Theoretical Computer Science 410 (2009) 5467–5480

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Going weighted: Parameterized algorithms for cluster editing

S. Böcker [a], S. Briesemeister [b], Q.B.A. Bui [a], A. Truss [a,*]

[a] Lehrstuhl für Bioinformatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, 07743 Jena, Germany
[b] Div. for Simulation of Biological Systems, ZBIT/WSI, Eberhard Karls Universität Tübingen, Germany

ARTICLE INFO

Keywords:
Exact algorithms
Fixed-parameter tractability
Data clustering

ABSTRACT

The goal of the Cluster Editing problem is to make the fewest changes to the edge set of an input graph such that the resulting graph is a disjoint union of cliques. This problem is NP-complete but recently, several parameterized algorithms have been proposed. In this paper, we present a number of surprisingly simple search tree algorithms for Weighted Cluster Editing assuming that edge insertion and deletion costs are positive integers. We show that the smallest search tree has size $O(1.82^k)$ for edit cost k, resulting in the currently fastest parameterized algorithm, both for this problem and its unweighted counterpart. We have implemented and compared our algorithms, and achieved promising results.[1]

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

The Weighted Cluster Editing problem is defined as follows: Let $G_w = (V, E)$ be an undirected graph. For every vertex pair $\{u, v\} \in \binom{V}{2} = \{\{u, v\} : u, v \in V, u \neq v\}$ we know the cost of deleting $\{u, v\}$ from G_w in case $\{u, v\} \in E$, or inserting $\{u, v\}$ into G_w in case $\{u, v\} \notin E$. Our task is to transform G_w into a transitive graph (a disjoint union of cliques) by applying edge modifications with minimum total cost. For our theoretical analysis, we assume that all pairs have *non-zero integer weight*. In the unweighted Cluster Editing problem, insertion or deletion cost are one for each vertex pair.

In application, the above task corresponds to clustering objects, that is, partitioning a set of objects into homogeneous and well-separated subsets. Clustering data still represents a key step of numerous biological and medical problems, such as class discovery for tissue identification using gene expression data. Here, a clustering corresponds to a vertex disjoint union of cliques. The input graph is corrupted and we have to clean (edit) the graph to reconstruct the clustering [14] under the *parsimony criterion*.

Previous work. NP-hardness of the unweighted Cluster Editing problem [14] was proven by Křivánek and Morávek [10]. Several heuristics were developed for Weighted Cluster Editing or rely on its graph-theoretic intuition, including CLICK [15] and FORCE [18]. The unweighted Cluster Editing problem is APX-hard and has a constant-factor approximation of 2.5 [17]. To find exact solutions, Grötschel and Wakabayashi [6] formulated the problem as an Integer Linear Program. The parameterized complexity of unweighted Cluster Editing, using the minimum number of edge modifications as the parameter k, is well-studied: Until recently, the fastest implemented algorithm had running time $O(2.27^k + n^3)$ on an n-vertex graph [5,8], while in theory, the best known algorithm has running time $O(1.92^k + n^3)$ [7]. Guo [9] presented a linear problem kernel. In contrast, the fixed-parameter tractability of Cluster Editing with "don't care edges", that is, edges whose modification cost is zero, is still an open problem [3].

* Corresponding address: Mathematics and Computer Science, Lehrstuhl fur Bioinformatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, 07743 Jena, Germany. Tel.: +49 3641 946453; fax: +49 3641 946452.

E-mail addresses: boecker@minet.uni-jena.de (S. Böcker), briese@informatik.uni-tuebingen.de (S. Briesemeister), bui@minet.uni-jena.de (Q.B.A. Bui), truss@minet.uni-jena.de (A. Truss).

[1] This is an extended version of two articles published in: Proc. of the 6th Asia Pacific Bioinformatics Conference, APBC 2008, in: Series on Advances in Bioinformatics and Computational Biology, vol. 5, Imperial College Press, pp. 211–220; and in: Proc. of the 2nd Conference on Combinatorial Optimization and Applications, COCOA 2008, in: LNCS, vol. 5038, Springer, pp. 289–302.

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.05.006

Our contributions. We present a problem kernel for Weighted Cluster Editing and branching strategies with search trees of sizes $O(3^k)$, $O(2.42^k)$, $O(2^k)$, and $O(1.82^k)$, where the parameter k is the minimum total cost of edge modifications. We concentrate on the case that edge insertion and deletion costs are positive integers, and sketch how to adopt our results for real-valued graphs where necessary.

In particular, we introduce a new branching strategy that is surprisingly simple, and show that the resulting search tree is of size $O(2^k)$. We then refine our analysis and show that, by accurately choosing edges to branch on, we obtain running time $O(1.82^k + n^3)$. Our algorithm is the fastest known for unweighted Cluster Editing and improves on the $O(1.92^k + n^3)$ algorithm in [7]. To the best of our knowledge, this is one of the few cases where a weighted problem allows for a more efficient fixed-parameter algorithm than its unweighted counterpart.

In Section 5, we compare running times of our algorithms for Weighted Cluster Editing. In our comparison, we use both simulated graphs and graphs that stem from protein similarity data and aim at clustering homologous proteins.

We find that the data reduction step of merging vertices significantly reduces running times and, in contrast to what theoretical bounds suggest, the $O(3^k)$ strategy usually outperformed the $O(2.42^k)$ strategy. The $O(2^k)$ and $O(1.82^k)$ strategies were significantly faster.

2. Preliminaries

Let V be the set of objects to be clustered, corresponding to the vertices of the graph, and let $n := |V|$. In this work, we consider only undirected graphs without self-loops and multiple edges. For brevity, we write uv as shorthand for an unordered pair $\{u, v\} \in \binom{V}{2}$. Let $s : \binom{V}{2} \to \mathbb{Z}$ be a *weight function* that encodes the input graph: For $s(uv) > 0$ a pair uv is an edge of the graph and has deletion cost $s(uv)$, while for $s(uv) < 0$, the pair uv is not an edge of the graph (we call it a *non-edge*) and has insertion cost $-s(uv)$. If $s(uv) = 0$, we call uv a *zero-edge*. Note that there are no zero-edges in the input graph, so that each pair of vertices is either an edge or a non-edge whose edit cost (deletion or insertion cost) is a positive integer. This is necessary solely to achieve provable running times. Nonetheless, zero-edges can appear in the course of computation and require additional attention when analyzing the algorithm.

When analyzing connected components, we regard zero-edges as non-existing. Throughout this paper, we assume that circles and paths do not contain zero-edges. A circle of length three is also called a *triangle*. We say that $C \subseteq V$ is a *clique* in an integer-weighted graph if all pairs $uv \in \binom{C}{2}$ are edges. If all vertex pairs of a connected component are either edges or zero-edges, we call it a *weak clique*. If all connected components of a graph are weak cliques, it is called *transitive*. Weak cliques in a transitive graph are also called *clusters*. An unweighted graph $G = (V, E)$ is transitive if and only if there exists no *conflict triple* in G, that is, three vertices vuw such that $vu, uw \in E$ but $vw \notin E$. Unfortunately, there exists no direct analogue of this statement for integer-weighted graphs. Vertices vuw form a *conflict triple* in an integer-weighted graph G_w if uv and uw are edges of G_w but vw is either a non-edge or a zero-edge. We distinguish two types of conflict triples vuw: if vw has weight zero then the conflict triple is called *weak*, whereas if vw is a non-edge then the conflict triple is called *strong*. In case the integer-weighted graph G_w contains no conflict triples, G_w is transitive. But the converse is obviously not true, as the example of a single weak conflict triple shows. A graph that does not contain any strong conflict triple is not necessarily transitive: For $V = \{u, v, w, x\}$ let uv, vw, wx be edges, let uw, vx be zero-edges, and let ux be a non-edge. The resulting graph is connected and contains no strong conflict triple, but is *not* a weak clique.

To solve Weighted Cluster Editing we first identify all connected components of the input graph and calculate the best solutions for all components separately, because an optimal solution never connects disconnected components. Furthermore, if the graph is decomposed during the course of the algorithm, then we recurse and treat each connected component individually. Our fixed-parameter algorithms often require a cost limit k: In case a solution with cost $\le k$ exists, the algorithm finds this solution; otherwise, "no solution" is returned. To find an optimal solution, we call the algorithm repeatedly, increasing k.

An unweighted Cluster Editing instance can be encoded by assigning weights $s(uv) \in \{+1, -1\}$. In the resulting graph, all conflict triples are strong. During data reduction and branching, we may set pairs uv to "forbidden" or "permanent", meaning that the status of uv cannot be changed in the future. In fact, permanent edges can be merged immediately: *Merging* uv means replacing the vertices u and v with a single vertex u', and, for all vertices $w \in V \setminus \{u, v\}$, replacing pairs uw, vw with a single pair $u'w$. See Section 3 for details. In this context, we say that we *join* vertex pairs uw and vw. The weight of the joined pair is $s(u'w) = s(uw) + s(vw)$. In case one of the pairs is an edge while the other is not, the parameter k is reduced by $\min\{|s(uw)|, |s(vw)|\}$. Note that we may join any combination of two edges, non-edges, or zero-edges when merging two vertices. We stress that joined pairs can be zero-edges.

For unweighted Cluster Editing, Guo [9] uses the concept of *critical cliques* to construct a kernel of size $4k_{\text{opt}}$. Critical cliques are cliques in the input graph that share the same neighborhood. In unweighted graphs, all vertices of a critical clique must end up in the same cluster, so we can always merge critical cliques. This idea does not apply directly to Weighted Cluster Editing but it is possible to adapt the concept as a data reduction by considering cliques with similar neighborhood [2]. However, this does not result in a kernel for Weighted Cluster Editing. When given an unweighted Cluster Editing instance, we merge all critical cliques and thus transform the graph into an integer-weighted graph with at most $4k_{\text{opt}}$ vertices. The graph where all critical cliques are merged can be easily constructed in $O(m + n)$ time [9] for an n-vertex and m-edge graph. The weight of any tuple uv is simply the product of the corresponding critical clique sizes $|C_u| \cdot |C_v|$.

