ELSEVIER Theoretical Computer Science 250 (2001) 1–30

Theoretical Computer Science

www.elsevier.com/locate/tcs

Fundamental Study

Enumerating all connected maximal common subgraphs in two graphs

Ina Koch [*]

MDC - Max-Delbrück-Centrum, Walter-Friedrich-Haus, Robert-Rössle-Str. 10, 13092 Berlin, Germany

Received March 1999; accepted June 2000
Communicated by M. Nivat

Abstract

We represent a new method for finding all connected maximal common subgraphs in two graphs which is based on the transformation of the problem into the clique problem. We have developed new algorithms for enumerating all cliques that represent connected maximal common subgraphs. These algorithms are based on variants of the Bron–Kerbosch algorithm. In this paper we explain the transformation of the maximal common subgraph problem into the clique problem. We give a short summary of the variants of the Bron–Kerbosch algorithm in order to explain the modification of that algorithm such that the detected cliques represent connected maximal common subgraphs. After introducing and proving several variants of the modified algorithm we discuss the runtimes for all variants by means of random graphs. The results show the drastical reduction of the runtimes for the new algorithms. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Graph theory; Connected maximal common subgraphs; Cliques; Bron–Kerbosch algorithm

Contents

[*] Corresponding author. Tel.: +49-30-9406-2733; fax: +49-30-9406-2834.
E-mail address: ikoch@mdc-berlin.de (I. Koch).

1. Introduction

In many applications it is important to find maximal common subgraphs in two graphs. Because the problem is NP-complete it cannot be solved for arbitrarily large graphs. Searching for *connected* maximal common subgraphs can reduce the complexity of the problem drastically, although it remains NP-complete.

The initiation for this work has been assumed from theoretical biology, where it is of great interest to compare proteins at different structural description levels in order to find structural motifs [17]. Proteins are complex structures. They consist of several 10 000 of atoms. There is no method comparing two proteins without using any additional information which is used for placing a seed before starting the comparison at the atomic description level. Therefore, proteins are considered at different abstraction levels. On the one side, to reduce the computational complexity and, on the other side, to detect hidden structural similarities, proteins are often considered at their *secondary structure* level. Secondary structure elements represent repetitive structural subunits. A protein can consist of up to 80 secondary structure elements. We can model protein structures as undirected labelled graphs, where the vertices represent secondary structure elements and the edges spatial neighbourhoods between them. Applying the clique algorithm for finding all structural similarities in two *protein graphs* often the complexity arises in such a way that the runtime increases up to some days and more. For a comparison of protein structures in an acceptable time we needed to develop a method which overcomes this problem. This was the motivation for beginning this work. Among other things the results are new algorithms which should be presented and discussed in this paper.

First of all we want to give some definitions. The terminology is based on that given by Harary [12]. Let us consider undirected labelled graphs $G=(V,E)$, which are defined by finite vertex sets V and sets of undirected edges $E\subseteq\mathscr{P}_2(V)$. $\mathscr{P}_2(V)$ is the set of all subsets of V with exactly two different vertices. Maximal common subgraphs (MCSs) are subgraphs which cannot be extended, i.e., a maximal subgraph cannot be a real subgraph of another maximal subgraph. We can formulate the *maximal common subgraph problem* as follows.

Definition 1.1 (*MCS problem*).
Instance: Two graphs $G_1=(V_1,E_1)$, $G_2=(V_2,E_2)$.
Solution: All MCSs $H=(V,E)$ such that H is isomorphic to subgraph $G_1'=(V_1',E_1')$ of G_1 and $G_2'=(V_2',E_2')$ of G_2, and the appropriate isomorphic mappings $f_1 : V(H)\rightarrow V(G_1)$ and $f_2 : V(H)\rightarrow V(G_2)$.

We can solve the problem by transforming the MCS problem into the clique problem. This has been suggested by Levi [18] for the calculation of maximal common-induced subgraphs. Note that we want to consider maximal common, and not maximal common-induced subgraphs. A clique is a complete maximal subgraph. If we want to find a clique with largest possible cardinality we speak of the *maximum*-clique problem. Because this problem is important for many practical applications, a lot of algorithms have been developed, for example the *branch and bound* algorithm of Balas and Samuelsson [3], the recursive backtracking method of Tarjan and Trojanowski [23], the recent methods by Balas and Yu [4], and Babel and Tinhofer [2]. These methods work for arbitrary graphs and have exponential runtimes. Other algorithms solve the problem for special graph classes with polynomial runtimes, for example for chordal graphs [8], for transitive free orientable graphs [10], and for $K_{1,3}$ free graphs [14].

For many applications all MCSs in two graphs are required, i.e., we have to enumerate all cliques in a graph. Outgoing from the fastest and most widely used algorithm, the Bron–Kerbosch [6] algorithm, 1973, we have modified this algorithm and its variants [15]. We distinguish between cliques which describe connected subgraphs and those which describe disconnected subgraphs. The new algorithms consider only those cliques which represent connected subgraphs during the search. So the solution tree is reduced drastically. The resulting runtimes make it possible to consider very large graphs. Thus, in our application for the comparison of protein structures now large proteins can be compared in a reasonable amount of time and space.

First, we want to explain the transformation of the maximal common subgraph problem into the clique problem (Section 2). Then we repeat shortly variants of the Bron–Kerbosch algorithm and prove their invariants (Section 3). Second we define the modified clique problem and discuss the new algorithms (Section 4). In Section 5, we explain some the used data structure. Finally in Section 6, we discuss the runtime results of all algorithms for protein graphs and random graphs. We finish with the conclusions in Section 7.

2. Transformation of the MCS problem

The transformation of one problem into another problem for which algorithms exist is a widely used technique in computer science. We can transform the MCS problem into the clique problem. The aim is to reduce the search space before using a rigorous algorithm.

Let us consider two undirected labelled graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ with n and m vertices, respectively. We can map parts of both graphs onto another. In this case we say that certain vertices and edges of graph G_1 are *compatible* to certain vertices and edges of graph G_2. We store the information of all possible compatibilities in a new graph, the so-called *product graph* or *compatibility graph*. Depending on searching induced subgraphs or subgraphs, we can generate a *vertex* product graph and an *edge* product graph resp.

Definition 2.1 (*Vertex product graph or vertex compatibility graph*). The *vertex product graph* $H_v = G_1 \circ_v G_2$ includes the vertex set $V_H = V_1 \times V_2$, in which the vertex pairs (u, v) with $u \in V_1$ and $v \in V_2$ have the same labels.

An edge between two vertices $u_H, v_H \in V_H$ with $u_H = (u_1, u_2)$ and $v_H = (v_1, v_2)$ exists exactly then, if $u_1 \neq v_1$ and $u_2 \neq v_2$, and if the vertex pair u_1, v_1 shares a common edge in G_1 with the same label as the common edge shared by $u_2 \neq v_2$ in G_2, or if u_1, v_1 and u_2, v_2 are not adjacent in G_1 and G_2, respectively.

If some vertices $(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)$, for $1 \leqslant k \leqslant |V_1| \cdot |V_2|$, in the vertex product graph are pairwise adjacent, then the subgraph in G_1 induced by the vertices $u_1, u_2, \ldots, u_k$ is isomorphic to the subgraph in G_2 induced by the vertices $v_1, v_2, \ldots, v_k$. The isomorphism is implicitly given by the vertices $(u_1, v_1), (u_2, v_2), \ldots, (u_k, u_k)$ of the vertex product graph, i.e., $h(u_i) = v_i$ for $i = 1, 2, \ldots, k$. Consequently, a maximal common induced subgraph corresponds to a maximal complete subgraph bijectively, i.e., to a clique in the vertex product graph H. This was proved by Levi [18].

The definition of the edge product graph is analogous to that of the vertex product graph.

Definition 2.2 (*Edge product graph or edge compatibility graph*). The *edge product graph* $H_e = G_1 \circ_e G_2$ includes the vertex set $V_H = E_1 \times E_2$, in which the edge pairs (e_i, e_j) with $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$ have to coincide in their edge labels and the corresponding end vertex labels.

There is an edge between two vertices $e_H, f_H \in V_H$ with $e_H = (e_1, e_2)$ and $f_H = (f_1, f_2)$, if $e_1 \neq f_1$ and $e_2 \neq f_2$, and if either e_1, f_1 in G_1 are connected via a vertex of the same label as the vertex shared by e_2, f_2 in G_2, or e_1, f_1 and e_2, f_2 are not adjacent in G_1 and in G_2, respectively.

Fig. 1 gives an example for the construction of the product graph H_e from two graphs G_1 and G_2. For more clearness only parts of H_e, G_1, and G_2 are depicted. The vertices in H_e are formed by compatible edges in G_1 and G_2, for example the edge pairs $(1, 1')$ and $(2, 2')$. For the edge pair $(2, 1')$ the corresponding edges are incompatible, because the edge labels differ, although the labels of the end vertices are equal.

The edges (straight lines) in H_e are formed by compatible edge pairs of G_1 and G_2. So, the edge pairs $(2, 2')$ and $(3, 3')$ in H_e are compatible, because both edges 2 and 3 in G_1 and $2'$ and $3'$ in G_2 are connected via a common vertex of the same label. Also the edge pairs $(1, 1')$ and $(4, 4')$ are compatible, because both $(1, 4)$ in G_1 and $(1', 4')$ in G_2 do not share a common vertex. On the other hand, the edge pairs $(2, 2')$ and $(3, 4')$ in H_e are incompatible, because edges 2 and 3 in G_1 are connected via a common vertex, but not edges $2'$ and $4'$ in G_2. These incompatibilities of two vertices in H_e are depicted by dotted lines.

