Computing All Maximal Clique Partitions in a Graph

Temur Kutsia and Cleo Pau

RISC, Johannes Kepler University Linz, Austria

Abstract. A clique in a graph is a set of its vertices which are all connected to each other. A clique partition of a graph is a set of its disjoint cliques, which cover all the vertices of the graph. A partition is maximal, if no two cliques in it can be joined into a bigger one. Computing one maximal clique partition is a well-studied problem and several algorithms exist for it.

In this paper, we consider the problem of computing all maximal clique partitions. We start from all maximal cliques of a graph and proceed by trying to make them disjoint, assigning shared vertices to one of the cliques they belong to. The challenge is to compute only the needed solutions. Our algorithm returns only maximal partitions, and each of them is computed once. Tests that detect failing branches early, guarantee that no false answer is first computed and then discarded. These properties make the branches of the algorithm's search space independent from each other. Therefore, the process is easily parallelizable. Choosing shared vertices by some heuristics, known, e.g., from maximal clique partition algorithms, one can compute preferred partitions earlier than the others, and also stop computation after obtaining a certain number of solutions.

The algorithm can be used in problems such as anti-unification with proximity relations or in the resource allocation tasks, when one looks for several alternative ways to allocate resources.

1 Introduction

We consider undirected graphs $G = (V, E)$, where V is the set of vertices and E is the set of edges. A *clique* in a graph is a set of vertices $W \subseteq V$ such that for each pair of vertices in W there is an edge in E. A clique is *maximal* if it is not a proper subset of another clique.

A *clique partition* of a graph G is a set of its cliques $\{C_1, \ldots, C_n\}$ such that $\bigcup_{i=1}^{n} C_i = V$ and $C_i \cap C_j = \emptyset$ for all $1 \leq i, j \leq n$, $i \neq j$.

Let $S_1 = \{C_1, \ldots, C_n\}$ and $S_2 = \{D_1, \ldots, D_m\}$ be two sets of cliques of the same graph. We say that S_1 is *subsumed* by S_2, written $S_1 \sqsubseteq S_2$, iff for all $1 \leq i \leq n$ there exists $1 \leq j \leq m$ such that $C_i \subseteq D_j$. If S_1 and S_2 are, in particular, partitions, then we also say that S_1 is a *subpartition* of S_2 if S_1 is subsumed by S_2.

A clique partition of a graph is *maximal* if it is not (properly) subsumed by another partition.

A graph may have several maximal clique partitions. In the literature, a problem that was studied intensively is to compute a maximal clique partition with the smallest number of cliques. Tseng's algorithm [14], introduced to solve this problem, was motivated by its application in the design of processors. Later, Bhasker and Samad [5] proposed two other algorithms. They also derived the upper bound on the number of cliques in a partition and showed that there exists a partition containing a maximal clique of the graph.

A problem closely related to clique partition is the vertex coloring problem [8], which requires to color the vertices of a graph in such a way that two adjacent vertices have different colors. In fact, a clique-partitioning problem of a graph is equivalent to the coloring problem of its complement graph. Both problems are NP-complete [9].

In this paper, we are interested in computing all maximal clique partitions in a graph. The original motivation comes from anti-unification with proximity relations. Anti-unification is a well-known technique in computational logic. It was introduced in [10, 11] and has been quite intensively investigated in the last years, see, e.g. [1–4]. Given two logic terms t_1 and t_2, it aims at computing a least general generalization of those terms. That means, one is looking for a term s from which t_1 and t_2 can be obtained by variable substitutions. Such an s is called a generalization of t_1

and t_2. Moreover, there should be no other generalization r of t_1 and t_2, which can be obtained from s by a substitution. For instance, if the terms are $f(a,a)$ and $f(b,b)$, then anti-unification computes their least general generalization $f(x,x)$. Replacing x by a (resp. by b) in it, one gets $f(a,a)$ (resp. $f(b,b)$). Note that $f(x,y)$ and just x are also generalizations of $f(a,a)$ and $f(b,b)$, but they are not least general. Anti-unification has been successfully used in inductive reasoning, inductive logic programming, reasoning and programming by analogy, term set compression, software code clone detection, etc.

