Theory of Computing Systems, Vol. 38(4), pp. 373-392, 2005

Graph-Modeled Data Clustering: Fixed-Parameter Algorithms for Clique Generation*

Jens Gramm† Jiong Guo‡ Falk Hüffner Rolf Niedermeier‡
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
gramm,guo,hueffner,niedermr@informatik.uni-tuebingen.de

Abstract

We present efficient fixed-parameter algorithms for the NP-complete edge modification problems Cluster Editing and Cluster Deletion. Here, the goal is to make the fewest changes to the edge set of an input graph such that the new graph is a vertex-disjoint union of cliques. Allowing up to k edge additions and deletions (Cluster Editing), we solve this problem in $O(2.27^k + |V|^3)$ time; allowing only up to k edge deletions (Cluster Deletion), we solve this problem in $O(1.77^k + |V|^3)$ time. The key ingredients of our algorithms are two easy to implement bounded search tree algorithms and an efficient polynomial-time reduction to a problem kernel of size $O(k^3)$. This improves and complements previous work. Finally, we discuss further improvements on search tree sizes using computer-generated case distinctions.

Keywords. NP-complete, graph problems, edge modification problems, data clustering, correlation clustering, fixed-parameter tractability, exact algorithms.

*A preliminary version of this paper was presented at the 5th Italian Conference on Algorithms and Complexity (CIAC 2003), Springer-Verlag, LNCS 2653, pages 108–119, held in Rome, Italy, May 28–30, 2003.

†Supported by the Deutsche Forschungsgemeinschaft (DFG), research project "OPAL" (optimal solutions for hard problems in computational biology), NI 369/2.

‡Supported by the Deutsche Forschungsgemeinschaft (DFG), junior research group "PIAF" (fixed-parameter algorithms), NI 369/4.

1 Introduction

Motivation and problem definition. There is a huge variety of clustering algorithms with applications in numerous fields (cf., e.g., [15, 17]). Here, we focus on problems closely related to algorithms for clustering gene expression data (cf. [29] for a recent survey) and so-called correlation clustering (with applications in document clustering and "agnostic learning" [3]). In this context, Shamir et al. [27] recently studied two NP-complete graph problems called Cluster Editing and Cluster Deletion.[1] These are based on the notion of a *similarity graph* whose vertices correspond to data elements and in which there is an edge between two vertices iff the similarity of their corresponding elements exceeds a predefined threshold. The goal is to obtain a *cluster graph* by as few edge modifications (i.e., edge deletions and additions) as possible; a cluster graph is a graph in which each of the connected components is a clique. Thus, we arrive at the edge modification problem Cluster Editing which is central to our work:

> **Input:** An undirected graph $G = (V, E)$, and a nonnegative integer k.
> **Question:** Can we transform G, by deleting and adding at most k edges, into a graph that consists of a disjoint union of cliques?

Cluster Deletion is the special case where edges can only be deleted. All these problems belong to the class of *edge modification problems,* see Natanzon et al. [23] for a recent survey.

Previous work. The most important reference point to our work is the paper of Shamir et al. [27]. Among other things, they showed that Cluster Editing is NP-complete and that there exists some constant $\epsilon > 0$ such that it is NP-hard to approximate Cluster Deletion to within a factor of $1 + \epsilon$. The NP-completeness of Cluster Editing, however, can already be extracted from work of Křivánek and Morávek [20] who studied more general problems in hierarchical tree clustering. In addition, Shamir et al. studied cases where the number of clusters (i.e., cliques) is fixed. Before that, Ben-Dor et al. [4] and Sharan and Shamir [28] investigated closely related clustering applications in the computational biology context, where they deal with modified versions of the Cluster Editing problem together

[1] The third problem Cluster Completion (only edge additions allowed) is easily seen to be polynomial-time solvable.

with heuristic polynomial-time solutions. Independently of Shamir et al.'s work, Bansal et al. [3] initiated the research on "correlation clustering" which is motivated by document clustering problems from machine learning. It can be easily seen that an important special case of the general problem—also studied by Bansal et al.—is identical to Cluster Editing. Bansal et al. mainly provide approximation results which partially have been improved by very recent work [6, 8, 11]. Notably, the best known approximation factor for Cluster Editing is 4 [6]; moreover, it is shown to be APX-hard (meaning that a polynomial-time approximation scheme (PTAS) is unlikely) [6]. Thus, there is a strong motivation to search for efficient fixed-parameter algorithms to solve Cluster Editing. From the more abstract view of graph modification problems, Leizhen Cai [5] (also cf. [10]) considered the more general problem (allowing edge deletions, edge additions, *and* vertex deletions) where the "goal graphs" have a "forbidden set characterization" with respect to "hereditary graph properties" and he showed that this problem is *fixed-parameter tractable* (refer to [2, 9, 10, 12, 13] for surveys on parameterized complexity and algorithms). In particular, Cai's result implies $O(3^k \cdot |G|^4)$ time algorithms for both Cluster Editing and Cluster Deletion where the forbidden set consists of a P_3 (i.e., a vertex-induced path consisting of three vertices).[2] Natanzon et al. [23] give a general constant-factor approximation for deletion and editing problems on bounded-degree graphs with respect to properties (such as being a cluster graph) that can be characterized by a finite set of forbidden induced subgraphs. Kaplan et al. [18] and Mahajan and Raman [22] considered other special cases of edge modification problems with particular emphasis on fixed-parameter tractability results. Khot and Raman [19] recently investigated the parameterized complexity of vertex deletion problems for finding subgraphs with hereditary properties.

New results. Following a suggestion of Natanzon et al. [23] (who note that, regarding their NP-hardness results for some edge modification problems, "... studying the parameterized complexity of the NP-hard problems is also of interest."), we present significantly improved fixed-parameter tractability results for Cluster Editing and Cluster Deletion. More precisely, we show that Cluster Editing is solvable in $O(2.27^k + |V|^3)$ worst-case time and that Cluster Deletion is solvable in $O(1.77^k + |V|^3)$ worst-case time. This gives simple and efficient exact algorithms for these

[2] A graph is a cluster graph iff it contains no P_3 as a vertex-induced subgraph. This will also be important for our work. Note that Shamir et al. [27] write "P_2-free," but according to the graph theory literature it should be called "P_3-free."

NP-complete problems in case of reasonably small parameter values k (number of deletions and additions or number of deletions only). In particular, we present an efficient data reduction by preprocessing, providing a *problem kernel* of size $O(k^3)$.

Structure of the paper. After providing some basics in Sect. 2, in Sect. 3 we describe a set of efficient data reduction rules that transform a given Cluster Editing instance into a "reduced graph" with $O(k^2)$ vertices and $O(k^3)$ edges. Then, in Sect. 4 and 5, we provide two depth-bounded search trees for Cluster Editing and Cluster Deletion. In the concluding Sect. 6, we summarize our findings, indicate opportunities for future work, and point out how the achieved search tree sizes can further be lowered using computer-generated case distinctions.

2 Preliminaries and Basic Notation

One of the latest approaches to attack computational intractability is to study parameterized complexity. For many hard problems, the seemingly unavoidable combinatorial explosion can be restricted to a "small part" of the input, the *parameter*, so that the problems can be solved in polynomial time when the parameter is fixed. For instance, the NP-complete Vertex Cover problem can be solved by an algorithm with $O(1.3^k + kn)$ running time [7, 24, 26], where the parameter k is a bound on the maximum size of the vertex cover set we are looking for and where n is the number of vertices in the given graph. The parameterized problems that have algorithms of $f(k) \cdot n^{O(1)}$ time complexity are called *fixed-parameter tractable*, where f can be an arbitrary function depending only on k, and n denotes the overall input size, see [2, 9, 10, 12, 13] for details.

