Exact Algorithms for Cluster Editing: Evaluation and Experiments

Sebastian Böcker[1,2], Sebastian Briesemeister[3], and Gunnar W. Klau[4,5]

[1] Institut für Informatik, Friedrich-Schiller-Universität Jena, Germany
boecker@minet.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] Department of Mathematics and Computer Science, Freie Universität Berlin, Germany
gunnar@math.fu-berlin.de
[5] DFG Research Center Matheon, Berlin, Germany

Abstract. We present empirical results for the Cluster Editing 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 $|V|$. 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.

1 Introduction

The Cluster Editing problem is defined as follows: Let $G = (V, E)$ be an undirected, loop-less 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.

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 [14]. Clustering algorithms for microarray data such as CAST [1] and CLICK [15] 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,

C.C. McGeoch (Ed.): WEA 2008, LNCS 5038, pp. 289–302, 2008.

see for instance [18]. 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 tuple, 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 [17]. 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 $4k_{\text{opt}}$. 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 $k = 5n$ in about an hour, where n, k are size and parameter of the reduced instance. The ILP approach solves instances with $n = 1\,000$ 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.

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. Such graphs are also called *transitive*.

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.

3 Data Reduction and Branching Algorithm

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.

Parameter-independent data reduction. 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*.

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 reduce k by $\min\{|s(uw)|, |s(vw)|\}$ [2].

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). If an edge uv with $s(uv) < 0$ satisfies $|s(uv)| \geq \sum_{w \in N(u)} s(uw)$ then set uv to forbidden.

Rule 2 (heavy edge rule, single end). If an edge uv satisfies $s(uv) \geq \sum_{w \in V \setminus \{u,v\}} |s(uw)|$ then merge vertices u, v.

Rule 3 (heavy edge rule, both ends). If an edge uv satisfies $s(uv) \geq \sum_{w \in N(u) \setminus \{v\}} s(uw) + \sum_{w \in N(v) \setminus \{u\}} s(vw)$, then merge u, v.

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.

Rule 5 (similar neighborhood). For an edge uv we define $N_u := N(u) \setminus (N(v) \cup \{v\})$, $N_v := N(v) \setminus (N(u) \cup \{u\})$ as the exclusive neighborhoods, and 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 $\Delta_v := s(v, N_v) - s(v, N_u)$. If uv satisfies

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

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

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 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.

Proving the correctness of Rule 5 is rather involved, we defer the details to the full paper. This rule turns out to be highly efficient but its computation is expensive: For integer-weighted graphs, we can find the maximum (1) using dynamic programming in time $O(|W|\, Z)$ where $Z := \sum_{w \in W} (s(uw) + s(vw))$. For real-valued edge weights we can only approximate the calculation by multiplying with a blowup factor and rounding. In practice, we use Rule 5 only in case no other rules can be applied.

Using parameter-dependent data reduction. We 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

$$icf(uv) = \sum_{w \in N(u) \cap N(v)} \min\{s(uw), s(vw)\}, \quad icp(uv) = \sum_{w \in N(u) \triangle N(v)} \min\{|s(uw)|, |s(vw)|\},$$

where $A \triangle B$ denotes the symmetric set difference of A and B. If $icp(uv)$ or $icf(uv)$ exceed k, we can set uv to "forbidden" or "permanent", respectively. In the latter case, we merge u,v and reduce k by $icp(uv)$ accordingly. We can also remove isolated cliques.

As an algorithm-engineering technique, we now describe fast methods to compute a 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 tuple 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 tuples uv according to the ratio $r(uv)$, then go through this sorted list from smallest to largest ratio. This leads to a 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.

We can use such lower bounds to make induced costs $icf(uv)$ and $icp(uv)$ 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. Then, we can set an edge to "forbidden" or "permanent" if $icp(uv) > k - b(G, uv)$ or $icf(uv) > k - b(G, uv)$ holds, resp.

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 [18]. For this study, we calculate an upper bound using a greedy approach that iteratively searches for edges where reduction rules almost apply. We find this reduction to be extremely effective in applications.

