Algorithms and Computability laboratory task

Jakoniuk Kamil, Sebastian Alejandro Espin Novillo, Prusak Patryk, Stańczyk Mikołaj

[Requirements 2](#_Toc152353392)

[Size of a graph 2](#_Toc152353393)

[Distance function 2](#_Toc152353394)

[Maximum clique 2](#_Toc152353395)

[Maximum common subgraph 2](#_Toc152353396)

[Maximum common induced subgraph 2](#_Toc152353397)

[Maximum common edge subgraph 3](#_Toc152353398)

[Input 3](#_Toc152353399)

[Summary 3](#_Toc152353400)

[Methods 3](#_Toc152353401)

[Modular product 3](#_Toc152353402)

[Maximal Clique 4](#_Toc152353403)

[Bron-Kerbosch 4](#_Toc152353404)

[Approximation 5](#_Toc152353405)

[Tests 5](#_Toc152353406)

[Technical details 6](#_Toc152353407)

[Representation 6](#_Toc152353408)

[Compilation and Execution 6](#_Toc152353409)

[Conclusions 7](#_Toc152353410)

# Requirements

## Size of a graph

In literature[[1]](#endnote-1), one will most often find the definition of size of a graph as the number of graph’s edges. To be more specific, having a graph G = (V, E). Where V is the set of vertices, and E is the set of edges, size of a graph is defined by |E|.  
Since, one of the goals of the laboratory task is to define the computational complexity of algorithms used, throughout our work we will work with a different definition. We assume that size of a graph is the number of edges plus number of vertices (|V| + |E|). Solely to avoid a situation where a non-empty graph would be of size 0.

## Distance function

As a distance function we use the measure of graph edit distance[[2]](#endnote-2). Graph edit distance measures the number of elementary operations needed to transform one graph isomorphic to another desired graph. Elementary operations include:

* Adding or removing isolated vertices
* Adding or removing edges

## Maximum clique

A *clique* of a graph *G* can be defined as a *complete subgraph* of *G[[3]](#endnote-3).* A subgraph is a graph whose vertices and edges are a subset of another graph, in this case graph *G*. Complete graph is a graph, in which every pair of distinct vertices is connected by an edge. A clique is maximal if it is not possible to add another vertex to the clique, in other words a maximal clique is not a subset of any other clique of a given graph[[4]](#endnote-4). A maximal clique is called maximum if there are no other maximal cliques with a larger number of vertices. In the case of multigraphs, we will be dealing with complete digraph[[5]](#endnote-5), which means that every pair of distinct vertices in such directed graph, is connected by an edge in each direction.

## Maximum common subgraph

We distinguish two categories of a maximum common subgraph: maximum common induced subgraph and maximum common edge subgraph[[6]](#endnote-6). In our program we find the maximum common induced subgraph.

### Maximum common induced subgraph

An *induced subgraph* is a graph constructed from a subset of vertices of a given graph G and all the edges connecting the subset of vertices present in graph G. A maximum common induced subgraph of a given set of graphs Sis graph Hthat is an induced subgraph of each of the graphs in set S and graph H has as many vertices as possible.

### Maximum common edge subgraph

A graph H is called a maximum common edge subgraph of given graph set S if it’s such a graph that is isomorphic to subgraphs of S and has got as many edges as possible.

## Input

Input given to the program must follow certain rules that have been commonly agreed upon.

* Input should be given in a form of a .txt file containing descriptions of one or more graphs separated by a blank line.
* The first line of the file has the number of graphs present in the file.
* Next lines concern the description of graphs.
  + First line (second line in total in case of the first graph) contains the number of vertices of the following graph.
  + Next the rows of the adjacency matrix of a given graph are presented in a format of a number, followed by space or newline, in the case of end of a row.
  + After the adjacency matrix the file might contain some additional data about the graph.

## Summary

The purpose of this task is to create an implementation of three algorithms:

* Finding a maximum common subgraph of two (or more) given graphs.
* Finding a maximum clique of a graph.
* Approximation to finding a maximum clique finding algorithm with polynomial time complexity.

The algorithms have been implemented in a form of an executable written in C language that accepts files in format presented in sections above. The implemented algorithms have been thoroughly tested. Results of these tests are presented in sections below.

