Algorithm Engineering for Cluster Editing

Sebastian Paarmann

A thesis submitted in
fulfillment of the requirements for the award of
Bachelor of Computer Science

June 16, 2021

I hereby declare that this thesis entitled "Algorithm Engineering for Cluster Editing" is the result of my own research except as cited in the references. This thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Signature :

Student : Sebastian Paarmann

Date :

Supervisor : Prof. Dr. Matthias Mnich

Co-Supervisor: Dr. Jens M. Schmidt

Acknowledgment

I would like to thank my supervisors for this thesis, Prof. Dr. Mnich, and Dr. Schmidt. They suggested participation in the PACE challenge and provided a lot of good advice and ideas for the implementation. It was a challenging but interesting topic and I learned a lot while working on it.

Additional gratitude goes to the Manke Förderstiftung Henstedt-Ulzburg, and especially Monika and Volker Manke, for the support I have received over the course of my studies at the TUHH.

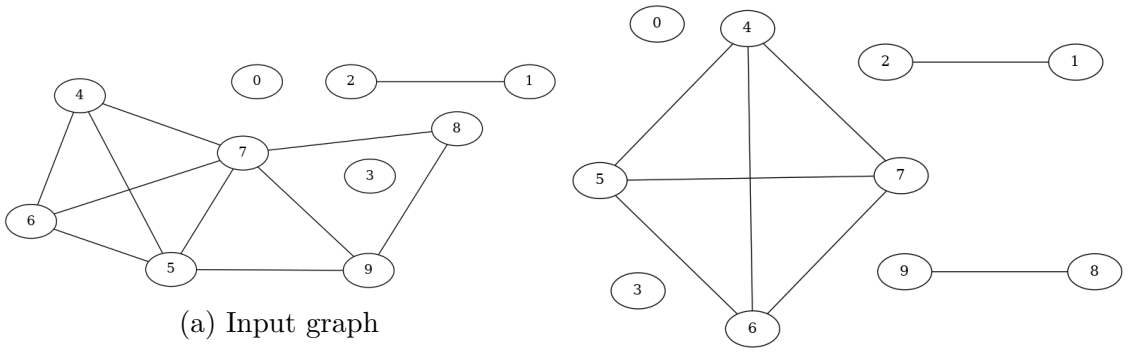
And last — but very much not least — I want to thank my friends and family for all of their support, as well as the opportunities to actually enjoy the time off from working on this thesis. The last year or so has not been easy on anyone, but it was immeasurably better because of all of you.

Sebastian Paarmann, Henstedt-Ulzburg

Contents

1 Introduction

A *cluster* graph, also called a *transitive* graph, is a graph which consists only of disjoint cliques. In the Cluster Editing problem, the input is an undirected graph G and an integer k greater 0, and the question is whether G can be made into a cluster graph using k or less *edge edits*. An edit is either inserting an edge or deleting an existing one. We also consider the Weighted Cluster Editing problem: The input has weights for each pair of vertices, both edges and non-edges, which are the deletion and insertion costs respectively. The question is then whether there is a solution with total edit costs at most k. See Figure 1.1 for an example: On the left is the first and smallest test instance used while developing the solver that is subject of this thesis and right is the cluster graph resulting from an optimal solution.



(a) Input graph

(b) Resulting cluster graph

Figure 1.1: First test instance of the PACE challenge

For this thesis, we have implemented a Cluster Editing solver as a submission to the Parameterized Algorithms and Computational Experiments (PACE) Challenge 2021 [23]. Among other goals, the aims of the PACE challenge include producing freely available performant implementations to the posed problems while also encouraging new theoretical developments. In the 2021 challenge, there are three different tracks for tackling different aspects of the problem: The *exact* track asks to find an optimal solution for a given graph as quickly as possible, the *heuristic* track asks to find a solution that is as good as possible within a time limit, and the *kernelization* track asks to reduce an input graph into a smaller equivalent instance. Our implementation is intended for the exact track.

Cluster Editing was initially introduced in the context of bioinformatics to clean data containing measurement errors by clustering gene expression patterns [1]. It has since been applied to a variety of problems in bioinformatics, including gene sequencing, comparative transcriptomics, metabolic profiling in spectrometry data, and more [25]. The Correlation Clustering problem, of which Cluster Editing is one important special case, has applications in machine learning contexts, for example to perform deduplication of data [6].

2 Preliminaries

First, some terms and notation used throughout this thesis: A graph $G = (V, E)$ has a vertex set V and an edge set $E \subseteq \binom{V}{2}$. Graphs are simple graphs, i.e. they contain no self-loops (an edge connecting a vertex with itself) or parallel edges. We abbreviate an edge or non-edge $\{u, v\} \in \binom{V}{2}$ as uv. We denote the *(open) neighborhood* of a vertex v in G as $N_G(v) = \{u \in V \mid uv \in E\}$. Similarly, $N_G[v]$ is the *closed neighborhood* of v, i.e. $N_G(v) \cup \{v\}$. If it is clear which graph is meant, we may also just write $N(v)$ and $N[v]$. For a set $U \subseteq V$, $G[U]$ is the subgraph *induced* by U on G, i.e. $G[U] = (U, \{uv \in E \mid u \in U \wedge v \in U\})$. In general, a *clique* is a fully connected graph. A clique in a graph G is a fully connected induced subgraph of G. A weighted graph is characterized by a weight function $s\colon \binom{V}{2} \to \mathbb{Z}$. We consider uv to be an edge if $s(uv) > 0$ and uv to be a non-edge if $s(uv) \leq 0$. We write $A \Delta B$ for the symmetric difference of two sets A and B: $A \Delta B = (A \setminus B) \cup (B \setminus A)$.

Formally the Cluster Editing problem can be described as follows: The input (G, k) consists of a simple unweighted graph $G = (V, E)$ and parameter $k \in \mathbb{N}$. A set $F \subseteq \binom{V}{2}$ is a *cluster editing* for G if the graph $(V, E \Delta F)$ is a cluster graph, i.e. a disjoint union of cliques. The *cost* of a cluster editing F is $|F|$. The task is to find a cluster editing F with cost less than k. In the PACE challenge, we want to solve the optimization version of this problem: The input is only a graph G and the goal is to find an optimal solution, i.e. a solution with the smallest amount of edits possible.

In the Weighted Cluster Editing problem, instead of an unweighted graph, the input contains a weighted graph encoded by the weight function $s\colon \binom{V}{2} \to \mathbb{Z}$. The cost of a cluster editing F is then $\sum_{e \in F} |s(e)|$. Our solver is intended for the unweighted problem, as required by the PACE challenge, but it internally converts its input into a weighted problem instance for more efficient solving. It could be adapted to take a weighted instance with minimal modifications. One can also consider the weighted problem with real-valued weights, but we will not do so further here.

Cluster Editing is NP-complete [3], so there is no polynomial time algorithm for solving it (unless P = NP). It is however *fixed-parameter tractable*. Fixed-parameter tractability (FPT) is a concept from parameterized complexity theory. A problem is called fixed-parameter tractable if there is an algorithm to solve it in time $f(k) * n^c$, where n is the size of the input, k is some additional parameter, c is a constant independent of both n and k, and f is some computable function. In general there can be many different parameters k for a single problem, for example some measure of the complexity of the input (e.g. maximum degree in a graph) or a limit on the size of the solution. In this way the exponential increase in runtime can be contained in the parameter, instead of the algorithm scaling exponentially with the input size. More detailed information can be found e.g. in the book of Cygan et al. [26]. In our case k describes the maximum amount of edits, i.e. the maximum size of the solution, while n is some measure of the graph size, for example

$|V|$. To solve the optimization problem that is required by the PACE challenge with a fixed-parameter algorithm, we simply execute the algorithm in a loop starting with $k = 1$ and increase k until we find a solution.

We make use of some important techniques from the field of parameterized algorithms, mainly *kernels* and data reduction rules, and *bounded search trees*. Data reduction rules take as input a problem instance, including a parameter, and produce a second instance that is equivalent to the first one. Two instances are equivalent if there is a solution for the second if and only if there is also a solution for the first instance. The goal then is to return an instance that is in some way *easier*, for example by making the input smaller or reducing the parameter. We describe the various reduction rules used in our algorithm in section 4.3. A kernel is a preprocessing algorithm to reduce the size of a problem instance for which an upper bound on the size of the returned instance can be given in terms of the input parameter k. Kernels are often implemented by using multiple reduction rules that are repeatedly applied.

