Applying a Cut-Based Data Reduction Rule for Weighted Cluster Editing in Polynomial Time

Hjalmar Schulz
Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

André Nichterlein
Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

Rolf Niedermeier
Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

Christopher Weyand
Karlsruhe Institute of Technology, Germany

Abstract

Given an undirected graph, the task in Cluster Editing is to insert and delete a minimum number of edges to obtain a cluster graph, that is, a disjoint union of cliques. In the weighted variant each vertex pair comes with a weight and the edge modifications have to be of minimum overall weight. In this work, we provide the first polynomial-time algorithm to apply the following data reduction rule of Böcker et al. [Algorithmica, 2011] for Weighted Cluster Editing: For a graph $G = (V, E)$, merge a vertex set $S \subseteq V$ into a single vertex if the minimum cut of $G[S]$ is at least the combined cost of inserting all missing edges within $G[S]$ plus the cost of cutting all edges from S to the rest of the graph. Complementing our theoretical findings, we experimentally demonstrate the effectiveness of the data reduction rule, shrinking real-world test instances from the PACE Challenge 2021 by around 24 % while previous heuristic implementations of the data reduction rule only achieve 8 %.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases Correlation Clustering, Minimum Cut, Maximum s-t-Flow

Digital Object Identifier 10.4230/LIPIcs.IPEC.2022.25

Supplementary Material *Software (Source Code)*: `https://github.com/venondev/AlmostCliquePoly`, archived at `swh:1:dir:8aca200ba4c16f1b357d20904271c630e5c4fa5a`

Acknowledgements In memory of Rolf Niedermeier, our colleague, friend, and mentor, who sadly passed away before this paper was published.

1 Introduction

The NP-hard Cluster Editing problem [4, 26], also known as Correlation Clustering [3], is one of the most popular graphs clustering approaches in algorithmics. Given an undirected graph, the task is to transform it into a disjoint union of cliques (also known as a cluster graph) by applying a minimum number of edge modifications (deletions or insertions). In the weighted variant Weighted Cluster Editing each pair of vertices comes with a weight and to goal is a find edge modifications of minimum summed weight to create a cluster graph. (Weighted) Cluster Editing has applications in fields such as bioinformatics [4], data mining [3], and psychology [27]. It gained high popularity in studies concerning parameterized algorithmics [1, 2, 9, 11, 14, 15, 22, 5, 7, 6] and algorithm engineering [15, 8, 17, 5]. The unweighted Cluster Editing was the problem selected for the PACE implementation challenge 2021 [21]. An important aspect of solving (Weighted) Cluster Editing, both in theory and practice, is kernelization. From a theoretical side,

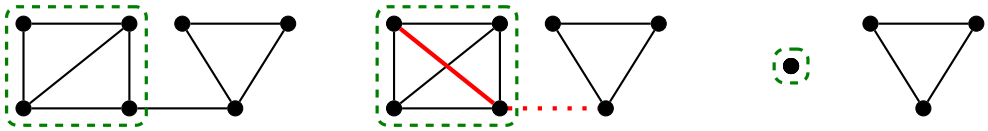

Figure 1 *Left:* Input graph with two "obvious" clusters, one being highlighted in green. *Middle:* Optimal Cluster Editing-solution for the input graph: the thick red edge in the green cluster is inserted and the red dotted edge between the clusters is deleted. *Right:* The graph obtained by merging the vertices in the green vertex set in the input (weights are not shown for visibility). If all weights are 1 in the input graph, then the vertex subset highlighted in green satisfies the condition of the data reduction rule: the minimum cut has weight 2 (cutting two edges), which is at least the cost of 2 to make the green vertex set an isolated clique (see middle for the two modified edges).

both the weighted and unweighted version of Cluster Editing admit polynomial-size problem kernels. Studies in this direction were initialized by Gramm et al. [15] for Cluster Editing, who provided a kernel with $\mathcal{O}(k^2)$ vertices and sparked follow up work [13, 16]. The smallest known kernels have $2k$ vertices [9, 10]. The $2k$-vertex kernel of Cao and Chen [9] also holds for Weighted Cluster Editing.

As to the practical side, state-of-the-art solvers for Cluster Editing and Weighted Cluster Editing rely on polynomial-time computable data reduction rules and the preprocessing routines are heavily optimized over time [8, 17, 5]. In fact, the winning solver of the PACE challenge 2021 solves about half of the instances by data reduction alone [5]. We contribute to this line of work by providing a divide & conquer-based, polynomial-time algorithm to apply a data reduction rule by Böcker et al. [8, Rule 4] for Weighted Cluster Editing. This data reduction rule works intuitively as follows: Let $S \subseteq V$ be a vertex subset. If the cost of splitting S into at least two parts is at least as high as the cost of cutting S from the rest of the graph and making S a clique, then merge all vertices in S; see Figure 1 for an illustration and Section 2 for the exact formulation of the data reduction rule. Given a vertex subset S, it is easy to check in polynomial time whether the data reduction rule is applicable on S (call such vertex subsets *applicable*). However, there was no efficient algorithm to find applicable vertex subsets; thus only heuristics were applied [8]. We provide experiments demonstrating that these heuristics miss many applicable vertex subsets in real world data sets: The heuristics merge on average only 8.1 % of the vertices in the input. However, the exhaustive application of the data reduction rule with our polynomial-time algorithm reveals that on average 24.2 % of the vertices in the input could be merged.

Our polynomial-time algorithm runs in $\mathcal{O}(n \cdot (T_{\text{mincut}}(n,m) + T_{s\text{-}t\text{-maxflow}}(n,m) + n^2))) \subseteq \mathcal{O}(nm^{1+o(1)} + n^3)$ time, where $T_{\text{mincut}}(n,m)$ and $T_{s\text{-}t\text{-maxflow}}(n,m)$ denote the time to compute in an edge-weighted graph with n vertices and m edges a minimum cut and a maximum s-t-flow, respectively.

2 Preliminaries

We set $\mathbb{N} := \{0, 1, 2, \ldots\}$ and set $\binom{S}{2}$ to be the set of all two-element subsets of a set S. Let Δ denote the symmetric difference. All graphs considered in this work are simple and undirected. Moreover, we assume that the input graph is always connected as connected components can be solved independently. A graph is a *cluster graph* if each connected component is a clique. For a weighted graph $G = (V, E, \omega)$, the weight function $\omega\colon \binom{V}{2} \to \mathbb{Z}$ implicitly defines the edges $E := \{uv \mid \omega(uv) > 0\}$. That is, a positive value of the weight function indicates an edge and a negative value (or zero) a non-edge. The cost of modifying

an edge uv is then the absolute value $|\omega(uv)|$ of its weight. For a vertex set $S \subseteq V$, we denote with $G[S]$ the graph induced by S. The decision variant of Weighted Cluster Editing is defined as follows:

Weighted Cluster Editing
Input: An undirected edge-weighted graph $G = (V, E, \omega)$ and $k \in \mathbb{N}$.
Question: Is there a set $P \subseteq \binom{V}{2}$ with $\sum_{uv \in P} |\omega(uv)| \leq k$, such that $G' = (V, E \Delta P)$ is a cluster graph?

