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
Abdurakhimov Muslimbek
J4133c
Accepted by
Dr Petr Chunaev

St. Petersburg
2023

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 Breadthfirst 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

Graphs are fundamental data structures used to represent and analyze various relationships and networks. In this context, we'll explore the basic concepts of graph representations and two common graph traversal algorithms: Depth-First Search (DFS) and Breadth-First Search (BFS).

Graph Representations:

Adjacency Matrix: An adjacency matrix is a square matrix used to represent a graph. Each row and column correspond to a vertex, and the elements of the matrix indicate whether an edge exists between the corresponding vertices. In an undirected unweighted graph, the matrix contains 0s and 1s, where 1 represents the presence of an edge between vertices.

Convenience: Adjacency matrices are convenient for checking the existence of edges between specific vertices and for performing operations like edge deletion and addition. They are suitable for dense graphs where many edges exist.

Adjacency List: An adjacency list is a data structure that represents a graph as a collection of lists. Each vertex has a list of its adjacent vertices. In an undirected unweighted graph, the lists contain the neighboring vertices.

Convenience: Adjacency lists are convenient for graphs with fewer edges or those that are sparse. They require less memory compared to adjacency matrices. They are useful for tasks involving vertex-centric operations and traversal algorithms.

Depth-First Search (DFS):

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It starts at a source vertex and visits all reachable vertices along a single path before exploring other paths.

Purpose: DFS is primarily used to discover connected components within a graph. It helps identify groups of vertices that are mutually reachable.

Breadth-First Search (BFS):

BFS is another graph traversal algorithm that explores the vertices in layers. It starts at a source vertex and explores all its neighbors before moving on to their neighbors.

Purpose: BFS is often used to find the shortest path between two vertices in an unweighted graph. It ensures that the shortest path is found before longer paths.

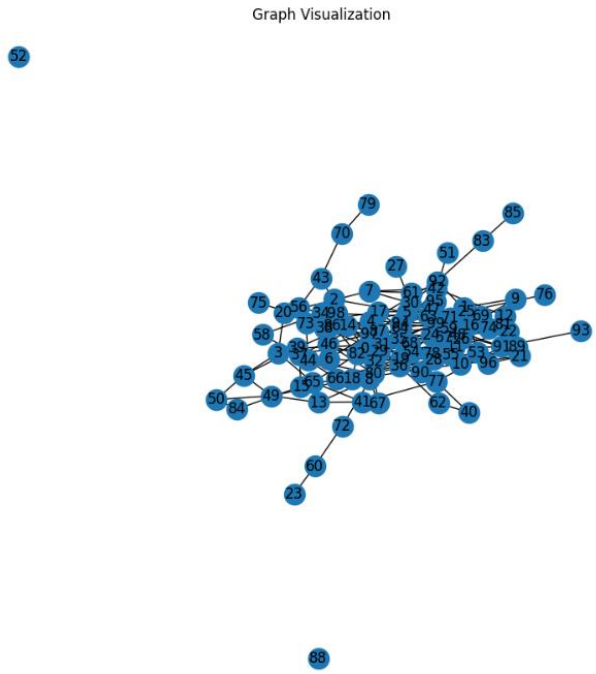
Results

PART I.

In this laboratory task, we focused on graph representations and basic graph algorithms, specifically Depth-First Search (DFS) and Breadth-First Search (BFS).

Generating Graph and Creating Representations:

We started by generating a random adjacency matrix for a simple undirected unweighted graph with 100 vertices and 200 edges. This matrix was designed to be symmetric and consisted of 0s and 1s, representing the absence or presence of edges between vertices. Additionally, we converted the adjacency matrix into an adjacency list.



Picture 1 – Random Undirected Unweighted Graph (100 Vertices, 200 Edges)

These representations serve different purposes:

The adjacency matrix is efficient for checking edge existence between specific vertices and for operations like edge manipulation in dense graphs. The adjacency list is suitable for graphs with fewer edges, and it facilitates vertex-centric operations and traversal algorithms.

