Faculty of Mathematics and Information Science

WARSAW UNIVERSITY OF TECHNOLOGY

Algorithms and computability

Final documentation

Aleksandra Kołakowska, Andrzej Litkowski

09.12.2021

Table of Contents

1. History of changes

1.1 Documentation

Date	Author	Description
25.10.2021	Andrzej Litkowski	Initial pseudocode with notes
28.10.2021	Aleksandra Kołakowska	Introduction, Description of exact and approximate solution, Bibliography
04.11.2021	Andrzej Litkowski	Pseudocode update
04.11.2021	Aleksandra Kołakowska	Description and introduction update
04.11.2021	Andrzej Litkowski	Pseudocode update
25.11.2021	Andrzej Litkowski	Initial code structure and output description, tests
25.11.2021	Aleksandra Kołakowska	Updated code structure and output descriptions, added building process and launching for the executable
25.11.2021	Andrzej Litkowski	Updated building, layout and test descriptions
25.11.2021	Aleksandra Kołakowska	Updated final documentation
09.12.2021	Aleksandra Kołakowska	Updated finding maximal common subgraph, test and input sections
09.12.2021	Andrzej Litkowski	Updated tests and input sections

Table 1. History of changes for documentation

1.2 Program

Date	Author	Description
04.11.2021	Andrzej Litkowski	Initial code - main algorithm
04.11.2021	Aleksandra Kołakowska	Updated main and added input, output and samples
19.11.2021	Aleksandra Kołakowska	Added error handling
25.11.2021	Andrzej Litkowski	Added tests
25.11.2021	Aleksandra Kołakowska	Updated main and input, generated python executable
26.11.2021	Andrzej Litkowski	Version in C# created (due to the fact that python executable was detected as malware on the lab computer)
26.11.2021	Aleksandra Kołakowska	Renamed example input files and added more input examples
09.12.2021	Aleksandra Kołakowska	Updated output, error handling and added example input

Table 2. History of changes for source files, executable and input

2. Introduction

The goal of this project is to find the exact and approximate solutions to two problems. For given two graphs, defined in the form of adjacency matrix, we have to find the maximal common subgraph and the minimal common supergraph.

First, let us define the subgraph G' of a graph G as a graph G' whose vertex set and edge set are subsets of those of G. If G' is a subgraph of $G,$ then G is said to be a supergraph[5] of G'. We can also recall that an edge-induced subgraph (later simply induced subgraph) is a subset of the edges of a graph G together with any vertices that are their endpoints.

Now, let us define the common induced subgraph of two graphs G and H as a graph that is an induced subgraph of both of these graphs. Thus, the maximum common induced subgraph of two graphs G and H is a common induced subgraph of these two graphs that has as many edges as possible.

Finally, we define the common supergraph of two graphs G and H as the graph that includes graphs G and H as subgraphs. Therefore, the minimum common supergraph of two graphs G and H is a common supergraph of these two graphs that is the smallest. Furthermore, we can use the fact that minimum common supergraph computation can be solved using the computation of maximum common subgraph computation[2].

3. Exact Solution for the minimal common supergraph

3.1 Main idea

The main idea behind the exact solution was to find all possible mappings of vertices from the larger graph to the smaller one. We can achieve that by finding all permutations of vertices of the larger graph and then treating several of these found ones as vertices of the smaller graph, with an assumption that order of the vertices of the smaller graph does not change.

The next step, done for each mapping, would be appending the edges of the smaller graph to the mapped matrix. As a result of these operations, we get a matrix that combines a permuted larger graph and smaller graph. Each element of the matrix can be one out of three possible values, ‘0’ (no edge), ‘1’ (edge corresponding to this element of the matrix is only in one of the graphs) or ‘2’ (edge corresponding to this element of the matrix is in both graphs).

Out of all mapped matrices, the one that has the most elements with a value ‘2’ has the most common edges. Thus, we know that matrix corresponds to the minimal common supergraph. Additionally, since we are checking sequences that describe the mappings of vertices, we can easily save that mapping.

3.2 Complexity analysis

In this section, we are going to analyse the complexity of the algorithm we proposed. Let's define n as a number of vertices in the larger graph and m as a number of vertices in the smaller graph.

We decided to omit the complexity of reading the input from the file and saving the output in order to focus on the complexity of the main algorithm.