Cuts and the Picard-Queyranne DAG. Let $G = (V, E, \omega)$ be a weighted graph. Let $U, W \subset V$ with $U \cap W = \emptyset$. We denote with $E(U, W)$ the edges between U and W; with $V(U, W)$ the vertices incident to $E(U, W)$; and with $\text{cost}(U, W)$ the summed cost of removing the edges $E(U, W)$. A cut c_V of G is a partitioning of the vertex set V into two non-empty partitions $U \subset V$ and $V \setminus U$, each being called a *side* of the cut. The minimum cut (*mincut*) of G is the cut of G with minimum cost; we denote its cost with $\text{mincut}(G) := \min_{U \subset V}\{\text{cost}(U, V \setminus U)\}$. Note that, the cuts consider just the edges of the graph and ignore the non-edges.

Let $s, t \in V$. All minimum s-t-cuts can be represented by a structure called a Picard-Queyranne DAG (PQ-DAG for short) [25]. It is constructed by considering the reachability relation in the residual network of any maximum s-t-flow and contracting strongly connected components. *Contracting* two vertices u and v means to consider them as one, new vertex x with neighborhood $N(u) \cup N(v)$. More precisely, for $w \in N(u) \cup N(v) \setminus \{u, v\}$, the weight of the edge to x becomes $\omega(xw) = \max(0, \omega(uw)) + \max(0, \omega(vw))$. Therefore, each node of the PQ-DAG represents a set of vertices of the original graph. For better distinction we say the PQ-DAG has *nodes* which represent subsets of *vertices* of the input graph. If the graph is undirected, then the DAG has only one sink (the strongly connected component containing s) and only one source (the strongly connected component containing t). A *closure* of a DAG is a set of nodes without outgoing arcs. Each closure of the PQ-DAG represents a minimum s-t-cut [25], that is, one side of a minimum s-t-cut. Thus, any postfix and any prefix in any topological ordering of the PQ-DAG represents a minimum s-t-cut. Moreover, for any minimum s-t-cut, there exists some topological ordering of the PQ-DAG with a prefix representing this cut. In this case, we say that the ordering *respects* the cut.

Merging Vertices & Data Reduction Rule. We will contract vertices when working with cuts. For Weighted Cluster Editing we need to *merge* vertices (these two notions coincide if each non-edge has weight 0). Merging two vertices u and v means to consider them as one, new vertex x. For each $w \in V \setminus \{u, v\}$, the weight of the (non-)edge to x becomes $\omega(xw) = \omega(uw) + \omega(vw)$. If w was adjacent to exactly one of u and v, then the solution size k is reduced by $\min(|\omega(uw)|, |\omega(vw)|)$. Intuitively, if a vertex w is (non-)adjacent to both u and v, then w is also (non-)adjacent to the new vertex. However, if w is adjacent to exactly one of u and v, then the new vertex is adjacent to w iff, of the pairs uw and vw, the pair representing an edge has higher weight than the one representing a non-edge. Merging a and b in the graph on the left side of Figure 2 results in the graph on the right side.

For a vertex set $S \subseteq V$, we call the summed cost of all non-edges in $G[S]$ the *deficiency* of S and define $\text{def}_G(S) := \sum_{u,v \in S} |\min(0, \omega(uv))|$. We can now formally state the condition triggering the data reduction rule.

▶ **Definition 2.1.** *A vertex subset $S \subseteq V$ with $|S| \geq 2$ is* applicable *if*

$$\text{mincut}(G[S]) \geq \text{def}_G(S) + \text{cost}(S, V \setminus S). \tag{1}$$

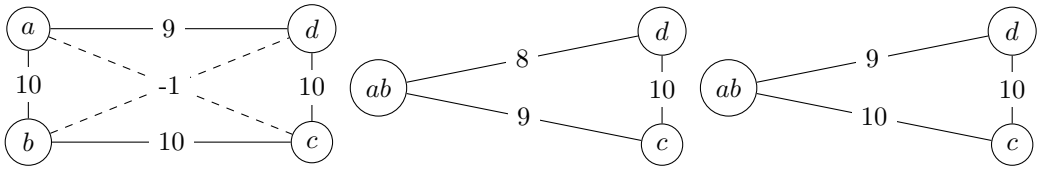

Figure 2 *Left:* A graph where the only applicable vertex set is $S = \{a, b, c, d\}$. *Middle:* Result of merging a and b. *Right:* Result of contracting a and b.

Algorithm 1 Applying Reduction Rule 2.2.

Input: A connected weighted graph $G = (V, E, \omega)$ and a vertex subset $A \subseteq V$.
Output: A largest applicable set $S \subseteq A$ if it exists, $\emptyset$ otherwise.

```
Function FindMergeSet(G, A)
    if |A| < 2 then return ∅
    c_A = (A_1, A_2) ← arbitrary mincut in G[A]
    S ⊆ A ← largest applicable set that is cut by c_A
    // here max returns the largest set
    return max(S, FindMergeSet(G, A_1), FindMergeSet(G, A_2))
```

▶ **Reduction Rule 2.2** (Almost clique; Böcker et al. [8, Rule 4])**.** Let $S \subseteq V$ be an applicable vertex set. Then merge the vertices within S and reduce the solution size accordingly.

3 A Polynomial-Time Algorithm to Apply Reduction Rule 2.2

In this section, we present a polynomial-time algorithm for applying Reduction Rule 2.2. More precisely, for a graph $G = (V, E, \omega)$ and set $A \subseteq V$, our algorithm finds the largest applicable set $S \subseteq A$ if such a set exists. Our algorithm follows a simple divide & conquer approach (see Algorithm 1) and starts with $A = V$. First, compute a mincut in the graph. Then, finding an applicable vertex set that is within a side of the cut is simply a recursive call. The interesting part is the conquer-step to find the largest applicable vertex set that has vertices on both sides of the cut. Such a set is either A itself or a proper subset of A. To find the latter in polynomial time, we need some structural insights presented in the following lemma. It provides some restrictive conditions on such sets (see left side of Figure 3 for an illustration of the setting).

▶ **Lemma 3.1.** *Let $G = (V, E, \omega)$ be a graph, $A \subseteq V$, and let $c_A = (A_1, A_2)$ be a mincut of $G[A]$. If there is an applicable set $S \subset A$ that is cut by c_A, that is, $S \cap A_1 \neq \emptyset$ and $S \cap A_2 \neq \emptyset$, then*

(a) $\mathrm{def}_G(S) = 0$ *and*
(b) $\mathrm{mincut}(G[S]) = \mathrm{mincut}(G[A]) = \mathrm{cost}(S, A \setminus S) = \mathrm{cost}(S, V \setminus S)$.

