Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

DANIEL BAUM

Finding All Maximal Cliques of a Family of Induced Subgraphs

ZIB-Report 03-53 (December 2003)

Finding All Maximal Cliques of a Family of Induced Subgraphs

Daniel Baum

Abstract

Many real world problems can be mapped onto graphs and solved with well-established efficient algorithms studied in graph theory. One such problem is the following: given a set of objects and an irreflexive and symmetric relation between these objects, find maximal subsets whose elements mutually satisfy this relation. This problem can be transformed to the problem of finding all cliques of an undirected graph by mapping each object onto a vertex of the graph and connecting any two vertices by an edge whose corresponding objects satisfy the given relation.

In this paper we study a related problem, where all objects have a set of binary attributes, each of which is either 0 or 1. We want to find maximal subsets of objects not only mutually satisfying a given relation; but, in addition, all objects of a subset also need to have at least one common attribute with value 1. This problem can be mapped onto a set of induced subgraphs, where each subgraph represents a single attribute. For attribute i, its associated subgraph contains those vertices corresponding to the objects with attribute i set to 1.

We introduce the notion of a maximal clique of a family, $\mathcal{G}$, of induced subgraphs of an undirected graph, and show that determining all maximal cliques of $\mathcal{G}$ solves our problem. Furthermore, we present an efficient algorithm to compute all maximal cliques of $\mathcal{G}$. The algorithm we propose is an extension of the widely used Bron-Kerbosch algorithm [6].

Keywords: Bron-Kerbosch algorithm, maximal cliques, graph with vertex properties, family of induced subgraphs, backtracking algorithm, branch and bound technique, recursion.

Mathematics subject classification: 05C85, 68R10.

1 Introduction

A clique of an undirected graph is a complete subgraph which is not contained in any other complete subgraph. The problem of determining all cliques or the maximum clique of a graph is NP-complete. Clique-finding algorithms are used in many application areas [1, 2, 3, 4, 5, 9, 11, 12, 16]. Some applications involve the comparison of small sets of points, such as pattern matching [4], or molecular structure analysis [5, 11]. In these applications, clique detection is used to identify similar subsets of points. Here, one is not interested in *the* largest point set, but in diverse large point sets. These large sets, which can be seen as proposals, are then analysed in a second step. Undesired sets can be filtered out automatically, but still several proposals might remain which need to be looked at by an expert user.

In the following, we give an example from the field of molecular biology. Yet, we believe that our algorithm can be applied in many other fields. In pharmacophore model generation one is interested in those atoms that are most likely to cause the binding of some drug to a receptor. Given, e.g., two molecules that bind to a common receptor, we are looking for structural similarities in those two molecules. This can be done by constructing a correspondence graph in the following way [5]. For each pair of atoms, one from each molecule, we generate a new vertex in the correspondence graph, if the two atoms have similar molecular properties. Two vertices in the correspondence graph therefore correspond to four atoms, two in each molecule. We connect two vertices by an edge, if the two pairs of atoms have similar distances with respect to some distance threshold. The cliques of the correspondence graph represent molecular substructures with similar distance matrices. They also give a one-to-one mapping of the atoms of one molecule to the other. Since space-reflection symmetries cannot be distinguished via distance matrices, symmetric substructures need to be filtered out in a post-processing step.

Molecules, in particular small drug-like molecules, are flexible. It is therefore often not sufficient to compare molecules by considering only a single molecular structure. In order to overcome this limitation, we have developed an approach to compare several molecular structures in one go making use of the similarities between all structures of a molecule. Our approach leads to pseudo-molecules where each atom can belong to several structures. This introduces binary attributes, since either an atom belongs to a molecular structure or not. For each molecular structure we get one attribute. The whole approach will be described elsewhere.

In this paper, we concentrate on one rather technical aspect of our new approach, which, however, is crucial for its efficiency. We show that computing cliques of a graph with binary attributes is equal to computing all maximal cliques of a family of induced subgraphs, where a maximal clique is defined as a clique of some induced subgraph that is not contained in any clique of any other induced subgraph. In the following, if not otherwise stated, we use the term maximal clique when refering to the maximal clique of a family of induced subgraphs. The computation of all maximal cliques can be easily done by first computing all cliques of all subgraphs and then determining those cliques that are maximal. If we have N induced subgraphs and we assume that for each subgraph we get M cliques, we need $O((N-1)M^2)$ comparisons. This is very expensive in terms of computational cost. In this paper, we give experimental results that suggest that the computation of all maximal cliques can be done in almost linear time, $O(M)$, with respect to the number of cliques, using an extension of the widely used Bron-Kerbosch algorithm [6]. Our algorithmic extension computes all cliques of all subgraphs in parallel and only keeps the maximal ones.

The problem of determining all cliques of a graph has been widely studied [6, 7, 8, 10, 13, 14, 15]. In 1965, Moon and Moser [13] were the first to give an exact upper bound to the number of cliques in graphs, which is $3^{n/3}$, where n is the number of vertices. This means, there exist graphs whose number of cliques grows exponentially with the number of vertices. In 1973, Bron and

Kerbosch [6] proposed an algorithm to compute all cliques of a graph using a branch-and-bound technique. This algorithm proved to be much more efficient than previous ones and to date is one of the most efficient ones. Johnston [8] developed variations of the Bron-Kerbosch algorithms which differ in performance depending on the size and density of the graphs. Loukakis [10] and Johnson et al. [7] gave algorithms to compute all maximal independent sets of a graph, which are the cliques of the complement graph. In 1989, Tomita et al. [15] proved the worst-case optimality for an algorithm which they derived from the Bron-Kerbosch algorithm. In their review on the maximum clique problem [14], Pardalos et al. also give a good survey of algorithms for generating all cliques of a graph, including a few more algorithms.