As the first step of the solution, we create $n!$ sequences of length n that are permutations of labels of vertices of the larger graph. This step could be potentially optimized during the actual implementation. Since vertices numbered more than m do not have corresponding vertex in the smaller graph, we could assume that their order does not matter. Thus, the number of created sequences would decrease from $n!$ to $\frac{n!}{(n-m)!}$.

Next, we have to recreate the matrix based on each of the sequences. This is a simple reading and writing operation of n^2 elements between matrices. Then, we have m^2 additions (appending the edges of the smaller graph to the generated matrix) and m^2 reading operations (counting common edges).

Finally, let's assume that we can replace m with n for Big O notation. We would have $c * n^2$ operations per sequence, with c being some constant. Since we have $n!$ sequences, the complexity is $\mathcal{O}(n^2 * n!)$.

4. Approximate Solution for minimal common supergraph

4.1 Main idea

In the exact solution, we check all $n!$ possible mappings of vertices from one graph to another. For n large enough the exact solution would not be an effective one. In order to reduce complexity, we can choose one mapping that we expect to be the best and start by checking just this one mapping.

Our idea for the choice of that mapping is quite simple: we map the vertex with the largest degree from one of the graphs to the vertex with the largest degree of the other one. Then, we map the vertex with the second largest degree from the first graph to the vertex with the second largest degree from the second graph. We repeat this process until we run out of vertices in one graph and then we just write the remaining vertices of the larger graph in the order of their degree. In case of ambiguity, such as several vertices having the same degree we use them in order of labels in the original input.

4.2 Complexity analysis

Similarly to the exact solution, there are $\mathcal{O}(n^2)$ operations per each created sequence. In this case, we have only one sequence - the one predicted to be the best one. In the approximate solution, we have to additionally consider the cost of creating that sequence. Initially, we read all entries in the matrix to find the degrees of all vertices. The order of that operation is $\mathcal{O}(n^2)$.

Then, we have to sort these vertices with one of the sorting algorithms that have the order $\mathcal{O}(n * log(n))$. Finally, we have to combine the sequences of both graphs. This can be done by traversing a longer sequence, which would require operations of the order $\mathcal{O}(n)$.

All of that combined gives us the complexity of $\mathcal{O}(n^2)$. We have to remember that there are more operations performed on the graphs, but it still results in the total complexity of the order $\mathcal{O}(n^2)$. This result is significantly better than the exact one, which was $\mathcal{O}(n^2 * n!)$.

5. Finding maximal common subgraph from minimal common supergraph

5.1 Exact solution

The main idea behind the exact solution for a minimal common supergraph was to find all possible mappings of vertices from the larger graph to the smaller one. Since the maximal common subgraph computation can be solved using the computation of minimum common supergraph computation[2], we can perform the same operations described in section 2.1.

As a result of these operations, we get a matrix that combines a permuted larger graph and smaller graph. Each element of the matrix can be one out of three values, ‘0’ (no edge), ‘1’ (edge corresponding to this element of the matrix is only in one of the graphs) or ‘2’ (edge corresponding to this element of the matrix is in both graphs).

Finally, as an output result, we get the minimal common supergraph - the matrix that has the most common edges. If the user wanted to see the maximal common subgraph, we would have to take only repeated edges. In order to achieve that, it would be enough to read only edges marked with '2' and remove all those marked with '1'.

5.2 Approximate solution

In the approximate solution, we use the computations done for the minimum common supergraph as described in Section 3.1. As an output result of the approximate solution, we get the minimal common supergraph - the matrix that has the most common edges. In case the user wanted to see the maximal common subgraph, it would be enough to read only edges marked as common and remove all those marked as only being in one of the graphs.

6. Pseudocode

In this section we aim to describe the pseudocode of the solution. Our implementation assumes that both graphs should have at least 2 vertices and at least 1 edge. We start by reading the graphs, in the form of adjacency matrices, along with their sizes from the file given as an argument. Then, we compare sizes of the graphs and create sequences based on the larger one. It is important to differentiate between smaller and bigger graphs and not only read them as first and second.

```
if APPROXIMATE:
    sequences = [ GenerateApproximateSequence() ]; //Only one sequence
wrapped in list
else:
    sequences = Calculate all permutations of numbers
1..biggerMatrixSize;

biggestMatch = 0;

loop sequence from sequences:
    rearrangedMatrix = Create matrix using bigger graph by rearranging
vertices to match the sequence;
    Superimpose smaller graph (both starting at 0, 0) on the
rearrangedMatrix by adding numbers in matrices;
    commonEdges = Calculate common edges by counting twos in combined
matrix;
    if commonEdges > biggestMatch:
        biggestMatch = commonEdges;
        Save sequence and rearrangedMatrix;

output biggestMatch, its sequence and saved matrix - it is min common
supergraph and max common subgraph can be read from there;

//Approximate logic
function GenerateApproximateSequence():
    return sequence of labels of vertices of bigger graph such that:
        i) label on index i is the j-th biggest degree vertex in bigger
graph and label i is the j-th biggest degree vertex in smaller graph
        ii) any vertices left in the bigger graph after all vertices of
the smaller one are used are put in descending degree order at the end
        iii) if there are several vertices with the same degree, smaller
label is used first;
```