Branching strategy. After parameter-independent data reduction, the remaining instance can be solved using a branching tree strategy. In these algorithms, we identify a conflict triple and then branch into sub-cases to repair this conflict. In practice, branching strategies that do merge vertices clearly outperform branching strategies that do not [2]. 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,[1]

[1] The branching number is the root of the characteristic polynomial and governs the asymptotic size of the search tree, see e.g. [12] for details.

then the resulting search tree has size $O(2^k)$. To find an optimal solution we call the algorithm repeatedly, increasing k in an interval defined by lower and upper bound for this problem instance. While traversing the search tree, we apply reduction rules in every node of the search tree. The simple Rules 1–3 and parameter-dependent rules 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 find an edge with minimal branching number, we approximate log branching numbers using two rational functions.

4 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

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} s(e) - \sum_{1 \le i < j \le n} s(ij) x_{ij} && && (2)\\
\text{subject to} \quad & + x_{ij} + x_{jk} - x_{ik} \le 1 && \text{for all } 1 \le i < j < k \le n && (3)\\
& + x_{ij} - x_{jk} + x_{ik} \le 1 && \text{for all } 1 \le i < j < k \le n && (4)\\
& - x_{ij} + x_{jk} + x_{ik} \le 1 && \text{for all } 1 \le i < j < k \le n && (5)\\
& x_{ij} \in \{0, 1\} && \text{for all } 1 \le i < j \le n \ . && (6)
\end{aligned}$$

The $3\binom{n}{3}$ *triangle inequalities* (3)–(5) of the ILP ensure that no conflict triple 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 problem polyhedron. Using methods from polyhedral combinatorics, Grötschel and Wakabayashi have studied the facial structure of the corresponding *clique partitioning polytope*. They have identified a number of classes of facet-defining inequalities. As proposed by the authors, we concentrate on the *2-partition inequalities*

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

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 (2) 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:

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

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

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

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

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

5 Datasets

In the absence of publicly available 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 tuples $uv \in \binom{V}{2}$ and delete or insert the edge uv in G. 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 upper, lower bounds, and general observations.

Protein similarity data. We also apply our algorithms to *weighted* instances that stem from biological data. Rahmann et al. [13] present a set of graphs derived from protein similarity data: The vertices of our graph are more than 192 000 protein sequences from the COG database [16]. The similarity $S(u, v)$ of two proteins u, v is calculated from $\log_{10}$ E-values of bidirectional BLAST hits. We use an E-value of 10^{-10} as our threshold indicating that two proteins are "sufficiently similar", so $s(uv) := S(u, v) - 10$. See [13] 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. [18] 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.

6 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. 3. 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 11 000 random instances. Results for n up to 1 000 are shown in Fig. 1. Similar results were obtained for larger n and other choices of k, data not shown. As one can see, the larger the graphs get, the better the reduction ratio on average. Most graphs are either reduced down to a

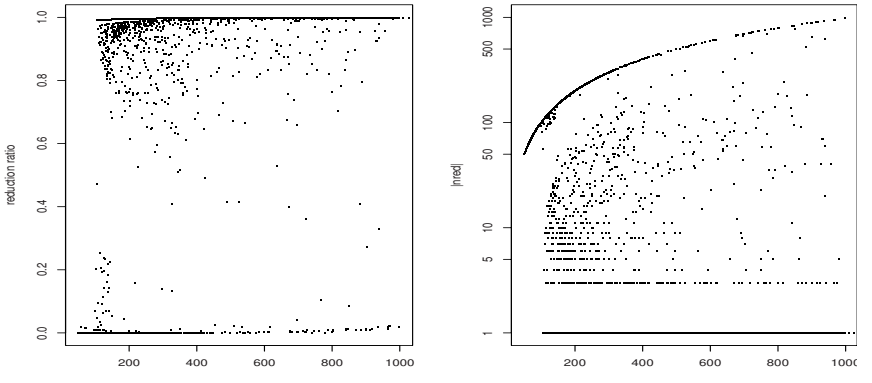

Fig. 1. Data reduction for fixed $k = 2\,000$ and variable graph size n: Left plot shows reduction ratio vs. n, right plot shows reduced graph size n_{red} vs. n. Both plots show 11000 instances.

