Faculty of Mathematics and Information Science
WARSAW UNIVERSITY OF TECHNOLOGY

Algorithms and computability project

Directed graph algorithms

Nikodem Olszowy
nikodem.olszowy.stud@pw.edu.pl

Filip Rak
filip.rak2.stud@pw.edu.pl

Mateusz Sudejko
mateusz.sudejko.stud@pw.edu.pl

Piotr Możeluk
piotr.mozeluk.stud@pw.edu.pl

Warsaw, 25th October 2023

Contents

1 Introduction

This document is a documentation for the project for Algorithms and computability. THe document consists of definitions, algotihms and the analysis of problems related to undirected graphs. The scope of our projects covers the problem of how to represent undirected grapphs, how to find some basic properties of such graph (i.e. size of the graph) and how to find both maximum clique for a graph and maximum common subgraph of two graphs.

2 Definitions

The purpose of the definitions section is to introduce more complicated and not obvious terms that needs explanation before they will be used in the documentation. It is obligatory to know all the definitions from this section to continue documentation reading.

2.1 Size of a graph

We have chosen to define the size of a graph as a number of vertices. For example to use Dijkstra's algorithm we would need to save the graph as a matrix and the size of it is going to be dependent on the number of vertices.

2.2 Degree of a vertex

The degree of the vertex v in graph G is a number of vertices in G that are neighbors of v which is the number of vertices that are connected to v and are in G.

2.3 k-degenerate graph

The k-degenerate graph is such a graph G that for all subgraphs of G the maximal degree in each of them is at most k which means that for any vertex in any subgraph of G its vertices have a degree smaller or equal to k [1].

2.4 Degeneracy

The degeneracy of the graph G is the smallest value k for which after removing k vertices the graphs is k-degenerated graph [1].

2.5 Clique in a graph

A clique of the given graph G is the a complete subgraph of this graph. Usually when we are referring to the cliques we are interested in a maximum clique which is a special subgraph that also has a property that it can't be extended by any additional vertex so that it is still a complete graph. In simple words the maximum clique is a clique of the graph that is not a subgraph of any other clique. Below there is a figure that visualises what cliques are [2].

2.6 NP-hard

An NP-hard problem is a computational problem that is at least as hard as the hardest problems in NP (nondeterministic polynomial time).

2.7 Heurestics

Heurestics are techniques used to find a solution quicker when classic methods are to slow. This involves algorithm randomization where instead of looking for something with given property we select element randomly to increase the speed. Then we can run such algorithms with given heurestics many times in order to obtain better results [3].

3 Reasonable metrics

3.1 Connectivity

Graphs can be categorized based on their connectivity. Metrics like the number of connected components, the size of the largest connected component, or measures like the graph's edge connectivity and vertex connectivity can be used to assess how connected a graph is [4]. In order to find out if a graph is connected or disconnected we can use a method described in below instructions:

1. Begin at any arbitrary node of the graph.
2. Proceed from that node using either depth-first or breadth-first search, counting all nodes reached.
3. Once the graph has been entirely traversed, if the number of nodes counted is equal to the number of nodes of G, the graph is connected; otherwise it is disconnected.

3.2 Diameter

The diameter of a graph is the longest shortest path between any two nodes. It provides insights into the "size" of the graph and can be a critical metric for understanding network efficiency or how quickly information can spread through a network. In order to obtain this metric we can use the Dijkstra's algorithm in combination with a binary heap to obtain the shortest-path tree [5]. This implementation has logarithmic time complexity and can be achieved as follows:

1. Create a Min Heap of size V (size defined in section 2.1). Every node of the min-heap contains the vertex number and distance value of the vertex.
2. Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is infinite.
3. While Min Heap is not empty, do the following :
   (a) Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.
   (b) For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and the distance value is more than the weight of u-v plus the distance value of u, then update the distance value of v.

3.3 Planarity

Determining whether a graph is planar (can be embedded in the plane without edge crossings) or non-planar is another important property. There are many methods to obtain this metric, some of the most notable include path, vertex and edge addition methods or the construction sequence method [6]. A linear complexity example would be the Hopcroft and Tarjan path addition method:

1. Count the number of edges, if |E|> 3 |V|- 6 then it is nonplanar
2. Apply Depth-first search, converting the graph into a palm tree T and numbering the vertices.
3. Find a cycle in the tree and deletes it, leaving a set of disconnected pieces
4. Check the planarity of each piece plus the original cycle
5. Determines whether the embeddings of the pieces can be combined to give an embedding of the entire graph.