In modern applications of anti-unification, one often has to deal with imprecise or vague information. In such circumstances, one tends to consider two objects the same, if they are "sufficiently close" to each other. However, such a proximity relation is not transitive: For instance, if one considers two cities being close to each other if the distance between them is not more than 200 km, then Salzburg is close to Linz (133 km) and Linz is close to Vienna (185 km), but Salzburg is not close to Vienna (300 km). Nontransitivity has to be dealt in a special way. Proximity relations (reflexive symmetric fuzzy binary relations) characterize the notion of 'being close' numerically. They become crisp once we fix the threshold from which on the distance between the objects can be called 'close'.

When one considers a proximity relation over the alphabet of a language, anti-unification problems do not have a single least general generalization anymore. For instance, if a is close to both b and c, but b and c are not close to each other, then $f(a,a)$ and $f(b,c)$ have two minimal common generalizations: $f(a,x)$ and $f(x,a)$. In such cases, one is interested in computing a minimal complete set of generalizations. This problem is closely related to computing all maximal clique partitions in an undirected graph. A (crisp version of a) proximity relation on symbols of the language can be modeled by such a graph. In this example, the graph would be $(\{a,b,c\},\{(a,b),(a,c)\})$. It has two maximal clique partitions $\{\{a,b\},\{c\}\}$ and $\{\{a,c\},\{b\}\}$ that tell exactly which symbols should be considered the same. In the first case these are a and b, leading to the generalization $f(a,x)$, and in the second case they are a and c, giving $f(x,a)$.

In a lighter view, all maximal clique partitions can be useful in designing a menu of various possible courses cooked with (leftover) ingredients which are not enough to be used in more than one dish. Parents of kids who are reluctant to eat would appreciate such a choice. In general, the resource allocation problem, when one looks for several alternative ways to allocate resources, can be an application area of the algorithm considered in this paper.

For a given graph, in order to compute its all maximal clique partitions, we use a kind of top-down approach. First, we compute a set of all maximal cliques of the graph. For this problem, there are several known algorithms, e.g. [6, 7, 12, 13]. After that, the idea is to keep each shared vertex only in one of the computed clique so that the obtained partition is maximal. Our algorithm computes all maximal clique partitions and is optimal in the following sense: Each maximal clique partition is computed only once, and generating and discarding false answers is avoided. If one does not want to get all the solutions, he/she can stop the algorithm after computing certain number of solutions, for instance. Besides, by imposing heuristics on the choice of shared vertices (to be assigned to a clique) one can give a priority to computing certain kind of partitions over the others (e.g., those containing a maximal clique, or those having a minimal number of cliques, etc.)

2 The Algorithm

For a graph G, we denote by *all-max-cliques*(G) the set of all its maximal cliques. We start with computing this set and give each of its element a name. For the graph in Fig. 1, there are four of them: $C_1 = \{1,2,3\}, C_2 = \{2,3,4\}, C_3 = \{4,5,6\}, C_4 = \{5,6,7\}$. These cliques will get revised