To get a common subgraph in G_1 and G_2 each edge pair in G_1 and G_2 (vertex in H_e) has to be compatible to the all those edge pairs in G_1 and G_2 (edges in H_e),

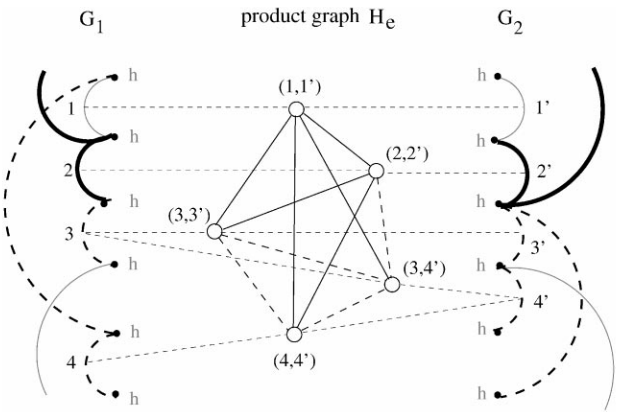

Fig. 1. A part of the construction of the edge product graph H_e from G_1 and G_2.

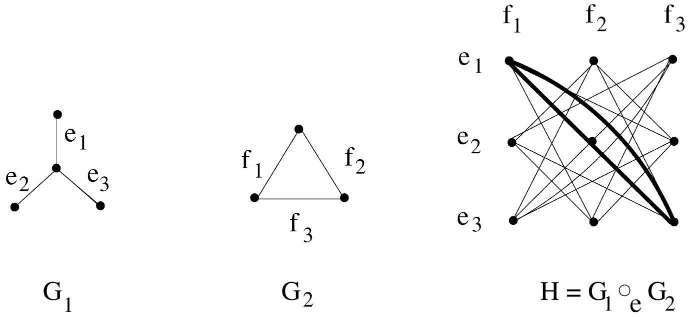

Fig. 2. An example in which cliques with three vertices do not necessarily define isomorphic subgraphs. Though the edge product graph H_e exibits a clique (bold edges) with the edge set $V_H = \{(e_1, f_1), (e_2, f_2), (e_3, f_3)\}$, G_1 (the Y-shape or triod) is not isomorphic to G_2 (the triangle).

which are forming a common subgraph. Thus, a clique in H_e corresponds to a maximal common subgraph in G_1 and G_2 [18].

If some vertices $(e_1, f_1), (e_2, f_2), \dots, (e_k, f_k)$ with $1 \leqslant k \leqslant |E_1| \cdot |E_2|$ in the edge product graph are pairwise adjacent, then the subgraph in G_1 represented by the edges $e_1, e_2, \dots, e_k$ is isomorphic to the subgraph in G_2 represented by the edges $f_1, f_2, \dots, f_k$. This edge isomorphism is valid if and only if there is no interchange of so-called *Y-shapes* or *triods* and triangles (see [24] for connected graphs and [22] for disconnected graphs). Fig. 2 shows an example for cliques with three vertices, which do not necessarily define isomorphic subgraphs, because of a triod\triangle interchange. To find MCSs we can also start from the edge graphs $L(G_1)$ and $L(G_2)$, and search for maximal common-induced subgraphs in the resulting vertex product graph (see Fig. 3). Outgoing from the edge product graph we find MCSs in G_1 and G_2. We yield the same result if we calculate the edge graphs $L(G_1)$ and $L(G_2)$, and search in the vertex product graph for maximal common induced subgraphs. After a conversion into vertex graphs these maximal-induced subgraphs correspond to MCSs in G_1 and G_2.

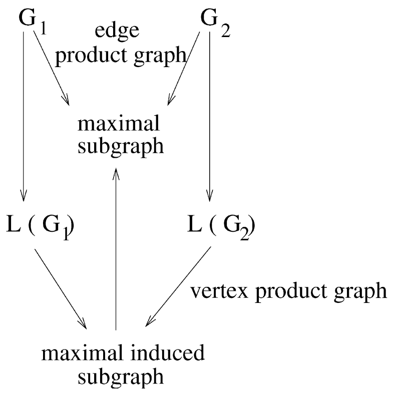

Fig. 3. The correlations between maximal subgraph and maximal-induced subgraph in two graphs and their edge graphs.

Because we want to find subgraphs and not induced subgraphs, only the edge product graph is of interest. We call the edge product graph H_e in this paper as *product graph* G. The product graph G contains all possible matches of edge pairs of the original graphs G_1 and G_2. Maximal common subgraphs in the original graphs G_1 and G_2 correspond to cliques in G. Consequently, we have transformed the MCS problem into the clique problem. All noncompatible edge pairs were excluded at the outset by this transformation.

3. The clique problem

The clique problem is one of the six basic problems of known NP-complete problems [7]. It is defined as a decision question in the following way:

Definition 3.1 (*Clique problem*).
Instance: A graph $G=(V,E)$, $0\leqslant k\leqslant |V|$.
Solution: Is there a complete subgraph $H=(V',E')$ of size k or larger?

This question is also equivalent both for complete and maximal complete subgraphs, because there must exists at least one maximal common subgraph if one has found a complete subgraph.

If we want to find a clique of maximal cardinality we have to solve the maximum-clique problem. Recent results demonstrate that the maximum-clique problem is hard to approximate in polynomial time within a factor $n^{1-\varepsilon}$ [13]. We want to consider the more complex problem of enumerating all cliques in a graph, the *all-clique problem*.

Definition 3.2 (*All-clique problem*).
Instance: A graph $G=(V,E)$.
Solution: All maximal complete subgraphs $H=(V',E')$ of G.

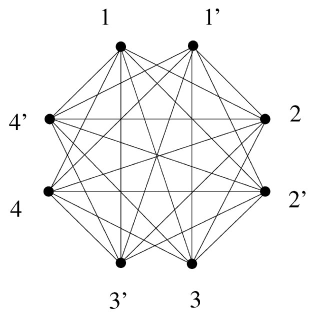

Fig. 4. A graph that consists of eight vertices which can be considered as the four vertex pairs $(1,1')$, $(2,2')$, $(3,3')$, and $(4,4')$. The vertices of these pairs are not adjacent, whereas all other vertices are adjacent.

In contrast to the maximum-clique problem which is NP-complete the all-clique problem is NP-hard. The number of cliques can go up exponentially. Fig. 4 shows an example that illustrates that the number of cliques increases with the number of vertices exponentially. Because the vertices of the n vertex pairs are not adjacent, a vertex does not belong to a clique which involves the other vertex of the pair. Each clique contains exactly one vertex of a pair, because otherwise no maximal complete subgraph exists. Thereby, the choice of the vertex in the pair is arbitrary. Thus, 2^n cliques arise in that graph.

Moon and Moser [21] show also the exponential relation between the maximum number of cliques $f(n)$ and the number of vertices n in a graph G. For $n \geqslant 2$ holds that $f(n) = 3^{n/3}$, if $n \equiv 0 \pmod 3$, $f(n) = 4.3^{[n/3]-1}$, if $n \equiv 1 \pmod 3$, and $f(n) = 2.3^{n/3}$, if $n \equiv 2 \pmod 3$. Additionally, they report for any graph G_n with n vertices for $n \geqslant 4$ that for the maximal number of the different clique sizes the upper bound is $g(n) \leqslant n - [\log n]$ and the lower bound is $g(n) \geqslant n - 2[\log n] - 1$. These results emphasize the complexity of the all-clique problem.

3.1. Algorithms

The branch-and-bound algorithm of Bron and Kerbosch [6] (*BK-algorithm*) works recursively and is reported as the fastest enumeration algorithm [5, 9]. It is a robust algorithm which can be modified easily. Several variants of the algorithm are known (see [15, 1]). The algorithms will be repeated shortly because they serve as basis for the new algorithms which find maximal connected common subgraphs.

3.1.1. The simple BK-algorithm

The algorithm finds all cliques in a graph exactly once. It works on the three sets C, P, and S.

Set C contains the set of vertices belonging to the current clique. Set P contains all vertices which can be used for the completion of C because they are adjacent to the vertex added at last to C. In S all vertices are collected, which can no longer be

Table 1
Algorithm 1 – the BK-algorithm

```
ENUMERATE-CLIQUES $(C, P, S)$
$\triangleright$ enumerates all cliques in an arbitrary graph $G$
$C$: set of vertices belonging to the current clique
$P$: set of vertices which can be added to $C$
$S$: set of vertices which are not allowed to be added to $C$
$N[u]$: set of neighbours of vertex $u$ in $G$

01  Let $P$ be the set $\{u_1, \ldots, u_k\}$;
02  **if** $P = \emptyset$ **and** $S = \emptyset$
03     **then** REPORT_CLIQUE;
04     **else for** $i \leftarrow 1$ **to** $k$
05              **do** $P \leftarrow P \setminus \{u_i\}$;
06                 $P' \leftarrow P$;
07                 $S' \leftarrow S$;
08                 $N \leftarrow \{v \in V \mid \{u_i, v\} \in E\}$;
09                 ENUMERATE_CLIQUES $(C \cup \{u_i\}, P' \cap N, S' \cap N)$;
10                 $S \leftarrow S \cup \{u_i\}$;
11              **od**;
12  **fi**;
```

used for the completion of C, because all cliques containing these vertices are always generated.

The algorithm (see Table 1) starts with the empty sets C and S. Initially, P includes all vertices of the graph G. If P and S are empty, a clique was found and will be reported (line 03). Besides each vertex of P is considered in a loop (lines 04–11), where the arbitrarily chosen vertex u_i is eliminated from P. P and S are copied into P' and S', respectively (line 06 and 07) for the recursion. The neighbours of vertex u_i are generated and stored in N (line 08). Vertex u_i is added to C and the recursion call with $P' \cap N$ and $S' \cap N$ takes place.

Theorem 3.3. *Algorithm* 1 *has following invariants*:

(i) *All vertices in C are pairwise adjacent*, *i.e.*, *C is a vertex set of a complete subgraph. Each vertex in G which is adjacent to all other vertices in C is either in P or in S.*

(ii) *Each vertex $u \in P$ is adjacent to all vertices in C. The vertices from P are used for the extension of C. Once a vertex from P is used for the extension of C it will be moved to S.*

(iii) *Each vertex $u \in S$ is adjacent to all vertices in C. The vertex sets of all cliques containing $C \cup \{u\}$ are already enumerated once.*

(iv) *If the call of the function* ENUMERATE_CLIQUES() *is finished*, *the vertex sets of all cliques containing C are enumerated exactly once.*

Proof. (i) At the beginning C is empty. Next, an arbitrary vertex $u \in P$ will be added to C. In the following, C is only extended by vertices from P which are all neighbours of all vertices in C. Thus the vertices in C build a complete subgraph which must not be a maximal one. Because P contains all vertices from G at the beginning, and P and S are always intersected with the neighbours of vertex $u \in P$ before the recursion call, each vertex from G, which is adjacent to all vertices in C, is either in P or in S.

