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
Tiulkov Nikita
J4133c

Accepted by
Dr Petr Chunaev

St.Petersburg
2021

1. Goal

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

2. Formulation of the 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.

3. Brief theoretical part

3.1. DFS

Depth-first search (DFS) is an algorithm for traversing a graph. The algorithm starts at a chosen root vertex and explores as far as possible along each branch before backtracking. Applied for: searching connected components, searching loops in a graph, testing bipartiteness, topological sorting, etc. The time complexity of DFS is O(|V| + |E|).

3.2. BFS

Breadth-first search (BFS) is an algorithm for traversing or searching a graph. The algorithm starts at a chosen root vertex and explores all of the neighbour vertices at the present depth prior to moving on to the vertices at the next depth level. It uses an opposite strategy to DFS, which instead explores the vertex branch as far as possible before being forced to backtrack and expand other vertices. Applied for: searching shortest path. The time complexity of BFS is O(|V| + |E|).

4. Results

***I*.** The random symmetric matrix 100x200 was generated by Python library - Networkx. The generated graph is shown on Figure 1.

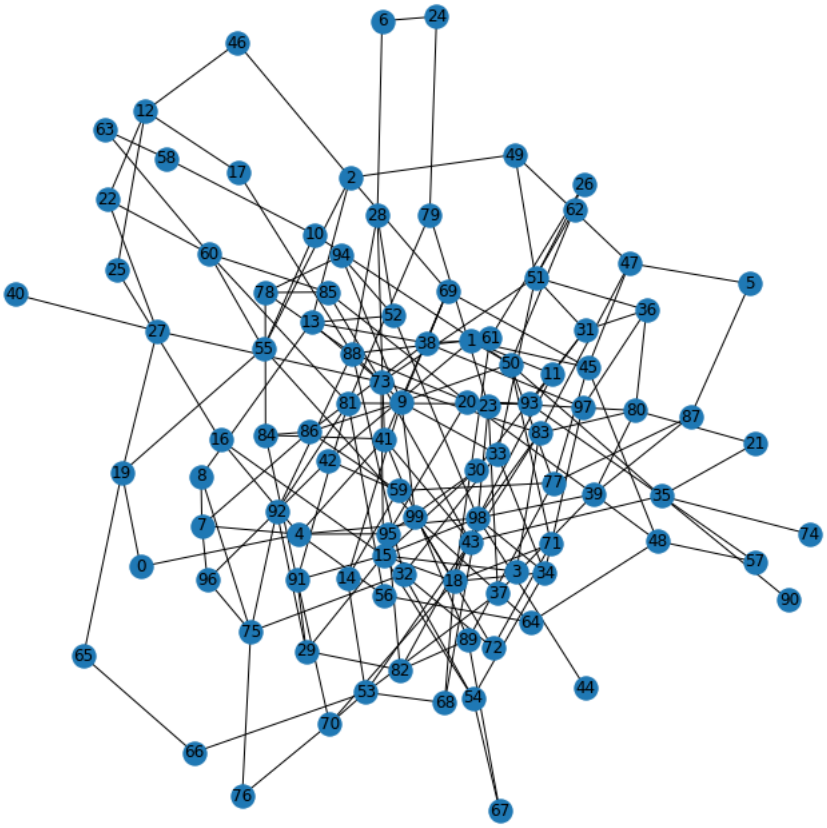

Figure 1: Graph

On Figure 2 we could see the first three rows of adjacency matrix. Node 1 connected to node 4 both in matrix and on graph. So it seems to matrix is well generated.

```
matrix([[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
         1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int64)
```

Figure 2: Adjacency matrix

On Figure 3 we could see the several rows of adjacency list.

```
{0: [19, 4],
 1: [38, 50, 45, 42, 11],
 2: [13, 69, 55, 46, 49],
 3: [83, 15],
 4: [41, 98, 95, 7, 16, 56, 0],
 5: [47, 87],
 6: [28, 24],
 7: [96, 8, 4, 86],
 8: [75, 7, 13],
 9: [94, 50, 33, 99, 86, 88, 69, 14],
 10: [58, 55, 50],
```

Figure 3: Adjacency list

Let's figure out distinguish between each of graph representation.

