weight can be represented by a floating point/integer number. If the graph is not weighted, all weights are by default 1.

Graphs can be used as models to many real-world examples. For instance, the Internet or a small LAN can be represented as a graph of communicating nodes. Social networks such as VK or Facebook also contain information that can be presented on a graph. For example the network of friendships – here undirected graph can be used, because as person A is friends with person B, then person B is friends with A. Metro stations can also be presented as graph, where the weighted edges could for example represent average time for a train to get from station A to B.

Graph can be represented using numerous ways. First and foremost, an **adjacency matrix** can be used to represent the graph connections. It is a simple matrix, where the crossings of indexes  $i_1$  and  $i_2$  define whether the edge  $(v_1, v_2)$  exists. If the edge exists, non-zero value is put into the matrix. Eventually it is 0 if the edge does not exist and two nodes are not directly connected. The matrix is symmetric if the edges are undirected.

[0, 1, 0]

[1, 0, 1]

[0, 1, 0]

This notation means that  $v_0$  is connected with  $v_1$ ,  $v_1$  is connected with  $v_0$  and  $v_2$ , and  $v_2$  is connected with  $v_1$ .

Second version of representation is **adjacency list**. Here, on the other hand, it is represented as a list of lists. The outer lists represents the nodes  $v_1 \dots v_n$ , whereas each inner list represents the nodes which are connected to it. Therefore  $[[1], [0, 2], [1]]$  means that node  $v_0$  is connected to  $v_1$ , and  $v_1$  is connected to  $v_0$  and  $v_2$ , but there is no connection between  $v_0$  and  $v_2$ .

Both ways of representation are useful. Adjacency matrix easily gives a possibility to depict also weights, which is a big benefit of it. On the other hand it is very storage-consuming, as it has size  $|V|^2$  which is quite much, especially if the graph is not dense (there are not many edges). Adjacency list is far more compact and thus more readable. We exactly know which vertex is connected to which other one. Nevertheless, it requires some modifications in order to introduce weights, thus an adjacency matrix might be better for this case (it already fits this case out of the box).

When it comes to graphs, they can be searched using several algorithms. The two approaches used for traversing the graph are DFS and BFS.

DFS means firstly the algorithm tries to find the most deep path and then backtrack one degree above and change the route a bit. It is often used for finding connected components, creating maps, solving mazes.

BFS means the algorithm explores always nodes around it equally distanced from the root. Firstly it checks the adjacent ones to root, then adjacent nodes to them and so on. It is good for mapping the graph, finding shortest path and exploring the vicinity to some node.

## Results

I am saving my graph in the class Graph which I created. The class randomly creates a graph with 100 nodes and 200 edges. It also contains methods for solving other problems, such as BFS and DFS. The class is stored in *generate\_graph.py*.

The result of the creation of the graph is the following (example):

Adjacency matrix:

[illegible]

Adjacency list:

0: [9, 17, 20, 64, 87]

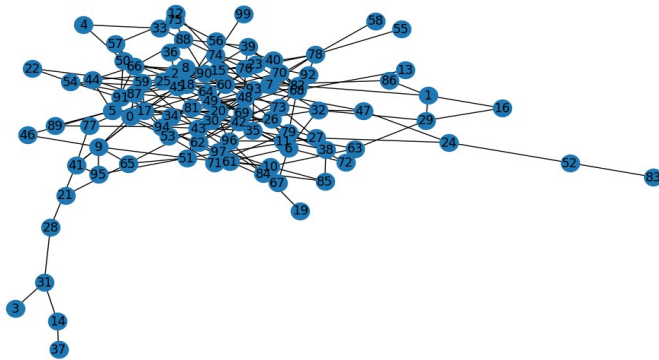
1: [16, 29, 82, 86]

2: [36, 44, 69]

3: [31]

4: [33, 50]

80



98

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

[illegible][illegible]

80: []

98: []

[10, 51, 9, 0]

10: [51, 61, 62, 63, 85]

51: [9, 10, 27, 43, 62, 65, 95]

9: [0, 17, 41, 46, 51, 65, 95]

0: [9, 17, 20, 64, 87]

## Conclusion

As we can see graphs can be both processed and written down using several algorithms. For denoting graphs **adjacency matrix** and **adjacency list** are the good ones. The first one is more versatile, but also takes far more memory, whereas the second one is very simple to understand and to use, but does not support weights. Based on that both approaches have their benefits and drawbacks.