(ii) From the stepwise intersection of P with the neighbours of the newly added vertices mentioned in proof (i) follows that each vertex $u \in P$ is adjacent to all vertices in C. The other statements follow directly from the algorithm. Lines 01, 04, and 09 indicate that only vertices from P are used for the extension of C. In line 05 vertex $u \in P$ which extends C is removed from P and added to S (line 10).

(iii) From the stepwise intersection of S with the neighbours of the newly added vertices mentioned in proof (i) follows that each vertex $u \in S$ is adjacent to all vertices in C. In the **for**-loop all neighbours of vertex $u \in P$ added to C are proved one after another for their possible extension of $C \cup \{u\}$ by the recursive calls of the function ENUMERATE_CLIQUES(). Thus, it is ensured that all cliques containing $C \cup \{u\}$ are enumerated after the recursion step.

(iv) It follows from (iii) that all cliques containing $C \cup \{u\}$ are reported. Because each vertex $u \in P$ added to C is written to S after the recursion step, S contains those vertices u at the respective recursion level which were already considered in the recursion. If P is empty anytime, i.e., no vertex exists which can be added to C, two possibilities arise for S:

(a) Set S is not empty if vertices in S are neighbours of vertex u such that the originating complete subgraph is not a maximal one because of (iii).

(b) Set S is empty if the vertex set of neighbours of vertex u is disjoint to S of the previous recursion level. So a new clique was found.

It is guaranteed by S that each clique is reported only once. □

Fig. 5 represents the *solution tree* or *recursion tree* for a graph G. The vertices $u \in P$ are used for the completion of C in increasing order. In this example the cliques were found during the first steps because vertex 1 added first to set C is a member of all cliques.

3.1.2. Recognition of equal subtrees

The recursion tree in Algorithm 1 can be reduced by some simple modifications.

Theorem 3.4. *It is sufficient to consider only those vertices from P in the* **for**-*loop which are not adjacent to a vertex $u \in P$.*

Proof. Assume, vertex $u_i \in P$ which is passed through the **for**-loop in Algorithm 1 is neighboured to all other vertices $u_{i+1}, \dots, u_k \in P$, which have not been considered so far. Then, it is not necessary to consider these vertices $u_{i+1}, \dots, u_k \in P$ anymore, because due to (iii) in Theorem 3.3 all cliques, which contain $C \cup \{u_i\}$ and so possibly contain also the vertices $u_{i+1}, \dots, u_k$, are found by addition of u_i to C and by the following recursion. If the vertices $u_{i+1}, \dots, u_k \in P$ would be considered in the **for**-loop nevertheless vertex u_i would be always in S. The vertex would remain in S' after the intersection because vertex u_i is adjacent to all $u_{i+1}, \dots, u_k \in P$. □

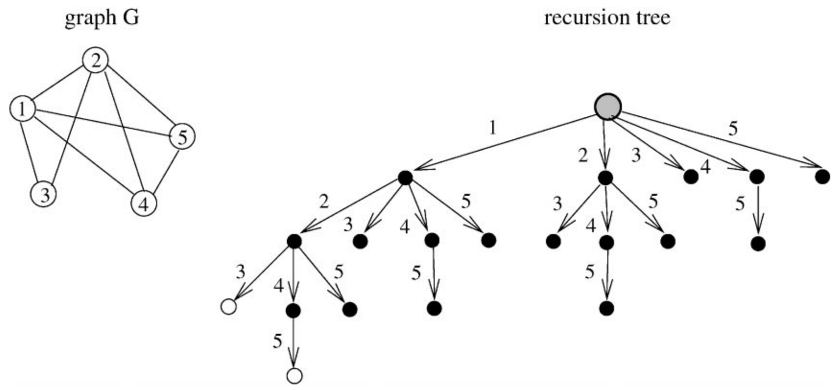

Fig. 5. The recursion tree of Algorithm 1 (right) for a graph G (left). The edges of the recursion tree are labelled by vertex u added to the current set C. The gray vertex is the root of the recursion tree. Paths from the root to a white end vertex describe cliques in G. Paths from the root to a black end vertex describe complete subgraphs in G which are not maximal, because they can be extended by at least one vertex from the nonempty set S.

Table 2
Algorithm 2 – a modificated enumeration algorithm

```
ENUMERATE_CLIQUES $(C, P, S)$
▷ enumerates all cliques in an arbitrary graph $G$

01  Let $P$ be the set $\{u_1, \ldots, u_k\}$;
02  if $P = \emptyset$
03   then if $S = \emptyset$ then REPORT_CLIQUE; fi;
04   else Let $u_t$ be a vertex from $P$;
05        for $i \leftarrow 1$ to $k$
06            do if $u_i$ is not adjacent to $u_t$
07                   then $P \leftarrow P \setminus \{u_i\}$;
08                        $P' \leftarrow P$;
09                        $S' \leftarrow S$;
10                        $N \leftarrow \{v \in V \mid \{u_i, v\} \in E\}$;
11                        ENUMERATE_CLIQUES $(C \cup \{u_i\},\ P' \cap N,\ S' \cap N)$;
12                        $S \leftarrow S \cup \{u_i\}$;
13               fi;
14           od;
15  fi;
```

Table 2 depicts Algorithm 2 which is based on Algorithm 1. Merely, line 04 which chooses vertex u_t and line 06, which proves the adjacency of a vertex u_i to a vertex u_t, are added. Fig. 6 represents the essentially smaller recursion tree for the graph in Fig. 5 using Algorithm 2.

Algorithm 2 is the second version of the BK-algorithm which is used frequently [11, 19, 20]. Because the choice of vertex $u_t \in P$ is arbitrary we can use some heuristics, such as the choice of that vertex with the largest vertex degree in order to decrease the vertex set that has to pass through the **for**-loop. The clique containing the vertex with

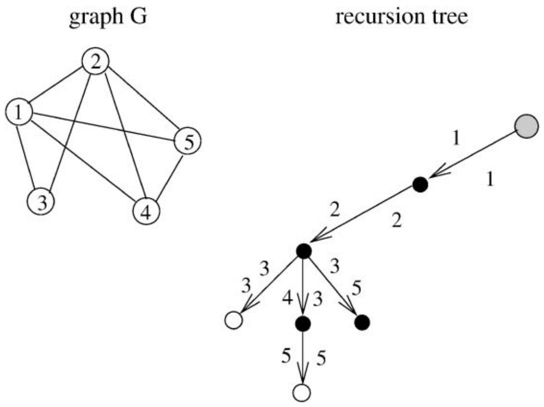

Fig. 6. The recursion tree of Algorithm 2 (right) for the graph G (left). The edges of the recursion tree are called by the respective vertex u added to C as black numbers. The light numbers describe the respective vertex u_t. The gray vertex is the root of the recursion tree. Paths from the root to a white end vertex describe cliques in G. Paths from the root to a black end vertex represent complete subgraphs in G which are not maximal.

the largest degree do not have to be the largest clique. In many cases (see Section 6) the choice of vertex $u_t \in P$ with the largest degree will result in the smallest recursion tree what is not stringent (Fig. 6).

Theorem 3.5. *The choice of a vertex $u_t \in P$ with the largest degree results not necessarily in a decreased recursion tree.*

Proof. The recursion trees of Algorithm 2 for the disconnected graph G with $2p+1$ vertices (see Fig. 7) should be considered in two variants. G consists of a complete subgraph with p vertices and an astral subgraph with $p+1$ vertices. The numbering of vertices in G is chosen such that the first p vertices are located in the complete subgraph and the vertex $p+1$ with the largest degree is located in the astral subgraph. In the first variant of Algorithm 2 we select the first vertex in P as vertex u_t (recursion tree 1 in Fig. 7). In the second variant of Algorithm 2 we choose the first vertex in P with the largest degree as vertex u_t (recursion tree 2 in Fig. 7).

Variant 1: The first u_t is vertex 1. Vertices which are not adjacent to vertex 1 pass through the **for**-loop. So $p+1$ branches arise at the root vertex. Let us consider the first branch in the root vertex. Because of Theorem 3.3 at the branch at which first vertex 1 is added to C the complete subgraph with p vertices is found. Let us consider the second branch in the root vertex. Because of Theorem 3.3 at the branch at which first vertex $p+1$ is added to C, p cliques of size 2 in the astral subgraph are found.

The branches that add the vertices $p+n$ with $n=2,3,\dots,p+1$ to C as the first vertex already cancel at the next level, because vertex $p+1$ always remains as the only neighbour of the vertices $p+1$. Vertex $p+1$ is an element of S (resulting from the second branch) and cannot be eliminated from S by intersection. Then, P is empty