Apart from the reduction rules, the general shape of the algorithm we implement is that of a bounded search tree algorithm, also called a branching algorithm. This is essentially the idea of backtracking: The algorithm takes some decision regarding the solution (e.g. adding or removing an edge) and then recursively executes on the resulting instance. If that recursive call does not yield a solution, a different decision is taken instead and the procedure is repeated. If it can be guaranteed that some sequence of decisions will lead to a solution if one exists, this kind of algorithm correctly solves the problem. It can be viewed as recursively moving through a search tree where each node is a "decision point". As the name suggests, the search tree should also be bounded: There should be a limit on the amount of branches taken in each node as well as a guarantee that after each decision the instance is substantially simpler such that we will eventually reach a leaf node where the solution (or its non-existence) is trivial.

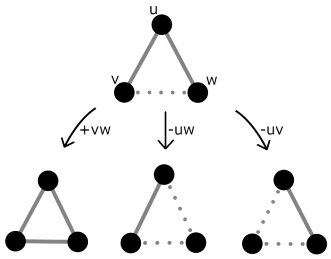


Figure 2.1: A conflict vuw and three ways to resolve it.

As an example, let us describe a simple search tree algorithm for Cluster Editing. We call three vertices $v, u, w \in V$ a *conflict triple* if $uv, uw \in E, vw \notin E$, i.e. the vertices form an induced P_3. A central observation regarding the Cluster Editing problem is that a graph is a cluster graph if and only if there is no conflict triple in the graph: If there is a conflict, v, u, and w are all part of the same connected component, but this component is not a clique because $vw \notin E$. One can also immediately see that if each connected component is a clique, there cannot be a conflict triple.

This leads directly to a search tree algorithm: The problem can be solved by recursively searching for a conflict triple and branching into three cases: add vw, remove uv, or remove uw, as illustrated in Figure 2.1. These are the only ways of resolving a conflict, and once there are no more conflicts, a solution can be returned. Looking at our criteria for a bounded search tree from above, this has a limit on the branches per node, exactly three. And for each branch we perform an edit on the graph and thus reduce k by 1, limiting the depth of the search. This algorithm gives a search tree of size $O(3^k)$, though we will see in the next sections that more efficient trees are possible.

3 Previous Work

Clustering problems in general have a very long history, so we will focus only on the specific Cluster Editing problem as formulated above. In 1999, Ben-Dor et al. [1] introduced the unweighted problem in order to perform clustering on gene expression patterns. They provided a stochastic model for the corruption of the true clustering introduced by measurements and present an $O(n^2 \log(n)^c)$ time algorithm that reconstructs the true clustering with high probability, as well as a heuristic approach based on a similar idea. Many other approaches for clustering gene expression data had been proposed and investigated already at this time, see [2] for an overview from 2001.

Shamir et al. [3] investigated Cluster Editing as well as the related Cluster Completion (only edge insertions allowed) and Cluster Deletion (only edge deletions allowed) problems further. They showed that Cluster Completion can be solved in polynomial time, and Cluster Deletion is NP-hard to approximate to within some constant factor, i.e. there is some ϵ such that it is NP-hard to approximate it to within a factor of $1 + \epsilon$. Most relevant for us, they showed that Cluster Editing is NP-complete. Additionally they also considered variants of the problems with a fixed number of clusters as input, which we will not further discuss here. Gramm et al. [5] also mention that the NP-completeness of Cluster Editing can already be derived from the work of Křivánek and Morávek [4], published in 1986.

Bansal et al. independently researched a group of problems they call Correlation Clustering [6]. In this formulation, "minimizing disagreements" between the input labelling and the output clustering is equivalent to the unweighted Cluster Editing problem, which they gave a constant-factor approximation for. This result was improved on in 2005 by Charikar et al. [7], who showed the problem to be APX-hard and gave an approximation with a constant factor of 4.

In 2003 Gramm et al. [5] gave initial results regarding fixed-parameter approaches for Cluster Editing specifically. Before that, Cai [8] investigated fixed-parameter tractability of graph modification problems characterized by forbidden subgraphs in a more general sense, the results of which imply existence of an $O(k^3 * |G|^4)$ time algorithm for Cluster Editing. Gramm et al. gave an $O(k^3)$ vertex kernel for the problem, as well as a straightforward $O(k^3 + n^3)$ time fixed-parameter algorithm. Additionally, they showed a more advanced branching strategy that improves the search tree size to $O(2.27^k)$, leading to an $O(2.27^k + n^3)$ time algorithm. This approach was implemented in practice for the first time and empirically compared to an LP-based solver by Dehne et al. in 2006 [9]. They found that the $O(2.27^k)$ search tree is indeed faster in practice than the simpler $O(k^3)$ branching. Gramm et al. also presented a method for generating search tree algorithms in a computer-assisted fashion and used it to show a $O(1.92^k)$ search tree [24].

Another avenue of research was started in 2007 by Rahmann et al. [10]. This introduced the first fixed-parameter algorithm for the weighted version of the problem, along with several data reduction rules to speed it up. They also introduced two heuristic algorithms.

Building on this, Böcker et al. published several papers in 2008. They introduced a variant of the refined branching strategy of Gramm et al. [5] for the weighted problem, resulting in an $O(2.42^k)$ time algorithm [11]. Most importantly, this is the first paper to introduce the idea of merging two vertices, which is an operation only possible on the weighted problem. As a result, with merging, Böcker et al. found the simpler $O(3^k)$ branching faster in practice than the refined branching strategy, largely due to the significant effect of merging. Merging is also the basis for a surprisingly simple $O(2^k)$ branching strategy [12] which means the fastest practical approach to solving the unweighted problem is now to transform it into a weighted instance and solve that. The paper also immediately improved upon this with an $O(1.82^k)$ search tree that is achieved by more carefully choosing which edge to branch on. This beat even the previous best theoretical size for the unweighted problem of $O(1.92^k)$. Lastly, Böcker et al. performed practical experiments with these algorithms [13], comparing them with a cut-and-branch based integer-linear programming implementation. They found both approaches to be competitive, with different performance characteristics. Notably, good data reduction rules make the FPT approach feasible even for graphs that require several thousand edge modifications, despite the worst-case running times.

This basic approach was improved further by Böcker and Damaschke. First, Damaschke introduced a theorem concerning the structure of graphs in which no edge is part of three conflict triples [16]. This was used to improve the search tree size to $O(1.76^k)$ [14]. Then zero-edges (which will be discussed in more detail later) can be used to show additional structural properties, allowing even more efficient branching for an $O(1.62^k)$ search tree [15].

In parallel, multiple problem kernels for Cluster Editing have also been developed. Note that we discuss them here separately only to make it easier to follow. In practice the work on kernels is often what makes the better algorithms possible: Better reduction rules introduced as part of a kernel allow constructing and proving a faster search tree algorithm. In fact, these are often developed as a single work. Such it is for the first kernel given by Gramm et al. [5], which has at most $2k^2 + k$ vertices and can be computed in $O(n^3)$ time.

Protti et al. [17] gave a kernel with at most $2k^2 + 4k$ vertices, slightly worse than that of Gramm et al. but still $O(k^2)$. It is based on the modular decomposition of a graph and has the advantage of only requiring running time $O(n + m)$. In 2007, Fellows et al. introduced a "crown-type structural reduction rule" [18] and showed it can be used to construct a kernel with $6k$ vertices that can be computed in polynomial time.

Based on both the ideas of the crown-type reduction and the concept of *critical cliques*, which had already been used for good effect in fixed-parameter approaches to other problems, Guo [19] presented first another $6k$ kernel with $O(n^3)$ runtime and immediately improved this to a kernel with at most $4k$ vertices computable in $O(nm^2)$ time. In 2011, Chen and Meng [20] combined the critical cliques concept

with the notion of an *editing degree* of a vertex to give a $2k$ kernel which runs in $O(nm)$ time.

In 2015 Hartung and Hoos developed and published a Cluster Editing solver that outperforms previous implementations [27]. They used the *Programming by Optimization* paradigm, using automated methods to optimize algorithm behavior and parameter choice and guide further manual optimization effort based on empirical tests. Their implementation relies mainly on a reduction technique that is worse than newer ones in terms of theoretical asymptotic behavior but is able to produce significant runtime improvements in practice on their varied test data.

Lastly, there are also *integer linear programming* (ILP) formulations of the problem and associated solvers which we have already mentioned in relation to other work above. As we implemented only a bounded search tree algorithm, we do not go into further detail regarding these approaches but note the early work of Grötschel and Wakabayashi [28] and the modern yokisho solver [29] that combines an ILP solver with data reduction rules.

4 Our Implementation

In this section we will describe the implementation of our solver in some detail. It is based on many techniques introduced by the papers mentioned above. Those are only briefly summarized there but we will give more detailed information about those we have implemented here.

There are 4 parts to this section: We start off by describing the high-level structure of the solver, followed by a more in-depth look at the two basic modifying operations that are performed on the graph during execution of the algorithm. Next, we describe the reduction rules used, and finally some details on the practical implementation of the described algorithm, e.g. used data structures.

