FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION
OF HIGHER EDUCATION
ITMO UNIVERSITY

Report

on the practical task No. 8

Practical analysis of advanced algorithms

Performed by

Dmitry Grigorev, Maximilian Golovach

J4133c

Accepted by

Dr Petr Chunaev

St. Petersburg

2022

Goal of the work

The goal of this work is to consider two algorithms:

- Kruskal's algorithm for Minimal Spanning Tree construction,
- Improved All-Pairs Shortest Paths algorithm

in terms of their theoretical foundations, theoretical properties, i.e. time and space complexity, and obtain an estimate on time complexity empirically. Furthermore, we have to implement this algorithms to become familiar with the complexity of their implementation in code.

The work is organized as follows: the chapter 1 corresponds to the analysis of Kruskal's algorithm. The chapter 2 is devoted to Improved All-Pairs Shortest Paths algorithm.

Chapter 1

Kruskal's algorithm

1.1. Formulation of the problem

Suppose we are given a connected weighted undirected graph $G(V, E, W)$. The task is to find in the graph Minimum Spanning Tree (MST) by means of Kruskal's algorithm as well as:

- become familiar with the algorithm,
- examine its time complexity and compare it with theoretical result,
- provide the information on techniques and data structures used in the implementation of the algorithm.

1.2. Brief theoretical part

Kruskal's's algorithm is an algorithm that finds Minimal Spanning Tree.

A Spanning tree is a subset to a connected graph G, where all the edges are connected, i.e, one can traverse to any edge from a particular edge with or without intermediates. Also, a spanning tree must not have any cycle in it by definition. Thus we can say that if there are $|V| = N$ vertices in a connected graph then the number of edges that a spanning tree may have is $N - 1$.

As for a minimum spanning tree (MST) (or minimum weight spanning tree) for a connected weighted undirected graph, it is a Spanning tree that has a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

The algorithm description is as follows:

1. Sort all given graph's edges by weight ascending.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat the step 2 until there are $N - 1$ edges in your spanning tree.

1.2.1. Some MST application examples

The standard application is to tackle a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect the offices with each other, and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It directly resulted in that the solution for this problem should be presented as a spanning tree, since if a network isn't a tree you can always remove some edges and save money. A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

1.2.2. Complexity analysis

The time complexity of the algorithm is $O(|E| \log |E|)$ or $O(|V| \log |E|)$. Sorting of edges by means of Tim Sort takes $O(|E| \log |E|)$ time. After sorting, we iterate through all edges and apply the find-union algorithm to check whether current edges reduce the number of connected components. The find and union operations can take at most $O(\log |V|)$ time. The value of $|E|$ can be at most $O(|V|^2)$, so $O(\log |V|)$ is $O(\log |E|)$ the same. Therefore, the overall time complexity is $O(|E| \log |E|)$ or $O(|V| \log |E|)$.

As for the space complexity, if one uses Tim Sort, it requires $O(|E|)$. Furthermore, the resulted spanning tree requires $O(|E|)$ memory. That is why the space complexity of Kruskal's algorithm is $O(|E|)$.

1.3. Results

At first we tested our implementation and the algorithm itself on whether they work properly. Two graphs we tested the algorithm on are presented in the figures 1.1 and 1.3. The resulted MSTs of these graphs are provided in the figures 1.2 and 1.4 correspondingly. One can visually check the MSTs' optimality according to the graphs.

As soon as we verified our implementation, we conducted the experiment on measuring of time complexity estimate. The test was performed on graphs with varying number of edges $|E|$. Here $|E| = 1, \ldots, 1250$. For each $|E|$ 5 measurements were executed to calculate mean time complexity in order to reduce the influence of outliers on the results. The results are presented in the figure 1.5 where one can see the fitted curve which surely corresponds to the theoretical time complexity of $O(|E| \log |E|)$.