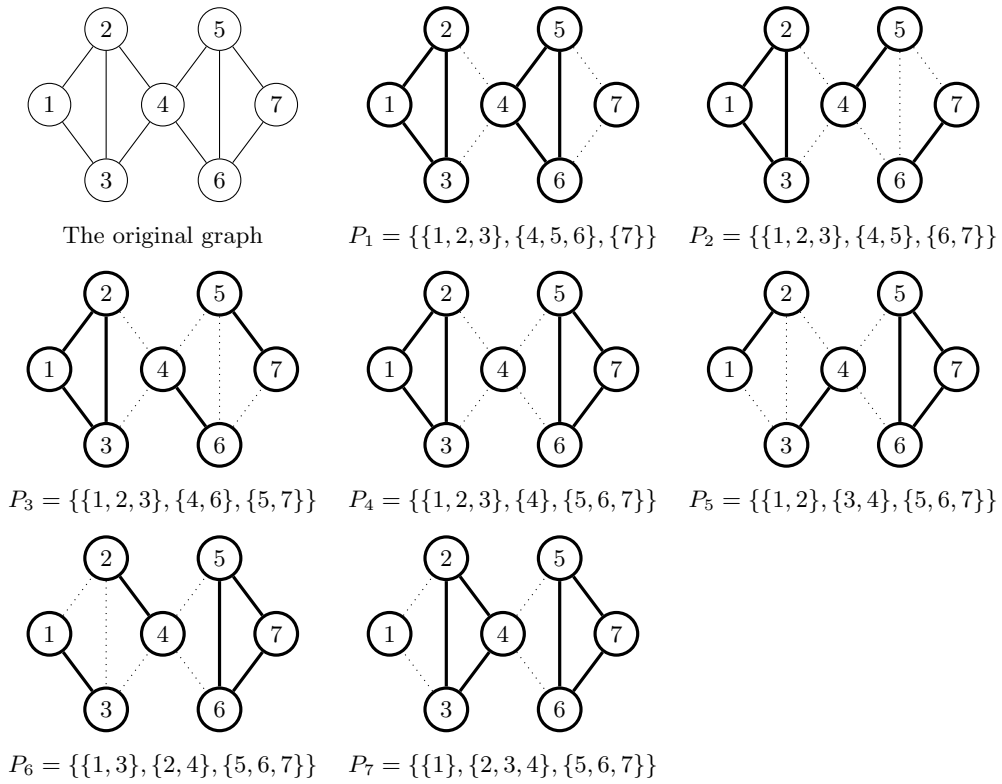

Fig. 1. All maximal clique partitions of a graph.

during computation by removing elements from them. At the end, we report those which are not empty.

After computing the initial cliques, we collect all shared vertices and indicate among which cliques they are shared. In the graph in Fig. 1, the shared vertices are 2, 3, 4, 5, and 6. We have $2 \in C_1 \cap C_2$, $3 \in C_1 \cap C_2$, $4 \in C_2 \cap C_3$, $5 \in C_3 \cap C_4$, and $6 \in C_3 \cap C_4$.

Our goal is to compute each solution exactly once. At the end, it can happen that some cliques consist of shared vertices only. Such cliques can have any of the names of the original cliques they originate from. For instance, the node 4 alone can form a clique either as C_2 or C_3, giving two identical partitions which differ only by the clique names:

$$C_1 = \{1,2,3\}, \quad C_2 = \{4\}, \quad C_3 = \emptyset, \quad C_4 = \{5,6,7\},$$
$$C_1 = \{1,2,3\}, \quad C_2 = \emptyset, \quad C_3 = \{4\}, \quad C_4 = \{5,6,7\}.$$

We want to avoid such duplicates. Therefore, for such alternatives we choose one single clique to which a shared vertex can belong in this configuration, and forbid the others. For the example graph in Fig. 1, we can allow the vertices 2 and/or 3 to form a clique as C_2, the vertex 4 to form a clique as C_3, and the vertices 5 and/or 6 to form a clique as C_4. (Note that allowing does not necessarily mean that we will get result cliques of that form. For instance, in C_4 we will have 7 as well.) Thus, the candidates for forbidden configurations are $C_1 \neq \{2\}, C_1 \neq \{3\}, C_1 \neq \{2,3\}, C_2 \neq \{4\}, C_3 \neq \{5\}, C_3 \neq \{6\}, C_3 \neq \{5,6\}$. This can be further simplified by observing that C_1 contains a non-shared vertex 1 and, hence, can not consist of shared vertices only. Therefore, we can omit the first three candidate disequations and obtain the forbidden configuration $C_2 \neq \{4\}, C_3 \neq \{5\}, C_3 \neq \{6\}, C_3 \neq \{5,6\}$.