3.4 Graph Spectrum

The eigenvalues of the adjacency matrix or Laplacian matrix of a graph form its spectrum. Spectral graph theory provides insights into graph properties. The most popular method to get the Laplacian is Laplacian via the incidence matrix method. A symmetric laplacian would be defined as follows [7]:
$L^{\text{sym}} := (D^+)^{1/2} L (D^+)^{1/2} = I - (D^+)^{1/2} A (D^+)^{1/2},$
Where D^+ is the Moore-Penrose inverse

3.5 Graph Colorability

Understanding how many colors are needed to color the vertices of a graph without adjacent vertices sharing the same color (chromatic number) is another important metric. Calculating the chromatic index is computationally hard. In fact all known algorithms are NP-complete (nondeterministic polynomial-time complete). Some of the examples include DSatur and recursive largest first (RLF) algorithms, additionaly it is not bad practice to use brute-force algorithms in this scenario. The instructions for the DSatur [8] algorithm are as follows:

1. Let v be the uncolored vertex in G with the highest degree of saturation. In cases of ties, choose the vertex among these with the largest degree in the subgraph induced by the uncolored vertices.
2. Assign v to the lowest color label not being used by any of its neighbors.
3. If all vertices have been colored, then end; otherwise return to Step 1.

4 Maximum clique in a graph

4.1 Bron-Kerbosch algorithm - Introduction

In 1973 Coenraad Bron and Joep Kerbosch published a paper in which they described an algorithm that is used to find a maximum clique in a given graph G. In this section the algorithm, its modifications, pseudocode and time and space complexity will be explained. All needed definitions can

be found in a definitions section so it is advised to read it before getting into other parts when all data mentioned above will be explained.

4.2 Bron-Kerbosch algorithm - Explanation

The basic algorithm consists of few steps. The base idea behind the algorithm is to use recursive backtracking which in simple words is a way of finding all possible cliques and selecting the one which is the highest using recursion. In the algorithm we will use three sets of vertices labeled as R, P and X. R is a set that will be storing vertices that belong to the clique. The set P and X at each stage will consist of vertices that are neighbors of vertices in R and were not considered yet in R. This means that two sets P and X are controlling the backtracking process. If there will be no more vertices in P and there will be at least one vertex in X then the algorithm is backtracking. If both X and P will be empty this means that in R we have vertices of the valid clique. In this case if the size of set R is the biggest number it will be stored and at the end we will have a number that represents the size of the biggest clique. Even though sets X and P holds at certain points neighbors of R it is easy to notice when going over simple example at some point we will encounter situations when we will have vertices in the set R, no vertices in P and some vertices in X. This is a mechanism of backtracking and the purpose of this is to avoid returning sets of cliques that consist of same vertices. Since we have a general idea on how the algorithm works it is time to prepare a pseudo code for finding the clique with the biggest size. Below there is a pseudocode and an example of how the algorithm behaves for a simple graph with 5 vertices (graph is also below and the example shows how the sets looks like at each stage.

```
algorithm bronKerbosch(max,R,P,X):
        if P = ∅ and X = ∅ then:
            maximum(max,sizeof(R))
        else
            for every vertex v in P:
                bronKerbosch(max, R∪{v}, P ∩ neigh(v), X ∩ neigh(v))
                P = P \ {v}
                X = X ∪ {v}
```

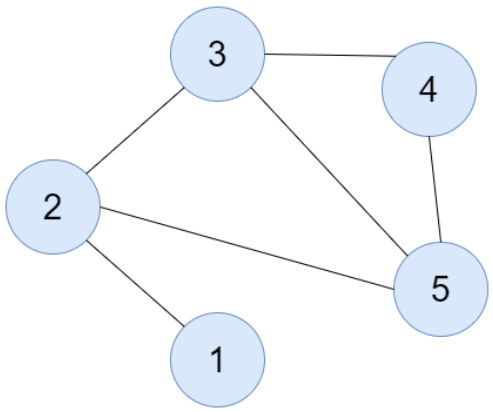

Figure 1: Graph for an example

The methods and the set at each point of the algorithm will look like this: We start with set R = ∅ and P = { 1,2,3,4,5,6} and X = ∅. Now we can start executing the algorithm with these sets as input and max equal to 0:

