# Data Reduction, Exact, and Heuristic Algorithms for Clique Cover

Jens Gramm[*]     Jiong Guo[†]     Falk Hüffner[†]     Rolf Niedermeier[†]

## Abstract

To cover the edges of a graph with a minimum number of cliques is an NP-complete problem with many applications. The state-of-the-art solving algorithm is a polynomial-time heuristic from the 1970's. We present an improvement of this heuristic. Our main contribution, however, is the development of efficient and effective polynomial-time data reduction rules that, combined with a search tree algorithm, allow for exact problem solutions in competitive time. This is confirmed by experiments with real-world and synthetic data. Moreover, we prove the fixed-parameter tractability of covering edges by cliques.

## 1 Introduction

Data reduction techniques for exactly solving NP-hard combinatorial optimization problems have proven useful in many studies [1, 3, 10, 16]. The point is that by polynomial-time executable reduction rules many input instances of hard combinatorial problems can be significantly shrunk and/or simplified, without sacrificing the possibility of finding an optimal solution to the given problem. For such reduced instances then often exhaustive search algorithms can be applied to efficiently find optimal solutions in reasonable time. Hence data reduction techniques are considered as a "must" when trying to cope with computational intractability. Studying the NP-complete problem to cover the edges of a graph with a minimum number of cliques ((EDGE) CLIQUE COVER)[1], we add a new example to the success story of data reduction, presenting both empirical as well as theoretical findings.

---

[*]Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany, gramm@informatik.uni-tuebingen.de. Supported by DFG project OPAL, NI 369/2.

[†]Institut für Informatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, D-07743 Jena, Germany, {guo|hueffner|niedermr}@minet.uni-jena.de. Jiong Guo and Falk Hüffner supported by DFG Emmy Noether research group PIAF, NI 369/4.

[1]We remark that covering *vertices* by cliques (VERTEX CLIQUE COVER or CLIQUE PARTITION) is of less interest to be studied on its own because it is equivalent to the well-investigated GRAPH COLORING problem: A graph has a vertex clique cover of size $k$ iff its complement graph can be colored with $k$ colors such that adjacent vertices have different colors.

Our study problem CLIQUE COVER, also known as KEYWORD CONFLICT problem [12] or COVERING BY CLIQUES (GT17) or INTERSECTION GRAPH BASIS (GT59) [9], has applications in diverse fields such as compiler optimization [19], computational geometry [2], and applied statistics [11, 18]. Thus, it is not surprising that there has been substantial work on (polynomial-time) heuristic algorithms for CLIQUE COVER [12, 14, 19, 18, 11]. In this paper, we extend and complement this work, particularly introducing new data reduction techniques.

Formally, as a (parameterized) decision problem, CLIQUE COVER is defined as follows:

> CLIQUE COVER
> **Input:** An undirected graph $G = (V, E)$ and an integer $k \geq 0$.
> **Question:** Is there a set of at most $k$ cliques in $G$ such that each edge in $E$ has both its endpoints in at least one of the selected cliques?

CLIQUE COVER is hard to approximate in polynomial time and nothing better than only a polynomial approximation factor is known [5].

We examine CLIQUE COVER in the context of parameterized complexity [7, 17]. An instance of a parameterized problem consists of a problem instance $I$ and a parameter $k$. A parameterized problem is *fixed-parameter tractable* if it can be solved in $f(k) \cdot |I|^{O(1)}$ time, where $f$ is a computable function solely depending on the parameter $k$, not on the input size $|I|$.

Our contributions are as follows. We start with a thorough mathematical analysis of the heuristic of Kellerman [12] and the postprocessing of Kou et al. [14], which is state of the art [19, 11]. For an $n$-vertex and $m$-edge graph, we improve the runtime from $O(nm^2)$ to $O(nm)$. Afterwards, as our main algorithmic contribution, we introduce and analyze data reduction techniques for CLIQUE COVER. As a side effect, we provide a so-called problem kernel for CLIQUE COVER, for the first time showing—somewhat surprisingly—that the problem is fixed-parameter tractable with respect to the parameter $k$. We continue with describing an exact algorithm based on a search tree. For our experimental investigations, we combined our data reduc-

tion rules with the search tree, clearly outperforming heuristic approaches in several ways. For instance, we can solve real-world instances from a statistical application [18]—so far solved heuristically [18, 11]—optimally without time loss. This indicates that for a significant fraction of real-world instances our exact approach is clearly to be preferred to a heuristic approach which is without guaranteed solution quality. We also experimented with random graphs of different densities, showing that our exact approach works extremely well for sparse graphs. In addition, our empirical results reveal that for dense graphs a data reduction rule that was designed for showing the problem kernel does very well. In particular, this gives strong empirical support for further theoretical studies in the direction of improved fixed-parameter tractability results for CLIQUE COVER, nicely demonstrating a fruitful interchange between applied and theoretical algorithmic research.

Not all details and proofs are given in this extended abstract.

## 2 Improved Heuristic

Kellerman [12] proposed a polynomial-time heuristic for CLIQUE COVER. This heuristic was improved by Kou, Stockmeyer, and Wong [14] by adding a postprocessing step; this version has been successfully applied to instruction scheduling problems [19] and in the analysis of statistical data [11]. Clearly, both versions of the heuristic run in polynomial time but in both cases a more precise analysis of their runtime was not given. In this section, for an $n$-vertex and $m$-edge graph, we analyze the runtime of both heuristics as $O(nm^2)$. We show how to slightly modify Kellerman's heuristic such that we can improve the runtime of both heuristics to $O(nm)$ by a careful use of additional data structures.