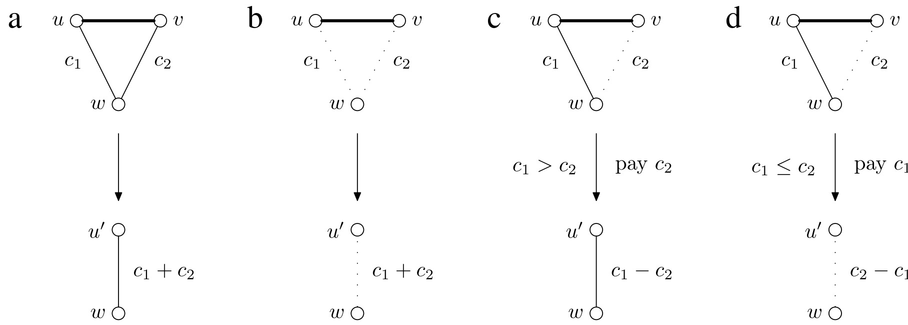

Fig. 1. Merging two vertices u, v into a new vertex u': Let $c_1 = |s(uw)|$, $c_2 = |s(vw)|$ be the edit costs. Dotted edges are nonexistent.

3. Data reduction

3.1. Problem kernel

In the beginning of our algorithm, and in each search node, we call the following data reduction routines in order to downsize the input graph as much as possible.

Rule 1: Remove cliques. Remove already existing cliques from the input graph.

Rule 2: Check for unaffordable edge modifications. For each set of two vertices u, v from V, we calculate a lower bound for the costs induced when uv is set to "permanent" or "forbidden", e.g. when the respective edge is modified. Let $N(v) := \{u \mid s(uv) > 0\}$ denote the set of neighbors of a vertex v, and let $A \triangle B$ be the symmetric set difference of sets A and B. We define induced costs $icf(uv)$ and $icp(uv)$ for setting uv to "forbidden" or "permanent", respectively:

$$icf(uv) = \max\{0, s(uv)\} + \sum_{w \in N(u) \cap N(v)} \min\{s(uw), s(vw)\} \tag{1}$$

$$icp(uv) = \max\{0, -s(uv)\} + \sum_{w \in N(u) \triangle N(v)} \min\{|s(uw)|, |s(vw)|\}.$$

This is how we make use of these values:

- For all $u, v \in V$ where $icf(uv) > k$: Insert uv if necessary, and set uv to "permanent" by assigning $s(uv) \leftarrow +\infty$.
- For all $u, v \in V$ where $icp(uv) > k$: Delete uv if necessary, and set uv to "forbidden" by assigning $s(uv) \leftarrow -\infty$.

If there is a pair uv such that both conditions hold simultaneously, the problem instance is not solvable.

Rule 3: Merge vertices incident to permanent edges. As soon as we set an edge uv to "permanent", we infer that u and v must be in the same clique in every solution. In this case, we *merge* u and v, creating a new vertex u'.

If w is a neighbor of both u and v, we create a new edge $u'w$ whose deletion costs as much as the deletion of both uw and vw. If w is neither a neighbor of u nor of v, we calculate the insertion cost of the nonexistent edge $u'w$ analogously. In case w is a neighbor of u or v but not both, uvw or vuw is a conflict triple, and we must decide whether we delete the edge connecting w with u or v, or we insert the nonexistent edge. By summing the weights (one of which is negative) to calculate $s(u'w)$ we carry out the cheaper operation, decreasing k accordingly, and maintain the possibility to edit $u'w$ later.

This is how we merge u and v into a new vertex u': For each vertex $w \in V \setminus \{u, v\}$ set $s(u'w) \leftarrow s(uw) + s(vw)$. Let $k \leftarrow k - icp(uv)$, and delete u and v from the graph. See Fig. 1. Note that these reduction rules conserve the optimal solution.

To start our data reduction, we have to compute $icf(uv)$ and $icp(uv)$ for all $u, v \in V$, which takes $O(n^3)$ time. Setting an edge to "forbidden" or "permanent" can reduce the parameter k because we might have to delete or insert an edge. If we merge a permanent edge, this can further reduce the parameter. This, in turn, may trigger other edges to become forbidden or permanent. In addition, setting an edge to "forbidden" or "permanent" will change the induced costs of other edges. This lemma shows how to execute our data reduction in running time $O(n^3)$ for integer-weighted input graphs and in time $O(n^3 \log n)$ and for a real-weighted input graph.

Lemma 1. *Data reduction according to Rules* 1–3 *can be carried out in running time $O(n^3)$ for integer-weighted graphs and $O(n^3 \log n)$ for real-valued edge weights.*

Proof. The induced costs $icf(uv)$ and $icp(uv)$ for each vertex pair $u, v \in V$ can be computed in $O(n)$ time. Therefore it initially takes $O(n^3)$ time to compute the induced costs of all $u, v \in V$. Note that during the data reduction, at most $\binom{n}{2}$ edges will be set to "forbidden", and at most $n - 1$ merge operations are executed before the graph collapses into a single vertex. If we deal with real-valued edges, we initialize the following data structures: For each $u \in V$ we use a binary heap to store all $icf(uw)$ and another binary heap to store all $icp(uw)$ for $w \in V$. This allows us to find $\max_w\{icf(uw)\}$ and $\max_w\{icp(uw)\}$

for each $u \in V$ in constant time. If edge weights are integers, we replace the binary heaps with double-linked lists, see [8] for details.

We repeatedly do the following: Using $\max_w\{icf(uw)\}$ and $\max_w\{icp(uw)\}$ for each $u \in V$, we find the overall maximum *icf* and *icp* value in time $O(n)$. We test if there exist $u, v \in V$ with $icf(uv) > k$ or $icp(uv) > k$. If no such u, v exist, we *stop*. Otherwise, we set the corresponding edge to "forbidden" or "permanent", we update parameter $k \leftarrow k - |s(uv)|$ and also the heaps for *icf* and *icp* values as described below. The running time of this part of the algorithm is $O(n^3 \log n)$.

Setting an edge uv to "forbidden" affects the values $icf(ux)$, $icf(vx)$, $icp(ux)$, and $icp(vx)$ for all vertices $x \in V$. We concentrate on updating $icf(ux)$, the other updates can be executed similarly. Let s_0 be our weight function before the update and s_1 after the update, then these functions agree except for $s_0(uv) \neq s_1(uv) = -\infty$. Analogously, let $icf_0(ux)$ and $icf_1(ux)$ denote "induced costs forbidden" of the tuple ux before and after the update, respectively. If $s_0(uv) \leq 0$ then no edge is deleted and we see from (1) that $icf_1(ux) = icf_0(ux)$ must hold. A similar argument resolves the case $s_0(xv) \leq 0$. If $s_0(uv) > 0$ and $s_0(xv) > 0$ then u,v as well as x,v were adjacent in the initial graph and $icf_1(ux) = icf_0(ux) - \min\{s_0(uv), s_0(xv)\}$ must hold. Clearly, computing $icf_1(ux)$ takes constant time. Updating all affected *icf* values and all binary heaps takes $O(n \log n)$ time: In case we have to decrease a key, we can remove the corresponding entry from the heap in time $O(\log n)$ and reinsert a new entry also in time $O(\log n)$. Because every edge can be set to "forbidden" at most once, and since there are $O(n^2)$ many edges, all updates induced from setting edges to "forbidden" take total time $O(n^3 \log n)$.

When we set an edge uv to "permanent", the data reduction merges u and v into a new vertex u' and deletes u, v from the graph. We iterate over all vertices $w \in V$ and first compute $s(u'w) \leftarrow s(uw) + s(vw)$ as well as $icf(u'w)$ and $icp(u'w)$ using (1). Analogous to the previous paragraph, this affects the values $icf(wx)$ and $icp(wx)$ for all vertices $x \in V$. For every vertex w, computing $icf(u'w)$, $icp(u'w)$ and updating all heaps takes time $O(n \log n)$, and so does updating the induced costs of all *icf* and *icp* values affected by $s(u'w)$. Hence, merging an edge can be executed in total time $O(n^2 \log n)$. There can be at most $n - 1$ merge operations, so the running time of all merge operations is also bounded by $O(n^3 \log n)$.

Finally, we can detect and remove all connected components that are cliques in time $O(n^2)$.

For integer-weighted graphs, using double-linked lists breaks the total running time for data reduction down to $O(n^3)$ [8]. □

The following lemma shows that our data reduction produces a problem kernel as the size of the resulting graph is polynomial in k.

Lemma 2. *Applying the above data reduction rules to an arbitrary instance of* Weighted Cluster Editing *with minimum modification cost of one, the resulting graph has at most $k^2 + 3k + 2$ vertices and $\frac{1}{2}k^3 + \frac{5}{2}k^2 + 5k + 2$ edges.*

Proof. Let $G = (V, E)$ be a weighted graph reduced with respect to our data reduction for some cost k. If G is not connected then we can treat each connected component individually, because inserting an edge between connected components is never optimal. In the following, we assume G to be connected and not to be transitive. To obtain a disjoint union of cliques G' from G we carry out $k = k_\mathrm{d} + k_\mathrm{i}$ edit operations, k_d deletions and k_i insertions. Since we assume minimum edit costs of one for each edge, we infer that $icp(uv)$ and $icf(uv)$ is always greater than, or equal to, the number of non-common neighbors or common neighbors of u and v, respectively. This does not hold after we have merged edges, because then edges of arbitrarily small weight can occur: But in this case, the parameter k has previously been decreased by at least the amount that is missing from the edge weight to one. We omit the peculiarities of this bookkeeping for the sake of brevity.