# Methods

To find a *maximum common induced subgraph* we make use of the fact that this problem can be reduced to finding a *modular product* of two graphs and then finding the *maximum clique* of the product graph[[7]](#endnote-7). A naïve approach would be to first multiply all input graphs and then find a maximum clique of the product in hopes to get a maximum common induced subgraph. This is not a correct approach. What it really does is it finds a maximum common induced subgraph between the product of n-1 (n being the number of graphs) graphs and the last graph. We approach the problem differently; the algorithm is as follows. Given n input graphs, if n is greater than 2, take first two graphs and find their modular product, then maximum cliques of that modular product. For each found maximum clique, map it to first graph in the product. Take the result and check whether it’s a subgraph of the second input graph to ensure correct solutions for multigraphs. Take the resulting maximum common subgraph of graph 1 and 2, set graph 1 to found maximum common subgraph, set graph 2 to next graph in the input and repeat the process. This way we make use of backtracking to explore all branches of computation to ensure that the found maximum common subgraph is the largest possible and is a subgraph of each of the input graphs.

By mapping a found maximum clique onto a graph, we understand recovering original indexes of vertices, and proper edges and weights between them. Moreover, the addition of checking whether a found maximum clique is a subgraph of the second graph in graph product might seem surprising and unnecessary, however, the reduction of finding a maximum common subgraph to that of finding a modular product and then a maximum clique of two graphs is true for graphs, not necessarily multigraphs. For multigraphs, we also want to ensure that weights of corresponding edges are the same. To do that found maximum clique, that has been mapped to first graph of graph product, needs to be permutated in all possible graphs that remain isomorphic to each other. One of the permutations must prove to be the same as found maximum clique mapped to second graph of modular graph product.

## Modular product

Multiplication of graphs can be defined in different ways. In the context of this task, we assume the following definition. Modular product of two graphs G and H is the *cartesian product* V(G) x V(H) where any vertices (u, v) and (u’, v’) are adjacent if and only if u is distinct from u’, v is distinct from v’ and one of the below statements is true:

* u is adjacent with u’ and v is adjacent with v’.
* u is not adjacent with u’ and v is not adjacent with v’[[8]](#endnote-8).

For multigraphs we assume that the new weight of an edge in product graphs is the product of corresponding edges in original graphs. Additionally, we define adjacency in multigraphs in the following way: two vertices u and v of a directed graph are adjacent if there exist a unique edge from u to v and a unique edge from v to u. In the case when the second condition of vertices not being adjacent is satisfied, the edge assumes the weight of 1.

## Maximal Clique

### Bron-Kerbosch

For finding a maximal clique we use the Bron-Kerbosch algorithm with pivoting that lists all maximal cliques of a given graph. Definition of recursive implementation[[9]](#endnote-9):

**algorithm** BronKerbosch2(R, P, X) **is**  
 **if** *P* and *X* are both empty **then**  
 report *R* as a maximal clique  
 choose a pivot vertex *u* in *P* ⋃ *X*  
 **for each** vertex *v* **in** *P* \ N(*u*) **do**  
 BronKerbosch2(*R* ⋃ {*v*}, *P* ⋂ N(*v*), X ⋂ N(*v*))  
 *P* := *P* \ {*v*}  
 *X* := *X* ⋃ {*v*}

The algorithm is initialized with empty R and X, P contains all the vertices of a given graph G. The set P can be thought of as a set of candidates, set R is a temporary set and X is the set of excluded vertices. The notation N(u) means the set containing the neighbors of vertex u. The process can be described as follows. Choose a pivoting vertex from P or X (to cut down the amount of computation branches), take a vertex v from P that is not a neighbour of the pivot. Add v to R and remove its non-neighbours from sets P and X, repeat the process for the rest of the vertices in P \ N(pivot) until P is empty, then if X is also empty report R as a maximal clique. Then backtrack to last chosen vertex and remove it from the set P and add it to the set X, continue until there are no more vertices to choose from in set P \ N(pivot).

To be more specific we use the variation with pivoting but in an iterative form to increase the efficiency of our program and to prevent stack overflow in the case of large input data[[10]](#endnote-10).

