Exact Algorithms for Cluster Editing: Evaluation and Experiments

Sebastian Böcker[1,2], Sebastian Briesemeister[3], and Gunnar W. Klau[4]

[1] Institut für Informatik, Friedrich-Schiller-Universität Jena, Germany, sebastian.boecker@uni-jena.de
[2] Jena Centre for Bioinformatics, Jena, Germany
[3] Div. for Simulation of Biological Systems, ZBIT/WSI, Eberhard Karls Universität Tübingen, Germany, briese@informatik.uni-tuebingen.de
[4] CWI, P.O. Box 94079, 1090 GB Amsterdam, Netherlands, gunnar.klau@cwi.nl

Preprint of: Sebastian Böcker, Sebastian Briesemeister and Gunnar W. Klau. Exact Algorithms for Cluster Editing: Evaluation and Experiments. *Algorithmica*, 60(2):316-334, 2011.

Abstract. The Cluster Editing problem is defined as follows: Given an undirected, loopless graph, we want to find a set of edge modifications (insertions and deletions) of minimum cardinality, such that the modified graph consists of disjoint cliques.
We present empirical results for this problem using exact methods from fixed-parameter algorithmics and linear programming. We investigate parameter-independent data reduction methods and find that effective preprocessing is possible if the number of edge modifications k is smaller than some multiple of $|V|$, where V is the vertex set of the input graph. In particular, combining parameter-dependent data reduction with lower and upper bounds we can effectively reduce graphs satisfying $k \leq 25\,|V|$.
In addition to the fastest known fixed-parameter branching strategy for the problem, we investigate an integer linear program (ILP) formulation of the problem using a cutting plane approach. Our results indicate that both approaches are capable of solving large graphs with 1000 vertices and several thousand edge modifications. For the first time, complex and very large graphs such as biological instances allow for an exact solution, using a combination of the above techniques.[5]

1 Introduction

The Cluster Editing problem is defined as follows: Let $G = (V, E)$ be an undirected, loopless graph. Our task is to find a set of edge modifications (insertions and deletions) of minimum cardinality, such that the modified graph consists of disjoint cliques.

[5] A preliminary version of this paper appeared under the title “Exact algorithms for cluster editing: Evaluation and experiments” in the Proceedings of the 7$^{\text{th}}$ Workshop on Experimental Algorithms, WEA 2008, in: LNCS, vol. 5038, Springer, pp. 289–302.

The Cluster Editing problem has been considered frequently in the literature since the 1980's. In 1986, Křivánek and Morávek [11] showed that the problem is NP-hard. The problem was rediscovered in the context of computational biology [15]. Clustering algorithms for microarray data such as CAST [1] and CLICK [16] rely on graph-theoretical intuition but solve the problem only heuristically. Studies in computational biology indicate that exact solutions of Cluster Editing instances can be highly application-relevant, see for instance [20]. This is even more the case for the weighted version of the problem, Weighted Cluster Editing: Given an undirected graph with modification costs for every vertex pair, we ask for a set of edge modifications with minimum total cost such that the modified graph consists of disjoint cliques.

The Cluster Editing problem is APX-hard [4] and has a constant-factor approximation of 2.5 [19]. In this article, we empirically investigate the power of methods that solve the problem to *provable optimality.* In 1989, Grötschel and Wakabayashi [8] presented a formulation of the Cluster Editing problem as an Integer Linear Program (ILP) and pointed out a cutting plane approach for its solution. Recently, the parameterized complexity of unweighted and weighted Cluster Editing, using the number (or total cost) of edge modifications as parameter k, has gained much attention in the literature [2, 6, 7]. Dehne et al. [5] present an empirical evaluation of parameterized algorithms from [7]. The fastest fixed-parameter algorithm for unweighted Cluster Editing actually transforms the problem into its weighted counterpart [3]. Guo [9] presents parameter-independent data reduction rules for unweighted instances that reduce an instance to a "hard" problem kernel of size $4k_{\text{opt}}$, where k_{opt} is the cardinality of the optimal solution for this unweighted instance. A reduction from unweighted to weighted instances of size at most $4k_{\text{opt}}$ is presented in [3]. These reductions allow us to shrink an instance even before any parameter k has been considered.

Our contributions. In the first part of our paper, we evaluate the performance of two parameter-independent data reduction strategies for unweighted Cluster Editing. We find that the efficiency of reduction is governed mostly by the ratio $k/|V|$. The unweighted kernel from [9] efficiently reduces nearly transitive graphs, but fails to reduce graphs with $k \geq \frac{1}{2}|V|$. We then present and evaluate parameter-independent reduction rules data for weighted graphs and find it to be even more effective in application. We combine the latter reduction with parameter-dependent reduction rules plus upper and lower bounds. This downsizes input graphs even more and fails to reduce graphs only when $k > 25|V|$ for large graphs.

To solve reduced instances, we implemented a branch-and-cut algorithm for Weighted Cluster Editing based on the ILP formulation proposed by Grötschel and Wakabayashi [8]. The ILP formulation of the problem has frequently been reported in the literature as being too slow for application, see for instance [10]. In contrast, we find that the cutting plane approach in [8] is capable of optimally solving large instances reasonably fast. We compare the performance of the fastest branching strategy in [3] and the cutting plane algorithm. We apply these methods to weighted instances resulting from unweighted graphs that have been fully reduced in advance using our data reduction. The FPT algorithm solves instances with $k = 5\,|V|$ in about an hour, where V and k are vertex set and parameter of the reduced instance, respectively. The ILP approach solves instances with $|V| = 1\,000$ in about an hour, almost independently of k. These approaches are particularly important for weighted input data, because we find data reduction to be less effective here.

Summarized, our experiments show that one can solve Cluster Editing instances on large graphs with several thousands of edge modifications in reasonable running time to provable optimality. In particular, feasible parameters k are orders of magnitude higher than what worst-case running times of the FPT approach suggest. We find that the most efficient way for solving Cluster Editing instances is a well-chosen combination of several approaches, namely data reduction techniques, lower/upper bounds, and a branch and cut algorithm. Test instances and source code of our approaches are available from `http://bio.informatik.uni-jena.de/peace/`.

2 Preliminaries

Throughout this paper, let $n := |V|$. We write uv as shorthand for an unordered pair $\{u, v\} \in \binom{V}{2}$. For weighted instances, let $s : \binom{V}{2} \to \mathbb{R}$ encode the input graph: For $s(uv) > 0$ an edge uv is present in the graph and has deletion cost $s(uv)$, while for $s(uv) \leq 0$ the edge uv is absent from the graph and has insertion cost $-s(uv)$. We call edges with $s(uv) = \infty$ "permanent" and with $s(uv) = -\infty$ "forbidden". A graph G is a disjoint union of cliques if and only if there exist no conflict triples in G: a *conflict triple* consists of three vertices vuw such that uv and uw are edges of G but vw is not. Note that the order of vertices vuw is important in the notation of conflict triples. Such graphs are also called *transitive.*

To solve a (weighted or unweighted) instance of Cluster Editing we first identify all connected components of the input graph. We calculate the best solutions for all components separately, because an optimal solution never connects disconnected components. In case the graph is decomposed during the course of our

tree search, then we recurse and treat each connected component individually. Recall that k is the number (or total weight) of edge modifications required to make the input graph transitive. Our fixed-parameter algorithms often require the parameter k to be part of the input: In case a solution with cost $\leq k$ exists, the algorithm finds this solution; otherwise, "no solution" is returned. To find an optimal solution we call the algorithm repeatedly, increasing k.