Let $C = (V_C, E_C)$ denote a largest clique in G'. Since we delete at most k_d edges, we infer that $|V_C| \geq |V|/(k_\mathrm{d} + 1)$. Now suppose u and v are two vertices in C: Clearly we can assume that u and v have at least $|V_C| - k_\mathrm{i} - 2$ common neighbors to end up in clique C after k_i edge insertions. Moreover, u and v have at most k common neighbors: otherwise they would have been reduced and merged by our data reduction. Thus, we infer $|V_C| - k_\mathrm{i} - 2 \leq k$. We conclude

$$\frac{|V|}{k_\mathrm{d} + 1} \leq |V_C| \leq k + k_\mathrm{i} + 2 = k_\mathrm{d} + 2k_\mathrm{i} + 2.$$

We can now define an upper bound for $|V|$:

$$|V| \leq k_\mathrm{d}^2 + 2k_\mathrm{i}k_\mathrm{d} + 2k_\mathrm{i} + 3k_\mathrm{d} + 2 \leq k^2 + 3k + 2.$$

The last inequality follows by substituting $k_\mathrm{i} = k - k_\mathrm{d}$ and differentiating for k_d, showing that there exist no extrema in the interval $k_\mathrm{d} \in [0, k]$. The same reasoning can be used to find an upper bound for the number of edges $|E|$. We omit the details and refer the reader to [4]. □

4. Branching strategies

We now describe four search tree algorithms for Weighted Cluster Editing with nonzero weights, starting with a simple $O(3^k)$ branching on conflict triples in Section 4.1. In Section 4.2 we adapt a branching strategy for unweighted Cluster Editing from [8] for the weighted problem, where we obtain a size $O(2.42^k)$ search tree. Afterwards, in Section 4.3, we present a new and simple branching idea which leads to a size $O(2^k)$ search tree, and in Section 4.4 we refine this idea and prove that the corresponding search tree has size $O(1.82^k)$. We stress that we demand integer weights for the running time

analyses in Sections 4.3 and 4.4. In contrast, the analyses in Sections 4.1 and 4.2 only require that the absolute value of each weight is at least one.

Recall that an undirected graph is transitive if and only if it does not contain a conflict triple. Each of our branching algorithms takes advantage of this observation: Search for a conflict triple and branch on a number of possibilities to destroy it. By this, we invoke a number of recursive calls, say l, on "simplified" instances of the problem where parameter k is decreased by some constants $a_1, a_2, \dots, a_l$. For branching vector $(a_1, a_2, \dots, a_l)$ we can compute a branching number using the characteristic polynomial, and this branching number in turn governs the asymptotic size of the search tree, see e.g. [11] for details.

4.1. Initial branching

Given a weighted graph $G = (V, E)$, we now describe a simple recursive algorithm that is guaranteed to find an optimal solution for Weighted Cluster Editing. Search for a conflict triple, and let u be the vertex of degree two and v, w be the leaves. For algorithmic reasons, we can set existent (nonexistent) edges to "permanent" ("forbidden") by assigning infinite edit costs to them. Recursively branch into three cases:

1. Insert vw, set uv, uw, and vw to "permanent".
2. Delete uv, set uw to "permanent" and uv and vw to "forbidden".
3. Delete uw, set uw to "forbidden".

In each branch, we lower k by the insertion or deletion cost required for the executed operation. If a connected component decomposes into two components, we calculate the optimum solutions for these components separately. If k falls below zero, we discard the respective branch of the algorithm. This strategy leads to the following theorem. Using interleaving [12], we reach:

Theorem 3. *If every edge of the arbitrarily weighted graph $G = (V, E)$ has weight of at least one, the* Weighted Cluster Editing *problem can be solved in $O(3^k + n^3 \log n)$ time, and in $O(3^k + n^3)$ time for integer weights.*

4.2. Refined branching strategy

In the following, we will refine the simple branching strategy resulting in a search tree of size $O(2.42^k)$, considering induced subgraphs of size 4. Unfortunately, the $O(2.27^k)$ branching strategy of Gramm *et al.* [8] cannot be used in the weighted case because it is based on an observation (Lemma 5) that does not hold for weighted graphs. We now modify this branching strategy accordingly.

Note that the automated search tree generator of Gramm *et al.* [7] also found an $O(2.42^k)$ search tree for induced subgraphs of size 4, but the branching strategy is not explicitly described there. If we consider induced subgraphs of size 5, this results in an $O(2.27^k)$ search tree [7]. The latter branching strategy requires case distinction with 20 initial cases and branching vectors of size at most 16. In comparison, our branching strategy distinguishes only four initial cases and branching vectors of length five.

Let vuw be a conflict triple as above. We distinguish the following cases:

(W1) Vertices v, w have no neighbors except for u, that is, $N(v) = \{u\}$ and $N(w) = \{u\}$.
(W2) Vertices v, w do not share a common neighbor, but there exists a vertex x such that, say, $vx \in E$. We distinguish two sub-cases: (W2a) $ux \in E$ (see Fig. 6), and (W2b) $ux \notin E$ (see Fig. 7).
(W3) Vertices v, w share a common neighbor $x \neq u$, so $vx \in E$ and $wx \in E$. We distinguish two sub-cases: (W3a) $ux \in E$ (see Fig. 8), and (W3b) $ux \notin E$ (see Fig. 9).

In case (W1) holds, we *ignore* the conflict triple vuw for the moment, and continue with the next triple. In all other cases, we branch as indicated by Figs. 6, 7, 8, 9 in Appendix A.

We describe the branching in detail for case (W2a), see Fig. 6: Here, edges uv, uw, ux, and vx are present in the induced graph. We branch into five sub-cases:

- Delete uw and set uw to "forbidden".
- Set uw to "permanent", delete uv, ux, and set uv, ux, vw, wx to "forbidden".
- Insert wx, set uw, ux, wx to "permanent", delete uv, vx, and set uv, vw, vx to "forbidden".
- Insert vw, set uv, uw, vw to "permanent", delete ux, vx, and set ux, vx, wx to "forbidden".
- Insert vw, wx and set all six edges to "permanent".

The branching strategies for case (W2b), (W3a), and (W3b) can be easily derived from Figs. 7, 8, 9.

One can easily check that if only conflict triples of type (W1) are present in a connected graph, this graph is a star graph, that is, a tree where all vertices but one are leaves. It is straightforward to quickly find an optimal solution for this case. We omit the details. Again using interleaving [12], the analysis of the refined branching strategy leads to Theorem 4.

Theorem 4. *If every edge of the weighted graph $G = (V, E)$ has a weight of at least one, then the total running time using our refined branching strategy is $O(2.42^k + n^3 \log n)$.*

4.3. Edge branching

We now describe a much simpler recursive algorithm in which we branch into two sub-cases to repair the conflict at a chosen conflict triple. Please recall that, in this section, and in Section 4.4, we only refer to integer-weighted graphs.

The **edge branching strategy** is as follows: Let uv be an edge of a (weak or strong) conflict triple vuw. Then, (a) set uv to forbidden, or (b) merge uv.

Let us first analyze this very simple strategy. One can easily check that this recursive procedure will at some point generate an optimal solution, because in every step we resolve a conflict triple. In the following, we will analyze the size of the search tree. When deleting an edge uv we decrease the parameter by $s(uv)$. When merging vertices u and v, for each vertex $w \in V \setminus \{u, v\}$ we join the pairs uw and vw into a single pair with weight $s(uw) + s(vw)$. If $s(uw) \neq -s(vw)$ then parameter k can be lowered by $\min\{s(uw), -s(vw)\}$. In case $s(uw) = -s(vw)$, the new pair is a zero-edge, and this would prevent us from decreasing our parameter when joining the zero-edge in a later stage of the algorithm. To circumvent this problem, we assume that joining uw and vw with $s(uw) = -s(vw)$ only reduces the parameter by $\min\{s(uw), -s(vw)\} - \frac{1}{2} = |s(uw)| - \frac{1}{2} \geq \frac{1}{2}$. If at a later stage we join this zero-edge with another pair, we decrease our parameter by the remaining $\frac{1}{2}$. Using this bookkeeping trick, our edge branching strategy has a branching vector of $(1, \frac{1}{2})$ that leads to a search tree of size $O(2.62^k)$.

We can easily improve this branching strategy by choosing a "good" edge uv, as follows: Choose the particular edge $uv \in E$ that *minimizes* the branching number of the corresponding branching step. The branching number is computed from branching vector (a, b) where a is the cost of deleting edge uv, while b is the cost of merging this edge. If one of these costs is zero, we say that the edge has an infinite branching number. Using the bookkeeping trick introduced above, an edge uv with a finite branching number is not necessarily part of *any* conflict triple: joining a zero-edge uw with a vertex pair vw generates cost $\frac{1}{2}$ irrespective of whether vw is an edge, non-edge, or zero-edge. So, even the edge with a minimum branching number might not be part of any conflict triple.

The following is a simple observation regarding unweighted graphs. We will make use of it in the search tree analysis, see Lemma 6.

Lemma 5. *Given a connected, unweighted graph G, if every edge of G is part of at most one conflict triple, then G is either a clique or a clique minus a single edge.*

Proof. If $G = (V, E)$ contains no conflict triple then G is a clique. Assume that there is at least one conflict triple vuw in G with $uv, uw \in E$ and $vw \notin E$. We constructively show that G is a clique minus the edge vw. If another vertex $x \in V \setminus \{u, v, w\}$ is adjacent to v then $ux \in E$ must hold, too: otherwise, uv is part of two conflict triples vuw and uvx contrary to our assumptions. Similarly, $ux \in E$ implies $vx \in E$. In conclusion, $ux \in E$ if and only if $vx \in E$. The same holds replacing v by w, and we infer that if some vertex x is adjacent to one of u, v, or w then it is adjacent to all of u, v, and w.