Proof. By assumption, c_A is a mincut in $G[A]$ that also cuts S, thus $\mathrm{mincut}(G[S]) \leq \mathrm{mincut}(G[A])$. Since S is a proper subset of A we have $\mathrm{mincut}(G[A]) \leq \mathrm{cost}(S, A \setminus S) \leq \mathrm{cost}(S, V \setminus S)$. Since $\mathrm{def}_G(S) \geq 0$ and S is applicable, we conclude that

$$\begin{aligned}\mathrm{mincut}(G[S]) \leq \mathrm{mincut}(G[A]) \leq \mathrm{cost}(S, A \setminus S) \leq \mathrm{cost}(S, V \setminus S)\\ \leq \mathrm{cost}(S, V \setminus S) + \mathrm{def}_G(S) \overset{(1)}{\leq} \mathrm{mincut}(G[S]).\end{aligned}$$

Hence all inequalities are equalities and $\mathrm{def}_G(S) = 0$. ◀

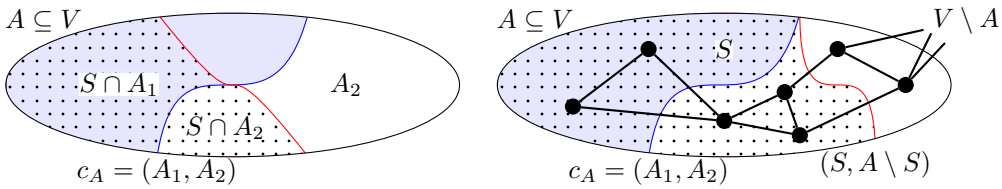

Figure 3 *Left:* Illustration of the setting of Lemmas 3.1 and 3.2 with a mincut (A_1, A_2) for $G[A]$ (black ellipse) and S (dotted area) overlapping with both sides of the cut (one side with blue background). *Right:* An example where a vertex set S (dotted area) with $A_1 \subseteq S$ and $S \cap A_2 \neq \emptyset$ exists. For simplicity, all edges have weight 1 and all non-edges have weight 0. The set A_1 (highlighted by blue background) contains two connected vertices. The mincut of $G[S]$ is 2, the same as both the cost of $c_A = (A_1, A_2)$ and of $(S, V \setminus S) = (S, A \setminus S)$ (two edges cut).

Lemma 3.1 already provides enough information to perform the conquer step in polynomial time: The set S induces another mincut $(S, A \setminus S)$ of $G[A]$. As all mincuts in a graph can be computed in polynomial time [24, 20], we can simply iterate over those and check if another mincut of $G[A]$ has an applicable set S as one side. We can, however, improve this and avoid computing all mincuts. To this end, we use the following observations.

▶ **Lemma 3.2.** *Let $G = (V, E, \omega)$ be a graph, $A \subseteq V$, and let $c_A = (A_1, A_2)$ be a mincut of $G[A]$. If there is an applicable set $S \subset A$ that is cut by c_A, that is, $S \cap A_1 \neq \emptyset$ and $S \cap A_2 \neq \emptyset$, then*

(a) *$(S, A \setminus S)$ is a mincut of $G[A]$,*
(b) *the endpoints of edges crossing c_A are part of S, i. e. $V(A_1, A_2) \subseteq S$,*
(c) *S includes either A_1 or A_2,*
(d) *S is not connected to vertices outside A, i. e. $\text{cost}(S, V \setminus A) = 0$, and*
(e) *$\text{def}_G(A_i \cup V(A_1, A_2)) = 0 = \text{cost}(A_i \cup V(A_1, A_2), V \setminus A)$ for $i = 1$ or $i = 2$.*

Proof.

(a) This follows from $\text{mincut}(G[A]) = \text{cost}(S, A \setminus S)$ from Lemma 3.1.
(b) If a vertex $v \in V(A_1, A_2)$ were not in S, then $c_S = (A_1 \cap S, A_2 \cap S)$ would be cut of $G[S]$ with $E(c_S) \subset E(c_A)$ and $\text{cost}(c_S) < \text{cost}(c_A) = \text{mincut}(G[A])$, contradicting Lemma 3.1.
(c) If S includes neither A_1 nor A_2, then we have $\text{cost}(S, A \setminus S) = \text{cost}(S, A_1 \setminus S) + \text{cost}(S, A_2 \setminus S) > \text{cost}(S, A_1 \setminus S) = \text{cost}(S \cup A_2, A \setminus (S \cup A_2))$ contradicting that $(S, A \setminus S)$ is a mincut of $G[A]$.
(d) This follows from $\text{cost}(S, A \setminus S) = \text{cost}(S, V \setminus S)$.
(e) This directly follows from (b), (c), (d), and $\text{def}_G(S) = 0$ (by Lemma 3.1). ◀

In practice, Lemma 3.2 (e) almost always rules out the existence of a set S going over the cut (A_1, A_2). In theory, however, such a set S fulfilling all the restrictions of Lemmas 3.1 and 3.2 can exist, see Figure 3 (right) for an example.

We already know that isolating S is a mincut of $G[A]$. In the following we identify vertices that must be on opposite sides of this cut. This reduces the mincut problem to a more managable s-t-cut problem. Those s-t-cuts can be represented by a PQ-DAG.

Since we assume the graph to be connected, only one side of c_A can be isolated from $V \setminus A$. Let A_1 be this side, hence $A_1 \subset S$ by Lemma 3.2 (c). By Lemma 3.2 (b, d), the sought-after set S contains also all vertices incident to edges crossing c_A and cannot contain vertices of A that are connected to the rest of the graph. Thus, if we contract all vertices in $A_1 \cup V(A_1, A_2)$ into one source s and all vertices in A with neighbors in $V \setminus A$ into one sink t, then S must be a minimum s-t-cut. See Figure 4 for an overview. Unfortunately, in the case that $V = A$,

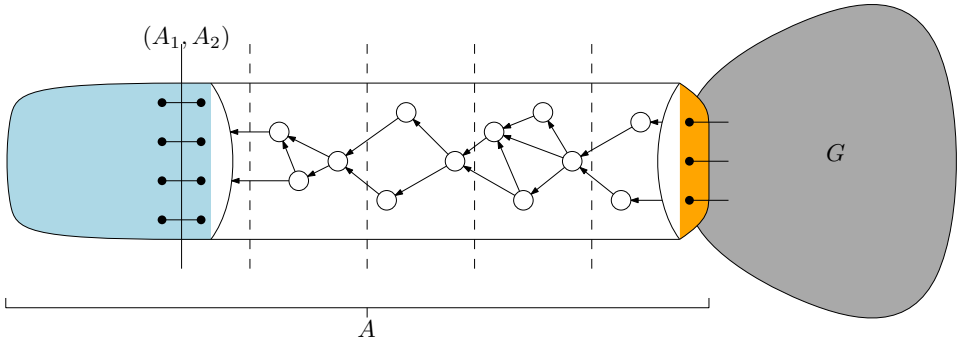

Figure 4 Layout of S, A, G and the PQ-DAG. Vertices that must be in S are colored blue. Vertices that must not be in S are colored orange. Those two colored sets are contained in the sink/source component, respectively, of the PQ DAG representing all minimum s-t-cuts between them.

