Algorithms and Computability

Finding maximum common subgraph
and minimum common supergraph
for given two graphs

Project description

Mateusz Kubiszewski
Szymon Świderski
Emilia Wróblewska

November 26, 2021

Contents

1 Introduction

The objective of this project is to try to solve two problems: finding a maximum common subgraph and finding a minimum common supergraph for any two given graphs. We assume that input graphs are directed, unweighted, don't have to be connected and don't have loops. In this document our team will describe both problems, propose an algorithm designed for each problem and explain how they work.

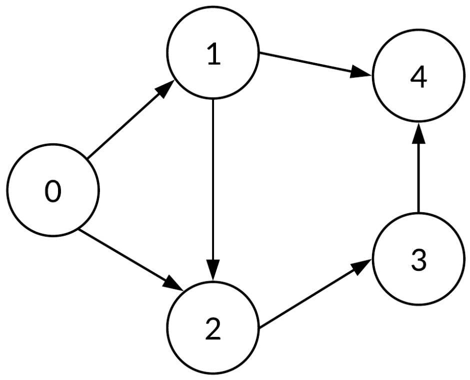
2 Problem description

There are a few common assumptions for both topics, hence they will be discussed before proceeding to the problems' descriptions. As it is mentioned in the introduction, we assume the input graphs to be:

- non-empty
- directed
- unweighted
- not necessarily connected
- without loops (i.e. without vertices connected to themselves)
- without multiple edges (we only allow a bidirectional connection between vertices, i.e. for any 2 vertices a, b in a graph, there can only be 1 edge from a to b, and one from b to a)

Moreover, all graphs processed by the algorithm will be represented using *adjacency matrices*, which store information about connections between vertices in an $|V| \times |V|$ arrays, where $|V|$ is the number of vertices in any given graph $G = (V, E)$. Considering described assumptions, all matrices in our program will contain only 0's or 1's, where a 1 at position (i, j) represents an edge directed from vertex i to vertex j. Naturally, for graphs without loops, all main diagonals entries of their matrices must be 0.

There are also terms, related to the adjacency matrices, that we will frequently use to describe algorithms in our documentation:
matrix permutation - a matrix representing a graph isomorphic to the original one, i.e. a matrix obtained by swapping corresponding rows and columns simultaneously (swapping rows 1 and 3 we must also swap columns 1 and 3)
isomorphism of graphs - an edge-preserving bijection between the vertex sets of two graphs



Adjacency Matrix

	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

Example of a graph with its matrix representation

2.1 Maximum common subgraph

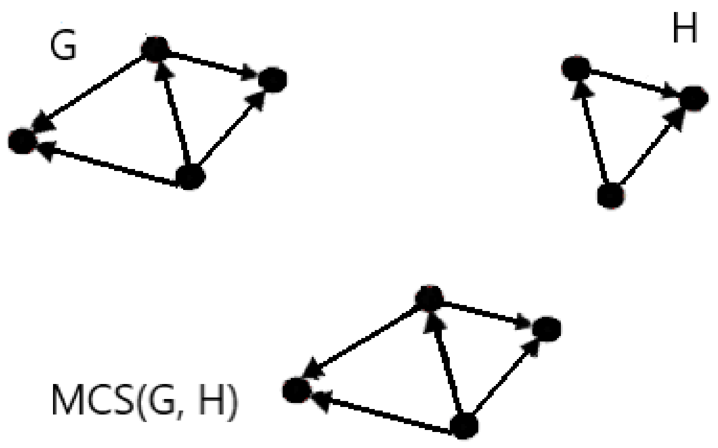
In graph theory, the *maximum common subgraph* may refer to either *maximum common induced subgraph*, which is an induced subgraph of two given graphs and has as many vertices as possible, or *maximum common edge subgraph*, which is a subgraph of two given graphs and has as many edges as possible. Given the choice, our team decided to implement an algorithm for finding the *maximum common edge subgraph* and the problem is defined as follows.

Given any 2 graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, which satisfy the assumptions, the objective is to find a graph H isomorphic to both a subgraph of G_1 and a subgraph of G_2 with as many edges as possible. An isomorphism for graphs is an edge-preserving bijection between the vertex sets of G and H. This means that any two vertices u and v of G are adjacent in G if and only if the corresponding vertices in H are adjacent in H.
The search for maximum common subgraph is performed through a simple maximization of the sum of common edges counted for each matrix permutation of two input graphs.

2.2 Minimum common supergraph

The *minimum common supergraph* of two graphs, G_1 and G_2, is defined as the smallest graph that includes, as subgraphs, both G_1 and G_2. We take into account any isomorphic representations of G_1 and G_2 as possible subgraphs of the desired minimum supergraph. Again, we aim to find the supergraph with the smallest number of edges possible and don't consider the vertices.

We observed that minimum common supergraph computation can be solved by means of maximum common subgraph computation, since both algorithms are based on counting edges of input graphs. The search for minimum common supergraph is performed by simple minimization of the sum of all edges found for each matrix permutation of two input graphs.



Minimum common supergraph

Finding maximal common subgraph and minimal common supergraph are NP-complete problems. In other words, they have a nondeterministic polynomial-time hardness and it is not known whether any polynomial-time algorithms will ever be found for solving those problems. Hence, our team designed an exact solution for each problem with factorial time complexity. The exact solutions are extremely inefficient and can be run only for very small graphs (i.e. with ≤ 10 vertices) to finish in reasonable amount of time. However, when an NP-complete problem must be solved, one approach is to use a polynomial algorithm to approximate the solution; the answer thus obtained will not necessarily be optimal but will be reasonably close. That is why, in order to run our algorithms for larger graphs, our team designed also an approximate solution with polynomial complexity for each problem, trying to make them as precise as possible.

3 Maximum common subgraph - Exact solution

The exact algorithm tries to obtain the solution by brute force, which means it enumerates all possible solutions and checks them against fitness function to return the best one. This approach is very straight-forward and it always chooses the best result but at the cost of very high time complexity (in this case - exponential complexity).

3.1 Pseudocode

Algorithm 1 *Maximum common subgraph (exact solution)*

```
1: $M_1 \leftarrow$ Adjacency matrix representing graph $G_1$
2: $M_2 \leftarrow$ Adjacency matrix representing graph $G_2$
3: $V_1 \leftarrow$ Set of all vertices in graph $G_1$
4: $V_2 \leftarrow$ Set of all vertices in graph $G_2$
Ensure: $|V_1| \geq |V_2| > 0$        $\triangleright$ Both matrices are not empty, and $M_1 \geq M_2$
5: $maxSubGraph \leftarrow$ Empty Matrix
6: $maxCommonEdges \leftarrow 0$
7: for each $M = GetAllPermutations(M_1)$ do
8:     $V \leftarrow$ Set of all vertices in graph $M$
9:     $x \leftarrow 0$
10:    while $x \leq (|V_1| - |V_2|)$ do
11:        $y \leftarrow 0$
12:        while $y \leq (|V_1| - |V_2|)$ do
13:            $subMatrix \leftarrow GetSubMatrix(x, y, M, |V_2|)$
14:            $commonMatrix \leftarrow FindCommonMatrix(subMatrix, M_2)$
15:            $E \leftarrow$ Set of all edges in graph $commonMatrix$
16:            if $|E| > maxCommonEdges$ then
17:                $maxCommonEdges \leftarrow |E|$
18:                $maxSubGraph \leftarrow commonMatrix$
19:            end if
20:            $y \leftarrow y + 1$
21:        end while
22:        $x \leftarrow x + 1$
23:    end while
24: end for
25: return $maxSubGraph$
```

This solution works by firstly obtaining all possible permutations of the larger matrix with *GetAllPermutations* function. Then it iterates through all of the obtained matrices and compares each permutation of the larger matrix with each permutation of the smaller matrix in the following way: from larger matrix take a submatrix of the same size as the smaller matrix (returned by method *GetSubMatrix*) and obtain an "intersection" of those two matrices with method *FindCommonMatrix*. This method returns a matrix with 1's only in entries where both input matrices have 1's as well and all other entries are 0. Then, the algorithm counts edges in the obtained "intersection" matrix and if the value is highest so far, the results are saved by overwriting the *maxCommonEdges* and *maxSubGraph* variables. After the algorithm finishes, the best solution saved in the *maxSubGraph* variable is returned.

NOTE: All methods used in this algorithm are common for all presented solutions and are described in section 7.