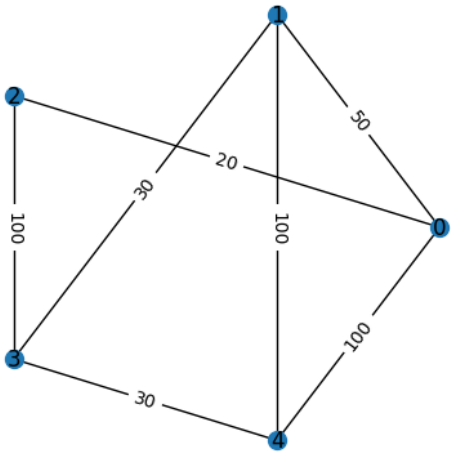

Figure 1.1: Arbitrary graph with 5 nodes

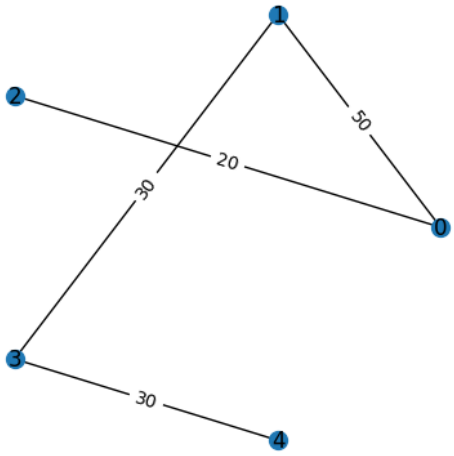

Figure 1.2: The resulted MST for the graph

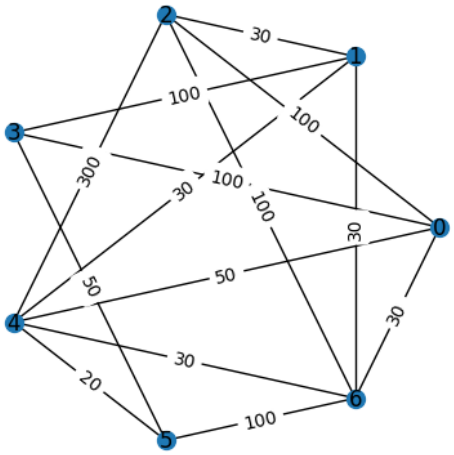

Figure 1.3: Arbitrary graph with 7 nodes

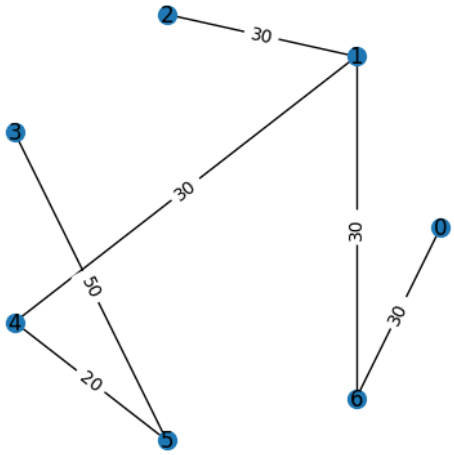

Figure 1.4: The resulted MST for the graph

If one wants to predict the expected time of execution for the graphs of larger number of edges, according to the fitted line they may expect for $n = |E| = 5000$ the execution time of ≈ 0.43 milliseconds, for $n = 10000$ — ≈ 1 second and for $n = 100000$ — ≈ 11.51 seconds what is quite fast even for large graphs.

1.4. Data structures and design techniques used in algorithms

In the random variables generation and building of adjacency matrix the Python's package **Numpy** is used due to its convenience. As a tool of drawing graphs the **networkx** package is used. To store and process graph here we used the approach of adjacency lists since multiple edges are allowed. The implementation of Tim Sort is provided by standard Python

