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

Zakhar Pinaev

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)

Problems and methods

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 theory is an extensive branch of discrete mathematics in which the properties of graphs are systematically studied. A graph is a system of objects of arbitrary nature (vertices) and connectives (edges) connecting some pairs of these objects.

The widespread use of graph theory in computer science and information technology can be explained by the concept of a graph as a data structure. In computer science and information technology, a graph can be described as a non-linear data structure. In nonlinear data structures, elements are located at different levels of the hierarchy and are divided into three types: original, generated, and the like.

A path in a graph is a finite sequence of vertices in which each vertex (except the last) is connected to the next in the sequence of vertices by an edge. If any two vertices in a graph are connected by a path, then such a graph is called connected. We can consider such a subset of the vertices of the graph such that every two vertices of this subset are connected by a path, and no other vertex is connected to any vertex of this subset. Each such subset, together with all the edges of the original graph connecting the vertices of this subset, is called a connected component.

Graphs in which all edges are links, that is, the order of the two ends of an edge in the graph is not essential, are called undirected. A graph without arcs, loops and multiple edges is called simple. A weighted graph is a graph whose vertices and / or edges are assigned "weights" - usually some numbers.

Results

I. In the first task, it was necessary to generate a random adjacency matrix for a simple undirected unweighted graph with 100 vertices and 200 edges. So, in this case, only 0 and 1 were supposed to be elements of the matrix. Figure 1 shows a visualized graph obtained on the basis of a randomly generated adjacency matrix.

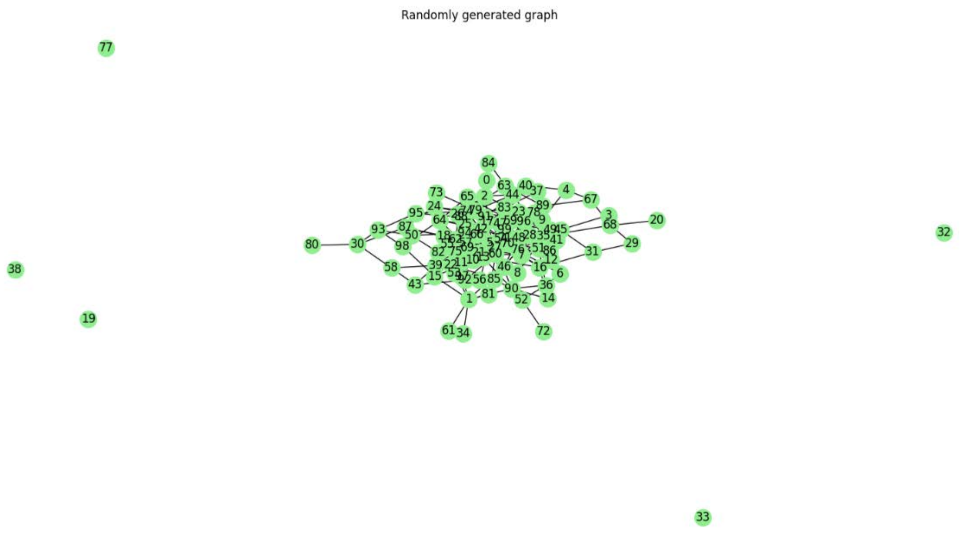
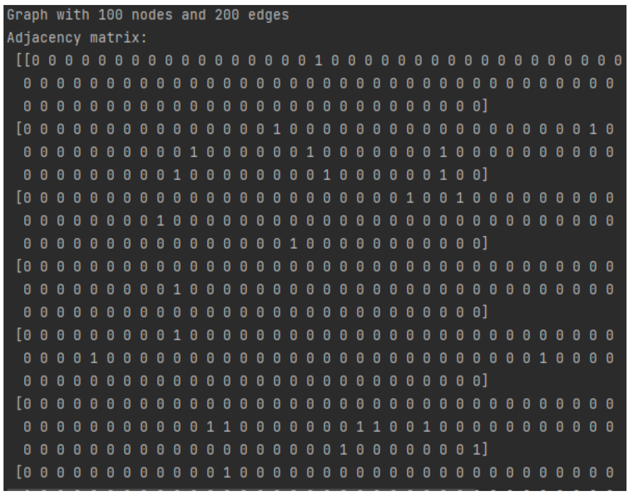