3.2 Complexity analysis

For the exact solution, the upper bound of complexity will be analysed. As per the definition, $g(n)$ is an upper bound of $f(n)$ (denoted by $f = \mathcal{O}(g)$) if the following condition holds:

$$f = \mathcal{O}(g)) \iff \exists_{c>0}\exists_{n_0 \in N}\forall_{n>n_0} c * g(n) \geq f(n)$$

Firstly, we assume that the input matrices M_1 and M_2 have the respective numbers of vertices $|V_1| = n$ and $|V_2| = k$ and $n \geq k$, hence we can use only n to bound the time complexity. The algorithm starts by generating all possible permutations of the larger matrix which gives the complexity of $\mathcal{O}(n!)$. Then, iterating through all those permutations adds another $\mathcal{O}(n!)$ which needs to be multiplied by $\mathcal{O}(n^2)$ because of the nested loop. Then, for current permutation of the larger matrix we perform two operations: *GetSubMatrix* ($\mathcal{O}(n^2)$ complexity) and *FindCommonMatrix* (also $\mathcal{O}(n^2)$). The final time complexity of the whole algorithm can be denoted as function T:

$$T(n) = n! + n! \cdot n \cdot n \cdot (n^2 + n^2)$$

$$T(n) = n! \cdot n^2 \cdot n^2$$

$$T(n) = \mathcal{O}(n! \cdot n^4)$$

As we can see, the complexity of this algorithm is extremely high and adding even one vertex to any of the input matrices would increase the computation time exponentially. However, at a cost of this tremendously high complexity, we can be sure that the algorithm find the best possible solution and the resulting graph is indeed the maximum common subgraph of the input graphs.

4 Minimum common supergraph - Exact solution

The exact algorithm for the minimum common supergraph problem is very similar to the previous one - it also enumerates all possible solutions and checks them against fitness function to return the most optimal one. However, the similarity also applies for the exponential complexity, which makes this solution extremely inefficient.

4.1 Pseudocode

Algorithm 2 *Minimum common supergraph (exact solution)*

```
1: $M_1 \leftarrow$ Adjacency matrix representing a graph $G_1$
2: $M_2 \leftarrow$ Adjacency matrix representing a graph $G_2$
3: $V_1 \leftarrow$ Set of all vertices in a graph $G_1$
4: $V_2 \leftarrow$ Set of all vertices in a graph $G_2$
Ensure: $|V_2| \geq |V_1| > 0$        ▷ Both matrices are not empty and $M_2 \geq M_1$
5: $SmallestSuperGraph \leftarrow$ Empty Matrix
6: $minCommonEdges \leftarrow$ Maximal integer number
7: for each $M = GetAllIsomporphicGraphsTo(M_2)$ do
8:     $x \leftarrow 0$
9:     while $x \leq (|V_2| - |V_1|)$ do
10:        $y \leftarrow 0$
11:        while $y \leq (|V_2| - |V_1|)$ do
12:            $newMatrix \leftarrow$ Copy of M
13:            $newMatrix \leftarrow InsertEdgesToMatrixAt(newMatrix, x, y, M_1)$
14:            if $|newMatrix.Edges| < minCommonEdges$ then
15:                $minCommonEdges \leftarrow newMatrix.Edges$
16:                $SmallestSuperGraph \leftarrow newMatrix$
17:            end if
18:            $y \leftarrow y + 1$
19:        end while
20:        $x \leftarrow x + 1$
21:    end while
22: end for
23: return $SmallestSuperGraph$
```

This algorithm checks all possible common super graphs and saves the smallest one. It starts by generating all graphs isomorphic to the larger graph (i.e. one with more vertices). For every found matrix permutation, denoted by variable M, it iterates through a fixed part of it and then generates all permutations of the smaller matrix. Each smaller matrix permutation, denoted by B, is "fitted" into the larger matrix permutation M in the following way: left upper corner of B is placed in the current (x, y) position of the larger matrix M and all edges that are present in B are copied into corresponding places in M. This is done by the method *InsertEdgesToMatrixAt* which returns a new matrix with edges from both input matrices. The number of edges in this resulting matrix, denoted by the variable *newMatrix*, is then compared with the lowest saved value so far and if its even lower, the best solution stored in the *SmallestSuperGraph* variable is overwritten. After the algorithm finishes, the

SmallestSuperGraph is returned as the most optimal solution found.

Generally, using the variables defined in the algorithm, the search for the mimnimum common supergraph is based on the following assumptions:

If every found position in M is described by x and y then the common super graph of M and B is a graph C defined as follows:

$$M = \{a_{0,0}, a_{0,1}, ..., a_{n,n}\}$$

$$B = \{b_{0,0}, b_{0,1}, ..., b_{m,m}\}$$

$$C = \{c_{0,0}, c_{0,1}, ..., c_{n,n}\}$$

$$a_{i,j} = 1 \Rightarrow c_{i,j} = 1$$

$$b_{i,j} = 1 \Rightarrow c_{i-x,j-y} = 1$$

Such a definition of C represents a common super graph for every isomorphic matrix M and every correct position (x, y) in M. Minimal common super graph is a graph C that has the smallest amount of edges.

4.2 Complexity analysis

Similarly to the maximum common subgraph algorithm, we firstly assume that the input matrices M_1 and M_2 have the respective numbers of vertices $|V_1| = k$ and $|V_2| = n$ and $n \geq k$. Again, for the simplicity of the notation we define the time complexity as a function of n, as the size of greater matrix bounds the worst possible scenario of the solution. This algorithm also starts by generating all possible permutations of the larger matrix and iterating through them ($\mathcal{O}(n!)$ complexity). For each permutation, it then performs a doubly nested loop with complexity $\mathcal{O}(n^2)$ in which there is only one more complex method - *InsertEdgesToMatrixAt* with complexity $\mathcal{O}(n^2)$. Hence, the complexity of the whole algorithm can be represented by:

$$T(n) = n! + n! \cdot n^2 \cdot n^2$$

$$T(n) = n! \cdot n^2 \cdot n^2$$

$$T(n) = \mathcal{O}(n! \cdot n^4)$$

Complexity of this algorithm is the same as that of the maximum common subgraph solution. And again, at the cost of extreme inefficiency we obtain the best possible solution to the minimum common supergraph problem.

5 Maximum common subgraph - Approximate solution

Quite opposite to the brute force approach, in approximate solution the goal is not to find the best possible answer, but to find one closest to the optimal, with as low time complexity as possible. In this case, we aim to reduce the complexity to a polynomial time in order to be able to test the solution for graphs larger than just 10 vertices.
This algorithm is very similar to the exact solution - to approximate the possible maximum common subgraph we decided to "sort" input graphs instead of obtaining all of their permutations. "Sorting" transforms the input graphs into similar forms and hence, increases the probability of finding the best solution. This way we should end up with a maximum common subgraph close to the one obtained by the exact algorithm.

5.1 Pseudocode

Algorithm 3 *Maximum common subgraph (approximate solution)*

```
1: $M_1 \leftarrow$ Adjacency matrix representing graph $G_1$
2: $M_2 \leftarrow$ Adjacency matrix representing graph $G_2$
3: $V_1 \leftarrow$ Set of all vertices in graph $G_1$
4: $V_2 \leftarrow$ Set of all vertices in graph $G_2$
Ensure: $|V_1| \geq |V_2| > 0$        ▷ Both matrices are not empty and $M_1 \geq M_2$
5: $SortedMatrixA \leftarrow TransformToSortedForm(M_1)$
6: $SortedMatrixB \leftarrow TransformToSortedForm(M_2)$
7: $maxSubGraph \leftarrow$ Empty Matrix
8: $maxCommonEdges \leftarrow 0$
9: $x \leftarrow 0$
10: while $x \leq |V_1| - |V_2|$ do
11:     $y \leftarrow 0$
12:     while $y \leq |V_1| - |V_2|$ do
13:         $subMatrix \leftarrow GetSubMatrix(x, y, M_1, |V_2|)$
14:         $commonMatrix \leftarrow FindCommonMatrix(subMatrix, M_2)$
15:         $E \leftarrow$ Set of all edges in graph $commonMatrix$
16:         if $|E| > maxCommonEdges$ then
17:             $maxCommonEdges \leftarrow |E|$
18:             $maxSubGraph \leftarrow commonMatrix$
19:         end if
20:         $y \leftarrow y + 1$
21:     end while
22:     $x \leftarrow x + 1$
23: end while
24: return $maxSubGraph$
```

In this algorithm, instead of generating isomorphic graphs, the input graphs are firstly transformed into "sorted" forms, i.e. they have as many 1's as possible in the upper left part of the matrix. It is done by the method *TransformToSortedForm* which is described in more detail in section 7. Then, the algorithm iterates through a part of the larger matrix and, just as in the exact solution, for each submatrix of size equal to the smaller input matrix, finds an "intersection" matrix, counts its edges and compares the result with the highest value so far. The results are saved by overwriting the *maxCommonEdges* and *maxSubGraph* variables, which are then returned as the final result.

5.2 Complexity analysis

We assume that the input matrices M_1 and M_2 have the respective numbers of vertices $|V_1| = n$ and $|V_2| = k$ and $n \geq k$. Again, we can bound the complexity by the size of greater matrix - n. This algorithm firstly sorts the two input matrices and this operation has time complexity $\mathcal{O}(n^2 + n^2)$ as described in section 7. Then, iterating through nested loop of size n gives $\mathcal{O}(n^2)$ complexity which has to be multiplied by $\mathcal{O}(n^2) + \mathcal{O}(n^2)$ from *GetSubMatrix* and *FindCommonMatrix* functions. In overall, the time complexity of the whole algorithm can be denoted as function T:

$$T(n) = n^2 + n^2 + n \cdot (n \cdot (n^2 + n^2))$$

$$T(n) = n^2 + n^2 \cdot n^2$$

$$T(n) = \mathcal{O}(n^4)$$

We obtained a polynomial time complexity which is much better than in the exact solution and would allow for testing the algorithm against much bigger inputs.