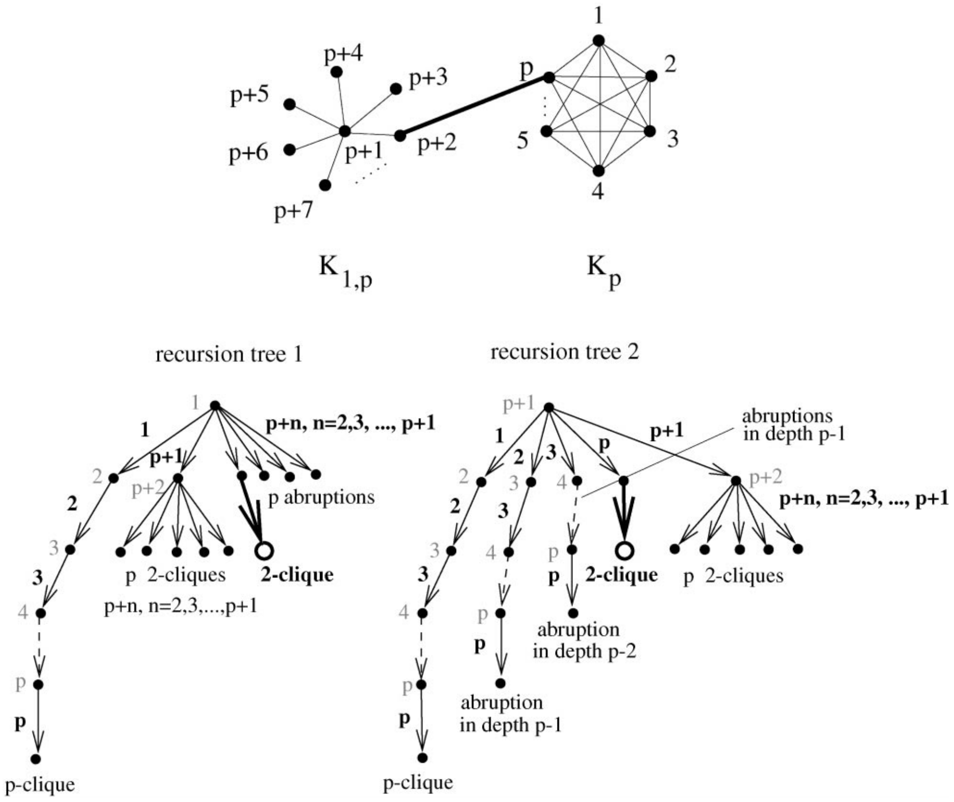

Fig. 7. A disconnected graph G with $2p+1$ vertices and the recursion trees of two variants of Algorithm 2. The light numbers indicate the respective vertex u_t. The bold numbers indicate vertices added to C. By insertion of the bold edge in G a connected graph arises. The additional vertices and edges in the recursion trees 1 and 2 are drawn lightly.

and S is not empty such that there is an abruption at this point. We yield $a_p = 3p+2$ as the number of vertices of the recursion tree 1. The addend $3p$ results from the number of the levels for building up the large clique with p vertices (p-clique) in the first branch, from the vertices of depth 2 for the p cliques of size 2 (2-cliques), and the vertices for the p abruptions of the last branches in depth 1. Additionally, we have to involve the root vertex and the branch vertices after adding $p+1$ in depth 1.

Variant 2: Now, the first vertex u_t is vertex $p+1$, because it exhibits the largest number of neighbours. Vertices that are not neighboured to $p+1$ pass through the **for**-loop. So $p+1$ branches arise at the root vertex. Let us consider the first p branches in the root vertex, which add the vertices of the complete subgraph to C. Because of Theorem 3.3 the p-clique is found at the branch that first adds vertex 1. At the branch that first adds vertex 2 a nonmaximal complete subgraph with $p-1$ vertices is found. That causes an abruption in depth 1, etc. until the abruption at the last vertex in depth 1.

Let us consider that branch in the root vertex where vertex $p+1$ is added to C. Because of Theorem 3.3 the p 2-cliques are found in the astral subgraph. Counting the vertices of the recursion tree 2, we yield $b_p = p + 2 + (1 + 2 + \cdots + p)$. The addend $p+2$ results from the p tree vertices in depth 1, from the root vertex, and from the branching vertex after adding $p+1$ in depth 1. The sum $1 + 2 + \cdots + p$ results from the vertices in the depths 1 to p. This part corresponds to $(p^2 + p)/2$ such that we yield $b_p = 2 + (p^2 + 3p)/2$ as number of tree vertices for Variant 2.

We see clearly that the recursion tree in Variant 2 grows faster than in Variant 1. In case $p = 3$ the number of vertices in the recursion tree is 11 for both variants. If $p = 4$ the recursion tree of Variant 2 exhibits two vertices more than of Variant 1. □

We yield a disconnected graph by inserting an edge between the astral and the complete subgraph in G. In Fig. 7 this edge is drawn in bold formation. We yield for both recursion trees an additional vertex such that the statement of Theorem 3.5 is also valid for connected graphs.

Lemma 3.6. *The vertex u_t can also be chosen from set S or $P \cup S$.*

Proof. (1) $u_t \in S$: Assume, if we find a clique C' through a path, S' is empty. I.g., a vertex v which is not adjacent to vertex u_t must be added to C such that S is empty after the intersection. If we add vertex v as first vertex at this branch point this clique C' is also reached because S becomes empty. Vertex v is passing through the **for**-loop because v is not adjacent to u_t. Vertex u_t can belong to S, and the clique C' will nonetheless be found via vertex v. So C' is not lost.

(2) $u_t \in S \cup P$: Because of the proven statements that vertex u_t can be chosen from P (Theorem 3.3) and from S (see above) u_t can also be chosen from $S \cup P$. □

4. The modified clique problem

A clique in the edge product graph G represents not only a pair of identical MCSs but also an isomorphism between the MCSs of this pair. So it is possible that a set of different cliques represents equal pairs of identical common subgraphs. The size of the automorphism group of a graph of the pair is expressed by the variety of its versions. To reduce the number of the reported cliques and hence the runtime of the algorithm it would be helpful to analyse symmetries in the MCs in order to circumvent the output of cliques which only represent different isomorphisms between a pair of MCSs. Large automorphism groups exist for MCSs which consist of different disconnected MCSs. If we search for connected MCSs in a connected graph the problem simplifies such that the MCSs, which consist of different disconnected subgraphs, must not be considered in the recursion tree during the search.

For the realization of that improvement in both algorithms the edges in the edge product graph $G = G(G_1, G_2)$ are divided into *c-edges* (*connected edges*) and *d-edges* (*disconnected edges*). They are labelled according to their division.

Definition 4.1 (*c-Edges and d-Edges*). Let (e_1,e_2) and (f_1,f_2) be two edge pairs of two graphs G_1 and G_2. An edge of the product graph G between the vertices (e_1,e_2) and (f_1,f_2) is called *c-edge* if the edges e_1,f_1 in G_1 or e_2,f_2 in G_2 exhibit a common vertex, otherwise the edge is called a *d-edge*.

Consequently, we search in the product graph for so-called *c-cliques*.

Definition 4.2 (*c-Clique*). A clique in a graph G which consists of c- and d-edges is called a *c-Clique* if it is formed by c-edges such that it is connected and acyclic.

Theorem 4.3. *A* c-Clique *in the product graph G is a clique that represents* connected *MCSs in the graphs G_1 and G_2 which form the product graph.*

Proof. ($\Rightarrow$) A c-clique that is spanned by c-edges (Definition 4.2) is given. A c-edge in G means nothing else than the according edge pair e_1,f_1 in G_1 is connected via a common vertex of G_1 and the edge pair e_2,f_2 in G_2 is connected via a common vertex of G_2. Thus, these edge pairs represent connected subgraphs in G_1 and G_2. If we consider a c-edge in G which is adjacent to that c-edge, again an edge will be added to the connected subgraphs in G_1 and G_2. This c-edge forms connected subgraphs in G_1 and G_2 in turn, because in the product graph G adjacent c-edges arise only then if the according edge pairs are connected via a common edge in G_1 and G_2. That results in connected MCSs, finally.

($\Leftarrow$) Connected MCSs in G_1 and G_2 consist of several adjacent edges. A connected edge pair in G_1 that can be mapped on a connected edge pair in G_2 exhibits a c-edge in G. If an adjacent edge is added to that connected edge pair in G_1 and G_2, a new mapping possibility of the new edge and an edge of the edge pair in G_1 to a new edge and an edge of the edge pair in G_2 exists. This mapping possibility is represented via a c-edge in G which is adjacent to the first c-edge. Consequently, the clique that corresponds to these MCSs is formed by c-edges such that it is connected and acyclic. □

As a result we can formulate the *all-c-clique problem* as follows:

Definition 4.4 (*All-c-clique problem*). *Instance:* a graph $G=(V,E)$ with $E=Z\cup U$. Let Z be the set of all c-edges in G, and D the set of all d-edges in G.

Solution: all maximal complete subgraphs $H=(V',E')$ whose c-edges span the graph H.

4.1. Algorithms

4.1.1. Modification of Algorithm 1

To solve the all-c-clique problem we can modify Algorithm 1 (see Table 3) easily. The set C of a clique is extended only by those vertices $u\in P$ which are adjacent to at least one vertex of C via a c-edge in G. In Algorithm 1 we divide the set P into

Table 3
Algorithm 3 – an algorithm for the detection of all c-cliques which is based on Algorithm 1

```
ENUMERATE_C_CLIQUES ($C, P, D, S$)
▷ enumerates all c-cliques in an arbitrary graph $G$
$P$: set of vertices which can be added to $C$, because they are neighbours of vertex $u$ via c-edges
$D$: set of vertices which cannot directly be added to $C$, because they are neighbours of $u$ via d-edges

01 Let $P$ be the set $\{u_1, \ldots, u_k\}$;
02 if $P = \emptyset$ and $S = \emptyset$
03  then REPORT_CLIQUES;
04  else for $i \leftarrow 1$ to $k$
05   do $P \leftarrow P \setminus \{u_i\}$;
06   $P' \leftarrow P$;
07   $D' \leftarrow D$;
08   $S' \leftarrow S$;
09   $N \leftarrow \{v \in V \mid \{u_i, v\} \in E\}$;
10   for all $v \in D'$
11    do if $v$ and $u_i$ are adjacent via a c-edge
12     then $P' \leftarrow P' \cup \{v\}$;
13          $D' \leftarrow D' \setminus \{v\}$;
14     fi;
15    od;
16   ENUMERATE_C_CLIQUES ($C \cup \{u_i\}, P' \cap N, D' \cap N, S' \cap N$);
17   $S \leftarrow S \cup \{u_i\}$;
18   od;
19 fi;
```

P and D such that P contains only vertices which are adjacent to at least one vertex of C via a c-edge and D only vertices which are not adjacent to any vertex in C via a c-edge. Now, as before C is extended only by vertices of P. If a vertex $u \in P$ is selected the vertices from D are proved whether they are adjacent to vertex u via a c-edge (**for**-loop in lines 10 to 15). In this case we eliminate this vertex from D (line 13) and add it to P (line 12). For the recursion the new set D is intersected with the neighbours of vertex u last added to C (line 16).

Analogously to Algorithms 1 and 2, we report a c-clique if P and S are empty. If P is empty the current set C cannot be extended. If S is empty no vertex set that contains C is reported as yet.