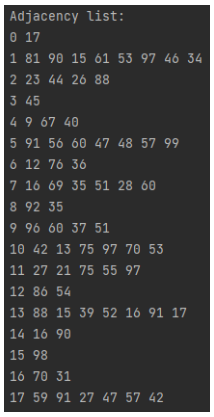
Figure 1 – graph obtained by generating a random adjacency matrix

Further, the aforementioned adjacency matrix was transformed into an adjacency list. Figures 2, 3 show the output of several rows of the adjacency matrix and adjacency list, respectively.



```
Graph with 100 nodes and 200 edges
Adjacency matrix:
 [[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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 1 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 1 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 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 1 0 0 1 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 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 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 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 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 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 1 0 0 0 0 0 0 0 1 1 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 1 0 0 0 0 0 0 0 1]
 [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
```

Figure 2 – output of the first rows of a randomly generated adjacency matrix for a simple undirected unweighted graph with 100 vertices and 200 edges



```
Adjacency list:
0 17
1 81 90 15 61 53 97 46 34
2 23 44 26 88
3 45
4 9 67 40
5 91 56 60 47 48 57 99
6 12 76 36
7 16 69 35 51 28 60
8 92 35
9 96 60 37 51
10 42 13 75 97 70 53
11 27 21 75 55 97
12 86 54
13 88 15 39 52 16 91 17
14 16 90
15 98
16 70 31
17 59 91 27 47 57 42
```

Figure 3 - Output of the first rows of the adjacency list obtained by transforming the mentioned adjacency matrix

In order to answer the question "Which purposes is each representation more convenient for?" it is necessary to consider the differences between an adjacency matrix and an adjacency list.

Adjacency Matrix:

- uses O(n^2) memory;
- it is fast to look up 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).

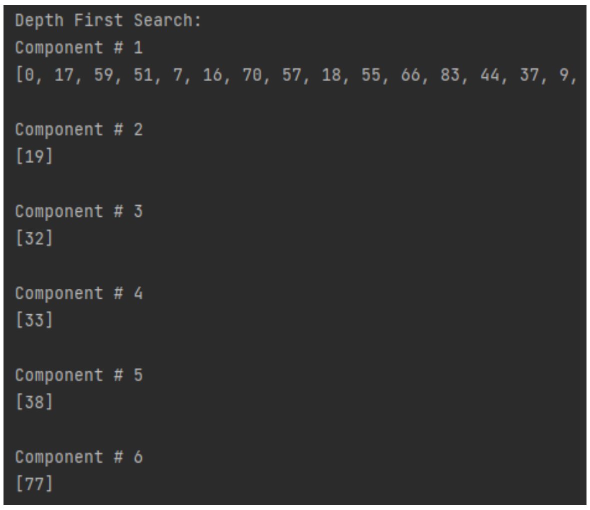
Adjacency List:

- 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 any node neighbors can be accessed directly;
- it is fast to add/delete a node – easier than the matrix representation;

So, if there are not many connections relative to the total number of nodes, it is better to use an adjacency list. If the graph is densely connected, then an adjacency matrix will be a better fit. The primary concern is memory usage. Namely, from the one hand, it is needed to store a list of node-ID’s or node-pointers in the adjacency list, from the other hand it is necessary to store boolean flags inside the adjacency matrix.

To sum up, adjacency matrix is a good solution for dense graphs, which implies having constant number of vertices, but on the other hand, the adjacency list is a good solution for sparse graphs, and it allows to change the number of vertices more efficiently, than if using an adjacent matrix.