6 Minimum common supergraph - Approximate solution

The approximate solution for minimum common supergraph also aims at reducing the complexity to a polynomial time, because we want to test it on larger graphs.
This algorithm is very similar to the exact solution described above, but instead of generating all permutations of both input matrices, it transforms them into "sorted" form with *TransformToSortedForm* function, described in section 7. Then, the main loop goes through a part of the bigger matrix in such a way that at each iteration of the loop, variables x and y describe the start position in the larger matrix at which the smaller matrix should be "fitted". Then, the larger matrix is copied and all edges from the smaller matrix are inserted into the copied matrix at appropriate positions - this logic is the same as in the exact solution. Finally, the graph with inserted edges is compared with the most optimal result so far and saved if it has even less edges than the current result. At the end, the algorithm returns the approximated

minumum common supergraph which should be fairly close to the optimal solution.

6.1 Pseudocode

Algorithm 4 *Minimum common supergraph (approximate solution)*

```
 1: $M_1 \leftarrow$ Adjacency matrix representing a graph $G_1$
 2: $M_2 \leftarrow$ Adjacency matrix representing a graph $G_2$
 3: $V_1 \leftarrow$ Set of all vertices in a graph $G_1$
 4: $V_2 \leftarrow$ Set of all vertices in a graph $G_2$
Ensure: $|V_2| \geq |V_1| > 0$        ▷ Both matrices are not empty and $M_2 \geq M_1$
 5: $M_1 \leftarrow TransformToSortedForm(M_1)$
 6: $M_2 \leftarrow TransformToSortedForm(M_2)$
 7: $SmallestSuperGraph \leftarrow$ Empty Matrix
 8: $minCommonEdges \leftarrow$ Maximal integer number
 9: $x \leftarrow 0$
10: while $x \leq (|V_2| - |V_1|)$ do
11:     $y \leftarrow 0$
12:     while $y \leq (|V_2| - |V_1|)$ do
13:         $newMatrix \leftarrow$ Copy of M
14:         $newMatrix \leftarrow InsertEdgesToMatrixAt(newMatrix, x, y, B)$
15:         if $|newMatrix.Edges| < minCommonEdges$ then
16:             $minCommonEdges \leftarrow newMatrix.Edges$
17:             $SmallestSuperGraph \leftarrow newMatrix$
18:         end if
19:         $y \leftarrow y + 1$
20:     end while
21:     $x \leftarrow x + 1$
22: end while
23: return $SmallestSuperGraph$
```

6.2 Complexity analysis

Just as in all previous cases, we firstly need to assume that the input matrices M_1 and M_2 have the respective numbers of vertices $|V_1| = k$, $|V_2| = n$ and $n \geq k$. The algorithm starts by sorting both input matrices which is done in $n^2 + n^2$ time, as described in the *TransformToSortedForm* function. Then, iterating through part of the larger matrix has $(\mathcal{O}(n^2)$ complexity. In each loop iteration, there are only two costly operations - copying matrix and inserting edges into larger matrix, which is done in n^2 and n^2 time. Since $n \geq k$, we can use only n to denote the worst-case scenario of the algorithm and the complexity can be represented by:

$$T(n) = n^2 + n^2 + n \cdot n \cdot (n^2 + n^2)$$

$$T(n) = n^2 + \cdot n^2 \cdot n^2$$

$$T(n) = \mathcal{O}(n^4)$$

This algorithm has polynomial time complexity which is incomparably better than in the brute force approach. We are able to obtain reasonably optimal solutions in a fraction of the exact algorithm time.

7 Common methods

This section describes methods used in all previously discussed solutions. Each method is presented with a pseudocode, a description of how it works and its time complexity analysis.

1) **GetSubMatrix**

Algorithm 5

```
procedure GetSubMatrix($xIndex, yIndex, Matrix, size$)
    $SubMatrix \leftarrow$ Empty Matrix
    $i \leftarrow$ xIndex
    while $i < xIndex + size$ do
        $j \leftarrow yIndex$
        while $j < yIndex + size$ do
            SubMatrix[i - xIndex][j - yIndex] = Matrix[i][j]
            $j \leftarrow$ j + 1
        end while
        $i \leftarrow$ i + 1
    end while
    return $SubMatrix$
end procedure
```

The *GetSubMatrix* method returns a square submatrix of given size which starts at position $(xIndex, yIndex)$ in the input matrix. To obtain the result, it copies numbers from the input matrix row by row, starting from the row given by $xIndex$; in each row it starts from the column given by $yIndex$ and proceeds to the next row when *size* numbers are copied.

Time complexity of this function is $T(n) = n \cdot n = \mathcal{O}(n^2)$, where $n = size$ of submatrix it returns. It is a very simple method iterating through a 2-dimensional array and copying appropriate values and hence it has quadratic complexity.

2) **FindCommonMatrix**

Algorithm 7

```
1:  procedure FindCommonMatrix($M_1, M_2$)
2:      $V_1 \leftarrow$ Set of all vertices in graph $M_1$
3:      $V_2 \leftarrow$ Set of all vertices in graph $M_2$
Ensure: $|V_1| == |V_2|$                              ▷ Both matrices have the same size
4:      $CommonMatrix \leftarrow$ Copy of $M_1$
5:      $i \leftarrow 0$
6:      while $i < |V_1|$ do
7:          $j \leftarrow 0$
8:          while $j < |V_1|$ do
9:              if $M_1[i][j] == 1$ and $M_2[i][j] == 0$ then
10:                 CommonMatrix[i][j] = 0
11:             end if
12:             $j \leftarrow$ j + 1
13:         end while
14:         $i \leftarrow$ i + 1
15:     end while
16:     return CommonMatrix
17: end procedure
```

The procedure *FindCommonMatrix* is applicable only for two matrices of the same size. It returns a third matrix of the same size which has only those edges which are present in both input matrices. To obtain the resulting matrix, it copies the first given matrix, iterates through both input matrices and checks each entry. We have four cases:

- If the entry is equal to 1 in both matrices, then it is a common edge. It is already marked as 1 in the resulting matrix, so no action needs to be taken.
- If the entry equals 1 in the first matrix and 0 in the second one, then it is not a common edge. This entry needs to be marked as 0 in the resulting matrix.
- If the entry is equal to 0 in the first matrix and to 1 in the second one, then it is also not a common edge. However, it is already marked as 0 in the resulting matrix, so no action needs to be taken.

- If the entry is equal to 0 in both matrices, then it is not a common edge and again, it is already marked as 0 in the resulting matrix.

Hence, the algorithm only needs to check for the second case, apply necessary changes and after iterating through the whole resulting matrix - return it.

Time complexity of this function is again $T(n) = n \cdot n = \mathcal{O}(n^2)$, where n is the size of either of the input matrices. Just as in the previous case, iterating through a 2-dimensional array gives quadratic complexity.

3) **GetAllIsomporphicGraphsTo(Matrix)**

Algorithm 8

```
procedure GetAllIsomporphicGraphsTo($Matrix$)
    $Matrix \leftarrow$ Matrix that will be transformed
    $P \leftarrow$ All permutations without repetitions of numbers from 0 to $|V_{Matrix}| - 1$
    $Result \leftarrow$ Empty array
    for each $array \leftarrow P$ do
        $newM \leftarrow$ Copy $Matrix$
        $i \leftarrow 0$
        while $i \leq array.size$ do
            if $array[i] \neq i$ then
                $newM \leftarrow SwapColumn(newM, array[i], i)$
                $newM \leftarrow SwapRow(newM, array[i], i)$
                $array \leftarrow Swap(array, array[i], i)$
            end if
            $i \leftarrow i + 1$
        end while
        $Result.append(newM)$
    end for
    return $Result$
end procedure
```

This function returns all graphs that are isomorphic to the given graph. It firstly generates all permutations of sequences of numbers from 0 to $|V_{Matrix}| - 1$. Each sequence represents the order of vertices in adjacency matrix of a graph. Then, the method iterates through the current sequence permutation and swaps appropriate rows and columns of the matrix, to match the order of numbers in the permutation. The result is added to the list of all generated isomorphic graphs which is returned after the loop finishes.

Time complexity of this function is $T(n) = n! + n! \cdot n = \mathcal{O}(n! \cdot n)$, where n is the number of vertices of the input graph. Because there are $n!$ permutations without repetitions of a set with size n and for every permutation the algorithm performs a loop with complexity $\mathcal{O}(n)$, the complexity of this function is very high and reaches $\mathcal{O}(n! \cdot n)$.