Our bounded search tree algorithms work recursively. The number of recursions is the number of nodes in the corresponding tree. This number is governed by homogeneous, linear recurrences with constant coefficients. It is well-known how to solve them and the asymptotic solution is determined by the roots of the characteristic polynomial (see, e.g., Kullmann [21] for more details). If the algorithm solves a problem of "size" s and calls itself recursively for problems of sizes $s - d_1, \ldots, s - d_i$, then $(d_1, \ldots, d_i)$ is called the *branching vector* of this recursion. It corresponds to the recurrence

$$t_s = t_{s-d_1} + \cdots + t_{s-d_i}$$

where t_s denotes the number of leaves in the search tree solving an instance

of size s and $t_j = 1$ for $0 \leq j < d$ with $d = \max(d_1, \ldots, d_i)$. This recurrence corresponds to the *characteristic polynomial*

$$z^d = z^{d-d_1} + \cdots + z^{d-d_i}.$$

If α is a root of the characteristic polynomial which has maximum absolute value and is positive, then t_s is α^s up to a polynomial factor. We call α the *branching number* that corresponds to the branching vector $(d_1, \ldots, d_i)$. Moreover, if α is a single root, then $t_s = O(\alpha^s)$; all branching numbers that occur in this paper are single roots.

The size of the search tree is therefore $O(\alpha^k)$, where k is the parameter and α is the largest branching number that will occur; in our case, for Cluster Editing, it will be shown that this branching number is about 2.27 and it belongs to the branching vector $(1, 2, 2, 3, 3)$ in Sect. 4.

We assume familiarity with basic graph-theoretical notations. If x is a vertex in an undirected graph $G = (V, E)$, then by $N_G(x)$ we denote the set of its neighbors, i.e.,

$$N_G(x) := \{\, v \mid \{x, v\} \in E \,\}.$$

With $\deg_G(x)$ we denote the *degree* of a vertex $x \in V$, i.e. $|N_G(x)|$. We omit the index G if it is clear from the context. The whole paper only works with *simple* graphs without *self-loops*. By $|G|$ we refer to the size of graph G, which is determined by the numbers of its vertices and edges.

In our algorithms, we use a table T to store annotations for the edges of the graph such that T has an entry for every pair of vertices $u, v \in V$ which can be empty or take one of the following values:

"*permanent*": In this case, $\{u, v\} \in E$ and the algorithm is not allowed to delete $\{u, v\}$ later on; or

"*forbidden*": In this case, $\{u, v\} \notin E$ and the algorithm is not allowed to add $\{u, v\}$ later on.

Note that, whenever the algorithms delete an edge $\{u, v\}$ from E, we set $T[u, v]$ to forbidden since it would not make sense to reintroduce previously deleted edges. In the same way, whenever the algorithms add an edge $\{u, v\}$ to E, we set $T[u, v]$ to permanent. In the following, when adding and deleting edges, we assume that we make these adjustments even when not mentioned explicitly.

3 Problem Kernel for Cluster Editing

A *reduction rule* replaces, in polynomial time, a given Cluster Editing instance (G, k) consisting of a graph G and a nonnegative integer k by a "simpler" instance (G', k') such that (G, k) is a yes-instance iff (G', k') is a yes-instance, i.e., G can be transformed into disjoint clusters by deleting/adding at most k edges iff G' can be transformed into disjoint clusters by deleting/adding at most k' edges. 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 Cluster Editing (the parameter is k) is said to have a *problem kernel* if, after the application of the reduction rules, the resulting reduced instance has size $f(k)$ for a function f depending only on k. (See [7] and [1] for two recent examples concerning the graph problems Vertex Cover and Dominating Set, respectively. There, the achieved problem kernel sizes are even linear in the parameters.)

We present three reduction rules for Cluster Editing. For each of them, we discuss its correctness and give the running time which is necessary to execute the rule. In our rules, we use table T as described in Sect. 2. Using the reduction rules, we show, at the end of this section, a problem kernel consisting of at most $2k^2 + k$ vertices and at most $2k^3 + k^2$ edges for Cluster Editing.

Although the following reduction rules also *add* edges to the graph, we consider the resulting instances as *simplified*. The reason is that for every added edge, the parameter is decreased by one. In the following rules, it is implicitly assumed that, when an edge is added or deleted, parameter k is decreased by one.

In the formulation of our rules, we use the following terminology. Given a graph $G = (V, E)$ and a vertex pair $v_i, v_j \in V$, we use *common neighbor* of v_i and v_j to refer to a vertex $z \in V$ with $\{z, v_i\} \in E$ and $\{z, v_j\} \in E$. Similarly, a *non-common neighbor* of v_i and v_j is a vertex $z \in V$ with $z \neq v_i$ and $z \neq v_j$ such that either $\{z, v_i\} \in E$ or $\{z, v_j\} \in E$ but not both.

Rule 1 For every pair of vertices $u, v \in V$:

1. If u and v have more than k common neighbors, then $\{u, v\}$ has to belong to E and we set $T[u, v] :=$ permanent. If $\{u, v\}$ is not in E, we add it to E.
2. If u and v have more than k non-common neighbors, then $\{u, v\}$ cannot belong to E and we set $T[u, v] :=$ forbidden. If $\{u, v\}$ is in E, we delete it.

3. If u and v have both more than k common and more than k non-common neighbors, then the given instance has no solution. □

Lemma 1. *Rule 1 is correct.*

Proof. **Case 1:** Vertices u and v have more than k common neighbors. If we did exclude $\{u, v\}$ from E, then we would have to, for every common neighbor z of u and v, delete $\{u, z\}$, $\{v, z\}$, or both. This, however, would require at least $k + 1$ edge deletions, a contradiction to the maximum of k edge modifications allowed.
Case 2: Vertices u and v have more than k non-common neighbors. If we did include $\{u, v\}$ in E, then we would have to, for every non-common neighbor z of u and v, edit one of the edges $\{u, z\}$ and $\{v, z\}$. Without loss of generality, let z be a neighbor of u and not a neighbor of v. Then, we would have to either delete $\{u, z\}$ from E or to add $\{v, z\}$ to E. With at least $k + 1$ non-common neighbors, this would require at least $k + 1$ edge modifications.
Case 3: Vertices u and v have more than k common neighbors and more than k non-common neighbors. From the proofs for Case 1 and Case 2 it is clear that it would require more than k edge modifications both when including $\{u, v\}$ in E and when excluding $\{u, v\}$ from E. □

Note that Rule 1 applies to every vertex pair $\{u, v\}$ for which the number of vertices which are neighbors of u or v (or both) is greater than $2k$.

Rule 2 For every three vertices $u, v, w \in V$:

1. If $T[u, v]$ = permanent and $T[u, w]$ = permanent, then $\{v, w\}$, if not already there, has to be added to E and $T[v, w] :=$ permanent.
2. If $T[u, v]$ = permanent and $T[u, w]$ = forbidden, then $\{v, w\}$, if already there, has to be deleted from E and $T[v, w] :=$ forbidden.

□

The correctness of Rule 2 is obvious. Regarding the running time, we analyze the interleaved application of Rules 1 and 2 together.

Lemma 2. *A graph can in $O(|V|^3)$ time be transformed into a graph which is reduced with respect to Rules 1 and 2.*

Proof. We present an algorithm to reduce a given graph $G = (V, E)$ to a graph $G' = (V', E')$ with respect to Rules 1 and 2 in $O(|V|^3)$ time. The algorithm processes Rules 1.1 and 1.2 and, with little additional effort, also Rules 1.3 and 2.
Data Structures.