7. Program

The program - along with the documentation, sources, and example - was placed inside the structure consisting of the following folders:

- Doc folder - contains this documentation file, *Documentation.pdf,*
- EXE folder - contains *AoC.exe*, the executable version of the program,
- Source folder - contains the Visual Studio solution files together with *Program.cs* containing its code
- Examples folder - contains input txt files with graphs in the form of an adjacency matrix.

7.1 Code structure

Inside the “Solution” directory, there is a Visual Studio solution that can be used to build the executable. The whole code is in the file “Program.cs”. At the beginning it contains helper functions:

- generating all permutations of n numbers[3],
- creating a permuted matrix from the sequence of vertices labels and a matrix with original graph,
- reversing permuted matrix to the original order of vertices knowing the sequence,
- calculating common edges from a matrix (edges marked with ‘2’),
- calculating degree of a vertex (by counting edges in proper matrix row),
- finding vertex with maximal degree in a graph not already present in given sequence,
- finding sequence of labels of vertices sorted by their degree,
- generating approximated sequence by matching vertices with highest degrees of both graphs,
- reading graphs from file, saving results to files and printing the results to the console(if the larger graph has at most 25 vertices)

The last section of the code contains the main function. It reads the graphs from the given file and performs the algorithm as described in the pseudocode. Additionally, time of execution of the algorithmic part is measured (without reading input and saving output).

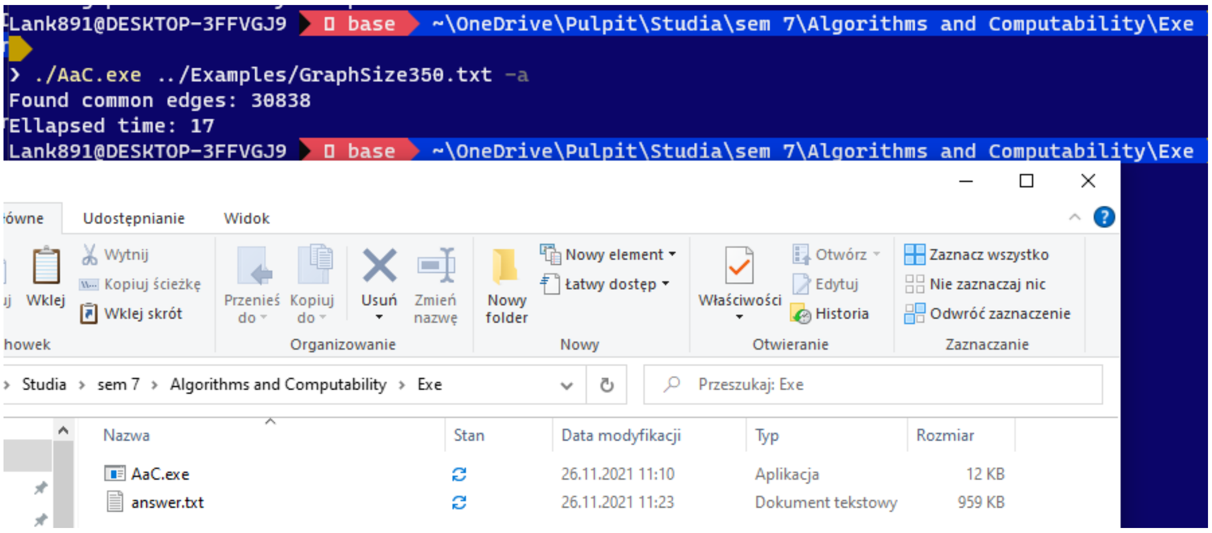
7.2 Executable building process

In order to build the executable, the solution file should be opened in Visual Studio. VS can build projects written with Net Framework 4.6 using standard “Run”. Executable should be built in either “bin/debug” or “bin/release” directory (depending on the option set in Visual Studio). The default executable name is “AaC.exe”.