As a quality measure for data reduction we use the *reduction ratio* $\frac{n-n_{\text{red}}}{n}$ where n_{red} denotes the number of vertices after reduction. A reduction ratio of close to 1 corresponds to a strong reduction whereas a reduction ratio of 0 corresponds to no reduction at all.

We can encode an unweighted Cluster Editing instance using a weighted graph with edge weights ± 1. In a weighted graph we can *merge* vertices u, v into a new vertex u' when edge uv is set to "permanent": For each vertex $w \in V \setminus \{u, v\}$ we join uw, vw such that $s(u'w) \leftarrow s(uw) + s(vw)$. Moreover, in case w is a non-common neighbor of u, v we can conclude that either uw or vw has to be edited to avoid a conflict triple. If this is the case, we turn out the cheaper edit operation in advance and reduce k by $\min\{|s(uw)|, |s(vw)|\}$ [2].

Branching strategy. After parameter-independent data reduction as described in the following sections, the remaining instance can be solved using a branching tree strategy. We are given a Cluster Editing instance together with a parameter k, and we want to decide whether there exists a solution of cost at most k. We use a recursive algorithm following the bounded search tree paradigm. We identify a conflict triple and then branch into two sub-cases to repair this conflict. By this, we invoke recursive calls on "simplified" instances of the problem where parameter k is decreased.

The fastest known branching strategy for Cluster Editing, both in theory and in practice, is surprisingly simple [3]: Let uv be an edge of a conflict triple vuw. Then, (a) set uv to forbidden, or (b) merge uv. If we always choose the edge uv with minimal branching number,[6] then the resulting search tree has size $O(2^k)$. In fact, using a slightly different order in which edges are processed one can show that this branching strategy leads to a search tree of size $O(1.82^k)$ [3], but this result is mainly of theoretical interest. To find an edge with minimal branching number, we approximate log branching numbers using two rational functions. Finally, note that we do not know the optimal cost k in advance: To find an optimal solution we call the algorithm repeatedly, increasing k.

[6] The branching number is the root of the characteristic polynomial of the branching vector and governs the asymptotic size of the search tree, see e.g. [13] for details.

3 Parameter-independent Data Reduction

We now present methods for the parameter-independent data reduction of (unweighted and weighted) CLUSTER EDITING instances. We describe various polynomial-time reduction rules and apply these rules over and over again until no further rule will apply. Since the presented data reduction is parameter-independent, we can apply it during preprocessing without considering any particular parameter k. Afterwards, we can solve the reduced graph with *any* algorithm for WEIGHTED CLUSTER EDITING.

A *critical clique* C in an unweighted graph is an induced clique such that any two vertices $u, v \in C$ share the same neighborhood, $N(u) \cup \{u\} = N(v) \cup \{v\}$, and C is maximal. For unweighted CLUSTER EDITING one can easily see that all vertices of a critical clique of the input graph end up in the same cluster of an optimal clustering [9]. Furthermore, there are at most $4k_{\text{opt}}$ critical cliques in a graph, where k_{opt} is the cost of an optimal solution. Guo [9] uses critical cliques to construct a kernel for unweighted CLUSTER EDITING of size $4k_{\text{opt}}$. For brevity, we omit the details of this reduction, and only note that it is based on inspecting the neighborhood (and second neighborhood) of large critical cliques. In the following, we call this the *unweighted kernel*.

For unweighted instances, all vertices of a critical clique C must end up in the same cluster: This implies that we can merge all vertices in C for the corresponding weighted instance [3]. Doing so, we have reduced an unweighted instance to a weighted one of size at most $4k_{\text{opt}}$. In addition, we may use the following reduction rules for *any weighted* instance:

Rule 1 (heavy non-edge rule). Set an edge uv with $s(uv) < 0$ to forbidden if

$$|s(uv)| \geq \sum\nolimits_{w \in N(u)} s(uw) \ .$$

Rule 2 (heavy edge rule, single end). Merge vertices u, v of an edge uv if

$$s(uv) \geq \sum\nolimits_{w \in V \setminus \{u,v\}} |s(uw)| \ .$$

Rule 3 (heavy edge rule, both ends). Merge vertices u, v of an edge uv if

$$s(uv) \geq \sum\nolimits_{w \in N(u) \setminus \{v\}} s(uw) + \sum\nolimits_{w \in N(v) \setminus \{u\}} s(vw) \ .$$

Lemma 1. *Rules 1 to 3 are correct, and can be carried out in time $O(n^3)$.*

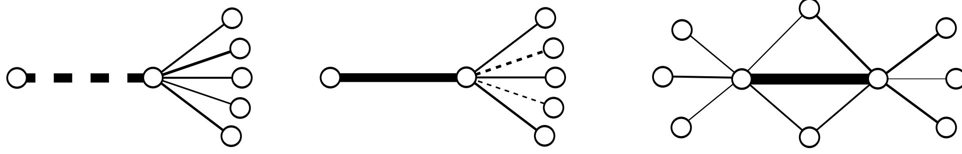

Fig. 1: Reduction rules 1–3. Heavy non-edge rule (left), heavy edge rule, single end (middle), and heavy edge rule, both ends (right).

Proof. It is easy to see that Rules 1 to 3 are correct. Also, each rule can be carried out in time $O(n)$. But checking these rules in each step would not lead to the desired running time. So, we calculate

$$r_1(uv) \leftarrow -s(uv) - \sum_{w \in N(u)} s(uw)$$
$$r_2(uv) \leftarrow s(uv) - \sum_{w \in V \setminus \{u,v\}} |s(uw)|$$
$$r_3(uv) \leftarrow s(uv) - \sum_{w \in N(u) \setminus \{v\}} s(uw) + \sum_{w \in N(v) \setminus \{u\}} s(vw)$$

and we can set uv to forbidden if $r_1(uv) \geq 0$, and merge uv if $r_2(uv) \geq 0$ or $r_3(uv) \geq 0$. We compute a list of all entries that are larger or equal to zero. Whenever we set an edge to forbidden, we have to update a linear number of entries, and each update can be performed in constant time. This may happen at most $O(n^2)$ times. We insert all entries that get larger or equal to zero, into our list of edges to update. Whenever we merge an edge, we have to update $O(n^2)$ entries, and again, each update can be performed in constant time. This may happen at most $O(n)$ times during the course of data reduction. The list of edge updates will never exceed $O(n^2)$ entries. □

Rule 4 (almost clique rule). For $C \subseteq V$ let k_C denote the min-cut value of the subgraph of G induced by vertex set C. If

$$k_C \geq \sum_{u,v \in C, s(uv) \leq 0} |s(uv)| + \sum_{u \in C, v \in V \setminus C, s(uv) > 0} s(uv)$$

then merge C.

Lemma 2. *Rule 4 is correct, and can be carried out in time* $O(n\,|C| + |C|^3)$.

