A Branch-And-Bound Algorithm for Cluster Editing

Thomas Bläsius
Karlsruhe Institute of Technology, Germany

Philipp Fischbeck
Hasso Plattner Institute, Potsdam, Germany

Lars Gottesbüren
Karlsruhe Institute of Technology, Germany

Michael Hamann
Karlsruhe Institute of Technology, Germany

Tobias Heuer
Karlsruhe Institute of Technology, Germany

Jonas Spinner
Karlsruhe Institute of Technology, Germany

Christopher Weyand
Karlsruhe Institute of Technology, Germany

Marcus Wilhelm
Karlsruhe Institute of Technology, Germany

Abstract

The cluster editing problem asks to transform a given graph into a disjoint union of cliques by inserting and deleting as few edges as possible. We describe and evaluate an exact branch-and-bound algorithm for cluster editing. For this, we introduce new reduction rules and adapt existing ones. Moreover, we generalize a known packing technique to obtain lower bounds and experimentally show that it contributes significantly to the performance of the solver. Our experiments further evaluate the effectiveness of the different reduction rules and examine the effects of structural properties of the input graph on solver performance. Our solver won the exact track of the 2021 PACE challenge.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases cluster editing

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.13

Supplementary Material
Software (Source Code): `https://github.com/kittobi1992/cluster_editing`
Software (Source Code): `https://doi.org/10.5281/zenodo.4892524`

Funding *Michael Hamann*: This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant WA 654/22-2.

Acknowledgements We thank Darren Strash for helpful discussions and literature research.

1 Introduction

In graph clustering, the goal is typically to partition the vertices into clusters such that there are many edges inside and few between clusters. The most clear-cut cases are so-called *cluster graphs* in which each connected component forms a clique. Thus, with one cluster for each connected component, there are no edges between clusters and all possible edges inside clusters exist. The *cluster editing problem* asks to use as few edge insertions and deletions as possible to transform a given graph into a cluster graph; thereby computing a clustering.

The cluster editing problem is NP-hard [18] and thus we cannot expect to solve it efficiently in general. Nonetheless there are algorithmic approaches using reduction rules [11, 12, 14] or search trees [8, 15]. The theoretically fastest known algorithm is by Böcker [7] with a running time of $O(1.62^k + n + m)$, where k is the number of edits (edge insertions plus deletions) and n, m are the number of vertices and edges of the graph, respectively. To encourage development and implementation of practical algorithms, the challenge of PACE 2021 [16] was to solve cluster editing. Our solvers won the exact [4] and heuristic [3] track.

In this paper, we describe the details of our exact solver [4] and present an in-depth evaluation. Roughly speaking our solver is a branch-and-bound algorithm: Whenever possible, we apply reduction rules to shrink the instance. When no reductions apply, we branch on the decision whether to put a pair of vertices in the same or in different clusters. To reduce the size of the resulting search tree, we compute lower bounds on the optimal solution and prune subtrees where the lower bound exceeds the upper bound computed by our heuristic solver.

Our lower bounds are based on so-called *packings* of small substructures for which we know optimal solutions. This approach has been used before to solve related problems [13] and for cluster editing in particular by packing paths of length 3 [15]. We generalize this approach to support larger substructures and weighted[1] instances; see Section 3.2. For the reduction rules, we use various rules from the literature [2, 9, 11, 12, 13, 14] as well as newly developed ones; see Section 3.3. One type of reduction rule are so-called *forced choices* that essentially look ahead one branching step, e.g., if putting two vertices in different clusters would yield a lower bound exceeding the upper bound, one must put them in the same cluster. Thus, the lower bounds and the reduction rules are intertwined in the sense that better lower bounds lead to more applications of the forced choices rules. In Section 4 we evaluate how effective and efficient different reductions and lower bounds are, using the instances from the PACE challenge. Additionally, we evaluate our algorithm on geometric inhomogeneous random graphs [10], which lets us study scaling behavior of our solver and its efficiency depending on certain instance properties. Our main findings are summarized as follows.

- The instances we can solve are usually already solved by just the reduction rules, i.e., once we have to apply branching we usually do not find a solution within reasonable time.
- The forced choices reduction rules are by far the most effective rules. This identifies good lower bounds as the key ingredient of our algorithm.
- Using packings of stars instead of paths of length 3 is still computationally feasible and yields substantially better lower bounds.
- The solver performs better on graphs with low average degree and if the graph is well-clusterable.
- The upper bounds computed by our heuristic solver are exceptionally good. In fact, the heuristic solver found an optimal solution on all instances where we know it.

2 Preliminaries

Let $G = (V, E)$ be a simple, undirected graph. The *cluster editing problem* asks to transform G into a disjoint union of cliques with the least number of edge edit operations. An edit is the deletion of an existing edge or the insertion of a missing edge. As a graph is a disjoint union of cliques if and only if it does not contain an induced path on three vertices (a P_3), the problem can also be seen as P_3-free editing.

The *weighted cluster editing problem* replaces the set of edges with a symmetric cost function $s : V \times V \mapsto \mathbb{Z}$. If $s(uv) > 0$, the pair uv is considered an edge with a deletion cost of $s(uv)$. For $s(uv) < 0$ the pair uv is a non-edge with an insertion cost of $-s(uv)$. A vertex pair with $s(uv) = 0$ is called a zero-edge that can be inserted or deleted for free. A solution to the weighted cluster editing problem is a partition of vertices. We associate a solution K with its corresponding equivalence relation $\equiv_K$. The cost of K for the instance (V, s) is defined as the total cost of edges between clusters and non-edges inside clusters. That is,

$$\mathrm{cost}(K, s) = \sum_{\substack{s(uv)>0 \\ u \not\equiv_K v}} |s(uv)| + \sum_{\substack{s(uv)<0 \\ u \equiv_K v}} |s(uv)|.$$

[1] Though the input is unweighted, our reduction rules as well as the branching lead to weighted instances.

3 A Branch-and-Bound Algorithm For Cluster Editing

Our algorithm uses branch-and-bound to solve the decision variant of cluster editing, which asks if there exists a solution with a cost of k or less. The optimization problem is solved by calling the decision variant with increasing values of k. At its core our algorithm is a simple recursive subroutine that computes a kernel by applying reductions, returns if the lower bound for the remaining instance is above k, and otherwise branches on the inclusion or exclusion of an edge in the solution, introducing *permanent* and *forbidden* edges into the instance. We select the edge to branch on by highest edit cost and tiebreak by the number of P_3s that overlap this edge. As outlined by Böcker et al. [7, 8], the endpoints of permanent edges can immediately be merged to obtain an equivalent weighted instance[2] of smaller size.

Since branching creates weighted instances, we initially apply a series of reductions that are only possible for unweighted instances before we run the recursive branch-and-bound algorithm. Furthermore, we split the initial instance into connected components and solve them separately because an optimal solution never connects them.

In the following we discuss the different parts of our algorithm. In Section 3.1, we mention our approach to obtain upper bounds. Section 3.2 introduces and generalizes the concept of lower bounds via conflict packings. Finally, we list the used reduction rules and explain their application in our algorithm in Section 3.3.

3.1 Upper Bounds

An upper bound for the optimal solution is crucial for any branch-and-bound algorithm to identify and prune branches that cannot lead to an optimal solution. For this, we use our heuristic solver that won the heuristic track of the 2021 PACE challenge [3]. The heuristic solutions are optimal on all 173 of the 200 test instances for the exact track we were able to solve (see Section 4). We refer to the solver description [3] for details about the algorithm.