- Adjacency matrix for G.
- Two $|V| \times (|V|-1)/2$ arrays C and N where, for a vertex pair $\{v_i, v_j\}$ with $i < j$, $C[i,j]$ ($N[i,j]$) contains the number of common (non-common) neighbors of v_i and v_j.
- Linked lists to store the vertex pairs that are candidates to be inserted into the edge set or to be deleted from the edge set. More precisely, we maintain lists $L_{c,0}, L_{c,1}, \ldots, L_{c,k}$, and L_p where $L_{c,r}$ with $1 \leq r \leq k$ contains those vertex pairs which have exactly r *common* neighbors. List L_p contains those vertex pairs which are scheduled to be set to permanent due to Rule 1.1 or Rule 2 (for instance, when they have more than k common neighbors). Similarly, we maintain lists $L_{n,0}, L_{n,1}, \ldots, L_{n,k}$, and L_f where $L_{n,s}$ with $1 \leq s \leq k$ contains those vertex pairs which have s *non-common* neighbors. List L_f contains those vertex pairs which are scheduled to be set to forbidden due to Rule 1.2 or Rule 2.
- Two $(|V| \times (|V|-1))/2$ arrays P_c and P_n. A vertex pair $\{v_i, v_j\}$ with $i < j$ is contained in at most one list from $L_{c,0}, L_{c,1}, \ldots, L_{c,k}$. If $\{v_i, v_j\}$ is contained in one of these lists, then array entry $P_c[i,j]$ contains a pointer to this list entry. If $\{v_i, v_j\}$ is contained in none of these lists, then $P_c[i,j]$ contains a null pointer. In an analogous way, $P_n[i,j]$ contains a null pointer or a pointer to an entry of $\{v_i, v_j\}$ in one list of $L_{n,0}, L_{n,1}, \ldots, L_{n,k}$.

Initialization. We assume that the adjacency matrix for G is given. For every vertex pair $v_i, v_j \in V$ with $i < j$, we initialize $C[i,j]$ and $N[i,j]$ by counting their common and non-common neighbors in the adjacency matrix. Based on these numbers, we add the pair $\{v_i, v_j\}$ to the appropriate list. If v_i and v_j have r common neighbors for $0 \leq r \leq k$, then $\{v_i, v_j\}$ is added to $L_{c,r}$. If v_i and v_j have more than k common neighbors, then $\{v_i, v_j\}$ is added to L_p. Analogously, $\{v_i, v_j\}$ is added to $L_{n,s}$ for s non-common neighbors, $0 \leq s \leq k$, and added to L_f for more than k non-common neighbors. If a vertex pair is added to one of $L_{c,0}, L_{c,1}, \ldots, L_{c,k}$, then a pointer to that entry is stored for that vertex pair in $P_c[i,j]$ (analogously in $P_n[i,j]$ if the vertex pair is added to one of $L_{n,0}, L_{n,1}, \ldots, L_{n,k}$). If now or in the following algorithm, one vertex pair $\{v_i, v_j\}$ satisfies both $C[i,j] > k$ and $N[i,j] > k$, then the algorithm terminates, reporting that the instance has no solution due to Rule 1.3.

Algorithm. In the following, we describe one iteration of the algorithm. The algorithm terminates when both L_p and L_f are empty. If at least one

of L_p and L_f is non-empty, the vertex pairs in L_p and L_f are waiting to be processed. One iteration of the algorithm processes one of these vertex pairs. In the following, we describe how to process a vertex pair $\{v_i, v_j\}$, $i < j$, taken from L_p. Processing a vertex pair from L_f will, then, work in an analogous way, and is omitted here.

For a vertex pair $\{v_i, v_j\}$ from L_p, v_i and v_j are scheduled to be connected by a permanent edge due to Rule 1 or Rule 2. We make sure that $\{v_i, v_j\}$ is in the edge set and we set $T[v_i, v_j] :=$ permanent. If, thereby, we added a new edge to the edge set, parameter k has to be decreased by one.

When adding a new edge to the edge set,

(a) we have to update the counters of common and non-common neighbors (if we change counters, then we also have to update the lists) and

(b) we have to test whether the added edge gives rise to an application of Rule 2.

Regarding (a), we assume that we add a new edge $\{v_i, v_j\}$. We assume that k still has its "old" value, i.e., it is not yet decreased by one. To update the counters and to test whether the lists have to be updated, we consider every vertex v_l with $v_l \neq v_i$ and $v_l \neq v_j$, since only for $\{v_i, v_l\}$ and $\{v_j, v_l\}$ the counters of common and non-common neighbors can change. Since we add a new edge, we have to consider the following situations (without loss of generality we assume $i < j < l$):

- Vertex v_l is a common neighbor of v_i and v_j. Then we set $N[i,l] := N[i,l] - 1$, $C[i,l] := C[i,l] + 1$, $N[j,l] := N[j,l] - 1$, and $C[j,l] := C[j,l] + 1$.

- Vertex v_l is a neighbor of exactly one of v_i and v_j. Without loss of generality we assume that v_l is a neighbor of v_j but not a neighbor of v_i. Then, we set $N[i,l] := N[i,l] - 1$, $C[i,l] := C[i,l] + 1$, and $N[j,l] := N[j,l] + 1$.

- Vertex v_l is neither a neighbor of v_i nor a neighbor of v_j. Then, we set $N[i,l] := N[i,l] + 1$ and $N[j,l] := N[j,l] + 1$.

If the value of an entry in C changes, we update lists $L_{c,r}$, $1 \leq r \leq k$, and L_p. If the value of an entry in N changes, we update lists $L_{n,s}$, $1 \leq s \leq k$, and L_f. Updating these lists is described using the example of increasing $C[i,l]$. If $C[i,l]$ is increased to a value of at most $k+1$, and $\{v_i, v_l\}$ is contained in one of the lists $L_{c,r}$, $1 \leq r \leq k$, then $\{v_i, v_l\}$ is contained in $L_{c,C[i,l]-1}$ (with

respect to the new value of $C[i,l]$). We remove $\{v_i, v_l\}$ from $L_{c,C[i,l]-1}$, using the pointer stored in $P_c[i,l]$. If the new value $C[i,l]$ satisfies $C[i,l] = k+1$, we move the entry to the end of list L_p and, otherwise, we move the entry to the end of list $L_{c,C[i,l]}$. When we move $\{v_i, v_l\}$ to L_p, we delete the pointer stored in $P_c[i,l]$ since it is no longer needed.

Having updated the counters and lists in the described way, we append list $L_{c,k}$ (still with respect to the old value of k) to the end of list L_p since, for the vertex pairs in $L_{c,k}$ and the new value of k, Rule 1.1 applies. In the same way, we append list $L_{n,k}$ to the end of list L_f due to Rule 1.2. Then, we actually decrease parameter k by 1.

Regarding (b), we test whether the permanent edge $\{v_i, v_j\}$ gives rise to an application of Rule 2 as follows. Again, we consider every vertex v_l with $v_l \neq v_i$ and $v_l \neq v_j$ and test whether $\{v_i, v_l\}$ is permanent but not $\{v_j, v_l\}$ or vice versa. Without loss of generality, we assume that $\{v_i, v_l\}$ is permanent but not $\{v_j, v_l\}$. If not already contained in L_p or L_f, we add the non-permanent vertex pair $\{v_j, v_l\}$ to the list L_p. Both tests, (a) and (b), can, by making use of the adjacency matrix and the arrays P_c and P_n, be done in $O(|V|)$ time. This completes the description of the iteration processing vertex pair $\{v_i, v_j\}$. When $\{v_i, v_j\}$ is processed, its corresponding entry is removed from L_p.

The outlined iteration is repeated until $k = 0$ or both L_p and L_f are empty. If parameter k reaches 0 before L_p and L_f are empty, then the given instance has no solution. If L_p and L_f are empty while $k \geq 0$, there is no remaining vertex pair for which Rule 1 or Rule 2 applies and, thus, the resulting graph is reduced with respect to Rules 1 and 2.

Running time. The initialization of the lists and arrays, i.e., to count the number of common and non-common neighbors for every vertex pair can be done in $O(|V|^3)$ time: For every vertex pair $\{v_i, v_j\}$, we consider all vertices v_l with $v_l \neq v_i$ and $v_l \neq v_j$. An entry for $\{v_i, v_j\}$ is added to the appropriate lists in constant time.

