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

Dmitriy Prusskiy

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg

2021

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 a 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 **adjacency matrix** – square matrix, where number of rows and columns equals to number of vertices ($|V|$). Each value in an adjacency 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 a sub-list can contain only indices of vertices *v*, adjacent 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 adjacency 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. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

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 the complete graph.

Results

The results were obtained using Python 3.8.5. The NetworkX library was used to draw graphs, obtain adjacency lists, and validate search algorithms.

The strategy to generate a random adjacency matrix was the following: generating 200 unique pairs (row, column), then filling these and symmetric cells with ones, other cells equal to zeros.

The first row of the result adjacency matrix is following:

[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., 1., 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., 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.]

The first three values of adjacency list are following:

{

0: [22, 58, 83],

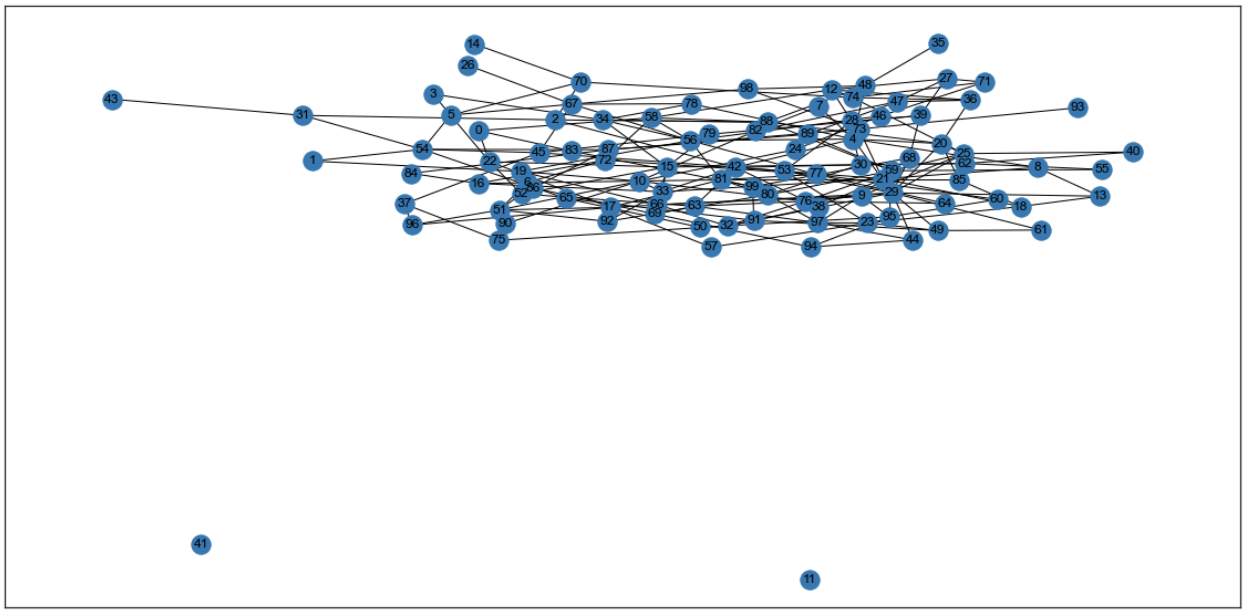
1: [19, 54],

2: [31, 42, 70, 88]

}

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 graphs, the matrix representation would be much simpler and more informative than list representation.

Visualisation of the graph:



Finding the shortest path between two vertices using BFS

To find the shortest path, one can store the path to each visited vertex during breadth-first search. The algorithm ends when the next visited vertex is the target. If BFS ends and none of the visited vertices is equal to the target, then the source and target vertices are in different connected components and there is no path between them.

The algorithm correctly finds all paths. The results of the algorithm match the results of the networkx.shortest_path method. Algorithm also correctly works when source and target vertices are in different connected components.

Finding connected components using DFS

To search for connected components in the graph, the following algorithm was used: starting from a random vertex, DFS is used, then all visited vertices are saved as one component, then these vertices are removed from the adjacency list. All of these steps are repeated until the adjacency list is empty.

In the picture above, one can see that the graph consists of three connected components. This algorithm accurately identifies all connected components. The results of the algorithm match the results of the networkx.components.connected_components method.

Conclusions

The goal of this study was to use different representations of graphs and basic algorithms on graphs. Analysis of the obtained results shows that different representations of graphs may be helpful in different tasks. When a graph is sparse, it is better to use an adjacency list, but for small weighted directed graphs it is better to use an adjacency matrix. Basic algorithms of breadth- and depth-first search can help to solve different tasks on graphs, such as finding the shortest path between two vertices or finding connected components.

Appendix

Source code can be found at https://github.com/T1MAX/itmo_algorithms/tree/main/task_5