3.2 Lower Bounds

For lower bounds, we use an idea from recent solvers for this and other similar problems [6, 13, 15]. The idea is to find a large set of vertex-pair disjoint P_3s. Recall that cluster editing can be seen as P_3-free editing. We call such a set a *conflict-packing* or *packing* for short. Since each P_3 in a packing needs at least one edit to resolve and no edit overlaps with more than one conflict, the size of the set is a lower bound on the required number of edits.

Finding a maximum disjoint set of conflicts is an independent set problem, which is hard to solve in general. We are not aware of complexity results for independent set on this specific kind of intersection graph. Hartung et al. [15] use the commonly known *small degree* heuristic and some random perturbation to find a maximal set of P_3s. Gottesbüren et al. [13] also use the concept of a conflict packing in their algorithm for the quasi-threshold editing problem, i.e., $\{C_4, P_4\}$-free editing. They propose to use a local search with random replacements and the 2-improvement heuristic for independent set to grow the packing [1].

With these heuristics, good P_3 packings can be found in reasonable time. However, P_3 packings have two major drawbacks. First, they are defined only for unweighted instances while the best known branch-and-bound algorithms for cluster-editing work on weighted instances [7, 8]. Second, each P_3 has two edges and one non-edge. Therefore a P_3 packing can never be larger than $|E|/2$ while many difficult instances require more than $|E|/2$ edits. We propose a framework to circumvent both these drawbacks by generalizing conflict packings.

[2] Forbidden edges have a weight of negative infinity.

We call two cost functions $a, b : V \times V \mapsto \mathbb{Z}$ *conflicting* if there is a vertex pair uv that is a non-edge in (V, a) and an edge in (V, b) or vice versa, i.e., if $|a(uv)+b(uv)| < |a(uv)|+|b(uv)|$. Let $a + b$ denote the element-wise addition of the functions a, b, that is, $(a + b)(uv) = a(uv) + b(uv)$. We call a set of cost functions P a *packing* for the instance (V, s) if (1) they are pairwise non-conflicting, (2) they are non-conflicting with s, and (3) they do not exceed s in any vertex pair, that is $\sum_{p\in P} |p(uv)| \leq |s(uv)|$. Note that $\sum_{p\in P} |p(uv)| = |\sum_{p\in P} p(uv)|$ because of property (1). Also note that property (1) actually follows from (2) and (3).

► **Theorem 1.** *For packing P of the instance (V, s), $\sum_{p\in P} opt(V, p) \leq opt(V, s)$.*

The theorem states that we can pack structures together and sum their lower bounds to obtain a lower bound for the initial instance. A P_3, for example, is represented by a cost function that is zero throughout except for its three (non-)edges. Therefore, the concept generalizes P_3 packings to weighted instances. Moreover, our formulation of a packing allows for other structures than P_3s. Recall that P_3 packings have the drawback that they cannot exceed $|E|/2$. To remedy this, we have to find other structures that have a better lower bound to edge ratio. Actually, a star S_k with k leaves (thus k edges and $\binom{k}{2}$ non-edges) cannot be solved with less than $k - 1$ edits. Coincidentally, a P_3 is a star with two leaves. One can even generalize from stars to complete bipartite graphs $K_{a,b}$ which cannot be solved in less than $a \cdot (b-1)$ edits. A star S_k is just a $K_{1,k}$. So there is a tradeoff between structures that are easy to find and pack and structures that have strong lower bounds.

We implemented a P_3 packing, a star packing, and a $K_{a,b}$ packing. In preliminary experiments, we observed that the quality of star and $K_{a,b}$ packings were similar while star packings were easier, and thus slightly quicker, to compute. We thus focus on star bounds in the following. Our implementation of the star packing builds upon the P_3 packing described by Gottesbüren et al. [13]. They go through all items in the packing and try to replace one with two currently not in the packing. To not get stuck in a local optimum, they also randomly replace an item with one other item with a small probability when it cannot be replaced by two new ones. We make three major changes. First, we introduce more mutations that change the lower bound by exactly one. For P_3s, the packing grows by removal of one P_3 and insertion of two new ones in its place. We never insert or remove stars with more than two leaves. Instead, we add the option to add/remove a leaf. Second, when possible we merge a star with another existing star instead of mutating it. The merge increases the lower bound of the packing by one. Third, we relax the termination condition for the local search. They stop if the packing does not grow for five iterations. In contrast, we continue while the average number of improving iterations is still above one in five, i.e., five times the number of improving iterations is greater or equal the number of total iterations. This leads to better packings for instances that benefit from longer local search while still being fast on instances that quickly hit a local maximum. Finally note that the packing is weighted but we only pack or modify unweighted structures. For performance, however, we associate an integer weight with each star to represent multiple identical overlapping stars.

3.3 Reduction Rules

There exist various reduction rules [2, 9, 11, 12, 13, 14] and we introduce additional ones (Forced Choices Single Merge and Clique-Like Subgraph). In the following, we discuss the reduction rules used by our solver and go into detail on how our solver applies the rules.

Twin Simple [14]. This rule merges vertices with identical neighborhoods and is part of the unweighted $4k$ kernel based on critical cliques [14]. The rule originally only works for unweighted instances. We generalize it to a pair of vertices in the weighted setting as follows.

We can merge u and v with $s(uv) \geq 0$ if their edit cost to every other vertex differs by the same constant positive factor, i.e., there exists a $c > 0$ such that $s(uw) = c \cdot s(vw)$ for every other vertex w. The correctness proof is analogous to the unweighted case and goes roughly as follows. If v being in a certain cluster produces cost X, then u being in this cluster produces cost cX. Thus, the cheapest cluster for v is also the cheapest cluster for u, though there could be multiple equally cheap clusters. In the latter case it is nonetheless still not worse to put u and v together as $s(uv) \geq 0$. We note that applying this rule repeatedly to an initially unweighted instance merges all critical cliques.

Twin Complex [9, Rule 5]. Let u, v be two nodes that are connected with an edge. The rule considers, for all possible ways to separate them into different cliques, the worst case cost of moving one into the clique of the other. If deleting the edge uv is at least as expensive as this worst case, then there is an optimal solution with u, v in the same clique and the edge can be contracted. The rule is checked with a dynamic programming (DP) approach [9].

Unfortunately, the DP degenerates when dealing with forbidden edges, i.e., edges with cost $-\infty$. In the following, we discuss why this problem exists and what we did to fix it. If u and v have a similar neighborhood, then there is no worst case where both, moving u into v's clique or vice versa, are expensive. Intuitively, the rule works because one of the two options is always cheap. Now consider a vertex w with non-edges to u and v. If another reduction finds the edge uw to be forbidden, i.e., $s(uw) = -\infty$, then two things happen to the DP. First, the solutions that put u and w in the same clique can be ignored, which is beneficial as it makes it more likely that v can be moved into the clique of u. Second, the worst case will put w and v together, which makes it impossible to move u into the cluster of v. Thus, due to the second implication, knowing that uw is forbidden can have a detrimental effect on the applicability of the reduction rule. In fact, the DP degenerates to the point that not even true twins (except for the forbidden edge to w) can be merged. To circumvent this problem, we remember the edit cost for edges that are marked forbidden throughout the whole algorithm. In the DP we then use the original weights (getting rid of the downside due to the second aspect) but still skip solutions that put forbidden node pairs in the same clique (still using the upside of knowing uw is forbidden for the first aspect).