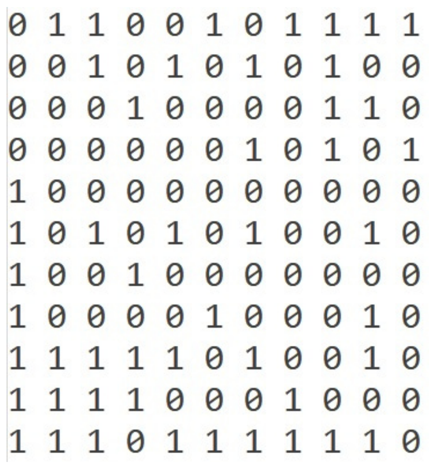
4) **TransformToSortedForm**

Algorithm 9

```
procedure TransformToSortedForm($Matrix$)
    $i \leftarrow 0$
    while $i \leq |Matrix.Vertices|$ do
        $j \leftarrow 0$
        while $j \leq |Matrix.Vertices|$ do
            if $DegRow(i) + DegCol(i) > DegRow(j) + DegCol(j)$ then
                $newM \leftarrow SwapColumn(Matrix, i, j)$
                $newM \leftarrow SwapRow(Matrix, i, j)$
            end if
            $j \leftarrow j + 1$
        end while
        $i \leftarrow i + 1$
    end while
    return $Matrix$
end procedure
```

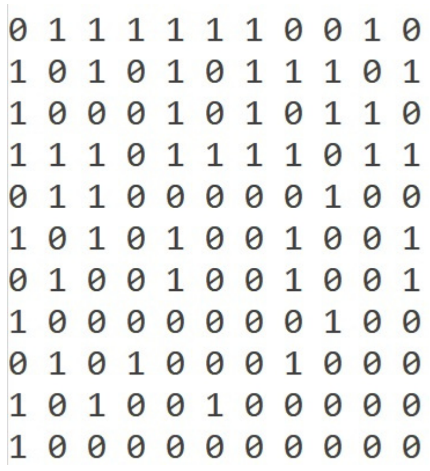
This function creates an adjacency matrix of the graph isomorphic to the given graph. The obtained matrix has rows and columns sorted in such a way that most of the ones are placed in the left top corner and most of the zeros are placed in the right bottom corner.

The algorithm structure is very similar to a simple bubble sort. Every index is compared with every other index in the matrix and if they are in the wrong order they are swapped. The value of the index is defined by the following function: $DegRow(i) + DegCol(i)$. This means that value of given index i is defined as the number of edges (sum of 1's) in i-th column and i-th row. If there exists i and j such that the value of index j is less than the value of index i, then row i is swapped with row j, and column i is swapped with column j. After checking all combinations of i and j we can be sure that the given matrix is transformed to most optimal, sorted form.



```
0 1 1 0 0 1 0 1 1 1 1
0 0 1 0 1 0 1 0 1 0 0
0 0 0 1 0 0 0 0 1 1 0
0 0 0 0 0 0 1 0 1 0 1
1 0 0 0 0 0 0 0 0 0 0
1 0 1 0 1 0 1 0 0 1 0
1 0 0 1 0 0 0 0 0 0 0
1 0 0 0 0 1 0 0 0 1 0
1 1 1 1 1 0 1 0 0 1 0
1 1 1 1 0 0 0 1 0 0 0
1 1 1 0 1 1 1 1 1 1 0
```

Example matrix before translation



```
0 1 1 1 1 1 1 0 0 1 0
1 0 1 0 1 0 1 1 1 0 1
1 0 0 0 1 0 1 0 1 1 0
1 1 1 0 1 1 1 1 0 1 1
0 1 1 0 0 0 0 0 1 0 0
1 0 1 0 1 0 0 1 0 0 1
0 1 0 0 1 0 0 1 0 0 1
1 0 0 0 0 0 0 0 1 0 0
0 1 0 1 0 0 0 1 0 0 0
1 0 1 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
```

Example matrix after translation

It is not possible to accurately place all 1's closest to one of matrix corners because the represented graph will stop being isomorphic to the original one. But generally, rows with the most 1's are placed at the top, and columns with the most 1's are placed on the left of the matrix, hence this algorithm makes the matrices of all graphs isomorphic to a given graph as similar as possible. Thanks to that, algorithms which use matrix permutations can be approximated with higher accuracy, without the need to generate all possible solutions.

Time complexity of this function is $T(n) = n \cdot n = \mathcal{O}(n^2)$, since we have only one nested loop and thanks to graphs being represented with adjacency matrices, the operations of swapping rows and columns have constant complexity.

5) **InsertEdgesToMatrix**

Algorithm 10

```
1:  procedure InsertEdgesToMatrixAt($M_B$, $x$, $y$, $M_S$)
Ensure: $|M_B| \geq |M_S| > 0$                    ▷ $M_B$ is the bigger matrix
2:      $i \leftarrow x$
3:      while $i \leq |V_{M_B}|$ and $i - x \leq |V_{M_S}|$ do
4:          $j \leftarrow y$
5:          while $j \leq |V_{M_B}|$ and $j - y \leq |V_{M_S}|$ do
6:              if $M_S[i-x][j-y] == 1$ then
7:                  $M_B[i][j] = M_S[i-x][j-y]$
8:              end if
9:              $j \leftarrow j + 1$
10:         end while
11:         $i \leftarrow i + 1$
12:     end while
13: end procedure
```

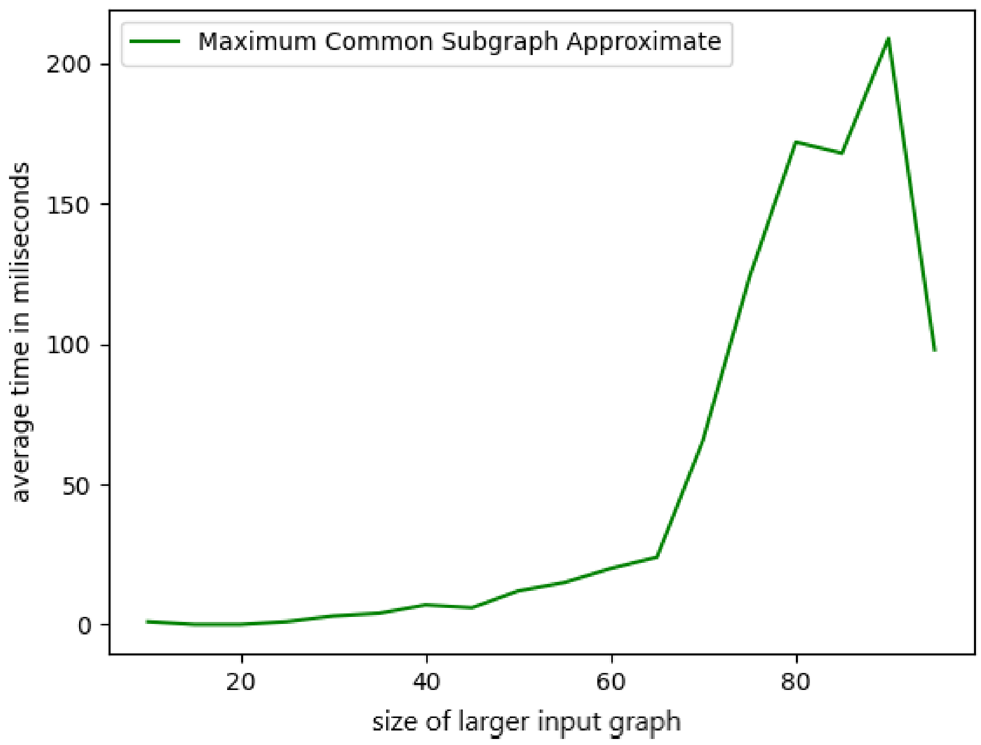
This function is used in the minimum common supergraph problem. It iterates through a submatrix of size $|V_{M_S}|$ x $|V_{M_S}|$ from the larger input matrix M_B, starting from the given position (x, y). If at a given position there is an edge in the smaller matrix, it is inserted into the larger matrix at a corresponding position. After the algorithm finishes, the larger matrix stores all edges from both graphs and makes a single solution for the minimum common subgraph problem.

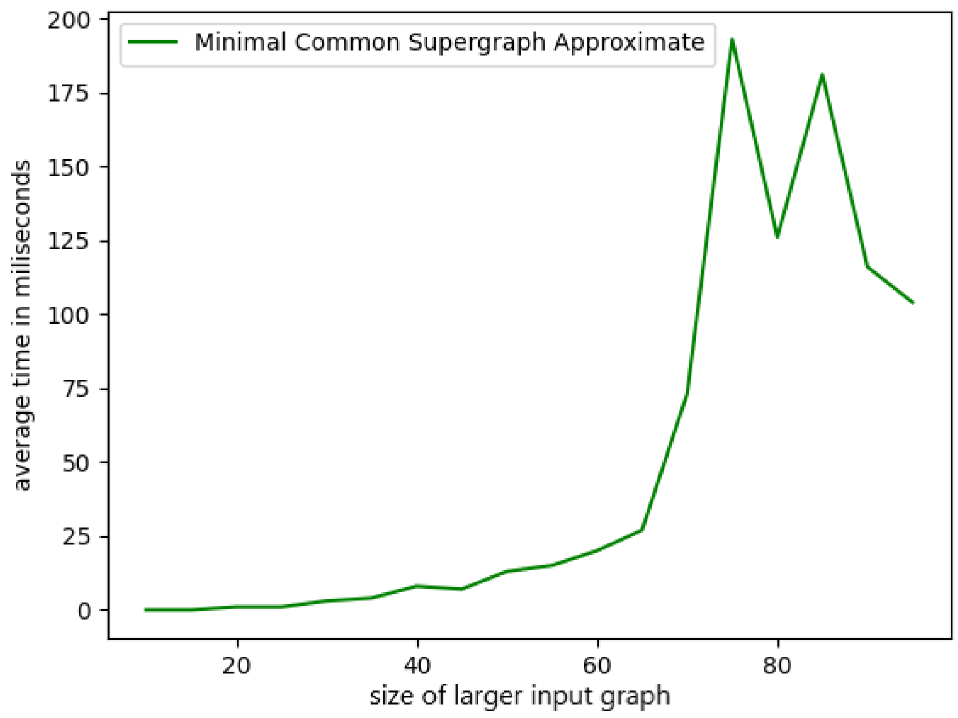
Time complexity of this function is $T(n) = n \cdot n = \mathcal{O}(n^2)$, where n is the number of vertices in the smaller graph. The complexity is again quadratic thanks to one nested loop with simple comparisons and assignment operations.