there is no $V \setminus A$ and we need another approach. Recall that $\text{def}_G(S) = 0$. Thus, if both sides of c_A have deficit, there is no applicable S. If both sides have no deficit, then the instance is trivial because the solution that transforms V to a clique has cost zero. Only if one side has deficit and the other has not, there may exist an applicable set. In this case the side without deficit must be included in S and at least one endpoint of an edge with negative weight cannot be in S. Therefore we can solve the $A = V$ case with two computations similar to the $A \subset V$ case by trying both endpoinds of an arbitrary deficit edge as the sink.

▶ **Lemma 3.3.** *Let $G = (V, E, \omega)$ be a weighted graph, $s, t \in V$, and $c_S = (S, V \setminus S)$ a minimum s-t-cut with $s \in S$, $t \in V \setminus S$ and $\text{mincut}(G[S]) = \text{cost}(c_S)$. Then, there exists a cut node $Q \subseteq S$ of the PQ-DAG that separates $S \setminus Q$ from $V \setminus S$.*

Proof. Let $Q_0, Q_1, \ldots, Q_p$ be a reverse topological ordering of the PQ-DAG of the residual graph of G that respects c_S. Let $Q_{\leq i} \coloneqq \bigcup_{j \leq i} Q_j$ and $Q_{>i} \coloneqq \bigcup_{j>i} Q_j$. The minimum s-t-cuts respected by this ordering have the form $c_i = (Q_{\leq i}, Q_{>i})$ (by definition they all have the same cost). Thus, $s \in Q_0$, $t \in Q_p$, $c_S = c_\ell$, and $S = Q_{\leq \ell}$, for some $\ell < p$.

$$\begin{aligned}\text{cost}(c_S) &= \text{cost}(c_\ell) = \text{cost}(c_{\ell-1}) = \text{cost}(Q_{\leq \ell-1}, Q_{>\ell-1}) \\ &= \text{cost}(Q_{\leq \ell-1}, Q_\ell) + \text{cost}(Q_{\leq \ell-1}, Q_{>\ell}) \geq \text{cost}(Q_{\leq \ell-1}, Q_\ell) \geq \text{mincut}(G[S]).\end{aligned}$$

Since $\text{mincut}(G[S]) = \text{cost}(c_S)$, the inequalities become equalities and $\text{cost}(Q_{\leq \ell-1}, Q_{>\ell})=0$. Therefore, Q_ℓ is the desired cut node. ◀

The ordering of cut-nodes in a DAG with one sink and one source is the same in all possible topological orderings. Hence, Lemma 3.3 gives potential candidate sets $S_1 \subseteq \ldots \subseteq S_\ell$ with $\ell < n$. For each set we need to verify inequality (1), that is, compute the mincut of $G[S]$ and check that the deficiency of $G[S]$ is zero (recall that $\text{cost}(S, V \setminus S) = \text{mincut}(G[A])$ by construction of the PQ-DAG). The deficiency can be easily verified in $\mathcal{O}(n^2)$ time in total for all candidate sets due to them being subsets of each other. The computation of the mincuts is not as easy. While we are not aware of a way to circumvent ℓ separate mincut computations, we can reduce the size of the involved graphs such that all these graphs have in total size $\mathcal{O}(n + m)$. To this end, we require another lemma with structural insights; see Figure 5 (left) for an illustration of the setting.

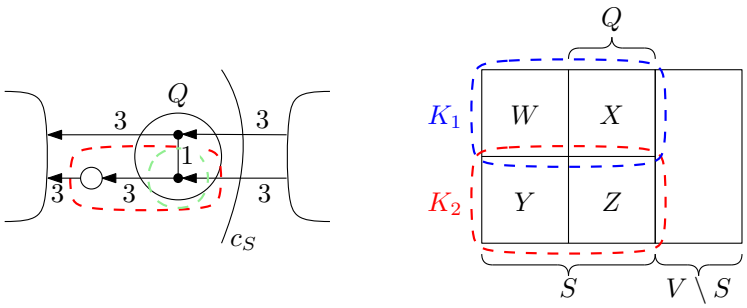

Figure 5 *Left:* The figure shows a example where $\text{mincut}(G[S]) < \text{cost}(c_S)$ given the setting of Lemma 3.4. Two mincuts of $G[S]$ are highlighted; one with a side contained in Q (green) and one with no side contained in Q (red). *Right:* Layout of the sets used during the proof of Lemma 3.4.

▶ **Lemma 3.4.** *Let $G = (V, E, \omega)$ be a weighted graph and $s, t \in V$. Further, let $c_S = (S, V \setminus S)$ be a minimum s-t-cut such that $s \in S$, $t \in V \setminus S$, $\text{mincut}(G[S]) < \text{cost}(c_S) = \text{mincut}(G)$, and there exists a cut node $Q \subseteq S$ of the PQ-DAG that separates $S \setminus Q$ from $V \setminus S$. Then there exists a cut $(Z, S \setminus Z)$ in $G[S]$ with $\text{cost}(Z, S \setminus Z) < \text{mincut}(G)$ and $Z \subseteq Q$.*

Proof. Let $c_K = (K_1, K_2)$ be a mincut of $G[S]$ with cost less than $\text{mincut}(G)$. First, we argue that c_K must split Q. Assume that c_K does not split Q and w.l.o.g. $Q \subseteq K_2$. Then, Q (and therefore also K_2) separates K_1 from $V \setminus S$. But this means that $(K_1, V \setminus K_1)$ is a cut of G with $\text{cost}(K_1, V \setminus K_1) = \text{cost}(K_1, K_2) < \text{mincut}(G)$. Hence, c_K must split Q.

Let $W = K_1 \setminus Q$, $X = K_1 \cap Q$, $Y = K_2 \setminus Q$, and $Z = K_2 \cap Q$. In the following, we will show that the cut of $G[S]$ that isolates Z is not larger than c_K which would prove the claim. The layout of the sets is visualized in Figure 5 (right).