One iteration of the algorithm takes at most $O(|V|)$ time, since the list entries can be accessed and moved in constant time, by using the pointers stored in arrays P_c and P_n. Beyond that, the iteration involves only a loop over all vertices in V.

There are less than $|V|^2$ iterations (at most one for every vertex pair) since, after a vertex pair is processed, it is removed from all lists. Summarizing, the total time to reduce the given graph is $O(|V|^3)$. □

The following rule completes the set of reduction rules proposed in this section.

Rule 3 Delete the connected components which are cliques from the graph. □

The correctness of Rule 3 is straightforward. Computing the connected components of a graph and checking for cliques can easily be done in linear time:

Lemma 3. *Rule 3 can be executed in $O(|G|)$ time.* □

Notably, for the problem kernel size to be shown, Rules 1 and 3 would be sufficient. Rule 2 is also taken into account since it is very easy and general and can be executed in the course of executing Rule 1. Thus, Rules 1 to 3 constitute a small set of general and easy reduction rules which yields a problem kernel with $O(k^2)$ vertices and $O(k^3)$ edges and which is computable in $O(|V|^3)$ time. Note that the $O(|V|^3)$ running time given here is only a worst-case bound and it is to be expected that the application of the rules is much more efficient in practice.
The following theorem shows that reducing a graph with respect to Rules 1 to 3 leads to a problem kernel for Cluster Editing.

Theorem 1. Cluster Editing *has a problem kernel which contains at most $2k^2 + k$ vertices and at most $2k^3 + k^2$ edges. It can be found in $O(|V|^3)$ time.*

Proof. Let $G = (V, E)$ be a graph which is reduced with respect to Rules 1 to 3. Without loss of generality, we assume that G is connected. For a non-connected graph G, we process every connected component separately. Since Rule 3 deletes all isolated cliques from the given graph, G is not a clique and we need at least one edge modification to transform, by a minimum number of edge modifications, $G = (V, E)$ into a graph $G' = (V, E')$, consisting of disjoint cliques. Let k be the minimum number of required edge modifications, namely k_a edge additions and k_d edge deletions. Under the assumption that G is reduced with respect to Rules 1 to 3, we will show by contradiction that $|V| \leq (2k + 1) \cdot k$ and that $|E| \leq \binom{2k+1}{2} \cdot k$ as follows.

Assume that $|V| > (2k + 1) \cdot k$. We distinguish two cases, namely the case that $k_a = 0$ and the case that $1 \leq k_a \leq k$. In both cases, we show a contradiction to our assumption that the graph is reduced with respect to Rule 1.

(Case 1): $k_a = 0$. We have k_d edge deletions, $1 \leq k_d = k$, to transform G into G'. Let $V_C \subset V$ denote the vertex set of a largest clique in G'. The vertices in V_C also form a clique in G since $k_a = 0$. Since G is

connected, at least one vertex $u \in V_C$ is connected to a vertex $v \notin V_C$. We further distinguish two subcases: Either v is not connected to any other vertex $u' \in V_C$ with $u' \neq u$ or there is a $u' \in V_C$ with $u' \neq u$ and $\{u', v\} \in E$.

(Case 1.1): Vertex v is not connected to any other vertex $u' \in V_C$ with $u' \neq u$. We can lower-bound the clique size by $|V_C| \geq |V|/(k_d+1)$: By k_d edge deletions, G is transformed into a graph G' containing at most k_d+1 cliques and, therefore, a largest clique in G' contains at least $|V|/(k_d+1)$ vertices. Firstly, we assume that $k_d \geq 2$ $^{(*)}$. Using our further assumptions that $|V| > (2k+1) \cdot k$ $^{(**)}$ and $k_d = k$ $^{(***)}$ we obtain

$$|V_C| \geq \frac{|V|}{k_d+1} \overset{(**)}{>} \frac{(2k+1)\cdot k}{k_d+1} \overset{(***)}{=} \frac{2k^2+k}{k+1} = \frac{k^2+k}{k+1} + \frac{k^2}{k+1} \overset{(*)}{\geq} k+1.$$

Consequently, $|V_C| \geq k+2$ and u has at least $k+1$ neighbors—all vertices $u' \in V_C$ with $u' \neq u$—which are not neighbors of v. This contradicts the assumption that G is reduced with respect to Rule 1. Secondly, assuming that $k_d = k = 1$ while $|V| > (2k+1) \cdot k = 3$, G' consists of two cliques; either both contain at least two vertices or one of them contains at least three vertices—both times, Rule 1 would apply, a contradiction.

(Case 1.2): There is a $u' \in V_C$ with $u' \neq u$ and $\{u', v\} \in E$. We can lower-bound the clique size by at least $|V|/k_d$: G is transformed into a graph G' containing at most k_d cliques and, therefore, a largest clique in G' contains at least $|V|/k_d$ vertices. With the assumptions $|V| > (2k+1) \cdot k$ $^{(*)}$ and $k_d = k$ $^{(**)}$, we obtain

$$|V_C| \geq \frac{|V|}{k_d} \overset{(*)}{>} \frac{(2k+1)\cdot k}{k_d} \overset{(**)}{=} 2k+1.$$

Consequently, V_C contains more than $2k+1$ vertices and at most k many of them are connected to v. Therefore, u has more than $k+1$ neighbors in this clique which are not neighbors of v, contradicting the assumption that G is reduced with respect to Rule 1.

(Case 2): $1 \leq k_a \leq k$**.** We know, since $k_a + k_d = k$, that $k_d < k$. Again, let $V_C \subseteq V$ denote the vertex set of a largest clique in G'. Since G' contains at most k_d+1 cliques, we have $|V_C| \geq |V|/(k_d+1)$. With $k_d < k$, this yields $|V_C| \geq |V|/k$ and, using $|V| > (2k+1) \cdot k$, we obtain

$$|V_C| > (2k+1). \tag{1}$$

Since the vertices of V_C form a clique in G' and at most k many edges are added in the transformation from G to G', in G there are at most k vertex pairs (v_i, v_j) with $i < j$ and $v_i, v_j \in V_C$ which are not connected by

an edge. In the following, we show that, under the two assumptions that $|V_C| > 2(k+1)$ and that $|\{(v_i, v_j) \mid v_i, v_j \in V_C, i < j, \{v_i, v_j\} \notin E\}| \leq k$, the graph cannot be reduced with respect to Rule 1. To this end, we consider the cases that $k_a = k$ and that $k_a < k$ separately.

(Case 2.1): $k_a = k$. We conclude that $V_C = V$ and G' consists of only one clique. Since $k_a = k \geq 1$, there are $v_i, v_j \in V$ with $i < j$ and $\{v_i, v_j\} \notin E$. Further, we know that V contains more than $2k+1$ vertices and, thus, there are more than $\binom{2k+1}{2}$ many vertex pairs. Out of these vertex pairs, at most k vertex pairs (including $\{v_i, v_j\}$) are not connected by an edge. By counting arguments, v_i and v_j have at least $k+1$ common neighbors in G and Rule 1 would apply, in contradiction to our assumption that G is reduced with respect to Rule 1.

(Case 2.2): $k_a < k$. We can conclude that there are $u \in V_C$ and $v \notin V_C$ such that $\{u, v\} \in E$. Due to inequality (1), there are more than $2k+1$ vertices in V_C. On the one hand, there are at least $(2k+1) - k_a - 1$ vertices $u' \in V_C$ with $\{u', u\} \in E$. On the other hand, there are at most $k_d - 1$ vertices $u' \in V_C$ with $\{u', v\} \in E$. Consequently, we have at least

$$(2k+1) - (k_a + k_d) = (2k+1) - k = k+1$$

vertices $u' \in V_C$ with $\{u', u\} \in E$ but $\{u', v\} \notin E$. This implies that u has at least $k+1$ neighbors in G which are not neighbors of v and Rule 1 applies. In both cases, for $k_a = k$ and for $1 \leq k_a < k$, we obtain a contradiction to the assumption that G is reduced since Rule 1 would apply.