Induced Cost Forbidden/Permanent (icf,icp) [9]. Let Δ denote the symmetric difference. The induced costs for setting a vertex pair to forbidden (icf) or permanent (icp) are

$$\begin{aligned}
\operatorname{icf}(uv) &= \sum_{w \in N(u) \cap N(v)} \min\{s(uw), s(vw)\} \\
\operatorname{icp}(uv) &= \sum_{w \in N(u) \Delta N(v)} \min\{|s(uw)|, |s(vw)|\}.
\end{aligned}$$

If the induced cost of setting a vertex pair to forbidden (icf) exceeds the current budget, then the pair must be merged. If the induced cost of setting a vertex pair to permanent (icp) exceeds the current budget, then the pair must be forbidden.

Heavy Non-Edge [9, Rule 1]. If $s(uv) < 0$ and $|s(uv)| \geq \sum_{w \in N(u)} s(uw)$, i.e., inserting the edge uv is at least as expensive as isolating u by cutting all of its edges, one can set uv to forbidden, forcing u and v to be in different clusters.

Heavy Edge, Single End [9, Rule 2]. If $s(uv) \geq \sum_{w \in V \setminus \{u,v\}} |s(uw)|$, i.e., deleting uv is at least as expensive as editing all other pairs involving u, one can merge u and v.

Heavy Edge, Both Ends [9, Rule 3]. If $s(uv) \geq \sum_{w \in N(u) \setminus \{v\}} s(uw) + \sum_{w \in N(v) \setminus \{u\}} s(vw)$, i.e., deleting uv is at least as expensive as deleting all other edges adjacent to u and v, one can merge u and v, as it is always better to let u and v form their own cluster of size 2 than to separate them.

Distance Three Rule [2]. Two vertices with distance three or more cannot be in the same cluster in an optimal solution. Therefore, all vertex pairs with distance three or more are initially marked as forbidden. This does not apply to weighted instances.

Forced Choices, all Pairs [13]. If setting an edge to forbidden or permanent would raise the lower above the upper bound, then the opposite edit must be performed. In other words, we identify an edge where, if branched on it, one branch would be pruned immediately.

A naive implementation of this rule is too slow as it requires a quadratic number of lower bound (i.e. packing) computations [13]. However all these packings are similar. Given a packing lower bound for the instance, we locally modify the packing for each vertex pair to obtain the required bounds. Because a packing changes only locally, this can be done significantly faster than computing $\binom{n}{2}$ packing lower bounds from scratch.

Forced Choices, Single Merge. Updating the lower bounds as in the previous rule is usually worse than computing the bounds from scratch. Thus, we additionally identify a constant number of edits that are unlikely to be included in the optimal solution and compute lower bounds for them from scratch. Specifically, we choose five vertex pairs and test if editing them to non-edges would be too expensive so that the rule produces a merge when applicable. We do not test for the converse because merges are far more rewarding than finding a single forbidden edge. To choose the five pairs, a heuristic estimates in advance which pairs would produce the highest cost when set to non-edges. Criteria for this heuristic are the cost of the edit as well as the number of overlapping P_3s with the vertex pair before and after the edit.

Clique-Like Subgraph. Given the instance (V, s) and the subset $C \subset V$, we define the *C-subinstance* (V, s_C) by setting $s_C(u, v) = s(u, v)$ if $u \in C$ or $v \in C$ and $s_C(u, v) = 0$ otherwise, i.e., if $u, v \in V \setminus C$. With this, we prove the following theorem.

▶ **Theorem 2.** *Let (V, s) be an instance of* WEIGHTED CLUSTER EDITING*, and let $C \subset V$ be a set of vertices. If an optimal solution for the C-subinstance isolates C into its own cluster, then there is an optimal solution for (V, s) that does so as well.*

The rule can be checked by solving the C-subinstance. We note that the instance (V, s_C) is likely easier than (V, s) due to the following observation. When looking at the graph with vertex set V with an edge between u and v if $s(u, v) > 0$, then we expect the closed neighborhood $N[C]$ of C to be rather small. Moreover, for a vertex $u \in V \setminus N[C]$ and any other vertex $v \in V$, we have $s_C(u, v) = 0$. Thus, we know that there is an optimal solution of (V, s_C) that has u as singleton, which reduces the instance to only the vertices in $N[C]$.

Reduction Order

Reductions are checked sequentially in a certain order. If one reduction was applied, the process rechecks all reductions starting with the first one. Before the loop repeats, a new lower bound is computed to check if the current branch can be pruned. We chose the order of reductions by decreasing effectiveness. The forced choices reductions are the most effective reductions and come first. Twin Complex is also very effective, but we do Twin Simple before that because it is faster and already catches some cases for Twin Complex. The remaining reductions run in $O(n^3)$ each with low constant factors so their order does not matter as much. The final order of reductions that are checked during the branching algorithm is:

1. Forced Choices, all Pairs (Star)
2. Forced Choices, all Pairs (P_3)
3. Twin Simple
4. Twin Complex
5. Induced Cost Forbidden/Permanent
6. Heavy Edge, Both Ends
7. Heavy Edge, Single End
8. Heavy Non-Edge

The initial instance is reduced differently. The Distance Three rule is applied once before the reduction loop since it is only applicable to unweighted instances. Then, the Clique-Like Subgraph reduction checks the clusters that are found by the heuristic solver. The optimal solution for the subinstances is computed with the exact solver itself. To keep the running time reasonable we skip subinstances with 50 vertices or more and run the solver with a timeout of 5 seconds. Finally, the other reductions are applied in a loop. During the loop, the order differs in three aspects from the order given in the list above. First, small connected components are brute-forced before the first item on the list. Second, Forced Choices Single Merge is added as a last reduction. Third, Force P_3 is applied before Force Star.

4 Experiments

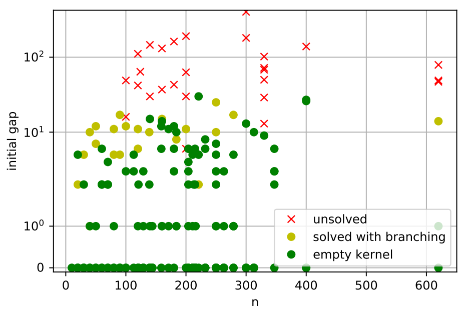
In Section 4.1, we discuss the performance of our solver, the efficiency and effectiveness of the reduction rules, and the quality of lower and upper bounds. In Section 4.2 we perform scaling experiments and determine how structural properties affect the solver performance on geometric inhomogeneous random graphs (GIRGs) [10], which are a generalization of hyperbolic random graphs [17]. As a generative network model, GIRGs can generate a series of similar instances that differ in a single property such as size, average degree, or clustering.

Setup. The experiments were run single threaded on a 4-Core Intel Xeon E5-1630v3 at 3.7GHz with 128GB DDR4 at 2133MHz. Each run has a soft timeout of one hour except for Figure 4a where it was 10 minutes per instance. Soft timeout means the current subroutine, is allowed to finish for the solver to terminate gracefully. To generate GIRGs, we use the efficient generator by Bläsius et al. [5]. The code for the experiments, raw data, execution logs, instances, as well as the plotting code can be found in a branch of our public repository[3].

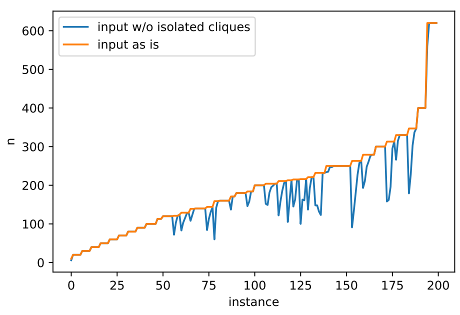
4.1 PACE Instances

In the following, we use the public and hidden instances from the 2021 PACE challenge to evaluate our solver. They represent a well balanced selection of instances from bioinformatics and data mining as well as randomly generated ones. Moreover, they are publicly available

[3] `https://github.com/kittobi1992/cluster_editing/tree/experiments`



(a) The initial gap between lower and upper bound.