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 1, 0],
       ...,
       [0, 0, 0, ..., 0, 1, 0],
       [0, 0, 1, ..., 1, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=int32)
```

Picture 2 – Random Undirected Unweighted Matrix (100 Vertices, 200 Edges)

In this section, we generated a random undirected unweighted graph with 100 vertices and 200 edges using the NetworkX library. To represent the connectivity between vertices, we created an adjacency matrix. An adjacency matrix is a square matrix where each row and column corresponds to a vertex, and the values (0s and 1s) in the matrix indicate whether there is an edge connecting the corresponding vertices.

PART II

After that we started to work on the second part of the task. In this section, we applied two fundamental graph traversal algorithms, Depth-First Search (DFS) and Breadth-First Search (BFS), to analyze the properties of the random graph we generated.

Connected Components using DFS:

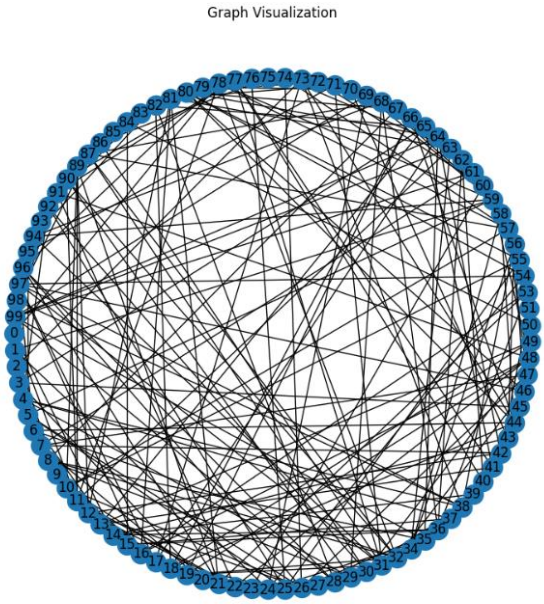
We employed Depth-First Search to identify the connected components within the graph. Connected components are subsets of vertices where each vertex is reachable from any other vertex in the same subset.

The DFS algorithm helped us discover how the graph was partitioned into different connected components. In our case, the number of connected components and their sizes varied, highlighting the graph's structural diversity.

Shortest Path using BFS:

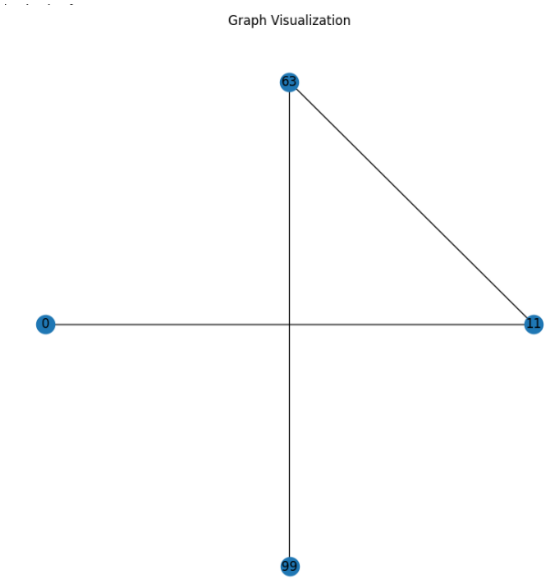
Using Breadth-First Search, we found the shortest path between two randomly chosen vertices within the graph.

BFS allowed us to explore the graph layer by layer, starting from the source vertex and expanding outward. This traversal strategy ensures that we find the shortest path first.



Picture 3 – Depth-First Search Result Visualization

After that, we applied the Breadth-First Search (BFS) algorithm to find the shortest path between two randomly selected vertices in our graph. The BFS algorithm systematically explores the graph by visiting all the vertices at the current level before moving on to the vertices at the next level. This characteristic makes it particularly useful for finding the shortest path between two nodes in an unweighted graph.



Picture 4 – Breadth-First Search Result Visualization

PART III.

In the implementation of Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms, several data structures and design techniques are employed to efficiently explore and traverse a graph. Below, we describe the key data structures and design elements used in these algorithms:

Depth-First Search (DFS):

Stack Data Structure: DFS uses a stack (either implemented explicitly or through a recursive call stack) to keep track of the vertices to explore. When a vertex is visited, it is pushed onto the stack, and exploration proceeds by selecting vertices from the stack.

Recursion: DFS can be elegantly implemented using recursion. Each recursive call corresponds to visiting a new vertex, and the call stack effectively simulates the stack data structure. When a vertex is visited, the algorithm recursively explores its neighbors.

Breadth-First Search (BFS):

Queue Data Structure: BFS employs a queue to maintain the order of vertices to be visited. The first-in-first-out (FIFO) nature of the queue ensures that vertices are visited in a level-wise manner, exploring all neighbors at one level before moving to the next.

Common Design Elements for Both Algorithms:

Graph Representation: Both DFS and BFS operate on a graph data structure. The graph is typically represented using an adjacency list or an adjacency matrix. An adjacency list is often preferred for sparse graphs, as it requires less memory and provides faster access to neighbors

Visited Set: To keep track of visited vertices and avoid revisiting them, both algorithms maintain a set or array to mark visited vertices. This is crucial to prevent infinite loops in the case of cyclic graphs.

Start and Target Vertices: In BFS for shortest path, the algorithm is provided with the starting vertex (source) and the target vertex (destination) to find the shortest path between them.

Parent or Predecessor Information: In BFS, information about the parent or predecessor of each vertex is stored. This information is used to reconstruct the shortest path once the target vertex is reached.

Termination Condition: Both algorithms need a termination condition. For DFS, this typically involves checking that the stack is empty. For BFS, it may involve reaching the target vertex or exploring all reachable vertices.

Conclusions

DFS is suitable for identifying connected components because it explores as deeply as possible into one component before moving to another. It's ideal for tasks like finding strongly connected components in directed graphs.

BFS, on the other hand, excels at finding the shortest path between two vertices in an unweighted graph. It guarantees the shortest path is found first. Both algorithms are efficient and widely used in various applications, such as network analysis, maze solving, and web crawling. The choice between them depends on the specific problem and the desired outcome.

Appendix

GitHub: site. – URL:
https://github.com/MrSimple07/AbdurakhimovM_Algorithms_ITMO/tree/main/Task5