7.3 Launching

Open the command line in any directory. Run the program via command line by providing its path (including program name and extension) with arguments. The arguments should be:

- path to the input file (mandatory)
- “-a” flag, to run the approximate algorithm instead of the exact one
- “-i” flag, to skip creating the output file (writing output file is the bottleneck in the program, if one is interested only in the number of common edges and execution time, those are available in the console after the algorithm is performed).



```
Lank891@DESKTOP-3FFVGJ9  base  ~\OneDrive\Pulpit\Studia\sem 7\Algorithms and Computability\Exe
> ./AaC.exe ../Examples/GraphSize350.txt -a
Found common edges: 30838
Ellapsed time: 17
Lank891@DESKTOP-3FFVGJ9  base  ~\OneDrive\Pulpit\Studia\sem 7\Algorithms and Computability\Exe
```

Nazwa	Stan	Data modyfikacji	Typ	Rozmiar
AaC.exe		26.11.2021 11:10	Aplikacja	12 KB
answer.txt		26.11.2021 11:23	Dokument tekstowy	959 KB

Figure 1. Example program launch. CLI is opened in the directory of the executable “AaC.exe” - so the program is called via “./AaC.exe”. Then there is a relative path to the input file. Flag “-a” was added to use an approximate algorithm. File “answer.txt” with the output was created in the same folder where CLI is opened.

7.4 Output

In the examples folder, we provided graphs that have from 3 to 3000 vertices. Since the output for the larger graphs could possibly crash the console, we decided to output to the console only the solutions for which the larger of the two graphs has at most 25 vertices.
For all graphs, the file "*answer.txt*" will be created in the folder where the user has opened their console (except when "-i" flag was provided. In the console and in the file one can find:

- information about type of algorithm (approximate/exact),
- number of common edges (we assumed graph can be directed so if it is not, all edges are counted **twice** - in both directions),
- Execution time of the algorithm in milliseconds (without reading input and writing to file),
- mapping sequence - which vertex of bigger graph corresponds to the n-th vertex of smaller one with all excessive vertices added to the end of the sequence,
- both of the original graphs,
- "Mapped Sequence on the Larger Graph" - larger graph where vertices are in the same order as in the mapping sequence,
- "Common subgraph/supergraph on larger graph" - larger graph with additional edges.

Two last graphs are the smallest(minimal) common supergraph found by the algorithm.
If the user wanted to see the largest(maximal) common subgraph found by the algorithm, it would be enough to read only edges marked with '2' and remove all those marked with '1'.

Additionally, the user can see the number of common edges and algorithm execution time in milliseconds at the bottom of the console - even when graphs are large and an option to skip output file was provided.

8. Tests

All the tests were performed on randomly generated directed graphs of the same size. The size of the graphs are defined by the number of vertices in the graph. The graphs used for creating plots are included in the Examples folder. The names of these example inputs are of the form “a_b_type” where a and b refer to the size(number of vertices) of graphs included in the file. The “type” part of the name describes whether the graphs included are isomorphic, whether they were randomly generated or the density of the graphs(amount of edges).
For both algorithms (exact and approximate), the obtained time is a combination of finding both maximal common subgraph and minimal common supergraph. All the tests were done on “debug” build without additional optimizations.

8.1 Correctness of the exact algorithm

We started by testing correctness of the exact algorithm by checking it on several types of graphs:

- isomorphic graphs(graphs are isomorphic if there exists a bijection between their vertex sets that preserves the number of edges between vertices),
- graph and its subgraph,
- small graphs that we solved by hand.

After testing about 15 different inputs we decided that it is enough to experimentally show the correctness of the algorithm.

8.2 Execution time of the exact algorithm

For the exact algorithm, we used 6 pairs of graphs that have between 5 and 10 vertices. The results can be seen in Table 3.

Size	5	6	7	8	9	10
Time [ms]	1	2	13	124	1418	17016

Table 3. Time of running exact algorithm for graphs of small size

Expected time complexity was of the order $O(n^2 \cdot n!)$. We created a graph (with logarithmic scale) of the time complexity and displayed it in Figure 2. In that figure, the horizontal axis corresponds to the size of both of the given graphs while the vertical one corresponds to the measured time. The blue points represent time values, while the orange ones represent the values of the function, $\frac{n^2 \cdot n!}{20000}$, fitted as closely as possible to the time values. We connected the blue points with a line and orange points with a line, in order to show where these values differ.