(b) Vertices before/after removing isolated cliques.

Figure 1 For the 200 PACE instances, the gap between upper and initial lower bound (left) and the number of vertices per instance (right). In the left plot, the color indicates if an instance was solved by reductions only, needed branching, or remained unsolved in the given one-hour time limit.

at the PACE website[4]. We discuss the effectiveness and efficiency of individual reduction rules as well as their combination used in the solver. Then, we compare the quality of the greedy upper bound to the lower bounds obtained by the P_3 and star packing, respectively.

Solver Performance. In total, the algorithm solves 173 of the 200 instances from the PACE challenge with a timeout of one hour. Most instances finish significantly faster than that; 98 are solved in just one second and 160 finish in under a minute. Figure 1a shows for each instance if it was solved and whether reductions produce an empty kernel or branching was necessary. The axes correspond to the number of nodes and the initial gap between upper and lower bound. The gap is a good indicator of difficulty for our solver while the number of nodes seems unrelated to difficulty. All unsolved instances have a gap above 10. Surprisingly, 151 instances are solved with reductions alone. For a comparative evaluation of solver performance with other state-of-the-art algorithms, we refer to the official report of the 2021 PACE challenge [16]. On the hardware used in the actual challenge and a 30 min timeout, our algorithm solved 171 instances, while the second best submission solved 160.

Reduction Effectiveness. To evaluate the effectiveness of the reduction rules, we compute a kernel with each rule separately, i.e., apply the rule exhaustively with a soft timeout of one hour. This results in one kernel per combination of rule and instance. Before the kernel is computed we apply the Distance Three reduction rule which marks all vertex pairs in distance three or more as forbidden. Isolated cliques are removed from the input instance and once more from the final kernel. Figure 1b shows the size of the instances with and without the removal of isolated cliques. The results of the kernelization experiments can be seen in Figure 2. Each plot has a box for each reduction rule which represents the kernels made with this rule. There are two columns; the left one includes all instances while the right one only includes instances where the initial star bound does not match the upper bound. The instances with a gap between upper and lower bound represent more difficult instances for the solver thus making reductions more valuable on them. The rules *force p3* and *force star* refer to the Forced Choices, all Pairs reduction with the respective lower bound. The combination of the Induced Cost Forbidden and Permanent rules is labeled with *icx*. We

[4] `https://pacechallenge.org/`

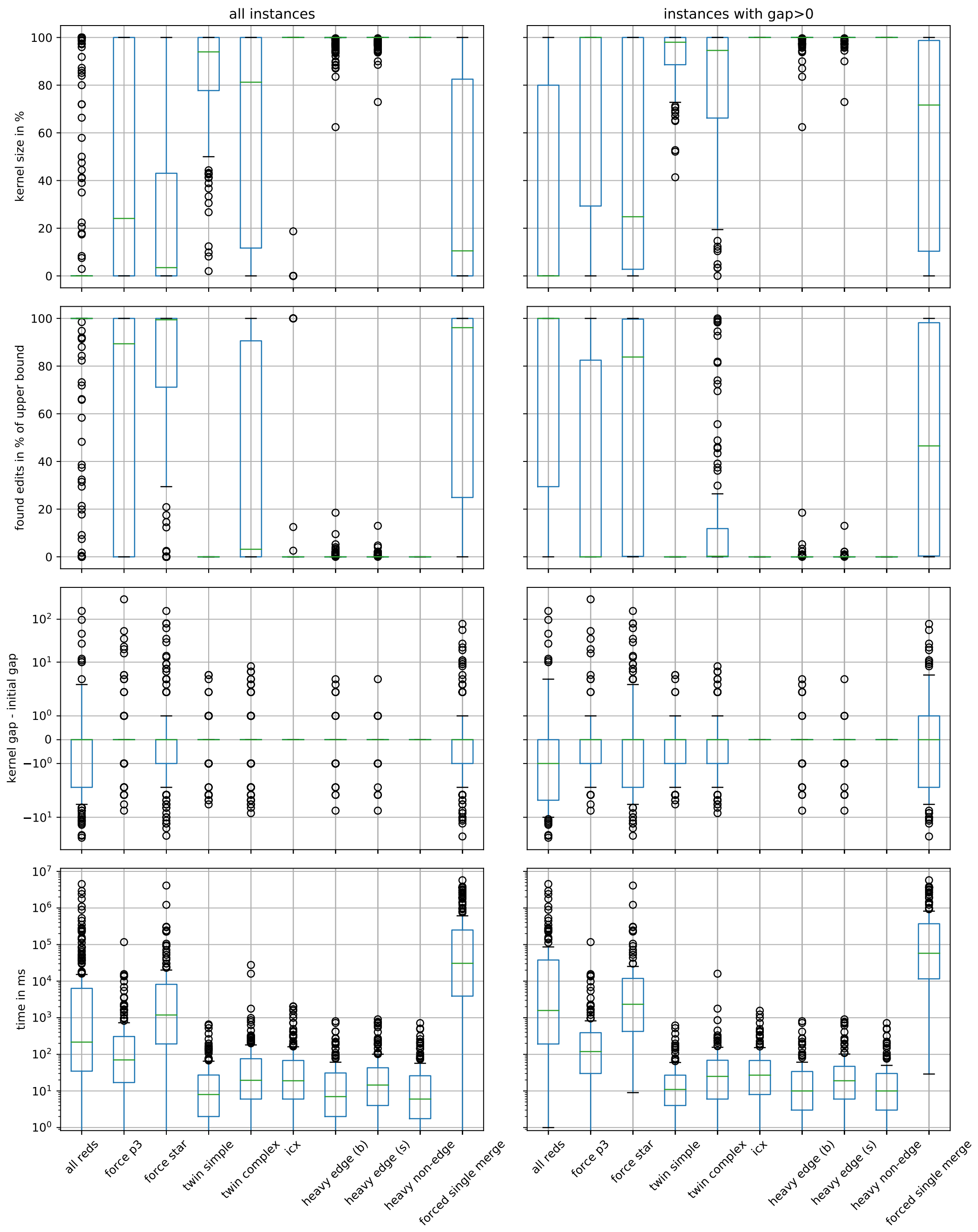


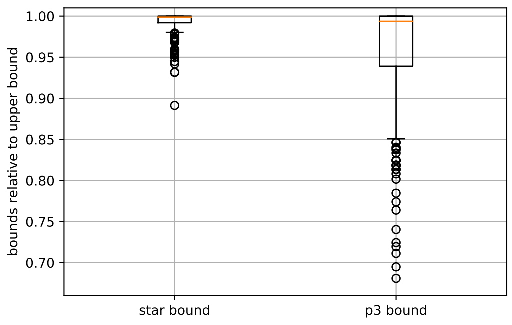
Figure 2 Kernels of PACE instances for each reduction rule. The rows show size, found edits, absolute gap change and time to compute the kernels. The left column includes all 200 instances while the right includes only the 121 instances with non-matching upper and lower bound. The label *all reds* refers to a combination of all other listed reduction rules.

also compute a kernel using all reduction rules (labeled as *all reds*) in the combination and order they are used by our solver to reduce the initial instance (see Section 3.3) excluding the brute-force of small components and the Clique-Like Subgraph reduction.