Lemma 4.5. *A c-clique can also be reported if set D is not empty.*

Proof. If D is not empty the product graph G still exhibits vertices which are adjacent to all vertices in C, but via d-edges. That is independent on whether P and S are empty. Fig. 8 gives an example for finding a c-clique with a nonempty set D. Vertices 3 and 5 are in D from the beginning, because they are connected via d-edges to the vertices of the c-clique. They remain in D also after the intersection because they are all neighbours to all vertices in C. □

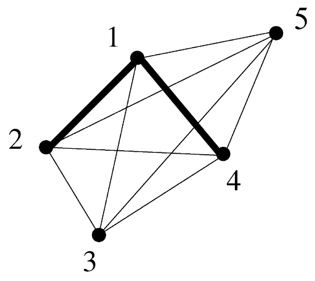

Fig. 8. A graph, in which a c-clique is found with Algorithm 3, although the set D is not empty. The c-edges are drawn in bold formation.

Table 4
Algorithm 4 – the initialization algorithm for Algorithms 3 and 5

```
INIT_ALG_3 $(C, P, D, S)$
$\triangleright$ initilization of Algorithms 3 and 5
$T$: set of vertices which have already been used for the initialization of ENUMERATE_C_CLIQUES

$T \leftarrow \emptyset$;
for all $u \in V$
do $P \leftarrow \emptyset$;
   $D \leftarrow \emptyset$;
   $S \leftarrow \emptyset$;
   $N \leftarrow \{v \in V | \{u, v\} \in E\}$;
   for each $v \in N$
    do if $u$ and $v$ are adjacent via a c-edge
     then if $v \in T$
             then$S \leftarrow S \cup \{v\}$;
             else $P \leftarrow P \cup \{v\}$;
          fi;
     else if $u$ and $v$ are adjacent via a d-edge
             then $D \leftarrow D \cup \{v\}$;
          fi;
     fi;
    od;
   ENUMERATE_C_CLIQUES $(\{u\}, P, D, S)$;
   $T \leftarrow T \cup \{u\}$;
od;
```

In contrast to Algorithms 1 and 2, Algorithm 3 cannot be started with an empty set C. Because each vertex in P must be connected to each vertex in C via a c-edge, P has to be empty at the beginning. We need an initialization algorithm (see Table 4).

We could start Algorithm 3 for each vertex $u \in V$ with the parameter list $(\{u\}, P, D, \emptyset)$, whereby P contains the neighbours of vertex u which are connected with u via a c-edge, and D contains the neighbours of u which are connected with u via a d-edge. In that case Algorithm 3 would report each c-clique with n vertices exactly n times.To avoid this we have to eliminate those vertices from P which have initialized Algorithm 3. We introduce a new set T that contains these vertices. In Algorithm 3 we have to prove at the place, where a vertex is moved from P to S, if the vertex has already been used as a start vertex. If the vertex is an element of set T it will be moved to S' otherwise to P'. We have to change line 12 in Algorithm 3 as follows:

```
11    **if** $v \in T$ **then** $S' = S' \cup v$; **else** $P' \leftarrow' \cup v$; **fi**;
```

Theorem 4.6. *Algorithm* 3 *exhibits the following invariants:*

(i) *All vertices in C are pairwise adjacent, and C is spanned by c-edges. Each vertex of G that is adjacent to all vertices in C is in P, D, or S.*
(ii) *Each vertex $u \in P$ is adjacent to all vertices in C and to at least one vertex in C via a c-edge. Vertices from P are used for the extension of C. Is a vertex from P once used for the extension of C or as initializing vertex of the function* ENUMERATE_C_CLIQUES() *it will be moved to S.*
(iii) *Each vertex $u \in S$ is adjacent via c-edges or d-edges to all vertices in C. The vertex sets of all c-cliques which contain the vertex set $C \cup \{u\}$ are already enumerated once.*
(iv) *Each vertex $u \in D$ is adjacent to all vertices in C via d-edges. A vertex from D can be used for the extension of P, if a vertex from P was moved to C, which is adjacent to that vertex in D via a c-edge.*
(v) *The set T contains only vertices which already have been used once for initializing the function* CNUMERATE_C_CLIQUES().
(vi) *If the call of the function* ENUMERATE_C_CLIQUES() *is finished, the vertex sets of all c-cliques are enumerated exactly once.*

Proof. (i) The set C contains one vertex u at the beginning of the call of ENUMERATE_C_CLIQUES(). C is extended only by vertices from P, i.e., by vertices which are adjacent to all vertices in C and to at least one of them via a c-edge. During each recursion call the sets P, D, and S are intersected with the neighbours of vertex u. Thus the sets P, D, and S contain only neighbours to all vertices in C. In the **for**-loop (lines 10–15) each vertex v that is adjacent to u is moved to P, D, or S.

(ii) Because of the stepwise intersection of P with the neighbours of the newly added vertices mentioned in the proof of (i) it is obvious that each vertex $u \in P$ is adjacent to all vertices in C and to at least one vertex in C via a c-edge. From lines 01, 05, and 16 follows that only vertices from P serve for the extension of C. In line 17 vertex u is moved to S after passing through the recursion. If $u \in T$ (line 12, the above modified line) vertex u is moved to S.

(iii) Because of the stepwise intersection of S with the neighbours of the newly added vertices mentioned in the proof of (i) it is obvious that each vertex $u \in S$ is adjacent to all vertices in C. Vertices which are adjacent to u via c-edges and which are also in T, i.e., which were already used as initializing vertices are moved to S

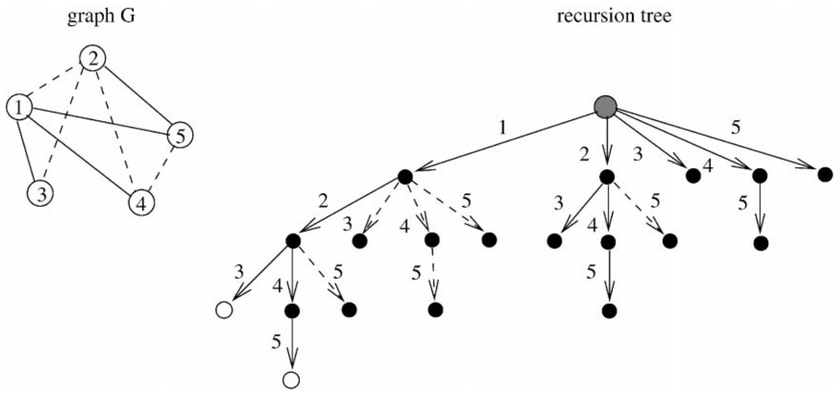

Fig. 9. The complete recursion tree of Algorithm 3 (right) for for the graph G (left). The c-edges are drawn as dotted lines. The vertices are labelled with the current vertex u which is added to the current set C. The grey vertex is the root of the recursion tree. Only the paths consisting of solid edges have to be considered. Paths from the root to a white end vertex describe c-cliques in G. Paths from the root to a black end vertex describe complete subgraphs which are spanned by c-edges, but which are not maximal and therefore do not represent cliques.

(line 10 in Algorithm 4). All neighbours of vertex $u \in P$, that was previously added to C, are examined one after another if they could extend $C \cup \{u\}$. This is done by the recursion calls of the function ENUMERATE_C_CLIQUES in the **for**-loop. So it is ensured that all cliques containing $C \cup \{u\}$ are enumerated.

(iv) Because of lines 13 and 14 in Algorithm 4 only those vertices are moved to D which are adjacent to vertex u via a d-edge. They are only eliminated from D if they are adjacent to a new vertex in C via a c-edge (line 13 in Algorithm 3).

(v) This statement follows directly from line 19 in Algorithm 4.

(vi) From (iii) follows that all c-cliques involving $C \cup \{u\}$ are reported. Because each vertex $u \in P$ that was added previously to C is moved to S after the recursion, S contains at the respective recursion level those vertices u which were already considered during the recursion. If S is intersected with the neighbours of a vertex added previously to C, two possibilities arise when P is empty.

(a) S is nonempty if vertices in S are neighbours of all vertices from C. So the arising complete subgraph is not maximal.

(b) S is empty if all vertices previously located in S are removed because they are not adjacent to at least one vertex in C.

The set S guarantees that each c-clique is reported only once in the function ENUMERATE_C_CLIQUES(). The set T guarantees that vertices which have served as initializing vertices are not added to P. □

Fig. 9 demonstrates the decrease of the recursion tree by solving the all-c-clique problem instead of the all-clique problem. To get all c-cliques it is sufficient to travers the paths consisting of solid edges. The other pathes will be not considered by the algorithm.

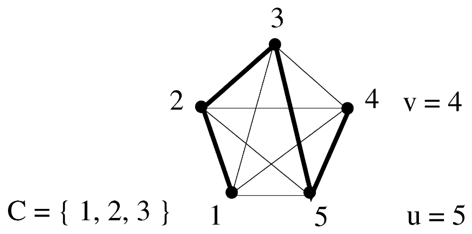

Fig. 10. An example for a state during the run of Algorithm 5 for searching a c-clique. The bold edges are c-edges. Set C contains vertices 1, 2, and 3. Vertex v can only be moved to C if vertex u is already in C.

4.1.2. Modification of Algorithm 2

A vertex v which is in D can later be added to C because it could be moved to P. This occurs when a vertex $u_i \in P$ is added to C which exhibits a c-edge to $v \in D$. Vertex v cannot be added to P or C previously, because it is adjacent via a c-edge to vertex u_i which is in P, but not yet in C (see Fig. 10). Thus, vertices from D can move to P one after another via a so-called *c-path* (see Definition 4.7). This phenomenon must be taken into account if we want to reduce the recursion tree by the detection of equal subtrees as in Algorithm 2. The corresponding Algorithm 5 is represented in Table 5. It must be examined for all neighbours v of the selected vertex u_i wether they are in P (line 13 in Algorithm 5). Then, vertex v moves to P'. If $v \in S$, v moves to S' (line 23). If $v \in D$ we have to decide whether vertex v should be moved from D to S' or to P'. If vertex v is adjacent to u_i via a c-edge (line 16) and has not been an initializing vertex (line 17), v moves to P' (line 19). Otherwise, v is added to S' (line 18).