Regarding the edge set of the connected component, we infer a contradiction from the assumption that $|E| > \binom{2k+1}{2}k$ in an analogous way as for the vertex set: Again, we let V_C be the vertex set of a largest clique in G' and we distinguish between the cases $k_d = 0$ and $k_d > 0$. If $k_d = 0$, then we can easily derive that $|V_C| > 2k+1$ and the contradiction follows in analogy to (Case 2.1) above. If $k_d > 0$, we derive (omitting some details here) that $|V_C| \geq 2k+1$. Then, the contradiction follows in analogy to (Case 2.2) above.

Summarizing, the reduced graph contains at most $2k^2+k$ vertices and at most $\binom{2k+1}{2}k = 2k^3 + k^2$ edges (otherwise, no solution exists). The running time follows directly from Lemmas 2 and 3. □

4 Search Tree Algorithm for Cluster Editing

In this section, we describe a recursive algorithm for Cluster Editing that follows the bounded search tree paradigm. The basic idea of the algorithm is

to identify a "conflict triple" consisting of three vertices and to branch into subcases to repair this "conflict" by adding or deleting edges between the three considered vertices. Thus, we invoke recursive calls on instances which are simplified in the sense that the value of the parameter is decreased by at least one. Before starting the algorithm and after every such branching step, we compute the problem kernel as described in Sect. 3. The running time of the algorithm is, then, mainly determined by the size of the resulting search tree. In Sect. 4.1, we introduce a straightforward branching strategy that leads to a search tree of size $O(3^k)$; in Sect. 4.2, we show how a more involved branching strategy leads to a search tree of worst-case size $O(2.27^k)$.

Note that the more general result of Leizhen Cai [5] as discussed in the introductory section also provides an algorithm with exponential factor 3^k. By way of contrast, however, he uses a sort of enumerative approach with more computational overhead (concerning polynomial-time computations). In addition, the search tree algorithm in Sect. 4.1 also lies the basis for a more refined search tree strategy with the improved exponential term 2.27^k. Since our mathematical analysis is purely worst-case, we expect that the search tree sizes would be usually much smaller in practical settings; this seems particularly plausible because our search tree strategy as discussed in Sect. 4.3 also allows numerous heuristic improvements of the running time and the search tree size without influencing the worst-case mathematical analysis.

4.1 Basic Branching Strategy

Central for the branching strategy described in this section is the following lemma observed in [27].

Lemma 4. *A graph $G = (V, E)$ consists of disjoint cliques iff there are no three vertices $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$.* □

Lemma 4 implies that, if a given graph does not consist of disjoint cliques, then we can find a conflict triple of vertices between which we either have to insert or to delete an edge in order to transform the graph into disjoint cliques. In the following, we describe the recursive procedure that results from this observation. Inputs are a graph $G = (V, E)$ and a nonnegative integer k, and the procedure reports, as its output, whether G can be transformed into a union of disjoint cliques by deleting and adding at most k edges.

- If the graph G is already a union of disjoint cliques, then we are done: Report the solution and return.

- Otherwise, if $k \leq 0$, then we cannot find a solution in this branch of the search tree: Return.
- Otherwise, identify $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$ (they exist with Lemma 4). Recursively call the branching procedure on the following three instances consisting of graphs $G' = (V, E')$ with nonnegative integer k' as specified below:

  (B1) $E' := E - \{u, v\}$ and $k' := k - 1$. Set $T[u, v] :=$ forbidden.

  (B2) $E' := E - \{u, w\}$ and $k' := k - 1$. Set $T[u, v] :=$ permanent, $T[u, w] :=$ forbidden, and $T[v, w] :=$ forbidden.

  (B3) $E' := E + \{v, w\}$ and $k' := k - 1$. Set $T[u, v] :=$ permanent, $T[u, w] :=$ permanent, and $T[v, w] :=$ permanent.

Proposition 1. Cluster Editing *can be solved in* $O(3^k \cdot k^2 + |V|^3)$ *time.*

Proof. The recursive procedure suggested above is obviously correct. Concerning the running time, we observe the following. The preprocessing in the beginning to obtain the reduction to a problem kernel can be done in $O(|V|^3)$ time (Theorem 1). After that, we employ the search tree with size clearly bounded by $O(3^k)$. Hence, it remains to justify the factor k^2 which stands for the computational overhead related to every search tree node. Firstly, note that in a further preprocessing step, we can once set up a linked list of all conflict triples. This is clearly covered by the $O(|V|^3)$ term. Secondly, within every search tree node (except for the root) we deleted or added one edge and, thus, we have to update the conflict list accordingly. Due to Theorem 1, we only have $O(k^2)$ graph vertices now and with little effort, one verifies that the addition or deletion of an edge can make at most $O(k^2)$ new conflict triples appear and it can make at most $O(k^2)$ conflict triples disappear. Using a doubly-linked list of all conflict triples, one can update the list, after adding or deleting an edge of the graph, in $O(k^2)$ time: after adding or deleting edge $\{v_i, v_j\}$, $v_i, v_j \in V$, we iterate over all $O(k^2)$ many vertices $v_l \in V$, $v_l \neq v_i$ and $v_l \neq v_j$. Only the status of the vertex triples $\{v_i, v_j, v_l\}$ can be changed by this modification, either by causing a new conflict (then, the triple has to be added to the conflict list) or by being a conflict solved by the modification (then, the triple has to be deleted from the conflict list). This update for one vertex triple can be done in constant time, by employing a hash table or by using a size-$|V|^3$ array to store, for every vertex triple, pointers to possible entries in the conflict list. Summarizing, the conflict list can be updated in $O(|V|) = O(k^2)$ time. □

In fact, it "does not really matter" what the polynomial factor in k is, as the interleaving technique of [25] can be applied improving Proposition 1:

Corollary 1. Cluster Editing *can be solved in* $O(3^k + |V|^3)$ *time.*

Proof. In [25], it was shown that, in case of a polynomial size problem kernel, by doing the "kernelization" repeatedly during the course of the search tree algorithm whenever possible, the polynomial factor in parameter k can be replaced by a constant factor. □

4.2 Refining the Branching Strategy

The branching strategy from Sect. 4.1 can be easily improved as described in the following. We still identify a conflict triple of vertices, i.e., $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$. Based on a case distinction, we provide for every possible situation additional branching steps. The amortized analysis of the successive branching steps, then, yields the better worst-case bound on the running time. We start with distinguishing three main situations that may apply when considering the conflict triple:

(C1) Vertices v and w do not share a common neighbor, i.e. $\nexists x \in V, x \neq u : \{v, x\} \in E$ and $\{w, x\} \in E$.

(C2) Vertices v and w have a common neighbor $x \neq u$ and $\{u, x\} \in E$.

(C3) Vertices v and w have a common neighbor $x \neq u$ and $\{u, x\} \notin E$.

Regarding case (C1), we show in the following lemma that, here, a branching into two cases (B1) and (B2) as described in Sect. 4.1 suffices.