The CLIQUE COVER heuristic by Kellerman [12], further-on referred to as CC-Heuristic 1, is described in pseudo-code in the left column of Fig. 1. To simplify notation, we use $V = \{1, \ldots, n\}$ as vertex set. The algorithm starts with an empty clique cover and successively, for $i = 1, \ldots, n$, updates the clique cover to account for edges $\{i, j\}$ with $j < i$. In case there are no edges between the currently processed vertex $i$ and the set $W$ of its already processed neighbors, a new clique is created, containing only $i$. Otherwise, we try to add $i$ to existing cliques where possible. After this, there may still remain uncovered edges between $i$ and $W$. To cover those edges, we create a new clique containing $i$ and its neighbors from one of the existing cliques such that the number of edges covered by this new clique is maximized. We repeat this process until all edges between $i$ and vertices from $W$ are covered.

To improve both the analysis and in some cases the

**Input:** Clique $C_l$, vertex $i$.
**Globals:** $S$, $I$
1    **for** $j \in N_>(j')$, with some arbitrary $j' \in C_l$**:**
2       **if** $l \in S[j] \wedge i \notin N_<(j)$**:**
3          $S[j] \leftarrow S[j] \setminus \{l\}$
4    **for** $j \in N_>(i)$**:**
5       update position of $l$ in sorted $I[j]$
6    $C_l \leftarrow C_l \cup \{i\}$

Figure 2: The *add-to-clique* subroutine.

result of the heuristic, we assume that the algorithm is aborted as soon as $m$ cliques are generated. In that case we can simply take the solution that covers each edge separately by a two-vertex clique.

It is relatively straightforward to observe that CC-Heuristic 1 runs in $O(nm^2)$ time.

To improve the runtime, the idea is to identify the "hot spots" and to use caching data structures that will give the computation of these hot spots basically for free, and spread the work of keeping this structure up-to-date throughout the rest of the program.

We maintain the following two tables for vertices $i$, $1 \leq i \leq n$, where $C_l$, $1 \leq l \leq m$, are the cliques generated by the algorithm:

$$S[i] := \{l \mid C_l \subseteq N_<(i)\},$$
$$I[i] := \{l \mid C_l \cap N_<(i) \neq \emptyset\}, \text{ sorted}$$
$$\text{descending according to } |C_l \cap N_<(i)|,$$

where the set $N_<(i)$ for a vertex $i$ is defined as

$$N_<(i) := \{j \mid j < i \text{ and } \{i, j\} \in E \text{ is uncovered}\},$$

with $\{i, j\} \in E$ called *uncovered* if $\forall 1 \leq l \leq m$ : $\{i, j\} \not\subseteq C_l$. The set $N_>(i)$ is defined analogously. The table $S[i]$ is used to keep track of the existing cliques to which a vertex $i$ may be added. Using $S[i]$ will avoid to inspect every existing clique individually in order to test whether vertex $i$ can be added to the clique (see lines 10 and 11 for CC-Heuristic 1 in Fig. 1). The table $I[i]$ is used to keep track of the existing cliques with which a vertex $i$ has an "uncovered overlap", i.e., with which $i$ shares yet uncovered edges. The cliques are kept sorted by the size of this uncovered overlap. Using table $I[i]$, we will avoid the costly computation of a clique with maximum uncovered overlap in line 18. Thus, tables $S$ and $I$ help to replace costly operations during the heuristic by constant-time look-ups in these tables. This comes at the price of having to keep these tables up-to-date throughout the heuristic.

We now first explain how the newly introduced tables are updated throughout the heuristic and, then,

**Input:**  Graph $G = (\{1, 2, \ldots, n\}, E)$ without isolated vertices.
**Output:**  A clique cover for $G$.

1    $k \leftarrow 0$    ▷ *number of cliques created so far*
2    **for** $i = 1, 2, \ldots, n$:
            ▷ *loop invariant: $C_1, \ldots, C_k$ cover all edges incident to vertices $v, w \leq i$*
3        $W \leftarrow \{j \mid j < i \text{ and } \{i, j\} \in E\}$
4        **if** $W = \emptyset$:
5            $k \leftarrow k + 1$
6            $C_k \leftarrow \{i\}$                                    | *add-to-clique*$(k, i)$
7        **else:**
8            ▷ *try to add $v_i$ to each of the existing cliques*
9            $U \leftarrow \emptyset$    ▷ *set of neighbors $j$ of $i$ where $\{i, j\}$ is already covered*
10           **for** $l = 1, \ldots, k$:                  $S' \leftarrow S[i]$
11               **if** $C_l \subseteq W$:                    **for** $l \in S'$:
                                                               **if** $C_l \not\subseteq U$:
12                   $C_l \leftarrow C_l \cup \{i\}$              *add-to-clique*$(l, i)$
13                   $U \leftarrow U \cup C_l$                    $U \leftarrow U \cup C_l$
14                   **if** $U = W$: **break**               **if** $U = W$: **break**
15           $W \leftarrow W \setminus U$
16           **while** $W \neq \emptyset$:
17               ▷ *for the remaining edges, try to cover as many as possible at a time*
18               $l \leftarrow \min\{l \mid 1 \leq l \leq k,$                $l \leftarrow \min\{I[i]\}$
                         $|C_l \cap W| \text{ is maximal}\}$
19               $k \leftarrow k + 1$                              $k \leftarrow k + 1$
20               $C_k \leftarrow (C_l \cap W) \cup \{i\}$          **for** $q \in (C_l \cap W) \cup \{i\}$:
                                                                     *add-to-clique*$(k, q)$
21               $W \leftarrow W \setminus C_l$
22   **return** $\{C_1, C_2, \ldots, C_k\}$