Starting from the initial set of cliques, our algorithm All-Maximal-Clique-Partitions performs the following steps:

1. Compute the set of shared vertices and the forbidden configurations.
2. If the set of shared vertices is empty, return the current set of cliques and stop.
3. Select a shared vertex and nondeterministically assign it to one of the cliques it belongs to. Remove the vertex from the other cliques and from the set of shared vertices.
4. For each pair of cliques C_i, C_j, where $C_i \subseteq C_j$, make C_i empty and adjust the set of shared elements. In addition, if C_i was the chosen clique for the shared elements, remove those elements from the forbidden list of C_j.
5. If the union of two nonempty cliques is a subset of an original clique, or if a forbidden configuration arises, stop the development of this branch with failure. Otherwise go to step 2.

Checking for the subset relations is needed to avoid computing cliques which are not maximal. For instance, the partition $C_1 = \{1,2\}$, $C_2 = \{3\}$, $C_3 = \{4\}$, $C_4 = \{5,6,7\}$ should be rejected because $\{1,2\} \cup \{3\}$ is a subset of the original C_1 clique. Step 5 helps to detect such situations early.

The partitions shown in Fig. 1 correspond to the following final values of cliques, computed by the All-Maximal-Clique-Partitions algorithm:

$$
\begin{array}{lllll}
P_1: & C_1 = \{1,2,3\}, & C_2 = \emptyset, & C_3 = \{4,5,6\}, & C_4 = \{7\}. \\
P_2: & C_1 = \{1,2,3\}, & C_2 = \emptyset, & C_3 = \{4,5\}, & C_4 = \{6,7\} \\
P_3: & C_1 = \{1,2,3\}, & C_2 = \emptyset, & C_3 = \{4,6\}, & C_4 = \{5,7\}. \\
P_4: & C_1 = \{1,2,3\}, & C_2 = \emptyset, & C_3 = \{4\}, & C_4 = \{5,6,7\} \\
P_5: & C_1 = \{1,2\}, & C_2 = \{3,4\}, & C_3 = \emptyset, & C_4 = \{5,6,7\}. \\
P_6: & C_1 = \{1,3\}, & C_2 = \{2,4\}, & C_3 = \emptyset, & C_4 = \{5,6,7\} \\
P_7: & C_1 = \{1\}, & C_2 = \{2,3,4\}, & C_3 = \emptyset, & C_4 = \{5,6,7\}.
\end{array}
$$

Before proving the properties of the algorithm, we illustrate it with an example.

Example 1. Let G be the graph shown in Fig. 1. We start with set of all maximal cliques in it: $C_1 := \{1,2,3\}$, $C_2 := \{2,3,4\}$, $C_3 := \{4,5,6\}$, $C_4 := \{5,6,7\}$. The set of all shared vertices, represented as elements of intersections, is $shared := \{2 \in C_1 \cap C_2,\ 3 \in C_1 \cap C_2,\ 4 \in C_2 \cap C_3,\ 5 \in C_3 \cap C_4,\ 6 \in C_3 \cap C_4\}$, and the set of all forbidden configurations is $fc := \{C_2 \neq \{4\}, C_3 \neq \{5\}, C_3 \neq \{6\}, C_3 \neq \{5,6\}\}$. The steps of the algorithm below are shown as the nodes in the complete search tree, and they are enumerated as the positions of those nodes in the search tree.

1. Step 3: $2 \in C_1,\ C_2 := \{3,4\},\ shared := shared \setminus \{2 \in C_1 \cap C_2\}$.
Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

1.1 Step 3: $3 \in C_1,\ C_2 := \{4\},\ shared := shared \setminus \{3 \in C_1 \cap C_2\}$.
Step 4: $C_2 \subseteq C_4,\ C_2 := \emptyset,\ shared = shared \setminus \{4 \in C_2 \cap C_3\}$.
Step 5 does not apply. Go to Step 2. It does not apply.

1.1.1 Step 3: $5 \in C_3,\ C_4 := \{6,7\},\ shared := shared \setminus \{5 \in C_3 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

1.1.1.1 Step 3: $6 \in C_3,\ C_4 := \{7\},\ shared := shared \setminus \{6 \in C_3 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $shared = \emptyset$, **Return** $C_1 = \{1,2,3\}$, $C_2 = \emptyset$, $C_3 = \{4,5,6\}$, $C_4 = \{7\}$.