Next, consider two vertices x, y adjacent to all u, v, w. If $xy \notin E$ then vxw and xvy are two conflict triples containing the edge xv that conflicts with our assumptions, so $xy \in E$ must hold. Finally, consider vertices x, z where x is adjacent to u, v, w while z is not adjacent to u, v, w, and assume $xz \in E$. Now the edge vx is part of the two conflict triples vxw and vxz, again a contradiction to our assumptions. So, any vertex $x \in V \setminus \{v, w\}$ must be adjacent to all other vertices in G. □

Lemma 6. *For an integer-weighted graph, the edge branching strategy that chooses an edge with minimum branching number has branching vector at least* $(1, 1)$.

Proof. Recall that if we create a zero-edge, this reduces k by at least $\frac{1}{2}$; and if we join a zero-edge, this reduces k by $\frac{1}{2}$. Let uv be the edge with minimum branching number. Note that removing uv induces cost $s(uv) \geq 1$, and let δ be the cost of merging uv. If $\delta \geq 1$ then we are done, so assume $\delta < 1$. This implies that at most one zero-edge was created or joined. In particular, uv is part of at most one conflict triple vuw, and there cannot be an edge that is part of two conflict triples. We transform the input graph into an unweighted graph G, where zero-edges and non-edges in the input graph are not present in G. By Lemma 5 above, the connected component containing vuw must be a clique minus vw in G. Regarding the weighted graph, all vertex pairs are edges except vw, which may be a non-edge or a zero-edge. If vw is a zero-edge then our branching will stop when merging uv, so assume that vw is a non-edge. We now show that, for this case, we can omit our bookkeeping trick of delayed parameter decrease.

We now either delete uv with cost $s(uv) \geq 1$, or merge uv. We distinguish the cases $s(uw) \geq -s(vw)$ and $s(uw) < -s(vw)$. If $s(uw) \geq -s(vw)$ holds then the joined pair has weight $s(uw) + s(vw) \geq 0$, the resulting connected component is a clique that can be removed from the graph, and we reduce the parameter k by $\min\{s(uw), -s(vw)\} \geq 1$. For $s(uw) < -s(vw)$ the joined pair has weight $s(uw) + s(vw) < 0$, so we have not generated a zero-edge. We can assume in our analysis that parameter k is reduced by the full $\min\{s(uw), -s(vw)\} \geq 1$. So, the branching vector is at least $(1, 1)$ as claimed. □

Hence, edge branching results in a search tree of size $O(2^k)$ for integer-weighted graphs.

4.4. Refined edge branching

We now refine our edge branching and show that the respective search tree has size $O(1.82^k)$. This results in the fastest known algorithm for unweighted Cluster Editing: the previously best-known branching strategy by Gramm et al. [7] results in a search tree of size $O(1.92^k)$. This algorithm uses complicated branching rules (more than 1300 cases) and has never

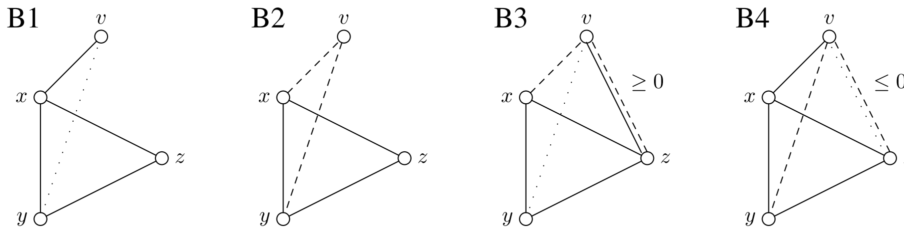

Fig. 2. Conditions (B1) to (B4) of edge sorting. Solid lines are edges, dashed lines are zero-edges, dotted lines are non-edges.

been implemented. To the best of our knowledge, the fastest implementation for unweighted Cluster Editing has running time $O(2.27^k + n^3)$ using 11 branching cases [5,8]. In contrast, our branching strategy is both fast and simple, using only two branching cases.

Theorem 7. *For an integer-weighted graph that contains no zero-edges, the* Weighted Cluster Editing *problem can be solved in* $O(1.82^k + n^3)$ *time.*

We modify the order in which edges are processed by the edge branching strategy, which allows for a simpler analysis of the running time behavior. We conjecture that Theorem 7 is also true for edge branching where edges are sorted with respect to branching number, but this requires further case distinctions.

Let G_w be an integer-weighted and connected graph. We say that we *branch on* an edge uv by setting uv to forbidden and recursing, and merging uv and recursing. To deal with zero-edges, we use the above bookkeeping trick: Creating a zero-edge induces cost $\geq \frac{1}{2}$, and resolving a zero-edge induces the remaining cost $\frac{1}{2}$. We choose an edge to branch on according to the following order:

(A) If there is an edge with branching vector $(1, \frac{3}{2})$ or better, then we branch on this edge.
(B) If there is an edge xy and a vertex z in G_w such that x,y,z form a triangle, and if there exist two additional vertices v_1,v_2 such that for both v_1,v_2 one of the following conditions holds (where x and y may be exchanged):
  (B1) xv_i is an edge and yv_i is a non-edge
  (B2) xv_i is a zero-edge and yv_i is a zero-edge
  (B3) xv_i is a zero-edge and yv_i is a non-edge, and zv_i is an edge or a zero-edge
  (B4) xv_i is an edge and yv_i is a zero-edge, and zv_i is a non-edge or a zero-edge.
  Then branch on xy.

If no such edge exists, we stop the recursion. We will show below that the remaining graph must be a clique, a clique minus one edge (where the last edge is either a zero-edge or a non-edge), a path, a circle, or contains only 4 vertices. We will also show how to solve this remaining instance in polynomial time. See Fig. 2 for the four initial cases of condition (B). To be more precise, there are ten different subcases of condition (B) that are combinations of (B1), . . ., (B4), taking into account that we can exchange x and y. We denote them by (B11) to (B44). See Fig. 3 for an exemplar branching.

If there exists an edge satisfying condition (A) then branching on this edge has branching number 1.76. The following lemma corresponds to condition (B) of edge sorting, and shows how we analyze two branching steps together: The first branching step can in fact result in a branching vector of (1, 1) but the next branching steps result in better branching vectors, leading to an overall branching number as desired.

Lemma 8. *Let G_w be an integer-weighted and connected graph, and assume that there is an edge xy that satisfies condition* (B). *Then, branching on xy and performing another branching step where edges to branch on are chosen according to the edge sorting, results in a branching vector of* $(2, \frac{5}{2}, 2, 3)$ *with branching number* ≤ 1.82.

Proof. Branching on edge xy leads to a branching vector of (1, 1): Deleting xy induces cost at least 1, and merging xy results in cost at least $2 \cdot \frac{1}{2} = 1$ since for each v_i a conflict triple or a zero-edge will be resolved. We will now show that after setting xy to "forbidden" there exists an edge with branching vector $(1, \frac{3}{2})$ and after merging xy there exists an edge with branching vector (1, 2). These are the worst-case branching vectors for the edge that is chosen in the next branching step.

First we analyze the case where xy is set to "forbidden", see Fig. 4: We show that now one of the edges xz or yz has branching vector $(1, \frac{3}{2})$. Setting xz or yz to forbidden results in cost 1. Merging xz or yz resolves the conflict triple xzy, resulting in cost 1 since xy is forbidden. If condition (B2), (B3), or (B4) holds then in addition, a zero-edge is resolved when merging xz or yz. If condition (B1) holds we distinguish two cases: If v_1z is a non-edge or a zero-edge, then we branch on xz, which either resolves an additional zero-edge, or resolves the conflict triple v_1xz. If v_1z is an edge then we branch on yz, which resolves the conflict triple v_1zy. Hence, either xz or yz have merging costs $\frac{3}{2}$.

Second we consider the case where x, y have been merged, see Fig. 5. Let w_{xy} be the vertex resulting from merging xy: We show that now, the edge $w_{xy}z$ has branching vector (1, 2). Deleting $w_{xy}z$ induces cost of 2 as $s(w_{xy}z) \geq 2$. Merging $w_{xy}z$ induces cost of $\frac{1}{2}$ for each v_i: If condition (B1) holds for v_i then $w_{xy}v_i$ is a zero-edge. Otherwise, we infer $s(xv_i) > 1$ or $s(yv_i) < -1$, so the initial branching on xy would have resulted in a branching vector of $(1, \frac{3}{2})$. Merging $w_{xy}z$ resolves this zero-edge. If condition (B2) holds then $w_{xy}v_i$ clearly is a zero-edge. For conditions (B3) and (B4) merging $w_{xy}z$ either resolves