Assume towards a contradiction that $\text{cost}(c_K) < \text{cost}(Z, S \setminus Z)$, i.e., $\text{cost}(Y \cup Z, W \cup X) < \text{cost}(Z, Y \cup W \cup X)$. Splitting the costs into their parts results in

$$\begin{aligned}&\text{cost}(Y, W) + \text{cost}(Y, X) + \text{cost}(Z, W) + \text{cost}(Z, X) = \text{cost}(Y \cup Z, W \cup X)\\ &< \text{cost}(Z, Y \cup W \cup X) = \text{cost}(Z, Y) + \text{cost}(Z, W) + \text{cost}(Z, X)\end{aligned}$$

and hence $\text{cost}(Y, W) + \text{cost}(Y, X) < \text{cost}(Z, Y)$.

With this we will get that the cut in $G[S]$ isolating $Y \cup W$ is strictly more expensive than the cut that isolates just W. In detail,

$$\begin{aligned}\text{cost}(X \cup Z, Y \cup W) &= \text{cost}(X, W) + \text{cost}(Z, W) + \text{cost}(Z, Y) + \text{cost}(X, Y)\\ &> \text{cost}(X, W) + \text{cost}(Z, W) + \text{cost}(Z, Y)\\ &> \text{cost}(X, W) + \text{cost}(Z, W) + \text{cost}(Y, W) + \text{cost}(Y, X)\\ &> \text{cost}(X, W) + \text{cost}(Z, W) + \text{cost}(Y, W)\\ &= \text{cost}(W, X \cup Y \cup Z)\end{aligned}$$

Since Q separates W and Y from $V \setminus S$, the cuts that isolate $Y \cup W$ or just W, respectively, have the same cost in G as they have in $G[S]$. Furthermore, the cut that isolates $Y \cup W$ in $G[S]$ is a mincut of G, because it is the cut in the PQ-DAG just before the cut node Q and thus has a cost of $\text{cost}(c_S) = \text{mincut}(G)$. But this means that isolating W in G is cheaper than the mincut. Refuting our assumption we prove the claim. ◀

Algorithm 2 Details to Line 4 in Algorithm 1.

Input: A connected weighted graph $G = (V, E, \omega)$, a vertex subset $A \subseteq V$, and a mincut $c_A = (A_1, A_2)$ for $G[A]$.

Output: A largest applicable set $S \subseteq A$ with $S \cap A_1 \neq \emptyset$ and $S \cap A_2 \neq \emptyset$ if existing, $\emptyset$ otherwise.

1 **Function** *LargestApplicableSetOverCut*(G, A, c_A)
2 **if** A *is applicable* **then return** A
3 **if** *Lemma 3.2 (e) excludes existence of* S **then return** $\emptyset$
4 **if** $V = A$ **then**
5 $uv \leftarrow$ arbitrary non-edge in A_i, $i \in \{1, 2\}$, with $\omega(uv) < 0$
6 $A_j \leftarrow$ side of c_A *not* containing uv
7 **return** max(*ApplicableSet*$(G, A_j, \{u\})$, *ApplicableSet*$(G, A_j, \{v\})$)
8 **else**
9 $A_j \leftarrow$ side of c_A with $\text{def}_G(A_j) = 0$ and $\text{cost}(A_j, V \setminus A) = 0$
10 $U \leftarrow$ vertices in A with neighbors in $V \setminus A$ // thus $U \cap A_j = \emptyset$
11 **return** *ApplicableSet*$(G[A], A_j, U)$
12 **Function** *ApplicableSet*(G, X, Y)
13 contract X into node s and Y into node t and construct PQ-DAG D
14 $Q_0, Q_1, \ldots, Q_p \leftarrow$ a reverse topological ordering of D // $s \in Q_0$ and $t \in Q_p$
15 $S \leftarrow \emptyset$
16 **for** $i \leftarrow 0$ **to** $q - 1$ **do**
17 $C_i \leftarrow$ vertices in G represented by $\bigcup_{j \leq i} Q_j$
18 **if** Q_i *is cut node in* D ***and*** C_i *is applicable* **then** $S \leftarrow C_i$
19 **return** S

Using Lemma 3.4, we can contract all vertices except Q_i before each mincut computation of S_i. Hence, we can efficiently compute the mincut for each candidate set S, resulting in the following overall theorem (see Algorithm 2 for pseudocode). We denote with $T_{\text{mincut}}(n, m)$ and $T_{s\text{-}t\text{-maxflow}}(n, m)$ the time to compute in an edge-weighted graph with n vertices and m edges a mincut and a maximum s-t-flow, respectively.

▶ **Theorem 3.5.** *Reduction Rule 2.2 can be applied in* $\mathcal{O}(n(T_{mincut}(n, m) + T_{s\text{-}t\text{-}maxflow}(n, m) + n^2))$ *time.*

Proof. We use Algorithm 1 with Line 4 being implemented with Algorithm 2.

All we need to show for the correctness of Algorithm 1 is that Algorithm 2 is correct. To this end, we need to show that if there is an applicable set S over the cut c_A, then Algorithm 2 will return such a set. (Note that Algorithm 2 never returns a non-applicable set as applicability is checked before returning a set.) By Lemma 3.3, the set S is characterized by a cut node in the PQ-DAG with s representing the side of c_A with deficit zero and t representing the vertices with neighbors in $V \setminus A$ (or t representing an endpoint of a non-edge with non-zero weight if $V = A$). Hence, all sets that could possibly be applicable are considered in Line 18. Thus, if an applicable set S exists, then an applicable set is returned.

It remains to show the claimed running time. To this end, start with Algorithm 1: If Lines 2 to 4 can be done in $\mathcal{O}(T_{\text{mincut}}(n, m) + T_{s\text{-}t\text{-maxflow}}(n, m) + n^2)$ time, then the claimed running time follows as there are at most $\mathcal{O}(n)$ recursive calls. The only nontrivial work in Lines 2 and 3 is the computation of a mincut. This can be done in $\mathcal{O}(T_{\text{mincut}}(n, m))$ time.

It remains to argue why Line 4 (that is, Algorithm 2) runs in $\mathcal{O}(T_{\text{mincut}}(n,m) + T_{s\text{-}t\text{-maxflow}}(n,m) + n^2)$ time: First, observe that the cost of the edges from A to $V \setminus A$ and of the edges from A_1 to A_2 can be simply transmitted in a recursive call of Algorithm 1. Hence, with simple bookkeeping, we can access $\text{cost}(A, V \setminus A)$ in constant time during the whole algorithm. Moreover, since to Algorithm 2 a mincut c_A of A is given, it follows that Lines 2 and 3 of Algorithm 2 can be done in $\mathcal{O}(n^2)$ time by simply iterating over all vertex pairs. The time to perform Lines 4 to 11 is dominated by the time required to execute the function in Line 12. Thus, it remains to show that this function can be computed in $\mathcal{O}(T_{\text{mincut}}(n,m) + T_{s\text{-}t\text{-maxflow}}(n,m) + n^2)$ time: Line 13 requires contraction of two vertex sets, the computation of one maximum s-t-flow, and the construction of the PQ-DAG from said flow. The maximum s-t-flow can be computed in $\mathcal{O}(T_{s\text{-}t\text{-maxflow}}(n,m))$ time (note that the contraction of X and Y results in a graph of size $\mathcal{O}(n+m)$). The contraction of the vertices, the construction of the PQ-DAG [25], and the computation of a reverse topological order can all be done in linear time. Thus, Lines 13 and 14 require $\mathcal{O}(T_{s\text{-}t\text{-maxflow}}(n,m))$ time.