1.1.1.2 Step 3: $6 \in C_4,\ C_3 := \{4,5\},\ shared := shared \setminus \{6 \in C_3 \cap C_4\}$.

Step 4 and Step 5 do not apply. Go to Step 2.

Step 2: $shared = \emptyset$, **Return** $C_1 = \{1,2,3\}$, $C_2 = \emptyset$, $C_3 = \{4,5\}$, $C_4 = \{6,7\}$.

1.1.2 Step 3: $5 \in C_4,\ C_3 := \{4,6\},\ shared := shared \setminus \{5 \in C_3 \cap C_4\}$.

Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

1.1.2.1 Step 3: $6 \in C_3,\ C_4 := \{5,7\},\ shared := shared \setminus \{6 \in C_3 \cap C_4\}$.

Step 4 and Step 5 do not apply. Go to Step 2.

Step 2: $shared = \emptyset$, **Return** $C_1 = \{1,2,3\}$, $C_2 = \emptyset$, $C_3 = \{4,6\}$, $C_4 = \{5,7\}$.

1.1.2.2 Step 3: $6 \in C_4,\ C_3 := \{4\},\ shared := shared \setminus \{6 \in C_3 \cap C_4\}$.

Step 4 and Step 5 do not apply. Go to Step 2.

Step 2: $shared = \emptyset$, **Return** $C_1 = \{1,2,3\}$, $C_2 = \emptyset$, $C_3 = \{4\}$, $C_4 = \{5,6,7\}$.

1.2 Step 3: $3 \in C_2,\ C_1 := \{1,2\},\ shared := shared \setminus \{3 \in C_1 \cap C_2\}$.

Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

1.2.1 Step 3: $4 \in C_2,\ C_3 := \{5,6\},\ shared := shared \setminus \{4 \in C_2 \cap C_3\}$.

Step 4: $C_3 \subseteq C_4,\ C_3 := \emptyset,\ shared := shared \setminus \{5 \in C_3 \cup C_4,\ 6 \in C_3 \cup C_4\}$

Step 5 dos not apply. Go to Step 2.

Step 2: $shared = \emptyset$, **Return** $C_1 = \{1,2\}$, $C_2 = \{3,4\}$, $C_3 = \emptyset$, $C_4 = \{5,6,7\}$.

1.2.2 Step 3: $4 \in C_3,\ C_2 := \{3\},\ shared := shared \setminus \{4 \in C_2 \cap C_3\}$.

Step 4 does not apply.

Step 5: $C_1 \cup C_2 = \{1,2,3\}$ which is one of the original cliques. **Fail**.

2. Step 3: $2 \in C_2,\ C_1 := \{1,3\},\ shared := shared \setminus \{2 \in C_1 \cap C_2\}$.

Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

2.1 Step 3: $3 \in C_1,\ C_2 := \{2,4\},\ shared := shared \setminus \{3 \in C_1 \cap C_2\}$.

Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

2.1.1 Step 3: $4 \in C_2,\ C_3 := \{5,6\},\ shared := shared \setminus \{4 \in C_2 \cap C_3\}$.

Step 4: $C_3 \subseteq C_4,\ C_3 := \emptyset,\ shared := shared \setminus \{5 \in C_3 \cup C_4,\ 6 \in C_3 \cup C_4\}$

Step 5 dos not apply. Go to Step 2.

Step 2: $shared = \emptyset$, **Return** $C_1 = \{1,3\}$, $C_2 = \{2,4\}$, $C_3 = \emptyset$, $C_4 = \{5,6,7\}$.

2.1.2 Step 3: $4 \in C_3,\ C_2 := \{2\},\ shared := shared \setminus \{4 \in C_2 \cap C_3\}$.

Step 4 does not apply.

Step 5: $C_1 \cup C_2 = \{1,2,3\}$ which is one of the original cliques. **Fail**.

2.2 Step 3: $3 \in C_2,\ C_1 := \{1\},\ shared := shared \setminus \{3 \in C_1 \cap C_2\}$.

Step 4 and Step 5 do not apply. Go to Step 2. It does not apply.