Although we can reduce the recursion tree by the above described modification of Algorithm 2 the effort for the additional comparisons is too high, such that the runtime reduces not such drastically as by the modification from Algorithms 1 to 3.

Definition 4.7 (*c-Path*). For a given set D a c-path is a path from vertex u to vertex v in the graph G such that all its edges are c-edges and all its vertices except vertex u belong to set D.

Theorem 4.8. *Although the reduction of vertices which have to pass through the* **for**-*loop* (*lines* 05–32), *realized by the condition in line* 06 *of Algorithm* 5, *still all c-cliques are reported exactly once.*

Proof. Let C' be an arbitrary c-clique in G. Let us consider an arbitrary place in the recursion tree with the sets C, P, D, and S with $C \subseteq C'$. It should be demonstrated that the c-clique C' with $P' = S' = \emptyset$ is reported by the choice of a vertex u_t and the reduction of the **for**-loop, if only those vertices $u \in P$ enter the **for**-loop,

(1) which are either not adjacent to vertex u_t or
(2) which are adjacent to a vertex in D which is not adjacent to vertex u_t via a c-path.

Table 5
Algorithm 5 – an enumeration algorithm for the detection of all c-cliques which is based on Algorithm 2

```
ENUMERATE_Z_CLIQUES ($C, P, D, S$)
▷ enumerates all c-cliques in an arbitrary graph $G$

01 Let $P$ be the set $\{u_1, \dots, u_k\}$;
02 if $P = \emptyset$
03    then if $S = \emptyset$ then REPORT_CLIQUE; fi;
04    else Let $u_t$ be a vertex from $P$;
05       for $i \leftarrow 1$ to $k$
06          do if $u_i$ is not adjacent to $u_t$ or $u_i$ is connected via a c-path
06                with a vertex from $D$ that is not adjacent to $u_t$
07                then $P \leftarrow P \backslash \{u_i\}$;
08                   $P' \leftarrow P$;
09                   $D' \leftarrow D$;
10                   $S' \leftarrow S$;
11                   $N \leftarrow \{v \in V | \{u_i, v\} \in E\}$;
12                   for all $v \in D'$
13                      do if $v \in P$
14                         then $P' = P' \cup \{v\}$;
15                         else if $v \in D$            \\ can v be added to $P$?
16                               then if $v$ and $u_i$ are adjacent via a c-edge
00                                        \\ is v an initializing vertex?
17                                     then if $v \in T$
18                                           then $S' = S' \cup \{v\}$;
19                                           else $P' \leftarrow P' \cup \{v\}$;
20                                        fi;
21                                        $D' \leftarrow D' \backslash \{v\}$;
22                                  fi;
23                               else if $v \in S$ then $S' = S' \cup \{v\}$; fi;
24                            fi;
25                         fi;
26                      od;
27                fi;
28                ENUMERATE_C_CLIQUES ($C \cup \{u_i\}, P' \cap N, D' \cap N, S' \cap N$);
29                $S \leftarrow S \cup \{u_i\}$;
30          od;
31 fi;
```

Let us consider the following cases:

(i) If C' contains a vertex $u \in P$ which is not adjacent to u_t, so vertex u fulfils (1) and can enter the **for**-loop.

(ii) Now, let $C' \cap P$ contain only vertices which are adjacent to vertex u_t. Then, we can distinguish the following two cases:

  (a) If $u_t \in C'$, u_t can enter the **for**-loop because u_t is not adjacent to itself. Then, (1) is also fulfilled.

  (b) Now, let $u_t \notin C'$. Set C' is found at the branch point with the set C (see Fig. 11) in the unrestricted recursion tree. $P^{(i)}$ and $S^{(i)}$ with $0 \leqslant i \leqslant k$ should be the sets P and S, respectively, which are found at certain vertices on the

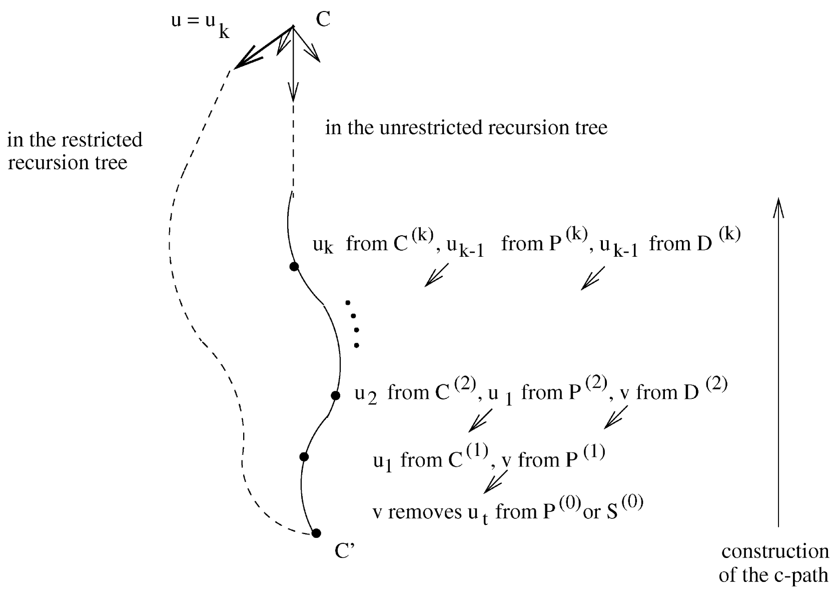
u = u_k
C
in the restricted recursion tree
in the unrestricted recursion tree
u_k from C^(k), u_k-1 from P^(k), u_k-1 from D^(k)
u_2 from C^(2), u_1 from P^(2), v from D^(2)
u_1 from C^(1), v from P^(1)
v removes u_t from P^(0) or S^(0)
C'
construction of the c-path

Fig. 11. The outline of the proof for Algorithm 5.

path from C to C' in the unrestricted recursion tree. Because $u_t \notin C'$, vertex u_t must be removed from $P^{(0)}$ and $S^{(0)}$ respectively anytime. But this cannot be done by a vertex from P because all vertices from $C' \cap P$ are adjacent to vertex u_t (see (ii)). Consequently, it can only be reached via a vertex v from D. Such a vertex v cannot be adjacent to vertex u_t, because u_t would always remain in $P^{(0)}$ and $S^{(0)}$ respectively during the intersection.

It remains to show that in this case we can find a vertex $u \in C'$ which has entered the **for**-loop, because it fulfils (2). Thereto, it is required that this vertex u exhibits a c-path to a vertex $v \in D$ which is not adjacent to vertex u_t. Let us construct this path now.

We consider vertex $v \in D$ that we have found above. Vertex v must attain $P^{(0)}$ anytime in order to move to set C' later. This occurs only if vertex v is adjacent via a c-edge to a vertex $u_1 \in P^{(1)}$ which has entered $C^{(0)}$. If vertex $u_1 \in P$ holds we choose $u = u_1$, and we have finished. Otherwise $u_1 \in D$ holds, and we continue with the construction. Vertex u_1 must in turn be moved from D to $P^{(1)}$. That occurs only if vertex u_1 is adjacent via a c-edge to a vertex $u_2 \in P^{(2)}$ which has entered $C^{(1)}$. We can continue until we have found a vertex $u_k \in P$. Thus, a c-path from vertex u_k to a vertex v arises, and $u = u_k$ fulfils the condition (2).

Herewith, we demonstrated the possibility for the reduction of the **for**-loop at a branch point in the recursion tree. Attaining the next branch point in the recursion tree the same possibility for the reduction of the **for**-loop exists, etc. until we reach the c-clique C' also in the reduced recursion tree. □

Table 6
The data structure for objects

```
typedef struct obj_type
{
    int vertex;
    int set;
    struct obj_type *next_set;
    struct obj_type *prev_set;
    struct obj_type *next_vertex;
    struct obj_type *prev_vertex;
} OBJ_TYPE;
```

5. Data structure

We used a special data structure with the functions *insert*, *remove*, and *membership*. Usually, it is not possible to find an implementation where all these functions are running in constant time. But in our special case we could realize this efficient kind of implementation. We have implemented a network of objects of the structure *obj_type* (see Table 6).

We consider two types of paths through this network realized by two lists of addresses of these objects. The first list *set_array* stores the addresses of objects for the sets C, P, S, and additionally D for Algorithms 3, 4, and 5 in the order as they emerge during the run. The second list *vertex_array* has the length of the cardinality of the product graph G. It contains the addresses of those objects, which have considered the respective vertex last. The sets and vertices are subscripted by their numbers in the arrays *set_array* and *vertex_array* respectively.

Tables 7 and 8 represent the functions NEW_SET(), INSERT(), MEMBERSHIP(), REMOVE_OBJECT(), REMOVE_SET(), and REMOVE_VERTEX_FROM_SET() for handling of the data structure *obj_type*. We always create a new object by moving the pointer *next_vertex* to the set M, when a new vertex u is inserted into set M. We assign the address of M in *set_array* to the pointer *prev_vertex* of the new object. Analogously, we assign the address of vertex u in *vertex_array* to the pointer *prev_set* of the new object. The pointer *next_set* of vertex u is written to the pointer *next_set* of the new object. Via both lists *set_array* and *vertex_array* we have fast entries to the network. Coevally, we have realized a double catenation of the objects, once over the sets and once over the vertices. Along the pointer chain $set\_array[M].next\_vertex \to \cdots \to next\_vertex$ we yield all vertices belonging to set M. Along the pointer chain $vertex\_array[u].next\_set \to \cdots \to next\_set$ we yield all sets, which were passed by vertex u.