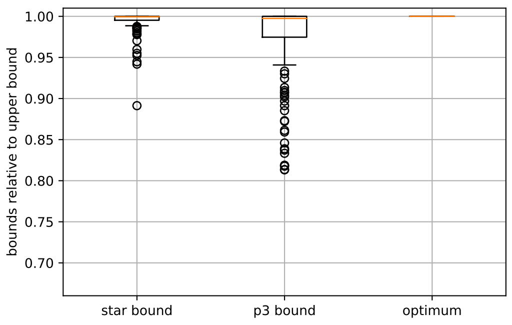
To evaluate the quality, we use three different measures. First, the number of vertices in the kernel, second, the number of edits that are already found, and third, whether the lower and upper bound get closer after computing the kernel. The number of vertices is a typical metric for kernel quality. For the second measure, the existence of an FPT algorithm for cluster editing indicates that the difficulty of the problem (the exponential part) is due to the size of the solution (the number of edits) rather than the size of the instance. In that sense, the percentage of edits that are already found during kernelization is a better indicator for progress towards a solution than instance size. The third measure, the gap between upper and lower bound, represents the difficulty of the kernel for any branch-and-bound solver using the lower bound that was used to compute the gap. With the change of the gap we estimate whether the kernel is easier or harder for our algorithm than the initial instance.

The first row shows the number of vertices in the produced kernels relative to the size of the initial instance without isolated cliques. The icx rules, both heavy edge rules and the heavy non-edge rule do not reduce the instance in the median. The heavy non-edge rule finds only forbidden edges. Therefore, the only way this rule could possibly reduce the number of vertices is by isolating a clique that is removed by our postprocessing. The simple twin and complex twin reductions are more effective with the complex twin producing smaller kernels. The non-zero gap instances prove to be harder for the twin reductions but the rules still find some application. The reductions based on forced choices produce the smallest kernels with an average size of 26% for force star, 44% for force P_3 and 36% for forced single merge. Best of all is the combination of all reduction rules that produces an empty kernel on more than 75% of instances and more than 50% of instances with a positive gap. On average, *all reds* reduces the instance to a size of 18%. While not explicitly shown, the instances with zero gap are interesting, too. The force star and the forced single merge reductions should produce an empty kernel in this case. While they indeed always apply initially, they do not always produce an empty kernel. This is because after the instance was reduced a few times it becomes weighted. Computing good packings for weighted instances becomes more difficult and might not be sufficient for the forced choices rules. In case of the forced single merge, the larger instances time out before the kernel is finished. Both these phenomenon happen rather rarely. On zero-gap instances the forced star produces a kernel size of 1.26% on average and 5.08% for forced single merge.

The second row shows the number of found edits that *must* be included in an optimal solution, i.e., the value that k is lowered by during kernelization. This value is normalized relative to the upper bound and represents another kind of progress towards solving the instance. The higher this value, the fewer choices remain to be fixed to solve the instance. Note that the values should be inverted when comparing with kernel size because for this plot 100% means solved while for instance size 0% means solved. Most reductions indicate similar results as for the kernel size. A notable exception is the simple twin rule. Since this rule only merges vertices with identical neighborhood, it never produces any cost but just reduces the instance. Interestingly, the progress made due to found edits is slightly better than the kernel size for the forced choices rules. For example, the force p3 kernel finds ca. 90% of edits in the median while the median kernel shrinks the instance by less than 80%. Since cluster editing is FPT in the number of edits, this is contrary to the expectation that the number of edits instead of the size of the instance should be responsible for the instance difficulty.



(a) Lower bounds via P_3, star packing (all instances).

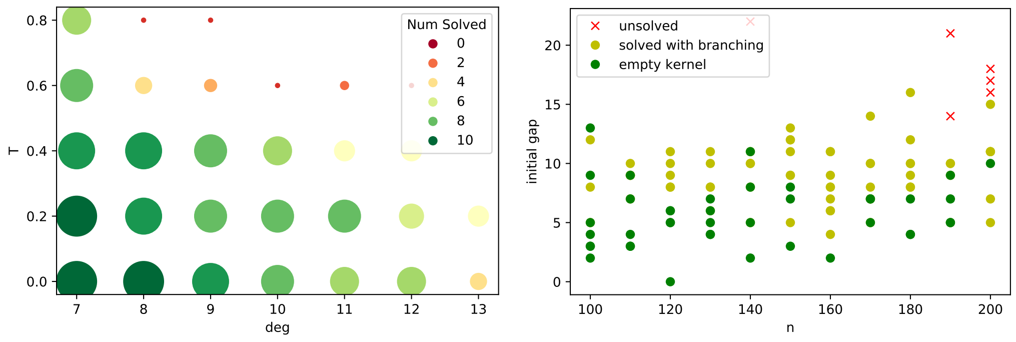


(b) Lower bounds and optimum for solved instances.

Figure 3 Packing lower bounds via P_3 and star packings on all instances (left) and on solved instances (right). The right plot additionally contains a column for the optimum. All values are relative to the respective upper bound for the instance. Note that the y-axis ranges from 0.7 to 1.0.

The third row shows the absolute change in gap after the kernel was computed, i.e., how much the difference between upper and lower bound has changed due to the kernelization. The y-axis uses symmetric log-scaling. A positive value means the gap grew and a negative value means the gap shrank. It might seem unintuitive that the gap can grow as previous lower bounds (before kernelization) still hold for the kernel. To explain this, consider two types of progress. The number of performed edits is *hard* progress, the lower bound represents *soft* progress. Once the total progress reaches the upper bound, the instance is solved. Hard progress is final but soft progress is temporary in the sense that, when actually solving the remaining instance, each reduction or branching step produces a new instance for which it might be more difficult to find good lower bounds. In general, there is no clear tendency for any reduction rule to only grow or only shrink the gap. The variance is very high to both sides. Thus, the inaccessibility of the kernel for soft progress sometimes outweighs the hard progress. In other cases it is the other way round or soft progress is even easier to achieve on the kernel. This is, e.g., the case for the simple twin rule, which makes no hard progress at all but has outliers to both sides. The median gap change is zero when looking at all instances. This is not surprising since 79 of the 200 instances already start with a gap of zero. For instances with a non-zero initial gap (the right column), the combination of all reductions actually reduces the gap by one for the median instance.

Reduction Efficiency. To evaluate the kernels by performance, we measure the time it takes to compute them. The last row of Figure 2 shows the results. The y-axis is logarithmic. Note that the highest outliers are approximately at $3.6 \cdot 10^6$ms which equals the soft timeout of one hour. All but the forced choices rules have a comparable run time with less than 100ms for more than 75% of the instances. Of these rules, just the twin complex and the icx rule have outliers over 1s and are in general the slowest of the non-forced choices rules. Note that the icx rule can be exhaustively applied in time $O(n^3)$ which is the same time one execution of the rule takes [8]. We instead apply the rule repeatedly since its run time is dominated by the forced choices rules. In the median, force p3 is slightly below 100ms, force star takes just above 1s, and forced single merge approximately one minute. All reductions combined are faster than force star but slower than force p3. Since all reductions produce by far the best kernels, this speaks for the order in which they are applied.



(a) Number of solved instances by T and degree. **(b)** The initial gap between lower and upper bound.