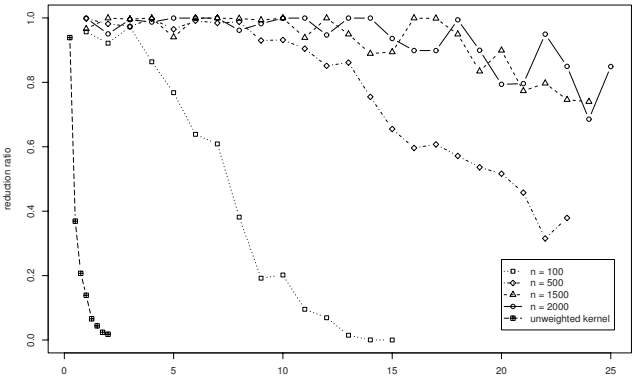

Fig. 2. Average reduction ratio vs. ratio k/n for $n = 100, 500, 1\,500, 2\,000$. Note that the unweighted kernel is practically independent from graph size n.

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 and apply the unweighted kernel. See Fig. 2 for resulting reduction ratios. We find reduction ratios of the unweighted kernel to be mostly independent of the actual graph size. 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. 2 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 observed that large graphs of size $n = 2\,000$ are reduced by 80% for $k = 25$ and by more than 90% for $k = 18n$.

Figure 3 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 $c = 5, 10, 15, 20$.[2] For the unweighted kernel we observe significantly reduced graphs only for $c = 0.25$. Most interestingly, for the weighted data reduction, the ratio of significantly reduced graphs increases for larger graphs.

[2] 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.

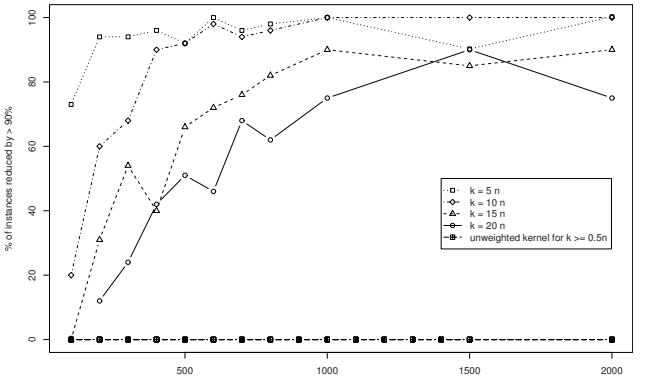

Fig. 3. Percentage of instances which are reduced more than 90% for varying graph size n and $k = cn$ for $c = 5, 10, 15, 20$

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 this end, we 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 had a relative error of 17.9 % on average. Calculating tighter upper bounds by, say, a heuristic such as FORCE [18] 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.

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

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

7 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 $k \approx 1n, 2n, \ldots, 10n$, and every bin contains 28 graphs on average. As described in Sec. 6, 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 4 shows the resulting running times.

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. 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

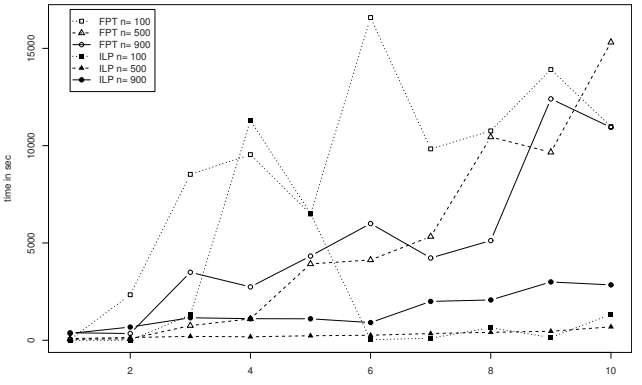

Fig. 4. Running times of FPT branching and ILP branch-and-cut in seconds, for varying ratio k/n and $n = 100, 500, 900$

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 for average running time computation.

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
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

Cluster Editing instances and clearly outperforms the fastest fixed-parameter algorithm for these instances. We stress that ILP requires preprocessing by parameter-independent data reduction since its performance is solely dependent on the input graph size. 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.

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. 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 24

hours. In contrast, the ILP algorithm was able to solve most instances with less than 500 vertices in only some minutes.

8 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 ILP is suitable for solving large instances with many modifications.