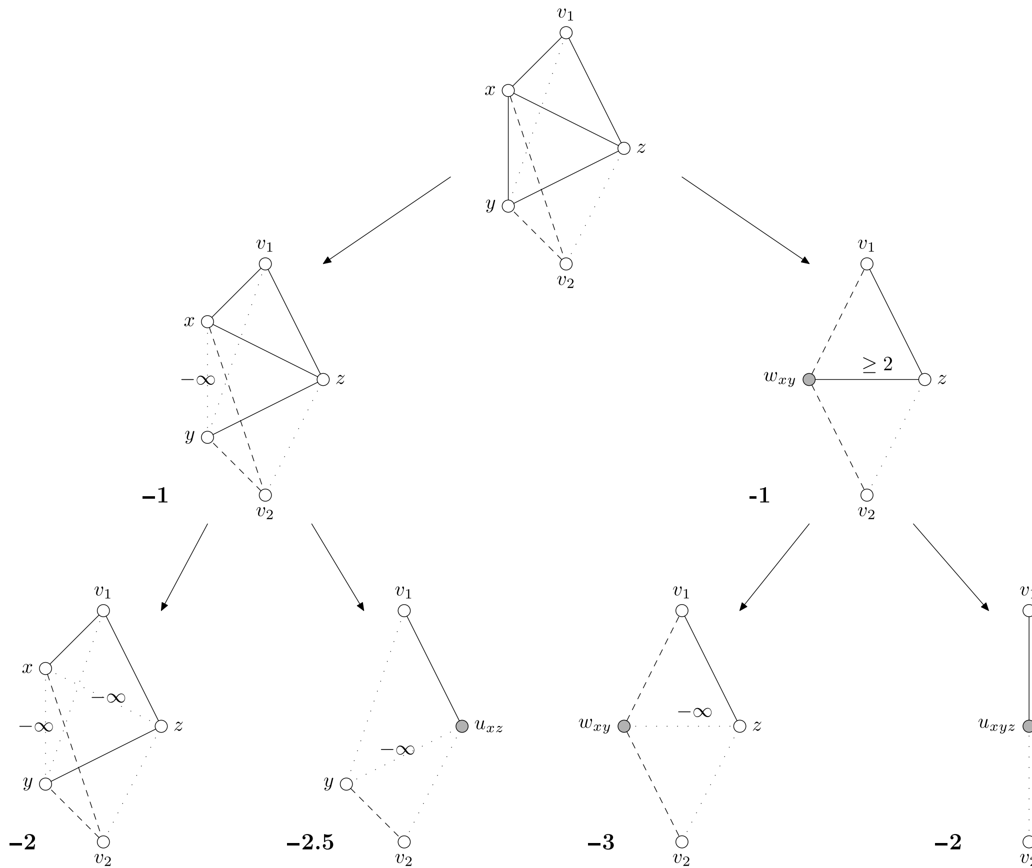

Fig. 3. An example for branching with branching vector $(2, \frac{5}{2}, 2, 3)$ under condition (B12): (B1) holds for vertex v_1 and (B2) holds for vertex v_2.

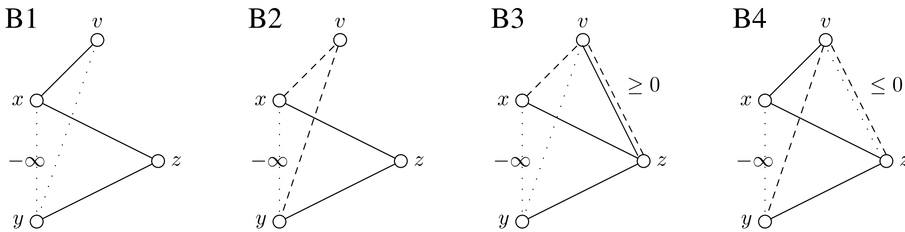

Fig. 4. Branching conditions (B1) to (B4) after *xy* is set to "forbidden".

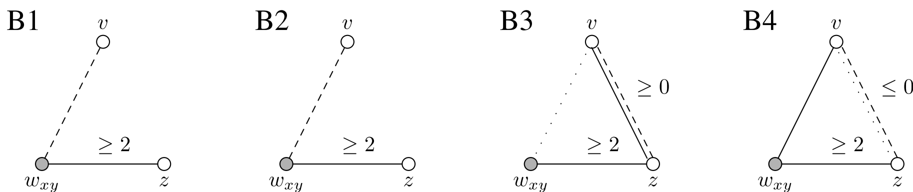

Fig. 5. Conditions (B1) to (B4) after merging *xy*.

a zero-edge v_iz or a conflict triple $w_{xy}zv$ or $vw_{xy}z$. These observations hold both for v_1 and v_2, so merging $w_{xy}z$ results in total cost $\frac{2}{2}$.

We cannot guarantee that the edge branching strategy will actually branch on edges *xz* or *yz* (after *xy* has been set to forbidden) and $w_{xy}z$ (after merging *xy*) in the next step of the branching. But we have shown that edges with branching numbers 1.755 and 1.6191 exist after the first step of the branching. With regards to the first case, one can easily check that all possible branching vectors with branching number ≤ 1.755 are of the form $(a, b/2)$ for integers $a \geq 1$ and $b \geq 3$.

Similarly, all branching vectors with branching number ≤ 1.6191 are of the form $(a, b/2)$ for integers $a \geq 1$ and $b \geq 4$, or $a \geq 2$ and $b \geq 2$. This shows that even if we pick other edges in the second step of our branching, we still can guarantee branching vector $(2, \frac{5}{2}, 2, 3)$ with branching number 1.82. □

The following is again an observation regarding unweighted graphs:

Lemma 9. *Let G be a connected, unweighted graph. Assume that there is no edge in G that is part of three conflict triples, and there exists no triangle uvw in G such that uv is part of two conflict triples. Then G is a clique, a graph with at most one non-edge, a $K_{1,3}$ (a star graph with four vertices), a path, or a circle.*

Proof. Assume that $G = (V, E)$ contains at least one edge xy that is part of two conflict triples: Otherwise, Lemma 5 guarantees that G is a clique or a clique minus a single edge. Let u and v be two vertices involved in conflict triples with xy. This implies that either xu or yu is an edge, and that either xv or yv is an edge. Assume that there exists another vertex $z \notin \{x, y, u, v\}$ with $xz \in E$: If $yz \in E$ then xyz is a triangle as excluded by our assumptions, and if $yz \notin E$ then xy is part of three conflict triples. So, no such z can exist and neither x nor y can be connected to any other vertex.

We distinguish two cases of intersecting conflict triples: the two conflict triples are either of the form uxy and uxv (asymmetric case), or uxy and xyv (symmetric case). Note that we can exchange u and v in the asymmetric case.

Let us consider the asymmetric case first. Assume there exists another vertex $w \notin \{x, y, u, v\}$ with $uw \in E$. Then the edge xu is part of two conflict triples yxu and xuw. If $uv \notin E$ then xu is part of three conflict triples. If $uv \in E$ then the edge xu is part of a triangle xuv that is excluded by our assumptions. This implies that no such vertex w can exist, and the connected graph G is a $K_{1,3}$.

For the symmetric case, assume that there exists another vertex $w \notin \{x, y, u, v\}$ with $uw \in E$. Now, the edge xu is part of two conflict triples yxu and xuw, again in symmetric arrangement. If some $z \notin \{x, y, u, v, w\}$ exists with $uz \in E$, we can show again that xu is part of three conflict triples or part of a triangle excluded by our assumptions. The same holds true for a vertex w with $vw \in E$. Repeating this argument we show that all vertices in the connected graph G have degree one or two, so G is a path or a circle. □

Let us now assume that there is no edge that satisfies branching conditions (A) or (B). Again, we transform the integer-weighted graph into an unweighted graph G where zero-edges of the integer-weighted graph are transformed into non-edges in G. Clearly, G does not contain an edge that is part of three conflict triples. Using Lemma 9 we infer that G is either one of the graph structures described there, or there exists an edge xy that is part of a triangle xyz and that is part of two conflict triples. In the first case, we have reduced the weighted graph as claimed: The weighted graph is a clique, a clique minus one edge, a path, a circle, or contains only four vertices. In the second case, there is an edge xy that is contained in a triangle and two conflict triples for which branching condition (B) does not apply. It can be shown by rather technical analysis that, in all cases, the weighted graph is a weak clique or a graph with exactly one non-edge. See Appendix B for details.

If the remaining graph is a (weak) clique, we are finished. If it is a graph with one non-edge uv, we can solve it in polynomial time by calculating a minimum u-v-cut. In case the cost of the cut is higher than $-s(uv)$, we insert uv and are finished, otherwise we cut the graph according to the minimum u-v-cut and obtain two (weak) cliques. If the graph has at most four vertices, we can easily try all possibilities of solving it. The only remaining case is that the graph is a path or a circle. Our next step is to prove that such a graph can be solved in polynomial time.

Assume that our weighted graph is a path with n vertices. We will now show how to compute an optimal solution in time $O(n^2)$.

Let $v_1, v_2, \ldots, v_n$ be the vertices of the path, so $s(v_i v_{i+1}) > 0$ and $s(v_i v_j) \leq 0$ for $j > i+1$. First, we may assume that any clique in an optimal solution will contain all vertices in an interval $v_i, v_{i+1}, \ldots, v_{j-1}, v_j$: If there is a clique C in the optimal solution such that $v_i, v_j \in C$ but $v_{i+1}, \ldots, v_{j-1} \notin C$ where $j \geq i + 2$, then cut C into two parts by removing all edges $v_{i'} v_{j'}$ for $i' \leq i$ and $j' \geq j$ from the solution.

We now compute an array K where $K[i, j]$ is the cost of inserting all edges $v_{i'} v_{j'}$ for $i \leq i', i' + 2 \leq j'$, and $j' \leq j$. So, $K[i, j]$ is the cost for completing $v_i, \ldots, v_j$ to a clique. We can compute $K[i, j]$ in time $O(n^2)$ using the recurrence $K[i, j] = K[i, j-1] + K[i+1, j] - K[i+1, j-1] + (-s(v_i, v_j))$ with initialization $K[i, i] = 0$ for all i.