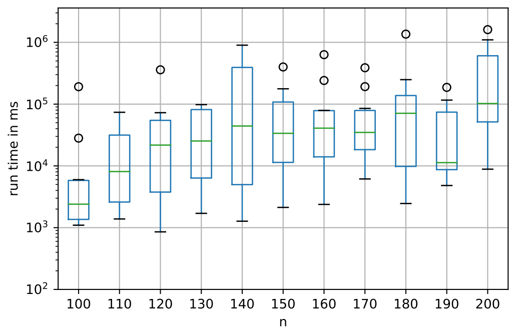
Figure 4 Solved instances by degree and T (left) and the initial gap on growing GIRGs (right). The colors in the right plot indicate if an instance was solved by reductions only, needed branching, or remained unsolved in the given time limit. The left plot maps color and size to solved instances.

Bound Quality. Figure 3 compares the P_3 bound, the star bound, and the optimum solution. The values are given relative to the upper bound computed for the instance. Figure 3a aggregates this over all instances while Figure 3b contains only solved instances and additionally shows the optimum solution. Note that the y-axis begins at 0.7 which means that even the worst outlier is already fairly good. The first observation is that the optimum is at 1.0 relative to the upper bound with no variance between instances. In fact, the upper bound from our heuristic solver matches the optimum on all 173 instance we can solve. Therefore, the lower bounds can be considered relative to the optimum; at least for the right plot. In total, the star bound is significantly better than the P_3 bound with all but one instances having a bound at more than 90% of the upper bound. The orange line for the median is only slightly lower for P_3 compared to the star bound but in this context this is huge. The number of edits in an optimal solution is approximately 1800 on average and goes as high as 27000. In contrast to this, we did not solve any instance with an initial gap of more than 30 (see Figure 1a), which is less than 2% of the 1800 edits needed on average. The average gap over all instances is 123 for P_3 and 15 for the star bound. Of course the average is heavily biased by the huge number of edits for the larger instances. Nevertheless, the 75th percentile ordered by absolute gap represents a gap of 43 for P_3 and 9 for the star bound, which makes the difference between solvable and unsolvable in this context.

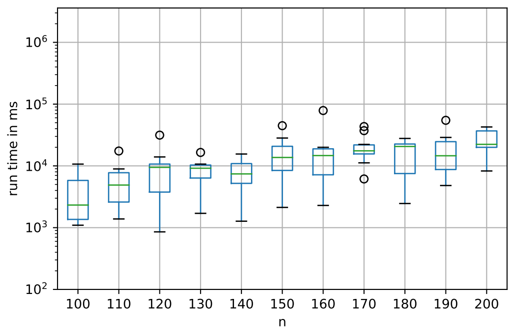
4.2 Scaling Experiments

We use geometric inhomogeneous random graphs [10] to benchmark the solver for a growing number of vertices, temperature, and average degree. Unless noted otherwise, the number of vertices is 150, the average degree is ten, the power-law exponent describing the degree distribution is 2.9, the temperature parameter, which controls the degree of clustering, is zero (meaning high clustering) and the dimension of the ground space torus is two.

Temperature and Average Degree. Figure 4a shows the effect of clustering and average degree on solver performance. For each combination of temperature and average degree, the plot shows how many of ten instances were solved in less than ten minutes. There is a clear threshold behavior that instances with both, high temperature and high average degree, are rarely solved. High average degree (13) and low temperature (0.0) is manageable with four



(a) Run time in milliseconds on solved instances.



(b) Run time in milliseconds for the initial kernel.

Figure 5 The run time of the solver in total (left) and to compute the initial kernel (right).

of ten instances solved; high temperature (0.8) and low average degree (7) even more so with seven of ten instances solved. In contrast, the algorithm solved only 4 of the 80 instances with temperature at least 0.6 and average degree at least 10.

Graph Size. Figure 4b gives an overview of which instances could be solved with or without branching. The axes are the size of the graph and the initial gap between lower and upper bound. As expected, the instances that could not be solved have a higher gap. Compared to the PACE instances, fewer can be solved without branching and only one has matching initial upper and lower bounds. Nevertheless, 41 of these 110 instances are solved by reductions alone. An instance with only 140 vertices was not solved and has a substantially higher gap than the others of the same size. GIRGs with the same configuration can vary greatly in difficulty for our solver. Two instances with 190 nodes and three with 200 nodes are unsolved.

Figure 5a and 5b show the total run time of the solver and the time spent with initial reductions, respectively. Although the run time of the solver differs up to three orders of magnitude between instances of the same size, the median time to solve an instance grows from 100 to 140 vertices. After that, the growth becomes less pronounced such that the variance makes it hard to estimate a clear trend. Also the last two columns are biased because of the instances that timed out. We explain the high variance by the comparatively small range of n that can reliably be solved and the low number of samples. Thus, in the range from 150 to 200 vertices the random sampling of position and degree distribution affects the difficulty of the GIRG for our solver more than the size of the graph. Nevertheless, the time to compute the initial kernel grows steadily with increasing number of nodes (see Figure 5b).

5 Conclusion

We present an exact branch-and-bound algorithm for the cluster editing problem. Moreover, we propose new reduction rules as well as formalize an improved technique to obtain lower bounds via subgraph packings, which contributes significantly to the success of the solver. We evaluate the lower bounds as well as various reductions rules on the instances of the 2021 PACE challenge. The lower bounds match the optimum on 79 of the 173 instances we were able to solve. For the reduction rules, by far the most effective ones are the rules that depend on lower bounds to identify forced choices, i.e., edge pairs that must or must not be edited in any optimal solution. They produce kernels with a small number of vertices and reduce k (the number of allowed edits) to an even greater extent. Combining all reductions used by the solver produces an empty kernel on more than 75% of all instances. We also investigate the effect of size, clustering, and density on our algorithm in a scale-free network model. While the size of the graph has a small effect on performance, the combination of high density and low clustering produces remarkably hard instances.

References

1 Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. *Journal of Heuristics*, 18(4):525–547, 2012. doi:10.1007/s10732-012-9196-4.

2 Lucas Bastos, Luiz Satoru Ochi, Fábio Protti, Anand Subramanian, Ivan César Martins, and Rian Gabriel S Pinheiro. Efficient algorithms for cluster editing. *Journal of Combinatorial Optimization*, 31(1):347–371, 2016. doi:10.1007/s10878-014-9756-7.

3 Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. PACE solver description: KaPoCE: A heuristic cluster editing algorithm. In *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)*, pages 31:1–31:4, 2021. doi:10.4230/LIPIcs.IPEC.2021.31.

4 Thomas Bläsius, Philipp Fischbeck, Lars Gottesbüren, Michael Hamann, Tobias Heuer, Jonas Spinner, Christopher Weyand, and Marcus Wilhelm. PACE solver description: The KaPoCE exact cluster editing algorithm. In *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 27:1–27:3, 2021. doi:10.4230/LIPIcs.IPEC.2021.27.

5 Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. In *27th Annual European Symposium on Algorithms (ESA 2019)*, pages 21:1–21:14, 2019. doi:10.4230/LIPIcs.ESA.2019.21.

6 Thomas Bläsius, Tobias Friedrich, David Stangl, and Christopher Weyand. An efficient branch-and-bound solver for hitting set. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 209–220, 2022. doi:10.1137/1.9781611977042.17.

7 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.

8 Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. Going weighted: Parameterized algorithms for cluster editing. *Theoretical Computer Science*, 410(52):5467–5480, 2009. doi:10.1016/j.tcs.2009.05.006.

9 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.

10 Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *Theoretical Computer Science*, 760:35–54, 2019. doi:10.1016/j.tcs.2018.08.014.

11 Yixin Cao and Jianer Chen. Cluster editing: Kernelization based on edge cuts. *Algorithmica*, 64(1):152–169, 2012. doi:10.1007/s00453-011-9595-1.