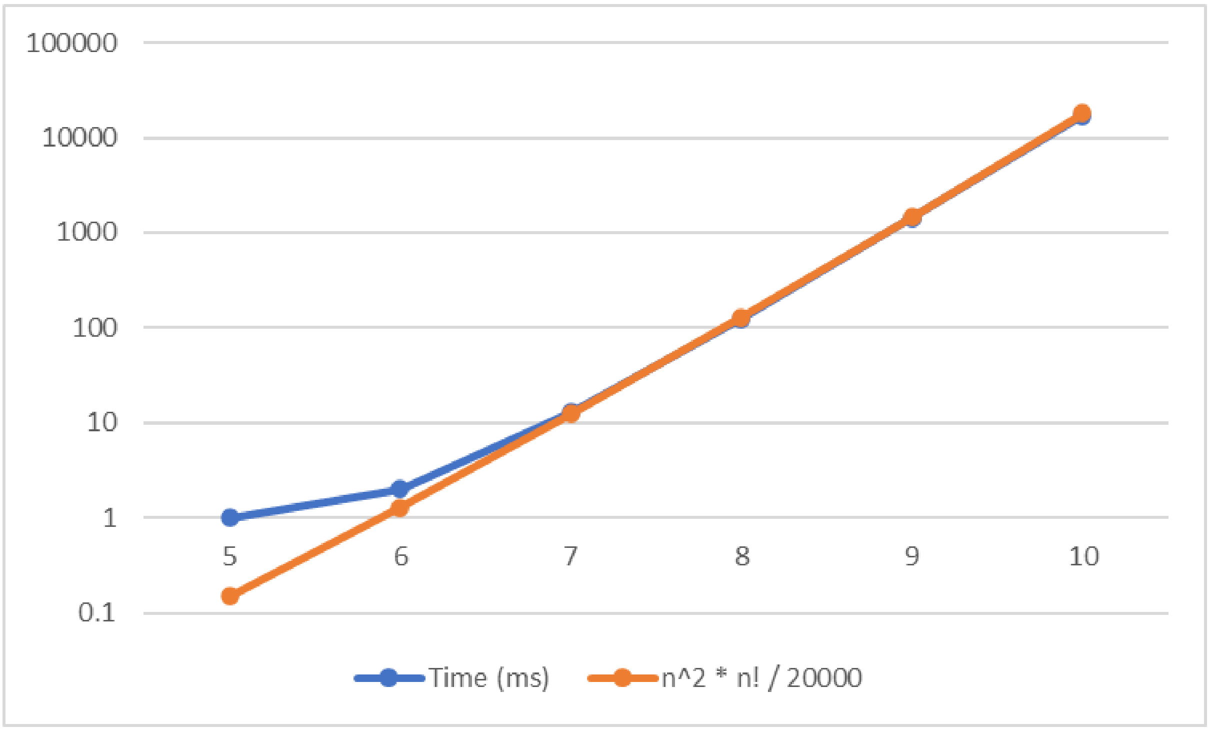


Figure 2. Time of execution of the exact algorithm with a function fitting the points

8.3 Execution time of the approximate algorithm

Approximate algorithm was tested in similar fashion but with 8 bigger graphs - the sizes were between 800 and 2000 vertices with step 200 and one additional graph with 3000 vertices to check if the formula predicts time well (formula was made from data between 800 and 2000, then prediction for 3000 was calculated and checked, if it is close enough). The times can be seen in Table 4.

Size	800	1000	1200	1400	1600	1800	2000	3000
Time [ms]	46	78	93	130	175	219	274	622

Table 4. Time of running approximate algorithm for graphs of bigger size

The expected time complexity was $O(n^2)$ and so we tried to fit the function of the form $an^2 + bn + c$ to match the times with a satisfactory result. Similarly to the exact algorithm, the times and the values of fitted curves can be seen in Figure 3.