In his article [10], Loukakis claims that his algorithm, LMIS, for generating all maximal independent sets, is approximately 10 times faster than the Bron-Kerbosch (BK) algorithm [6]. We implemented both algorithms and tested them on various random graphs of size 10 to 200 with densities ranging from 10% to 90%. We could not verify the results given by Loukakis. On the contrary, we found that the implementation of the BK algorithm was always superior to that of LMIS. These differences might be due to our implementations. However, we did not see how we could further improve the performance of LMIS. Due to these experiments and due to the fact, that the BK algorithm is rather easy to implement, we chose to extend the BK algorithm. We are certain, however, that other algorithms can be extended in a similar fashion. One disadvantage of the BK algorithm is that it needs $O(n^2)$ space. However, in general, for graphs for which space problems become an issue the run times are also inacceptable. Thus, the size of the graphs is restricted by both running time and space requirement, not only space requirement.

The rest of the paper is structured as follows. In section 2 we describe two algorithms, a basic version and the BK algorithm, to compute all cliques of an undirected graph. The reader familiar with the BK algorithm can skip this section and go directly to section 3, which explains the modifications necessary to compute all maximal cliques of a family of induced subgraphs. Computational results are given in section 4, and section 5 concludes the paper. In appendix A we give the pseudo code for the algorithm described in section 3.

2 Finding all Cliques of an Undirected Graph

In order to allow an easy understanding of our algorithmic extension (section 3), we decided to develop the algorithm in three steps. In section 2.1 we will recall the basic algorithm used for the BK algorithm. Here the basic principles used in all three algorithms are described. Section 2.2 describes the BK algorithm, which we will extend in section 3 to compute all maximal cliques. Sections 2.1 and 2.2 will closely follow the description given by Bron and Kerbosch in [6]. The BK algorithm uses a branch-and-bound technique to make the algorithm more efficient by cutting off branches of the search tree that will not lead to new cliques at a very early stage. The crucial part of our extension described in section 3 will therefore deal with the modification of this branch-and-bound technique.

Before continuing, we want to introduce some notations and terminology that is used throughout this paper.

- $G = (V, E)$ denotes an arbitrary *undirected graph*, where $V = \{1, 2, \ldots, n\}$ is the *set of vertices* of G, and $E \subseteq V \times V$ is the *set of edges* of G.
- A graph $G = (V, E)$ is called *complete*, if all its vertices are pairwise adjacent, i.e., $\forall i, j \in V, i \neq j : (i, j) \in E$.
- For a subset $S \subseteq V$, we call $G(S) = (S, E \cap S \times S)$ the *subgraph induced by S*.
- A subset $C \subseteq V$ is called *clique*, if its induced subgraph, $G(C)$, is complete and is not contained in any other complete subgraph of G. This definition of a clique is in slight contrast to the definition given earlier. From now on, we will refer to the definition given here.

In all of the following three variations of the algorithm, three sets of vertices play a major role.

- The first set is called CS (for *C*omplete *S*ubgraph), since the set of vertices in CS induces a complete subgraph of G. At any time during the execution of the backtracking algorithm, CS is the set to be extended by a vertex on branching, or to be reduced by a vertex on backtracking.
- The set CA (for *CA*ndidates), contains all vertices that will be used to extend CS towards a clique.
- The set NOT contains all vertices that were previously used to extend CS and are now explicitely excluded from the extension.

The statements concerning the three sets have to be slightly modified for section 3, but to avoid confusion we leave it at this for the moment. The two sets CA and NOT contain all vertices not contained in CS but adjacent to all vertices in CS. At recursion depth i, the sets CA and NOT are denoted by CA_i and NOT_i, respectively. In the following, let $G = (V, E)$ be the graph for which we want to compute all cliques.

2.1 Basic Clique Detection Algorithm

The core of the algorithm is a recursively defined extension operator that uses the three sets described above. A call of the operator generates all extensions of the current set CS by adding vertices from the set CA but not from the set NOT. The reason for excluding the vertices in NOT is that all extensions of CS containing those vertices have already been generated at an earlier stage of the algorithm. Initially, the sets CS and NOT_0 are set to empty sets, and CA_0 to V. Then, at recursion depth i, the extension operator performs the following five steps.

(1) If CA_i is empty, return,
else, take the first candidate, c, and remove it from CA_i.
(2) Add c to CS.
(3) Create new sets CA_{i+1} and NOT_{i+1} from the old sets CA_i and NOT_i by removing all vertices not adjacent to c, keeping the old sets intact.
(4) Call the extension operator to operate on the sets CS, CA_{i+1}, and NOT_{i+1}.
(5) Upon return, remove c from CS and add it to NOT_i. Go to (1).