12 Jianer Chen and Jie Meng. A 2k kernel for the cluster editing problem. *Journal of Computer and System Sciences*, 78(1):211–220, 2012. doi:10.1016/j.jcss.2011.04.001.

13 Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. Engineering Exact Quasi-Threshold Editing. In *18th International Symposium on Experimental Algorithms (SEA 2020)*, volume 160, pages 10:1–10:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SEA.2020.10.

14 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.

15 Sepp Hartung and Holger H. Hoos. Programming by optimisation meets parameterised algorithmics: A case study for cluster editing. In *Learning and Intelligent Optimization*, pages 43–58. Springer, 2015. doi:10.1007/978-3-319-19084-6_5.

16 Leon Kellerhals, Tomohiro Koana, André Nichterlein, and Philipp Zschoche. The PACE 2021 Parameterized Algorithms and Computational Experiments challenge: Cluster editing. In *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)*, pages 26:1–26:18, 2021. doi:10.4230/LIPIcs.IPEC.2021.26.

17 Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3), 2010. doi:10.1103/physreve.82.036106.

18 Mirko Křivánek and Jaroslav Morávek. NP-hard problems in hierarchical-tree clustering. *Acta informatica*, 23(3):311–323, 1986. doi:10.1007/BF00289116.

A Missing Proofs

▶ **Lemma 3.** *Let a, b, c be three cost functions. If they are pairwise non-conflicting, then $(a + b)$ and c are non-conflicting.*

Proof. Let uv be a vertex pair with $c(uv) > 0$. If $c(uv) > 0$, then $a(uv) \geq 0$ and $b(uv) \geq 0$ since both are non-conflicting with c. Thus $(a+b)(uv) = a(uv)+b(uv) \geq 0$ which means that $(a+b)$ is non-conflicting with c on this vertex pair. The case for $c(uv) < 0$ works analogously. The case where $c(uv) = 0$ cannot lead to a conflict for c with any other function. ◀

▶ **Lemma 4.** *Let a, b be two non-conflicting cost functions. If for all vertex pairs uv, $|a(uv)| \leq |b(uv)|$, then* $\text{opt}(V, a) \leq \text{opt}(V, b)$.

Proof. Let K be a solution for (V, a). Lemma 4 follows from the fact that all vertex pairs that are edited in $\text{cost}(K, a)$ are present in $\text{cost}(K, b)$ with greater or equal absolute value. ◀

▶ **Lemma 5.** *For non-conflicting cost functions a, b,* $\text{opt}(V, a) + \text{opt}(V, b) \leq \text{opt}(V, a + b)$.

Proof. Let K be an optimal solution of $(V, a + b)$. Since a, b and $a + b$ are all non-conflicting (due to Lemma 3), there is never a non-edge in one instance that is an edge in the other. Thus we get,

$$\begin{aligned}
\text{opt}(V, a) + \text{opt}(V, b) &\leq \text{cost}(K, a) + \text{cost}(K, b) \\
&= \sum_{\substack{a(uv)<0 \\ u\equiv_K v}} |a(uv)| + \sum_{\substack{a(uv)>0 \\ u\not\equiv_K v}} |a(uv)| + \sum_{\substack{b(uv)<0 \\ u\equiv_K v}} |b(uv)| + \sum_{\substack{b(uv)>0 \\ u\not\equiv_K v}} |b(uv)| \\
&= \sum_{\substack{(a+b)(uv)<0 \\ u\equiv_K v}} |(a + b)(uv)| + \sum_{\substack{(a+b)(uv)>0 \\ u\not\equiv_K v}} |(a + b)(uv)| \\
&= \text{cost}(K, a + b) = \text{opt}(V, a + b).
\end{aligned}$$ ◀

▶ **Theorem 1.** *For packing P of the instance (V, s), $\sum_{p\in P} opt(V, p) \leq opt(V, s)$.*

Proof. Let $c : V \times V \mapsto \mathbb{Z}$ be the element-wise addition of all functions in P which is non-conflicting with s by Lemma 3. Moreover, Lemma 5 implies that $\sum_{p\in P} \text{opt}(V, p) \leq \text{opt}(V, c)$. Since P is a packing for (V, s) the third property of packings states that for all vertex pairs uv: $|c(uv)| \leq |s(uv)|$. Therefore, Lemma 4 results in $\text{opt}(V, c) \leq \text{opt}(V, s)$. ◀

▶ **Theorem 2.** *Let (V, s) be an instance of* Weighted Cluster Editing, *and let $C \subset V$ be a set of vertices. If an optimal solution for the C-subinstance isolates C into its own cluster, then there is an optimal solution for (V, s) that does so as well.*

Proof. Let $\mathcal{P}$ be any solution of (V, s). We construct a new partition $\mathcal{P}^\star$ that isolates C such that $\operatorname{cost}(\mathcal{P}^\star, s) \leq \operatorname{cost}(\mathcal{P}, s)$. For every cluster $A \in \mathcal{P}$, the partition $\mathcal{P}^\star$ contains $A \setminus C$. Additionally $\mathcal{P}^\star$ contains C.

For a pair $u, v \in V$, we say that $\mathcal{P}$ *splits* u and v if u and v are in different clusters of $\mathcal{P}$; otherwise $\mathcal{P}$ *joins* u and v. Similarly, for $C \subseteq V$, we say that $\mathcal{P}$ *splits* C if $\mathcal{P}$ splits at least one pair of vertices in C. Otherwise, if C is contained in a cluster of $\mathcal{P}$, then $\mathcal{P}$ *joins* C. We regularly need to sum over only negative or only positive cost. To simplify notation in these cases, let $s^+, s^- : V \times V \to \mathbb{N}$ be defined as $s^+(u, v) = \max(0, s(u, v))$ and $s^-(u, v) = \max(0, -s(u, v))$. The cost of the solution $\mathcal{P}$ is then defined as

$$\operatorname{cost}(\mathcal{P}, s) = \sum_{\substack{u,v \in V \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) + \sum_{\substack{u,v \in V \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v).$$

For the subset $C \subset V$ and its complement $T = V \setminus C$ it will be useful to split the sum in $\operatorname{cost}(\mathcal{P}, s)$ by vertex pairs within C, pairs between C and T, and pairs within T. We have

$$\begin{aligned} \operatorname{cost}(\mathcal{P}, s) = &\sum_{\substack{u,v \in C \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \ + \sum_{\substack{u,v \in C \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \ + \\ &\sum_{\substack{(u,v) \in C \times T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \ + \sum_{\substack{(u,v) \in C \times T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \ + \\ &\sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \ + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v). \end{aligned} \tag{1}$$

Now we can bound the cost of $\mathcal{P}^\star$ in the original instance. Consider the six sums of $\operatorname{cost}(\mathcal{P}^\star, s)$ as in Equation (1). The first sum evaluates to 0 as $\mathcal{P}^\star$ does not split pairs in C. Similarly, the fourth sum evaluates to 0, as $\mathcal{P}^\star$ does not join vertices from C with vertices from T. For the second and third sum, we can drop the condition that $\mathcal{P}$ joins and splits u, v, respectively, as $\mathcal{P}$ joins all pairs in C and splits all pairs between C and T. Finally, $\mathcal{P}^\star$ splits a vertex pair $u, v \in T$ if and only if $\mathcal{P}$ does, i.e., we can exchange $\mathcal{P}^\star$ with $\mathcal{P}$ in the fifth and sixth sum. Thus, writing the remaining sums in order $5, 6, 2, 3$, we obtain

$$\begin{aligned} \operatorname{cost}(\mathcal{P}^\star, s) &= \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \ + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \ + \sum_{u,v \in C} s^-(u, v) \ + \sum_{(u,v) \in C \times T} s^+(u, v) \\ &= \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \ + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \ + \ \operatorname{opt}(V, s_C) \\ &\leq \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} s^+(u, v) \ + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} s^-(u, v) \ + \ \operatorname{cost}(\mathcal{P}, s_C) \\ &= \operatorname{cost}(\mathcal{P}, s). \end{aligned}$$