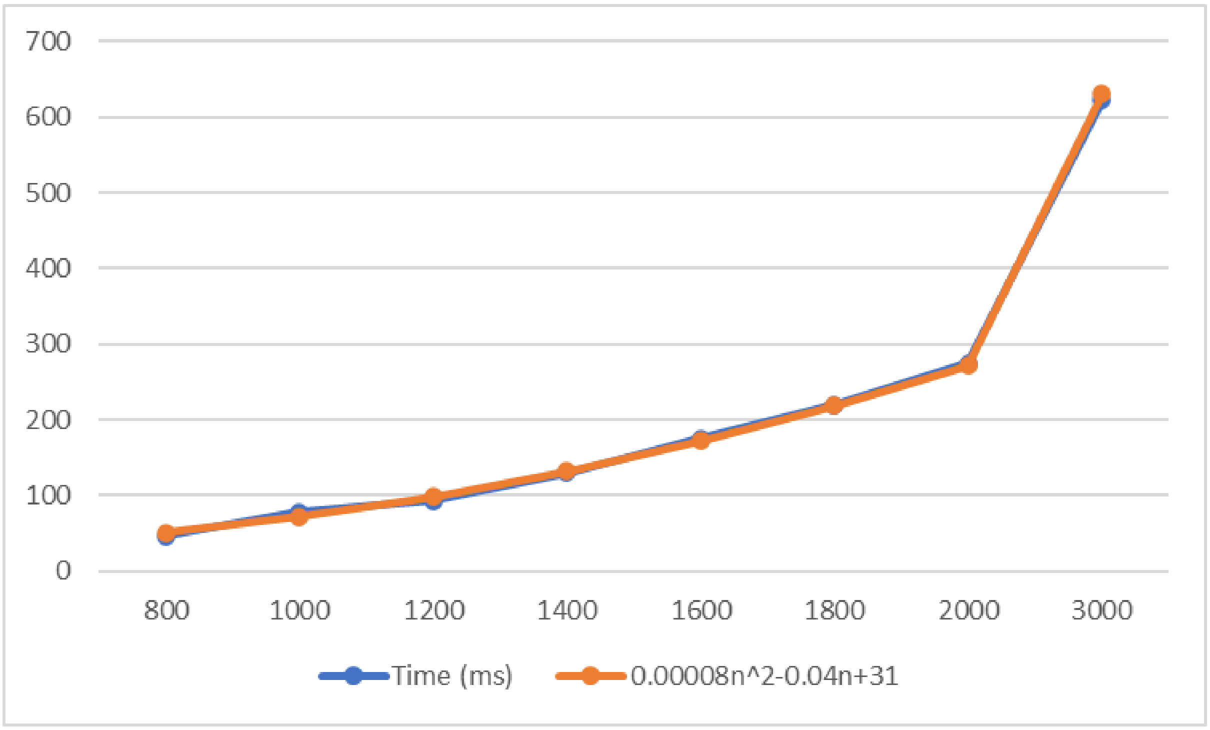


Figure 3. Time of execution of the approximate algorithm with a function fitting the points. Note that the X axis does not have equal steps at the end.

8.4 Comparison

There is no doubt that the approximate algorithm runs much faster - graphs with 3000 vertices need less than a half the time of the exact algorithm for a graph with 9 vertices in the exact algorithm. Additionally we compared the number of common edges found by both algorithms for small graphs (between 5 and 10 vertices). The results are visible in the following table:

Size	5	6	7	8	9	10
Exact	14	16	18	24	26	34
Approx	11	12	14	19	16	26
Found [%]	78.57143	75	77.77778	79.16667	61.53846	76.47059
Average [%]	74.75415					

Table 5. Comparison of common edges found in smaller graphs for exact and approximate algorithms

The number of vertices found by the approximate algorithm is between 60 and 80%. It does not depend on the size. The average is around 75%.
We were able to find special graphs where the approximate algorithm is unusable. Due to the fact that in case of several vertices with the same degree the first one in the input file is chosen, we can construct a pair of isomorphic graphs where the approximate algorithm could show 0 common edges (*5_5_iso.txt*). The created algorithm could be improved further by adding better criteria in case of conflict.

9. Bibliography

[1] Maximum common subgraph: some upper bound and lower bound results https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-7-S4-S6 [27.10.2021]

[2] On the Minimum Common Supergraph of Two Graphs https://link.springer.com/article/10.1007/PL00021410 [27.10.2021]

[3] Printing all permutations of a given string https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/ [27.10.2021]

[4] Definition of graphs https://mathworld.wolfram.com/ [03.11.2021]

[5] Definition of a supergraph https://sciendo.com/pdf/10.2478/forma-2018-0009 [03.11.2021]