Title page:

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
Abizer Safdari
J4134c
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

- A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, *A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes*.
- Graphs are used to solve many real-life problems. Graphs are used to represent networks.
- Depth-first search 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. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.
- Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree . The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
- An **adjacency matrix** is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected the adjacency matrix is symmetric.
- an **adjacency list** is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbours of a vertex in the graph. This is one of several commonly used representations of graphs for use in computer programs.

Results

```
[[  0.   1.   2. ...  98.  99. 100.]
 [  1.   0.   0. ...   0.   0.   0.]
 [  2.   0.   0. ...   1.   1.   1.]
 ...
 [ 98.   0.   1. ...   0.   0.   0.]
 [ 99.   0.   1. ...   0.   0.   0.]
 [100.   0.   1. ...   0.   0.   0.]]
```

Figure 1: Randomly generated Adjacency matrix of a simple undirected and unweighted graph with 100 vertices and 200 edges

- The first row and column of the matrix indicates the vertix number
- there are 200 edges in the graph, the adjancency matrix contains 400 ones and remaining zeros

```
1.array  3,  4,  7,  8,  11,  12,  14,  17,  20,  25,  26,  32,  33,  40,  41,  45,  46,  47,
49,  51,  54,  58,  59,  63,  64,  68,  69,  72,  75,  76,  77,  80,  82,  85,  89,  93,
2.array  5,  6,  7,  9,  12,  14,  15,  16,  17,  18,  20,  23,  25,  26,  27,  29,  38,  41,
44,  46,  50,  52,  53,  56,  57,  58,  62,  65,  67,  70,  73,  74,  76,  78,  80,  81,  83,
85,  86,  87,  88,  89,  90,  92,  93,  94,  97,  98,  99,  100,
3.array  1,  4,  6,  11,  16,  22,  23,  25,  27,  28,  30,  31,  32,  33,  34,  37,  38,  41,
42,  46,  48,  50,  51,  54,  56,  57,  60,  61,  63,  65,  66,  67,  69,  75,  76,  77,  78,
79,  80,  81,  82,  83,  90,  92,  100,
4.array  1,  3,  5,  14,  22,  23,  27,  29,  30,  42,  47,  49,  56,  57,  58,  60,  62,  63,
65,  67,  72,  74,  76,  78,  79,  80,  82,  86,  88,  92,  93,  98,  99,  100,
5.array  2,  4,  12,  13,  16,  17,  21,  27,  31,  34,  38,  39,  40,  42,  44,  46,  47,  48,
52,  55,  59,  60,  61,  62,  66,  68,  70,  71,  73,  74,  75,  77,  78,  80,  82,  83,  84,
85,  91,  96,
6.array  2,  3,
7.array  1,  2,
8.array  1,
9.array  2,
11.array  1,  3,
12.array  1,  2,  5,
13.array  5,
14.array  1,  2,  4,
15.array  2,
16.array  2,  3,  5,
17.array  1,  2,  5,
18.array  2,
20.array  1,  2,
21.array  5,
22.array  3,  4,
23.array  2,  3,  4,
```

25.array 1, 2, 3,
26.array 1, 2,
27.array 2, 3, 4, 5,
28.array 3,
29.array 2, 4,
30.array 3, 4,
31.array 3, 5,
32.array 1, 3,
33.array 1, 3,
34.array 3, 5,
37.array 3,
38.array 2, 3, 5,
39.array 5,
40.array 1, 5,
41.array 1, 2, 3,
42.array 3, 4, 5,
44.array 2, 5,
45.array 1,
46.array 1, 2, 3, 5,
47.array 1, 4, 5,
48.array 3, 5,
49.array 1, 4,
50.array 2, 3,
51.array 1, 3,
52.array 2, 5,
53.array 2,
54.array 1, 3,
55.array 5,
56.array 2, 3, 4,
57.array 2, 3, 4,
58.array 1, 2, 4,
59.array 1, 5,
60.array 3, 4, 5,
61.array 3, 5,
62.array 2, 4, 5,
63.array 1, 3, 4,
64.array 1,
65.array 2, 3, 4,
66.array 3, 5,
67.array 2, 3, 4,
68.array 1, 5,
69.array 1, 3,
70.array 2, 5,
71.array 5,
72.array 1, 4,

```
73.array  2,  5,
74.array  2,  4,  5,
75.array  1,  3,  5,
76.array  1,  2,  3,  4,
77.array  1,  3,  5,
78.array  2,  3,  4,  5,
79.array  3,  4,
80.array  1,  2,  3,  4,  5,
81.array  2,  3,
82.array  1,  3,  4,  5,
83.array  2,  3,  5,
84.array  5,
85.array  1,  2,  5,
86.array  2,  4,
87.array  2,
88.array  2,  4,
89.array  1,  2,
90.array  2,  3,
91.array  5,
92.array  2,  3,  4,
93.array  1,  2,  4,
94.array  2,
96.array  5,
97.array  2,
98.array  2,  4,
99.array  2,  4,
100.array  2,  3,  4,
```

Figure 2: Randomlt generated Adjacency list of a simple undirected and unweighted graph with 100 vertices and 200 edges.