We will now explain, what the set NOT is needed for. A necessary condition for the set CS to be a clique is that the set CA is empty. This is not sufficient, however, because if NOT still contains an element, CS cannot be maximal, since by adding any element from NOT to CS we obtain an even larger complete subgraph. This is due to our definition of the set NOT, that any element therein must be adjacent to all vertices in CS. To summarise, in order for CS to be a clique, both sets CA and NOT must be empty.

From this observation we can deduce the following bound condition for our algorithm. If at some stage of the algorithm, NOT contains a vertex, say v, adjacent to all vertices in CA, we know that calling the extension operator from this point will not lead to any new clique. The reason for this is that during all extensions from this point onwards v will remain in NOT, and thus, NOT can never become empty.

2.2 Bron-Kerbosch Algorithm

The basic algorithm described above generates all cliques in a lexicographic order. This is due to the way of selecting a new candidate from CA, i.e., we always take the first candidate. The Bron-Kerbosch algorithm chooses a new candidate such that the bound condition becomes true at the earliest possible stage. This means that we cut off all branches that do not lead to any new clique. We recall that the bound condition of our algorithm was formulated as: there exists a vertex in NOT which is adjacent to all vertices in CA. Achieving this at an early stage requires two things at each branching point.

(1) Determine the vertex v from $NOT \cup CA$ with the largest number of adjacent vertices in CA, or as Bron and Kerbosch put it, with the least number of non-adjacent vertices in CA.

(2) If v was found to be in CA, we take v as the next candidate. On backtracking of the extension operator, v is moved to NOT.

   At this point v is in NOT, and we continue by selecting those candidates first which are not adjacent to v. Note that this selection of candidates happens at the same branching point. If all non-adjacent vertices of v have been added to NOT the bound condition holds and we can backtrack.

Since the BK algorithm does not select candidates in lexicographic order, the cliques are also not generated in lexicographic order. Moreover, the algorithm tends to generate large cliques first. This is due to the applied branching strategy.

3 Finding All Maximal Cliques of a Family of Induced Subgraphs

Assume a set of objects with binary attributes or discrete properties, and an irreflexive and symmetric relation defined between pairs of objects. Our goal is to find maximal subsets of objects mutually satisfying the given relation with the additional constraint, that all objects of the subset have at least one attribute or property in common. This problem can be mapped to the problem of finding all maximal cliques of a family of induced subgraphs of a graph $G = (V, E)$, whereby we map each object to a vertex of G. An edge between two vertices in G exists if the given relation between their associated objects is satisfied. Each binary attribute can then be mapped to a separate induced subgraph of G.

Definition 3.1 *Given a graph $G = (V, E)$ and a set of binary vertex attributes $B_1, B_2, \ldots, B_m$, where $B_i : V \mapsto \{0, 1\}$, we can define sets $S_i \subseteq V$ as $S_i = \{v \in V \mid B_i(v) = 1\}$. Note, that the sets S_i are not necessarily disjoint.*

Remark 3.2 *Every set of discrete properties can be transformed into a set of binary attributes. Let $P : V \mapsto \{p_1, p_2, \ldots, p_l\}$ be a discrete property, then P can be mapped onto l binary attributes B_i of the form*

$$B_i(v) = \begin{cases} 1 & \text{if } P(v) = p_i \\ 0 & \text{otherwise} \end{cases} \quad , i = 1, \ldots, l.$$

Definition 3.3 (Maximal Clique) *Let $\mathcal{G} = \{G(S_1), G(S_2), \ldots, G(S_m)\}$ be the family of subgraphs of G induced by the sets S_i. Let further $\mathcal{C}_i$ be the family of all cliques of $G(S_i)$. Then a clique $C \in \mathcal{C}_i$ is a maximal clique of $\mathcal{G}$, if there does not exist a clique $\tilde{C} \in \mathcal{C}_j$, such that $\tilde{C} \supset C$. We denote the set of all maximal cliques of $\mathcal{G}$ by $MC = \{C \mid C$ is a maximal clique of $\mathcal{G}\}$.*

Definition 3.4 (Attributed Graph) *Let V be a set of vertices with $n = |V|$, let E be a set of edges, and let $B_1, B_2, \ldots, B_m$ be a set of binary vertex attributes. For each $v_i \in V$ we define a set A_i by $A_i = \{l \mid B_l(v_i) = 1\}$, which we call the attribute set of vertex v_i. We can now define the attributed graph $G_{\mathcal{A}}$ as the triple $(V, E, \mathcal{A})$, where $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$.*

Definition 3.5 *Let $G_{\mathcal{A}} = (V, E, \mathcal{A})$ be an attributed graph. A subset $C \subseteq V$ is called a clique of $G_{\mathcal{A}}$, if C satisfies the following three properties.*

(1) C induces a complete subgraph of the graph $G = (V, E)$.

(2) $\bigcap_{v_i \in C} A_i \neq \emptyset$.

(3) C is maximal in the sense that there does not exist a subset $\tilde{C} \subseteq V$, satisfying properties (1) and (2), such that $\tilde{C} \supset C$.

Remark 3.6 *Since the set $\bigcap_{v_i \in C} A_i$ is undefined for $C = \emptyset$, we define $\bigcap_{v_i \in C} A_i = \bigcup_{v_i \in V} A_i$, if $C = \emptyset$.*

Lemma 3.7 *Let $G = (V, E)$ be a graph and $B_1, B_2, \ldots, B_m$ a set of binary vertex attributes. Let $\mathcal{G}$ be the family of induced subgraphs of G associated with B_i, and let $G_{\mathcal{A}} = (V, E, \mathcal{A})$ be the attributed graph defined by G and B_i. Then a subset $C \subseteq V$ is a maximal clique of $\mathcal{G}$ if and only if C is a clique of $G_{\mathcal{A}}$.*

Proof: We prove Lemma 3.7 by showing both directions separately.