Next, let $D[j]$ be the cost of an optimal solution for the subgraph induced by vertices $v_1, \ldots, v_j$. To compute an optimal solution for the path that also contains vertex v_{j+1} we have to distinguish two cases: either $v_j v_{j+1}$ is not an edge of this optimal solution, then we delete edge $v_j v_{j+1}$ and use the optimal solution on vertices $v_1, \ldots, v_j$. Or, v_{j+1} is part of a clique in the optimal solution with vertices $v_{i+1}, \ldots, v_{j+1}$, then we delete edge $v_i v_{i+1}$ for $i \geq 0$, and add the cost of an optimal solution on vertices $v_1, \ldots, v_i$ to the cost of completing $v_{i+1}, \ldots, v_{j+1}$ to a clique. Set $S(j) := s(v_j, v_{j+1})$ and $S(0) = 0$. We reach the recurrence

$$D[j+1] = \min_{i=0,\ldots,j} \{D[i] + S(i) + K[i+1, j+1]\}$$

with initialization $D[0] = 0$. Clearly, D can be computed in $O(n^2)$ time, and the optimal solution can be constructed using backtracing.

Regarding circles, we just note that either all vertices of the circle are part of the same clique in the optimal solution, or one of the edges of the circle has to be deleted. By iteratively deleting one edge from the circle and solving the recurrence for paths, we can find an optimal solution in $O(n^3)$ time.

Table 1
Average running times for artificial data with $O(2.42^k)$ branching strategy without merging, $O(2.42^k)$ branching strategy with merging, $O(3^k)$ branching strategy, edge branching strategy and edge branching strategy with reduction. Numbers in lines labeled "# unfinished" are numbers of instances not finished after one week of computation. There were ten instances per bucket for sizes 10–50, five instances for sizes 60–100. For the computation of average running times, we ignored unfinished instances.

Size of instance	10	20	30	40	50	60	70	80	90	100
Average # edit	8.3	28.1	66.7	115.5	183.2	263.0	351.6	459.0	594.0	728.6
2.42^k (no merging)	11 ms	69 ms	8.3 s	31.4 min	47.8 h	n/a	n/a	n/a	n/a	n/a
# unfinished	–	–	–	–	1	5	5	5	5	5
2.42^k strategy	12 ms	56 ms	1.9 s	52.2 s	28 min	17.5 h	26.3 h	n/a	n/a	9.4 h
# unfinished	–	–	–	–	–	–	3	5	5	4
3^k strategy	10 ms	54 ms	1.0 s	29 s	7.6 min	26.6 h	57.7 h	19 days	n/a	n/a
# unfinished	–	–	–	–	–	–	1	4	5	5
edge branching	4 ms	16 ms	238 ms	2.5 s	18.2 s	5.5 h	17.7 h	13.8 h	34.8 h	6.6 days
# unfinished	–	–	–	–	–	–	–	–	2	4
with reduction [2]	3 ms	14 ms	169 ms	1.9 s	1.6 s	31.6 s	43.2 s	23.3 s	165.2 s	36.8 s
# unfinished	–	–	–	–	–	–	–	–	–	–

Now we have the means to prove Theorem 7.

Proof (*Theorem* 7). From the above, we infer that our search tree has size $O(1.82^k)$. This results in a total running time of $O(1.82^k \cdot k^8 + n^3)$: Initially, we run our parameter-dependent data reduction in time $O(n^3)$, see Lemma 1. This data reduction results in a problem kernel with $O(k^2)$ vertices. For every edge, we compute the branching number that results from deleting and merging this edge in total time $O(k^6)$. Similarly, we can check for the substructures for branching condition (B) in time $O(k^8)$. We get rid of the polynomial factor by interleaving [12], that is, performing data reduction repeatedly during the course of the search tree algorithm whenever possible. This reduces the total running time to $O(1.82^k + n^3)$. Alternatively, we can deduce this running time from the fact that 1.82 is rounded up, so that the polynomial factor is already covered by the O-notation. The remaining structures can be solved in polynomial time. □

Regarding Weighted Cluster Editing instances with real-valued weights, the edge branching strategy is also guaranteed to find the optimal solution. Let k be the cost parameter. We want to decide whether there is a solution of cost at most k. To estimate the worst-case running time, we have to assume again that all vertex pairs have weight at least one [1]. We redo our simple analysis from Section 4.3: Whenever joining two pairs of vertices results in a pair with absolute weight smaller than one, we put aside $\frac{1}{2}$ using our bookkeeping technique. This pair may later be part of a conflict triple, and when editing this pair we decrease k by $\frac{1}{2}$ we put aside earlier because the absolute weight of this pair can be arbitrarily small. A similar analysis to that given in this section shows that the worst-case branching vector reduces to $(\frac{1}{2}, 2, 2)$ and the size of the search tree is $O(2.39^k)$.

5. Computational results

We have implemented the edge branching algorithm with search for the edge with maximum branching number in C++. We apply our data reduction from Section 3 to every instance in advance and when traversing the search tree. The program accepts nonnegative real values as edge modification costs. All running times were measured on an AMD Opteron-275 2.2 GHz with 6 GB of memory running Solaris 10.

We want to explore and compare the performance of our branching and compare it to the previously fastest branching strategy for Weighted Cluster Editing.

For our evaluation, we use both artificial and biological data. We generate artificial instances by first constructing a transitive graph with n vertices by uniformly drawing clique sizes in $\{1, \ldots, n\}$ until all vertices have been used up. Next, we perturb this graph: for each pair uv we delete or insert an edge uv with probability 0.15. Running times are reported in Table 1. Our biological instances stem from protein similarity data, generated using more than 192 000 protein sequences from the COG dataset [16]. Edge weights are computed from log E-values of bidirectional BLAST hits using a threshold of 10^{-10}: The modification cost of each vertex pair is the difference between the logarithms of the threshold and the respective protein's E-values. In the resulting graph, 3964 connected components are not transitive, and 3788 of these have up to 100 vertices. See Rahmann *et al.* [13] for more details. Running times for these data sets are shown in Table 2. Recently, Wittkop *et al.*[18] showed that the Cluster Editing model leads to valid clusterings when applied to protein similarity data and manages to outperform other methods.

As one can see, merging vertices clearly leads to drastic improvements, and unlike what theoretical running times suggest, the $O(2.42^k)$ is mostly outperformed by the $O(3^k)$ strategy when we merge edges. Edge branching is much faster than the other branching algorithms, and performance is increased by several orders of magnitude. For comparison, we also report running times of the FPT algorithm from [2] that uses the same edge branching strategy but, in addition, employs new parameter-independent reduction rules to cut down instance sizes before branching, and further heuristic improvements.

Table 2
Average running times for protein similarity data using the $O(2.42^k)$ branching strategy without merging, $O(2.42^k)$ branching strategy, $O(3^k)$ branching strategy, edge branching strategy and edge branching strategy with reduction. Again, "# unfinished" gives the number of instances not finished after seven days of computation. Unfinished instances were ignored for computing average running times.

Size of instance	3–10	11–20	21–30	31–40	41–50	51–60	61–70	71–80	81–90	91–100
No. of instances	2080	726	308	178	181	117	93	57	28	26
Average # edit	2.32	11.78	30.92	62.02	101.6	137.4	176.4	227.8	430.9	315.0
2.42^k (no merging)	3 ms	26 ms	640 ms	2.0 min	10.3 min	1.1 h	1.7 h	1.4 h	44.8 min	40.8 min
# unfinished	–	–	–	–	2	5	4	9	7	7
2.42^k strategy	2 ms	14 ms	184 ms	2.2 s	7.1 min	20.4 min	4.4 min	4.8 h	4.6 h	7.1 min
# unfinished	–	–	–	–	–	1	1	2	2	3
3^k strategy	2 ms	12 ms	103 ms	680 ms	91 s	24 min	46.8 min	6.8 h	3.7 h	49 s
# unfinished	–	–	–	–	–	–	–	1	1	2
edge branching	0.8 ms	5 ms	22 ms	203 ms	1.3 s	47.7 s	2.3 s	43.4 s	46.2 min	31.8 min
# unfinished	–	–	–	–	–	–	–	1	–	1
with reduction [2]	11 ms	7 ms	31 ms	91 ms	218 ms	517 ms	581 ms	1.2 s	2.0 s	3.2 s
# unfinished	–	–	–	–	–	–	–	–	–	–

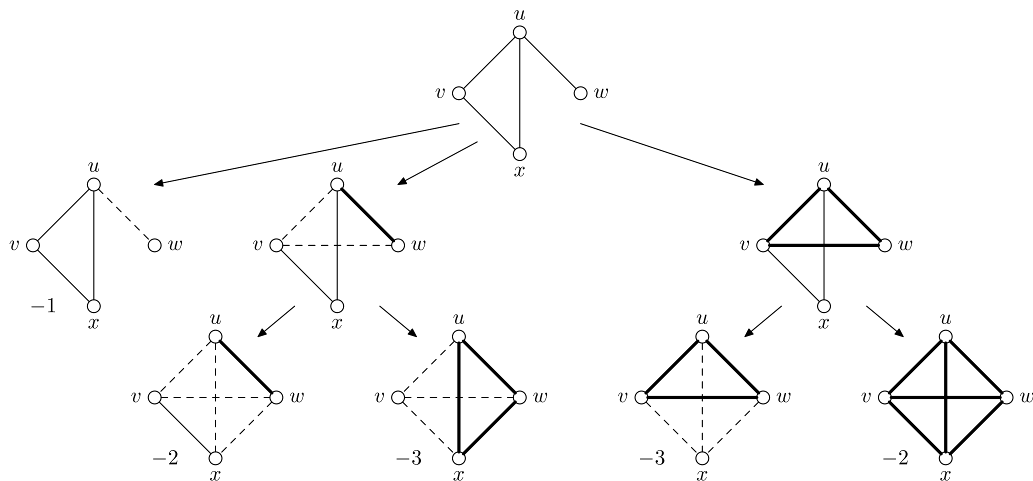

Fig. 6. Case (W2a): Vertices v, w do not share a common neighbor; v has a neighbor x connected with u.