- The above figure represents the adjacency list of the above described adjacency matrix. In the list first element represents the vertix and the list attached to the following vertix represents the vertices connected to it through its edges.
- Generally the space complexity of adjacency matrix is O(V^2) , and the space complixity of adjancency list is O(V +E) , where V ,E represents the number of vertices and edges repectively.

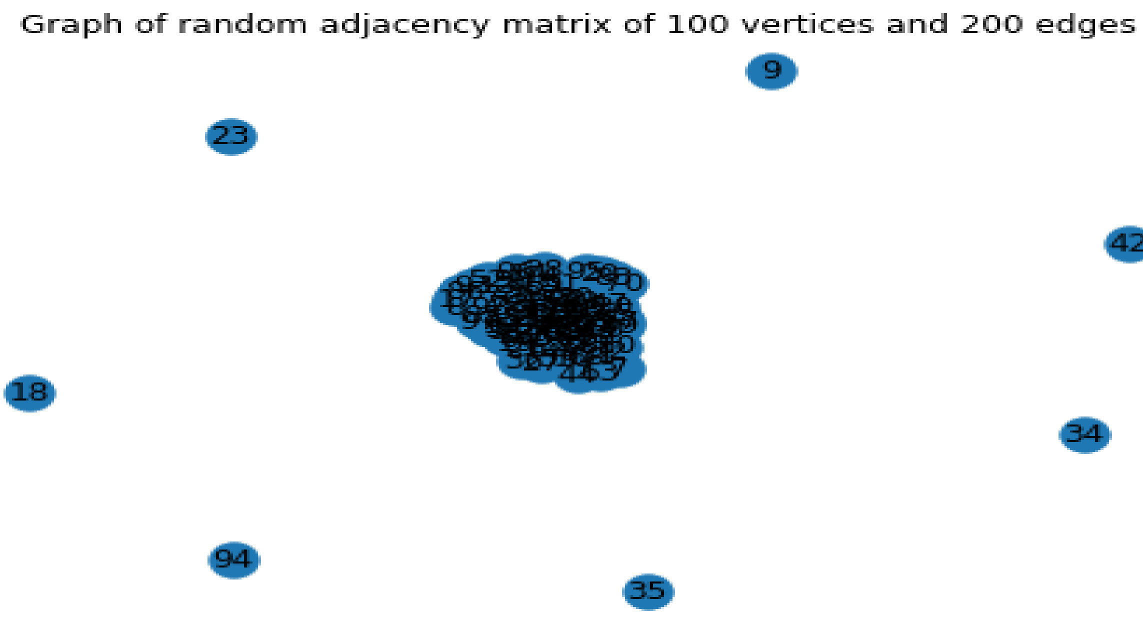
Graph of random adjacency matrix of 100 vertices and 200 edges
9
23
42
18
34
94
35

Figure 3: Randomlt generated graph of a simple undirected and unweighted graph with 100 vertices and 200 edges.