Figure 1: Comparison of CC-Heuristic 1 by Kellerman [12] and the improved CC-Heuristic 2 in pseudo-code. Code in the right column indicates that in CC-Heuristic 2 the code replaces the corresponding lines of CC-Heuristic 1 shown in the left column.

we show how they are used to modify CC-Heuristic 1. Initially, the entries $S[i]$ and $I[i]$ are empty for all $1 \leq i \leq n$. To handle all modifications to our partial clique cover and to update tables $S$ and $I$, we introduce the function *add-to-clique*, presented in pseudo-code in Fig. 2. The function adds one vertex $i$ to a clique $C_l$ and updates these tables as follows.

- Table $S$: The grown $C_l$ might not be subset of $N_<(j)$ anymore for some $j$ with $l \in S[j]$. This is accounted for in lines 1–3. To this end, we need to find all vertices $j$ such that clique $C_l$ *is* subset of $N_<(j)$ *before* adding $i$ to $C_l$ but *is not* subset of $N_<(j)$ *after* adding $i$ to $C_l$. All these $j$ are certainly found when inspecting all neighbors of one arbitrarily selected element $j'$ of the old $C_l$. If, for some $j$, $l \in S[j]$ but $i \notin N_<(j)$, then we remove $l$ from $S[j]$.

- Table $I$: For each $j$ in $N_>(i)$, we percolate $l$ to its right place in $I[j]$ since $C_l \cap N_<(j)$ has grown by one. This is done in lines 4–5.

Clearly, after these updates, the addition of $i$ to $C_l$ has been correctly accounted for.

Fig. 1 shows how function *add-to-clique* is used to modify CC-Heuristic 1 in order to obtain our new heuristic to which we refer as CC-Heuristic 2. More precisely, we replace every addition of a vertex to a clique by a call to the new function *add-to-clique*; this explains the changes of lines 6, 12, and 20 and makes sure that in these modifications the tables $I$ and $S$ are updated.

Table $S$ is then used to speed up lines 10 to 14 of the old heuristic where the currently processed vertex $i$ is added to existing cliques where possible. Here, instead of testing for each existing clique individually whether this is possible (as done in CC-Heuristic 1), CC-Heuristic 2 directly accesses these cliques in table entry $S[i]$. Moreover, we also change the heuristic by adding vertex $i$ to an existing clique $C_l$ only if there are edges between $i$ and $C_l$ which are yet uncovered. In contrast, CC-Heuristic 1 adds vertex $i$ to every possible existing clique $C_l$ (unless all edges connecting $i$ to pre-

viously processed vertices are already covered). Thus, in some cases CC-Heuristic 1 may add vertex $i$ to an existing clique while not covering previously uncovered edges, and, in other cases, vertex $i$ is not added to an existing clique although this would be possible. Therefore, our modification makes the algorithm more consistent and additionally this change will be essential in the runtime proof.

Table $I$ is used to speed up lines 18 to 20 of the old heuristic where new cliques are generated for the still uncovered edges connecting the currently processed vertex $i$ to its already processed neighbors in $W$. More precisely, we generate a clique containing $i$ and $C_l \cap W$ where $C_l$ is chosen such that it maximizes the overlap with $W$. While the choice of $l$ was time-consuming in CC-Heuristic 1, it is now done by a constant-time look-up in table entry $I[i]$. Then, the new clique $C_k$ is built by adding each element individually, using *add-to-clique*. This concludes the changes of CC-Heuristic 2 in comparison with CC-Heuristic 1.

To infer the runtime we observe the following essential invariant which applies to the solutions generated by CC-Heuristic 2:

LEMMA 2.1. *In a solution generated by CC-Heuristic 2, the sum of solution clique sizes is at most $2m$.*

It is not clear how to prove the invariant stated in Lemma 2.1 for CC-Heuristic 1. The reason lies in line 11 of the heuristic where CC-Heuristic 1 may add new vertices to a clique without covering previously uncovered edges. This is prevented by the modification introduced in CC-Heuristic 2 and we can infer the following runtime:

THEOREM 2.1. *CC-Heuristic 2 runs in $O(nm)$ time.*

*Proof.* We first analyze the runtime of *add-to-clique*. Clearly, each of the for loops iterates at most $n$ times. The set operations in lines 2 and 3 can be implemented to run in constant time by using bitmaps. The update in line 5 can also be done in constant time by using $n$ buckets for each $I[i]$ and an additional map that allows us to find the bucket where $l$ resides. In summary, one *add-to-clique* call takes $O(n)$ time.