8 Results

The algorithms were tested on 72 randomly generated pairs of matrices. Those matrices are divided into two sets based on the size of the bigger matrix. To test the exact algorithms we used matrices of sizes from 4 up to 10 vertices and for testing the approximate algorithms we used sizes from 10 up to 95, incremented by 5 (i.e. 10,15,...90,95). For every set of matrices with one size, we calculate the average value of the results of the given algorithm. Because of that, we have a good representation of how the algorithm behaves for graphs of a given size. We decided to check our results in the four following categories.

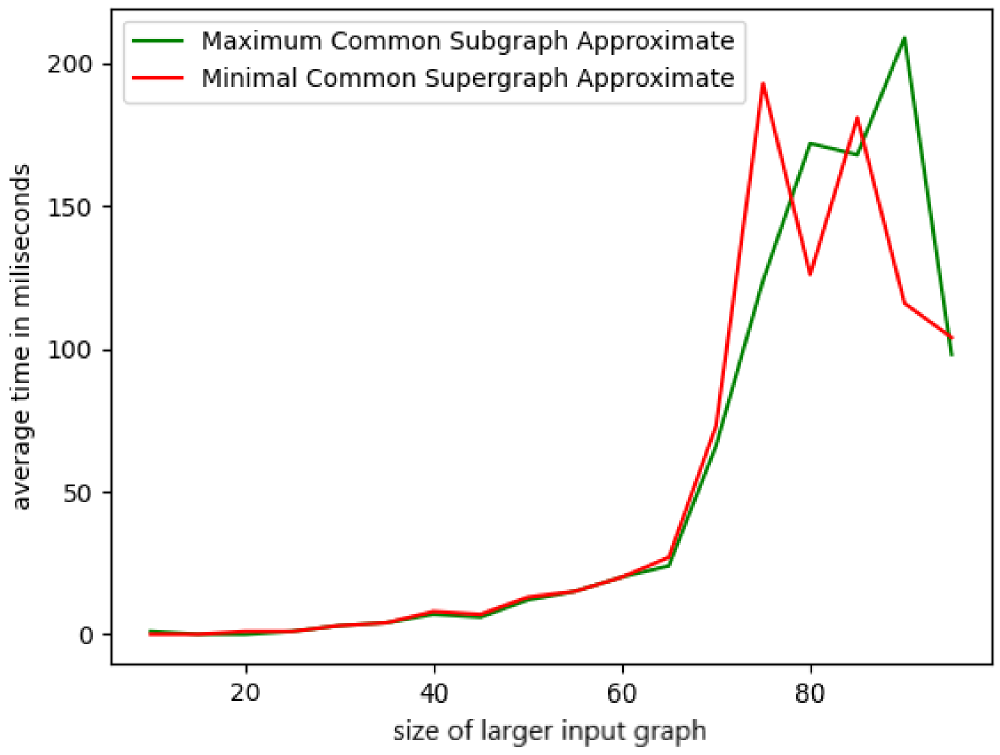
8.1 Speed tests of approximate algorithms





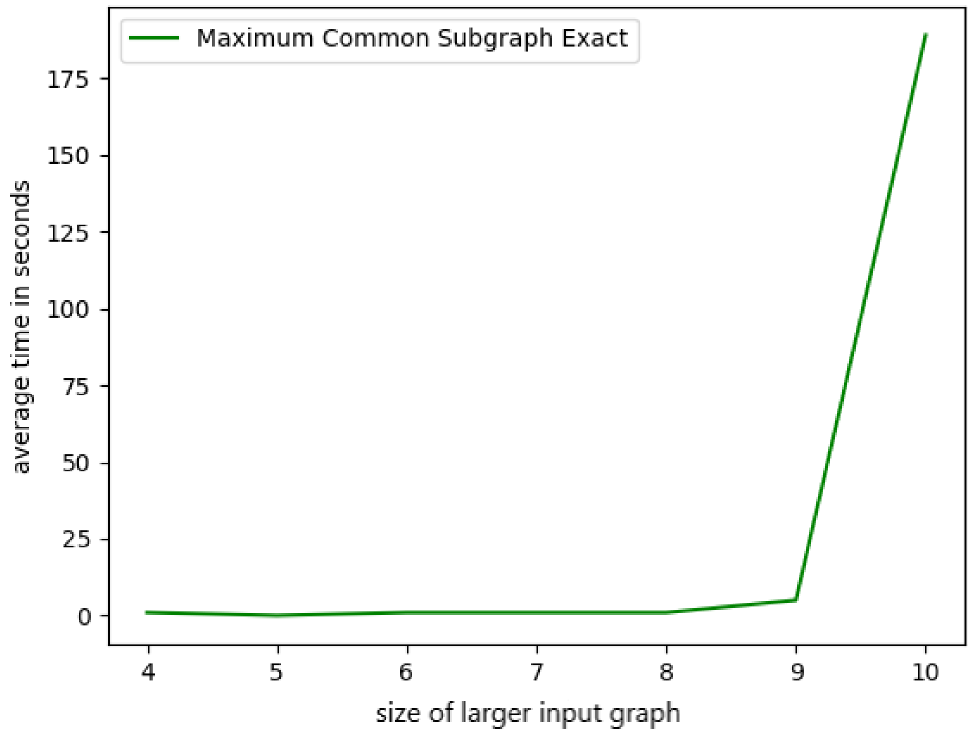
We can see that approximate algorithms are very fast as it takes around 0.2 seconds to find a maximal common subgraph for graphs with around 80 vertices. Both algorithms (for maximal subgraph and minimal supergraph) have similar time execution because they have the same complexity. We can see small distortions in this graph for graphs bigger than 80 vertices but we can neglect that fact, since graphs are generated randomly.

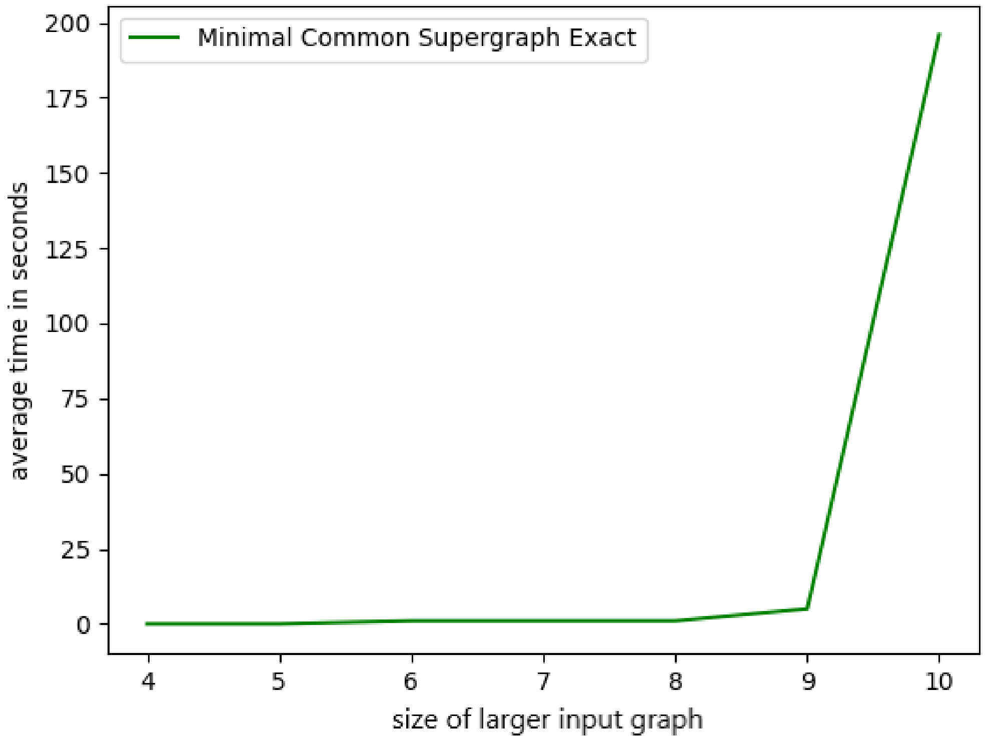
Comparison of the time execution of the maximal common subgraph and minimal common supergraph algorithms looks as follows:



As we can see, time execution for both algorithms is very similar. It comes from the fact that both algorithms were run on the same set of graphs.