It remains to argue that the for-loop in Lines 16 to 18 can be done in $\mathcal{O}(T_{\text{mincut}}(n,m)+n^2)$ time. The bottleneck here is the check whether the candidate set C_i is applicable in Line 18. By inequality (1) and Lemma 3.1, this involves checking for each C_i: (a) $\text{cost}(C_i, V \setminus C_i) = \text{cost}(C_i, A \setminus C_i)$, (b) $\text{def}_G(C_i) = 0$, (c) $\text{mincut}(G[C_i]) = \text{cost}(C_i, A \setminus C_i)$. By construction of the PQ-DAG we have $\text{cost}(C_i, V \setminus C_i) = \text{cost}(C_i, A \setminus C_i)$ (only vertices in $Y \subseteq Q_p$ can have neighbors outside A). For (b) we only need to check vertex pairs within C_i that were not checked in the previous iteration C_{i-1} as $C_{i-1} \subseteq C_i$. Thus, for all sets $C_0, \ldots, C_{q-1}$ this can be done in $\mathcal{O}(n^2)$. (In fact, if the first set C_i has deficiency larger than zero, then this holds for all subsequent sets and the loop can be aborted.) It remains to show that (c) can be done in $\mathcal{O}(T_{\text{mincut}}(n,m))$ time: To this end, we only check if $\text{mincut}(G[C_i]) < \text{cost}(C_i, A \setminus C_i)$ as $\text{mincut}(G[C_i]) > \text{cost}(C_i, A \setminus C_i)$ would contradict $(C_i, A \setminus C_i)$ being a mincut of A. Exploiting Lemma 3.4, we contract all vertices in $C_i \setminus Q_i$ into one vertex x and compute a mincut c_x in the resulting graph $G[\{x\} \cup Q_i]$. If $\text{cost}(c_x) < \text{cost}(C_i, A \setminus C_i)$, then we know that C_i is not applicable as $\text{mincut}(G[C_i]) \leq \text{cost}(c_x)$. Otherwise, if $\text{cost}(c_x) \geq \text{cost}(C_i, A \setminus C_i)$, then, by Lemma 3.4, we know that $\text{mincut}(G[C_i]) = \text{cost}(C_i, A \setminus C_i)$ and C_i is applicable. Thus, the running time is

$$\sum_{i=1}^{p-1} T_{\text{mincut}}(|Q_i + 1|, |E(Q_i)| + |Q_i|) \in O(T_{\text{mincut}}(n, n+m)) = O(T_{\text{mincut}}(n,m))$$

as we assume the input graph to be connected. Thus, Reduction Rule 2.2 can be applied in $\mathcal{O}(n \cdot (T_{\text{mincut}}(n,m) + T_{s\text{-}t\text{-maxflow}}(n,m) + n^2)))$ time. ◀

Since $T_{\text{mincut}}(n,m) \in \mathcal{O}(m^{1+o(1)})$ [20, 23] and $T_{s\text{-}t\text{-maxflow}}(n,m) \in m^{1+o(1)}$ for polynomially bounded capacities [12], it follows that Reduction Rule 2.2 can be applied in $\mathcal{O}(nm^{1+o(1)} + n^3)$ time for polynomially bounded weights.

▶ **Corollary 3.6.** *If all weights are polynomially bounded, then Reduction Rule 2.2 can be applied in $\mathcal{O}(nm^{1+o(1)} + n^3)$ time*

4 Experimental Evaluation

In this section, we discuss the effectiveness and the recursion behavior of Algorithm 1. To this end, we provide a basic implementation as proof of concept to demonstrate by how much several graphs can be reduced when Reduction Rule 2.2 is applied exhaustively. Moreover, we briefly analyze the recursion depth of Algorithm 1.

Implementation. We implemented Algorithm 1 in the programming language Julia. The implementation and the test setup can be found on Github[1].

Our implementation of Algorithm 1 resolves the conquer step of finding a candidate set over the mincut $c_A = (A_1, A_2)$ in Line 4 as follows: First A and the restrictions in Lemma 3.2 (e) are tested (exactly as in Lines 2 and 3 in Algorithm 2). Note that in *all* test instances one of the two if statements were triggered, that is, either the current set A was applicable or Lemma 3.2 (e) certifies that there is no applicable set over the cut c_A. Hence we did not implement the sophisticated approach presented in Algorithm 2. Instead, as a fallback, our implementation would find such an applicable set S by simply iterating over *all* mincuts (see discussion after Lemma 3.1).

If an applicable vertex set S is found in the conquer step (Line 4 of Algorithm 1), then we do *not* recurse (Line 5). Instead, we merge the vertices in S and run the algorithm again on the newly created graph. Thus, our implementation *exhaustively* applies Reduction Rule 2.2.

To compute one (and all) mincut(s) of a graph, we use the implementation by Henzinger et al. [19], which uses integer weighted graphs. Therefore our implementation also requires integer weighted input graphs.

Setup. To test the implementation, we used the weighted instances that were converted into unweighted instances for the *PACE Challenge 2021*[2]. This dataset includes primarily biologically motivated graphs and additionally randomly generated graphs. For a more detailed description of the dataset we refer to the PACE report [21].

We treat each connected component of the test graphs as a single graph. We only tested the algorithms on graphs with 100 or more vertices and only graphs which are not cluster graphs already, as such instances are easy to solve. This resulted in 204 different graphs from the PACE challenge dataset, with 150 real-world instances and 54 randomly generated instances. The largest graphs have around $3,000$ vertices. The edge weights are floating-point values. As our implementation uses integer weights, we multiplied the edge weights by a factor of 1000 and rounded afterwards.

We also implemented two other approaches for finding applicable sets to compare the results of our algorithm. The first one is the *Large Neighborhood* (LN) approach, similar to the data reduction rule used by Cao and Chen [9] in their $2k$-vertex kernel for Weighted Cluster Editing. They essentially test for every vertex $u \in V$ whether the closed neighborhood $N[u]$ is applicable. The second one, simply called *Heuristic*, is the implementation of the heuristic presented by Böcker et al. [8] for applying Reduction Rule 2.2, which we embedded in our implementation. Both of these approaches are run exhaustively. Note that both these approaches can fail to find applicable sets although the graph contains such a set, see Figure 2 for an example where the *Large Neighborhood* approach fails.