The first equality follows from the premise of the theorem that there is an optimal solution for the C-subinstance that isolates C. Any solution for the subinstance that isolates C has to pay for all non-edges in C as well as all *edges* from C to $V \setminus C$. Moreover the solution that keeps all other vertices as singletons incurs no additional cost beyond this. Therefore the premise can alternatively be stated as

$$\sum_{u,v \in C} s^-(u, v) \ + \sum_{(u,v) \in C \times T} s^+(u, v) = \operatorname{opt}(V, s_C). \tag{2}$$

For the inequality, we have $\text{opt}(V, s_C) \leq \text{cost}(\mathcal{P}, s_C)$ because $\mathcal{P}$ is also a solution for (V, s_C). Regarding the last equality, note that $\text{cost}(\mathcal{P}, s_C)$ coincides with $\text{cost}(\mathcal{P}, s)$ except that the last two terms of Equation (1) evaluate to 0 for $\text{cost}_S(\mathcal{P}, s_C)$, i.e.,

$$\text{cost}(\mathcal{P}, s) = \text{cost}(\mathcal{P}, s_C) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ splits } u,v}} \text{cost}^+(u,v) \; + \sum_{\substack{u,v \in T \\ \mathcal{P} \text{ joins } u,v}} \text{cost}^-(u,v).$$

In conclusion, we obtain $\text{cost}(\mathcal{P}^\star) \leq \text{cost}(\mathcal{P})$, which proves the claim. ◀

B Unused Reductions

There are other reduction rules in the literature, which did not make it into our solver for different reasons, which we briefly discuss in the following.

The Clique-Like Subgraph rule is similar to Rule 4 by Böcker et al. [9], which is based on min-cuts. Since the min-cut rule can be computed more efficiently it could be useful in a solver, but we found it to be ineffective during preliminary testing.

We implemented the reduction rules by Cao and Chen [11] leading to a kernel of size $2k$, which is the smallest known kernel for cluster editing. However, our preliminary experiments showed that these reduction rules were dominated by other rules. Moreover, the rules explicitly exclude zero-edges, which we can get due to edge contractions, and it is not obvious how to adapt the rules to this setting.

Böcker et al. [9] suggest the improvement to the Induced Cost Forbidden/Permanent rules to add a lower bound for the graph without u and v to the induced costs to obtain an even stronger rule. The P_3-packing bound exactly captures this idea, even if it does not explicitly compute icf / icp, because all triangles formed with uv have a combined cost of icf / icp and can all be included in the packing because they share only the vertex pair uv. Moreover, the P_3-packing bound is strictly stronger in a weighted setting because it can additionally use the residual cost of edges uw and vw for all $w \in V \setminus \{u, v\}$ after each P_3 constituting the icf / icp is removed. Although the Forced Choices All Pairs rule with P_3 packings as bound dominates this improved version of the icp/icf rules, we keep the icp/icf rules in the solver as a failsafe to detect possible errors in the involved implementation of the forced choices rule.

Finally, there are the rules used in the unweighted $4k$ and $2k$ kernels [12, 14]. We have not implemented or adapted them to a weighted setting except for our generalization of the Simple Twin rule. The kernels are based on critical cliques, i.e., a clique containing nodes with identical closed neighborhood and the Simple Twin rule merges all critical cliques when applied repeatedly. Moreover, some other rules from these kernels are captured by our implemented rules. E.g., the Induced Cost Forbidden rule dominates rule 1 from the unweighted $2k$-kernel [12]. Nevertheless, the rules could be useful in future work.

C Reproducibility

The random components such as the generation of GIRGs or the local search to find a packing bound use the Mersenne Twister algorithm of the C++ standard template library. They are seeded as to produce deterministic results. In fact, each lower bound computation uses the same seed therefore producing the same output when given identical inputs. For each set of input parameters for the GIRG model we generate 10 instances with different seeds.

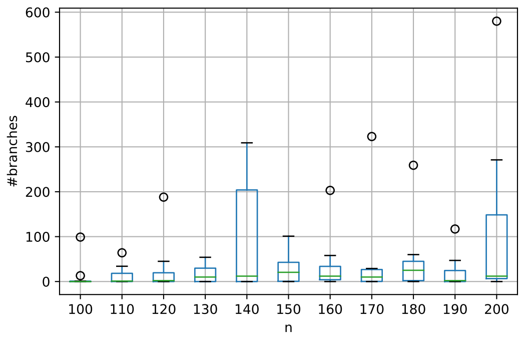
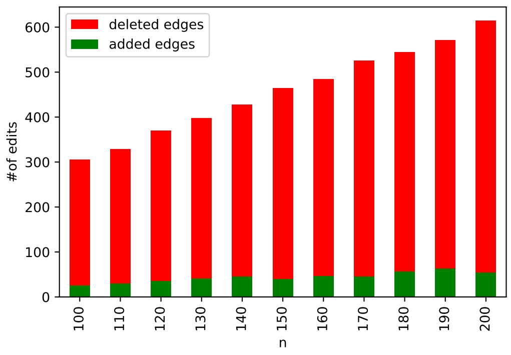
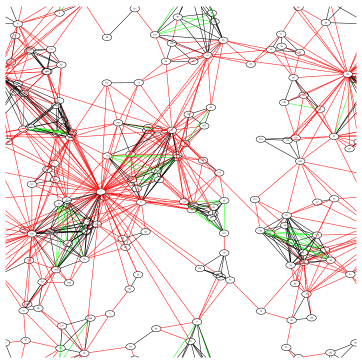


Figure 6 The number of branching decisions of the solver.



(a) Average number of edits in an optimal solution.



(b) Visualization of an optimal solution for a GIRG.

Figure 7 The number of edits in an optimal solution (left) and a possible optimal solution (right). The left plot shows the average over ten instances. The right GIRG was generated with default parameters and edits are indicated by color. Thus, the green edges are not in the generated GIRG.

D Search Space on Growing GIRGs

Figure 6 shows the number of branching decisions while solving GIRGs of different size. The plot includes only the solved instances. The results confirm that reductions contribute the most to solver performance. The number of branches is six on average. More than half of all instances are solved with less than ten branches. There is no clear trend for growing number of vertices. All graph sizes have outliers that need far more branching decisions. The instance with the highest number of branches that was still solved has 580.

E Solution Structure on GIRGs

Figure 7b shows the edits of an optimal solution for a GIRG with the default parameters listed above. Note that the positions for the vertices are sampled in a unit torus where opposite borders are identified. Due to the scale-free degree distribution, some vertices have very high degree in the input instance. Most of those edges are deleted and the high-degree node is placed in the largest clique in close proximity. There are many small cliques in the resulting cluster graph and more edges are deleted than inserted. Figure 7a confirms this observation. The plot shows the number of deleted or added edges in an optimal solution over growing graph size. Each bar represents the average over all solved instances for this size. Due to the average degree of ten, the number of edges in the input is between 500 and 1000. The number of total edits grows approximately linear in the size of the graph and deletes about half of all edges. In fact, between 44 and 64 percent of the edges are deleted which seems to be independent of graph size. The number of inserted edges is comparatively low but also grows with growing number of vertices.