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 will make the source code of our reduction and cluster editing tools, as well as the data used in this article publicly available. Furthermore, we plan to implement a web interface for our tools in order to give a large community access to our exact clustering tools and to facilitate comparison and evaluation.

Acknowledgments. 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. Ben-Dor, A., Shamir, R., Yakhini, Z.: Clustering gene expression patterns. J. Comput. Biol. 6(3-4), 281–297 (1999)
2. Böcker, S., Briesemeister, S., Bui, Q.B.A., Truß, A.: A fixed-parameter approach for weighted cluster editing. In: Proc. of Asia-Pacific Bioinformatics Conference (APBC 2008). Series on Advances in Bioinformatics and Computational Biology, vol. 5, pp. 211–220. Imperial College Press (2008)
3. Böcker, S., Briesemeister, S., Bui, Q.B.A., Truß, A.: Going weighted: Parameterized algorithms for cluster editing (Manuscript) (2008)
4. Charikar, M., Guruswami, V., Wirth, A.: Clustering with qualitative information. J. Comput. Syst. Sci. 71(3), 360–383 (2005)
5. Dehne, F., Langston, M.A., Luo, X., Pitre, S., Shaw, P., Zhang, Y.: The cluster editing problem: Implementations and experiments. In: Bodlaender, H.L., Langston, M.A. (eds.) IWPEC 2006. LNCS, vol. 4169, pp. 13–24. Springer, Heidelberg (2006)

6. Gramm, J., Guo, J., Hüffner, F., Niedermeier, R.: Automated generation of search tree algorithms for hard graph modification problems. Algorithmica 39(4), 321–347 (2004)
7. Gramm, J., Guo, J., Hüffner, F., Niedermeier, R.: Graph-modeled data clustering: Fixed-parameter algorithms for clique generation. Theor. Comput. Syst. 38(4), 373–392 (2005)
8. Grötschel, M., Wakabayashi, Y.: A cutting plane algorithm for a clustering problem. Math. Program. 45, 52–96 (1989)
9. Guo, J.: A more effective linear kernelization for Cluster Editing. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 36–47. Springer, Heidelberg (2007)
10. Kochenberger, G.A., Glover, F., Alidaee, B., Wang, H.: Clustering of microarray data via clique partitioning. J. Comb. Optim. 10(1), 77–92 (2005)
11. Křivánek, M., Morávek, J.: NP-hard problems in hierarchical-tree clustering. Acta Inform. 23(3), 311–323 (1986)
12. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press, Oxford (2006)
13. Rahmann, S., Wittkop, T., Baumbach, J., Martin, M., Truß, A., Böcker, S.: Exact and heuristic algorithms for weighted cluster editing. In: Proc. of Computational Systems Bioinformatics (CSB 2007), vol. 6, pp. 391–401 (2007)
14. Shamir, R., Sharan, R., Tsur, D.: Cluster graph modification problems. Discrete Appl. Math. 144(1–2), 173–182 (2004)
15. Sharan, R., Maron-Katz, A., Shamir, R.: CLICK and EXPANDER: a system for clustering and visualizing gene expression data. Bioinformatics 19(14), 1787–1799 (2003)
16. Tatusov, R.L., Fedorova, N.D., Jackson, J.D., Jacobs, A.R., Kiryutin, B., Koonin, E.V., Krylov, D.M., Mazumder, R., Mekhedov, S.L., Nikolskaya, A.N., Rao, B.S., Smirnov, S., Sverdlov, A.V., Vasudevan, S., Wolf, Y.I., Yin, J.J., Natale, D.A.: The COG database: an updated version includes eukaryotes. BMC Bioinformatics 4, 41 (2003)
17. van Zuylen, A., Williamson, D.P.: Deterministic algorithms for rank aggregation and other ranking and clustering problems. In: Proc. of Workshop on Approximation and Online Algorithms (WAOA 2007). Lect. Notes Comput. Sc., vol. 4927, pp. 260–273. Springer, Heidelberg (2008)
18. Wittkop, T., Baumbach, J., Lobo, F., Rahmann, S.: Large scale clustering of protein sequences with FORCE – a layout based heuristic for weighted cluster editing. BMC Bioinformatics 8(1), 396 (2007)