bronKerbosch(0,∅, {1,2,3,4,5,6}, ∅)
    bronKerbosch(0,{1}, {2}, ∅)
        bronKerbosch(0, {1,2}, ∅, ∅)
        maximum(0,2)
P = {2,3,4,5} and X = {1}
bronKerbosch(2,∅,{2,3,4,5},{1})
    bronKerbosch(2, {2}, {3,5}, {1})
        bronKerbosch(2, {2,3}, {5}, ∅)
            bronKerbosch(2,{2,3,5}, ∅, ∅)
            maximum(2,3)
        P = {5} and X = {3}
        bronKerbosch(3, {2}, {5}, {3})
            bronKerbosch(3, {2,5}, ∅, {3})
            backtracking since P is an empty set and X is a non-empty set
P = {3,4,5} and X = {1,2}
bronKerbosch(3,∅, {3,4,5},{1,2})
    bronKerbosch(3, {3}, {4,5}, {2})
        bronKerbosch(3, {3,4}, {5}, ∅)
            bronKerbosch(3,{3,4,5}, ∅, ∅)
            maximum(3,3)
        P = {5} and X = {4}
        bronKerbosch(3, {3}, {5}, {4})
            bronKerbosch(3, {3,5}, ∅, {4})
            backtracking since P is an empty set and X is a non-empty set
P = {4,5} and X = {1,2,3}
bronKerbosch(3,∅, {4,5},{1,2,3})
    bronKerbosch(3, {4}, {5}, {3})
        bronKerbosch(3, {4,5}, ∅, {3})
        backtracking since P is an empty set and X is a non-empty set

As we can see there is a lot of steps. In next section we will cover how we can improve the algorithm to obtain the one that operate faster.

4.3 Bron-Kerbosch algorithm - Improvement

As we saw in previous section the algorithm goes through initial vertices at first iteration. This is the moment where we can actually improve the algorithm since in each step we usually have a vertex that suits the best. In this case instead of iterating over all vertices before entering for loop we select a vertex with highest degree and then the set over which we will be iterating is a set consisting of the selected vertex and the vertices that are not neighbors of the selected vertex since in the highest clique that we are building using recursion. This means that the pseudocode looks like this:

```
algorithm bronKerbosch(max,R,P,X):
        if P = ∅ and X = ∅ then:
            maximum(max,sizeof(R))
        else
            select vertex w as pivot point
            for every vertex v in P  neigh(w):
                bronKerbosch(max, R∪{v}, P ∩ neigh(v), X ∩ neigh(v))
                P = P \ {v}
                X = X ∪ {v}
```

Now when we run the same graph but for the algorithm with a pivot point the execution will looks like this: bronKerbosch(0,∅, {1,2,3,4,5}, ∅)

```
w = pivot({1,2,3,4,5})
bronKerbosch(0,{3}, {2,4,5}, ∅)
w = pivot({2,4,5})
    bronKerbosch(0,{2,3}, {5}. ∅)
    w = pivot({5})
        bronKerbosch(0,{2,3,5}, ∅, ∅)
        maximum(0,3)
    bronKerbosch(3, {3,4}, {5}, ∅)
    w = pivot({5})
        bronKerbosch(0,{3,4,5}, ∅, ∅)
        maximum(3,3)
P = {1,2,4,5}
X = {3}
bronKerbosch(3, {1}, {2}, ∅)
w = pivot({2})
    bronKerbosch(3, {1,2}, ∅, ∅)
    maximum(3,2)
P = {2,4,5}
X = {1,3}
bronKerbosch(3, {5}, {2,4}, {3})
w = pivot({2,4}             bronKerbosch(3,{2,5}, ∅, {3})
        backtracking since P is an empty set and X is a non-empty set
    bronKerbosch(3, {4,5}, ∅, {3})
        backtracking since P is an empty set and X is a non-empty set
```

As you can see the code has much less lines of code to execute, This happens when the graph is not that sparse. However the time complexity for worst case scenario is the same. Yet in average the time execution will be faster. The time complexity of the algorithms of our choice will be covered later, but for now all we need to know that the time complexity will be still exponential. The goal of this project is to finish with the algorithms that will have the polynomial complexity. THis means that to make our algorithms faster in case of time complexity we will achieve the simplification by reducing the precision of our outcomes. Next section covers the greedy algorithm with optimization.

