НОВ БЪЛГАРСКИ УНИВЕРСИТЕТ

ДОМАШНА РАБОТА №2

CSCB039 Алгоритми и програмиране

пролет 2024/2025

Разработил:

Теодор Мангъров

факултетен номер: F113621

Ръководител:

доц. д-р Ласко Ласков

София, 2025г.

Problem

Let $G = (V, E)$ be a directed graph. Its square graph $G^2 = (V, E^2)$ is such that an edge $(u, v) \in E^2$ if and only if in G there is a (u, v)-path that contains at most two edges. In other words, either there must be an edge (u, v), or there must be a vertex $w \in V$, such that $(u, w), (w, v) \in E$. Design and implement an efficient algorithm that calculates G^2 if G is represented with adjacency lists.

Preparation

To accomplish the goals of the task, at first we need to implement a basic structure that will represent the graph G, from which we are going to derive G^2. Our structure is going to be simple and its functionalities will be described in several procedures, which are going to be members of the same graph object.

Basic structure

An overview of the header file (.h) is provided further down in the text.

```
// Graph.h
#include <unordered_map>
#include <unordered_set>
#include <vector>


struct Graph {
      using GraphType = std::unordered_map<char, std::vector<char>>;

      GraphType m_graph;

      bool addVertex(char vertex);
      bool addEdge(char v1, char v2);
      bool printGraph();

      Graph squareGraph();

};
```

Basic graph definition

The main container used to represent the graph is the so-called unordered_map.
This is a STL data structure that works with pair of key and value. It has an ordinance with respect to the keys and does not allow key duplicates. Unordered_map is a good data structure for our case, because we

need to describe the vertices, which are unique, and at the same time keep a list of neighbors for each of those vertices. (also called an adjacency list). We don't need an ordering for the vertices, but rather just avoiding duplicates. For easier reading through the code, a custom type is implemented to hold the specific signature of unordered_map in our case. That signature is described by a simple character type for each vertex, in this case working as the key, and a vector that represents the adjacency list for a specific vertex - the value. The specific type is called "GraphType" and it's just a simple nickname for the long signature of unordered_map.
The member type of the structure is named "m_graph".

Furthermore, we have few basic procedures for inserting vertices and edges, as well as printing the whole graph, thus giving us the basic functionality for our graph. However, the procedure that we are going to spend the most time on in this paper is "squareGraph()".

Understanding the problem and defining goals

Reading through the description of our problem, we are asked to create an algorithm that calculates the square graph (G^2) with the present vertices of G.
This means if we have a graph G, we need to create a new graph defined as G^2, in which exists every vertex from G, with the distance between vertices being at least 2.
So if we have vertices u and v, we must have a dividing vertex w on which can be connected two edges leading to u and v, creating a total distance between u and v of 2. If this requirement is met, u and v are added into G^2 by a single edge (u, v).

If in G there exists:

$$u \rightarrow w \rightarrow v$$

… in G^2 it becomes:

$$u \rightarrow v$$

This is achieved by the squareGraph function. Before diving into the explanation of the algorithm it is necessary to have an already defined graph on which we can work through, and whose results we are going to compare to the original, thus verifying the validity of our algorithm. Describing a basic graph and the whole work on it is shown in the main entry of our program here:

```
int main() {

        Graph G;
        G.addVertex('A');
        G.addVertex('B');
        G.addVertex('C');
        G.addVertex('D');

        G.addEdge('A', 'B');
        G.addEdge('B', 'C');
        G.addEdge('C', 'D');
```

```
        G.printGraph();


        std::cout << "\nG^2\n";
        Graph g2 = G.squareGraph();
        g2.printGraph();


        return 0;
}
```

As shown by the code in the main entry, first we define the vertices and then we connect them with edges. The graph gets printed to the console followed by its squared version. This is done to distinguish the differences between the original graph G and the modified squared graph G^2.

The squareGraph procedure

Now, let's finally take a look at the squareGraph() procedure. We start with the function definition and we will work through the rest of it progressively.

```
Graph squareGraph() {
        Graph squareGraphObj;
        for (const auto& [u, vertexneighborsSet] : m_graph) {
                std::unordered_set<char> distance2;

                for (char v : vertexneighborsSet) {
                        squareGraphObj.m_graph[u].push_back(v);


                        for (char w : m_graph.at(v)) {
                                if (
            w != u && std::find(m_graph.at(u).begin(), m_graph.at(u).end(), w) == m_graph.at(u).end()
                                ) {
                                        distance2.insert(w);
                                }
                        }
                }

                for (char element : distance2) {
                        squareGraphObj.m_graph[u].push_back(element);
                }
        }
        return squareGraphObj;
}
```

Algorithm explanation

A separate object of the Graph type (different from "GraphType") gets created, which then will be filled with all vertices that are of distance of 2. Further down it is returned to the previous stack layer.

The algorithm iterates through each vertex u in G and for every u there exists v, representing each of the neighbors of u. Those neighbors are added to G^2.

Furthermore all vertices of distance 2 are inserted into an unordered set if they are not already present in G, after which G^2 gets modified again and then returned.

Time complexity

We can call V the set of all vertices in a graph,
E the set of all edges in a graph
and $deg(u)$ be the number of neighbors (degree) for each $u \in V$.

We can define a simplified version of the algorithm by the following pseudocode:

```
for each u:
    for each v in neighbors(u):
        for each w in neighbors(v):
            check if w is in neighbors(u)
```

Given the fact that the algorithm has nested for loops, whose count reaches 3, by excluding the runtime of all operations like find() and other container based algorithms, we can get an estimate time complexity of $O(V^3)$, where for each vertex depends the number of present neighbors $deg(u)$.

Bibliography

1. Kenneth Rossen “Discrete Mathematics and Its Applications”, Eighth edition. | New York, NY : McGraw-Hill, [2019], pp. 703–737
2. “Graph Power.” Wikipedia, 18 July 2024. Wikipedia, https://en.wikipedia.org/w/index.php?title=Graph_power&oldid=1235225666
3. https://math.stackexchange.com/questions/15575/square-of-a-graph
4. Milanič, Martin, et al. “A Characterization of Line Graphs That Are Squares of Graphs.” Discrete Applied Mathematics, vol. 173, Aug. 2014, pp. 83–91. ScienceDirect, https://doi.org/10.1016/j.dam.2014.03.021
5. https://www.researchgate.net/figure/A-graph-G-and-its-square-G-2_fig1_220441604
6. Singh, Rishabh. “Learn Graph Data Structure (C++).” Medium, 24 Feb. 2025, https://medium.com/@RobuRishabh/learn-graph-data-structure-c-c505e6159f6a
7. Weisstein, Eric W. Graph Power. https://mathworld.wolfram.com/GraphPower.html. Accessed 6 June 2025.