We omit the simple proof of this lemma, and just note that a mincut can be found in time $O(mn + n^2 \log n)$ for n vertices and m edges [17]. Rule 4 cannot be applied to all subsets $C \subseteq V$ so we greedily choose reasonable subsets: We start with a vertex $C := \{u\}$ maximizing $\sum_{v \in V \setminus \{u\}} |s(uv)|$, and successively add vertices such that in every step, vertex $w \in V \setminus C$ with maximal connectivity $\sum_{v \in C} s(vw)$ is added. In

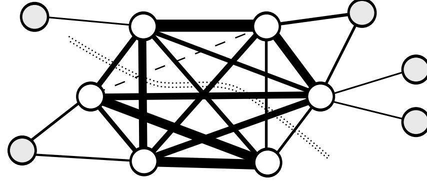

Fig. 2: Reduction rule 4, almost-clique rule. Almost-clique C (white) and neighborhood (gray).

case the connectivity of the best vertex is twice as large as that of the runner-up, we try to apply Rule 4 to the current set C. We cancel this iteration if the newly added vertex u is connected to more vertices in $V \setminus C$ than to vertices in C.

For an edge uv we define

$$N_u := N(u) \setminus (N(v) \cup \{v\}) \quad \text{and} \quad N_v := N(v) \setminus (N(u) \cup \{u\})$$

as the exclusive neighborhoods of u and v. Set $W := V - (N_u \cup N_v \cup \{u, v\})$. For $U \subseteq V$ set $s(v, U) := \sum_{u \in U} s(v, u)$. Let $\Delta_u := s(u, N_u) - s(u, N_v)$ and $\Delta_v := s(v, N_v) - s(v, N_u)$.

Rule 5 (similar neighborhood). If uv satisfies

$$s(uv) \geq \max_{C_u, C_v} \min\bigl\{s(v, C_v) - s(v, C_u) + \Delta_v, s(u, C_u) - s(u, C_v) + \Delta_u\bigr\} \qquad (1)$$

where the maximum runs over all subsets $C_u, C_v \subseteq W$ with $C_u \cap C_v = \emptyset$, then merge uv.

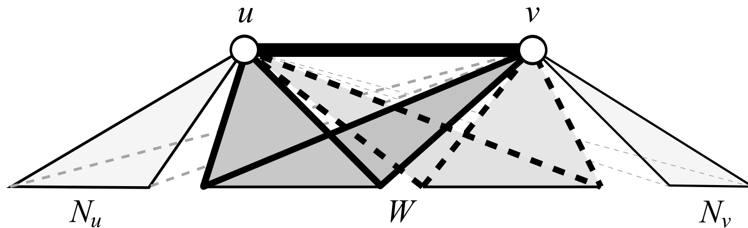

Fig. 3: Reduction rule 5, similar neighborhood rule.

Rule 5 turns out to be highly effective but its computation is expensive. In practice, we use Rule 5 only in case no other rules can be applied.

Lemma 3. *Rule 5 is correct.*

Proof. Given an edge uv of a weighted graph G that satisfies $N := N(u) \setminus \{v\} = N(v) \setminus \{u\}$, so u and v have identical neighborhoods. Suppose that in the optimal solution G', u, v do not belong to the same cluster. Let C'_u and C'_v be the clusters in G' containing u and v, respectively, and set $C_u := C'_u \setminus \{u\}$ and $C_v := C'_v \setminus \{v\}$. For the moment, let us assume $C_u \subseteq N$ and $C_v \subseteq N$. Set $C_* := N \setminus (C_u \cup C_v)$. In the optimal solution, the cost for removing edges incident with u and v are $s(u,v) + s(u, C_v) + s(v, C_u) + s(u, C_*) + s(v, C_*)$ where $s(v, U) := \sum_{u \in U} s(vu)$. If we remove v from cluster C'_v and place it with cluster C'_u instead, then we can leave uv untouched and calculate costs $s(u, C_v) + s(v, C_v) + s(u, C_*) + s(v, C_*)$. From the optimality of G' we infer

$$s(uv) \leq s(u, C_v) + s(v, C_v) - s(u, C_v) - s(v, C_u) = s(v, C_v) - s(v, C_u).$$

We can also place u with C'_v, and in total we infer

$$s(uv) \leq \min\bigl\{s(v, C_v) - s(v, C_u), s(u, C_u) - s(u, C_v)\bigr\}. \tag{2}$$

In reality, we initially have no information regarding the optimal solution. Assuming that this solution satisfies $C'_u \subseteq N$ and $C'_v \subseteq N$, we still infer

$$s(uv) \leq \max_{C_u, C_v} \min\bigl\{s(v, C_v) - s(v, C_u), s(u, C_u) - s(u, C_v)\bigr\} \tag{3}$$

where the maximum runs over all subsets $C_u, C_v \subseteq N$ with $C_u \cap C_v = \emptyset$. In case $s(uv)$ exceeds the maximum in (3) then u, v must be elements in the same cluster of an optimal solution, and we may merge u, v. If we require $N(u) \cup \{u\} = N(v) \cup \{v\}$ but allow for arbitrary disjoint clusters C_u, C_v, then we can show similarly to above that (3) must hold where the maximum runs over all subsets $C_u, C_v \subseteq V \setminus \{u, v\}$ with $C_u \cap C_v = \emptyset$.

Finally, let us consider an edge uv such that $N(u) \cup \{u\} \neq N(v) \cup \{v\}$. Now, $N_u = N(u) \setminus (N(v) \cup \{v\})$, $N_v = N(v) \setminus (N(u) \cup \{u\})$, and $W = V - (N_u \cup N_v \cup \{u, v\})$. Let u, v belong to different clusters C_u and C_v of the optimal solution. Recall that (2) holds because our solution is optimal. We define $\Delta_u := s(u, N_u) - s(u, N_v)$ and $\Delta_v := s(v, N_v) - s(v, N_u)$. Similar to above, we can estimate the cost of moving u to C_v or v to C_u, respectively. Finally, (3) becomes

$$s(uv) \leq \max_{C_u, C_v} \min\bigl\{s(v, C_v) - s(v, C_u) + \Delta_v, s(u, C_u) - s(u, C_v) + \Delta_u\bigr\}$$

where the maximum runs over all subsets $C_u, C_v \subseteq W$ with $C_u \cap C_v = \emptyset$. So, in case u, v satisfy (1) then u, v must belong to the same cluster of an optimal solution:

In case of equality, we can find at least one solution that is optimal and shows this property. □

How can we efficiently find the maximum of (1) over all subsets C_u, C_v? A first, naïve approach results in $O(3^{|W|})$ running time what is obviously unsatisfactory. We formalize this problem as follows: We are given a set B of pairs (x, y) of integers. For parameter-independent data reduction, we will set

$$B := \big\{(s(u, w), s(v, w)) \,:\, w \in W\big\}. \tag{4}$$

We use the notations $\sum_x B := \sum_{(x,y)\in B} x$ and $\sum_y B := \sum_{(x,y)\in B} y$. We have to assign all pairs in B to three buckets B_0, B_1, B_2 such that

$$\min\Big\{\textstyle\sum_x B_1 - \sum_x B_2,\ \sum_y B_2 - \sum_y B_1\Big\} \tag{5}$$

is *maximized*. The pairs in B_0 are ignored, so a lower bound for the maximum is zero. A trivial approach requires $3^{|B|}$ running time what is obviously undesirable.