Lemma 5. *Given a graph* $G = (V, E)$*, a nonnegative integer* k *and* $u, v, w \in V$ *with* $\{u, v\} \in E$*,* $\{u, w\} \in E$*, but* $\{v, w\} \notin E$*. If* v *and* w *do not share a common neighbor besides* u*, then branching case (B3) cannot yield a better solution than both cases (B1) and (B2), and can therefore be omitted.*

Proof. Consider a clustering solution G' for G where we did add $\{v, w\}$ (see Fig. 1). We use $N_{G \cap G'}(v)$ to denote the set of vertices which are neighbors of v in G and in G'. Without loss of generality, assume that $|N_{G \cap G'}(w)| \leq |N_{G \cap G'}(v)|$. We then construct a new graph G'' from G' by deleting all edges adjacent to w. It is clear that G'' is also a clustering solution for G. We compare the cost of the transformation $G \to G''$ to that of the transformation $G \to G'$:

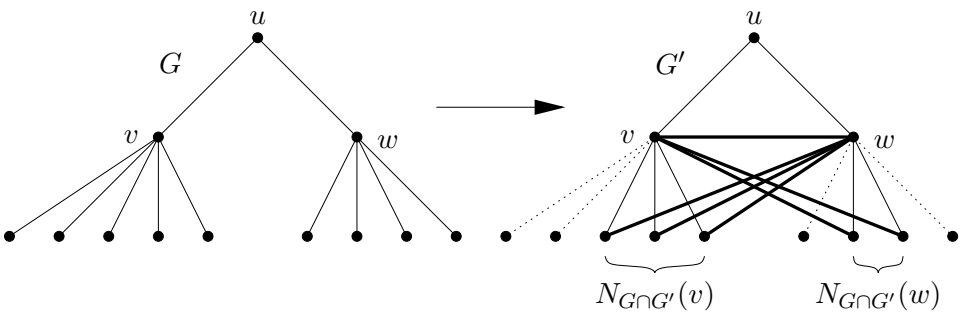

Figure 1: In case (C1), adding edge $\{v, w\}$ does not need to be considered. Here, G is the given graph and G' is a clustering solution of G by adding edge $\{v, w\}$. The dashed lines denote the edges being deleted to transform G into G', and the bold lines denote the edges being added. Observe that the drawing only shows that parts of the graphs (in particular, edges) which are relevant for our argumentation

- -1 for not adding $\{v, w\}$,
- $+1$ for deleting $\{u, w\}$,
- $-|N_{G\cap G'}(v)|$ for not adding all edges $\{w, x\}$, $x \in N_{G\cap G'}(v)$,
- $+|N_{G\cap G'}(w)|$ for deleting all edges $\{w, x\}$, $x \in N_{G\cap G'}(w)$.

Herein, we omitted possible vertices which are neighbors of w in G' but not neighbors of w in G because they would only increase the cost of transformation $G \to G'$.

In summary, the cost of $G \to G''$ is not higher than the cost of $G \to G'$, i.e., we do not need more edge additions and deletions to obtain G'' from G than to obtain G' from G. □

Lemma 5 shows that in case (C1) a branching into two cases is sufficient, namely to recursively consider graphs $G_1 = (V, E-\{u, v\})$ and $G_2 = (V, E-\{u, w\})$, each time decreasing the parameter value by one.

For case (C2), we change the order of the basic branching. In the first branch, we add edge $\{v, w\}$. In the second and third branches, we delete edges $\{u, v\}$ and $\{u, w\}$, as illustrated by Fig. 2.

- Add $\{v, w\}$ as labeled by ② in Fig. 2. The cost of this branch is 1.

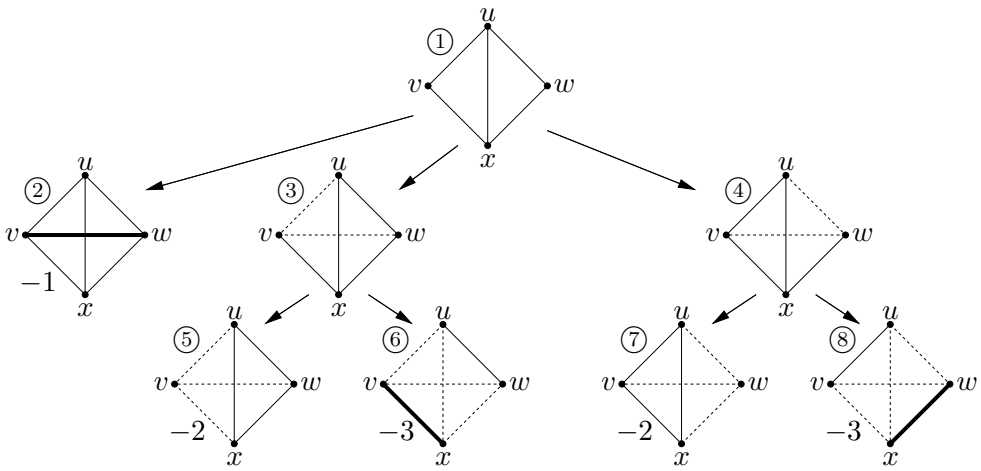

Figure 2: Branching for case (C2). Bold lines denote permanent, dashed lines forbidden edges

- Mark $\{v, w\}$ as forbidden and delete $\{u, v\}$, as labeled by ③. This creates the new conflict triple u, v, x. To resolve this conflict, we make a second branching. Since adding $\{u, v\}$ is forbidden, there are only two branches to consider:
  - Delete $\{v, x\}$, as labeled by ⑤. The cost is 2.
  - Mark $\{v, x\}$ as permanent and delete $\{u, x\}$. With reduction rule 2 from Sect. 3, we then delete $\{w, x\}$, too, as labeled by ⑥. The cost is 3.
- Mark $\{v, w\}$ as forbidden and delete $\{u, w\}$ (④). This case is symmetric to the previous one, so we have two branches with costs 2 and 3, respectively.

In summary, the branching vector for case (C2) is $(1, 2, 3, 2, 3)$.
For case (C3), we perform a branching as illustrated by Fig. 3:

- Delete $\{u, v\}$, as labeled by ②. The cost of this branch is 1.
- Mark $\{u, v\}$ as permanent and delete $\{u, w\}$, as labeled by ③. With Rule 2, we can additionally mark $\{v, w\}$ as forbidden. We then identify a new conflict triple u, v, x. Not being allowed to delete $\{u, v\}$, we can make a 2-branching to resolve the conflict:

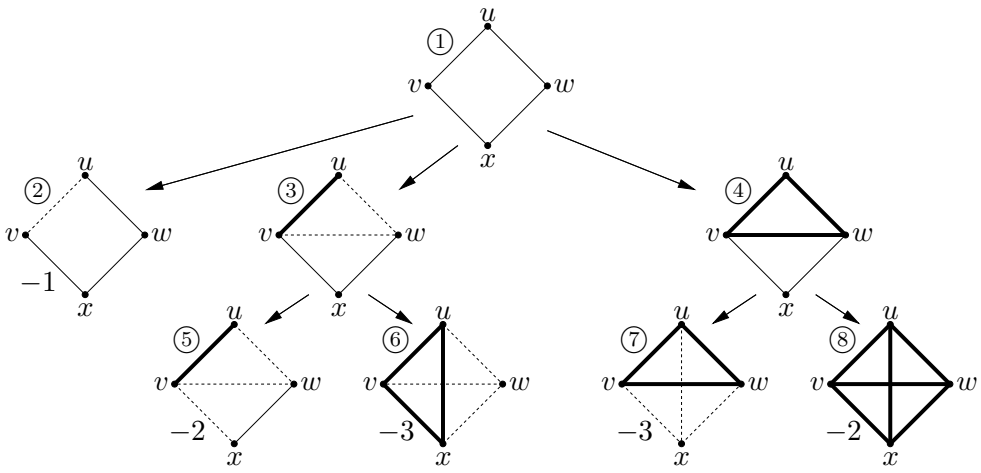

Figure 3: Branching for case (C3)

– Delete $\{v, x\}$, as labeled by ⑤. The cost is 2.

– Mark $\{v, x\}$ as permanent. This implies $\{u, x\}$ needs to be added and $\{w, x\}$ to be deleted due to reduction rule 2, as labeled by ⑥. The cost is 3.

- Mark $\{u, v\}$ and $\{u, w\}$ as permanent and add $\{v, w\}$, as labeled by ④. Vertices u, w, and x form a conflict triple. To solve this conflict without deleting $\{u, w\}$, we make a 2-branching:

  – Delete $\{w, x\}$ as labeled by ⑦. We then also need to delete $\{v, x\}$. The cost is 3. Additionally, we can mark $\{u, x\}$ as forbidden.

  – Add $\{u, x\}$, as labeled by ⑧. The cost is 2. Additionally, we can mark $\{u, x\}$ and $\{v, x\}$ as permanent.