4.4 Greedy algorithm with optimization - Introduction and explanation

The greedy algorithm is a simple algorithm to find a potential maximal clique of the given graph G. We begin with selection of the initial vertex and we add it to the set of the maximal clique. Then we look for next vertex from the set of the neighbors of the selected vertex. Then we do it as long as the set of neighbors is not empty. If the set becomes empty then we compare the size of clique with the current maximal size (initialized with 0). Then we remove all vertices from the clique from the original graphs. We also remove all edges where we can find the vertex from the clique set [10]. In this algorithm we do not have precisely defined which vertex we should choose. We can make use of the heurestics. In this case we will randomize the selection of the vertex at any point. For better precision we can repeat the algorithm since in this algorithm for different vertex selection we can obtain different results, but all results will be valid cliques so this means that we can take maximum out of all results that we obtained.

4.5 Greedy algorithm - pseudocode and an example

Below there is a pseudocode for the algorithm explained in the previous section. The input is the current maximum clique and the set of vertices of given graph G denoted as P:

```
algorithm greedyRandom(max,P):
    R = ∅
    while P is not empty:
        N = P
        while N is not empty:
            v = random(P)
            N = N ∩ neigh(v)
            R = R ∪ {v}
        maximum(max,sizeof(R)
        P = P \ R
        R = ∅
```

Now we will go over the execution of the algorithm described for the same graph for which we did execution analysis in the section regarding bron-Kerbosch graph:

R = $\emptyset$
N = {1.2.3,4,5}
v = 3
N = {2,4,5}
R = {3.}
v = 4
N = {5}
R = {3,4}
v = 5
N = $\emptyset$
R = {3.4.5}
maximum(0,3)
R = $\emptyset$
P = {1,2}
v = {1}
N = {2}
R = {1}
v = {2}
N = $\emptyset$
R = {1,2}
maximum(3,2)
P = $\emptyset$
R = $\emptyset$

4.6 Bron-Kerbosch with pivot - Time Complexity

Our algorithm in general consists of two aspects when it comes to the recursion process. First aspect is the number of recursive calls and the second is the depth of each recursive call. The number of recursive calls is related to the size of the largest maximal clique. In worst case for k vertices it will be of size k. Moreover, for each vertex there are two recursive calls performed. One is adding vertex to the set of clique and the other where it is excluded from further consideration. This means that the number of calls is equal in worst case scenario to 2^k. When it comes to the depth of the recursion it is determined by the maximal clique only so in this case it will be just k. This leads us to the conclusion that the complexity of this algorithm is $O(2^k)$.

4.7 Greedy algorithm - Time complexity

Greedy algorithm consists of two parts. First we select one vertex from the graph and we find a clique for this vertex. Then we are deleting these vertices from the graph. In worst case we will have to walk through some part of vertices and for each vertex we will have to find all its neighbors. This means that the time complexity of such algorithm is $O(k^2$ where k is the number of vertices. We can notice that we managed to find an algorithm that will approximate the maximal clique problem solution with polynomial time complexity. To get better results we should run the algorithm multiple times to increase the chance of finding best vertex ordering.

5 Testing Performance

5.1 Introduction

In this part we are going to be checking the performance of our Matlab implementation for Bron-Kerbosch and GreedyRandom algorithms. First of all we need a script generating random graphs with different adjustments (number of vertices and edges). To get convincing results we have divided graphs into some categories. First category is number of vertices, we chose to test graphs of sizes 40–50 and 90–100. Second category is number of edges, we chose to test graphs such that there is 75, 50 or 25 percent chance that there is a connection between two vertices. This gives us 6 groups to test the average time to execute function BronKerbosch, GreedyRandom and the average error of GreedyRandom algorithm. Another adjustment that we have checked is how the number of repeated executions of GreedyRandom algorithm affects the time and precision as when we repeat the algorithm we take the greatest value obtained for the result.

5.2 Method

To calculate time execution of the function we have used the most common Matlab function for that named 'Tic' and 'Toc'. We have placed these two so that they calculate the execution of the wanted function in seconds and then saved it to an array of all results. Afterwards we are able to get the mean of the results. The same goes for the precision calculation. We store the errors of GreedyRandom algorithm in an array and calculate mean in the end. Since BronKerbosch always gets greater or equal result than GreedyRandom we don't need to calculate absolute error. For the tests we generated 100 random graphs in every category.