4.1 High-Level Overview

The solver consists of two main parts: The solver function, Algorithm 2, that performs interleaved reduction and the branching strategy, calling itself recursively; and a *driver*, Algorithm 1 that splits the input graph into its connected components and then calls the recursive solver function with increasing parameter k until a solution is found. Before calling the solver method, the driver also converts the input into a weighted problem instance and performs an initial set of reductions. This procedure is described in more detail in section 4.3.

As mentioned before, we implement a bounded search tree approach with multiple data reduction techniques. We currently use a very simple branching strategy: We take a conflict triple v, u, w and branch into two cases: merging uv, and forbidding uv. These operations will be described in more detail in the next section. Due to how merging leaves the final status of the edge from the merged vertex to w open, only two branches are required. This strategy was introduced by Böcker et al. [12] and shown to lead to a search tree size of $O(2.62^k)$. As described above, Böcker et al. have also introduced more advanced branching strategies, with even smaller search trees, which we would also like to implement but have not so far due to time constraints.

The reductions are split into several categories. There is the initial step of taking an unweighted instance and turning it into a reduced weighted instance. A set of parameter-independent reduction rules can also be applied directly after this, before the main search-tree algorithm. These parameter-independent rules are also applied interleaved during the search. And finally, any reductions that depend on having a maximum cost parameter k can be applied only once the search has started.

During the recursive search, we must track the current graph, which is modified for each recursive step, the current value for k as well as the set of edits done so far in the current branch. Because we modify the graph not just by inserting and deleting edges over the course of the algorithm, but also by merging vertices (which

effectively removes a vertex; see below for details), we also need to keep track of which vertices in the current graph correspond to which vertices of the original graph. One can either keep tracking edits throughout the branching search in terms of the original graph, or reconstruct the edit set at the end based on the final graph and the information about how vertices correspond. Theoretically, the latter is more optimal in terms of runtime because it only needs to do some work once the solution has been found, while the former approach has a slight overhead while searching. Our testing has indicated the difference is negligible however and we currently implement the former approach. What this means in practice is that if we perform an edit on the graph, we also look up which original vertices correspond to the two vertices involved and record edits for each possible pair.

Algorithm 1 Driver

```
function ComputeOptimalClusterEditing(G : Graph)
    edits ← ∅
    for all comp ∈ Components(G) do
        instance ← MakeWeightedInstance(comp)
        ▷ Initial reduction both reduces the instance and gives a lower bound
        (instance, lb1) ← InitialReduction(instance)
        lb2 ← LowerBound(instance)
        k ← Max(lb1, lb2)
        repeat
            solution = SolveClusterEditing(instance, k, ∅)
            k ← k + 1
        until solution ≠ ∅
        edits ← edits ∪ solution
    end for
    return edits
end function
```

4.2 Basic Operations

Ultimately we want to find a set of edge insertions and deletions. Thus, as we move through the search tree and execute reduction rules, inserting and deleting edges are precisely the operations we want to perform. However, if we for example delete an edge as part of branching into some part of the search tree, we want to ensure it does not get re-added later: All potential solutions with that edge not deleted are part of the other branch. If we did not ensure this, we would lose the guarantee of forward progress in the algorithm because we could get into a cycle of repeatedly deleting and adding an edge as branching operation.

As a consequence the operations we really want to perform are setting an edge to "permanent" or "forbidden" so the edit cannot be undone later in the same part of the search tree. Recall that we are working with a weighted graph, so (non-)edge weights encode the cost of inserting or deleting an edge, and k is the maximum cost we can still afford. Thus, we can simply set the weight of the edge to ∞ or $-\infty$ respectively. Clearly, with a finite k, the edges can then never be changed again.

Algorithm 2 Recursive Solver

```
function SolveClusterEditing(instance, k, edits)
    if k < 0 ∨ k < LowerBound(instance) then
        return ∅
    end if
    Reduction(instance, k, edits)
    if k < 0 ∨ k < LowerBound(instance) then
        return ∅
    end if
    if ¬HasConflictTriple(instance) then
        return edits
    end if
    (v, u, w) ← GetConflictTriple(instance)

    ▷ Branch 1: Forbid uv
    delInstance ← Clone(instance)
    delK ← k
    delEdits ← ForbidEdge(delInstance, delK, uv, edits)
    delSolution ← SolveClusterEditing(delInstance, delK, delEdits)
    if delSolution ≠ ∅ then
        return delSolution
    end if

    ▷ Branch 2: Merge uv
    mergeEdits ← MergeVertices(instance, k, uv, edits)
    mergeSolution ← SolveClusterEditing(instance, k, mergeEdits)
    return mergeSolution
end function
```

This also leads us to one of the largest advantages of working with a weighted graph: If we can guarantee two vertices will be part of the same cluster in any solution (in the current part of the search tree), i.e. we set them to permanent, we can actually *merge* them into a single vertex, thus reducing the size of the problem every time we perform this operation. Merging is only possible in a weighted context because we need weights to ensure the resulting problem instance is still equivalent to the previous one: Consider a vertex w adjacent to both u and v with $s(uw) = 1$ and $s(vw) = 1$. If we merge u and v into u' we need to now set $s(u'w) = 2$ because deleting the edge $u'w$ is equivalent to deleting both original edges uw and vw. This technique was introduced by Böcker et al. [11]. Let us now describe the two basic operations we perform in detail:

Forbidding Of the two, forbidding an edge is the much simpler operation. If uv currently exists, deleting it will generate cost. We thus reduce k by $\max\{s(uv), 0\}$. If $s(uv) > 0$, we record the edit(s) from removing the edge. We then set $s(uv) = -\infty$ to mark it as forbidden and prevent it from being reintroduced later.

Merging Merging two vertices u and v is more complicated. Conceptually, we introduce a new vertex u' and insert it into the graph while removing u and v. Merged vertices are considered to have a permanent edge between them (we can't later split them apart again), so if $s(uv) \leq 0$, we record the edit(s) from inserting the edge. This also generates cost, so we reduce k by $\max\{-s(uv), 0\}$. Then, for all $w \in V \setminus \{u, v\}$, we set $s(u'w) = s(uw) + s(vw)$.

It is possible that before the merge uw existed and vw did not, or vice-versa. In that case, this merge procedure will also effectively insert or delete one of these edges, depending on their weights. Any edits resulting from such a case are also recorded as edit, and reduce k appropriately. Note however that while *primary* edits (forbidding or merging uv) result in uvs new status being permanent and never changing again (in this part of the search tree), this kind of *secondary* edit occurring as a side-effect of merging can be changed again later in the algorithm, by editing $u'w$.

From a theoretical point of view there is an additional complication. The described merging procedure can create edges with weight 0, if $s(uw) = -s(vw)$. This leads to a problem when trying to prove asymptotic runtime / search tree size because it means we could perform branching operations that do not reduce the parameter k at all if only zero-edges are involved. To solve this Böcker et al. [12] reduce the parameter by 0.5 less than it otherwise would be when creating a zero-edge, and then reduce it by the remaining 0.5 when modifying the zero-edge's weight to something else again, thus guaranteeing some reduction in k for that operation.

4.3 Reduction Rules

Recall that we use reduction rules to transform the problem into an equivalent problem that is easier to solve. The general principle is to analyze the graph or some part of it (potentially also taking into account the current parameter) and prove that certain edits either will be made by an optimal solution, or will never be made by an optimal solution, and then take those decisions now, thereby shrinking the remaining search space. This leads to reduction rules that are *correct* in the sense that the reduced instance has a solution if and only if the original instance has one.

Critical Cliques One can obtain an equivalent weighted instance from an unweighted one by simply setting $s(uv) = 1$ for all $uv \in E$ and $s(uv) = -1$ for all $uv \notin E$. We use a slightly more complex procedure that can already reduce the graph while making it weighted, based on the notion of *critical cliques*.

Definition 4.3.1. A *critical clique* K is an induced clique in G where all vertices of K have the same set of neighbors outside of K, and K is maximal under this property.

An optimal cluster editing never splits a critical clique, i.e. every critical clique K of the input graph will be contained in a single cluster for any optimal solution [19]. Guo uses this observation and more to construct a $4k$ kernel for the unweighted cluster editing problem. We can take advantage of the technique of merging vertices instead: Since all vertices in a critical clique end up in the same

cluster, we can merge every critical clique into a single vertex, which will transform the unweighted input into an integer-weighted graph. The resulting graph has at most $4k_{opt}$ vertices [12, 19], providing a simple lower bound for the parameter at the start of the algorithm, and is equivalent in that if we find an optimal solution for this graph, we also have one for the original input.