**algorithm** Iterative\_BK\_pivot(P) **is**  
 Stack := null  
 Stack.push({}, P, {})  
 **while** Stack is not empty **do**  
 R, P, X := Stack.pop()  
 **if** P and X are both empty **then**  
 report R as a maximal clique  
 **else**  
 Choose pivot vertex q from P ⋃ X  
 **if** P \ N(q) is not empty **then**  
 v := some vertex in P \ N(q)   
 Stack.push(R, P \ {v}, X ⋃ {v})  
 Stack.push(R ⋃ {v}, P ⋂ N(v), X ⋂ N(v))

The principle of operation is identical as before, with the exception that recursion is simulated with the use of a stack data structure. Extracting a maximum clique from the proposed solution is straightforward, we simply only take the results with highest number of vertices. The worst case running time of Bron-Kerbosch algorithm with pivoting is O()[[11]](#endnote-11)

### Approximation

An important part of the task is to find an approximation algorithm to the algorithm that has got an exponential time complexity in our solution. In this case, it means that an approximation algorithm must be defined to find maximum clique of a graph. The proposed algorithm was described in the paper called “Approximation Algorithms for Combinatorial Problems”[[12]](#endnote-12) and it follows given steps:

1. Set SUB = ~, REST = N.
2. If REST = ~, halt and return SUB.
3. Let y ∈ REST be that element connected to the most other elements of REST.
4. Set SUB = SUB ⋃ {y}. REST = REST -- {points not connected to y}.
5. Go to 2.

Where N = number of vertices. These steps find a good enough approximation of the maximum clique. The time complexity of this algorithm is O (n log n) which is faster than the polynomial complexities.

# Tests

## Performance

The implemented solution has been tested regarding its performance. Below we present results of graphs of size 5, 10, 11, 12 ,13 ,14 of amounts 1, 2, 3, 5. The graphs have been randomly created with a 50% chance of an edge between two vertices, graphs are undirected, however the program also of course accepts directed graphs. The results depending on how many edges there are in each graph can highly differ, the more edges the more possible branches of computation.

// TODO insert tables here, in a format: table of 1 graphs of different sizes, table of 2 graphs of different sizes and so on, maybe create charts too

The running time of the algorithm grow exponentially fast with input graph size, which is reasonable given that the main algorithm used (Bron-Kerbosch) has got the worst time complexity of O(). Profiling with tools such as *gprof* also reveals that this function is the most time consuming, it is being called multiple times throughout the execution, because it is being used as a part of the algorithm for finding maximum common subgraph. We have also tested the sole capability of the program in terms of finding maximum cliques, by providing input of single graphs of size: 100, 200, 300, 400, 500, 600.

// TODO: put table here

We run the same tests for multigraphs in two variations. With number of edges between vertices having a possible value from 1 to 10, and from 1 to 5. As previously the graphs are directed and have a 50% chance of two edges being connected. The smaller the range, the more possible branches of computation. We take into account that in the case of graph isomorphism of multigraphs number of edges between corresponding vertices needs to be the same, so if the range is higher certain branches can be discarded earlier.

//TODO: Put multigraph tables here, for 1 to 10 and 1 to 5

The capabilities of solely finding maximum cliques have also been tested for multigraphs, here, however there is no difference, since the Bron-Kerbosch algorithm cares only about whether some vertices are adjacent, not whether there is one edge between them or more than one.

# Technical details

## Representation

There are two main ways to represent a graph inside a computer program:

* Adjacency Matrix
* Adjacency List

The advantages of using an adjacency matrix are that it can be represented through a simple array, given we know the number of vertices beforehand (before allocating the whole needed memory, which can be at runtime, for example during reading of the input file). Its access times are also faster compared to the adjacency list which can be implemented via a linked list. However, if one wants to operate on big graphs, space complexity grows quite fast and although algorithms implemented with adjacency matrices are in general faster, they consume much more memory. We have decided to remain with adjacency matrix implementation. Throughout the development of our solution, for quite some time, our program was implemented in both ways: with adjacency matrix and with adjacency list, taking the consideration the time complexity of problem at hand, we decided to remain with the solution offering more speed of computation.