Lemma 5.1. *The insertion of sets and vertices takes place in constant time.*

Proof. The insertion of a set M takes place by appending the set to the head of the list *set_array*, i.e., in constant time. The insertion of a vertex u takes place by the creation of a new object, which will be appended to the head of the list *vertex_array*[u]. Thus, the time for insertion remains bounded to a constant. □

Table 7
The functions NEW_SET(), INSERT(), and MEMBERSHIP()

```
$o$: object
$no$: number of objects
$ns$: number of sets
$size$: array for the set sizes
$set\_array$: list with the addresses of sets
$vertex\_array$: list with the addresses of vertices

int NEW_SET()
▷ introduces a new set and returns the new set number.
$set\_array[ns].next\_vertex = NIL$;
$size[ns] \leftarrow 0$;                         // initializes the set size
$ns \leftarrow ns + 1$;                          // increments the set number
return(ns);

void INSERT($u, s$)
▷ inserts a vertex $u$ to set $s$.
NEW_OBJECT();                                     // creates a new object $o$
$no \leftarrow no + 1$;                          // increases the number of objects
$o \rightarrow vertex \leftarrow u$;              // inserts vertex $u$ to object $o$
$o \rightarrow set \leftarrow s$;                 // inserts set $s$ to object o
$o \rightarrow prev\_vertex \leftarrow \&set\_array[s]$;   // inserts object $o$ to $set\_array$
$o \rightarrow next\_vertex \leftarrow set\_array[s].next\_vertex$;
$set\_array[s].next\_vertex \leftarrow o$;
if $o \rightarrow next\_vertex$
   then $o \rightarrow next\_vertex \rightarrow prev\_vertex \leftarrow o$;
fi;
$o \rightarrow prev\_vertex \leftarrow \&vertex\_array[u]$;   // inserts $o$ to $vertex\_array$
$o \rightarrow next\_set \leftarrow vertex\_array[u].next\_set$;
$vertex\_array[u].next\_set \leftarrow o$;
if $o \rightarrow next\_set$
   then $o \rightarrow next\_set \rightarrow prev\_set \leftarrow o$;
fi;
$size[s] \leftarrow size[s] + 1$;                // increases the set size

int MEMBERSHIP($u, s$)
▷ answers the question after the membership of vertex $u$ to a set $s$.
$x \leftarrow 0$;                                // initializes the counter for searched sets
$o \leftarrow vertex\_array[u].next\_set$;       // sets the initial object
while $o$ and $o \rightarrow set \neq s$
  do $o \leftarrow o \rightarrow next\_set$;     // passes the next_set-pointer
     $x \leftarrow x + 1$;
  od;
if $x > 8$ and $o$                               // $u$ was not found in the last 8 sets
  then return(0);
fi;
return ($o \neq 0$ and $o \rightarrow set = s$);
```

Lemma 5.2. *The question for the membership of a vertex u to a set M can be answered in constant time.*

Proof. According to Theorem 3.3(i) for Algorithms 1 and 2 and Theorem 4.6(i) for Algorithms 3, 4, and 5 each vertex of the product graph G is situated either in C, P,

Table 8
The functions REMOVE_OBJECT(), REMOVE_SET(), and REMOVE_VERTEX_FROM_SET()

```
void REMOVE_OBJECT($o$)
▷ removes an object $o$.
// replaces the pointers in the previous set and previous vertex
$o \rightarrow prev\_set \rightarrow next\_set \leftarrow o \rightarrow next\_set$;
$o \rightarrow prev\_vertex \rightarrow next\_vertex \leftarrow o \rightarrow next\_vertex$;
if $o \rightarrow next\_set$
   then $o \rightarrow next\_set \rightarrow prev\_set \leftarrow o \rightarrow prev\_set$;
fi;
if $o \rightarrow next\_vertex$
   then $o \rightarrow next\_vertex \rightarrow prev\_vertex \leftarrow o \rightarrow prev\_vertex$;
fi;
$no \leftarrow no - 1$;                          // decreases the number of objects

void REMOVE_SET($s$)
▷ removes a set $s$.
$ns \leftarrow ns - 1$;
if $s \neq ns$                                  // decreases the number of sets
   then exit;
fi;
while $set\_array[s].next\_vertex$              // removes all objects in the set
   do  REMOVE_OBJECT($set\_array[s].next\_vertex$);
   od;

void REMOVE_VERTEX_FROM_SET($u, s$)
▷ removes a vertex $u$ from set $s$.
$o \leftarrow vertex\_array[u].next\_set$;
if $o = 0$ or $o \rightarrow set \neq s$;
   then exit();
fi;
REMOVE_OBJECT($o$);                             // removes object $o$ of vertex $u$
$size[s] \leftarrow size[s] - 1$;               // decreases the number of objects
in the set
```

or S for Algorithms 1 and 2, and also in D for Algorithms 3, 4, and 5. A vertex can only occur in these sets which are appearing during one recursion step. Thus, we are interested in knowing whether a vertex occurs in the last six (for Algorithms 1 and 2) or eight (for Algorithms 3–5), respectively, created sets. So it is sufficient to consider at most eight sets, namely the sets C, P, S, and D, and the modified sets for the recursion call C', P', S', and D'. Thus, the search time through the list, which contains the respective sets in which a vertex can occur, is bounded to a constant. □

Lemma 5.3. *The removing of sets and vertices takes place in constant time.*

Proof. By Lemma 5.2 the access to that place in the net, where the object to be deleted is located, takes place in constant time. The removing of a set or a vertex is performed by moving of the pointers *next_vertex* and *prev_vertex*, and the pointers *next_set* and *prev_set* respectively, and a deletion of the object. □

Because of Lemmata 5.1, 5.2, and 5.3 the used data structure realizes the functions *insert*, *remove*, and *membership* in constant time. The needed time for one recursion step is constant.

6. Runtime results

6.1. Results from graphs derived from protein structures

First, we want to discuss the runtime results for those graphs which were derived from protein structures. In this case graphs to be compared are limited in their size by nature. They exhibit at most 80 vertices and 100 edges. About one quarter of them has more than 10 vertices and 10 edges. The vertex degree is not larger than five. Though these graphs are not very large and dense, the arising product graphs can be large, because they have only two different vertex labels and three different edge labels, so that a lot of matches can appear. Most product graphs exhibit 100–150 vertices and 2000–4000 edges. The number of circles differs from 1000 to 5000. The number of triangles is mostly much lower (500–1000) than the number of circles. The product graphs derived from protein structures cannot be assigned to a certain graph class.

The runtimes for three examples with small, medium-sized, and large product graphs are compiled in Table 9. They were performed on a SUN/SPARC-computer 10/30.

The algorithms are abbreviated as follows. *BK* stands for Algorithm 1. *BKYP* denotes Algorithm 2. The *Y* indicates the introduction of vertex u_t, and *P* indicates the set from where vertex u_t was chosen. *BKYPN* is the name for Algorithm 2 whereas the element with the largest number of neighbours was selected from set *P* as vertex u_t. Analogously, *BKYS* indicates that vertex u_t is chosen from set *S*. The algorithms containing a *Y* in their names are called also *Y-variants*. The names of the algorithms, which calculate connected maximal common subgraphs, are extended by a *C* for *connected*, e.g., *BKC*, *BKCYP* etc. The same types of variants are used. These algorithms are called *C-variants*. *BKC* denotes Algorithm 3 and *BKCYP* Algorithm 5.

We see that the runtime improvement is drastically for medium-sized and large product graphs (see Table 9). The number of cliques found by *C-variant*s is significantly lower than in case of the other variants, especially for large cliques. The number of possible substructures is much higher in case of searching cliques than searching c-cliques. Consequently, much more cliques arise than c-cliques. For example, for a clique of size 9 the difference between the number of cliques of 105 743 and of 16 for c-cliques is the most significant one. All these 105 743 cliques had to be checked for possible extensions by the algorithm, so that the recursion tree grows during the search accordingly.

Using the new algorithms it became possible to solve the problem of protein structure comparison for large proteins in a reasonable amount of time and space. Despite of the modification of the MCS-problem the results are biologically correct [16, 17].

Table 9
Runtimes, clique sizes, and numbers of cliques for product graphs derived from protein structures

Size	Small	Medium	Large	Size	Small	Medium	Large
	runtime				runtime		
alg.	(s)	(min:s)		alg.	(s)	(min:s)	
BK	0.02	172:55	>4 days	BKC	0.02	0:69	18:08
BKYP	0.02	45.71	>4 days	BKCYP	0.02	0:73	18:17
BKYPN	0.03	26.84	>4 days	BKCYPN	0.04	0:70	18:17
BKYS	0.03	20.79	>4 days	BKCYS	0.03	0:72	17:93
Size	Number of cliques			Size	Number of c-cliques		
1				1	2	15	65
2	3			2	1	20	75
3	2			3	1	24	48
4	1			4	1	30	62
5		55		5		7	39
6		2473		6		13	94
7		27075		7		2	73
8		90304		8		4	35
9		105743		9			16
10		47398		10			4
11		9484		11			
12		738		12			
13		8		13			

6.2. Results for random graphs

To investigate the runtimes of the algorithms for several large and dense graphs we have generated random graphs which serve as product graphs with 100, 300, and 600 vertices with respective different edge densities. The number of edges are 1/9, 1/6, and 1/3 of all possible edges. 1/3 of all edges are labelled as d-edges, the others as c-edges.

The results for random graphs were performed on a SUN-Enterprise 4000 with six processors and 765 MB main memory. The processors are Ultra-Sparc-1-processors with a clock rate of 176 MHz. The results are compiled in the Tables 10–12. Each table contains the graph sizes, the numbers and sizes of the reported cliques as well as the runtimes and the sizes of the recursion trees (rec. trees) per algorithm (alg.). We observe the same tendencies in the runtime performance as in case of graphs derived from protein structures.