Adjacency matrix:

- Uses O(n^2) memory
- It is fast to lookup and check for presence or absence of a specific edge between any two nodes O(1)
- It is slow to iterate over all edges
- It is slow to add/delete a node; a complex operation O(n^2)
- It is fast to add a new edge O(1)

Adjacency list:

- Memory usage depends more on the number of edges (and less on the number of nodes), which might save a lot of memory if the adjacency matrix is sparse
- Finding the presence or absence of specific edge between any two nodes is slightly slower than with the matrix O(k); where k is the number of neighbors nodes
- It is fast to iterate over all edges because you can access any node neighbors directly
- It is fast to add/delete a node; easier than the matrix representation
- It is fast to add a new edge O(1)

II. Using depth-first search to find connected components of the generated graph we obtained next results:

Total number of connected components = 1

[0, 19, 55, 60, 81, 93, 35, 50, 77, 87, 39, 80, 36, 31, ..., 23, 37, 46, 5, 1, 11, 54, 67, 89, 44, 14, 66, 65, 91, 62, 21, 10, 58, 63, 90, 74]

This search went in depth as expected. Neighbours was visited only after rollback to starting node. Since we went through all node and vertices the time complexity of this algorithm is O(|V| + |E|).

Using breadth-first search to find a shortest path between two random vertices we obtained next results:

start node is 67, end node is 20

shortest path is [67, 89, 82, 95, 20]

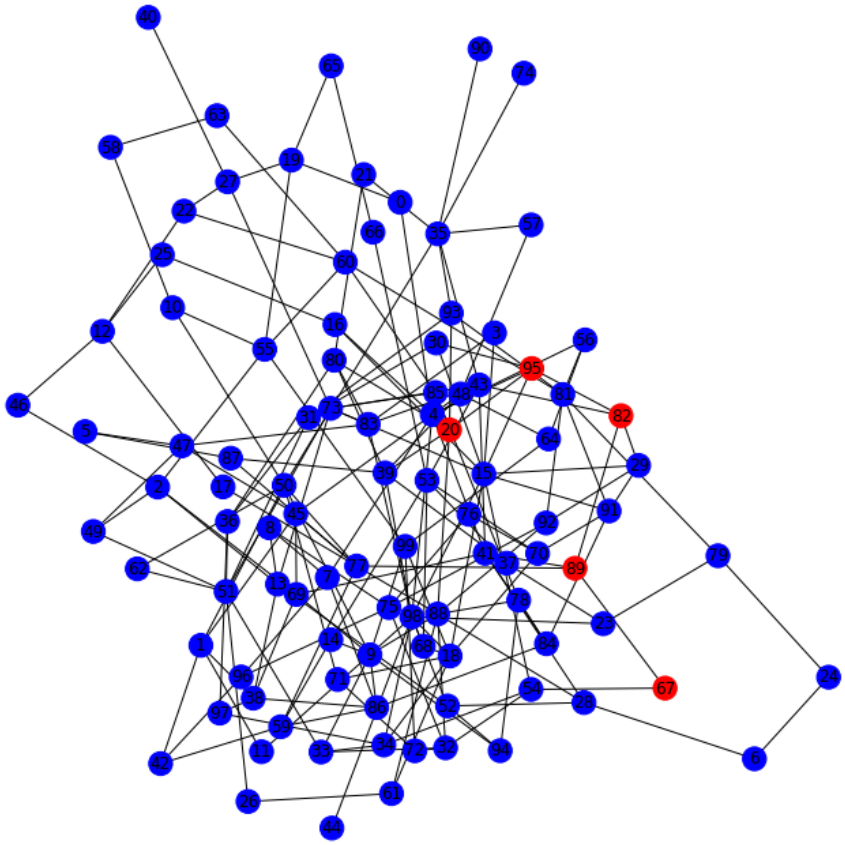

Figure 4: Path between random nodes

This method visits firstly neighbours, so it is conveniently to find shortest path. But since we don not know about shortest path between two random nodes, this algorithm has O(|V| + |E|) time complexity.

III. For adjacency matrix and adjacency list we were using data structures like 2D-array and list correspondingly. DFS based on recursion algorithm (but there is iterative implementation) and list data structure, which contains visited nodes. BFS based on iterative algorithm with queue data structure.

5. Conclusions

As a result of this work, graphs and its main concepts (like representation was studied; DFS and BFS algorithms were investigated in the tasks of finding connected components of the graph and finding a shortest path between two random vertices.

Appendix

The code of algorithms you can find on GitHub: https://github.com/FranticLOL/ITMO_Algorithms