We also implemented a data structure commonly called a Vector, which is a dynamic array, it proved to be very useful in the implementation of the Bron-Kerbosch algorithm. Vectors have also been used to simulate a simple Stack data structure in the implementation of iterative variation of the Bron-Kerbosch and in backtracking needed to explore all branches of computation in finding a maximum common subgraph.

## Possible Improvements

There is nearly always a one way or another to improve the performance of a computer program, our implementation is no different. Profiling with *gprof* reveals that implemented program spends most of the execution time in iterative Bron-Kerbosch algorithm, this became even more apparent with the implementation of finding a maximum common subgraph. A potential improvement to the efficiency of our solution would be to instead use a different algorithm for finding a maximum common subgraph, for example the McSplit algorithm.

To improve space complexity, one could choose more restrictive types. Knowing that graphs number of edges in each direction between vertices, in other words weights of edges, won’t be greater than certain number, i.e. 256 and will always be greater or equal 0, one could use a unsigned char for storing the weights instead of a regular int, which would result in half the size for each stored weight.

## Compilation and Execution

To compile the program on a machine with *gcc* installed:

gcc main.c -o main

To then run it with an input file located in a folder ‘input’:

./main ./input/example.txt

Or:

main.exe ./input/example.txt

Depending on whether one uses the PowerShell or the standard cmd. Similarly for multiple input files:

main.exe ./input/example1.txt ./input/example2.txt

The program accepts multiple input files and multiple input graphs in each file in a format described in previous sections. To make use of all the capabilities of the program, it should be compiled and executed on a 64-bit system. Naturally, the program has been thoroughly tested with Valgrind, resulting in a memory safe and efficient implementation.

# Conclusions

The requirements such as distance function, size of a graph, maximum clique, maximum common subgraph have been defined. The problem of finding maximum cliques and maximum common subgraph has been successfully solved by implemented program. Given an input in specific format describing a certain number of graphs the program outputs maximum cliques of each graph and a maximum common induced subgraph for all input graphs. The heart of the implemented solution is the iterative Bron-Kerbosch algorithm with pivoting. Out of different variations (recursive, with or without pivoting) we have chosen the most efficient one. Moreover, an approximation algorithm for finding maximum cliques has been successfully implemented, resulting in a polynomial finding of an approximation of a maximum clique, what follows an approximation algorithm for finding maximum common subgraph has also been implemented. Lastly, the program has been thoroughly tested and checked for memory leaks to ensure robustness and to allow the program to process large graphs.

1. Balakrishnan, V. K. (1997). Graph Theory (1st ed.). McGraw-Hill [↑](#endnote-ref-1)
2. https://link.springer.com/chapter/10.1007/978-3-319-27252-8\_2 [↑](#endnote-ref-2)
3. https://mathworld.wolfram.com/Clique.html [↑](#endnote-ref-3)
4. https://www.intechopen.com/chapters/12929 [↑](#endnote-ref-4)
5. https://mathworld.wolfram.com/CompleteDigraph.html [↑](#endnote-ref-5)
6. https://en.wikipedia.org/wiki/Maximum\_common\_subgraph [↑](#endnote-ref-6)
7. Barrow, H.; Burstall, R. (1976), "Subgraph isomorphism, matching relational structures and maximal cliques", Information Processing Letters, 4 (4): 83–84 [↑](#endnote-ref-7)
8. https://en.wikipedia.org/wiki/Modular\_product\_of\_graphs [↑](#endnote-ref-8)
9. https://www.sciencedirect.com/science/article/pii/S0304397508003903?via%3Dihub [↑](#endnote-ref-9)
10. https://stackoverflow.com/questions/76141667/iterative-version-of-the-bron-kerbosch-algorithm-with-pivoting [↑](#endnote-ref-10)
11. https://www.sciencedirect.com/science/article/pii/S0304397500002863?via%3Dihub [↑](#endnote-ref-11)
12. David S. Johnson, Approximation algorithms for combinatorial problems, Journal of Computer and System Sciences, Volume 9, Issue 3, 1974, Pages 256-278, ISSN 0022-0000, https://doi.org/10.1016/S0022-0000(74)80044-9. [↑](#endnote-ref-12)