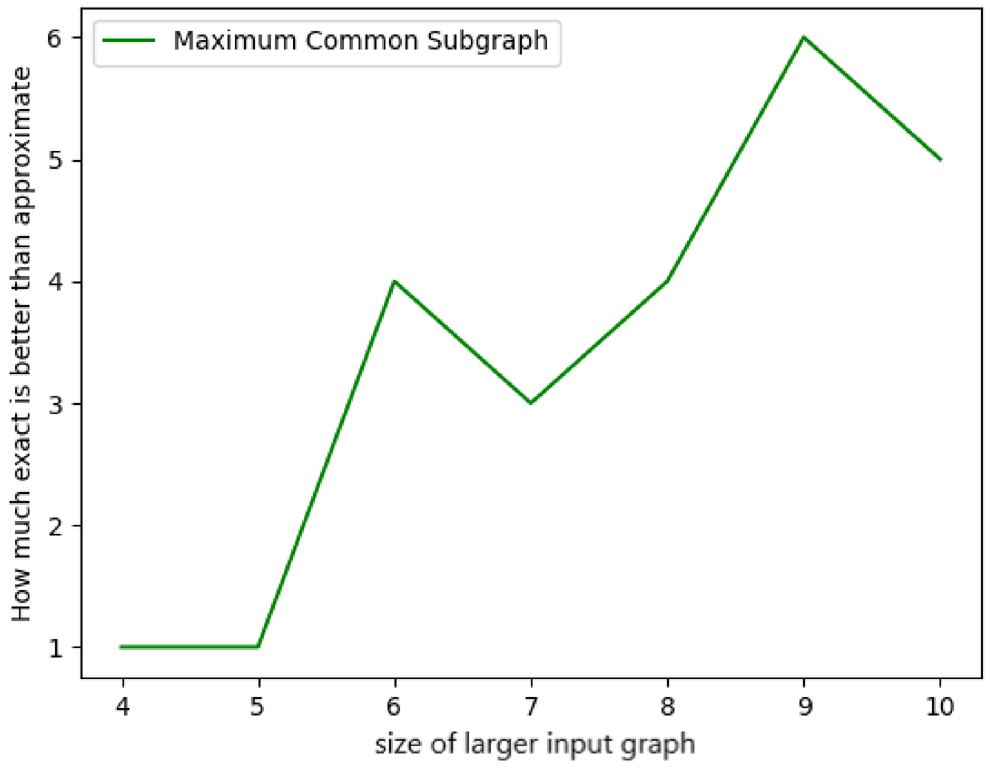
8.2 Speed tests of exact algorithms

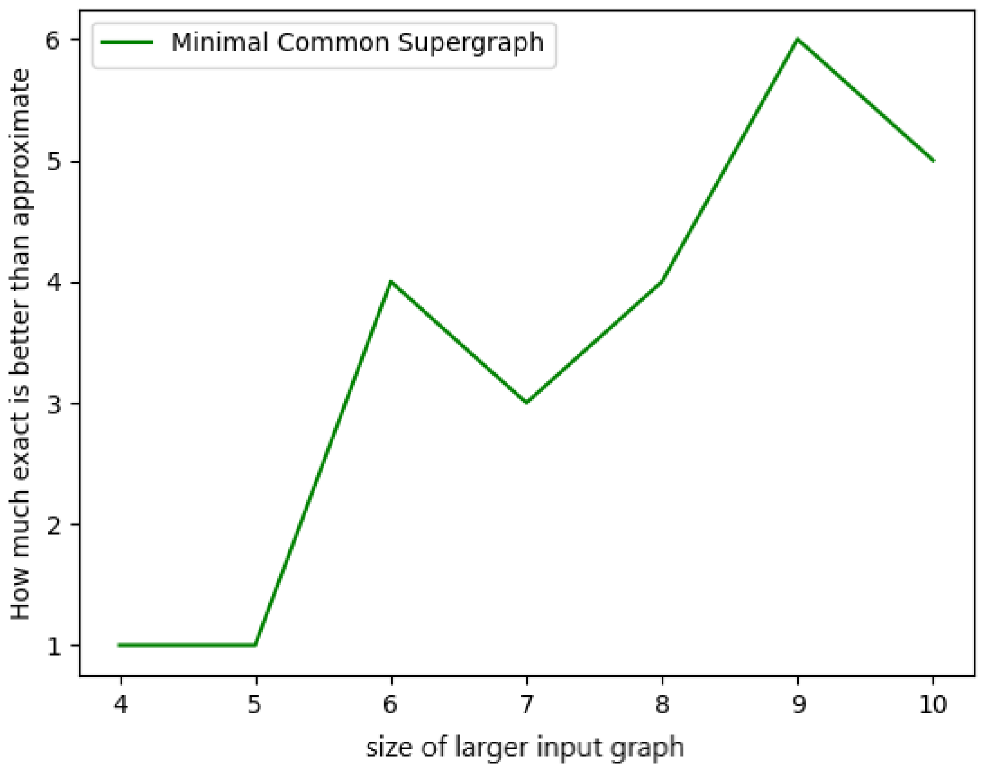




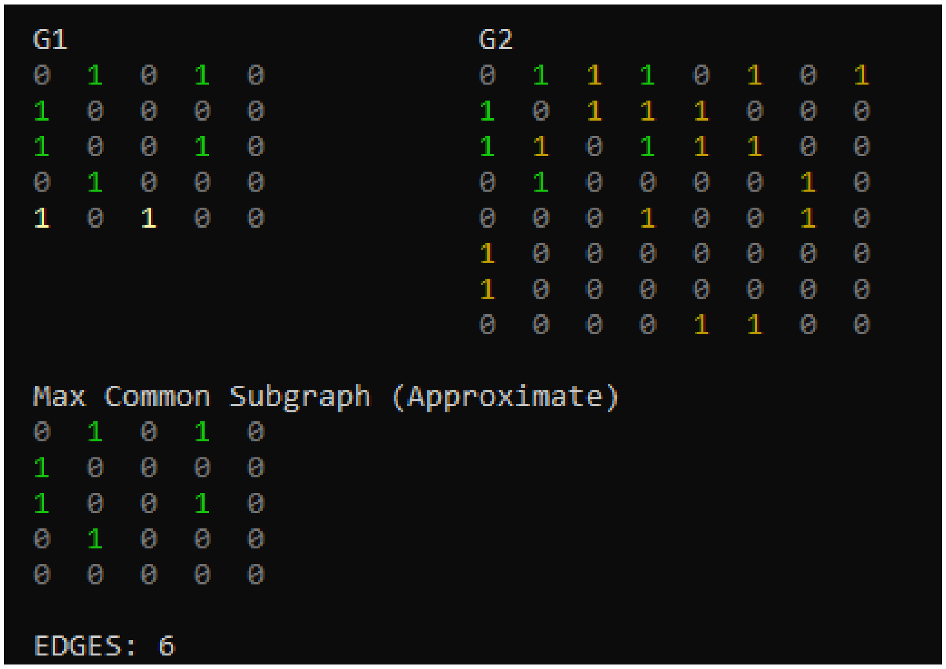
Again, we can see that execution times of the exact algorithms are very similar. This is the result we expected, because the time complexity of both algorithms is almost the same and the plots clearly illustrate the exponential growth of their performance. The time of execution reaches around 3 minutes for graphs with only 10 vertices, so we can easily deduce it would be impossible to run those algorithms for larger graphs on a regular computer.

