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
Dmitry Grigorev, Maximilian Golovach
J4133c

Accepted by
Dr Petr Chunaev

St. Petersburg
2022

1. Goal of the work

The goal of this work consists of the following points:

- to become familiar with graphs, ways of their representation and some algorithms of traversing around them,
- to apply these algorithms to one generated graph.

2. Formulation of the problem

Suppose we have a graph G which is simple unweighted undirected and built from generation of its adjacency matrix. The subproblems here are:

1. we need to get adjacency list from the matrix,
2. then discuss when these two ways of graph representation are more convenient as opposed to the second one,
3. apply Depth-first-search to find connected components in the graph,
4. apply Breadth-first-search to find one of the shortest paths between two arbitrary vertices in the graph,
5. analyze the obtained results.

3. Brief theoretical part

The theory is taken from [1].

3.1. Basics of graphs

Any graph can be formally defined as follows:

Definition 1. A graph $G = (V, E, g)$ is a discrete structure which is defined by a set of **vertices** (or **nodes**) $V = V(G)$, a set of **edges** $E = E(G)$ and a **relation** g that associates each edge with 2 vertices from V.

An edge from V is incident to two nodes which are called its **endpoints**. The number of vertices and edges of a graph are called its **order** and **size** correspondingly. It is possible to name an edge by the nodes it is incident to: for example, if two nodes $u, v \in V$ are connected, then we name an edge $(u, v) \in E$ as the edge which connects these two vertices.

Definition 2 (self-edge)**.** A **self-loop** is an edge with the same endpoints.

Definition 3 (multiple edge)**.** A number of edges with the same endpoints are called **multiple edges**.

Definition 4 (simple graphs and multigraphs)**.** **Simple** graph is a graph without any self--loops and multiple edges. Otherwise, this is a **multigraph**.

In order to traverse around graphs, there are some definitions in the theory of graphs.

Definition 5 (walks, trails and paths)**.** A **walk** $W = (v_0, e_1, v_1, e_2, \dots, v_{n-1}, e_n, v_n)$ of a graph G with the **initial** node v_0 and the terminal one v_n is an alternating sequence of nodes and edges where each edge e_i is incident to nodes v_{i-1} and v_i.
A **trail** is a walk with no repeated edges.
A **path** is a trail with no repeated vertices save the initial and terminal ones.

It is worth mentioning that in an arbitrary graph two random vertices may have no walks connecting them.

Definition 6 (connectivity of graph)**.** A graph G is called **connected** if any pair of two vertices has a path connecting them. Otherwise, this graph is **disconnected**.

Disconnected graph consists of some connected subgraphs called **(connected) components**. Distance $d(u, v)$ between two nodes u and v is the length of the shortest path between them.

3.2. Representation ways of graphs

Adjacency matrix

An **adjacency matrix** $A = (a_{ij})_{i,j=1}^{n}$ of a simple graph G of order n is a matrix where $a_{i,j} = 1$ if there is an edge which joins vertex i with vertex j and $a_{i,j} = 0$ otherwise. In case of an undirected graph its adjacency matrix is symmetric. To store such a matrix it requires $O(n^2)$ of memory.

Adjacency list

An array of lists with each list representing a vertex and its neighbors (adjacent nodes) in a linked list is a **adjacency list**. If n and m are order and size of a graph G, then the storage of adjacency list requires $O(n + m)$ memory.

3.3. Two basic algorithms of traversing around graphs

Depth First Search

The idea if this algorithm is quite simple: move further firstly as far as it possible. The adjacent nodes (with respect to the starting one) in this algorithm are processed later than those located far away. The algorithm is described in 1 where the iterative approach is used with application of stack data structure.

Algorithm 1 DFS (Iterative approach)

```
Require: graph $G(V, E)$, starting vertex $v$
  stack $S \leftarrow \emptyset$, boolean array $isVisited[n]$
  # initialization
  for each $u \in V$ do
    $isVisited[u] \leftarrow false$
  end for
  $Push(v, S)$
  while $S \neq \emptyset$ do
    $u \leftarrow Pop(S)$
    if $isVisited[u] = false$ then
      $isVisited[u] \leftarrow true$
      for each $w \in V : (u, w) \in E$ do
        $Push(w, S)$
      end for
      Process $u$
    end if
  end while
Ensure: some result of processing of the graph $G$
```

Time complexity of this algorithm is $O(n + m)$ and space complexity is $O(n)$ due to storage of stack and additional array of statuses.

Breadth First Search

The idea of BFS is opposite to DFS: all close vertices are processes at first and the farthest ones at last. To organize such a way of traversing we have to use queue data structure. The algorithm is presented in 2. Here an array of processing status is also used in order to prevent each node from extra inclusion into the queue.