We use dynamic programming to find the maximum of (5). To this end, set $X := \sum_{(x,y)\in B} |x|$ and $Y := \sum_{(x,y)\in B} |y|$. We define Boolean dynamic programming matrices $D_j[-X \dots X, -Y \dots Y]$ as follows: Let $B = \{(x_1, y_1), \dots, (x_k, y_k)\}$ be the set of pairs. We set $D_j[x, y]$ to 'true' if there exists a partition B_0, B_1, B_2 of $\{(x_1, y_1), \dots, (x_j, y_j)\}$ such that

$$\textstyle\sum_x B_1 - \sum_x B_2 = x \quad \text{and} \quad \sum_y B_2 - \sum_y B_1 = y.$$

Clearly, $D_0[x, y]$ is 'true' if and only if $(x, y) = (0, 0)$. Now, we can assign an element (x_j, y_j) to one of the three buckets and, hence,

$$D_j[x, y] = \big(D_{j-1}[x, y] \text{ or } D_{j-1}[x + x_j, y - y_j] \text{ or } D_{j-1}[x - x_j, y + y_j]\big).$$

Using this recurrence, D_k can be computed in time $O(kXY)$ and space $O(XY)$. Now, the maximum of (5) equals

$$\max_{D_k[x,y]=\text{'true'}} \big\{\min\{x, y\}\big\}. \tag{6}$$

To find a solution for (1) we have to take into account Δ_u and Δ_v. We compute D_k as above, initializing B from (4). If uv satisfies

$$s(uv) \geq \max_{D_{|W|}[x,y]=\text{'true'}} \big\{\min\{x + \Delta_u, y + \Delta_v\}\big\} \tag{7}$$

then uv must be part of the optimal solution, so we can merge uv.

Unfortunately, the quadratic running time of the above dynamic programming is too slow in applications, in particular if we take into account real-valued edge weights (see below). But we can improve running time and space to linear, as follows: We define $M_j[x]$ to be the *maximal* index y such that $D_j[x, y]$ is 'true'. We initialize $M_0[0] = 0$ and $M_0[x] = -\infty$ for all $x \neq 0$. We use the recurrence

$$M_j[x] = \max\bigl\{M_{j-1}[x],\, M_{j-1}[x + x_j] - y_j,\, M_{j-1}[x - x_j] + y_j\bigr\}$$

and compute the maximum as $\max_x \min\{x, M_k[x]\}$. To prove that this value equals the maximum in (6), we just point out that an entry $D_j[x, y] =$ 'true' *dominates* an entry $D_j[x', y] =$ 'true' for $x > x'$: No "descendant" of the latter entry will be used for the computation of (6). From the above, we infer:

Lemma 4. *Given an integer-weighted graph, then for each edge uv, Rule 5 can be applied in time $O(|W|\, Z)$ and space $O(Z)$ where $Z := \sum_{w \in W}(s(uw) + s(vw))$.*

Clearly, if $s(uv) \leq \min\{\Delta_u, \Delta_v\}$ then uv cannot satisfy equation (7) and we may proceed to the next edge. On the other hand, in case uv satisfies

$$s(uv) \geq \frac{1}{2}\Bigl(\sum\nolimits_{w \in W} |s(uw) - s(vw)| \; + \Delta_u + \Delta_v\Bigr)$$

then we can merge uv without computing the dynamic programming table. For edges that fall in non of these categories, we can use these bounds do decide in which order edges will be tested using Rule 5.

Regarding real-valued instances, we note that our dynamic programming cannot directly be applied for real-valued weights. We stress that we can multiply weights in an edge-weighted graph by an arbitrary constant $c \in \mathbb{R}$ without changing the optimal solution of the problem. We round edge weights during the course of dynamic programming, where some care has to be taken: We have to ensure that the computed bound is in fact an upper bound for an edge weight $s(uv)$ if u, v are not part of the same cluster, see (3). Note that by rounding, we may lose some edges that we could merge using exact computations. But using a (sufficiently large) multiplicative constant we can trade space and running time for exactness of the solution. We omit further details.

Finally, we note that we can also apply the above reduction rules while traversing the search tree. Rules 1–3 are applied in every node of the search tree, whereas the two more involved Rules 4 and 5 are applied only every sixth step, to optimize running times.

4 Parameter-dependent Data Reduction

In the parameter-dependent case, we are given a CLUSTER EDITING instance together with a parameter k, and we have to decide whether there exists a solution of cost at most k. We want to use the parameter-dependent data reduction for WEIGHTED CLUSTER EDITING from [2]: We define induced costs $icf(uv)$ and $icp(uv)$ for setting uv to "forbidden" or "permanent" by

$$\begin{aligned} icf(uv) &= \sum\nolimits_{w \in N(u) \cap N(v)} \min\{s(uw), s(vw)\}, \quad \text{and} \\ icp(uv) &= \sum\nolimits_{w \in N(u) \triangle N(v)} \min\{|s(uw)|, |s(vw)|\}, \end{aligned} \tag{8}$$

where $A \triangle B$ denotes the symmetric set difference of A and B. If $icp(uv) + \max\{0, -s(u,v)\}$ or $icf(uv) + \max\{0, s(u,v)\}$ exceed k, we can set uv to "forbidden" or "permanent", respectively. In the latter case, we merge u,v and reduce k by $icp(uv) + \max\{0, -s(u,v)\}$ accordingly. We can also remove isolated cliques. We use the parameter-dependent data reduction before we start the branching algorithm, and also in the course of the branching.

As an algorithm-engineering technique, we now describe fast methods to compute lower bounds on the cost of a weighted instance. Clearly, such bounds can be used to stop search tree recursion more efficiently. Assume that there exist t conflict triples in our instance G, k. For every pair uv let $t(uv)$ denote the number of conflict triples in G that contain uv, and let $r(uv) := |s(uv)| / t(uv)$. To resolve t conflicts in our graph we have to pay at least $t \cdot \min_{uv}\{r(uv)\}$. A more careful analysis shows that we can sort pairs uv according to the ratio $r(uv)$, then go through this sorted list from smallest to largest ratio. This leads to a second, tighter lower bound but requires more computation time.

Our third lower bound proved to be most successful in applications: Let CT be a set of edge-disjoint conflict triples. Then, $\sum_{vuw \in CT} \min\{s(uv), s(uw), -s(vw)\}$ is a lower bound for solving all conflict triples. Since finding the set CT maximizing this value is computationally expensive, we greedily construct a set of edge-disjoint conflict triples CT and use the above sum as a lower bound.

The most effective use for the above lower bounds, is to make induced costs $icp(uv)$ and $icf(uv)$ tighter: let $b(G, uv)$ be a lower bound that *ignores* all edges uw and vw for $w \in V \setminus \{u, v\}$ in its computation. If

$$\begin{aligned} icp_*(uv) &:= icp(uv) + \max\{0, -s(uv)\} + b(G, uv) \; > k \quad \text{or} \\ icf_*(uv) &:= icf(uv) + \max\{0, s(uv)\} + b(G, uv) \; > k \end{aligned} \tag{9}$$

holds for an edge uv, then we can set uv to "forbidden" or "permanent", respectively.