2.2.1 Step 3: $4 \in C_2,\ C_3 := \{5,6\},\ shared := shared \setminus \{4 \in C_2 \cap C_3\}$.

Step 4: $C_3 \subseteq C_4,\ C_3 := \emptyset,\ shared := shared \setminus \{5 \in C_3 \cup C_4, 6 \in C_3 \cup C_4\}$

Step 5 dos not apply. Go to Step 2.

Step 2: $shared = \emptyset$, **Return** $C_1 = \{1\}$, $C_2 = \{2,3,4\}$, $C_3 = \emptyset$, $C_4 = \{5,6,7\}$.

2.2.2 Step 3: $4 \in C_3,\ C_2 := \{2,3\},\ shared := shared \setminus \{4 \in C_2 \cap C_3\}$.

Step 4 does not apply.

Step 5: $C_1 \cup C_2 = \{1,2,3\}$ which is one of the original cliques. **Fail**.

The returned partitions are exactly those shown in Fig. 1.

Example 2. Let G be the graph given by its all maximal cliques: $C_1 := \{1, 2\}$, $C_2 := \{1, 3\}$, $C_3 := \{3, 4\}$, $C_4 := \{2, 5\}$. The set of all shared vertices, represented as elements of intersections, is $shared := \{1 \in C_1 \cap C_2,\ 2 \in C_1 \cap C_4,\ 3 \in C_2 \cap C_3\}$, and the set of all forbidden configurations is $fc := \{C_1 \neq \{1\}, C_1 \neq \{2\}, C_2 \neq \{3\}\}$. The steps of the algorithm below are shown as the nodes in the complete search tree, and they are enumerated as the positions of those nodes in the search tree.

1. Step 3: $1 \in C_1,\ C_2 := \{3\},\ shared := shared \setminus \{1 \in C_1 \cap C_2\}$.
Step 4: $C_2 \subseteq C_3,\ C_2 := \emptyset,\ shared := shared \setminus \{3 \in C_2 \cap C_3\}$.
Step 5 does not apply. Go to Step 2. It does not apply.

1.1 Step 3: $2 \in C_1,\ C_4 := \{5\},\ shared := shared \setminus \{2 \in C_1 \cap C_4\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $shared = \emptyset$, **Return** $C_1 = \{1, 2\}$, $C_2 = \emptyset$, $C_3 = \{3, 4\}$, $C_4 = \{5\}$.

1.2 Step 3: $2 \in C_4,\ C_1 := \{1\},\ shared := shared \setminus \{2 \in C_1 \cap C_4\}$.
Step 4 do not apply.
Step 5: C_1 is forbidden by fc. **Fail**.

2. Step 3: $1 \in C_2,\ C_1 := \{2\},\ shared := shared \setminus \{1 \in C_1 \cap C_2\}$.
Step 4: $C_1 \subseteq C_4,\ C_1 := \emptyset,\ shared = shared \setminus \{2 \in C_1 \cap C_4\}$.
Step 5 does not apply. Go to Step 2. It does not apply.

2.1 Step 3: $3 \in C_2,\ C_3 := \{2\},\ shared := shared \setminus \{3 \in C_2 \cap C_3\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $shared = \emptyset$, **Return** $C_1 = \emptyset$, $C_2 = \{1, 3\}$, $C_3 = \{4\}$, $C_4 = \{2, 5\}$.

2.2 Step 3: $3 \in C_3,\ C_2 := \{1\},\ shared := shared \setminus \{3 \in C_2 \cap C_3\}$.
Step 4 and Step 5 do not apply. Go to Step 2.
Step 2: $shared = \emptyset$, **Return** $C_1 = \emptyset$, $C_2 = \{1\}$, $C_3 = \{3, 4\}$, $C_4 = \{2, 5\}$.

Hence, the algorithm computes three maximal clique partitions. One can see how the forbidden configuration prevented to compute the partition $C_1 = \{1\}$, $C_2 = \emptyset$, $C_3 = \{3, 4\}$, $C_4 = \{2, 5\}$ in the node **1.2**, which would be a duplicate of the partition computed in the node **2.2**.