To analyze the runtime of the modified algorithm, we note that the total runtime is dominated by *add-to-clique* calls. We analyze the number and total runtime of *add-to-clique* calls amortized over the whole algorithm: With Lemma 2.1 the sum of clique sizes is at most $2m$ and therefore *add-to-clique* is called at most $2m$ times. Since lines 13 and 14 take at most $O(n)$ time and are called only on occasion of an *add-to-clique* call, their runtime can be also subsumed here. This

shows a total runtime of $O(nm)$ for all *add-to-clique* calls (including lines 13 and 14).

Leaving aside the *add-to-clique* calls and lines 13/14, all other operations inside the main loop—iterating over all $n$ vertices—can be done in $O(m)$ time. This is in particular true for line 11: Using bitmaps in combination with a doubly-linked structure, testing $C_l$ for containment in a set $U$ can be done in $|C_l|$ steps. With Lemma 2.1 we infer that testing *all* existing cliques for containment in a set $U$ can be done in $O(m)$ time. $\square$

Finally, we turn our attention to the postprocessing proposed by Kou et al. [14] as an addition for CC-Heuristic 1. They proposed to test every clique found by CC-Heuristic 1 for redundancy: For every clique $C$ in the solution of CC-Heuristic 1, it is tested whether $C$ is "subsumed" by the remaining cliques of the solution, i.e., whether each of $C$'s edges is also covered by another clique of the solution. If $C$ is subsumed, then $C$ is deleted from the solution. Kou et al. left it open to give a precise estimation of the time complexity of this postprocessing.

The following result is straightforward to obtain by using appropriate caching data structures.

PROPOSITION 2.1. *The postprocessing proposed by Kou et al. [14] on instances returned by CC-Heuristic 2 can be done in $O(nm)$ time.*

## 3 Data Reduction

A (*data*) *reduction rule* replaces, in polynomial time, a given CLIQUE COVER instance $(G, k)$ consisting of a graph $G$ and a nonnegative integer $k$ by a "simpler" instance $(G', k')$ such that $(G, k)$ has a solution iff $(G', k')$ has a solution. An instance to which none of a given set of reduction rules applies is called *reduced* with respect to these rules. A parameterized problem such as CLIQUE COVER (the parameter is $k$) is said to have a *problem kernel* if, after the application of the reduction rules, the reduced instance has size $f(k)$ for a function $f$ depending only on $k$. It is a well-known result from parameterized complexity theory that the existence of a problem kernel implies fixed-parameter tractability for a parameterized problem [7, 17].

We formulate reduction rules for a generalized version of CLIQUE COVER in which already some edges may be marked as "covered." Then, the question is to find a clique cover of size $k$ that covers all non-covered edges. Clearly, CLIQUE COVER is the special case of this annotated version where no edge is marked as covered.

We start by describing an initialization routine that sets up auxiliary data structures *once* at the beginning of the algorithm such that the *many* applications of the

subsequent Rule 2 become cheaper in terms of runtime. Moreover, the data structures initialized here are also used by the exact algorithm proposed in Sect. 4. From the reduction rules below, only Rule 1 updates these auxiliary data structures.

**Initialization** We inspect every edge $\{u, v\}$ of the original graph. We use two auxiliary variables: We compute a set $N_{\{u,v\}}$ of its common neighbors and we determine whether the vertices in $N_{\{u,v\}}$ induce a clique. More precisely, we compute a positive integer $c_{\{u,v\}}$ which stores the number of edges interconnecting the vertices of $N_{\{u,v\}}$.

The following is easy to see.

LEMMA 3.1. *The proposed initialization can be done in* $O(m^2)$ *time.*

We start the presentation of reduction rules with a trivial rule removing isolated elements.

RULE 1. *Remove isolated vertices and vertices that are only adjacent to covered edges.*

LEMMA 3.2. *Rule 1 is correct. Every application of Rule 1 including the update of auxiliary variables can be executed in* $O(nm)$ *time.*

The next reduction rule is concerned with maximal cliques. Note that we can safely assume that an optimal solution consists of maximal cliques only since a non-maximal clique in a solution can always be replaced by a maximal clique it is contained in. The following rule identifies maximal cliques which have to be part of *every* optimal solution.

RULE 2. *If an edge* $\{u, v\}$ *is contained only in exactly one maximal clique* $C$, *i.e., if the common neighbors of* $u$ *and* $v$ *induce a clique, then add* $C$ *to the solution, mark its edges as covered, and decrease* $k$ *by one.*

LEMMA 3.3. *Rule 2 is correct. Every application of Rule 2 can be executed in* $O(m)$ *time.*

Rules 1 and 2 imply that all degree-1 and degree-2 vertices are removed from the instance. Further, they imply that an isolated clique is deleted: Its vertices belong to exactly one maximal clique; the clique, if it contains more than one vertex, is added to the solution by Rule 2 and its vertices are "cleaned up" by Rule 1.

In the following we present two interrelated reduction rules Rules 3' and 3. Rule 3' is subsumed by Rule 3. Nevertheless we choose to present both rules separately since Rule 3' is easier to understand and more efficient to implement. Moreover, as will be shown in Theorem 3.1, already Rule 3' is sufficient to show a problem kernel for CLIQUE COVER.
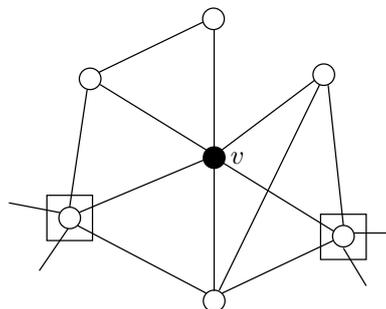


Figure 3: An illustration of the partition of the neighborhood of a vertex $v$. The two vertices with rectangles are exits, the others are prisoners.