Algorithm 2 BFS

```
Require: graph $G(V, E)$, starting vertex $v$
  queue $Q \leftarrow \emptyset$, array boolean $Status[n]$
  # initialization
  # Status 0 means 'not visited', 1 — 'in queue' or 'visited'
  for each $u \in V \backslash \{v\}$ do
      $Status[u] \leftarrow false$
  end for
  $Status[v] \leftarrow true$
  $Enqueue(v, Q)$
  while $Q \neq \emptyset$ do
      $u \leftarrow Dequeue(Q)$
      for each $w \in V : (u, w) \in E$ do
          if $Status[w] = false$ then
              $Status[w] \leftarrow true$
              $Enqueue(w, Q)$
          end if
      end for
      Process $u$
  end while
Ensure: some result of processing of the graph $G$
```

The important property of the BFS in case of simple undirected unweighted it builds shortest paths automatically thank to its construction. The time complexity is $O(n + m)$ and space complexity is $O(n)$ due to storage of queue and additional array of statuses.

4. Results

4.1. Ways of representation

So, first of all, we generated one random adjacency matrix which provide for us the graph depicted in the figure 1. As soon as the adjacency matrix was obtained, we construct the adjacency list of our graph.

Let us analyze the current results. In the graph 2 several rows of the adjacency matrix

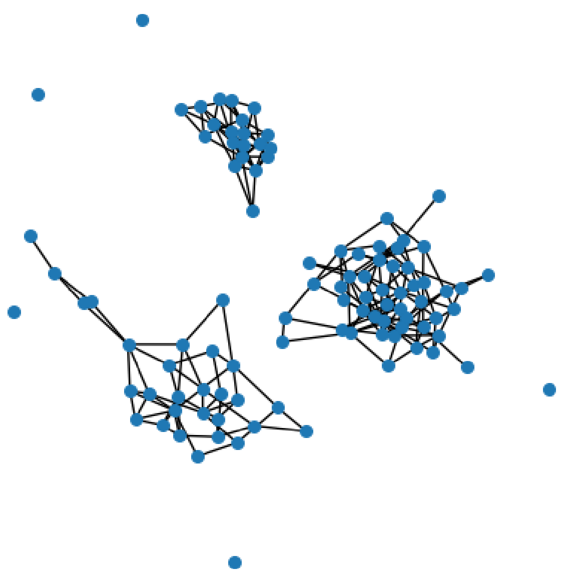
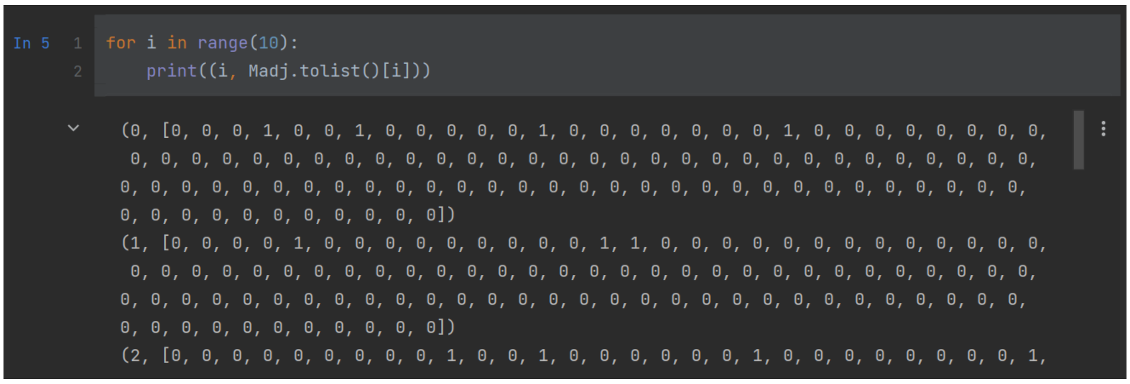


Figure 1: The generated graph

are presented. At the same time in the figure 3 some pairs $(node, adjacent\ nodes)$ of adjacency list are shown. As has been seen, the adjacency matrix looks sparse and it may be inefficient to store it for such a graph and the use of adjacency list is more profitable. In spite of that, determining the adjacency of two arbitrary nodes requires $O(n)$ time in case of adjacency lists and $O(1)$ time when one uses adjacency matrix so in case of dense graphs it is more efficient to use the latter one.