Looking at the algorithm, one can easily notice that if we did not have the steps 4 and 5, we would compute all maximal clique partitions (since at Step 3 we assign shared vertices to one of the cliques they belong to). However, in addition, subpartitions of these partitions might also be generated. It might also happen that the same maximal partition is computed more than once. Therefore, we need to show that in steps 4 and 5 we eliminate exactly those subpartitions and duplicates.

Strictly speaking, one can omit Step 4 completely. In the subsequent splitting of shared vertices between C_i and C_j, if a shared vertex goes from C_j to C_i (or if forbidden configuration involving C_i arises), Step 5 will block the development of the branch, effectively imitating the behavior of Step 4, but doing it in several steps. Step 4 is there to reduce this extra work.

Theorem 1. *Each set of cliques computed by the* All-Maximal-Clique-Partitions *for the given graph is a maximal clique partition for the graph.*

Proof. Let $S = \{C_1, \ldots, C_n\}$ be a set of cliques of the given graph G the algorithm returns.

First, show that S is a clique partition of G. The algorithm works by removing vertices from a precomputed set of all maximal cliques of G. Hence, every element of each C_i is a vertex of G. During computation, no vertex gets lost: For Step 4 it is obvious, and for Step 5 it follows from the fact that if two cliques are included in a bigger original clique, then that bigger clique will appear on a neighboring branch and its elements will not get lost. For forbidden configurations it is also easy to see, because those vertices that are forbidden in one clique, are allowed in another. At the end of the computation, they either remain where they are allowed for a clique, or appear in the forbidden ones together with some other vertices. In any case, they are not lost.

Hence, no vertex of G is missing from S. Since the algorithm stops when there are no shared vertices, we get that $C_i \cap C_j = \emptyset$ for all $1 \leq i, j \leq n$, $i \neq j$ and, hence S is a clique partition of G.

Now we need to show that the partition is maximal. Assume by contradiction that it is not. Then there are $C_i, C_j \in S$ such that $C_i \cup C_j$ would be also a clique in G. But then $C_i \cup C_j$ is a subset of an maximal clique in the original clique set where the algorithm starts from. Therefore, Step 5 would block the development of the branch of the algorithm and S would not be computed. A contradiction.

Next, we show that the algorithm does not compute duplicates:

Theorem 2. All-Maximal-Clique-Partitions *computes each maximal clique partition exactly once.*

Proof. Since shared vertices are distributed to the cliques they belong (Step 3), the duplication may arise when the same set of shared vertices is collected in a clique with one name in one branch, and in a clique with a different name in another branch. However, the forbidden configurations prevent such cases. They declare which clique (identified by its name) may consist of which shared vertices only. Such clique names are uniquely determined. The same set of shared vertices can not form a clique with different names in different partitions. Therefore, the second part of Step 5 prevents to compute the same partition more than once.

The algorithm is complete:

Theorem 3. All-Maximal-Clique-Partitions *computes all (and only) maximal clique partitions.*

Proof. We need to prove that the steps 4 and 5 do not eliminate any maximal clique partitions.

Step 4 prevents to generate clique sets containing cliques of the form $C_i \setminus V$ and $C_j \setminus (C_i \setminus V)$ for $V \subset C_i$, where $C_i \subseteq C_j$. Such a clique set S_0 will be subsumed by a clique set S_1 obtained by taking $V = C_i$. Therefore, for each partition P_0 originating from S_0 there will be a partition P_1 originating from S_1 such that $P_0 \sqsubseteq P_1$. Hence, removing the execution branch which cuts S_0 will not eliminate any maximal clique partition of the given graph.

The first part of Step 5 prevents to generate clique sets containing nonempty cliques C_i and C_j such that $C_i \cup C_j \subseteq C$, where C is one of the original maximal cliques. Such a clique set S_0 will be subsumed by a clique set S_1 which retains C. Hence, no maximal clique partition is lost by eliminating S_0 and proceeding to compute partitions from S_1.