Finding the critical cliques is possible in $O(n + m)$ time. In practice we implemented a simpler $O(n^2)$ time algorithm: Until all vertices have been assigned to a critical clique, choose one unassigned vertex to put in a new critical clique, and add every other unassigned vertex with the same closed neighborhood to the same critical clique. Our implementation time was constrained and this reduction is only ever executed once for every connected component of the input graph, so we focused our efforts on parts that have a more significant effect on total runtime instead of implementing the more complicated $O(n + m)$ time algorithm.

Parameter-independent reduction As described above, parameter-independent reduction can be applied once as reduction at the very start of the algorithm, but also during the search tree algorithm: After some branches in the search tree have been taken, more reduction operations may become applicable in the context of the current branch. Concretely, we implemented reduction rules 1–5 as introduced by Böcker et al. [13]. We repeat the rules here, but omit proofs of runtime and correctness. We do however give the general idea for why they are correct, along with notes about efficient implementation. Rules 1–3 are rather simple, so we will only give brief intuition for their correctness. A schematic representation of rules 1–3 can be seen in Figure 4.1.

Rule 1 *(heavy non-edge rule)* Forbid a non-edge uv with $s(uv) < 0$ if

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

Regarding correctness, if adding uv is more expensive than even isolating u completely would be, an optimal solution never inserts uv.

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

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

Regarding correctness, if removing uv is more expensive than editing every other edge and non-edge incident with u, an optimal solution never removes uv.

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

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

Regarding correctness, if removing uv is more expensive than isolating u and v completely (except from each other), an optimal solution never removes uv.

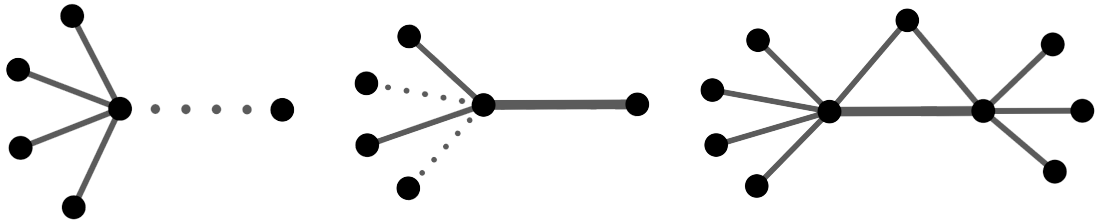


Figure 4.1: Illustrations of rules 1–3, from left to right

We collectively call rules 1–3 the "fast parameter-independent reduction rules." Böcker et al. apply these rules in every node of the search tree [13]. In our testing we found it to be better to apply even these faster reductions only at some interval. More details on our strategy for applying reductions can be found in section 5.

These rules can be applied most efficiently by combining all of them into a single pass. Our implementation is very similar to the strategy described by Böcker et al. [13] for achieving $O(n^3)$ runtime: For every pair uv we calculate and store

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

Note that rule 3 is symmetric, i.e. the order of u and v does not matter, and we can avoid calculating the same value twice. Rules 1 and 2 however are not symmetric, so the order does matter and a value is calculated for both orders.

Now the r values tell us for which pairs the respective rule can be applied: uv can be forbidden if $r_1(uv) \geq 0$ and uv can be merged if $r_2(uv) \geq 0$ or $r_3(uv) \geq 0$. We first construct a list of the entries for which this applies. Then we repeatedly take the next element from this list and apply the corresponding reduction, until no elements are left. Of course both forbidding and merging an edge can have effects on the applicability of the rules on other edges. By analyzing which r values can change by a given reduction and how, we construct a routine that can directly apply the change to all other affected r values, without having to recompute any entirely. Note that these updates can lead to both entries being removed from and entries being added to the list of values ≥ 0. For the full update procedure we refer to the source code associated with this thesis, where it is implemented and also described and justified in some detail.

Rule 4 *(almost clique rule)* Let k_C be the min-cut value of $G[C]$, for $C \subseteq V$. The vertices of C can be merged if

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

Correctness is again relatively easy to see: If the cheapest way of separating C into two (or more) clusters in the solution (i.e. the min-cut value of $G[C]$) is more

expensive than turning C into an isolated clique (adding any missing edges within C, removing any edges from C to $V \setminus C$), then we know an optimal solution will not split C into two (or more) clusters and can thus merge all vertices of C into one.

The more difficult part of applying this rule is deciding on which subsets C to check and possibly apply it for. Trying all possible subsets would be much too slow in even reasonably sized graphs. Böcker et al. describe a method of iteratively constructing subsets to test that can be executed quickly while still giving reasonable results: Start with a single vertex $C = \{u\}$ that maximizes $\sum_{v \in V \setminus \{u\}} |s(uv)|$. Then repeatedly add a vertex $w \in V \setminus C$ that has maximum connectivity in the set, i.e. one that maximizes $\sum_{v \in C} s(vw)$. Whenever the connectivity of the vertex w added is at least twice as large as that of the next-best vertex, try to apply rule 4 with the current C. Stop if the added vertex w has more edges to vertices in $V \setminus C$ than to vertices in C.

Now we come to the most complicated of the parameter-independent reduction rules, rule 5. First some definitions: For any $U \subseteq V$ define $s(v, U) := \sum_{u \in U} s(v, u)$. For an edge uv, define the *exclusive neighborhoods* of u and v as

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

and set $W := V \setminus (N_u \cup N_v \cup \{u, v\})$ as well as $\Delta_u := s(u, N_u) - s(u, N_v)$, $\Delta_v := s(v, N_v) - s(v, N_u)$. With this we can specify rule 5:

Rule 5 *(similar neighborhood)* Merge uv if

$$s(uv) \geq \max_{C_u, C_v} \min\{s(v, C_v) - s(v, C_u) + \Delta_v, s(u, C_u) - s(u, C_v) + \Delta_u\} \tag{4.1}$$

with the maximum running over all $C_u, C_v \subseteq W$ with $C_u \cap C_v = \emptyset$.

The correctness of this rule is not as obvious as for rules 1–4. Böcker et al. [13] give a full proof. Roughly speaking, one can show an upper bound for $s(uv)$ if u and v were to be in different clusters in an optimal solution based on analyzing the neighborhoods of u and v. If $s(uv)$ exceeds that upper bound, we know an optimal solution has u and v in the same cluster and can merge them.

Here too, performant implementation is of some concern. Mostly it is not obvious how to efficiently calculate the maximum running over all disjoint subsets of W. Böcker et al. describe a dynamic programming approach for calculating it in $O(|W|Z)$ time and $O(Z)$ space for $Z := \sum_{w \in W}(s(uw) + s(vw))$ which we have implemented and will describe in further detail here.

We need to partition W into three sets: C_u, C_v, and the remainder $R := W \setminus (C_u \cup C_v)$. First define the notation $\sum_u S := \sum_{w \in S} s(uw)$ and $\sum_v S := \sum_{w \in S} s(vw)$ for $S \subseteq W$. We want to find a partition that maximizes

$$\min\{\sum_v C_v - \sum_v C_u, \sum_u C_u - \sum_u C_v\}. \tag{4.2}$$

To find this maximum, we take a dynamic programming approach. Set $X := \sum_{w \in W} |s(uw)|$ and $Y := \sum_{w \in W} |s(vw)|$. Let $W = \{w_1, \ldots, w_k\}$. Then we define

boolean matrices $D_j[-X \dots X, -Y \dots Y], 0 \leq j \leq k$ where $D_j[x, y]$ is 'true' if there is a partition C_u, C_v, R of only $\{w_1, \dots, w_j\}$ such that

$$\sum_u C_u - \sum_u C_v = x \text{ and } \sum_v C_v - \sum_v C_u = y.$$

We can then determine the maximum of (4.2) as

$$\max_{D_k[x,y]='\text{true}'} \min\{x, y\}. \tag{4.3}$$

To find this maximum, we first observe that $D_0[x, y]$ is only 'true' for $(x, y) = (0, 0)$. For every subsequent w_j we can assign it to one of the three sets, giving the following recurrence with which we can calculate D_k in $O(kXY)$ time and $O(XY)$ space:

$$\begin{aligned} D_j[x, y] =& D_{j-1}[x, y] \\ & \vee D_{j-1}[x + s(u, w_j), y - s(v, w_j)] \\ & \vee D_{j-1}[x - s(u, w_j), y + s(v, w_j)] \end{aligned}$$

This quadratic time is clearly much better than the naive $O(|W|^3)$ approach but we can do even better with a linear time algorithm. Define $M_j[x]$ to be the *maximal* index y such that $D_j[x, y]$ is 'true'. Clearly $M_0[0] = 0$ and we otherwise initialize $M_0[x] = -\infty$ for $x \neq 0$. We can construct M_k using almost the same recurrence as D_k:

$$M_j[x] = \max\{M_{j-1}[x], M_{j-1}[x + s(u, w_j)] - s(v, w_j), M_{j-1}[x - s(u, w_j)] + s(v, w_j)\}$$