RULE 3'. *If there is an edge* $\{u, v\}$ *whose endpoints have exactly the same closed neighborhood, i.e., for which* $N[u] = N[v]$, *then mark all edges incident to* $u$ *as covered. To reconstruct a solution for the unreduced instance, add* $u$ *to every clique containing* $v$.

Comparing $N[u]$ and $N[v]$ for each edge $\{u, v\}$, we can in $O(nm)$ time search an edge for which Rule 3' is applicable and invoke the rule.

For formulating a generalization of Rule 3' we introduce additional terminology. For a vertex $v$, we partition the set of vertices that are connected by an uncovered edge to $v$ into *prisoners* $p$ with $N(p) \subseteq N(v)$ and *exits* $x$ with $N(x) \setminus N(v) \neq \emptyset$.[2] We say that the prisoners *dominate* the exits if for every exit $x$ there is a prisoner connected to $x$. An illustration of the concept of prisoners and exits is given in Fig. 3.

RULE 3. *If there is a vertex* $v$ *which has at least one prisoner and whose prisoners dominate its exits, then mark all edges incident to* $v$ *as covered. To reconstruct a solution for the unreduced instance, add* $v$ *to every clique containing a prisoner of* $v$.

Observe that a vertex $v$ is always a prisoner of a vertex $u$ with $u \neq v$ and $N[u] = N[v]$ (and vice versa). Thus, Rule 3' is subsumed by Rule 3.

LEMMA 3.4. *Rule 3 is correct. Every application of Rule 3 can be executed in* $O(n^3)$ *time.*

*Proof.* For the correctness note that, by definition, every neighbor of $v$'s prisoners is also a neighbor of $v$ itself. If a prisoner of $v$ participates in a clique $C$, then $C \cup \{v\}$ is also a clique in the graph. Therefore,

---

[2]We remark that the concept of prisoners and exits (and, in addition, "gates") was introduced for data reduction rules designed for the DOMINATING SET problem [4]. The strength of these rules has been proven theoretically [4] as well as empirically [3].

it is correct to add $v$ to every clique containing a prisoner in the reduced graph. Next, we show that all edges adjacent to $v$ are covered by the cliques resulting by adding $v$ to the cliques containing $v$'s prisoners. W.l.o.g. we can assume that prisoners are not "isolated," i.e., they are connected to other prisoners or exits since, otherwise, Rules 1 and 2 would delete the isolated prisoner. Now, we consider separately the edges connecting $v$ to prisoners and edges connecting $v$ to exits. Regarding an edge $\{v, w\}$ to a non-isolated prisoner $w$, vertex $w$ has to be part of a clique $C$ of the solution for the instance after application of the rule. Therefore, the edge $\{v, w\}$ is covered by $C \cup \{v\}$ in the solution for the unreduced instance. Regarding an edge $\{v, x\}$ to an exit $x$, the exit $x$ is dominated by a prisoner $w$ and therefore $x$ has to be part of a clique $C$ with $w$. Therefore, the edge $\{v, x\}$ is covered by $C \cup \{v\}$ in the solution for the unreduced instance.

For executing the rule, we inspect every vertex $v$ to test whether the rule is applicable. To this end, we inspect every neighbor $u$ of $v$. In $O(n)$ time, we determine whether $u$ is an exit or a prisoner. Having identified all prisoners, we can for every exit $u$ determine in $O(n)$ time whether $u$ is dominated by a prisoner.

LEMMA 3.5. *Using Rules 1 to 3, in $O(n^4)$ time one can generate a reduced instance where none of these rules applies any further.*

From a theoretical viewpoint, the main result of this section is a problem kernel with respect to parameter $k$ for CLIQUE COVER:

THEOREM 3.1. *A CLIQUE COVER instance reduced with respect to Rules 1 and 3' contains at most $2^k$ vertices or, otherwise, has no solution.*

*Proof.* Consider a graph $G = (V, E)$ that is reduced with respect to Rules 1 and 3' and has a clique cover $C_1, \ldots, C_k$ of size $k$. We assign to each vertex $v \in V$ a binary vector $b_v$ of length $k$ where bit $i$, $1 \leq i \leq k$, is set iff $v$ is contained in clique $C_i$. If we assume that $G$ has more than $2^k$ vertices, then there must be $u \neq v \in V$ with $b_u = b_v$. Since Rule 1 does not apply, every vertex is contained in at least one clique, and since $b_u = b_v$, $u$ and $v$ are contained in the same cliques. Therefore, $u$ and $v$ are connected. As they also share the same neighborhood, Rule 3' applies, in contradiction to our assumption that $G$ is reduced with respect to Rule 3'. Consequently, $G$ cannot have more than $2^k$ vertices.