5.3 Results

100 Graphs	BronKerbosch time (mean)	GreedyRandom time (mean) (50/10 reps)	Average error
(40–50) 75%	1.1204	0.2362/0.0425	2%/5%
(90–100) 75%	63.3364	1.1495/0.3105	5%/11%
(40–50) 50%	0.0862	0.5225/0.0363	0%/1%
(90–100) 50%	0.6671	1.9835/0.1493	1%/5%
(40–50) 25%	0.0074	0.1604/0.0321	0%/0%
(90–100) 25%	0.0346	0.6169/0.1271	0%/0%

5.4 Conclusion

Summarising, we can see that as the number of edges and vertices grows the time execution grows with it, which is not surprising. On the other hand we can see that greater number of edges affects the time performance of BronKerbosch algorithm, but GreedyRandom algorithm is affected in much less drastic way making it much better for bigger graphs.

6 Maximum common subgraph

A maximum common subgraph (MCS) is a graph that is simultaneously isomorphic to two subgraphs of two given graphs. There are two variants of the MCS problem: the maximum common induced subgraph (MCIS) problem and the maximum common edge subgraph (MCES) problem. In

the MCIS problem, the goal is to find a subgraph with as many vertices as possible that preserves both the edges and vertices structure of the original graphs. In the MCES problem, the goal is to find a graph with as many edges as possible that preserves only the edges of the original graphs. The MCS problem is classified as NP-hard and remains challenging computationally. In this paper, we will focus on the maximum common induced subgraph problem.

6.1 Algorithm based on clique detection

The algorithm was discovered by Levi[14], Borrow, and Burstall[15] who realized that maximal clique could be used to find the MCIS. They found out that by using the modular product of two graphs G_1, G_2 and then finding the maximum clique for the newly produced graph they were able to locate the MCIS.
The modular product of two graphs, sometimes referred to as a compatibility graph, is a graph with the set of verites$V_1 \times V_2$ of graphs G_1 and G_2, where the set of edges of G_1 is denoted as $E(G_1)$, with edges represented by a pair of vertices (u_i, u_j), and analogously for G_2 with $E(G_2)$ and (v_i, v_j), then an edge between two vertices exists in the modular product if $(u_i, u_j) \in E(G_1)$ and $(v_i, v_j) \in E(G_2)$ or if $(u_i, u_j) \notin E(G_1)$ and $(v_i, v_j) \notin E(G_2)$.[16]

6.2 Algorithm based on clique detection - Pseudocode

```
function [adjacencyMatrix, indexMapping] = modularProduct(matrix1,
   matrix2)
    nextVertex = 1
    association = emptyMap()
    edges = emptyList()

    indexMapping = emptyList()

    for v1 = 1 to size(matrix1, 1)
        for v2 = 1 to size(matrix2, 1)
            for w1 = 1 to size(matrix1, 1)
                for w2 = 1 to size(matrix2, 1)
                    if v1  w1 AND v2  w2 AND matrix1(v1, w1) == matrix2(v2,
                       w2) AND matrix1(w1, v1) == matrix2(w2, v2)
                        if NOT keyExists(association, [v1, v2])
                            association(key1) = nextVertex
                            indexMapping[nextVertex] = [v1, v2]
                            nextVertex = nextVertex + 1
                        end

                        if NOT keyExists(association, [w1, w2])
                            association(key2) = nextVertex
                            indexMapping[nextVertex] = [w1, w2]
                            nextVertex = nextVertex + 1
                        end
```

```
                    edges.add([association([v1, v2]), association[w1, w2
                        ]])
                end
            end
        end
    end

    for i = 1 to size(edges)
        adjacencyMatrix(edges[i][1], edges[i][2]) = 1
    end

    return adjacencyMatrix, indexMapping
end


function output = maximumCommonSubgraph(graph1, graph2)
    % graph1, graph2: Adjacency matrices of the input graphs
    [modular, indexMapping] = modularProduct(graph1, graph2);
    allCliques = BronKerboschDirected(modular);
    maxClique = maximum_clique(allCliques);
    indexMappingGraph1 = [];

    for i = 1 to length(maxClique)
        indexMappingGraph1 = concatenateLists(indexMappingGraph1,
            indexMapping{maxClique[i]}[1])
    end
    for i = 1 to size(graph1, 1)
        for j = 1 to size(graph1)
            if isMember(i, indexMappingGraph1) AND isMember(j,
                indexMappingGraph1)
                output(i, j) = graph1(i, j);
            end
        end
    end

    isolatedVertices = sum(output, 1) == 0 AND sum(output, 2)' == 0;
    output = submatrix(output, ~isolatedVertices, ~isolatedVertices);
end
```