The maximum can then be computed as $\max_x \min\{x, M_k[x]\}$. This works because if we have two entries $D_j[x, y]$ and $D_j[x', y]$ with $x > x'$, the latter is "dominated" by the former in the sense that in the computation of (4.3) no descendant of the latter is used. To correctly apply (4.1) we also need to take into account Δ_u and Δ_v, so we can say that it is correct to merge uv if

$$s(uv) \geq \max_x\{\min\{x + \Delta_u, M_k[x] + \Delta_v\}\}.$$

We can take two additional shortcuts: If $s(uv) \leq \min\{\Delta_u, \Delta_v\}$, uv cannot possibly satisfy the real condition and we can skip to the next edge. On other other hand, if

$$s(uv) \geq (\sum_{w \in W} |s(uw) - s(vw)| + \Delta_u + \Delta_v)$$

then uv will definitely satisfy the real condition and we can merge uv without going through the dynamic programming steps. Lastly, we will point out that this description is slightly incomplete in the presence of weights that are $-\infty$. In that case the described procedure would lead to dynamic programming tables of infinite size which clearly cannot work. This can be solved by explicitly maintaining an extra entry for infinity in addition to the main table.

As rules 4 and 5 are more expensive to compute, we apply them on an even larger interval than that of rules 1–3, and we only apply rule 5 once no other rules can be applied.

Parameter-dependent reduction The last category of reduction rules is those requiring a parameter k to be applied. That means we can only use them once the branching search has started, and then interleave them with the branching strategy like the other rules.

As a parameter-dependent reduction rule, we implemented the *induced cost* reduction, also introduced by Böcker et al. [11]. The idea is relatively simple: For each pair uv we can quickly calculate a lower bound on the costs that setting them to forbidden or merging them will incur. If we see that we cannot afford forbidding them anymore, we can merge them now, and vice-versa. More specifically, define $\mathsf{icf}(uv)$ and $\mathsf{icp}(uv)$, the *induced cost of forbidding* uv and the *induced cost of setting* uv *permanent*, as:

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

These definitions are readily apparent: Forbidding uv requires that at some point all common neighbors of u and v must be separated from either u or v. Marking uv as permanent (which, in our case, would mean immediately merging it) requires that all non-common neighbors of u and v must at some point either be connected to the one they are not a neighbor of, or disconnected from the one they are a neighbor of. These already provide lower bounds which we can make slightly better by also including the cost of editing uv itself (forbidding it generates cost if it's not present, etc.). This leads to the following rules:

- For $u, v \in V$ where $\mathsf{icf}(uv) + \max\{0, s(uv)\} > k$: Merge u and v.
- For $u, v \in V$ where $\mathsf{icp}(uv) + \max\{0, -s(uv)\} > k$: Forbid uv.

This is already rather effective, but the better we can make the lower bound, the more effective the reduction becomes. In the next section we will describe a lower bound $b(G, uv)$ that completely ignores all edges uw and vw for all $w \in V \setminus \{u, v\}$ in its computation. It can then be safely added on to the previous bounds, resulting in $\mathsf{icf}(uv) + \max\{0, s(uv)\} + b(G, uv) > k$ and $\mathsf{icp}(uv) + \max\{0, -s(uv)\} + b(G, uv) > k$ as tests. This improved variant is significantly better than the less tight lower bound, even though it requires calculating this lower bound for each pair.

4.4 Lower Bounds

Computing lower bounds on the cost of a solution for the remaining problem provides an effective way to cull parts of the search tree early. To this end we calculate a lower bound $b(G)$ in every search tree node and discard the branch if $b(G) > k$.

We use a very simple lower bound, also described by Böcker et al. [13]: For a set of edge-disjoint conflict triples CT, $\sum_{vuw \in CT} \min\{s(uv), s(uw), -s(vw)\}$ is a lower bound on the cost of the whole graph. Calculating an *optimal* CT, one that maximizes this sum, is expensive, but just greedily constructing a set still produces

a reasonable bound (in practice we actually maintain such a list throughout the algorithm instead of constructing one on demand, more details are in section 4.5).

As mentioned above, we also have use for $b(G, xy)$, a lower bound that entirely ignores x and y in its computation. In principle this could be a completely separate computation, but for simplicity and performance reasons, we use the same computation and set CT as for $b(G)$, except we ignore all conflict triples vuw where any of v, u, w are equal to x or y when calculating the sum.

4.5 Implementation Details

While the previous sections were largely theoretical, in this section we will describe some of the concrete implementation decisions we have made, especially with a focus on maintaining good performance: Good algorithmic approaches are necessary to tackle any NP-complete problem, but efficient implementation can also make a significant difference.

We chose Rust [21] as our implementation language. Rust is a modern language that can produce well optimized native code with performance similar to e.g. C++, while also providing memory safety, a more advanced type system, and other features that allow greater productivity.

Graph Storage Recall that we work on undirected, weighted graphs. A purely adjacency-list based approach is not practical because in our setting non-edges also have weights (that are negative). Consequently, we initially stored a graph as a simple triangular matrix of weights.

While this is a very space-efficient form, testing quickly revealed it is actually much more efficient to simply store a full rectangular matrix which stores each weight twice, at least as long as this doesn't lead to excessive memory usage. We expect the difference largely comes from cheaper reads of weights in the matrix: In the general case, reading the weight associated with two arbitrary vertices requires a branch (or at least finding which vertex index is the smaller one) to calculate the correct index into the triangular matrix, while reading from the full matrix only requires a multiplication and an addition. To an extent this can be mitigated by taking advantage of access patterns where the relative ordering of the vertices is known statically. We generally tried to utilize such patterns where possible, but even so, the full rectangular matrix approach was significantly faster. Note that paying attention to the same access patterns is still worthwhile however, as they also improve access locality and predictability when iterating with the rectangular matrix.

The matrix works well for quickly reading and writing specific weights, but, as with any adjacency-matrix, iterating over the neighbors of a vertex is linear in the total vertex count, not the neighbor count. As many operations and reductions require iteration over neighbors, we also experimented with hybrid approaches to graph storage, with both an adjacency matrix and adjacency lists. In our tests, the overhead of maintaining the adjacency lists outweighed any benefit in iteration time however. Abu-Khzam et al. [22] discuss an interesting hybrid representation specifically designed to facilitate efficient branch-and-reduce search algorithms. It

enables very fast implementation of several common operations, as well as "undo"ing them when moving on to another branch. Unfortunately, significant parts of it are not immediately applicable to weighted graphs, reducing its effectiveness in our setting. Within the bounds of this thesis we ultimately stick with the rectangular weight matrix, but this avenue is nevertheless promising if a similar representation that supports weighted graphs could be found.

We also note that we currently store all weights as floating-point numbers. This is perhaps not entirely expected as we specifically only handle integer-weighted instances and also avoid producing any non-integer weights over the course of the algorithm. We use them mainly because it makes it very convenient to deal with any infinities we use as weights. Floating point numbers can be positive or negative infinity and handling that correctly is built into the hardware. For example summing many weights together should result in $-\infty$ if any of those weights are $-\infty$. Storing integers and doing this manually is more complex in terms of writing and maintaining the code and also seems likely to be slower at runtime than the native hardware handling.

Merging vertices As discussed in section 4.2, from a theoretical perspective, merging vertices u and v can be modelled as removing both of them from the graph and instead adding a new vertex u'. In practice there are significant advantages to keeping the graph storage fixed-size. This not only lets us avoid reallocating and copying the storage, it also makes it much easier to roll back changes (which we discuss in more detail below). To this end, a graph is composed of not just a matrix, but also a boolean mask storing whether each vertex is actually present in the graph. Merging u and v then instead involves assigning the weights that were described for u' to u and marking v as not present in the graph anymore. Operations such as iterating over all vertices, or over neighbors, check the presence mask and skip vertices that are not present. Additionally, recall from section 4.2 that we keep track of a list of edits in terms of the original vertices. Thus, on every merge we modify the map from current graph vertices to original input graph vertices such that u now maps to its previous vertices as well as those of v.

Undo Over the course of the algorithm, we explore a large search tree of possible operations to find the optimal solution. At any fork point, the solver first takes one branch and searches further. If no solution can be found in that branch, it then tries the other branch. To take the second branch, we need to again have the initial state at the fork point available to continue from. A very simple way to achieve this is to simply clone the entire relevant state before taking the first branch. This is however very inefficient: Not only is cloning repeatedly expensive in terms of runtime, it also leads to a large amount of memory usage because the full set of previous states has to be kept around while descending the tree until a leaf is reached.