4.1 Results

Effectiveness. We were first interested in how much the graph size shrank after applying the data reduction rule with the various algorithms. As one can see in Figure 6 (left side), Algorithm 1 merges 24.2% of the vertices of the real-world instances, roughly three times the amount of vertices, compared to the LN approach, which merges 8.1% of the vertices. The algorithms perform very poorly on the random instances, with nearly the same amount

[1] `https://github.com/venondev/AlmostCliquePoly`

[2] The scripts for collecting and converting the graphs can be found at `https://github.com/PACE-challenge/Cluster-Editing-PACE-2021-instances`.

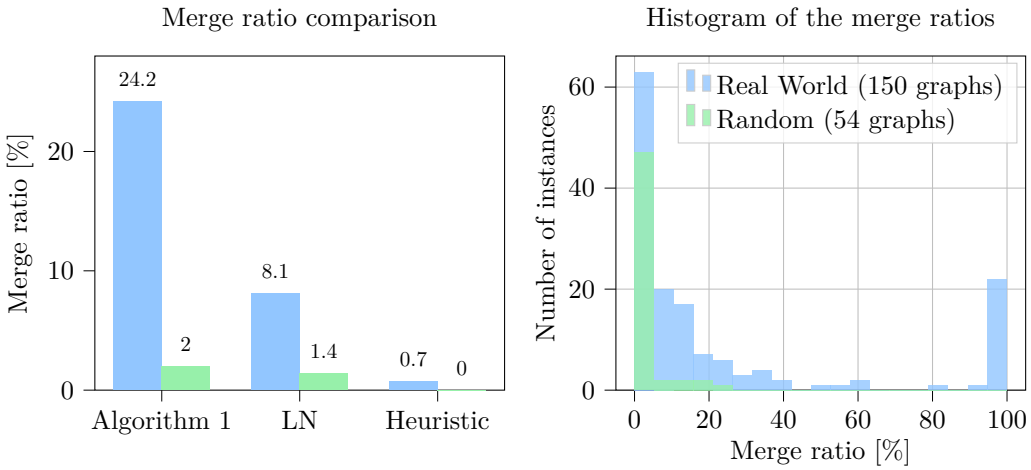

Figure 6 *Left:* Comparison of the average merge ratio (number of vertices merged / n) of the three algorithmic approaches Algorithm 1, "Large Neighborhood" (LN) by Cao and Chen [9], and "Heuristic" by Böcker et al. [8] on the two instance categories real world and random. *Right:* Two histograms (blue behind green) showing the number of instances with respect to the merge ratio (number of vertices merged / n) for Algorithm 1. Note that the input graphs were not cluster graphs.

of merged vertices, 1.4% using the LN approach and 2% using Algorithm 1. The heuristic implementation by Böcker et al. [8] does not perform well on the dataset, with 0.7% for real-world instances and 0% for random instances.

When looking at the histogram of merge ratios for Algorithm 1 in Figure 6 (right side), it shows that for most graphs of the test set the rule reduces the graph size only slightly or not at all. But there are also 23 graphs which got solved *almost* entirely, that is, the resulting instance contained at most 5% of the initial number of vertices. When using Algorithm 1, 11.3 % of the instances got solved completely, around 1.5 % when using the LN approach, and 0.5% when using the heuristic by Böcker et al. [8].

Recursive calls of Algorithm 1. The depth of the recursion of Algorithm 1 is between $\log(n)$ (splitting the graph in half in each recursive call) and n (cutting exactly one vertex off in each recursive call). In our implementation the depth can be smaller than $\log(n)$: If we find an applicable set, then we merge it and run the algorithm again. The reported recursion depth is then the maximum over all runs on the instance.

Unfortunately, we observe that for most instances the recursion depth is close to n, see Figure 7 for an overview. This means that in most recursion steps, the mincut only cuts out a single vertex. Consequently, the size of the set $A \subseteq V$ that the algorithm looks at within each recursion step only decreases slowly. As a result, on most on the instances Algorithm 1 computes many mincuts on large graphs, resulting in high running times.

A preliminary test underlined the issue with the running time: For most of the instances, our basic implementation of Algorithm 1 is more than ten times slower than the LN approach. For some instances, our basic implementation is up to 1000 times slower.

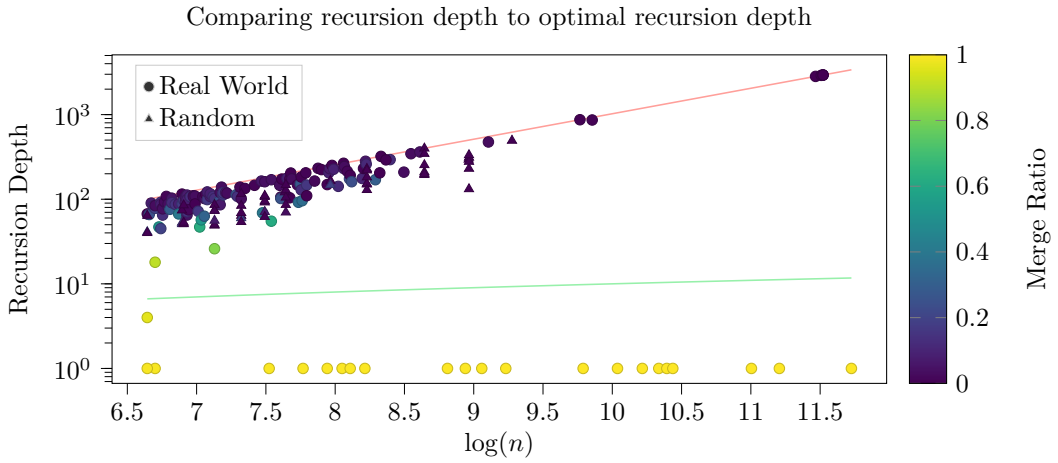

Figure 7 Comparing the recursion depth of the instances. If Algorithm 1 was run multiple time on an instance (due to merging some an applicable vertex set), then the maximum recursion depth over all runs is plotted. The red line denotes the upper bound of the recursion depth, which is the number of vertices n. If the algorithm does not find a set of vertices to merge, then the green line denotes the lower bound $\log(n)$ of the recursion depth. The instances are colored according to their respective merge ratio (number of vertices merged / n). The only instances below (or close) to the green line are instances with merge ratio close to one.

4.2 Summary

Our experiments show that Reduction Rule 2.2 can work well on the test dataset, reducing the real-world graphs by 24% on average if applied exhaustively. In this regard Algorithm 1 outperforms the other approaches. Notably, Algorithm 1 solves 11% of the graphs completely, including some larger graphs with up to $3,000$ vertices. Unfortunately, Algorithm 1 still comes at the cost of a high running time. Hence, we suggest that (an improved implementation of) Algorithm 1 could be used as preprocessing before running a branch&bound solver but probably not during branching itself.