COROLLARY 3.1. CLIQUE COVER *is fixed-parameter tractable with respect to parameter $k$.*

---

**Input:**   Graph $G = (\{1, 2, \ldots, n\}, E)$.
**Output:**   A minimum clique cover for $G$.
1    $k \leftarrow 0$; $X \leftarrow$ **nil**
2    **while** $X =$ **nil**:
3        $X \leftarrow branch(G, k, \emptyset)$
4        $k \leftarrow k + 1$
5    **return** $X$

6    **function** $branch(G, k, X)$:
7    **if** $X$ covers $G$: **return** $X$
8    reduce $(G, k)$
9    **if** $k < 0$: **return nil**
10   choose $\{i, j\}$ such that $\binom{|N_{\{i,j\}}|}{2} - c_{\{i,j\}}$ is minimal
11   **for each** maximal clique $C$ in $N[i] \cap N[j]$:
12       $X' \leftarrow branch(G, k - 1, X \cup \{C\})$
13       **if** $X' \neq$ **nil**: **return** $X'$
14   **return nil**

Figure 4: Exact algorithm for CLIQUE COVER.

The result of Corollary 3.1 might be surprising when noting that many graph problems that involve cliques turn out to be hard in the parameterized sense. For example, the NP-complete CLIQUE problem is known to be W[1]-complete with respect to the clique size [7, 17]. Another example even more closely related to CLIQUE COVER is given by the NP-complete CLIQUE PARTITION problem, which is also hard in the parameterized sense. Herein, we ask, given a graph, for a set of $k$ cliques covering all *vertices* of the input graph (in contrast to covering all *edges* as in CLIQUE COVER). CLIQUE PARTITION is NP-hard already for $k = 3$ [9]. It follows that there is no hope for obtaining fixed-parameter tractability for CLIQUE PARTITION with respect to parameter $k$, unless P = NP.

## 4 Exact Search Tree Algorithm

Search trees are a popular means of exactly solving hard problems. The basic method is to identify for a given instance a small set of simplified instances such that the given instance has a solution if at least one of the simplified instances has one. The corresponding algorithm branches recursively into each of these instances until a stop criterion is met.

The search tree algorithm presented here for CLIQUE COVER works as follows. We choose an uncovered edge, enumerate all maximal cliques this edge is part of, and then branch according to which of these cliques we add to the clique cover. The recursion stops as soon as a solution is found or $k$ cliques are generated without finding a solution. The algorithm is presented in pseudo-code in Fig. 4.

Regarding the choice of the edge to branch on, we would, ideally, like to branch on the edge that is contained in the least number of maximal cliques. However, this calculation would be costly. Therefore, we make use of the infrastructure set up for an efficient incremental application of Rule 2. The initialization described in Sect. 3 provides a set $N_{\{i,j\}}$ containing the common neighborhood of edge $\{i,j\}$ and a counter $c_{\{i,j\}}$ containing the number of edges in the common neighborhood of its endpoints. Therefore, $\binom{|N_{\{i,j\}}|}{2} - c_{\{i,j\}}$ is the number of edges missing in the common neighborhood of edge $\{i,j\}$ as compared to a clique (the *score*). For branching, we choose the edge with the lowest score. If the score is 0, then the edge is contained in only one maximal clique (and thus will be reduced). If the score is 1, the edge is contained in exactly two maximal cliques. Generalizing this, it is plausible to assume that an edge is contained in few maximal cliques if its score is low.

Having chosen the edge to branch on, we determine the set of maximal cliques the edge is contained in using a variant of the classical Bron–Kerbosch algorithm [6] by Koch [13].

We use the branching routine within an iterative deepening framework, that is, we impose a maximum search depth $k$ and increase this limit by one when no solution is found.

Combining the data reduction rules described in Sect. 3—which yield a problem kernel for CLIQUE COVER—with the search tree algorithm described here, we obtain a competitive fixed-parameter algorithm for CLIQUE COVER that can solve problem instances of considerable size (a few hundred vertices) efficiently (see Sect. 5).

## 5 Experimental Results

In this extended abstract we focus on the newly developed exact algorithm with data reduction rules—eperimental investigations of CC-Heuristic 1 are kept to a minimum and of CC-Heuristic 2 are completely omitted. This is to become part of the full version of the paper.

We implemented the search tree algorithm from Sect. 4 and the data reduction rules from Sect. 3. The program is written in the Objective Caml programming language [15] and consists of about 1200 lines of code. The source code is free software and available from the authors on request. Graphs are implemented using a purely functional representation based on Patricia trees. This allows to (conceptually) modify the graph in the course of the algorithm without having to worry about how to restore it when returning from the recursion. Moreover, it allows for quick intersection operations

|   | $n$ | $m$ | Clique cover size | |
|---|---|---|---|---|
|   |   |   | Heuristic | Optimal |
| A | 13 | 55 | 4 | 4 |
| B | 17 | 86 | 6 | 5 |
| C | 124 | 4847 | 50 | 49 |
| D | 121 | 4706 | 48 | 48 |
| E | 97 | 3559 | 34 | 31 |

Table 1: Clique cover sizes for five real-world CLIQUE COVER instances, where "Heuristic" is CC-Heuristic 1 with the postprocessing by Kou et al. [14].

on neighbor sets, as required for some reduction rules. The cache data structure $c$ described in Sect. 3 is implemented using a priority search queue.

We tested our implementation on various inputs on an AMD Athlon 64 3400+ with 2.4 GHz, 512 KB cache, and 1 GB main memory, running under the Debian GNU/Linux 3.1 operating system.