To avoid this, we instead track a list of operations done on the graph (setting a weight, and marking a vertex as not present), along with the information needed to undo it (the two vertices and the previous weight, and which vertex was marked respectively.) When we reach a fork point, we note the current length of this "operation log", or *oplog*, and proceed with the first branch. To later try the other

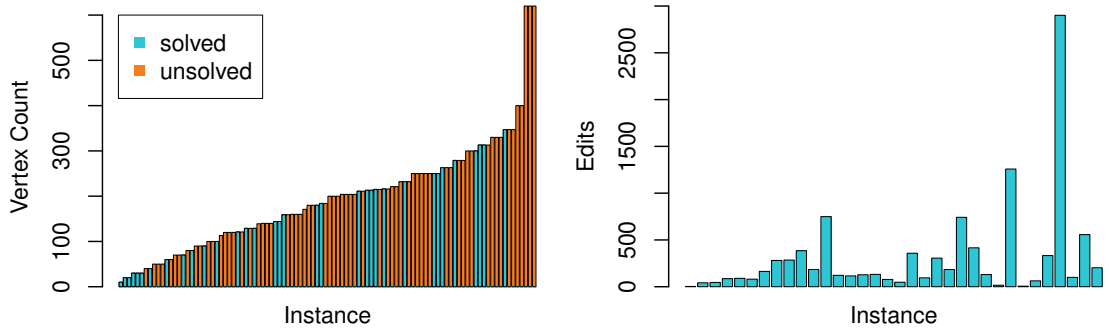
branch, we undo all operations in reverse order until the length of the oplog matches the value we recorded again.

This is much faster in terms of runtime than constantly making copies of the entire relevant state. It also helps with memory usage: While it doesn't change the asymptotic behavior (we still keep some additional state for each branch in the current stack), it does in practice reduce memory usage so much that it has been a complete non-issue in our testing on the public PACE challenge problem instances.

Conflict Tracking Over the course of the algorithm, we keep track of where conflict triples are in the graph. Whenever we insert or delete an edge in the graph, we perform a corresponding update for the conflict tracking data structure. We actually store three different sets of information: A boolean mask for all triples in the graph that is true where the triple is a conflict, a list of edge disjoint conflicts, and a mapping from edges to indices into this list. The list of edge disjoint conflicts is used to very quickly calculate a lower bound on the size of the solution, as described earlier. The mapping and the mask structures are used to efficiently perform updates for edge insertions and deletions in $O(n)$ time.

5 Experimental results

The PACE challenge provides 100 public test instances of increasing size and also maintains another 100 private instances for the eventual comparison of all submitted solvers. We tested our solver on and developed it against the 100 public instances, numbered 1, 3, ..., 199. We have so far computed optimal solutions for 34 of them overall, and can compute 31 of those within the 30 minutes per instance time limit set by the challenge. See Figure 5.1 for an overview of instance size measured in vertex count as well as which instance we have solved on the left and, for the instances we have solved, the size of an optimal solution on the right. We performed our tests on the HPC cluster of the TUHH using nodes with Intel Xeon E5-2670 and E5-2670v2 processors and running CentOS 7.9. We used a "nightly" version of rustc 1.53.0, due to our use of one not-quite-stable standard library API. Specifically for the tests on the cluster we used nightly rustc 2021-03-24. The longest tests we did ran for 7 days and resulted in the mentioned 34 instances being solved, with the last solved instance finishing after 1 day and about 16 hours.



(a) Instance Sizes (b) Solution Sizes

Figure 5.1: Public Problem Instances of the PACE Challenge

Reduction effectiveness We first present some results regarding the effectiveness of the initial reduction steps. These are comparatively easy to measure as we can simply execute the reduction on all instances and record the result. In Figure 5.2 we give an overview of how effective the critical clique reduction is for the test instances. Notably, results are very varied, with 35 instances not being reduced at all, one instance being almost solved entirely (from 250 vertices in the input to 5 vertices), and other results at various places in between. This is somewhat expected considering the critical clique approach ultimately doesn't produce any edits, it only identifies and merges groups of vertices that are already cliques. Thus, the effectiveness is entirely subject to the structure of the input graph.

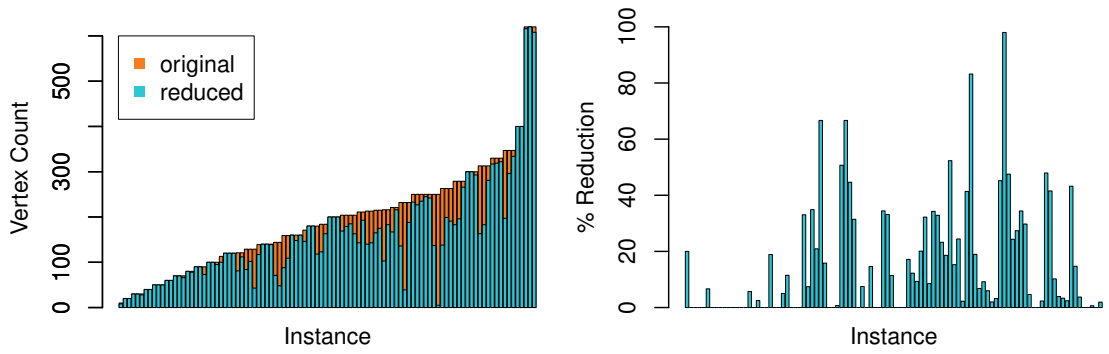


Figure 5.2: Critical Clique Reduction Effectiveness

We present the same statistics for rules 1–5 applied together and for the complete initial reduction in Figures 5.3 and 5.4. Note that rule 1 forbids edges instead of merging vertices, so it does not have a direct effect on these graphs (it may still enable other reduction steps however). We observe that many of the larger peaks occur for the same instance for both the critical clique reduction and rules 1–5. In Figure 5.4 we can see that overall reduction is in fact most effective with all reductions enabled, as expected. Considering the negligible runtime of the reductions on graphs of the size used in the challenge, it is definitely reasonable to always run the full set.

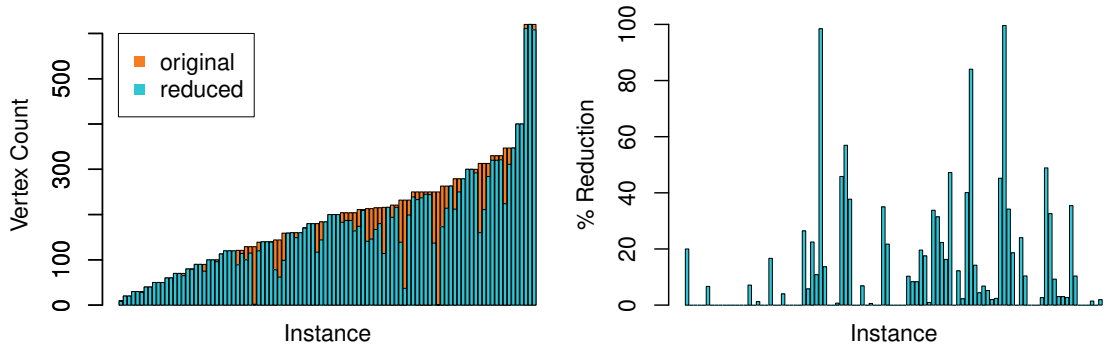


Figure 5.3: Rules 1–5 Effectiveness

An additional factor to consider is that while the effectiveness of the critical clique reduction is fully captured in the size of the reduced graphs, the same is not true for the other reductions. While most of them (with the exception of rule 1) also perform merging of vertices as their only operation on the graph, this can also result in various additional edits, hopefully making the complete solution easier to find. For an impression of how effective the reductions are in this regard, in Figure 5.5 we show what percentage of the amount of edits in an optimal solution is found by the reductions (for the 34 instances we have solved so far). Comparing the full set of reductions with only using rules 1–5, we can see that while the critical clique reduction does not produce any edits on its own, the simplified graph it produces is easier to reduce by the other rules.