The above pseudocode presents the method described above to find the maximum common induced isomorphic subgraph.
The function modularProduct iterates through all cells of matrix1 and matrix2 and checks if the condition for the modular graph holds. If it holds then we add the edge to the list of edges which at the end are placed into the adjacency matrix. The association map protects from creating the same vertex more than one time. Introducing the indexMapping is essential because input graphs are directed so every newly created vertex would lose track of the original enumeration which would cause the false output to appear at the end of the MCS function.
the MCS function creates the modular graph and then finds the maximum_clique for this graph. At

the end, we are using the indexMaping to extract the maximum common subgraph from the input graph. Keeping track of indexes assures that direction in directed graphs is preserved.
The complexity of this code is $O(r_1^2 * r_2^2 + 3^{\frac{n}{3}})$ where $n = max(r_1, r_2)$ and r_i is number of rows of adjacency matrix representing graphs G_1 and G_2 respectively. This algorithm could run with "different complexity" by substituting the Bron-Kerbosch algorithm with Tsukiyama[17] however this algorithm is input-sensitive so when the number of cliques is low it could perform better but the complexity of this algorithm (the worst case) is the same.
The polynomial approximation can be obtained by substituting the Bron-Kerbosch algorithm with a greedy algorithm which was described above then the complexity would be $O(r_1^2 * r_2^2) + max(r_1, r_2)^2$.

7 Bibliography

References

[1] Degeneracy of the graph
https://en.wikipedia.org/wiki/Degeneracy_(graph_theory)

[2] Maximal Clique of the graph
https://www.sciencedirect.com/topics/mathematics/maximal-clique

[3] Heurestic
https://en.wikipedia.org/wiki/Heuristic_(computer_science)

[4] Graph connectivity
https://en.wikipedia.org/wiki/Connectivity_(graph_theory)

[5] Binary heap Dijkstra Algorithm
https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo–8/

[6] Graph planarity testing
https://en.wikipedia.org/wiki/Planarity_testing

[7] Symmetric Laplacian via Incidence Matrix
https://en.wikipedia.org/wiki/Laplacian_matrix#Symmetric_Laplacian_via_the_incidence_matrix

[8] The DSatur Algorithm
https://en.wikipedia.org/wiki/Graph_coloring#Heuristic_algorithms

[9] A Partitioning Algorithm for Maximum Common Subgraph Problems.
McCreesh Ciaran, Prosser Patrick, Trimble James. (2017).

[10] A greedy algorithm for maximum clique
https://iq.opengenus.org/greedy-approach-to-find-single-maximal-clique/

[11] https://rajsain.files.wordpress.com/2013/11/randomized-algorithms-motwani-and-raghavan.pdf

[12] Kann, Viggo (1992), "On the approximability of the maximum common subgraph problem", STACS 92: 9th Annual Symposium on Theoretical Aspects of Computer Science Cachan, France, February 13–15, 1992, Proceedings, Lecture Notes in Computer Science, vol. 577, Springer Science Business Media, pp. 375–388

[13] Zuckerman, D. (2006), "Linear degree extractors and the inapproximability of max clique and chromatic number", Proc. 38th ACM Symp. Theory of Computing, pp. 681–690

[14] G. Levi, A note on the derivation of maximal common subgraphs of two directed or undirected graphs, CALCOLO 9 (1973) 341–352.

[15] H. G. Barrow, R. M. Burstall, Subgraph isomorphism, matching relational structures and maximal cliques, Inf. Proc. Lett. 4 (1976) 83–84.

[16] Duesbury, E., Holliday, J.D. and Willett, P. orcid.org/0000–0003–4591–7173 (2017) Maximum Common Subgraph Isomorphism Algorithms. MATCH Communications in Mathematical and in Computer Chemistry, 77 (2). pp. 213–232. ISSN 0340–6253

[17] Tsukiyama, S.; Ide, M.; Ariyoshi, I.; Shirakawa, I. (1977), "A new algorithm for generating all the maximal independent sets", SIAM Journal on Computing, 6 (3): 505–517, doi:10.1137/0206036.