In case of small graphs (see Table 10) there are no significant differences in the runtimes, although the size of the recursion trees is much higher for the non-C-variants than for the C-variants. Also the numbers of cliques are higher than the numbers of c-cliques. The differences in the runtime between non-Y-variants and Y-variants are also small in case of graphs with 100 vertices. Within the algorithms which do not calculate c-cliques the Y-variants are exiguously faster than the simple BK-algorithm for graphs with 100 vertices for sparse as well as for dense graphs, although the recursion trees are

Table 10
Runtimes, sizes of the recursion trees, clique sizes, and numbers of cliques for random graphs with 100 vertices

	100 vertices						
Edges	532	778	1601		532	778	1601
alg.	Runtime (s) size of rec. tree			alg.	Runtime (s) size of rec. tree		
BK	0.07	0.11	1.43	BKC	0.02	0.05	0.37
	859	1561	12389		304	547	3433
BKYP	0.06	0.12	0.99	BKCYP	0.04	0.06	0.37
	782	1323	8606		298	538	3257
BKYPN	0.07	0.11	0.93	BKCYPN	0.05	0.07	0.39
	737	1292	8011		298	534	3241
BKYS	0.07	0.11	1.04	BKCYS	0.04	0.06	0.39
	792	1376	8588		302	544	3321
Size	Clique number			Size	c-Clique number		
1				1	2		
2	158	63	1	2	95	68	3
3	182	438	197	3	37	123	151
4	6	54	1573	4	1	15	510
5	1		616	5			221
6			41	6			13
7			1	7			
8			1	8			

significantly smaller in case of Y-variants. Comparing results of the BKYP-variant with those of the BKYPN-variant the BKYPN-variant is always somewhat slower than the BKYP-variant, also for dense as well as for sparse graphs. This is due to the additional steps needed find the element from P with the largest number of neighbours.

The C-variants are always faster than the non-C-variants (see also Tables 3 and 6). The number of cliques is reduced drastically by searching c-cliques. Also the runtimes are decreased drastically. This effect occurs already for graphs with 100 vertices. The numbers of the recursions give an imagination of the sizes of the solution trees, and also demonstrate the effectiveness of the new algorithms.

Coevally, we see that the differences between the single variants of the C-algorithms are minimal even for large graphs no matter their density. Here we observe the same behaviour as in case of non-C-variants. It is noteworthy that the simple variant BKC always works faster than the other three Y-variants, although the recursion trees are significantly larger for the BKC-variant, especially for large and more dense graphs. This is caused by the additional requests in the search for a vertex u_t or for vertices, which are not adjacent to u_t or which are adjacent to u_t and are connected to a vertex from set D, which is not adjacent to vertex u_t via a c-path (see line 06 in Algorithm 5). Thus, considering the Y-variants for the c-clique problem, the algorithms become not faster, because the additional requests in the algorithms can be very time consuming.

Table 11
Runtimes, sizes of the recursion trees, clique sizes, and numbers of cliques for random graphs with 300 vertices

	300 vertices						
Edges	4938	7400	14773		4938	7400	14773
alg.	Runtime (s) size of rec. tree			alg.	Runtime (s) size of rec. tree		
BK	1.26	5.23	254.63	BKC	0.44	1.33	64.74
	11694	34762	958796		3656	9692	285640
BKYP	1.23	4.63	192.82	BKCYP	0.47	1.40	67.62
	10659	30119	665217		3578	9320	264850
BKYPN	1.29	4.39	171.91	BKCYPN	0.47	1.42	67.94
	10398	28715	612726		3571	9306	263523
BKYS	1.25	4.55	172.92	BKCYS	0.45	1.40	164.68
	10608	29354	595304		3619	9418	595304
Size	Clique number			Size	c-Clique number		
2	139	1		2	214	18	
3	3976	5336		3	1150	2019	28
4	557	5364	11973	4	164	1500	6809
5	3	291	93131	5	2	95	35737
6		2	39772	6		1	16594
7			2665	7			1347
8			43	8			30

In contrast to that the Y-variants of the non-C-algorithms always exhibit an advantage of runtime opposite to the simple BK-variant.

7. Conclusions

We have developed novel algorithms for finding all connected maximal common subgraphs in arbitrary graphs which are significantly faster than algorithms for enumerating maximal common subgraphs known so far. The new algorithms solve the problem for searching connected maximal common subgraphs. The strong decrease of the runtime is due to removing all cliques which represent disconnected subgraphs already during the search. Thus, in the recursion tree the paths which lead to disconnected subgraphs can be cut early so that the size of the recursion tree is reduced significantly. Nevertheless the problem remains NP-hard, now using the novel algorithms larger graphs can be examined than it was possible hitherto.

The new algorithms are based on the Bron–Kerbosch algorithm. We have explained the modifications of this algorithm to solve the all-c-clique Problem. We have presented and discussed some variants of the modified algorithm. Several heuristics were introduced to improve the runtime performance. But all these variants became not faster because it is too expensive to search for special vertices which should firstly pass

Table 12
Runtimes, sizes of the recursion trees, clique sizes, and numbers of cliques for random graphs with 600 vertices

	600 vertices						
Edges	19 670	29 661	59 585		19 670	29 661	59 585
	Runtime (s)				Runtime (s)		
alg.	size of rec. tree			alg.	size of rec. tree		
BK	13.09	74.49	11604.35	BKC	3.47	17.17	3156.34
	76099	308464	23415175		21556	83690	7542693
BKYP	13.00	70.24	8594.01	BKCYP	3.80	18.86	3260.09
	69544	266724	16000304		21016	80667	6927312
BKYPN	12.69	68.28	8738.24	BKCYPN	3.85	19.08	3293.56
	68023	260102	15197537		20943	80472	6900203
BKYS	12.87	67.84	7623.95	BKCYS	3.74	19.01	3278.04
	68540	257648	13972060		21173	81421	6996827
Size		Clique number		Size	c-Clique number		
2	12			2	109	1	
3	21509	11115		3	6899	6392	
4	8250	68846	4949	4	2322	20541	5725
5	161	8943	935205	5	44	2833	421481
6		112	1858200	6		49	804477
7			359975	7			180516
8			14410	8			8495
9			136	9			97
10			1	10			1

through the main for-loop. Maybe, for special graphs the runtime can be decreased by some of the heuristics, but that was not yet investigated.

In case of comparing protein structures the conventional algorithms were not applicable for large structures because of the size of the arising product graphs. Using the new algorithms the comparisons could be done mostly in a few seconds or in the scope of minutes [16]. The algorithms are robust and fast. They can be applied to arbitrary graphs. The need to calculate maximal common substructures occurs frequently. In many cases the connected maximal common substructures are sufficient solutions. We think that the presented algorithms can be applied easily in many fields.

Acknowledgements

I would like to thank Dr. Frieder Kaden and Prof. Egon Wanke for the many critical and fruitful discussions. Also I thank the referees for the helpful comments. Parts of the work were done during my stay at the GMD-German Research Center for Information Technology.

References

[1] E.A. Akkoyunlu, The enumeration of maximal cliques of large graphs, SIAM J. Comput. 2 (1973) 1–6.

[2] L. Babel, G. Tinhofer, A branch and bound algorithm for the maximum clique problem, Methods Models Oper. Res. 34 (1990) 207–217.

[3] E. Balas, H. Samuelsson, A node covering algorithm, Naval Res. Log. Quart. 24 (1977) 213–233.

[4] E. Balas, C.S. Yu, Finding a maximum clique in an arbitrary graph, SIAM J. Comput. 15 (1986) 1054–1068.

[5] A.T. Brint, P. Willett, Algorithms for the identification of three-dimensional maximal common substructures, J. Chem. Inform. Comput. Sci. 2 (1987) 311–320.

[6] C. Bron, J. Kerbosch, Algorithm 457 – finding all cliques of an undirected graph, Comm. ACM 16 (1973) 575–577.

[7] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory on NP-Completeness, W.H. Freeman and Company, San Francisco, CA, 1979.

[8] F. Gavril, Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of chordal graphs, SIAM J. Comput. 1 (1972) 180–187.

[9] L. Gerhards, W. Lindenberg, Clique detection for nondirected graphs: two new algorithms, Computing 21 (1979) 295–322.

[10] M. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press, New York, 1980.

[11] H.M. Grindley, P.J. Artymiuk, D.W. Rice, P. Willett, Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm, J. Mol. Biol. 229 (1993) 707–721.

[12] F. Harary, Graph Theory, R. Oldenburg Verlag, München - Wien, 1974.

[13] J. Håstad, Clique is hard to appromimate within $n^{1-\varepsilon}$, in: Proceedings of the 38th Ann. Symp. Found. Comp. Science – FOCS'96, IEEE, Burlington, Vermont, Washington, Brussels, Tokyo, 1996, pp. 627–633.

[14] W.-L. Hsu, Y. Ikura, G.L. Nemhauser, A polynomial algorithm for maximum weighted vertex packings on graphs without long odd cycles, Math. Programming 20 (1981) 225–232.

[15] H.J. Johnston, Cliques of a graph – variations on the Bron–Kerbosch algorithm, Int. J. Comput. Inform. Sci. 5 (1976) 209–238.

[16] I. Koch, T. Lengauer, Detection of distant structural similarities in a set of proteins using a fast graph-based method, in: T. Gaasterland, P. Karp, K. Karplus, C. Ouzounis, C. Sander, A. Valencia (Eds.), Proc. 5th Int. Conf. on Intelligent Systems for Molecular Biology, AAAI Press, Menlo Park, CA, 1997, pp. 167–178.

[17] I. Koch, T. Lengauer, E. Wanke, An algorithm for finding maximal common subtopologies in a set of protein structures, J. Comput. Biol. 3 (1996) 289–306.

[18] G. Levi, A note on the derivation of maximal common subgraphs of two directed or undirected graphs, Calcolo 9 (1972) 341–352.

[19] T. Madej, J.-F. Gibrat, S.H. Bryant, Threading a database of protein cores, PROTEINS:, Struct. Function Genetics 23 (1995) 356–369.

[20] K. Mizguchi, N. Gō, Comparison of spatial arrangements of secondary structural elements in proteins, Protein Eng. 8 (1995) 353–362.

[21] J.W. Moon, L. Moser, On Cliques in graphs, Israel. J. Math. 3 (1965) 23–28.

[22] V. Nicholson, C.-C. Tsai, M. Johnson, M. Naim, A subgraph isomorphism theorem for molecular graphs, in: R.B. King, D.H. Rouvray (Eds.), Graph Theory and Topology in Chemistry, Elsevier Science Publishers B.V., Amsterdam, 1987, pp. 226–230.

[23] R.E. Tarjan, A.E. Trojanowski, Finding a maximum independent set, SIAM J. Comput. 6 (1977) 537–546.

[24] H. Whitney, Congruent graphs and the connectivity of graphs, Amer. J. Math. 54 (1932) 150–168.