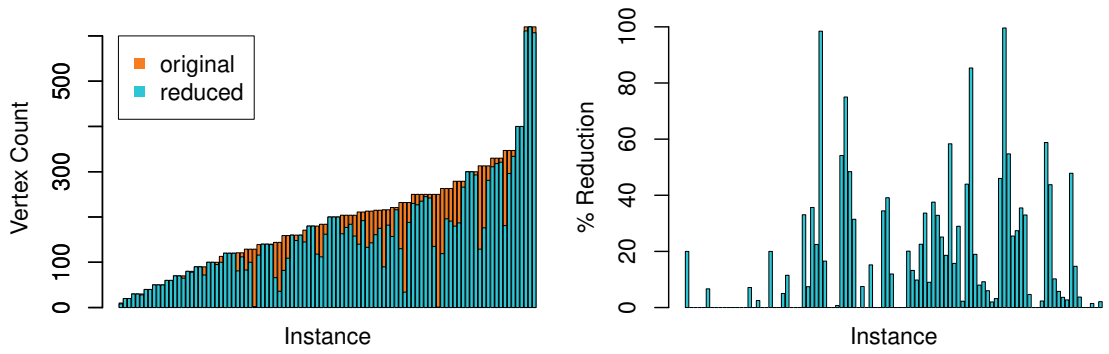
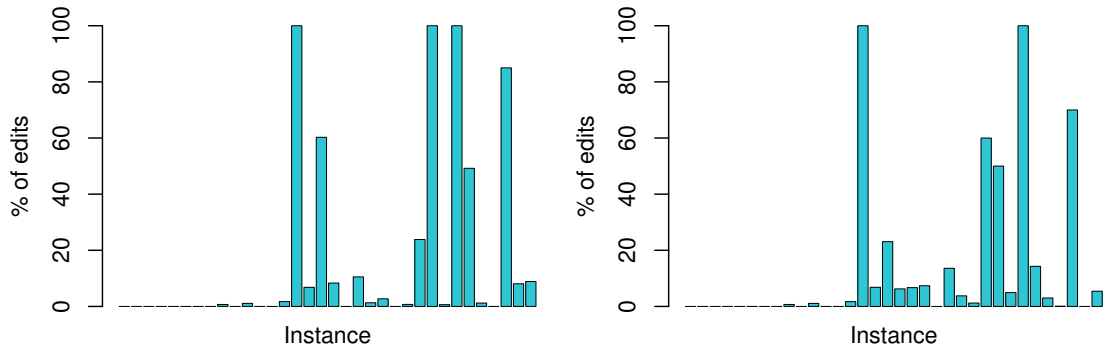


Figure 5.4: Full Initial Reduction Effectiveness



(a) Full Initial Reduction

(b) Only rules 1–5

Figure 5.5: % of edits of an optimal solution found by initial reduction steps

As described in section 4.3, we apply all reduction rules except for the critical clique reduction at points in the search tree too. It is somewhat harder to accurately measure the effectiveness of this, but we can at least get an impression of which rules are how effective. To this end, we measured how much "reduction in k" each reduction, as well as the branching decisions themselves, resulted in over the course of solving a whole instance. For every branch that is explored, whenever k is reduced due to some operation, we attribute this reduction to one of the mentioned causes. If we exit early from some branch, due to our lower bounds, the remaining parameter is attributed to "early exit". Note that this measurement is hardly perfect. For example exiting early with parameter k attributes k to early exit, but if we had explored the tree further down this path, there would have been more reduction observed in total, so the effect of early exiting is in some way underrepresented. We have also implemented a much more accurate and detailed form of collecting statistics around the effectiveness of reductions, but it is unfortunately much harder to analyze, and, more importantly, incurs so much overhead in its current implementation that it is impractical to use for anything but the smallest and easiest-to-solve instances.

With that in mind, Figure 5.6 summarizes the results of the described estimation. With our current settings, the induced cost reduction provides a very large portion of our performance. To some extent this likely just reflects our current choice of

parameters for the solver: We execute rules 1–5 only once every 200 nodes along a path in the tree, while we execute the induced cost reductions at every single node. Even on the nodes where we execute the full reduction, we first execute the induced cost reduction followed by rules 1–5. Our testing so far has shown these settings to be effective in achieving good runtime on our test instance, but it is possible a different combination could lead to still improved performance, perhaps resulting in a more balanced mix in the effectiveness of the reductions. Even with that in mind, these results do match our experience developing the solver: Implementing the induced cost reduction was one of the biggest single improvements we have observed, even though rules 1–5 were implemented first (and executed much more frequently before then).

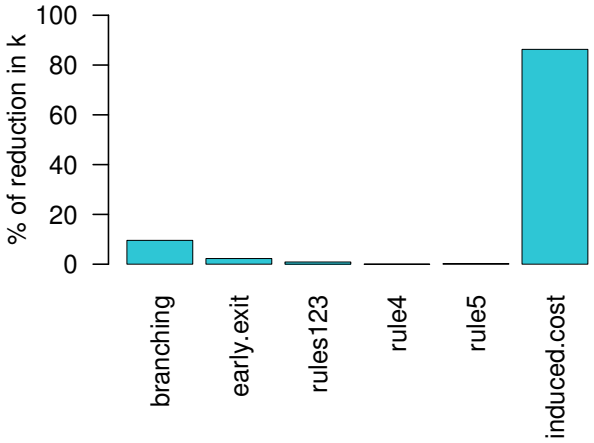
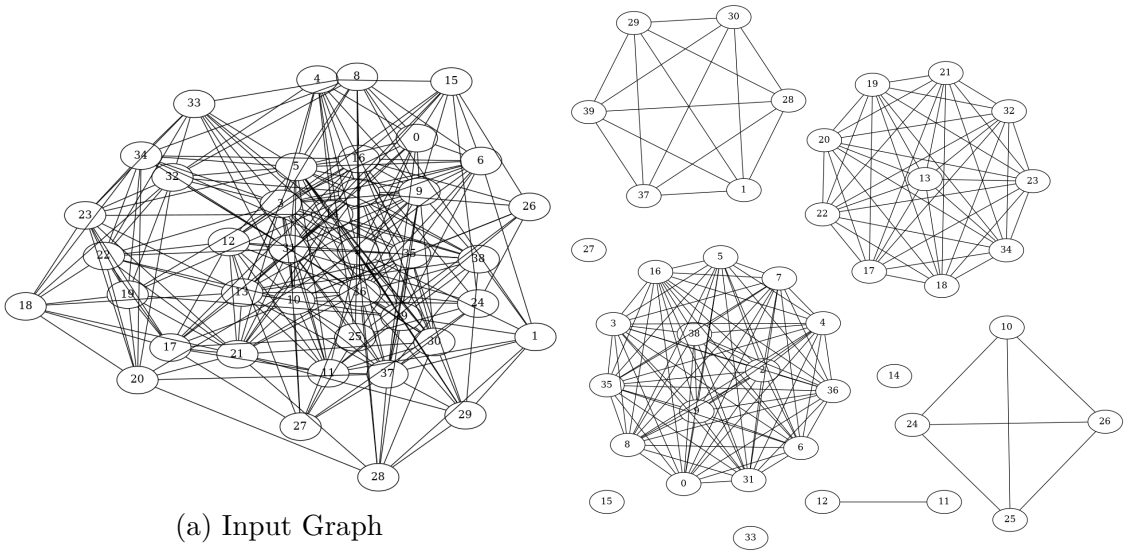


Figure 5.6: % of total reduction in k by source, averaged over all solved instances

An unsolved graph As a short example that even relatively small graphs can be hard to solve optimally, we show the "exact013" graph from the PACE public test instances in Figure 5.7. The graph has 40 vertices and 297 edges and is the smallest of the test instances which we have not yet solved. A full single clique with 40 vertices would have 780 edges, so while the graph does look fairly well connected, it is a good way off from that: such an editing would require 483 edits. On the right side of the figure we show the result of a cluster editing with 201 edits, which may however not be optimal. From our long test runs we know an optimal cluster editing has a size of at least 161 edits. At that point, with the current algorithm, it already takes longer than a day to fully exhaust the search space for a given parameter.

Unsuccessful Tests We have also experimented with some techniques that were ultimately not included in the current solver. We already mentioned above our experiments regarding the graph data structure and that a better one could potentially be advantageous but our ideas and tests so far have not been successful.

Our solver splits the input graph into its connected components and processes them individually once at the start, but does not attempt to do so again while solving. In theory it is possible to handle each component separately whenever one component is split into two while branching or during reduction, and some previous work suggests doing so. This intuitively seems like a good way to improve performance, as it means



(a) Input Graph

(b) Result of a cluster editing

Figure 5.7: Graph "exact013" and the graph resulting from *a* cluster editing, possibly not an optimal one

that after each split, reduction rules and other parts of the algorithm can work on a smaller graph, which is generally faster. In our implementation so far, we have however found it to decrease performance overall, though we cannot exclude the possibility that this is only due to inefficient implementation of the technique on our part.

We tested a weighted variant of the $k+1$ rule that Hartung and Hoos found very effective in an unweighted context [27]. The unweighted rule is that if two vertices u and v are part of $k+1$ or more conflict triples in G, it is always correct to edit uv because we cannot afford to resolve all the $k+1$ conflicts without editing uv. We tried a weighted variant that considers all conflicts involving uv and computes the minimal cost of resolving all of them without editing uv. If that cost exceeds k, we can edit uv by the same argument as above. The cost of applying this has so far also exceeded any benefit it brings, possibly because the induced cost reductions combined with a lower bound that we described earlier already catches many cases where this rule could apply.