($\rightarrow$) Let C be a maximal clique of $\mathcal{G}$. We show that properties (1) through (3) of Definition 3.5 are satisfied.

(i) From Definition 3.3 follows the existence of an index j, such that C is a clique of $G(S_j)$, hence C induces a complete subgraph of G, satisfying property (1).

(ii) From Definition 3.1 and the statement above follows that $B_j(v) = 1$, $\forall v \in C$. From Definition 3.4 then follows that $j \in \bigcap_{v_i \in C} A_i$, and hence $\bigcap_{v_i \in C} A_i \neq \emptyset$, which satisfies property (2).

(iii) Let's assume C is not maximal. Then there exists a set $\tilde{C} \supset C$, satisfying (1) and (2). From (2) then follows the existence of an index $j \in \bigcap_{v_i \in \tilde{C}} A_i$ and hence $\tilde{C} \subseteq S_j$. From (1) we know that $\tilde{C}$ induces a complete subgraph of G. This means that either $\tilde{C}$ itself is a maximal clique of $\mathcal{G}$ or $\tilde{C}$ is contained in an even larger set, which is a maximal clique of $\mathcal{G}$. Thus, C cannot be a maximal clique of $\mathcal{G}$ which contradicts our choice of C. Hence, property (3) is also satisfied and we have shown that C is a clique of $G_{\mathcal{A}}$.

($\leftarrow$) Let C be a clique of $G_{\mathcal{A}}$. Because of property (1) we only need to show that C is a contained in any S_j and that C is maximal.

(i) From property (2) and Definition 3.4 follows the existence of an index $j \in \bigcap_{v_i \in C} A_i$, such that $B_j(v) = 1$, $\forall v \in C$, and hence $C \subseteq S_j$.

(ii) Let's assume C is not a maximal clique of $\mathcal{G}$, i.e., $\exists \tilde{C} \supset C$, such that $\tilde{C}$ is a maximal clique of $\mathcal{G}$. This means, $\tilde{C}$ induces a complete subgraph of G, and hence, property (1) is satisfied for $\tilde{C}$. Property (2) must be satisfied as well, since from (i) we know that $\exists j : \tilde{C} \subseteq S_j$, and thus $j \in \bigcap_{v_i \in C} A_i$, hence $\bigcap_{v_i \in C} A_i$ is not empty. Since properties (1) and (2) are satisfied for $\tilde{C}$ and also $\tilde{C} \supset C$, property (3) is not true for C, and hence, C cannot be a clique of $G_{\mathcal{A}}$, which contradicts our choice of C. Hence, our assumption must be wrong and C must be a maximal clique of $\mathcal{G}$. □

From Lemma 3.7 directly follows that we can compute MC by computing all cliques of $G_{\mathcal{A}}$, which, as we will show in the following section, can be done without computing all cliques of all induced subgraphs, and hence, is much more efficient.

3.1 Extension of the Bron-Kerbosch Algorithm

As mentioned at the beginning of section 2, the definition of the sets CS, NOT, and CA need to be modified to fit the definition of a clique of $G_{\mathcal{A}}$. For the set CS it is not sufficient to induce a complete subgraph of G, but at any time it must also be true that $\bigcap_{v_i \in CS} A_i \neq \emptyset$. Similarly, the vertices in NOT and CA not only need to be adjacent to all vertices in CS, but it must also hold that $\forall v_j \in NOT \cup CA : (\bigcap_{v_i \in CS} A_i) \cap A_j \neq \emptyset$.

Having redefined the sets CS, NOT, and CA as stated above, the algorithm for finding all cliques of $G_{\mathcal{A}}$ basically works as the algorithm described in section 2.1. However, the bound condition, which is the crucial part of our extension, still needs to be modified.

Remark 3.8 *We say that a vertex v_i is adjacent to a vertex v_j with respect to attribute l, if v_i is adjacent to v_j in G and $l \in A_i \cap A_j$. Similarly, v_i is non-adjacent to v_j with respect to attribute l, if either v_i is non-adjacent to v_j in G or $l \notin A_i \cap A_j$.*

Lemma 3.9 (Bound Condition) *We can now formulate the bound condition for the algorithm to compute all cliques of $G_{\mathcal{A}}$ as follows. If at some stage of the algorithm NOT contains a vertex, say v_j, adjacent to all vertices in CA with respect to some attribute $l \in \bigcap_{v_i \in CS} A_i$, calling the extension operator from this point will not lead to any new clique.*

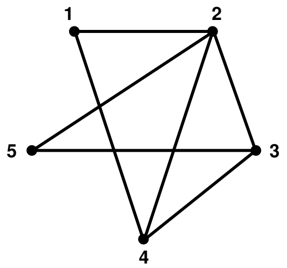

Figure 1: A simple graph. Let $\mathcal{S} = \{S_1 = \{1,2,5\}, S_2 = \{2,3,4\}, S_3 = \{1,3,4,5\}\}$, i.e., $A_1 = \{1,3\}$, $A_2 = \{1,2\}$, $A_3 = \{2,3\}$, $A_4 = \{2,3\}$, and $A_5 = \{1,3\}$. Then the graph $G_\mathcal{A}$ contains the cliques $\{2,3,4\}$, $\{2,1\}$, $\{2,5\}$, $\{3,5\}$, and $\{4,1\}$.