To use this powerful reduction during (parameter-independent) preprocessing, we generate a problem instance (G, k) from G by using an *upper bound* for the modification costs of G as our parameter k. There exist a multitude of possibilities to compute such upper bounds, because we can use any heuristic for the problem and compute the cost of its solution, see for instance [20]. For this study, we calculate an upper bound using a greedy approach that iteratively searches for edges where reduction rules almost apply. In detail, we search for an edge uv such that $\max\{icp_*(uv), icf_*(uv)\}$ is maximum. In case $icp_*(uv) > icf_*(uv)$, we set uv to "forbidden", otherwise we set uv to "permanent". Since only one edge is selected at a time we avoid the case that $icp_*(uv) = icf_*(uv) = \infty$. We find this reduction to be extremely effective in applications.

5 Integer Linear Programming and Branch-and-Cut

In this section we describe an algorithm for Weighted Cluster Editing, which is based on mathematical optimization. It relies on the following integer linear programming (ILP) formulation due to Grötschel and Wakabayashi [8].

Let x be a binary decision vector with $x_e = 1$ if edge e is part of the solution and $x_e = 0$ otherwise, for all $e \in E$. Then, an optimal solution to Weighted Cluster Editing can be found by solving

$$\text{minimize} \quad \sum_{e \in E} s(e) - \sum_{1 \le i < j \le n} s(ij) x_{ij} \tag{10}$$

$$\text{subject to} \quad + x_{ij} + x_{jk} - x_{ik} \le 1 \qquad \text{for all } 1 \le i < j < k \le n \tag{11}$$

$$+ x_{ij} - x_{jk} + x_{ik} \le 1 \qquad \text{for all } 1 \le i < j < k \le n \tag{12}$$

$$- x_{ij} + x_{jk} + x_{ik} \le 1 \qquad \text{for all } 1 \le i < j < k \le n \tag{13}$$

$$x_{ij} \in \{0, 1\} \qquad \text{for all } 1 \le i < j \le n \ . \tag{14}$$

The $3\binom{n}{3}$ *triangle inequalities* (11)–(13) of the ILP ensure that no conflict triple as shown in Fig. 4 (a) occurs in the solution. The above ILP formulation can already be used to solve instances of Weighted Cluster Editing to provable optimality.

A faster algorithm can be obtained by a mathematical analysis of the corresponding *clique partitioning polytope*. Using methods from polyhedral combinatorics, Grötschel and Wakabayashi have studied its facial structure and could identify a number of classes of facet-defining inequalities. As proposed by the authors, we concentrate on the *2-partition inequalities*

$$\sum_{i \in S, j \in T} x_{ij} - \sum_{i \in S, j \in S} x_{ij} - \sum_{i \in T, j \in T} x_{ij} \le \min\{|S|, |T|\} \ ,$$

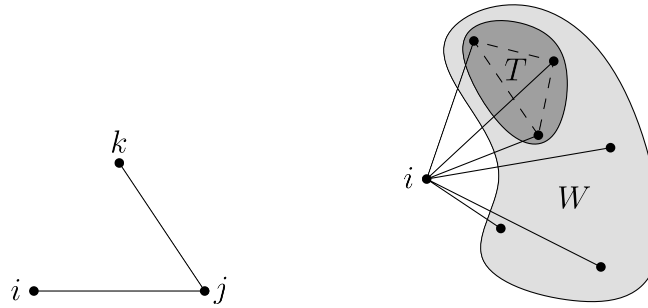

Fig. 4: Left: Forbidden conflict triple. Here, the corresponding triangle inequality is $x_{ij} + x_{jk} - x_{ik} = 2 > 1$. Right: Illustration of the heuristic separation of 2-partition inequalities.

where S and T are disjoint and nonempty subsets of V.

There is an exponential number of 2-partition inequalities. We therefore do not generate them at once but follow a *cutting plane* approach, adding 2-partition inequalities only if they are violated by a current fractional solution. We have implemented a variant of the iterative cutting plane method proposed by Grötschel and Wakabayashi. We start optimizing the LP relaxation (10) with an empty constraint set. Let x^* denote the vector corresponding to an intermediate solution of the linear programming relaxation. We first check whether x^* violates any triangle inequalities. If this is the case, we add the violated inequalities, resolve, and iterate. Otherwise, we check whether x^* is integral. If so, we stop, and x^* is an optimal solution. If x^* has fractional entries, we heuristically try to find violated 2-separation inequalities in the following manner, see also Fig. 4 (b):

For every node $i \in V$ we look at the nodes in $W := \{j \in V \setminus \{i\} \mid x^*_{ij} > 0\}$. Then, we pick a node $w \in W$ and iteratively construct a subset T of W, setting initially $T = \{w\}$ and adding nodes $k \in W$ to T if $x^*_{ik} - \sum_{j \in T} x^*_{jk} > 0$. Finally, we check whether

$$\sum_{j \in T} x^*_{ij} - \sum_{j \in T} \sum_{k \in T, k \neq j} x^*_{jk} > 1 \quad .$$

If this is the case, we add the violated 2-partition inequality [7]

$$\sum_{j \in T} x_{ij} - \sum_{j \in T} \sum_{k \in T, k \neq j} x_{jk} \leq 1 \quad .$$

If we find cutting planes in the separation procedure we iterate, otherwise we branch.

[7] Note that the heuristic separation procedure described in [8] contains a mistake, the term $-\sum_{i \in T, j \in T} x_{ij}$ is missing there, and the generated inequality leads to invalid cuts.

6 Datasets

In the absence of publicly available unweighted graph datasets that meet our requirements (note that the datasets used in [5] are far too small for our evaluations) we concentrate on the following two datasets:

Random unweighted graphs. Given a number of nodes n and parameter k, we uniformly select an integer $i \in [1, n]$ and define i nodes to be a cluster. We proceed in this way with the remaining $n \leftarrow n - i$ nodes until $n \leq 5$ holds: In this case, we assign all remaining n nodes to the last cluster. Starting from this transitive graph $G = (V, E)$ we choose k' distinct vertex pairs $uv \in \binom{V}{2}$ and delete or insert the edge uv in G. Here, k' is chosen slightly above the desired modification cost. Let k denote the minimum number of modifications to make G transitive, then $k \leq k'$. For instances where we cannot compute exact modification costs k, we estimate k using the average of upper and lower bound.

Protein similarity data. We also apply our algorithms to *weighted* instances that stem from biological data. Rahmann et al. [14] present a graph derived from protein similarity data: The vertices of our graph are more than 192 000 protein sequences from the COG database [18]. The similarity $S(u, v)$ of two proteins u, v is calculated from $\log_{10}$ E-values of bidirectional BLAST hits. An E-value threshold of 10^{-10} was used to indicate that two proteins are "sufficiently similar", so $s(uv) := S(u, v) - 10$. See [14] for more details.

The graph encoded by s contains 50 600 connected components: 42 563 components are of size 1 or 2, and 4 073 components are cliques of size ≥ 3. The remaining 3 964 components serve as our evaluation instances. Only 11 instances have more than 600 vertices. As a side comment, we mention that Wittkop et al. [20] evaluate several clustering methods for this application, and find that Weighted Cluster Editing methods show the best clustering quality.

Evaluation platform. All algorithms were implemented in C++, the branch-and-cut algorithm (ILP) uses the Concert interface to the commercial CPLEX solver 9.03. Running times were measured on an AMD Opteron-275 2.2 GHz with 6+ GB of memory.