**Real Data.** We first tested the implementation on five "real-world" instances from an application in graphical statistics [18] (see Table 1). Currently, heuristics like that of Kou et al. [14] are used to solve the problem in practice [18, 11]. With our implementation of CC-Heuristic 1, the runtime is negligible for these instances ($< 0.1$ s). Our implementation based on the search tree with data reduction could solve all instances to optimality within less than two seconds. We observe that the heuristic produces reasonably good results for these cases; previously nothing was known about its solution quality. In summary, the application of our algorithm in this area seems quite attractive, since we can provide provably optimal results within acceptable runtime bounds.

**Random Graphs.** Next, we tested the implementation of the exact algorithm on random graphs, that is, graphs where every possible edge is present with a fixed probability. It is known that with high probability a random graph has a large clique cover of size $O(n^2/\log^2 n)$ [8]. Therefore, relying on branching and a not too large search tree is unlikely to succeed, and reduction rules are crucial. The results are presented in Fig. 5. In the following, the "size" of an instance means the number of vertices. We exhibit three trials: Sparse graphs with $m \approx n \log n$, graphs with edge probability 0.1, and graphs with edge probability 0.15. For the denser graphs outliers occur: for example for graphs of size 51 and edge probability 0.15, all instances could be solved within a second but one, which took 25 minutes. In contrast, sparse graphs can be solved uniformly very quickly, and the growth even seems to be subexponen-
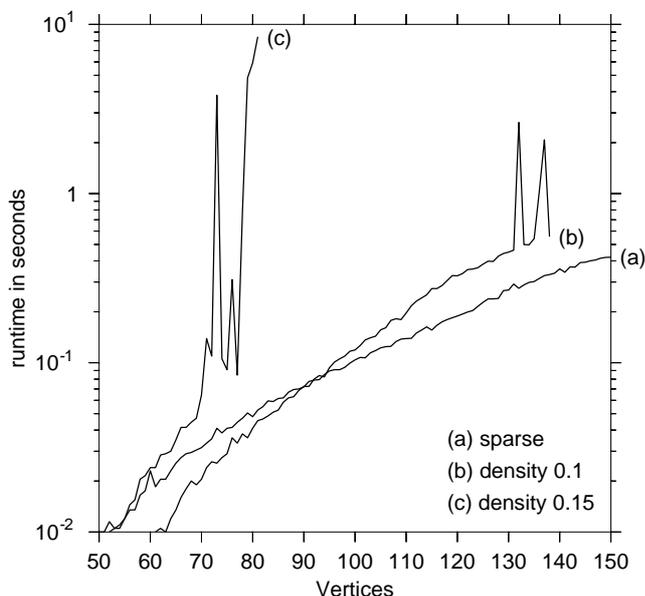
Figure 5: Runtime for random graphs. Each point is the average over 20 runs.



Figure 6: Average runtime for graphs with $n \approx 200$ and $m \approx 2000$ constructed by randomly placed cliques.

tial: Instances of size 1 000 can still be solved within 100 seconds and instances of size 2 000 within 11 minutes, with a standard deviation for the runtime of $< 2\%$. Our approach is very promising for sparse instances up to moderate size, while for denser instances probably a fallback to heuristic algorithms is required to compensate for the outliers.

The presence of extreme outliers for some combinations of parameters makes it difficult to get a clear picture based only on combining statistics such as averages. Therefore, we show measurements for several concrete instances in Table 2. For edge probability 0.1 and 0.15, respectively, we select an instance that takes very long, and additionally present two arbitrary instances with similar parameters. For sparse graphs, no such outliers occur, so we show three arbitrary instances of similar size.

**Synthetic Data.** Real instances are not completely random; in particular, in most sensible applications the clique cover is expected to be much smaller than that of a random graph. The fixed-parameter tractability of our algorithm also promises a better runtime for instances where the clique cover is small. To examine this, we generated random graph instances with approximately 200 vertices and 2000 edges by successively completing random sets of random size to form cliques until at least 2000 edges are present, but no more than 2020. By choosing the maximum size of the placed cliques, the number of placed cliques is (roughly) controlled. Figure 6 shows the resulting runtimes. In
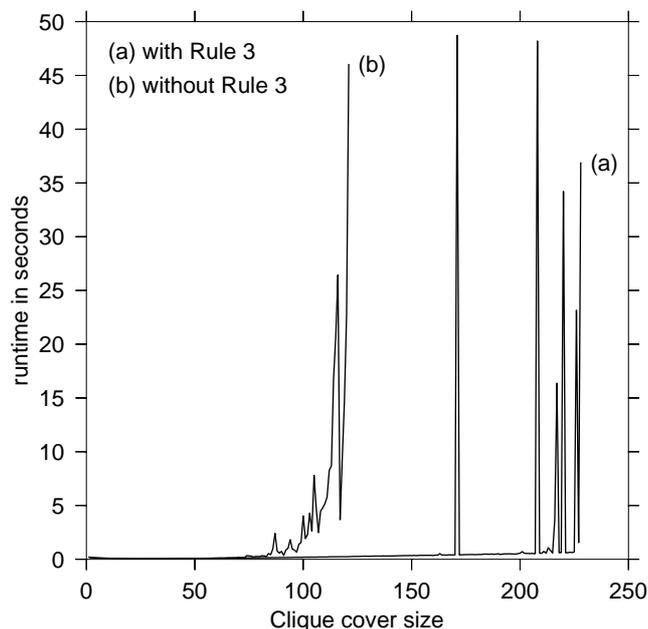
fact, these comparably large and dense instances can be solved very quickly when the size of the clique cover is small. Below a clique cover size of about 150, performance is also very smooth; no outliers occur. In contrast, the performance becomes erratic for more than about 170 cliques, with frequent occurrences of instances taking very long to solve. In summary, this makes our exact algorithm also attractive for the numerous applications where we can expect a small clique cover as solution.