Proof: Let $v_j \in NOT$ be adjacent to all vertices in CA with respect to index $l \in \bigcap_{v_i \in CS} A_i$. Now, let's assume there exists a clique C of $\mathcal{G}_\mathcal{A}$ which can be found by calling the extension operator from this point. Then it must be true that $CS \subset C \subseteq (CS \cup CA)$. From

$$l \in \bigcap_{v_i \in CS} A_i \quad \text{and} \quad l \in A_j \cap (\bigcap_{v_i \in CA} A_i) \quad \text{follows}$$

$$l \in (\bigcap_{v_i \in CS} A_i) \cap A_j \cap (\bigcap_{v_i \in CA} A_i) \quad .$$

Since $C \subseteq (CS \cup CA)$, it must be true that

$$(\bigcap_{v_i \in C} A_i) \cap A_j \supseteq (\bigcap_{v_i \in CS} A_i) \cap A_j \cap (\bigcap_{v_i \in CA} A_i) \quad \text{, and hence}$$

$$l \in (\bigcap_{v_i \in C} A_i) \cap A_j = (\bigcap_{v_i \in C \cup \{v_j\}} A_i) \neq \emptyset.$$

Thus, for $C \cup \{v_j\}$ property (2) is satisfied. Since C induces a complete subgraph of G, and v_j is adjacent to all vertices in C, $C \cup \{v_j\}$ also induces a complete subgraph of G, and hence, property (1) is also satisfied for $C \cup \{v_j\}$. This means, property (3) cannot be satisfied for C, since clearly $C \cup \{v_j\} \supset C$. Hence, C cannot be a clique of $\mathcal{G}_\mathcal{A}$, which completes the proof of Lemma 3.9. $\square$

Remark 3.10 *Note, that the following bound condition is wrong. If at some stage of the algorithm, NOT contains a vertex, say v_i, and for each vertex v_j in CA there exists an attribute $l \in \bigcap_{v_k \in CS} A_k$, such that v_i is adjacent to v_j with respect to attribute l, calling the extension operator from this point will not lead to any new clique.*
In order to see this, let's look at the simple example in figure 1. After generating cliques $\{2,3,4\}$, $\{2,1\}$, and $\{2,5\}$, the algorithm would backtrack until $CS = \emptyset$, i.e., vertex 2 will be moved to NOT. Now vertex 2 is adjacent to each vertex in CA and also has at least one attribute in common with each vertex. Therefore the algorithm would stop at this point and the two cliques $\{3,5\}$ and $\{4,1\}$ would not be found.

	CS	CA	NOT	$(\bigcap_{v_i \in CS} A_i)$	clique
(1)	$\emptyset$	$\{1,2,3,4,5\}$	$\emptyset$	$\{1,2,3\}$	–
(2)	$\{2\}$	$\{1,3,4,5\}$	$\emptyset$	$\{1,2\}$	–
(3)	$\{2,3\}$	$\{4\}$	$\emptyset$	$\{2\}$	–
(4)	$\{2,3,4\}$	$\emptyset$	$\emptyset$	$\{2\}$	$\{2,3,4\}$
(5)	$\{2,3\}$	$\emptyset$	$\{4\}$	$\{2\}$	–
(6)	$\{2\}$	$\{1,4,5\}$	$\{3\}$	$\{1,2\}$	–
(7)	$\{2,1\}$	$\emptyset$	$\emptyset$	$\{1\}$	$\{2,1\}$
(8)	$\{2\}$	$\{4,5\}$	$\{3,1\}$	$\{1,2\}$	–
(9)	$\{2,5\}$	$\emptyset$	$\emptyset$	$\{1\}$	$\{2,5\}$
(10)	$\{2\}$	$\{4\}$	$\{3,1,5\}$	$\{1,2\}$	–
(11)	$\emptyset$	$\{1,3,4,5\}$	$\{2\}$	$\{1,2,3\}$	–
(12)	$\{3\}$	$\{4,5\}$	$\{2\}$	$\{2,3\}$	–
(13)	$\{3,5\}$	$\emptyset$	$\emptyset$	$\{3\}$	$\{3,5\}$
(14)	$\{3\}$	$\{4\}$	$\{2,5\}$	$\{2,3\}$	–
(15)	$\emptyset$	$\{1,4\}$	$\{2,5,3\}$	$\{1,2,3\}$	–
(16)	$\{4\}$	$\{1\}$	$\{2,3\}$	$\{2,3\}$	–
(17)	$\{4,1\}$	$\emptyset$	$\emptyset$	$\{3\}$	$\{4,1\}$

Table 1: Execution of our algorithm for the simple graph of figure 1. At lines (10) and (14) the algorithm backtracks because vertices 3 and 2, respectively, are adjacent to all vertices in CA.