7 Data Reduction Results

We now compare the performance of the unweighted kernel [9] and the weighted data reduction from Sec. 3 on the dataset of random *unweighted* graphs. To allow

for a fair comparison with the weighted data reduction, we merge all permanent edges of the unweighted kernel, resulting in an integer-weighted graph with even fewer vertices. This seems reasonable since both ILP and edge branching can handle integer-weighted input graphs. For the weighted data reduction, we first merge all critical cliques in the input graph. Next, we use weighted reduction rules plus the parameter-dependent reduction rules as described in Sec. 4. Despite the additional reduction steps, the reduced graph can have $4k_{\text{opt}}$ vertices for both approaches: A disjoint union of k paths of length 3 is not reduced by any reduction rule.

For our first evaluation, we concentrate on the weighted reduction strategy. For fixed $k = 2\,000$ and varying $n = 100, \ldots, 5\,000$ we study reduction ratio and absolute size of the resulting graph for $12\,000$ random instances. Results for n up to $1\,000$ are shown in Fig. 5. Similar results were obtained for larger n and other choices of k, data not shown. For fixed k, we find that the larger the graphs get, the better the reduction ratio is on average. Most graphs are either reduced down to a few vertices or stay unreduced. Only a few reduced graphs end up in a "twilight zone" between these extremes. This effective reduction is *not* due to the upper bound $n \leq 4k = 8\,000$: In fact, the absolute size of reduced graphs gets smaller when input graphs get larger. This might seem counterintuitive at first glance, but larger graphs show smaller relative defects, which allows weighted reduction rules to more "aggressively" merge or delete edges.

The above evaluation indicates that reduction results do not only depend on k and n directly, but even more so on the ratio k/n. In our second evaluation, we choose $n \in \{100, 500, 1\,500, 2\,000\}$ and set $k := c \cdot n$, for varying factors $c \in \{0.25, 0.5, \ldots, 2.0\}$. For every combination of n and k we create 10 input graphs

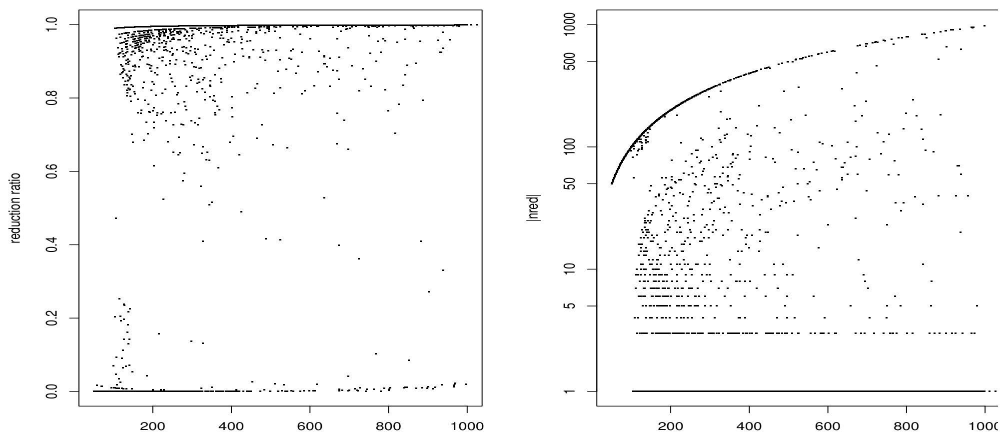

Fig. 5: Data reduction for fixed $k = 2\,000$ and variable graph size n: Left plot shows reduction ratio vs. n, right plot shows reduced graph size n_{red} vs. n. Both plots show $11\,000$ instances.

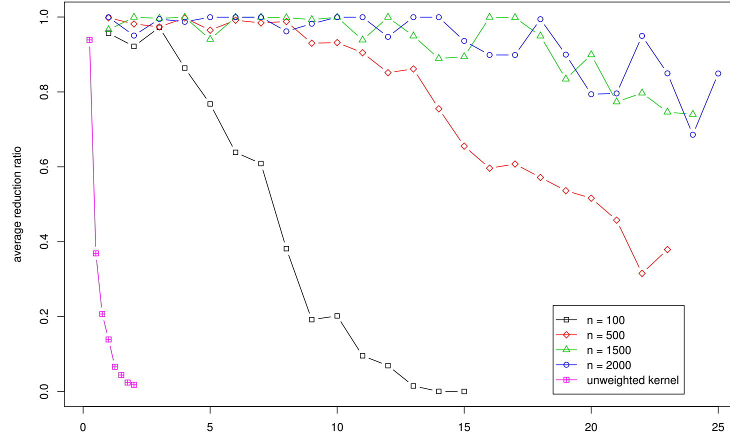

Fig. 6: Average reduction ratio vs. ratio k/n for $n = 100, 500, 1\,500, 2\,000$. Note that the unweighted kernel is practically independent from graph size n.

and apply the unweighted kernel. See Fig. 6 for resulting reduction ratios. We find reduction ratios of the unweighted kernel to be mostly independent of the actual graph size n. The unweighted kernel is very effective for graphs with $k \leq \frac{1}{2}n$, and graphs are downsized to half of their original size on average. For $k \geq 2n$ no reduction is observed. To evaluate the weighted data reduction we again set $k := c \cdot n$, for factors $c \in \{1, 2, \ldots, 25\}$. For every combination of k and graphs size with $n < 1\,000$ ($n \geq 1\,000$) we create 50 (20) input graphs. See again Fig. 6 for reduction ratios. We observe that the weighted data reduction is much more effective than the unweighted kernel. Here, the reduction ratio depends strongly on the ratio k/n and, less pronounced, also on the graph size n. We observe that large graphs of size $n = 2\,000$ are reduced by 80% for $k = 25n$ and by more than 90% for $k = 18n$. Performance of our data reduction on parameters k and n is more complex than the simple ratio k/n suggests: The number of edges and non-edges in the input is quadratic in n, but many reduction rules only consider the neighborhood of a constant number of vertices and, hence, a number of edges linear in n. We find that data reduction performance can be roughly estimated using the value k/n^c for $c \approx 1.5$, we omit further details.

Figure 7 shows the ratio of input graphs being reduced by more than 90%. For the weighted data reduction, we vary the number of vertices n and set $k := cn$ for $c = 5, 10, 15, 20$.[8] For the unweighted kernel we observe significantly reduced graphs only for $c = 0.25$. Note that for the weighted data reduction, the ratio of significantly reduced graphs increases for larger graphs.

[8] We also performed experiments for all $c = 0.25, 0.5, 0.75, 1, 2, 3, \ldots, 25$ but find that results follow the same trend, data not shown.

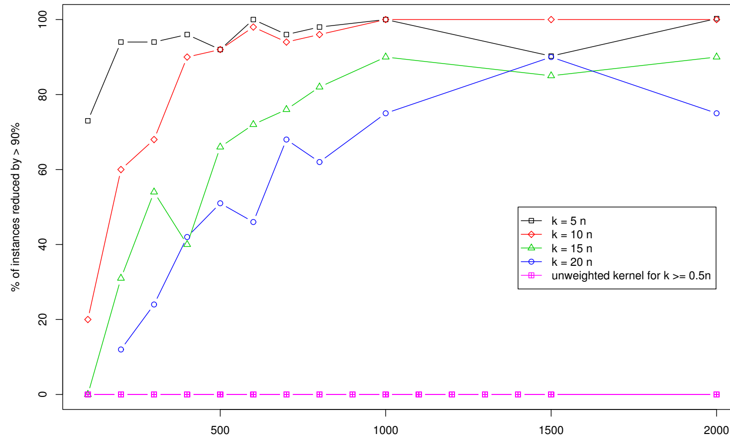

Fig. 7: Percentage of instances which are reduced more than 90% for varying graph size n and $k = cn$ for $c = 5, 10, 15, 20$.

