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
Anastasia Miroshnikova
j4133c
Accepted by
Dr Petr Chunaev

St. Petersburg
2020

Goal

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

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.

Brief theoretical part

Graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called link or line). In **simple graph** two vertices can be connected directly only with one edge. **Unweighted graph** is such a graph where all the edges have the same weight (or no weight at all).

Graphs can be represented as **adjancecy matrix** – square matrix, where number of rows and columns equals to number of vertices (|V|). Each value in a adjancecy matrix is equal to the weight of the edge that connects respective vertices. An adjacency matrix takes up $\Theta(|V|^2)$ storage.

Another common representation is **adjacency list**, the list of lists. Each sub-list with index u corresponds to a vertex u and contains a list of edges (u, v) that originate from u. For simple graphs such sub-list can contain only indices of vertices v, adjancent to vertex u. An adjacency list takes up $\Theta(|V| + |E|)$ space.

Adjacency lists are quicker for the task of giving the set of adjacent vertices to a given vertex than **adjancency matrices** – O(|neighbors|) for the former vs O(|V|) for the latter.

Breadth-first search (**BFS**) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

It uses the opposite strategy of **depth-first search**, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.

Both search strategies make O(|V| + |E|) steps to walk through complete graph.

Results

Generating a graph

The simple random graph with 100 nodes and 200 edges was generated using Python 3.6.9 programming language. The strategy to generate it was as follows: let us sample up to |E| indices which belong to the upper triangle of adjacency matrix, them map these indices to the adjacency matrix, initially filled with zeros, and also fill the reflected indices.

The first row of the matrix looks as follows:

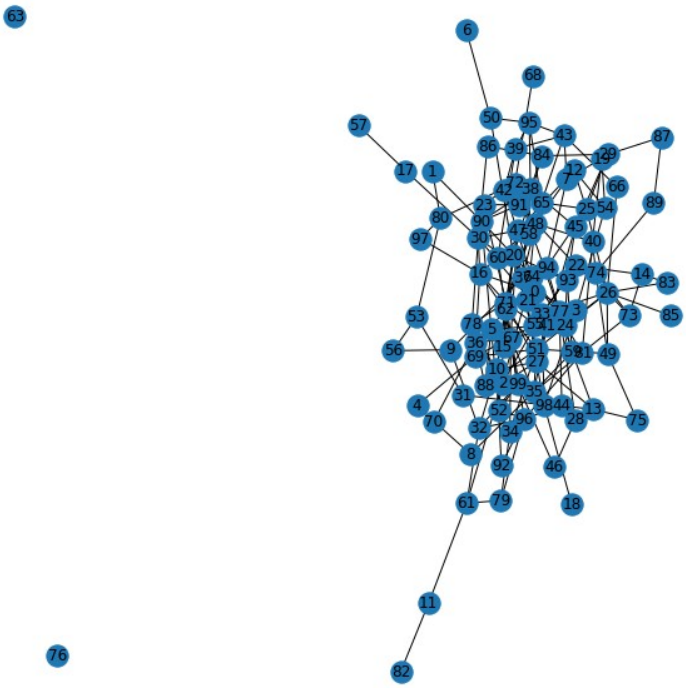
```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The matrix was then converted to the adjacency list (actually I used adjacency dict because it looks better). The first 3 values of the list are:

```
0: {16, 41, 3}
1: {80, 90}
2: {64, 33, 78, 27, 61}
```

The adjacency matrix of this graph looks much less informative because the graph is sparse, therefore the matrix has lots of zeros. But if we were to work with small weighted directed graph, the matrix representation would be much simpler and more informative than list representation.

The graph was visualized using *networkx* library. It can be seen that the graph has 3 connected components – the big one containing 98 vertices and two components containing 1 vertex each. The most outstanding from the connected component is the vertex number 82. I will further analyze the shortest paths in the graph it will be shown that most of the longest shortest paths end in this vertex.



Search of connected components

Depth-first search strategy was used to find connected components of the graph. The algorithm was implemented with non-recursive design and makes use of a queue – the data structure which is able to make operations “add element to end” and “get the first element” in constant time.

The algorithm has an external loop (iteration over all elements in a graph, the algorithm finds an unseen vertex which will be a root for the search of all vertices connected to it), and internal loop, where it iterates over all elements in a dynamic queue.

To count the number of constant operations the following strategy was used: when the algorithm either checks whether the vertex is in a queue, or draws a vertex from queue and puts it into a list containing all vertices for the current component, 1 is added to the counter.

The algorithm successfully found all 3 connected components. If algorithm started search in isolated vertices, only 1 operation was required. When it started to walk over big component of the graph, it took 498 operations. In total, it gives us (100 vertices + 2 * 200 edges), which is not quite O(|V| + |E|) complexity, but rather O(|V| + 2|E|). This happens because each edge in undirected graph actually represents two directed edges – the first one indicates that the vertex u is a child of vertex v, and the second one – that vertex v is a child of vertex u. Therefore, when the algorithm walks through the graph, it has to look at each vertex twice.

Search for shortest paths in a graph

Breadth-first search strategy was used to find shortest paths between all the components. The algorithm was implemented in non-recursive design, as well as depth-first search, and also makes use of queue data structure. The search of all possible shortest paths between all vertex pairs was conducted independently – none of the searches used any information from the previous search.

The number of operations was calculated as follows: if the algorithm draws an element from a queue, or checks whether the element is already visited, or it should be added to a queue, 1 is added to the counter.

The longest shortest path in a graph appeared to be 7 edges. Seventeen vertex pairs have this shortest path length, fifteen of them ending in vertex 82. It required from 263 operations to 495 operations to find these paths.

The number of operations is greatly influenced by the order of the vertices. It was found that when we try to find a path between the isolated vertex and a vertex from connected component, it takes only one operation to find out if there is any path to any vertex if we started from an isolated vertex. In contrary, algorithm has to visit every node and every edge in connected component to check that there is no way to an isolated vertex, and it takes 498 operations. This is again O(|V| + 2|E|) complexity.

Conclusions

Random simple graph with 100 nodes and 200 edges was generated. The analysis shown that there happened to be 3 connected components, two of them being isolated nodes. The depth-first search had to make O(|V| + 2|E|) operations to visit all the vertices and check all the bidirectional bonds between them.

The analysis of the shortest paths between the nodes shown that the longest shortest path length is 7 edges, and 17 node pairs have it. Vertex 82, which is visibly outstanding from the others, happened to be the end of 15 of them. The breadth-first search can also make up to O(|V| + 2|E|) operations to find the shortest path between the nodes since it also visits all the nodes and checks every bidirectional bond.

Appendix

Source code can be found at https://github.com/Miffka/algo_itmo/tree/master/task5.