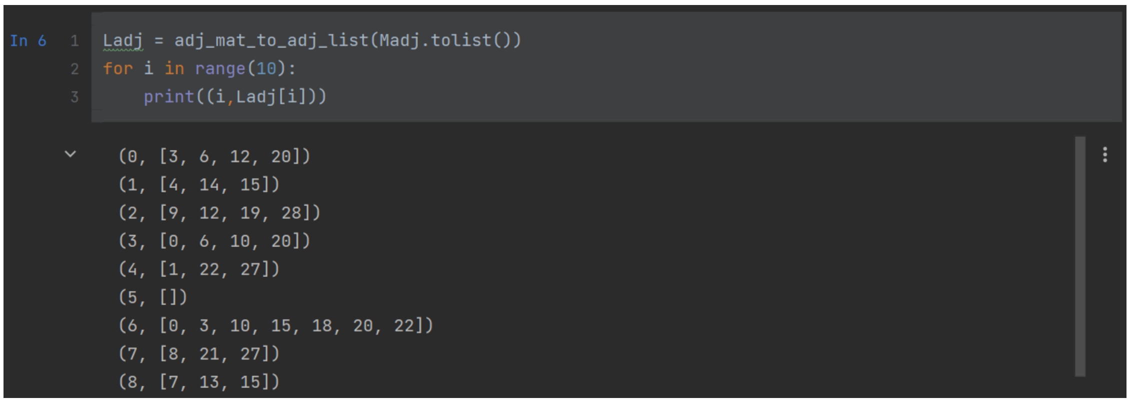
```
In 5
for i in range(10):
    print((i, Madj.tolist()[i]))
```

```
(0, [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 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])
(1, [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 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])
(2, [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
```

Figure 2: Some pairs of nodes and corresponding rows in the adjacency matrix

4.2. Search algorithms

To determine connected components in the graph and to find some paths between arbitrary pairs of vertices we have applied DFS and BFS correspondingly. In the picture

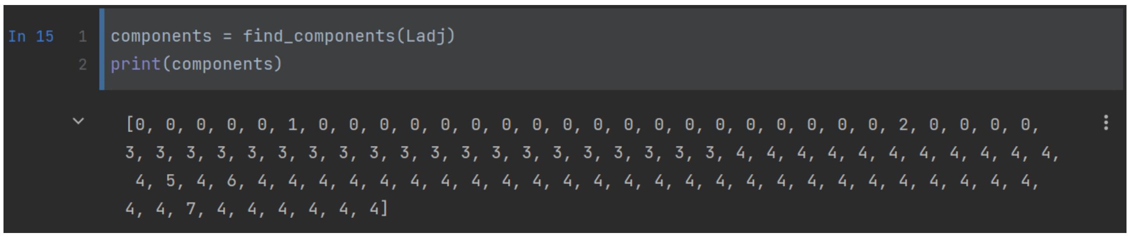


```
In 6
Ladj = adj_mat_to_adj_list(Madj.tolist())
for i in range(10):
    print((i,Ladj[i]))
```

```
(0, [3, 6, 12, 20])
(1, [4, 14, 15])
(2, [9, 12, 19, 28])
(3, [0, 6, 10, 20])
(4, [1, 22, 27])
(5, [])
(6, [0, 3, 10, 15, 18, 20, 22])
(7, [8, 21, 27])
(8, [7, 13, 15])
```

Figure 3: Some pairs of nodes and corresponding neighbors in the adjacency list

4 the output of components search by DFS is illustrated. As one can see, there are 8 components. On order to visualize the correctness of this algorithms, we provide the figure 5 of the colored graph. Indeed, the algorithm performed right according to the figure.



```
In 15
components = find_components(Ladj)
print(components)
```

```
[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, 2, 0, 0, 0, 0,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
 4, 5, 4, 6, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 7, 4, 4, 4, 4, 4, 4]
```

Figure 4: Result of DFS application on the graph

As for BFS, we applied it to specific pairs of nodes using the information from the adjacency lists. The results with found paths are provided in the figure 6. For one pair, as expected, no paths were found and here one can figure it out from the results of DFS in the picture 4. For another one with two adjacent vertices the found path is just the corresponding edge which connects them. The latter pair here is connected with a bit longer path. Moreover, contingency lists for the nodes in this path are provided.

5. Data structures and design techniques used in algorithms

In the random variables generation and building of adjacency matrix the Python's package **Numpy** is used due to its convenience. As a tool of drawing the graphs the **networkx** package is used. In the BFS and DFS the Python's list data structure is applied to organize queue and stack. In the BFS shortest path finding the list of nodes' predecessors is also used in order to remember all built paths (instead of storage of these whole paths themselves).