6. Conclusion

We have presented several branching strategies for (weighted) Cluster Editing, including a surprisingly simple branching strategy that leads to the fastest known parameterized algorithm for (integer-weighted) Cluster Editing with respect to theoretical running time bounds. We believe that we can prove even better worst-case running times for this same strategy, using a refined, automated analysis similar to [7].

We implemented different branching strategies and evaluated their performance. Together with further improvements reported in [2], our best algorithm allows one to solve weighted Cluster Editing instances with several hundred edge modifications in a matter of seconds. This clearly proves the practical usefulness of our approach and constitutes a huge improvement over [5] where unweighted instances with 50 edge modifications required several hours of computation. Wittkop et al. [18] recently demonstrated the power of Weighted Cluster Editing for clustering homologous proteins, so algorithms both fast in theory and efficient in practice are highly desirable.

Acknowledgments

We thank Svenja Simon and Thilo Muth for helping with the implementation. S. Briesemeister gratefully acknowledges financial support from LGFG Promotionsverbund "Pflanzliche Sensorhistidinkinasen" at the University of Tübingen. Q.B.A. Bui is funded by the DFG project PABI (Parameterized Algorithms in Bioinformatics) BO 1910/5.

Appendix A. Illustration of the refined branching strategy

See Figs. 6–9.

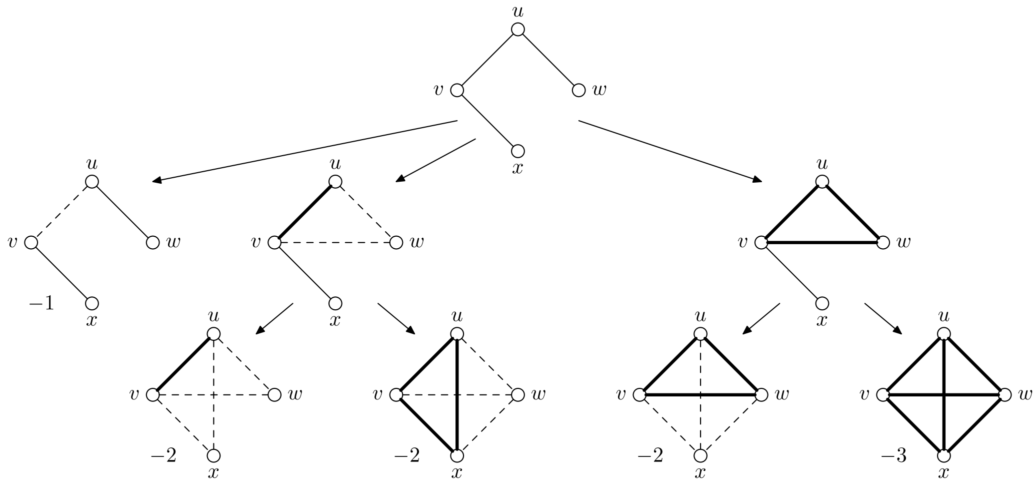

Fig. 7. Case (W2b): Vertices v, w do not share a common neighbor; v has a neighbor x not connected with u.

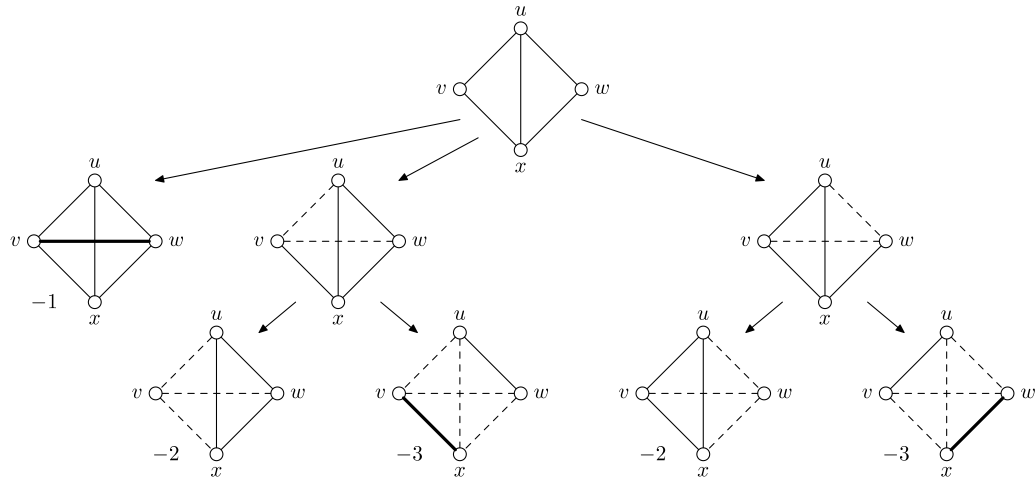

Fig. 8. Case (W3a): Vertices v, w share a common neighbor x, and $ux \in E$; see Fig. 2 in [8].

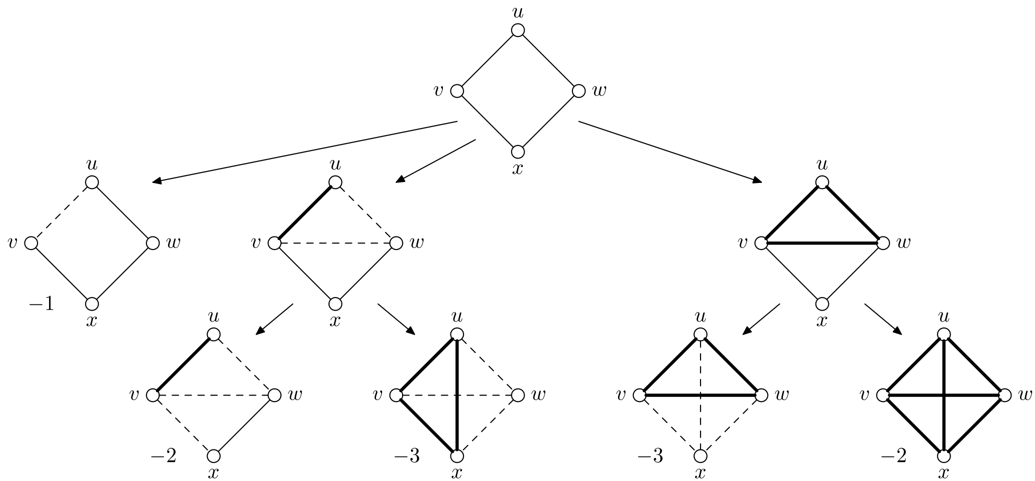

Fig. 9. Case (W3b): Vertices v, w share a common neighbor x, and $ux \notin E$; see Fig. 3 in [8].

Appendix B. Graphs that do not meet refined edge branching conditions

In this part of the Appendix, we have a closer look at edges that are part of a triangle and two conflict triples but do not fulfill conditions (B11)–(B44). See Fig. 10 for an illustration.

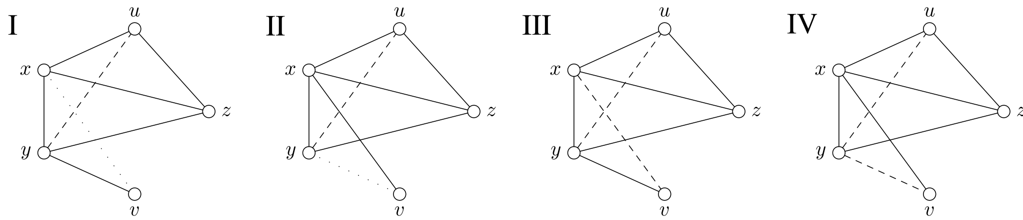

Fig. 10. Illustration of the four cases that are not covered by the branching conditions of the refined edge branching strategy. Dashed edges represent zero-edges, dotted edges are non-edges.

We show that a graph that contains such an edge but does not fulfill branching conditions (A) or (B) is already a weak clique (zero-edges allowed) or a graph with at most one non-edge. For our analysis, we assume that the graph is connected.

Case (I). We can see that vz must be an edge, otherwise we could branch on yv with branching vector $(\frac{3}{2}, 1)$: If we merge yv we lower k by $\frac{1}{2}$ for each of the two conflict triples xyv and vyz and by another $\frac{1}{2}$ for joining yu, and if we delete yv we lower k by 1. The pair uv must be an edge as well. If it were a zero-edge, we could apply branching condition (B12) to yv, if it was a non-edge, we could apply (B13) to yv. The induced subgraph of the five vertices we have considered has exactly one non-edge. It remains to prove that there cannot be further non-edges in the graph.

Let us consider another vertex z'. Envision that there cannot be further zero-edges incident to x or y since this would lead to branching with vector $(\frac{3}{2}, 1)$ at xy. z' is connected to x if and only if z' is connected to y because otherwise xy would be part of three conflict triples. Analogously, $z'x \in E$ iff $z'u \in E$, and $z'y \in E$ iff $z'v \in E$, else, we could branch on xu or yv with branching vector $(\frac{3}{2}, 1)$. So a vertex is connected with x, y, u, or v if and only if it is connected with all these vertices.

Now assume there is a new vertex z' connected with z. Then z' must also be connected with v: If vz' was a non-edge, we could apply branching condition (B11) to vz, if vz' was a zero-edge, we could branch on yv with branching vector $(\frac{3}{2}, 1)$. Hence, a neighbor of z (or a neighbor of any other vertex connected with x, y, u, and v) must also be connected with x, y, u, and v.

Now let us consider two vertices z_1, z_2 that are both connected with x, y, u, and v. If z_1z_2 was a non-edge, we could branch on xz_1 with condition (B11), so z_1z_2 must be an edge or a zero-edge.

All in all, xv is the only one non-edge in the graph induced by x, y, u, and v. All other vertices are connected to these four vertices. Among each other, they are linked by edges or zero-edges, so G_w contains exactly one non-edge.