Let's define the set $ATTR = \bigcap_{v_i \in CS} A_i$. One question that remains to be answered is, which branching strategy we need to apply for a vertex in NOT satisfying the bound condition to come about at the earliest possible stage. Let us recall that in the BK algorithm we first determined the vertex with the least number of non-adjacent vertices in $NOT \cup CA$. Here, instead of determining the overall number of non-adjacent vertices in $NOT \cup CA$, for each attribute $i \in ATTR$ we determine the number of non-adjacent vertices in $NOT \cup CA$ with respect to i. We fix both the vertex, say v, and the attribute, say l, with the least number of non-adjacent vertices in $NOT \cup CA$ with respect to l. If v is in CA we select v as next candidate. On backtracking, v is added to NOT. If v already was in NOT or has finally been added to NOT, we take those vertices in CA as new candidates first that are not adjacent to v with respect to attribute l. If all of these vertices have been removed from CA, all vertices remaining in CA are adjacent to v with respect to attribute l. Hence, no new clique can be generated from this point, since adding the vertex v to any set CS generated from this point will form a clique of larger size. Table 1 gives an account of the execution of our algorithm for the graph in figure 1. Pseudo code of the algorithm can be found in appendix A.

4 Results

We implemented our algorithmic extension in C++ and performed tests on randomly generated graphs. Several parameters could be varied in our tests. First of all there is the size of the graph $G = (V, E)$, given by the number of vertices, i.e., $|V|$. Second, there is the density D of G, given by $D = 2|E|/(|V|(|V| - 1)$. Third, we have to decide on the number of induced subgraphs of G. And fourth, we have the size of the subgraphs, that we need to make a decision on. In all

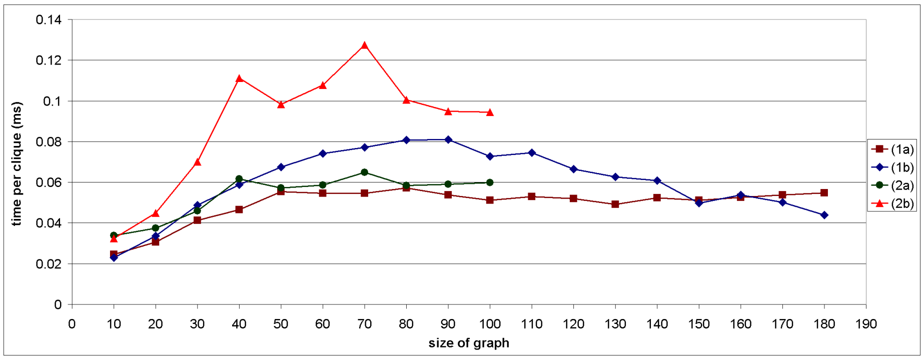

Figure 2: The computation time per clique is plotted against the size of the graph. The times are averaged over several graph densities in two different ways, (*a) and (*b), which are explained in the text. The graphs (1*) and (2*) correspond to subgraph sizes of 30% and 50%, respectively.

our tests we fixed the number of subgraphs to 32. Concerning the size of the subgraphs, we only made tests with subgraphs of the same size and either fixed the size to 30% or 50%.

4.1 Comparison with Bron-Kerbosch algorithm

In order to compare our algorithm with the BK algorithm, we accomplished tests similar to the first test case in [6]. We did not perform tests with the so called Moon-Moser graphs [13], i.e., graphs with $3^{n/3}$ cliques.

For two different subgraph sizes, 30% and 50%, we computed all maximal cliques for graphs with size ranging from 10 to 180 and 10 to 100, respectively, and densities with values from 10% to 90% with a step size of 10%. We then generated the averaged computation time per clique for a particular graph size in two ways.

(1) For each density we computed the averaged time per clique and then averaged these times again over the densities from 10% to 90%.

(2) We computed the overall number of cliques and the overall time for all densities, and then averaged the time over all cliques.

From the plot in figure 2 we can deduce that the time per clique is hardly dependent on the size of the graph. This is in accordance with the results given by Bron and Kerbosch in [6]. However, one might argue that averaging over a family of densities is not very meaningful. We therefore performed tests for several fixed densities in which we only varied the size of the graph.

For a subgraph size of 30%, the results can be seen in figure 3. One can observe a slight but steady increase in the time per clique for densities 10%, 30%, and 50%. For a density of 70% the time per clique increases at first, but then stays pretty much constant. Finally, for a density of 90% we observe a very rapid increase in time per clique up to a graph size of 50 vertices and then an equally rapid decrease. The results for a subgraph size of 50% were similar.

From the tests we have made, we can conclude that the computation time per clique hardly depends on the size of the graph. But even if we admitted

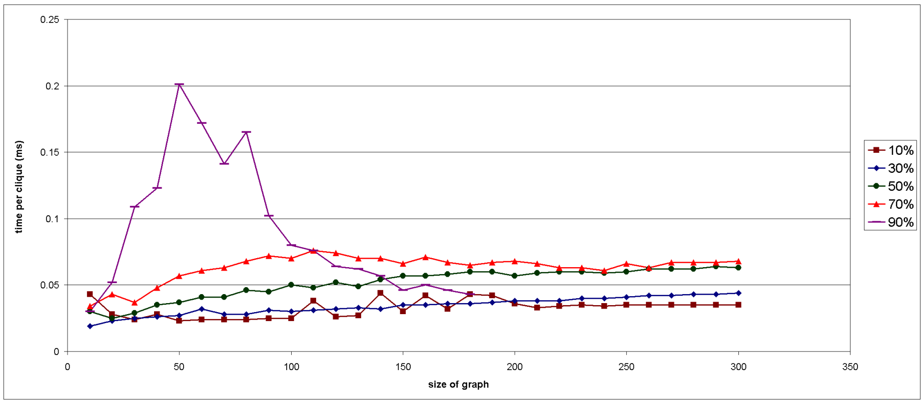

Figure 3: Results for a subgraph size of 30%. For densities of 10%, 30%, 50%, 70%, and 90% the time per clique is plotted against the size of the graph. For the density of 90% there was no data available for a graph size greater than 180 due to the large memory requirement.