Forbidden configurations simply prevent the same partitions from reappearing under different names. Also here, there is no danger of losing a maximal partition.

Given a graph G, the clique-multiplicity number of a vertex v of G is the number of cliques in the set of of all maximal cliques of G to which v belongs: *clique-mult*$(v, G) = |\{C \mid C \in$ *all-max-cliques*(G) and $v \in C\}|$.

Theorem 4. *Let G be a graph with the set of vertices $\{v_1, \ldots, v_n\}$. Then the cardinality of the set of all maximal clique partitions of G is at most $\prod_{i=1}^{n}$ clique-mult(v_i, G).*

Proof. The result directly follows from the fact that shared vertices are distributed in their containing cliques in all possible ways.

It is easy to see that this upper bound can be reached. Just consider the graph with two maximal cliques: $C_1 = \{p_1, \ldots, p_n, \mathit{true}\}$ and $C_2 = \{p_1, \ldots, p_n, \mathit{false}\}$. The set of all maximal clique partitions imitates the truth assignment in propositional logic, containing 2^n maximal clique partitions.

This theorem implies that the algorithm is exponential in the number of vertices shared among multiple cliques. On the other hand, the length of each branch of the algorithm is polynomially bounded, since it requires at most as many steps as there are vertices shared among multiple cliques. Besides, the branches can be executed independently, in parallel of each other.

We made a prototype implementation of the algorithm in the programming language of Mathematica [15] (the Wolfram language). Some test examples can be seen online at

http://www.risc.jku.at/people/tkutsia/software/all-maximal-clique-partitions-examples.htm.

3 Conclusion

We developed an algorithm for computing all maximal clique partitions in an undirected graph. The algorithm starts from a set of all maximal cliques and revises it, reducing the number of shared vertices by assigning them to one of the cliques they belong to. In this process, we avoid computing and discarding false answers by detecting the failing branches early, and compute each partition only once. The set if computed partitions can be exponentially large with respect to the number of vertices shared among multiple cliques. Each partition can be computed in polynomial time (starting from all maximal cliques). The algorithm can be used to compute a limited number of maximal partitions. Guiding by heuristics for choosing shared nodes, one can give the priority to one kind of partitions over the others, computing the preferred ones earlier.

Acknowledgements. This work has been supported by Austrian Science Fund (FWF) under project 28789-N32.

References

1. Hassan Aït-Kaci and Gabriella Pasi. Lattice operations on terms with fuzzy signatures. *CoRR*, abs/1709.00964, 2017.
2. María Alpuente, Santiago Escobar, Javier Espert, and José Meseguer. A modular order-sorted equational generalization algorithm. *Inf. Comput.*, 235:98–136, 2014.
3. Alexander Baumgartner and Temur Kutsia. A library of anti-unification algorithms. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557. Springer, 2014.
4. Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. A variant of higher-order anti-unification. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 113–127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
5. Jayaram Bhasker and Tariq Samad. The clique-partitioning problem. *Computers and Mathematics with Applications*, 22(6):1–11, 1991.
6. Coenraad Bron and Joep Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM*, 16(9):575–576, 1973.
7. Frédéric Cazals and Chinmay Karande. A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407(1-3):564–568, 2008.

8. Tommy R Jensen and Bjarne Toft. *Graph coloring problems*, volume 39. John Wiley & Sons, 2011.
9. Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.
10. Gordon D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.
11. John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.
12. Etsuji Tomita. Efficient algorithms for finding maximum and maximal cliques and their applications. In Sheung-Hung Poon, Md. Saidur Rahman, and Hsu-Chun Yen, editors, *WALCOM: Algorithms and Computation, 11th International Conference and Workshops, WALCOM 2017, Hsinchu, Taiwan, March 29-31, 2017, Proceedings.*, volume 10167 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2017.
13. Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.
14. Chia-Jeng Tseng. *Automated synthesis of data paths in digital systems.* PhD thesis, Dept. of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburg, PA, 1984.
15. Stephen Wolfram. *The Mathematica book, 5th Edition.* Wolfram-Media, 2003.