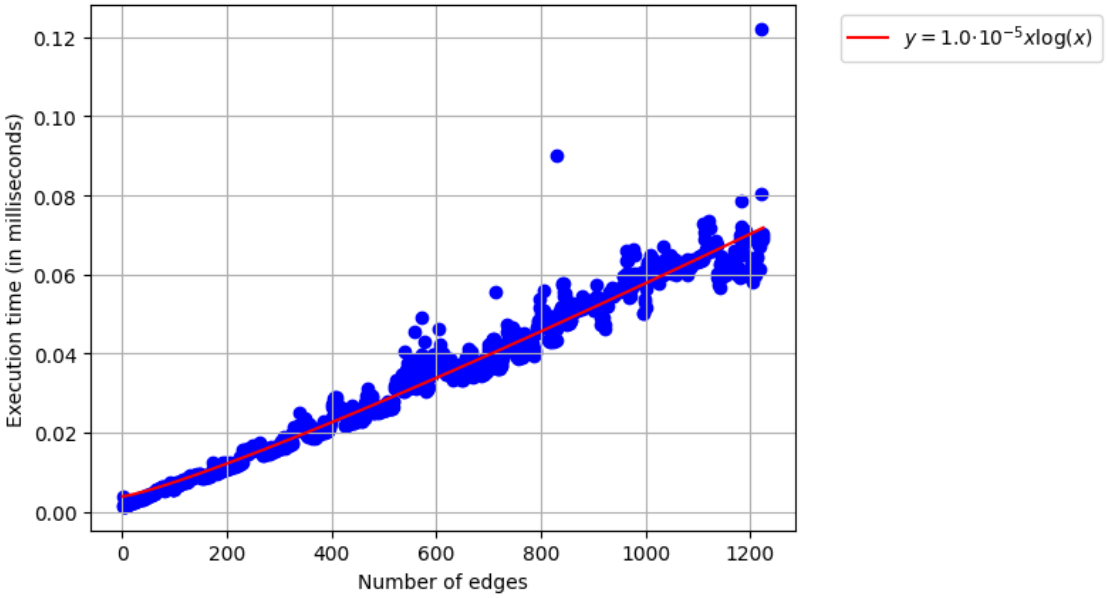

Figure 1.5: Results of time complexity measurements on graphs with $n = |E|$ edges. The points are the mean execution times across 5 trials for each $n = 1, \ldots, 1250$. The line is fitted to the points.

function **sorted**. The implementation of Kruskal's algorithm also includes the approach **Union-Find** [1] to detect cycles.

Chapter 2

Improved All-Pairs Shortest Paths algorithm

2.1. Formulation of the problem

Suppose we are given a simple weighted directed graph $G(V, E, W)$. W may contain negative weights but negative cycles are not allowed. The task is to apply Improved All-Pairs Shortest Paths algorithm to find all shortest paths for each pair of nodes in the graph and also:

- become familiar with the algorithm,
- examine its time complexity and compare it with theoretical result,
- provide the information on techniques and data structures used in the implementation of the algorithm.

2.2. Brief theoretical part

The theory is taken from [2].

Let $G(V, E, W)$ be a simple weighted directed graph, W may contain negative components. The task is to find all shortest (in terms of W) paths between each pair of nodes. Moreover, it has zeros on main diagonal and ∞ in the components which correspond to the edges not presented in the graph. Let $n = |V|$.

Let $\{L^{(k)} = (l_{ij}^k)\}$ be the sequence of matrices defined the minimum weight of each path where m indicates that any path has at most m edges. For $m = 0$ the matrix $L^{(0)}$ looks as follows:

$$l_{ij}^{(0)} = \begin{cases} 0, \text{ if } i = j, \\ \infty, \text{ if } i \neq j. \end{cases}$$

For $m \geq 1$ $l_{ij}^{(m)}$ are defined recursively:

$$l_{ij}^{(m)} = \min \left\{ l_{ij}^{(m-1)}, \min_{k=1,\ldots,n} \{ l_{ik}^{(m-1)} + w_{kj} \} \right\} \underbrace{= \min_{k=1,\ldots,n} \{ l_{ik}^{(m-1)} + w_{kj} \}}_{\text{since } w_{jj}=0 \ \forall j}.$$

By the definition any shortest path has at most $n - 1$ edges so

$$\forall\ i, j\ l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \ldots$$

and it is sufficient to calculate n matrices consequentially. Moreover, the task of calculation $L(m)$ is similar to calculation of special matrix multiplication.

The ordinary matrix multiplication $C = A \cdot B$ is defined by expression $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. The special one is resulted by substitutions of operations:

- $\min \to +$,
- $+ \to \cdot$

So due to the common nature with the ordinary matrix multiplication, this one also has $O(n^3)$ time complexity.

Algorithm 1 Special matrix multiplication(L, W)

```
n ← |V|
L' = (l'_ij) — result matrix n × n
for i = 1, ..., n do
    for j = 1, ..., n do
        l'_ij ← ∞
        for k = 1, ..., n do
            l'_ij ← min{l'_ij, l_ik + w_kj}
        end for
    end for
end for
```

Ensure: L'

So the All-Pairs Shortest Paths algorithm requires $O(n)$ special matrix multiplications and, thus, has $O(n^4)$ time complexity.

The idea of improvement is quite evident: instead of calculation of all $L^{(1)}, \ldots, L^{(n-1)}$ we can calculate only $L^{(1)}, L^{(2)}, L^{(4)}, L^{(8)}, \ldots, L^{(2^{\lceil \log_2 (n-1) \rceil})}$ and the last matrix $L^{(2^{\lceil \log_2 (n-1) \rceil})} = L^{(n-1)}$ since shortest paths in the graph with n vertices have at most $n-1$ edges in their structure.