In case we only use parameter-independent reduction rules from Sec. 3, the weighted reduction is only slightly better than the unweighted kernel, data not shown. We find the combination of parameter-dependent data reduction and lower/upper bounds to be the reason for the effective reduction. To justify this claim, we consider 50 random graphs with $n = 100$ vertices, and either 25 or 100 edge modifications, respectively. We first merge critical cliques in every instance. Using only Rules 1–3 for parameter-independent data reduction we reach average reduction ratios of 0.655 (for $k = 25$) and 0.130 (for $k = 100$). Using Rule 4 in addition to the above Rules 1–3 does not significantly improve reduction ratios. Using Rules 1–5 leads to average reduction ratios of 0.783 (for $k = 25$) and 0.131 (for $k = 100$). But if we use, instead of Rules 1–5, only the parameter-dependent rules in combination with lower and upper bound, then reduction ratios significantly increase to 0.987 (for $k = 25$) and 0.948 (for $k = 100$).

To this end, we also estimate the accuracy of our lower and upper bound. We find that our lower bound has a relative error of 1.7 % on average, and the upper bound has a relative error of 17.9 % on average. Calculating tighter upper bounds by, say, a heuristic such as FORCE [20] will further improve the performance of our weighted data reduction.

Running times of data reduction. Using the unweighted kernel, most of the instance were reduced in less than a minute, instances of size 2 000 in about one hour computation time. Graphs with k around n need more computation time than graphs with lower or greater k since reduction rules are checked very often but rarely applied. Running times of the weighted data reduction are equally high for k around $5n$,

whereas for k around $20n$ running times are slightly higher. Making the data reduction run fast has not been the focus of our research, because we assumed running times of data reduction to be negligible to the following (exponential-time) step of the analysis. We do not report details and just note that reducing graphs of size 500 took 51.23 seconds on average but at most 9.09 minutes, whereas reducing graphs of size 2 000 took 1.61 hours on average and at most 23.69 hours. Our experiments show that many graphs are reduced to trivial or very small instances, so the exponential-time step of the algorithm has very small running times. We believe that by optimizing our data reduction algorithm we can achieve significantly reduced running times in the future.

Data reduction results for weighted instances. We also apply our weighted data reduction strategy to the protein similarity data. In this case, however, parameter k does not reflect the complexity of the instance: here, edges might have modification costs ≤ 1 and, hence, the total modification costs may equal 1 even if thousands of edge modifications are necessary. Instead, we use the number of edge modifications as a complexity measure of an instance. Table 1 shows results of the weighted data reduction. We find that the data reduction reduces weighted instances not as much as unweighted instances. This is mainly caused by the fact that our lower and upper bounds are not as tight as for the unweighted case. In detail, our lower bound has a relative error of 3.6 % on average, and the upper bound had a relative error of 54.7 % on average. In contrast to our findings for unweighted instances, we observe that larger graphs are reduced less effectively than smaller graphs. This can be attributed to the fact that the number of edge modifications is growing faster than linear. Furthermore, parameter-independent reduction rules are less efficient on large weighted graphs, since it gets less likely that an edge weight is greater than a sum over $O(n)$ other edge weights.

graph size n	3 - 49	50 - 99	100 - 149	150 - 199	200 - 249	250-299	300+
No. of instances	3453	341	78	22	24	20	25
av. reduction ratio	0.84	0.89	0.73	0.68	0.66	0.58	0.35

Table 1. Protein similarity data: Average reduction ratio for different graph size n.

8 Integer Linear Programming and Search Tree Results

We want to compare the performance of the FPT branching algorithm approach and the ILP-based branch-and-cut method. For this evaluation, we use random

unweighted graphs and reduce them by the weighted data reduction. Reduced graphs are sorted into bins for sizes $n \approx 100, 200, \ldots, 900$ and costs $k \approx 1n, 2n, \ldots, 10n$. Every bin contains 28 graphs on average. As described in Sec. 7, most graphs are either reduced completely or not at all, so building these reduced graphs is computationally expensive. For each reduced instance we apply the FPT branching algorithm and ILP with an upper limit of 6 hours of running time. For average running times, we count unfinished instances as 6 hours. Figure 8 shows the resulting running times.

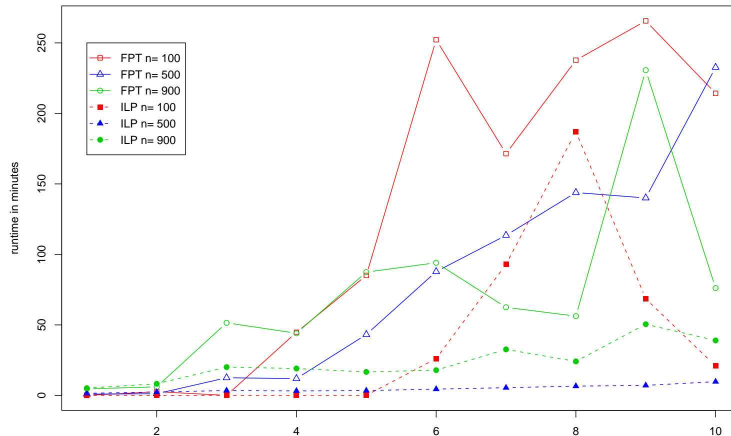

Fig. 8: Running times of FPT branching and ILP branch-and-cut in seconds, for varying ratio k/n and $n = 100, 500, 900$.

Running times of the fixed-parameter algorithm most strongly depend on the ratio k/n and, to a smaller extent, on the actual parameter k. Instances with modification cost $k \approx 5n$ need about one hour of computation to be solved. Note that running times for FPT branching are much better than worst-case running time analysis suggests, and dependence on the actual parameter k is much less pronounced than expected. We believe that this is mainly due to the good lower bound estimation for the parameter-dependent data reduction used in interleaving, and also the vertex merging operation.

The limiting factor for the ILP algorithm is the size of the input graph whereas dependence on modification costs k is much less pronounced, with the exception of the peak around $k = 8n$ for the instances with $n = 100$. A possible explanation for this peak is that particularly difficult instances arise around some ratio k/n^c with $c \in [1, 2]$, which corresponds to $k/n \approx 8$ for $n = 100$. We expect a similar behavior also for $n = 500$ and $n = 900$ with corresponding ratios k/n far larger than 10 and, thus, outside the data ranges shown in this plot. Besides these difficult

Size red. instance	3–49	50–99	100–149	150–199	200–249	250–299	300–1400
No. red. instances	297	52	16	10	9	2	19
Unfinished FPT	0	0	1	1	2	2	15
time* FPT	125 ms	23.9 s	44.1 min	4.52 min	47.3 min	n/a	8.98 min
time FPT	125 ms	23.9 s	2.19 h	2.47 h	5.95 h	24 h	18.98 h
Unfinished ILP	0	0	0	0	1	1	10
time* ILP	17 ms	6.97 s	5.30 min	18.20 min	76.2 min	6.85 min	1.67 h
time ILP	17 ms	6.97 s	5.30 min	18.20 min	3.80 h	12.06 h	13.42 h

Table 2. Running times on reduced protein similarity data for FPT branching and ILP. Instances that did not finish after 24 hours of computation were ignored (time*) or set to 24 hours (time) for average running time computation, respectively.

instances, we observe that small instances with only 100 vertices are solved within seconds, and medium graphs of size 500 are solved within minutes. We find that ILP is well-suited for medium-size Cluster Editing instances and clearly outperforms the fastest fixed-parameter algorithm examined in this paper for these instances. We stress that ILP requires preprocessing by parameter-independent data reduction since its performance is strongly dependent on the graph size n. Only for large graphs with very low modification costs $k \leq 2n$, the FPT algorithm may outperform the cutting plane algorithm. High running times of the cutting plane approach for large instances are, however, mostly *not* due to their structural complexity but to the large number of triangle inequalities that have to be checked in the current implementation. Once a better separation strategy has been found, we expect the branch-and-cut algorithm to perform well even on larger instances.