8.3 Comparison of approximate and exact algorithms

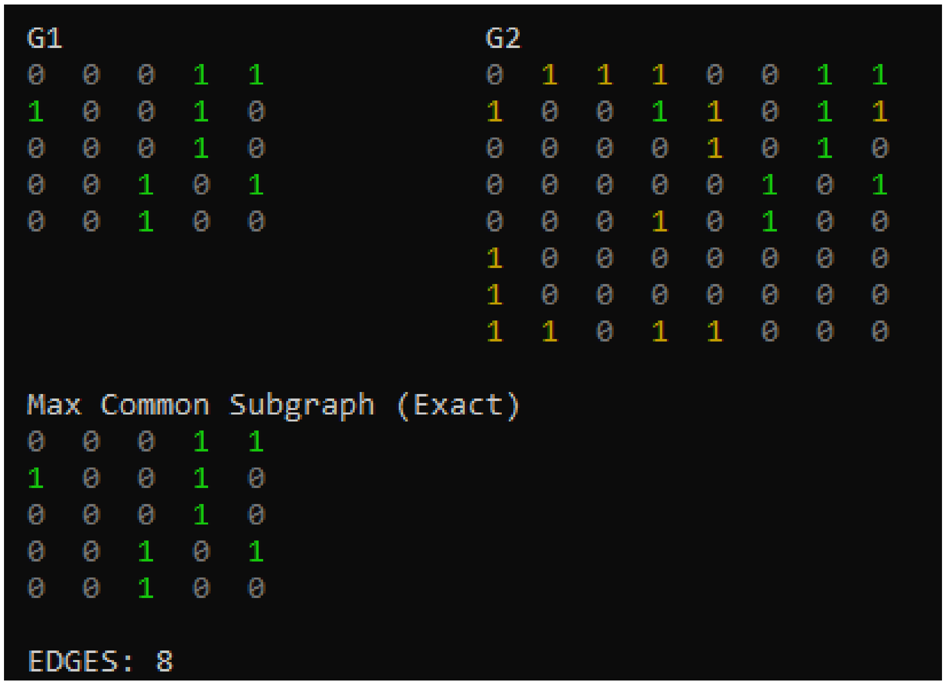




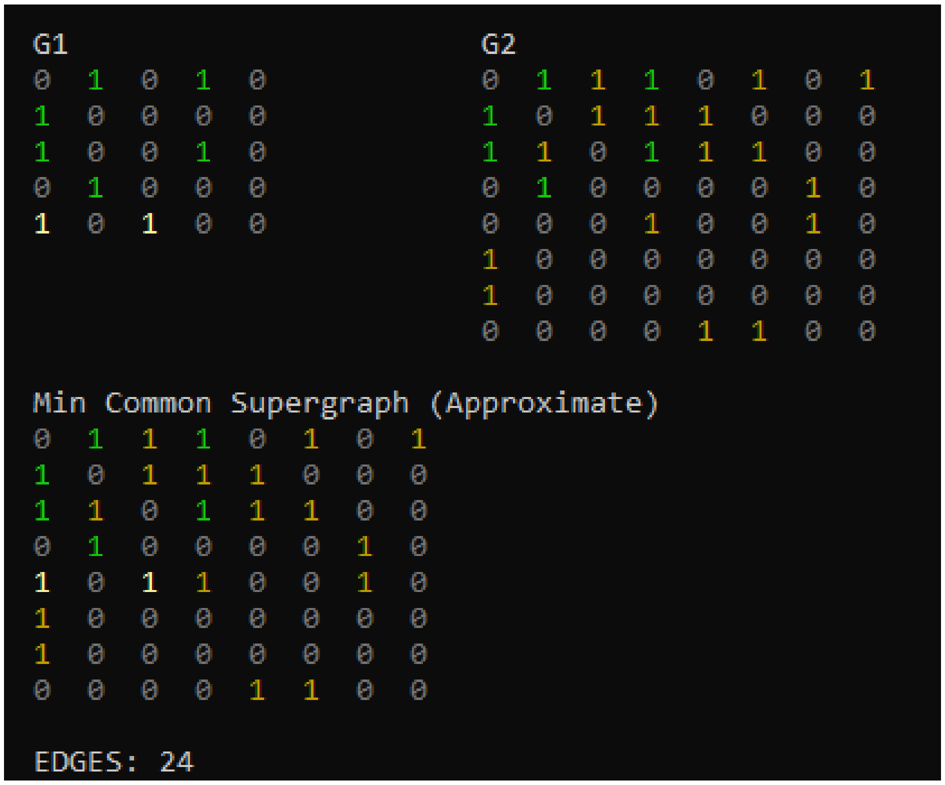
The above diagrams present the difference between number of edges of the graph resulting from the approximate algorithm and the number of edges resulting from the exact algorithm. We can easily see that as the size of input graphs increases, the accuracy approximate algorithm is getting worse. But the most interesting fact is that those two diagrams are identical. We checked both algorithms on many examples and after taking any two random graphs, the difference between the number of edges found by exact algorithm and approximate algorithm for the maximal common subgraph problem was always the same as the difference between number of edges for exact and approximate solutions for minimal common supergraph. Below we present one of the examples from the console output:



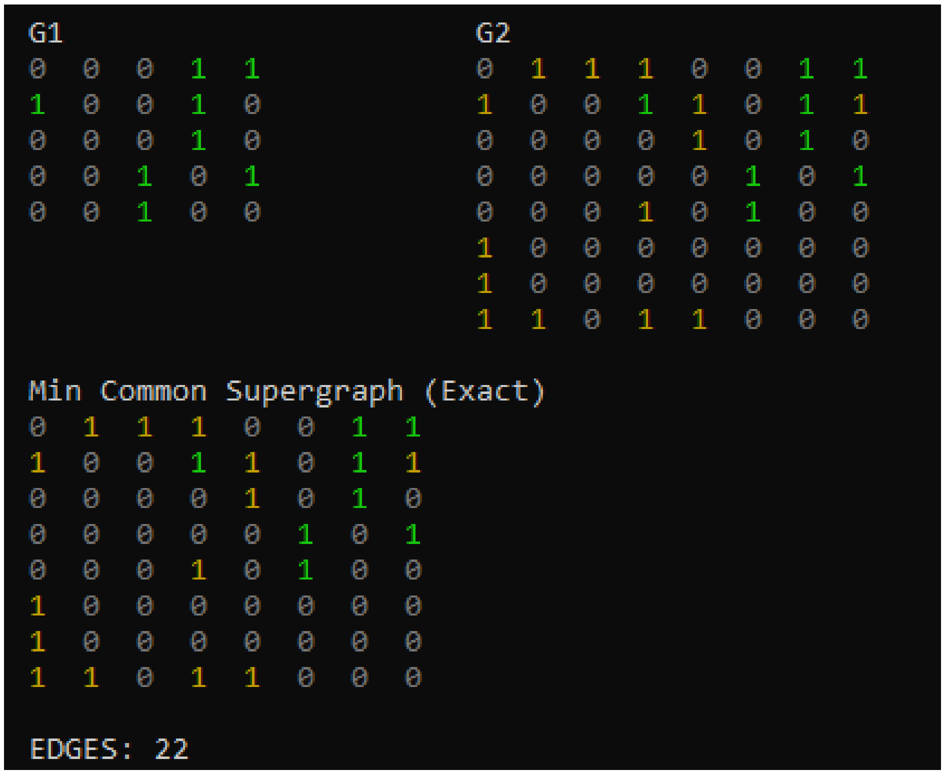
```
G1                                    G2
0   1   0   1   0                     0   1   1   1   0   1   0   1
1   0   0   0   0                     1   0   1   1   1   0   0   0
1   0   0   1   0                     1   1   0   1   1   1   0   0
0   1   0   0   0                     0   1   0   0   0   0   1   0
1   0   1   0   0                     0   0   0   1   0   0   1   0
                                      1   0   0   0   0   0   0   0
                                      1   0   0   0   0   0   0   0
                                      0   0   0   0   1   1   0   0

Max Common Subgraph (Approximate)
0   1   0   1   0
1   0   0   0   0
1   0   0   1   0
0   1   0   0   0
0   0   0   0   0

EDGES: 6
```



```
G1                                    G2
0   0   0   1   1                     0   1   1   1   0   0   1   1
1   0   0   1   0                     1   0   0   1   1   0   1   1
0   0   0   1   0                     0   0   0   0   1   0   1   0
0   0   1   0   1                     0   0   0   0   0   1   0   1
0   0   1   0   0                     0   0   0   1   0   1   0   0
                                      1   0   0   0   0   0   0   0
                                      1   0   0   0   0   0   0   0
                                      1   1   0   1   1   0   0   0

Max Common Subgraph (Exact)
0   0   0   1   1
1   0   0   1   0
0   0   0   1   0
0   0   1   0   1
0   0   1   0   0

EDGES: 8
```