It follows that the branching vector for case (C3) is $(1, 2, 3, 3, 2)$.

In summary, this leads to a refinement of the branching with a worst-case branching vector of $(1, 2, 2, 3, 3)$, yielding branching number 2.27. Since the recursive algorithm stops whenever the parameter value has reached 0 or below, we obtain a search tree size of $O(2.27^k)$. This results in the following theorem.

Theorem 2. Cluster Editing *can be solved in* $O(2.27^k + |V|^3)$ *time.* □

4.3 Heuristic Improvements

The following rules do not affect the worst-case time complexity since there is no guarantee that any of them ever applies; however, they might be useful for a practical implementation, since they are fairly cheap and can help reduce the size of the search tree substantially if they do apply.

4.3.1 Branching Rules

- If $\{u, v\} \in E$ and u and v do not have a common neighbor, branch into two cases: either delete $\{u, v\}$, or delete all edges adjacent to u and v except $\{u, v\}$, leaving $\{u, v\}$ as a 2-clique. With an argument similar to that used in Lemma 5, it is easy to see that an optimal solution can be found in one of the two described subcases. This branching is usually noticeably better than that of Theorem 2 (case (C1)); e.g., with u and v both being degree 3 vertices, the branching corresponds to the branching vector $(1, 4)$ and the branching number 1.39.

4.3.2 Reduction Rules

In some cases, no branching is needed, and an instance G with parameter k can be directly replaced with a simplified instance G' with parameter k'. The correctness of the following rules can be easily seen with the above branching rules and symmetry arguments.

Let u, v, w, x, y be distinct vertices.

- If $\deg(u) = \deg(v) = 1$ and $N(u) = N(v) = \{w\}$, then delete $\{u, w\}$ and set $k' := k - 1$.
- If $\deg(u) = 1, \deg(v) = 2$, $N(u) = \{v\}$ and $N(v) = \{u, w\}$, then delete $\{v, w\}$ and set $k' := k - 1$.
- If $\deg(u) = 2, \deg(v) = \deg(w) = 3$ and $N(u) = \{v, w\}, N(v) = \{u, w, x\}, N(w) = \{u, v, y\}$, then delete $\{v, x\}$ and $\{w, y\}$ and set $k' := k - 2$.

Presumably, these rules can be further generalized, e.g. to handle cliques with few connections to outside vertices.

4.3.3 Bail-Out Rules

Some branches in the search tree need not be followed, since either they cannot lead to a solution, or because it is known that for any solution they

might lead to, we find another solution which is at least as good in another branch.

- Let G_0 be the original input graph and let G be the graph in the current state of the algorithm. If G contains a vertex v with $\deg_G(v) \geq 2\deg_{G_0}(v)$ then the current branch of the search tree can be omitted, since we can be certain to find an optimal solution in another branch of the search tree. This rule is correct as can be seen as follows: for any possible clustering solution G' of G, we can construct another clustering solution G'' by removing all edges adjacent to v in G'. Clearly, the cost of transforming G_0 to G'' is not higher than the cost of transforming G_0 to G'.

5 Cluster Deletion

From the basic 3-branching algorithm for Cluster Editing in Sect. 4.1, it is straightforward to get an $O(2^k + |V|^3)$ time algorithm for Cluster Deletion as follows. Given a conflict triple consisting of vertices $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$, the insertion of an edge is not allowed here and, thus, we only need to make a branching into two cases: Either delete edge $\{u, v\}$ or delete edge $\{u, w\}$. In the remainder of this section, we show how the branching number can be improved from 2 to 1.77.

As in Sect. 4.2, we start with identifying $u, v, w \in V$ with $\{u, v\} \in E$, $\{u, w\} \in E$, but $\{v, w\} \notin E$, and distinguish the following three cases:

(C1) Vertices v and w do not share any common neighbor besides u, i. e., $\nexists x \in V, x \neq u : \{v, x\} \in E$ and $\{w, x\} \in E$.

(C2) Vertices v and w have a common neighbor $x \neq u$ and $\{u, x\} \notin E$.

(C3) Vertices v and w have a common neighbor $x \neq u$ and $\{u, x\} \in E$.

Regarding case (C1), we distinguish three subcases:

(C1.1) Vertices v and w have no other neighbors besides u:
It is easy to observe that we do not need to make any branching, it suffices to delete one arbitrary edge from $\{u, v\}$ and $\{u, w\}$ to resolve this conflict without branching.

(C1.2) Vertices v or w have a neighbor $x \neq u$ with $\{u, x\} \notin E$. We assume that vertex v has such a neighbor, i. e., $\{v, x\} \in E$, $\{w, x\} \notin E$, and $\{u, x\} \notin E$. We make a branching into two cases:

- Delete edge $\{u, v\}$. The cost of this branch is 1.
- Mark edge $\{u, v\}$ as permanent and delete $\{u, w\}$. Since $\{u, v\}$ is permanent and there is no edge between u and x, edge $\{v, x\}$ has to be deleted as well. Thus, the cost of this branch is 2.

(C1.3) All neighbors of vertices v and w are also neighbors of u. Let x be such a neighbor and assume that $\{v, x\} \in E$, $\{w, x\} \notin E$, and $\{u, x\} \in E$. We make a branching into two cases:

- Delete $\{u, w\}$. The cost of this branch is 1.
- Mark $\{u, w\}$ as permanent and delete $\{u, v\}$. Since edge $\{u, w\}$ is permanent and there is no edge between w and x, edge $\{u, x\}$ has to be deleted as well. The cost of this branch is 2.

Summarizing the three subcases, we have for case (C1) a worst-case branching vector of $(1, 2)$.

For case (C2), we apply the following branching:

- Delete $\{u, v\}$. The cost of this branch is 1.
- Mark $\{u, v\}$ as permanent and delete $\{u, w\}$. We then identify a new conflict triple v, u, x. Since edge $\{u, v\}$ is permanent, we can only resolve this conflict by deleting $\{v, x\}$. The cost of this branch is 2.

Consequently, the branching vector for case (C2) is $(1, 2)$.

For case (C3), we apply the following branching, illustrated in Fig. 4.

- Delete $\{u, v\}$ (see ② in Fig. 4). This creates a new conflict triple x, u, v. To resolve this conflict, we make a 2-branching:
  - Delete $\{v, x\}$ (③). The cost of this branch is 2.
  - Mark $\{v, x\}$ as permanent and delete $\{u, x\}$ (④). However, this implies that $\{w, x\}$ needs to be deleted. The cost is 3.
- Delete $\{u, w\}$ (⑤). This creates a new conflict triple x, u, w. To resolve this conflict, we make a 2-branching:
  - Delete $\{w, x\}$ (⑥). The cost of this branch is 2.
  - Mark $\{w, x\}$ as permanent (⑦). However, this implies that $\{u, x\}$ and also $\{v, x\}$ have to be deleted. The cost is 3.

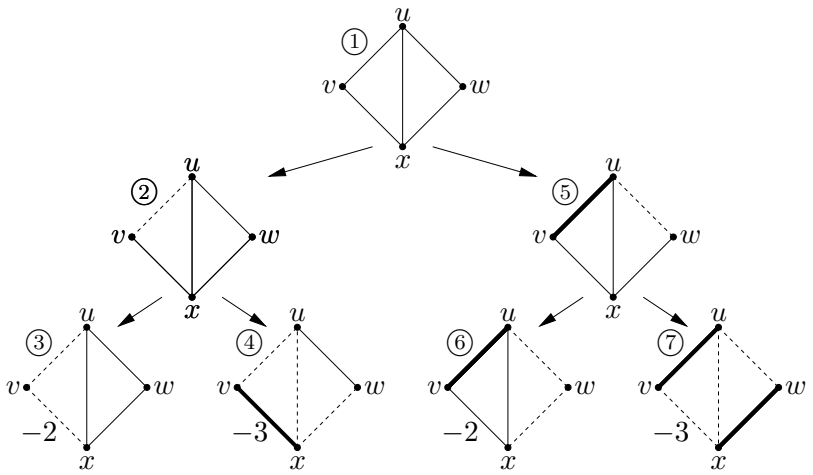

Figure 4: Branching for case (C3) of the search tree algorithm for Cluster Deletion. Bold lines denote permanent, dashed lines forbidden edges

It follows that the branching vector for case (C3) is $(2, 3, 2, 3)$.