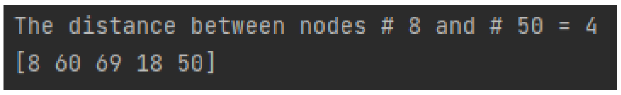
II. Firstly, in the second task it was necessary to find connected components of the graph using depth-first search. Figure 4 shows the output of the program for finding connected graph components.



```
Depth First Search:
Component # 1
[0, 17, 59, 51, 7, 16, 70, 57, 18, 55, 66, 83, 44, 37, 9,

Component # 2
[19]

Component # 3
[32]

Component # 4
[33]

Component # 5
[38]

Component # 6
[77]
```

Figure 4 - output of the program for searching for connected components of a graph using depth-first search (for each component, the vertices included in it are given)

Further, in the second task, it was necessary to find the shortest path between two random vertices using breadth-first search. Figure 5 shows the output of the program for finding the shortest path between two random vertices.



```
The distance between nodes # 8 and # 50 = 4
[8 60 69 18 50]
```

Figure 5 – shortest path between two random vertices (8 and 50) and its length

As a result, both algorithms considered in the work were successfully applied in the second task. First, looking at Figures 1 and 4, it can be seen that the depth-first search algorithm allows to correctly determine the number of connected components of the graph. Secondly, during several experiments and a detailed study of the graph, it was found that the breadth-first search algorithm allows to correctly determine the shortest path and its length, although it is difficult to see by looking at the graph itself.

III. Throughout the work, the focus is on a data structure such as a graph. The graph itself can be built based on an adjacency list and adjacency matrix. Obviously, in Python, an adjacency list was most sensibly implemented with a data structure such as a list. Moreover, the adjacency matrix was also implemented using a list. However, in terms of storage, each of the data structures had their own characteristics. For example, in an adjacency list, vertices are represented by a numeric index in the list, in which each item refers to a list of neighboring vertices. On the other hand, in the adjacency matrix, the vertices are represented by a numerical index in the list, in which each element refers to the list of all other vertices in the graph, and the elements take on the values 0 or 1, reflecting the absence or presence of an edge between the vertices, respectively. This difference in data storage leads to the differences described in task 2.

Conclusion

As a result of the work, various representations of graphs and basic algorithms on graphs, such as depth-first search and breadth-first search, have been investigated.

The adjacency matrix and adjacency list are considered as graph representations. During the experiments, it was revealed that it is necessary to choose one or another graph representation depending on the task, since each of the representations has its own advantages and disadvantages, described in the results of task 2.

Further, in the course of researching algorithms on graphs, depth-first search in the problem of finding connected components of a graph, as well as breadth-first search in the problem of determining the shortest path between two vertices were successfully implemented. These algorithms can be useful for creating routing tables in networks, in navigation systems, for finding public transport routes and etc.

Appendix

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
```

```python
import random
def convert_to_adjacency(matrix):
    start = 0
    res = []
    lst = []
    n = len(matrix)
    for i in range(n):
        res.append(lst*n)
    while start < n:
        y = matrix[start]
        for i in range(len(y)):
            if y[i] == 1:
                res[start].append(i)
        start += 1
    return res
def convert_to_matrix(graph):
    matrix = []
    for i in range(len(graph)):
        matrix.append([0]*len(graph))
        for j in graph[i]:
            matrix[i][j] = 1
    return matrix
def main():
    vert = 100
    edges = 200
    G = nx.generators.gnm_random_graph(vert, edges)
    print(G)
    adj_matr = nx.adjacency_matrix(G, weight=None)
    with np.printoptions(edgeitems=100):
        print("Adjacency matrix: \n", adj_matr.todense())
    print("Adjacency list:")
    for line in nx.generate_adjlist(G):
        with np.printoptions(edgeitems=100):
            print(line)
    print()
    print("Depth First Search:")
    A = list(nx.connected_components(G))
    for i in range(len(A)):
        print("Component #", i+1)
        iterator = iter(A[i])
        item0 = next(iterator, None)
        print(list(nx.dfs_preorder_nodes(G, source=item0)))
        print()
    source = random.randint(0, vert-1)
    destin = random.randint(0, vert - 1)
    print(list(nx.bfs_edges(G, source=source)))
    print()
    steps = nx.shortest_path_length(G=G, source=source, target=destin)
    print("The distance between nodes #", source, "and #", destin, "=",
steps)
    path_sh = list(nx.shortest_path(G=G, source=source, target=destin))
    print(path_sh)
    plt.figure()
    ax = plt.gca()
    ax.set_title("Randomly generated graph")
    nx.draw(G, pos=nx.spring_layout(G), node_color='lightgreen', ax=ax,
with_labels=True)
    plt.show()


if __name__ == "__main__":
    main()
```