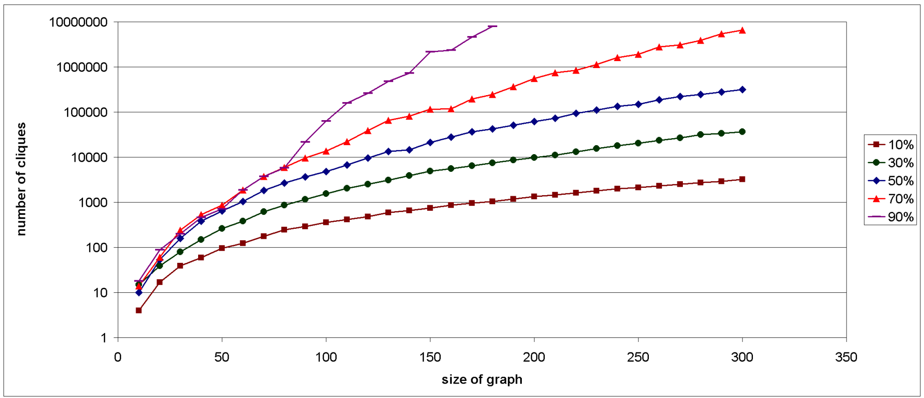

Figure 4: Number of cliques plotted against the size of the graph for a subgraph size of 30% and densities from 10% to 90%. Note the logarithmic scale of the y-axis.

a small increase of the computation time per clique with the graph size, the increase would only be of minor impact compared to the increase of the number of cliques, which is exponential. Figure 4 shows the rapid increase of the number of cliques for different densities.

4.2 Computation of all Maximal Cliques via all cliques of all Subgraphs

In order to compare our results with the approach to compute all cliques of all subgraphs first, we implemented a straight-forward algorithm. We initialized the set of maximal cliques with the set of cliques of the first subgraph. We then iteratively modified the current set of maximal cliques by looking at all the cliques of each remaining subgraph in turn.

- If a clique is contained in any of the cliques of the current set of maximal

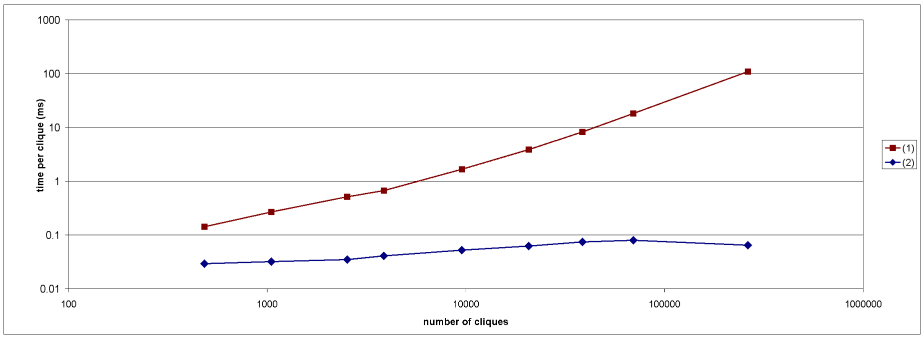

Figure 5: Time per clique plotted against the number of cliques for a graph size of 120 and a subgraph size of 30%. The data points correspond to densities of 10% to 90%. Note, that both axes use a logarithmic scale. Plot (1) shows the times for computing all maximal cliques via all cliques of all subgraphs. Plot (2) represents the new approach presented in this paper.

cliques, it is discarded.

- If a clique contains any current maximal clique, we replace this maximal clique by the new clique. We then need to check all maximal cliques not yet tested, whether they are also contained in the new clique.

Figure 5 shows our results. One can observe, that the time per clique grows linearly or even super-linearly with the number of cliques. This is in accordance with the theoretical computation time, that the time of the algorithm grows quadratically with the number of cliques. We are certain, that we can improve the running time of this approach, but we believe that in general it will be much slower than our approach.

5 Conclusion

The contributions of this paper are as follows. First, we introduced the notion of maximal cliques of a family of induced subgraphs. Second, we showed that all maximal cliques of a family of induced subgraphs can be computed directly by computing all cliques of a special graph where each vertex is associated with an attribute set. Third, we extended the algorithm by Bron and Kerbosch [6] to compute all maximal cliques of a family of induced subgraphs and showed that this extension is as efficient as the original algorithm.

Acknowledgments. I would like to thank Johannes Schmidt-Ehrenberg for many helpful discussions, and Marcus Weber and Frank Cordes for valuable hints concerning the paper.

References

[1] Dukka Bahadur K.C., Tatsuya Akutsu, Etsuji Tomita, Tomokazu Seki, and Asao Fujiyama. Point matching under non-uniform distortions and protein side chain packing based on an efficient maximum clique algorithm. *Genome Informatics*, 13:143–152, 2002.