In summary, the worst case is case (C3), where the branching vector is $(2, 3, 2, 3)$ which corresponds to branching number 1.77. In analogy to Theorem 2, we obtain the following theorem:

Theorem 3. Cluster Deletion *can be solved in* $O(1.77^k + |V|^3)$ *time.* □

6 Conclusion

Adopting a parameterized point of view [2, 9, 10, 12, 13], we have shed new light on the algorithmic tractability of the NP-complete problems Cluster Editing and Cluster Deletion. We developed efficient fixed-parameter algorithms in both cases and the algorithms seem easy enough in order to allow for efficient implementations.

We feel that the whole field of data clustering problems might benefit from more studies on the parameterized complexity of the many problems related to this field. There appear to be numerous parameters (e.g., number of clusters, number of "data cleaning operations", dimensionality of the data space) that make sense in this context.

Ongoing and future work. In ongoing work, we try to provide efficient implementations of our algorithms. It has to be investigated which of the reduction and branching rules are of real practical importance and which of them (although necessary for the theoretical worst-case analysis) only increase the administrative overhead instead of really speeding up the algorithms. Furthermore, it might be interesting to investigate how standard heuristic techniques such as branch-and-bound or A* from artificial intelligence can be used in our approach to obtain further speed-ups in practice. We plan to experiment with real-world clustering data and to incorporate additional features (such as edge and/or vertex weights) in order to deal with more realistic settings.

In recent work, we started to provide a general framework for computer-generated search trees for graph modification problems [14, 16]. Using the sheer computing power of machines and obtaining a large number of case distinctions, we achieved at least theoretical improvements over the search tree sizes given in this paper. More precisely, the improved search tree bounds achieved are $O(1.92^k)$ for Cluster Editing and $O(1.53^k)$ for Cluster Deletion. To what extent these lowered worst-case bounds also have practical significance remains an issue of future research. Note that the computer-generated search trees have a significantly increased number of branching cases which causes increased overhead in the implementation etc.

Theoretical challenges. Shamir et al. [27] showed that so-called p-Cluster Editing is NP-complete for $p \geq 2$ and p-Cluster Deletion is NP-complete for $p \geq 3$. Herein, p denotes the number of cliques that should be generated by as few edge modifications as possible. Hence, there is no hope for fixed-parameter tractability with respect to parameter p, because fixed-parameter tractability with respect to parameter p would thus imply P = NP. Moreover, Shamir et al. consider the p-Cluster Completion problem (where only edge additions are allowed)[3] and claim an algorithm running in $O(n^p)$ time.[4] Further parameterized complexity investigations concerning the parameter p in graph clustering problems seem appropriate.

We conclude with two concrete open questions concerning data reduction by preprocessing, i.e., problem kernelization:

[3] Observe that general Cluster Completion (without bound p on the number of clusters) is trivially polynomial-time solvable by simply determining all connected components of the graphs and transforming them into cliques.

[4] The algorithm described by them seems to have $O(p^n)$ running time but the $O(n^p)$ running time can be shown by additional arguments.

1. Is there a significantly better reduction to a problem kernel for Cluster Deletion than we have for Cluster Editing? Note that in Sect. 5 for Cluster Deletion we implicitly made use of the problem kernelization as given for Cluster Editing in Sect. 3.

2. Do Cluster Editing and Cluster Deletion even allow for problem kernels of linear size $O(k)$? For Vertex Cover on general graphs [7] and Dominating Set on planar graphs [1] such results are known, but it seems hard to derive similar results in our setting.

Acknowledgment. We thank Jochen Alber (Tübingen) and Elena Prieto-Rodriguez (Newcastle, Australia) for inspiring discussions and two anonymous referees of *Theory of Computing Systems* for comments that helped improving the presentation.

References

[1] J. Alber, M. R. Fellows, and R. Niedermeier. Efficient data reduction for Dominating Set: a linear problem kernel for the planar case. In *Proc. of 8th SWAT*, volume 2368 of *LNCS*, pages 150-159. Springer, 2002. Long version to appear under the title "Polynomial-Time Data Reduction for Dominating Set" in *Journal of the ACM*.

[2] J. Alber, J. Gramm, and R. Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229:3–27, 2001.

[3] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. In *Proc. of 43rd IEEE FOCS*, pages 238-247, 2002.

[4] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6(3/4):281–297, 1999.

[5] Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58:171–176, 1996.

[6] M. Charikar, V. Guruswami, and A. Wirth. Clustering with qualitative information. In *Proc. of 44th IEEE FOCS*, pages 524–533, 2003.

[7] J. Chen, I. Kanj, and W. Jia. Vertex Cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[8] E. D. Demaine and N. Immorlica. Correlation clustering with partial information. In *Proc. of 6th APPROX*, volume 2764 of *LNCS*. Springer, 2003.

[9] R. G. Downey. Parameterized complexity for the skeptic (invited paper). In *Proc. of 18th IEEE Conference on Computational Complexity*, pages 147–169, 2003.

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

[11] D. Emanuel and A. Fiat. Correlation clustering – minimizing disagreements on arbitrary weighted graphs. In *Proc. of 11th ESA*, volume 2832 of *LNCS*, pages 208–220. Springer, 2003.

[12] M. R. Fellows. Parameterized complexity: the main ideas and connections to practical computing. In *Experimental Algorithmics*, volume 2547 of *LNCS*, pages 51–77. Springer, 2002.

[13] M. R. Fellows. New directions and new challenges in algorithm design and complexity, parameterized (invited paper). In *Proc. of 8th WADS*, volume 2748 of *LNCS*, pages 505–519. Springer, 2003.

[14] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated generation of search tree algorithms for graph modification problems. In *Proc. of 11th ESA*, volume 2832 of *LNCS*, pages 642–653. Springer, 2003.

[15] P. Hansen and B. Jaumard. Cluster analysis and mathematical programming. *Mathematical Programming*, 79:191–215, 1997.

[16] F. Hüffner. *Graph Modification Problems and Automated Search Tree Generation*. Diploma Thesis, WSI für Informatik, Universität Tübingen, October 2003.

[17] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall, 1988.

[18] H. Kaplan, R. Shamir, and R. E. Tarjan. Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM Journal on Computing*, 28(5):1906–1922, 1999.

[19] S. Khot and V. Raman. Parameterized complexity of finding subgraphs with hereditary properties. *Theoretical Computer Science*, 289:997–1008, 2002.

[20] M. Křivánek and J. Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Informatica*, 23(3):311–323, 1986.

[21] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.

[22] M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31:335–354, 1999.

[23] A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113:109–128, 2001.

[24] R. Niedermeier and P. Rossmanith. Upper bounds for Vertex Cover further improved. In *Proc. of 16th STACS*, volume 1563 of *LNCS*, pages 561–570. Springer, 1999.

[25] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73:125–129, 2000.

[26] R. Niedermeier and P. Rossmanith. On efficient fixed-parameter algorithms for Weighted Vertex Cover. *Journal of Algorithms*, 47(2):63–77, 2003.

[27] R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. In *Proc. of 28th WG*, volume 2573 of *LNCS*, pages 379–390. Springer, 2002.

[28] R. Sharan and R. Shamir. CLICK: A clustering algorithm with applications to gene expression analysis. In *Proc. of 8th ISMB*, pages 307–316. AAAI Press, 2000.

[29] R. Sharan and R. Shamir. Algorithmic approaches to clustering gene expression data. In T. Jiang et al. (eds): *Current Topics in Computational Molecular Biology*, pages 269–300. The MIT Press, 2002.