The FPT search tree algorithm can easily be modified to enumerate all optimal solutions, whereas this is more complicated using the ILP. For our random instances we find that the vast majority of graphs only have one optimal solution, with the exception of small graphs with large edit costs, for which a huge number of optimal solutions can exist.

Results for weighted instances. We now compare the performance of FPT branching and ILP using protein similarity data. We reduced all instances in the protein dataset using our weighted data reduction strategy, resulting in 365 non-trivial instances. In Tab. 2 we report running times of the two methods, where we stopped computation after 24 hours. The FPT branching algorithm is usually fast enough for graphs with up to 200 vertices, but for most larger graphs, no solution can be computed within the time limit. In contrast, the ILP algorithm was able to solve most instances with less than 500 vertices in only some minutes.

Using data reduction and ILP, we were able to solve all but 12 of the 3 964 instances in less than 24 hours, and all but 19 in less than 60 minutes. This implies

that we can process the complete dataset of 3 964 protein similarity instances in less than two days, and guarantee an optimal solutions for 99.52 % of the instances.

9 Conclusion

Our results demonstrate that computing exact solutions of Cluster Editing instances is no longer limited to small or almost transitive graphs, thus invalidating what has often been reported in previous work. Using data reduction for Weighted Cluster Editing in combination with parameter-dependent rules and lower/upper bounds strongly improves the ability to shrink down input instances in polynomial running time. Even complex input graphs that are far from transitive and that have modification costs much larger than the number of vertices, can often be reduced very effectively.

We also compared the fastest known FPT branching algorithm for Cluster Editing against a branch-and-cut approach for this problem, based on the ILP formulation by Grötschel and Wakabayashi. Both algorithms perform well, and reduced graphs with hundreds of vertices and thousands of edge modifications are processed in acceptable running time. In particular, our results suggest that mathematical optimization techniques based on ILP formulations are suitable for solving large instances with many modifications. Here, realizing alternative separation strategies, as, for instance, proposed in [12] for the related comparability editing problem, seems to be a promising way to solve even larger instances to provable optimality in reasonable running time.

We believe that better upper bounds will allow even larger instances of (unweighted and weighted) Cluster Editing to be solved exactly in the future. We plan to implement a web interface for our tools in order to give a large community access to our exact clustering tools.

Acknowledgments. We thank Anke Truß and Bao Bui for many helpful ideas and comments. We thank Svenja Simon for support with evaluation and implementation. S. Briesemeister gratefully acknowledges financial support from LGFG Promotionsverbund "Pflanzliche Sensorhistidinkinasen" at the University of Tübingen.

References

1. A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *J. Comput. Biol.*, 6(3-4):281–297, 1999.
2. S. Böcker, S. Briesemeister, Q. B. A. Bui, and A. Truß. A fixed-parameter approach for weighted cluster editing. In *Proc. of Asia-Pacific Bioinformatics Conference (APBC 2008)*, volume 5

of *Series on Advances in Bioinformatics and Computational Biology*, pages 211–220. Imperial College Press, 2008.

3. S. Böcker, S. Briesemeister, Q. B. A. Bui, and A. Truß. Going weighted: Parameterized algorithms for cluster editing. In *Proc. of Conference on Combinatorial Optimization and Applications (COCOA 2008)*, volume 5165 of *Lect. Notes Comput. Sc.*, pages 1–12. Springer, 2008.
4. M. Charikar, V. Guruswami, and A. Wirth. Clustering with qualitative information. *J. Comput. Syst. Sci.*, 71(3):360–383, 2005.
5. F. Dehne, M. A. Langston, X. Luo, S. Pitre, P. Shaw, and Y. Zhang. The cluster editing problem: Implementations and experiments. In *Proc. of International Workshop on Parameterized and Exact Computation (IWPEC 2006)*, volume 4169 of *Lect. Notes Comput. Sc.*, pages 13–24. Springer, 2006.
6. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, 39(4):321–347, 2004.
7. J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. *Theor. Comput. Syst.*, 38(4):373–392, 2005.
8. M. Grötschel and Y. Wakabayashi. A cutting plane algorithm for a clustering problem. *Math. Program.*, 45:52–96, 1989.
9. J. Guo. A more effective linear kernelization for cluster editing. *Theor. Comput. Sci.*, 410(8-10):718–726, 2009.
10. G. A. Kochenberger, F. Glover, B. Alidaee, and H. Wang. Clustering of microarray data via clique partitioning. *J. Comb. Optim.*, 10(1):77–92, 2005.
11. M. Křivánek and J. Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Inform.*, 23(3):311–323, 1986.
12. R. Müller. On the partial order polytope of a digraph. *Mathematical Programming*, 73:31–49, 1996.
13. R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.
14. S. Rahmann, T. Wittkop, J. Baumbach, M. Martin, A. Truß, and S. Böcker. Exact and heuristic algorithms for weighted cluster editing. In *Proc. of Computational Systems Bioinformatics (CSB 2007)*, volume 6, pages 391–401, 2007.
15. R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. *Discrete Appl. Math.*, 144(1–2):173–182, 2004.
16. R. Sharan, A. Maron-Katz, and R. Shamir. CLICK and EXPANDER: a system for clustering and visualizing gene expression data. *Bioinformatics*, 19(14):1787–1799, Sep 2003.
17. M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 4:585–591, 1997.
18. R. L. Tatusov, N. D. Fedorova, J. D. Jackson, A. R. Jacobs, B. Kiryutin, E. V. Koonin, D. M. Krylov, R. Mazumder, S. L. Mekhedov, A. N. Nikolskaya, B. S. Rao, S. Smirnov, A. V. Sverdlov, S. Vasudevan, Y. I. Wolf, J. J. Yin, and D. A. Natale. The COG database: an updated version includes eukaryotes. *BMC Bioinformatics*, 4:41, 2003.
19. A. van Zuylen and D. P. Williamson. Deterministic algorithms for rank aggregation and other ranking and clustering problems. In *Proc. of Workshop on Approximation and Online Algorithms (WAOA 2007)*, volume 4927 of *Lect. Notes Comput. Sc.*, pages 260–273. Springer, 2008.
20. T. Wittkop, J. Baumbach, F. Lobo, and S. Rahmann. Large scale clustering of protein sequences with FORCE – a layout based heuristic for weighted cluster editing. *BMC Bioinformatics*, 8(1):396, 2007.