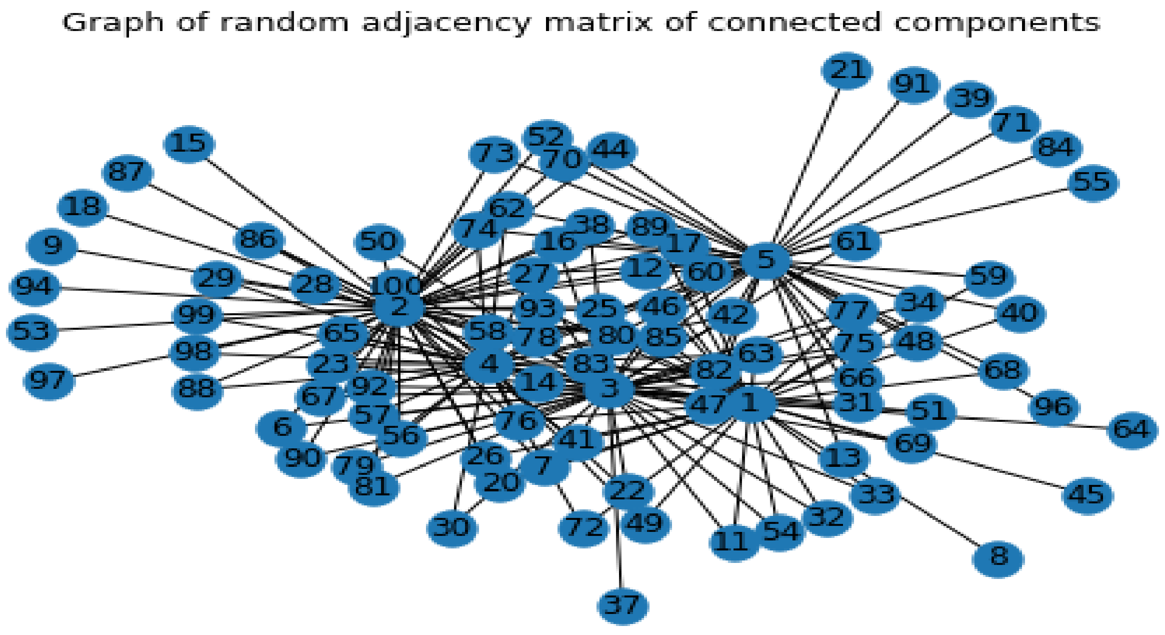

Figure 4: One of the Connected componenet of the graph

No. of connected components:8
Connected components:
[0, 2, 3, 4, 1, 5, 6, 8, 11, 13, 14, 15, 16, 17, 19, 22, 24, 25, 26, 28, 37, 40, 43, 45, 49, 51, 52, 55, 56, 57, 61, 64, 66, 69, 72, 73, 75, 77, 79, 80, 82, 84, 85, 86, 87, 88, 89, 91, 92, 93, 96, 97, 98, 99, 12, 20, 30, 33, 38, 39, 41, 46, 47, 54, 58, 59, 60, 65, 67, 70, 74, 76, 81, 83, 90, 95, 21, 29, 48, 62, 71, 78, 10, 27, 31, 32, 36, 50, 53, 68, 7, 44, 63]
[9]
[18]
[23]

[34]
[35]
[42]
[94]

Conclusions

- Depth First Search is better used when you need to search the entire tree. It's easier to implement than BFS, and requires less state.
- While BFS requires you store the entire 'frontier', DFS only requires you store the list of parent nodes of the current element.
- BFS is used to find the shortest path between any two nodes in an unweighted graph. Whereas, we cannot use DFS for the same.
- If you know a solution is not far from the root of the tree, a breadth first search (BFS) might be better.
- If the tree is very deep and solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be faster.
- If the tree is very wide, a BFS might need too much memory, so it might be completely impractical.

Appendix

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct  8 10:45:04 2020

@author: abizer
"""

import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from collections import defaultdict
# genarating a random adjacency matrix for a undirected and unweighted graph
def a_matrix(x):
    ''' generates a adjacancy matrix of size x'''
    g=np.zeros([x+1, x+1])
    g[:, 0]= np.arange(x+1)
    g[0, :]=np.arange(x+1)
    for i in range(1,x+1):
        for j in range(1,x+1):
            if i !=j:
                g[i, j]=g[j, i]=np.random.randint(0,2)
            if sum(sum(g[1:, 1:]))>=400:
                break
        if sum(sum(g[1:,1:]))>=400:
            break
    return g
adj_mat=a_matrix(100)
print(adj_mat)

#generate adjacency list
def convert(a):
    adjList = defaultdict(list)
    for i in range(1,len(a)):
        for j in range(1,len(a[i])):
                       if a[i][j]== 1:
                           adjList[i].append(j)
    return adjList
adj_list = convert(adj_mat)
for i in adj_list:
    print(i, end =".array")
    for j in adj_list[i]:
        print("  {},".format(j), end ="")
    print()
```

```
G = nx.from_numpy_matrix(np.array(adj_mat[1:, 1:]))
nx.draw(G, with_labels=True)
plt.title('Graph of random adjacency matrix of 100 vertices and 200 edges')
plt.show()

# generating graph from the adjacancy matrix
def graph(m):
    G=nx.Graph()
    for i in range(1,len(m)):
        for j in range(1, len(m)):
            if adj_mat[i][j]==1:
                G.add_edge(i, j)
                G.add_edge(j,i)
    return G

# visulizing the graph
G=graph(adj_mat)
nx.draw(G, with_labels=True)
plt.title('Graph of random adjacency matrix of connected components')
plt.show()

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
        visited[v] = True
        print(v, end = ' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)

    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self, v):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph)+1)

        # Call the recursive helper function
        # to print DFS traversal
        self.DFSUtil(v, visited)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from
            # queue and print it
            s = queue.pop(0)
            print (s, end = " ")

            # Get all adjacent vertices of the
            # dequeued vertex s. If a adjacent
            # has not been visited, then mark it
            # visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
```

```
g = Graph()
for i in range(1,101):
        for j in range(1, 101):
            if adj_mat[i][j]==1:
                g.addEdge(i, j)
                g.addEdge(j,i)
g.DFS(1)
g.BFS(1)
```