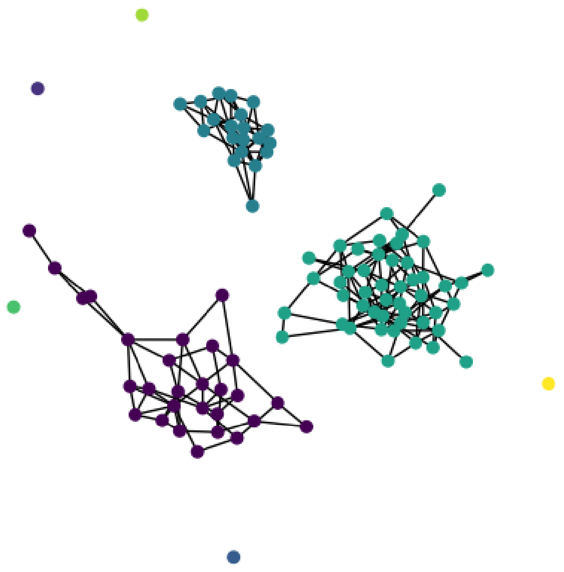
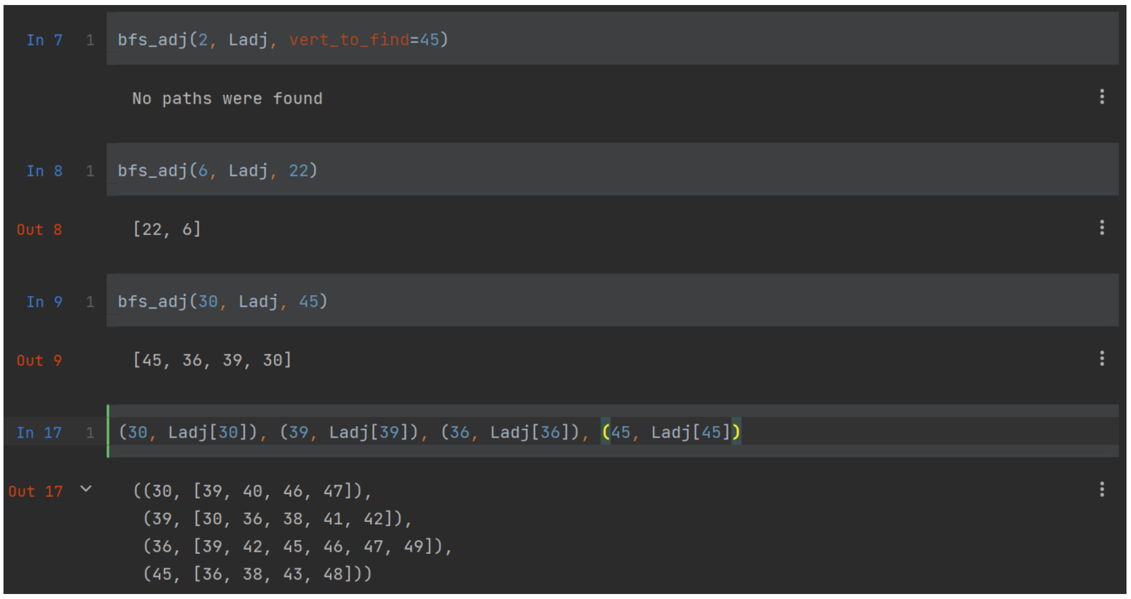


Figure 5: The graph with colored vertices with respect to component each of them belongs to



```
In 7   bfs_adj(2, Ladj, vert_to_find=45)

       No paths were found

In 8   bfs_adj(6, Ladj, 22)

Out 8  [22, 6]

In 9   bfs_adj(30, Ladj, 45)

Out 9  [45, 36, 39, 30]

In 17  (30, Ladj[30]), (39, Ladj[39]), (36, Ladj[36]), (45, Ladj[45])

Out 17 ((30, [39, 40, 46, 47]),
        (39, [30, 36, 38, 41, 42]),
        (36, [39, 42, 45, 46, 47, 49]),
        (45, [36, 38, 43, 48]))
```

Figure 6: Some results of BFS application on the graph: no paths were found, path between neighbors and a bit long path

6. Conclusion

As the result of this work, we become familiar with graphs, their ways of representation and some ways of traversing. The ways of representation were compared by convenience in practice and in theory according to the density or sparsity of graphs. The results of BFS and DFS were analyzed and considered as plausible. The implementations of BFS, DFS, adjacency list construction and random adjacency matrix generation are provided. Data structures and design techniques which were used in the implementations were discussed. The work goals were achieved.

7. Appendix

Algorithms implementation code is provided in [2].

Bibliography

1. Erciyes K. Guide to Graph Algorithms. — Springer Cham, 2018. — ISBN: 9783319732350.
2. Grigorev D., Golovach M. Code repository. — `https://github.com/dmitry-grigorev/AlgoAnalysisDevelopment`. — 2022.