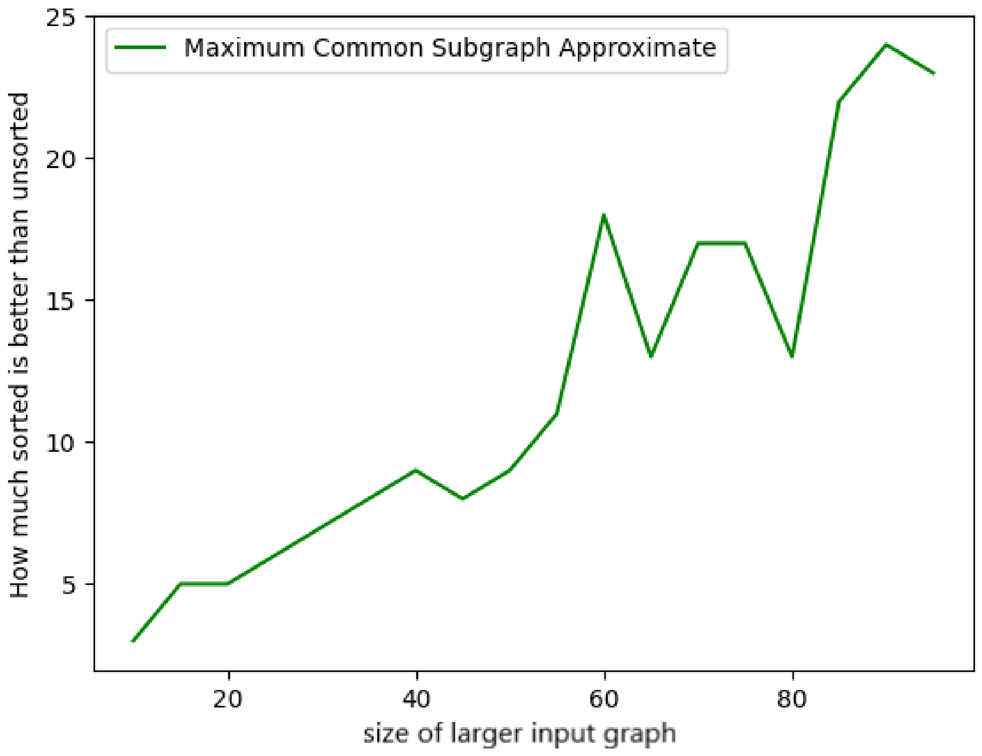
```
G1                  G2
0  1  0  1  0       0  1  1  1  0  1  0  1
1  0  0  0  0       1  0  1  1  1  0  0  0
1  0  0  1  0       1  1  0  1  1  1  0  0
0  1  0  0  0       0  1  0  0  0  0  1  0
1  0  1  0  0       0  0  0  1  0  0  1  0
                    1  0  0  0  0  0  0  0
                    1  0  0  0  0  0  0  0
                    0  0  0  0  1  1  0  0

Min Common Supergraph (Approximate)
0  1  1  1  0  1  0  1
1  0  1  1  1  0  0  0
1  1  0  1  1  1  0  0
0  1  0  0  0  0  1  0
1  0  1  1  0  0  1  0
1  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0
0  0  0  0  1  1  0  0

EDGES: 24
```

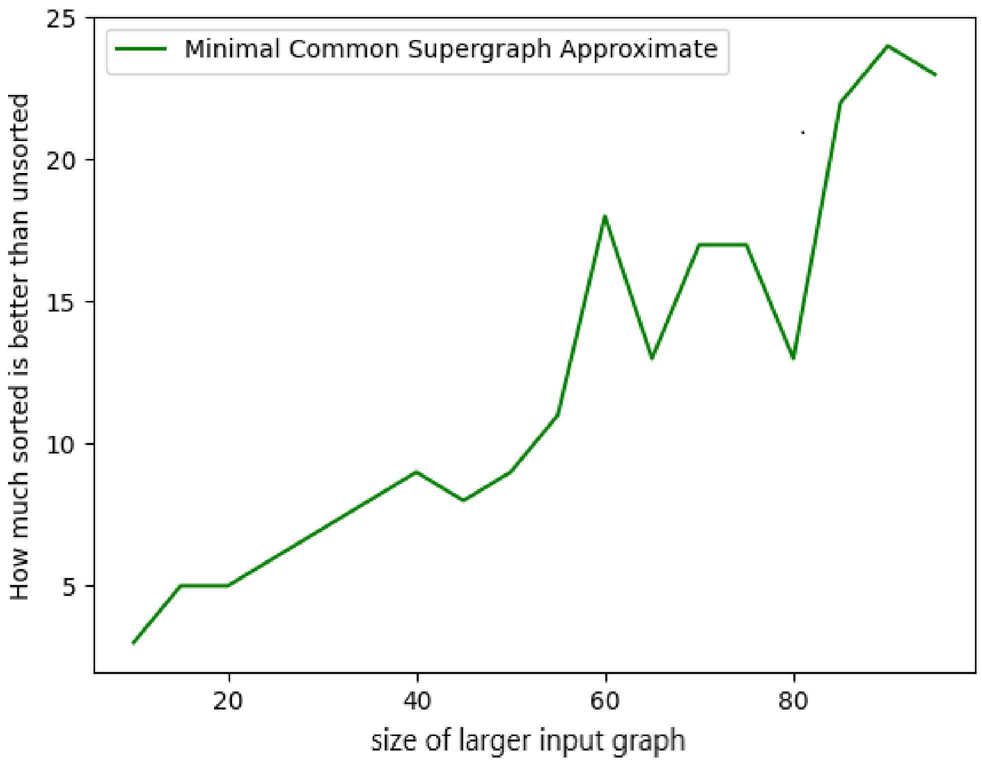


```
G1                  G2
0  0  0  1  1       0  1  1  1  0  0  1  1
1  0  0  1  0       1  0  0  1  1  0  1  1
0  0  0  1  0       0  0  0  0  1  0  1  0
0  0  1  0  1       0  0  0  0  0  1  0  1
0  0  1  0  0       0  0  0  1  0  1  0  0
                    1  0  0  0  0  0  0  0
                    1  0  0  0  0  0  0  0
                    1  1  0  1  1  0  0  0

Min Common Supergraph (Exact)
0  1  1  1  0  0  1  1
1  0  0  1  1  0  1  1
0  0  0  0  1  0  1  0
0  0  0  0  0  1  0  1
0  0  0  1  0  1  0  0
1  0  0  0  0  0  0  0
1  0  0  0  0  0  0  0
1  1  0  1  1  0  0  0

EDGES: 22
```

It is apparent that in both cases the difference in the number of edges between exact and approximate solution is exactly two, which is the same. This just a single example, but the situation was identical for every 2 graph we tested, which is also proved by the plots at the beginning of this section.

8.4 Comparison of approximate algorithms with and without sorted form





On this plots we can observe the difference between the number of edges in graph found by approximate algorithms which sort the input graph and the number of edges in graph found by approximate algorithms which do not sort the input graphs. It is apparent that the larger the input graph, the bigger effect the sorting has. In case of minimal common supergraph for graphs of 80 vertices the sorting version of algorithm found graphs with around 15 edges less than algorithm without sorting and in case of maximal common subgraph algorithm that is sorting the matrices found around 15 edges more than algorithm that do not sort the graphs. Which means that sorting the matrices makes the approximation much better especially for bigger graphs. It is worth remembering that using sorting in approximation algorithms does not affect the complexity. But the most interesting thing is that difference between algorithm which uses sorting and the one which does not is the same for both algorithms. Hence we have the same situation as for the difference in performance between the exact and approximate solutions.

8.5 Conclusions

From the presented observations we can conclude that finding exact solutions is very time consuming and almost impossible for very large graphs (at least with current technology). Even for small graphs with only 10 vertices it takes around 3 minutes to calculate the solution on a regular computer, so for larger graphs we have to approximate the solutions. To obtain a better approximation we decided to transform graphs into sorted versions. This procedure doesn't affect the complexity of the algorithm, but it allows us get slightly better results. In our test we discovered that the improvement gets better as size of the graphs to which we apply algorithms gets larger.

9 Instruction

This project was implemented using *C#* programming language in Visual Studio, which is able to create an executable file with the program, so no additional software or environment is needed to run it. It is enough to double-click the executable in the provided EXE folder.
(NOTE: The executable was created in the 64-bit version to allow for more memory usage, thus it will not work on the 32-bit machines - in such case the appropriate executable should be compiled in Visual Studio after changing the compiler in project settings).
The program opens the console in which all steps are preceded by appropriate descriptions, thus the user should be able to use the software without problems. Commands to provide the input (for example specifying size of graphs to generate) should be accepted by pressing "Enter" key, but the rest of options runs immediately after choosing appropriate key.
At the beginning, the user can choose to run the algorithms on random graphs generated by the program or to provide their own input graphs from a file. In case of choosing the file input the user should type in the absolute path of the file (i.e. "C:/Example/test.txt"). The content of the file should have the following format:

```
5
0 1 1 0 0
0 0 1 0 1
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
4
0 1 1 0
0 0 0 0
0 1 0 0
0 1 0 0
```

Where:

1) first line contains the number of vertices (i.e. the size) of the first graph, let's say n.

2) lines second to $2 + n - 1$ contain the respective rows of first matrix, where each row is a sequence of 0's and 1's and each two digits in a row are separated by a single white space.

3) line $2 + n$ contains the number of vertices of the second graph, let's say k.

4) lines $2 + n + 1$ to $2 + n + k$ contain the respective rows of the second matrix, where each row is a sequence of 0's and 1's and each two digits in the row are separated by a single white space.

In case of choosing the random generation of graphs, the user is simply asked to specify the size of both input matrices - if at least one of the provided sizes will be more than 10, then only the approximate versions of the algorithms will be available. This is because the exact algorithms have exponential time complexity and even for graphs with only 10 vertices it takes around 3 minutes to find the solution.

NOTE: The "Source" folder contains a folder "AlgorithmsComputabilityProject" with all source files and code, and an .sln file which enables to run the solution in the Visual Studio IDE. If it is already installed on the computer, then a double-click on the .sln file would automatically launch the Visual Studio and enable to view all the source files in one place or even compile the project again.

10 Version list

Date	**Version**	**Comment**
28.10.2020	1.0	Sections 1-7 without complexity analysis
04.11.2020	1.1	Complexity analysis of the solution
23.11.2020	1.2	Section 8 - Results
24.11.2020	1.3	Conclusions and instructions
25.11.2020	1.4	Correction and version list