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

*Mariia Koroleva*

*J42322c*

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**

An ***(undirected) graph*** is a pair G = (V , E), where V is a set whose elements are called vertices (or nodes), and E is a set of two-sets (sets with two distinct elements) of vertices, whose elements are called edges (or links). A ***simple graph*** allows only one edge between a pair of vertices. A ***weighted graph*** is a graph in which a weight is assigned to each edge. Here, an ***unweighted graph*** is considered.

The ***adjacency matrix*** is a matrix whose rows and columns are indexed by vertices and whose cells contain a Boolean value that indicates whether the corresponding vertices are adjacent (for weighted graphs, it contains corresponding weights instead of 1s.). The matrix (stored as a 2D array) requires O(|V| 2 ) of space.

The ***adjacency list*** is a collection of lists containing the set of adjacent vertices of a vertex. The list (stored as an 1D array of lists) requires O(|V| + |E|) of space.

***Depth-first search (DFS)*** is an algorithm for traversing or searching a graph. The algorithm starts at a chosen root vertex and explores as far as possible along each branch before backtracking.

***Breadth-first search (BFS)*** is an algorithm for traversing or searching a graph. The algorithm starts at a chosen root vertex and explores all of the neighbour vertices at the present depth prior to moving on to the vertices at the next depth level.

**Results**

Graph visualization:

Изображение выглядит как небо, внешний, линия, день

Автоматически созданное описание

Adjecency matrix(fragment):

[[0. 0. 0. 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. 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.]]

Adjecency list (fragment):

[[64, 73, 77, 84, 96], [23, 76, 82, 83, 88], [7, 79], [10, 15, 27, 73, 97], [32, 38, 64, 87], [27, 88], [22, 44], [2, 19, 82], [17, 55, 72, 77, 93], [31, 54, 78]]

Which purposes is each representation more convenient for? The adjacency matrix is very indicative for representing a multigraph, while the adjacency list is significantly more space-efficient for sparse graphs, i.e., graphs in which most pairs of vertices are not connected by edges.

Speaking of data structures used in algorithms considered in this lab, the list were mostly used. However, for a stack of visited vertices sets are more convenient to guarantee the uniqueness of all the numbers.

**Conclusions**

Thanks to this lab, we were able to:

- Study in detail the concept of a graph and its types;

- Learn about the two most convenient ways of graph representation, namely the adjacency matrix and the adjacency list;

- Implement a non-trivial algorithm for constructing an adjacency matrix using symmetry properties and learn new functions of the numpy library;

- Implement functions using a depth-first search algorithm and a breadth-first search algorithm.

**Appendix**

*Part I*

import numpy as np

**def** random\_adj\_matrix(num\_verts, num\_edges):

    '''

    Generates a random adjacency matrix for a simple undirected unweighted graph

    Parameters:

    ----------

    num\_verts - int, number of vertices

    num\_edges - int, number of edges

    Output:

    result - 2D array

    '''

    num\_sells\_triangle = int((num\_verts - 1) \* num\_verts / 2) *# counting the number of sells that will be symmetric*

    edge\_indices = np.random.choice(range(num\_sells\_triangle), num\_edges, replace=False) *# picking random connections (pairs)*

    triangle\_list = np.zeros(num\_sells\_triangle) *# initializing the list*

    triangle\_list[edge\_indices] = 1 *# marking the connections*

    result = np.zeros((num\_verts, num\_verts)) *# initializing the result matrix*

    triangle\_indices = np.triu\_indices(num\_verts, k=1) *# all the indices of the upper triangle (the diagonal is not included)*

    result[triangle\_indices] = triangle\_list *# pasting the random ones into the result matrix*

    result = result + np.rot90(np.fliplr(result)) *# pasting symmetric ones*

    return result

**def** adj\_matrix\_to\_list(matrix):

  '''

  Transfer adjecency matrix into adjecency list

  Paramers:

  matrix - 2D array

  Output:

  result - 1D array

  '''

  result = [set(j for j, cell in enumerate(row) if cell == 1) for row in matrix]

  return result

m = random\_adj\_matrix(100, 200)

adj\_list  = adj\_matrix\_to\_list(m)

import networkx as nx

import matplotlib.pyplot as plt

**def** edges\_for\_graph(matrix):

  '''

  Getting the edges in format suitable for networkx library

  '''

  result = [[(i, j) for j in range(len(matrix)) if matrix[i][j] == 1] for i in range(matrix.shape[0])]

  return result

edges = edges\_for\_graph(m)

*# Visualizing the graph*

G=nx.Graph()

G.add\_nodes\_from(list(range(100)))

for ed in edges:

  G.add\_edges\_from(ed)

options = {

    'node\_size': 200,

    'width': 0.5,

    'with\_labels': True,

    'font\_weight': 'bold'

}

nx.draw\_random(G, \*\*options)

*#nx.draw(G, with\_labels=True, font\_weight='bold')*

*# Printing several rows of the adjecency matrix and the adjecency list*

print('Adjecency matrix:\n')

print(m[..., :10, :10])

print('\n')

print('Adjecency list:\n')

print(adj\_list[:10])

*Part II*

**def** dfs(graph, start, visited=None):

    '''

    This function uses Depth-first search algorithm to find connected components of a given grapgh.

    Parameters:

    ----------

    graph: list of sets, graph's adjecency list

    start: int, a number of a vertice to start from

    visited: set, numbers of vertices visited before (every vertice should be visited not more than once)

    Output:

    visited - a set of numbers of visited vertices

    '''

    if visited is None:

        visited = set()

    visited.add(start)

    print(start)

    if len(graph[start] - visited) == 0:

        print('End of one connected part (subgraph)')

    for next in graph[start] - visited:

        dfs(graph, next, visited)

    return visited

dfs(adj\_list, 0)

import collections

**def** bfs(graph, start, end):

  '''

  Shows the shortest path between two vertices

  Params:

  ------

  graph: list, graph's adjecency list

  start: int, a number of a vertice to start from

  end: int, a number of an ending vertice

  '''

  if start == end:

    print('The start an the end is the same')

  visited = set()

  queue = [[start]]

  result = []

  iter = 0

  while queue:

      result = queue.pop(0)

      next = result[-1]

      if not next in visited:

         for neighbour in graph[next]:

            new\_result = list(result)

            new\_result.append(neighbour)

            queue.append(new\_result)

            if neighbour == end:

              return new\_result

         visited.add(next)

      iter += 1

      if iter > 10000:

        return new\_result

  print('There is no path between {} and {}'.format(start, end))

bfs(adj\_list, np.random.randint(0, 100), np.random.randint(0, 100))