The resulted algorithm, called Improved All-Pairs Shortest Paths, has time complexity of $O(n^3 \log n)$. The description of the algorithms is provided in 2.

The algorithm space complexity is $O(n^2)$ since we have to store $n \times n$ matrix of distances at each step.

Algorithm 2 Improved All-Pairs Shortest Paths

Require: W — adjacency matrix of simple weighted directed graph G

$n \leftarrow |V|$

$L^{(0)} \leftarrow W$

$m \leftarrow 1$

while $m < n - 1$ **do**

$L^{(2m)} \leftarrow$ Special Matrix Multiplication$(L^{(m)}, L^{(m)})$

$m \leftarrow 2m$

end while

Ensure: $L^{(m)} = L^{(n-1)}$ — matrix of all shortest paths

2.3. Results

As soon as algorithm was implemented, as first we conducted some tests on whether the algorithm works properly. Two of all tested graphs are presented in the figures 2.1 and 2.3. The first is a graph which is 5-complete if we ignore directions on edges, another is an arbitrary graph with 7 nodes. The results of the algorithm application to these graphs are depicted in the figures 2.2 and 2.4 correspondingly. One can validate the results according to the graphs.

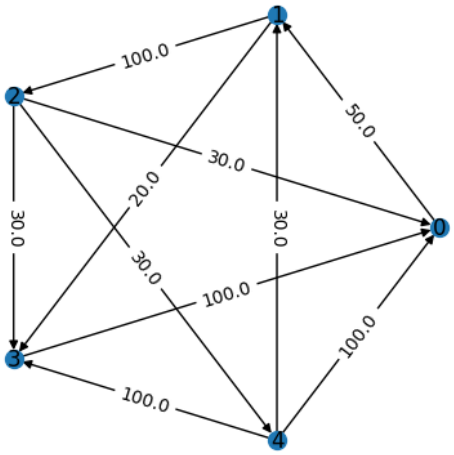

Figure 2.1: Directed 5-"complete" graph

```
In 6  1  improved_all_pairs_shortest_paths(Madj_with_inf.tolist())

Out 6    [[0, 50.0, 150.0, 70.0, 180.0],
          [120.0, 0, 100.0, 20.0, 130.0],
          [30.0, 60.0, 0, 30.0, 30.0],
          [100.0, 150.0, 250.0, 0, 280.0],
          [100.0, 30.0, 130.0, 50.0, 0]]
```

Figure 2.2: The resulted matrix of shortest paths for the graph

Further, we conducted the experiment on measuring of time complexity estimate. The test was performed on graphs which are n-complete if we omit directions on edges. Here $n = 1, \ldots, 150$. For each n 5 measurements were executed to calculate mean time complexity in order to reduce the influence of outliers on the experiment. The results are presented in

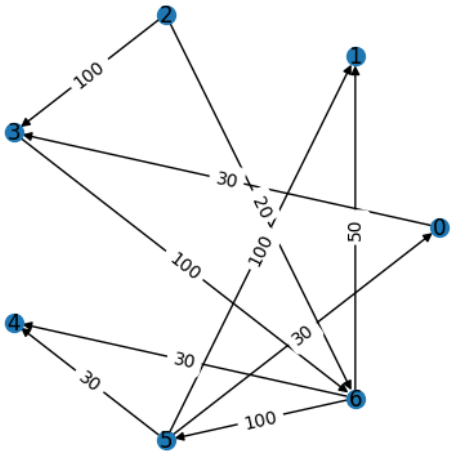

Figure 2.3: Directed graph with 7 nodes

```
In 25   1  improved_all_pairs_shortest_paths(Madj2_with_inf.tolist())

Out 25     [[0, 180.0, inf, 30.0, 160.0, 230.0, 130.0],
           [inf, 0, inf, inf, inf, inf, inf],
           [150.0, 70.0, 0, 100.0, 50.0, 120.0, 20.0],
           [230.0, 150.0, inf, 0, 130.0, 200.0, 100.0],
           [inf, inf, inf, inf, 0, inf, inf],
           [30.0, 100.0, inf, 60.0, 30.0, 0, 160.0],
           [130.0, 50.0, inf, 160.0, 30.0, 100.0, 0]]
```

Figure 2.4: The resulted matrix of shortest paths for the graph

the figure 2.5 where one can see the fitted curve which surely reflects the theoretical time complexity of $O(n^3 \log n)$.