One avenue for improving the implementation is the computation of a mincut. Currently, if a mincut separates one vertex from the graph, then in the next recursive call the algorithm of Henzinger et al. [19] is cold-started to compute a new mincut on the slightly altered graph. Here the use of, for example, the dynamic algorithm of Henzinger et al. [18] seems promising.

5 Conclusion

In this work we provided the first polynomial-time algorithm to apply Reduction Rule 2.2. As the current running time is still quite high, an immediate open question is about better theoretical guarantees. Dealing with the bad recursive behavior would be a first step in this direction. Another question is whether our algorithm could be made working with approximate cuts instead of (optimum) mincuts. On the practical side, our experiments demonstrate the potential effectiveness of Reduction Rule 2.2 in real world instances, if applied exhaustively. Thus, the question is whether there are better tradeoffs between effectiveness and efficiency and whether state-of-the-art solvers [5] would benefit.

References

1 Faisal N. Abu-Khzam. On the complexity of multi-parameterized cluster editing. *Journal of Discrete Algorithms*, 45:26–34, 2017.

2 Faisal N. Abu-Khzam, Judith Egan, Serge Gaspers, Alexis Shaw, and Peter Shaw. Cluster editing with vertex splitting. In *Proceedings of the 5th International Symposium on Combinatorial Optimization (ISCO '18)*, volume 10856 of *LNCS*, pages 1–13. Springer, 2018.

3 Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56:89–113, 2004. doi:10.1023/B:MACH.0000033116.57574.95.

4 Amir Ben-Dor, Ron Shamir, and Zohar Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6(3-4):281–297, 1999. doi:10.1089/106652799318274.

5 Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. A branch-and-bound algorithm for cluster editing. In *Proceedings of the 20th International Symposium on Experimental Algorithms (SEA '22)*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

6 Sebastian Böcker. A golden ratio parameterized algorithm for cluster editing. *Journal of Discrete Algorithms*, 16:79–89, 2012. doi:10.1016/j.jda.2012.04.005.

7 Sebastian Böcker and Jan Baumbach. Cluster Editing. In *Proceedings of the 9th Conference on Computability in Europe, CiE 2013*, volume 7921 of *LNCS*, pages 33–44. Springer, 2013.

8 Sebastian Böcker, Sebastian Briesemeister, and Gunnar W. Klau. Exact algorithms for cluster editing: Evaluation and experiments. *Algorithmica*, 60(2):316–334, 2011. doi:10.1007/s00453-009-9339-7.

9 Yixin Cao and Jianer Chen. Cluster Editing: Kernelization based on edge cuts. *Algorithmica*, 64(1):152–169, 2012.

10 Jianer Chen and Jie Meng. A $2k$ kernel for the cluster editing problem. *Journal of Computer and System Sciences*, 78(1):211–220, 2012.

11 Jiehua Chen, Hendrik Molter, Manuel Sorge, and Ondrej Suchý. Cluster editing in multi-layer and temporal graphs. In *Proceedings of the 29th International Symposium on Algorithms and Computation (ISAAC '18)*, volume 123 of *LIPIcs*, pages 24:1–24:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

12 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *CoRR*, abs/2203.00671, 2022. doi:10.48550/arXiv.2203.00671.

13 Michael R. Fellows, Michael A. Langston, Frances A. Rosamond, and Peter Shaw. Efficient parameterized preprocessing for Cluster Editing. In *Proceedings of the 16th International Symposium on Fundamentals of Computation Theory (FCT '07)*, volume 4639 of *LNCS*, pages 312–321. Springer, 2007. doi:10.1007/978-3-540-74240-1_27.

14 Fedor V. Fomin, Stefan Kratsch, Marcin Pilipczuk, Michał Pilipczuk, and Yngve Villanger. Tight bounds for parameterized complexity of Cluster Editing with a small number of clusters. *Journal of Computer and System Sciences*, 80(7):1430–1447, 2014.

15 Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Graph-modeled data clustering: Exact algorithms for clique generation. *Theory of Computing Systems*, 38(4):373–392, 2005.

16 Jiong Guo. A more effective linear kernelization for cluster editing. *Theoretical Computer Science*, 410(8-10):718–726, 2009. doi:10.1016/j.tcs.2008.10.021.

17 Sepp Hartung and Holger H. Hoos. Programming by optimisation meets parameterised algorithmics: a case study for cluster editing. In *Proceedings of the 9th International Conference on Learning and Intelligent Optimization, LION 2015*, volume 8994 of *LNCS*, pages 43–58. Springer, 2015. doi:10.1007/978-3-319-19084-6_5.

18 Monika Henzinger, Alexander Noe, and Christian Schulz. Practical fully dynamic minimum cut algorithms. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX 2022)*, pages 13–26. SIAM, 2022. doi:10.1137/1.9781611977042.2.

19 Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Finding all global minimum cuts in practice. In *Proceedings of the 28th Annual European Symposium on Algorithms (ESA '20)*, volume 173, pages 59:1–59:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

20 David R Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000.

21 Leon Kellerhals, Tomohiro Koana, André Nichterlein, and Philipp Zschoche. The PACE 2021 Parameterized Algorithms and Computational Experiments Challenge: Cluster Editing. In *Proceedings of the International Symposium on Parameterized and Exact Computation (IPEC '21)*, volume 214, pages 26:1–26:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.IPEC.2021.26.

22 Christian Komusiewicz and Johannes Uhlmann. Cluster editing with locally bounded modifications. *Discrete Applied Mathematics*, 160(15):2259–2270, 2012.

23 Jason Li. Deterministic mincut in almost-linear time. *CoRR*, abs/2106.05513, 2021.

24 Hiroshi Nagamochi, Yoshitaka Nakao, and Toshihide Ibaraki. A fast algorithm for cactus representations of minimum cuts. *Japan Journal of Industrial and Applied Mathematics*, 17(2):245–264, 2000.

25 Jean-Claude Picard and Maurice Queyranne. On the structure of all minimum cuts in a network and applications. In *Combinatorial Optimization II*, pages 8–16. Springer, 1980.

26 Ron Shamir, Roded Sharan, and Dekel Tsur. Cluster graph modification problems. *Discrete Applied Mathematics*, 144(1-2):173–182, 2004.

27 Esther Ulitzsch, Qiwei He, Vincent Ulitzsch, Hendrik Molter, André Nichterlein, Rolf Niedermeier, and Steffi Pohl. Combining clickstream analyses and graph-modeled data clustering for identifying common response processes. *Psychometrika*, 86(1):190–214, 2021. doi:10.1007/s11336-020-09743-0.