6 Outlook & Conclusion

We plan to continue work on the solver until the submission deadline for the PACE challenge. There are still many potential options for improvement. Parameters for tweaking the behavior of the solver can be optimized. There is likely room for optimization in the existing implementation. More reduction rules can be implemented and tested, as well as the strategy for applying the current ones improved. We are also planning to try the more advanced branching strategies by Böcker et al. [15]

An additional avenue potentially worth investigating is supporting multithreading in the solver, or potentially allowing splitting computation over multiple instances of the solver to support usage on a cluster of compute nodes. In the context of the PACE challenge, the solver can only make use of one core, so we have not pursued this so far. For practical applications it could however provide a significant benefit. There are several approaches that would allow splitting up the work to be performed in parallel in a relatively straightforward manner. We also have the additional advantage of our solver being written in Rust, which can guarantee thread-safety statically at compile time, making it possible to adapt existing single-threaded code without fear of introducing hard-to-diagnose bugs.

To recap, we have implemented a solver for the NP-complete Cluster Editing problem. It is based on the parameterized complexity concepts of data reduction and bounded search trees. The solver internally works on a weighted version of the problem, because this enables operations that can reduce the search tree size significantly. We implemented and tested a variety of reduction rules to speed up the solver, and our implementation was written from the start with practical performance in mind.

The solver will be submitted to the Parameterized Algorithms and Computational Experiments Challenge 2021. Results, including comparison to other submissions, are expected in July 2021. The solver and its complete source code will be available through the PACE submission, as well as on the personal repository of the author [1] once the PACE submission is complete.

[1] https://github.com/spaarmann/cluster-editing, the state as of the writing of this thesis is tagged as 'thesis'.

Bibliography

[1] A. Ben-Dor, R. Shamir, and Z. Yakhini, "Clustering Gene Expression Patterns," Journal of Computational Biology, vol. 6, no. 3–4, pp. 281–297, Oct. 1999, doi: 10.1089/106652799318274.

[2] R. Shamir and R. Sharan, "Algorithmic Approaches to Clustering Gene Expression Data," in Current Topics in Computational Biology, 2001, pp. 269–300.

[3] R. Shamir, R. Sharan, and D. Tsur, "Cluster graph modification problems," Discrete Applied Mathematics, vol. 144, no. 1, pp. 173–182, Nov. 2004, doi: 10.1016/j.dam.2004.01.007.

[4] M. Křivánek and J. Morávek, "NP-hard problems in hierarchical-tree clustering," Acta Informatica, vol. 23, no. 3, pp. 311–323, Jun. 1986, doi: 10.1007/BF00289116.

[5] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier, "Graph-Modeled Data Clustering: Fixed-Parameter Algorithms for Clique Generation," in Algorithms and Complexity, vol. 2653, R. Petreschi, G. Persiano, and R. Silvestri, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 108–119.

[6] N. Bansal, A. Blum, and S. Chawla, "Correlation Clustering," Machine Learning, vol. 56, no. 1, pp. 89–113, Jul. 2004, doi: 10.1023/B:MACH.0000033116.57574.95.

[7] M. Charikar, V. Guruswami, and A. Wirth, "Clustering with qualitative information," Journal of Computer and System Sciences, vol. 71, no. 3, pp. 360–383, Oct. 2005, doi: 10.1016/j.jcss.2004.10.012.

[8] L. Cai, "Fixed-parameter tractability of graph modification problems for hereditary properties," Information Processing Letters, vol. 58, no. 4, pp. 171–176, May 1996, doi: 10.1016/0020-0190(96)00050-6.

[9] F. Dehne, M. A. Langston, X. Luo, S. Pitre, P. Shaw, and Y. Zhang, "The Cluster Editing Problem: Implementations and Experiments," in Parameterized and Exact Computation, Berlin, Heidelberg, 2006, pp. 13–24, doi: 10.1007/11847250_2.

[10] S. Rahmann, T. Wittkop, J. Baumbach, M. Martin, A. Truß, and S. Böcker, "Exact and Heuristic Algorithms for Weighted Cluster Editing," in Computational Systems Bioinformatics, University of California, San Diego, USA, Sep. 2007, pp. 391–401, doi: 10.1142/9781860948732_0040.

[11] S. Böcker, S. Briesemeister, Q. A. Bui, and A. Truß, "A Fixed-Parameter Approach for Weighted Cluster Editing," 2008, doi: 10.1142/9781848161092_0023.

[12] S. Böcker, S. Briesemeister, Q. B. A. Bui, and A. Truss, "Going Weighted: Parameterized Algorithms for Cluster Editing," in Combinatorial Optimization and Applications, Berlin, Heidelberg, 2008, pp. 1–12, doi: 10.1007/978-3-540-85097-7_1.

[13] S. Böcker, S. Briesemeister, and G. W. Klau, "Exact Algorithms for Cluster Editing: Evaluation and Experiments," Algorithmica, vol. 60, no. 2, pp. 316–334,

Jun. 2011, doi: 10.1007/s00453-009-9339-7.

[14] S. Böcker and P. Damaschke, "Even faster parameterized cluster deletion and cluster editing," Information Processing Letters, vol. 111, no. 14, pp. 717–721, Jul. 2011, doi: 10.1016/j.ipl.2011.05.003.

[15] S. Böcker, "A golden ratio parameterized algorithm for Cluster Editing," Journal of Discrete Algorithms, vol. 16, pp. 79–89, Oct. 2012, doi: 10.1016/j.jda.2012.04.005.

[16] P. Damaschke, "Bounded-Degree Techniques Accelerate Some Parameterized Graph Algorithms," in Parameterized and Exact Computation, vol. 5917, J. Chen and F. V. Fomin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 98–109.

[17] F. Protti, M. D. da Silva, and J. L. Szwarcfiter, "Applying Modular Decomposition to Parameterized Bicluster Editing," in Parameterized and Exact Computation, Sep. 2006, pp. 1–12, doi: 10.1007/11847250_1.

[18] M. Fellows, M. Langston, F. Rosamond, and P. Shaw, "Efficient Parameterized Preprocessing for Cluster Editing," in Fundamentals of Computation Theory, vol. 4639, E. Csuhaj-Varjú and Z. Ésik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 312–321.

[19] J. Guo, "A more effective linear kernelization for cluster editing," Theoretical Computer Science, vol. 410, no. 8, pp. 718–726, Mar. 2009, doi: 10.1016/j.tcs.2008.10.021.

[20] J. Chen and J. Meng, "A 2k kernel for the cluster editing problem," Journal of Computer and System Sciences, vol. 78, no. 1, pp. 211–220, Jan. 2012, doi: 10.1016/j.jcss.2011.04.001.

[21] "Rust Programming Language." https://www.rust-lang.org/ (accessed Apr. 20, 2021).

[22] F. N. Abu-Khzam, K. A. Jahed, and A. E. Mouawad, "A Hybrid Graph Representation for Exact Graph Algorithms," arXiv:1404.6399 [cs], Apr. 2014, Accessed: Apr. 20, 2021. [Online]. Available: http://arxiv.org/abs/1404.6399.

[23] "PACE Challenge." https://pacechallenge.org/about/ (accessed Apr. 24, 2021).

[24] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier, "Automated Generation of Search Tree Algorithms for Hard Graph Modification Problems," Algorithmica, vol. 39, no. 4, pp. 321–347, Aug. 2004, doi: 10.1007/s00453-004-1090-5.

[25] S. Böcker and J. Baumbach, "Cluster Editing," in The Nature of Computation. Logic, Algorithms, Applications, Berlin, Heidelberg, 2013, pp. 33–44, doi: 10.1007/978-3-642-39053-1_5.

[26] M. Cygan et al., Parameterized Algorithms. Springer International Publishing, 2015.

[27] S. Hartung and H. H. Hoos, "Programming by Optimisation Meets Parameterised Algorithmics: A Case Study for Cluster Editing," in Learning and Intelligent Optimization, vol. 8994, C. Dhaenens, L. Jourdan, and M.-E. Marmion, Eds. Cham: Springer International Publishing, 2015, pp. 43–58.

[28] M. Grötschel and Y. Wakabayashi, "A cutting plane algorithm for a clustering problem," Mathematical Programming, vol. 45, no. 1, pp. 59–96, Aug. 1989, doi: 10.1007/BF01589097.

[29] GitHub: ls-cwi/yoshiko. CWI - Life Sciences group, 2020, https://github.com/ls-cwi/yoshiko.