Case (II). In this case, we know that at least uv or zv must be an edge, else xv would be contained in three conflict triples vxy, vxz, and vxu.

Neither of uv and zv may be a non-edge because otherwise we could apply branching condition (B11) to xv.

Let us suppose there is a further vertex z' that is connected to x or y. We show that there are no non-edges between z' and the vertices x, y, u, and v. Again, $xz' \in E$ if and only if $xy' \in E$, so z' is connected to x and y. Now uz' is an edge, else we could branch on xy under condition (B14). If vz' was a non-edge, we could branch on xv under condition (B11), so vz' is either a zero-edge or a non-edge.

Now assume that a new vertex z' is connected to u. If xz' is a non-edge, Case (I) applies at xu, so we do not have consider this case further. If xz' was a zero-edge, we could perform a $(\frac{3}{2}, 1)$ branching on xy. Hence, z' is connected to x with all implications shown in the last paragraph.

If we introduce a vertex z' connected to v, we see that $xz' \in E$: If xz' was a non-edge, we could branch on xv with branching condition (B11). If xz' was a zero-edge, we had a $(\frac{3}{2}, 1)$ branching on xy. As shown above, z' is also connected to y and u, and $z'v$ may be an edge or a zero-edge.

Finally, let us consider the case of a vertex z' connected to z. Then yz' is an edge, otherwise we could branch on yz with branching vector $(\frac{3}{2}, 1)$.

We have shown that there are no non-edges between x, y, u, v, z and their neighbors. Introducing a neighbor of one of these neighbors is equivalent to introducing a neighbor of z.

It remains to prove that there are no non-edges between vertices in $V \setminus \{x, y, u, v\}$. Consider two arbitrary vertices $z_1, z_2 \in V \setminus \{x, y, u, v\}$. We know that they are connected to x, y, and u, and that each of vz_1, vz_2 is either an edge or a zero-edge. If z_1z_2 was a non-edge or a zero-edge, we could branch on yz with branching vector $(\frac{3}{2}, 1)$.

We conclude that the only non-edge in G_w is yv.

Case (III). First we observe that vz must be an edge, or else we could branch on yv with branching vector $(\frac{3}{2}, 1)$. If uv was a non-edge, uz would meet the criteria of Case (II), so we can assume that uv is an edge or a zero-edge.

If we introduce another vertex z' that is connected with one of the vertices u, v, x, or y, we observe that z' must be connected with the other three vertices as well: Each of the edges xy, xu, yv already shares vertices with two zero-edges, so they can neither be in a further conflict triple nor incident to further zero-edges. Otherwise, we could perform a $(\frac{3}{2}, 1)$ branching on one of these edges. So if z' is connected to one vertex of xy, xu, or yv, it must also be connected to the other vertex of the edge.

If z' is connected with z (or any other neighbor of u, v, x, and y), it is also connected with u, v, x, and y. Otherwise, zz' would be part of the (weak or strong) conflict triples xzz', yzz', uzz', and vzz'. This is not possible since no edge is part of more than two conflict triples if branching condition (A) does not apply.

It remains to check how neighbors of u, v, x, and y are connected to each other: If an edge z_1z_2 between two such vertices is a non-edge, we can apply Case (I) to uz. Otherwise G_v is a weak clique.

Case (IV). Here we notice that at least one of the pairs uv, zv must be an edge, otherwise xv would be part of three conflict triples. With this observation, vz must be an edge or one of the former cases applies: If vz were a non-edge, Case (I) would apply at uz, if it was a zero-edge, Case (III) would apply at uz. Similarly, uv must be an edge or a zero-edge, otherwise Case (II) would apply at uz. We will now show that the graph is a weak clique.

As a first step, we analyze how other vertices can be connected to x, y, z, u, and v. Again, as xy is already part of two conflict triples, every vertex $z' \in V \setminus \{x, y, z, u, v\}$ connected with x is also connected with y and vice versa, and there cannot be zero-edges incident to x or y, else we could branch on xy with branching vector $(\frac{3}{2}, 1)$. Since yz is also contained in two conflict triples, we analogously observe that a vertex z' is connected with y if and only if it is connected with z. Then xz', yz', and zz' are either all edges or all non-edges.

Now we have to distinguish the subcases (i) where uv is a zero-edge and (ii) the case where uv is an edge. First, let us assume that (i) uv is a zero-edge. In this case, xu and xv are also edges that are part of two conflict triples each, so $xz' \in E$ iff $uz' \in E$ and $xz' \in E$ iff $yz' \in E$, and there may not be zero-edges incident to u or v. We infer that z' is connected with x, y, z, u, or v if and only if it is connected with all these vertices.

Now (ii) let uv be an edge. Assume there is a vertex $z' \in V \setminus \{x, y, z, u, v\}$ that is connected with x, y, and z. Then uz' must be an edge or a zero-edge, otherwise branching condition (B14) would apply to uz. Analogously, vz' must be an edge or a zero-edge, or branching condition (B14) would apply to vz.

Now assume there is a vertex z' connected with u. If xz' was a non-edge, branching condition (B14) would apply to xu, if it was a zero-edge we could branch on xy with factor $(\frac{3}{2}, 1)$. So z' is connected with x and thus also with y and z, and vz' is at least a zero-edge.

Now we assume there is a vertex $z' \in V \setminus \{x, y, z, u, v\}$ that is connected with v. With the same reasoning as in the last paragraph except for u and v being swapped, we infer that z' is connected with x, y, and z, and uz' is at least a zero-edge.

We infer that every z' connected with x, y, z, u, or v is also connected with x, y, and z, and uz' and vz' are edges or zero-edges. Again, the same holds for neighbors of any such z'.

We still have to prove for both (i) and (ii) that two vertices z_1, z_2 that are connected with x, y, z, u, and v (but uz_1, vz_1, uz_2, and vz_2 may be zero-edges in subcase (ii)), are connected with each other.

If z_1z_2 was a zero-edge or a non-edge we had the opportunity for a $(\frac{3}{2}, 1)$ branching on yz_1, so z_1 and z_2 must be connected. With this observation we have shown that G_w is a weak clique.

References

[1] S. Böcker, S. Briesemeister, Q.B.A. Bui, A. Truss, A fixed-parameter approach for Weighted Cluster Editing, in: Proc. of Asia–Pacific Bioinformatics Conference, APBC 2008, in: Series on Advances in Bioinformatics and Computational Biology, vol. 5, Imperial College Press, 2008.

[2] S. Böcker, S. Briesemeister, G.W. Klau, Exact Algorithms for Cluster Editing: Evaluation and Experiments, in: Proc. of Workshop on Experimental Algorithms, WEA 2008, in: Lect. Notes Comput. Sc., vol. 5038, Springer, 2008, pp. 289–302.

[3] H.L. Bodlaender, L. Cai, J. Chen, M.R. Fellows, J.A. Telle, D. Marx, Open problems in parameterized and exact computation — IWPEC 2006, Technical Report UU-CS-2006-052, Department of Information and Computing Sciences, Utrecht University, 2006.

[4] S. Briesemeister, Weighted Cluster Editing for clustering biological data, Diplomarbeit, Friedrich-Schiller-Universität Jena, 2007.

[5] F. Dehne, M.A. Langston, X. Luo, S. Pitre, P. Shaw, Y. Zhang, The cluster editing problem: Implementations and experiments, in: Proc. of International Workshop on Parameterized and Exact Computation, IWPEC 2006, in: Lect. Notes Comput. Sc., vol. 4169, Springer, 2006, pp. 13–24.

[6] M. Grötschel, Y. Wakabayashi, A cutting plane algorithm for a clustering problem, Math. Program. 45 (1989) 52–96.

[7] J. Gramm, J. Guo, F. Hüffner, R. Niedermeier, Automated generation of search tree algorithms for hard graph modification problems, Algorithmica 39 (4) (2004) 321–347.

[8] J. Gramm, J. Guo, F. Hüffner, R. Niedermeier, Graph-modeled data clustering: Fixed-parameter algorithms for clique generation, Theor. Comput. Syst. 38 (4) (2005) 373–392.

[9] J. Guo, A more effective linear kernelization for Cluster Editing, Theor. Comput. Sci. 410 (8–10) (2009) 718–726.

[10] M. Křivánek, J. Morávek, NP-hard problems in hierarchical-tree clustering, Acta Inform. 23 (3) (1986) 311–323.

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

[12] R. Niedermeier, P. Rossmanith, A general method to speed up fixed-parameter-tractable algorithms, Inform. Process. Lett. 73 (2000) 125–129.

[13] S. Rahmann, T. Wittkop, J. Baumbach, M. Martin, A. Truss, S. Böcker, Exact and heuristic algorithms for weighted cluster editing, in: Proc. of Computational Systems Bioinformatics, CSB 2007, vol. 6, 2007, pp. 391–401.

[14] R. Shamir, R. Sharan, D. Tsur, Cluster graph modification problems, Discrete Appl. Math. 144 (1–2) (2004) 173–182.

[15] R. Sharan, A. Maron-Katz, R. Shamir, CLICK and EXPANDER: A system for clustering and visualizing gene expression data, Bioinformatics 19 (14) (2003) 1787–1799.

[16] 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, D.A. Natale, The COG database: An updated version includes eukaryotes, BMC Bioinformatics 4 (2003) 41.

[17] A. van Zuylen, 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, in: Lect. Notes Comput. Sc., vol. 4927, Springer, 2008, pp. 260–273.

[18] T. Wittkop, J. Baumbach, F. Lobo, S. Rahmann, Large scale clustering of protein sequences with FORCE — A layout based heuristic for weighted cluster editing, BMC Bioinformatics 8 (1) (2007) 396.