When it comes to searches done on graphs, there are two main methods and both are good for some particular things. Solving a maze would be hard with BFS, whereas for DFS it is natural to do it this way, especially that if a dead end is found the algorithm can **backtrack**. Similarly for finding connected components and exploring graph. BFS however is far better in finding the shortest path from some starting edge to the end.

## References

<https://networkx.org/documentation/stable/reference/classes/graph.html#networkx.Graph>

## Appendix

My project is hosted here:

<https://github.com/PatrykStronski/Algorithms5>

The whole adjacency list and adjacency matrix:

---adjacency list---

```
0: [9, 17, 20, 64, 87]
1: [16, 29, 82, 86]
2: [36, 44, 69]
3: [31]
4: [33, 50]
5: [20, 44, 46, 53, 62, 66, 87, 89, 91]
6: [26, 32, 35, 42, 61, 67, 72]
7: [30, 49, 59, 78, 86]
8: [45, 48, 60, 76]
9: [0, 17, 41, 46, 51, 65, 95]
10: [51, 61, 62, 63, 85]
11: [20, 38, 96]
12: [33, 74]
13: [68]
14: [31, 37]
15: [23, 59, 66, 78, 82, 87, 93]
16: [1, 29]
17: [0, 9, 30, 41, 53, 59, 74, 90]
18: [50, 56, 68, 69, 77]
19: [84]
20: [0, 5, 11, 82, 93]
21: [65]
22: [54, 59]
23: [15, 25, 68]
24: [27, 47, 52]
25: [23, 81, 87]
26: [6, 30, 53, 92]
```

27: [24, 51, 69, 73]  
28: [31, 41]  
29: [1, 16, 32, 63]  
30: [7, 17, 26, 35, 65, 73, 93, 94]  
31: [3, 14, 28]  
32: [6, 29, 42, 47, 48, 76]  
33: [4, 12, 76]  
34: [48, 49, 91, 96]  
35: [6, 30, 63, 81, 93, 94, 97]  
36: [2, 60]  
37: [14]  
38: [11, 69, 85]  
39: [70, 76, 88, 90]  
40: [56, 64, 68]  
41: [9, 17, 28, 77, 95]  
42: [6, 32, 62, 68, 69, 91, 97]  
43: [51, 59, 60, 84]  
44: [2, 5, 66, 87]  
45: [8, 54, 69, 82, 87]  
46: [5, 9]  
47: [24, 32, 60]  
48: [8, 32, 34, 49, 70, 76, 79, 81]  
49: [7, 34, 48, 76, 87, 96]  
50: [4, 18, 57, 60, 91]  
51: [9, 10, 27, 43, 62, 65, 95]  
52: [24, 83]  
53: [5, 17, 26, 71]  
54: [22, 45, 87, 91]  
55: [78]  
56: [18, 40, 75]  
57: [50, 59, 66, 75]  
58: [78]  
59: [7, 15, 17, 22, 43, 57]  
60: [8, 36, 43, 47, 50]  
61: [6, 10, 62, 67, 94]  
62: [5, 10, 42, 51, 61, 69]  
63: [10, 29, 35]  
64: [0, 40, 70, 94]  
65: [9, 21, 30, 51]  
66: [5, 15, 44, 57, 88, 90]  
67: [6, 61, 84]  
68: [13, 18, 23, 40, 42, 70, 79]  
69: [2, 18, 27, 38, 42, 45, 62]  
70: [39, 48, 64, 68, 82]  
71: [53, 79]  
72: [6, 79]  
73: [27, 30, 90]  
74: [12, 17, 92, 93, 99]  
75: [56, 57, 88]

76: [8, 32, 33, 39, 48, 49, 78]  
77: [18, 41]  
78: [7, 15, 55, 58, 76, 92, 93]  
79: [48, 68, 71, 72]  
80: []  
81: [25, 35, 48]  
82: [1, 15, 20, 45, 70]  
83: [52]  
84: [19, 43, 67, 85, 96]  
85: [10, 38, 84]  
86: [1, 7]  
87: [0, 5, 15, 25, 44, 45, 49, 54, 91, 94]  
88: [39, 66, 75, 90]  
89: [5, 94]  
90: [17, 39, 66, 73, 88]  
91: [5, 34, 42, 50, 54, 87]  
92: [26, 74, 78]  
93: [15, 20, 30, 35, 74, 78]  
94: [30, 35, 61, 64, 87, 89]  
95: [9, 41, 51]  
96: [11, 34, 49, 84]  
97: [35, 42]  
98: []  
99: [74]

---adjacency matrix---

[illegible]

[illegible]



[illegible]

[illegible]

[illegible]

[illegible]

[illegible]