For example, according to the fitted line we may expect for $n = 500$ the execution time of ≈ 2 minutes, for $n = 1000$ — ≈ 20 minutes and for $n = 10000$ — ≈ 18 days. This demonstrates why the direct application of even improved algorithm may be irrational. For sure one can optimize the special matrix multiplication in the way that ordinary multiplication may be optimized to the time complexity of $O(n^{2.37})$ (see [3]).

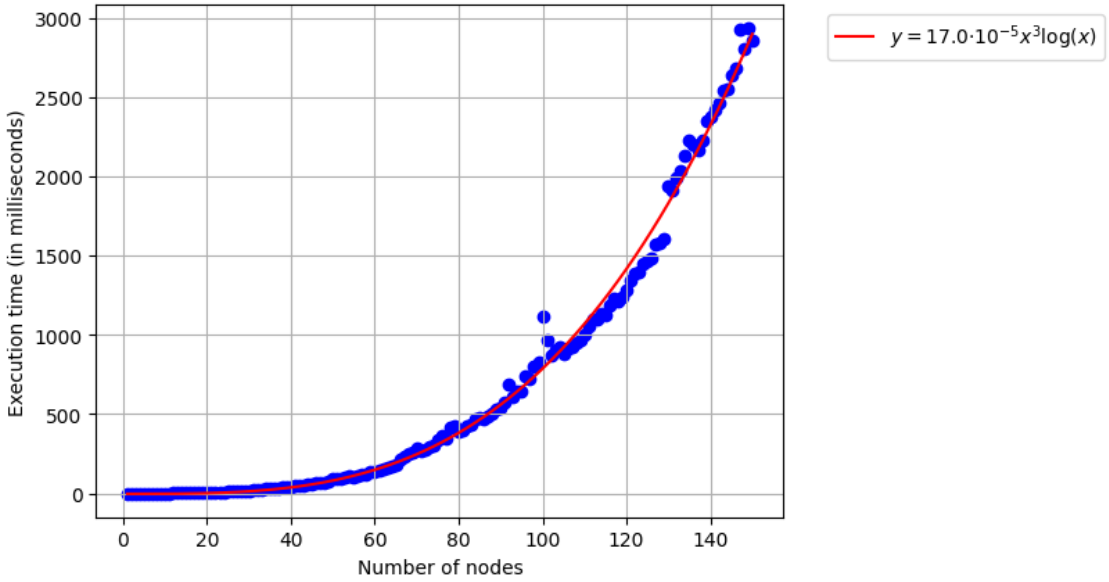

Figure 2.5: Results of time complexity measurements on complete graphs. The points are the mean execution times across 5 trials for each $n = 1, \ldots, 150$. The line is fitted to the points.

2.4. Data structures and design techniques used in algorithms

In the random variables generation and building of adjacency matrix the Python's package **Numpy** is used due to its convenience. As a tool of drawing graphs the **networkx** package is used. We used Python lists to store and process adjacency matrices.

Conclusion

As the result of this work, we have become familiar with Kruskal's and Improved All-Pairs Shortest Paths algorithms which are applied to weighted graphs. We performed some tests to examine the correctness of the algorithms and experiments to measure the time complexity in practice. The results surely correspond to the theoretical ones. The implementations for the algorithms are provided. Data structures and design techniques which were used in the implementations were discussed. The work's goals were achieved.

Appendix

Algorithms implementation code is provided in [4].

Bibliography

1. Conchon S., Filliâtre J.-C. A Persistent Union-Find Data Structure // Proceedings of the 2007 Workshop on Workshop on ML. — New York, NY, USA : Association for Computing Machinery. — 2007. — ML '07. — P. 37–46. — Access mode: https://doi.org/10.1145/1292535.1292541.
2. Introduction to Algorithms, Third Edition / Cormen T. H., Leiserson C. E., Rivest R. L., and Stein C. — 3rd ed. — The MIT Press, 2009. — ISBN: 0262033844.
3. Alman J., Williams V. V. A Refined Laser Method and Faster Matrix Multiplication. — 2020. — Access mode: https://arxiv.org/abs/2010.05846.
4. Grigorev D., Golovach M. Code repository. — https://github.com/dmitry-grigorev/AlgoAnalysisDevelopment. — 2022.