[2] D. H. Ballard and M. Brown. *Computer Vision.* Prentice-Hall, Englewood-Cliffs, 1982.

[3] F. Barahona, A. Weintraub, and R. Epstein. Habitat dispersion in forest planning and the stable set problem. *Operations Research*, 40:14–21, 1992.

[4] H. G. Barrow and R. M. Burstall. Subgraph isomorphism, matching relational structures and maximal cliques. *Information Processing Letters*, 4(4):83–84, 1976.

[5] Andrew T. Brint and Peter Willett. Algorithms for the identification of three-dimensional maximal common substructures. *J. Chem. Inf. Comput. Sci.*, 27:152–158, 1987.

[6] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[7] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27:119–123, 1988.

[8] H. C. Johnston. Cliques of a graph – variations on the Bron-Kerbosch algorithm. *International Journal of Computer and Information Sciences*, 5(3):209–238, 1976.

[9] J. E. Lecky, O. J. Murphy, and R. G. Absher. Graph theoretic algorithms for the PLA folding problem. *IEEE Transactions on Computer Aided Design*, 8(9):1014–1021, 1989.

[10] E. Loukakis. A new backtracking algorithm for generating the family of maximal independent sets of a graph. *Comp. & Maths. with Appls.*, 9:583–589, 1983.

[11] Y. Martin, M. Bures, E. Danaher, J. DeLazzer, and I. Lico. A fast new approach to pharmacophore mapping and its application to dopaminergic and benzodiazepine agonists. *Journal of Computer-Aided Molecular Design*, 7:83–102, 1993.

[12] W. Miller. Building multiple alignments from pairwise alignments. *Computer Applications in the Biosciences*, 9:169–176, 1993.

[13] J. W. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3:23–28, 1965.

[14] Panos M. Pardalos and Jue Xue. The maximum clique problem. *Journal of Global Optimization*, 4:301–328, 1994.

[15] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. An optimal algorithm for finding all the cliques. *Technical Report of IPSJ SIG Algorithms, No.012 - 013*, pages 91–98, 1989.

[16] Martin Vingron and Pavel A. Pevzner. Multiple sequence comparison and n-dimensional image reconstruction. In *Combinatorial Pattern Matching, 4th Annual Symposium, CPM 93*, volume 684, pages 243–253, 1993.

A Pseudo Code

Algorithm A.1: COMPUTEALLMAXIMALCLIQUES($V, E, \mathcal{A}$)

```
comment: Create new sets and call extension operator.
procedure EXTENDCSWITHNEWCAND(CS, NOT_d, CA_d, ATTR_d, v_j)
  CS ← CS ∪ {v_j}
  ATTR_{d+1} ← ATTR_d ∩ A_j
  comment: Fill new set NOT_{d+1}.
  NOT_{d+1} ← ∅
  for each w ∈ NOT_d
    do { if w is adjacent to v_j with respect to any l ∈ ATTR_{d+1}
           then NOT_{d+1} ← NOT_{d+1} ∪ {w}
  comment: Fill new set CA_{d+1}.
  CA_{d+1} ← ∅
  for each w ∈ CA_d
    do { if w is adjacent to v_j with respect to any l ∈ ATTR_{d+1}
           then CA_{d+1} ← CA_{d+1} ∪ {w}
  comment: Check whether there still exist candidates.
  if CA_{d+1} ≠ ∅
    then { comment: Call extension operator with new sets.
           EXTENDCS(CS, NOT_{d+1}, CA_{d+1}, ATTR_{d+1})
           CS ← CS \ {v_j}
    else if NOT_{d+1} = ∅
    then { comment: New clique found.
           output clique CS


comment: Recursively extend CS to generate all cliques containing CS.
procedure EXTENDCS(CS, NOT_d, CA_d, ATTR_d)
  comment: minNod is the current minimum number of disconnections.
  minNod ← size of CA_d
  comment: Determine v_j with least number of non-adjacent vertices in CA_d with respect to any l ∈ ATTR_d.
  for each v_i ∈ NOT_d ∪ CA_d
    do { for each l ∈ ATTR_d ∩ A_i
           do { nod ← 0
                for each w ∈ CA_d
                  do { if w is not adjacent to v_j with respect to l
                         then nod ← nod + 1
                       if nod = minNod
                         then break
                if nod < minNod
                  then { v_j ← v_i
                         minNod ← nod
                         minAttr ← l
  comment: If v_j is in set CA_d, it will be itself taken as new candidate.
  if v_j ∈ CA_d
    then EXTENDCSWITHNEWCAND(CS, NOT_d, CA_d, ATTR_d, v_j)
  comment: Choose new candidates to extend CS.
  for each w ∈ CA_d
    do { if w is not adjacent to v_j with respect to minAttr
           then EXTENDCSWITHNEWCAND(CS, NOT_d, CA_d, ATTR_d, w)


comment: Compute all cliques of G_A = (V, E, A).
main
  comment: Initialize sets.
  CS ← ∅
  NOT_0 ← ∅
  CA_0 ← V \ {v_i | v_i ∈ V with A_i = ∅}
  ATTR_0 ← ⋃_{v_i ∈ CA_0} A_i
  comment: Call extension operator.
  EXTENDCS(CS, NOT_0, CA_0, ATTR_0)
```