**Effectiveness of Rule 3.** The prisoner-exit-rule (Rule 3) is comparably complicated and expensive and has been developed in context with searching for a problem kernel. Does it really gain any benefit in practice? To examine this question, we repeated the previous experiment with Rule 3 disabled (see Figure 6). While initially similar, around a cover size of 80 the performance drops sharply, and outliers taking very long time to solve occur. This means that Rule 3 nearly doubled the range of instances that can be solved smoothly, and is clearly worthwhile.

## 6 Outlook

As seen in Table 2, there are some outliers with exceedingly high runtimes when compared to "similar" instances. Without the application of data reduction Rule 3, there were even more such outliers. This clearly indicates that Rule 3, which also leads to a size-$2^k$ prob-

|  | $n$ | $m$ | $|C|$ | runtime | search tree | Rule 1 | Rule 2 | Rule 3 |
|---|---|---|---|---|---|---|---|---|
| sparse | 602 | 3 837 | 3 230 | 21.77 | 6 463 | 6 434 | 5 214 833 | 1 192 |
|  | 602 | 3 786 | 3 239 | 21.55 | 6 479 | 3 472 | 5 243 941 | 0 |
|  | 603 | 3 910 | 3 340 | 23.37 | 66 81 | 6 365 | 5 576 130 | 0 |
| $p = 0.1$ | 152 | 1 202 | 628 | 4 860.24 | 76 856 966 | 103 117 567 | 126 934 019 | 166 594 479 |
|  | 152 | 1 130 | 627 | 0.83 | 1 578 | 1 126 | 196 093 | 4 972 |
|  | 151 | 1 207 | 644 | 1.01 | 1 603 | 1 715 | 206 732 | 6 260 |
| $p = 0.15$ | 80 | 531 | 243 | 24 002.49 | 1 063 679 952 | 1 327 673 517 | 397 529 584 | 225 500 975 |
|  | 80 | 492 | 244 | 0.11 | 1 248 | 898 | 29 450 | 1 753 |
|  | 80 | 501 | 242 | 0.73 | 33 769 | 47 098 | 38 331 | 9 488 |

Table 2: Statistics for selected CLIQUE COVER instances. Here, $p$ is the edge probability, runtime is in seconds, $|C|$ is the size of the clique cover, "search tree" is the number of nodes in the search tree, and "Rule $r$" is the number of applications of Rule $r$.

lem kernel, can cope with some of the outliers but not all. Hence it is an intriguing open question whether there are further data reduction rules that can cope with the remaining outliers. In parallel, this might also lead to a better upper bound on the problem kernel size and improved fixed-parameter tractability for CLIQUE COVER.

# References

[1] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proc. 6th ALENEX*, pages 62–69. SIAM, 2004.

[2] P. K. Agarwal, N. Alon, B. Aronov, and S. Suri. Can visibility graphs be represented compactly? *Discrete and Computational Geometry*, 12:347–365, 1994.

[3] J. Alber, N. Betzler, and R. Niedermeier. Experiments on data reduction for optimal domination in networks. In *Proc. 1st INOC*, pages 1–6, 2003. Long version to appear in *Annals of Operations Research*.

[4] J. Alber, M. R. Fellows, and R. Niedermeier. Polynomial-time data reduction for Dominating Set. *Journal of the ACM*, 51:363–384, 2004.

[5] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999.

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

[7] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[8] A. M. Frieze and B. A. Reed. Covering the edges of a random graph by cliques. *Combinatorica*, 15(4):489–497, 1995.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[10] J. F. Gimpel. A reduction technique for prime implicant tables. *IEEE Transactions on Electronic Computers*, EC-14:535–541, 1965.

[11] J. Gramm, J. Guo, F. Hüffner, R. Niedermeier, H.-P. Piepho, and R. Schmid. Algorithms for compact letters displays—comparison and evaluation. Manuscript, FSU Jena, Dec. 2005.

[12] E. Kellerman. Determination of keyword conflict. *IBM Technical Disclosure Bulletin*, 16(2):544–546, 1973.

[13] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1–2):1–30, 2001.

[14] L. T. Kou, L. J. Stockmeyer, and C.-K. Wong. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Communications of the ACM*, 21(2):135–139, 1978.

[15] X. Leroy, J. Vouillon, D. Doligez, et al. The Objective Caml system. Available on the web, 1996. http://caml.inria.fr/ocaml/.

[16] S. Mecke and D. Wagner. Solving geometric covering problems by data reduction. In *Proc. 12th ESA*, volume 3221 of *LNCS*, pages 760–771. Springer, 2004.

[17] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[18] H.-P. Piepho. An algorithm for a letter-based representation of all-pairwise comparisons. *Journal of Computational and Graphical Statistics*, 13(2):456–466, 2004.

[19] S. Rajagopalan, M. Vachharajani, and S. Malik. Handling irregular ILP within conventional VLIW schedulers using artificial resource constraints. In *Proc. CASES*, pages 157–164. ACM Press, 2000.