Fast Vertex Merging for Cluster Editing

Lucas Holten

Supervisor: Dr Emir Demirović

Delft University of Technology

June 27, 2021

Abstract

FPT algorithms for cluster editing are relatively successful in practice, however there is still the need for a fast weighted graph data-structure that allows merging vertices and deleting edges while keeping track of a lower bound. This paper presents such a data-structure with at worst $O(n^2)$ time complexities for basic operations on graphs with n vertices. Furthermore an open-source implementation of the data-structure and an algorithm for cluster editing is provided, written in Rust for performance and readability.

1 Introduction

Cluster Editing has been studied since the 1960s. Although the problem seems very abstract it has found some applications and inspired algorithms for other clustering problems [3]. The problem itself is intuitively to turn a graph into a disconnected set of fully connected clusters by adding and removing the minimum set of edges.

There have been gradual improvements to the time complexity of fixed parameter tractable algorithms for this, parameterized by the number of modifications k. The best known algorithms for unweighted graphs has time complexity $O(1.62^k + m + n)$ [2]. And the best time complexity above a special lower bound l is $O(4^{k-l} + m + n)$ [11]. Here n and m are the number of vertices and edges.

There has also been practical research into fixed parameter algorithms for cluster editing [4, 5, 7]. In particular [5] has provided the source code for an algorithm called PEACE that makes use of the same vertex merging as the best theoretical algorithm [2].

To make algorithms based on the theory such as PEACE faster, there is a need for a fast data-structure for vertex merging and cluster editing that can keep track of a lower bound. With this paper we hope to provide this and also improve on the lower bound that was used in previous papers.

Next to that the prior approach of approaching the value of k from below by repeatedly starting a new search with a higher value is challenged. Instead the cluster editing algorithm from this paper approaches the value of k from above.

An open source implementation of the data-structure and algorithm is provided written in pure Rust code that can be used as reference for future research. Rust was chosen for this as it is a high performance language with great readability and is excellent for working on complicated algorithms.

The rest of the paper is structured as follows: First in Section 2 the formal problem description is given with explanation of some of the important concepts. Then in Section 3 the implementation is explained. In Section 4 there are subsections for all the different aspects that have been experimented with. And this section includes a comparison to the results from [9]. Then Section 5 will quickly touch on the reproducability of the experiments. And finally Section 6 will give a general conclusion of this work and provide possible directions for future research.

2 Preliminaries

This section gives a quick overview of the required background knowledge to understand this paper. Next to that it also provides definitions of all required concepts.

Cluster Editing

We will use (V, w) to denote an undirected weighted graph with vertices V and edge weight function w. Edges are defined as sets of two vertices $e \in \binom{V}{2}$ and have a weight $w(e)$.

A cluster editing solution is a partition of all vertices into clusters such that the total weight of positive edges between cluster minus the total weight of negative edges in clusters is minimized. For an example see Figure 1. Finding this partition and the cost for weighted graphs is called Weighted Cluster Editing or Correlation Clustering.

An unweighted graph (V, E) can be transformed into a weighted graph by using $w(e) = 1$ if $e \in E$ else $w(e) = -1$, where E is the set of edges in the unweighted graph. This way any Unweighted Cluster Editing problem can be transformed into a Weighted Cluster Editing problem.

Fixed Parameter Tractability

Algorithms based on FPT theory often use a recursive algorithm that decreases the fixed parameter k for every recursive call. The run time of these algorithms depends on how much k can be reduced in each recursive call. This can be easily represented by a branching vector. For example an algorithm that always has two recursive calls, where k decreases by 1 and 2 would have branching vector $(1, 2)$. From the branching vector it is possible to calculate the branching

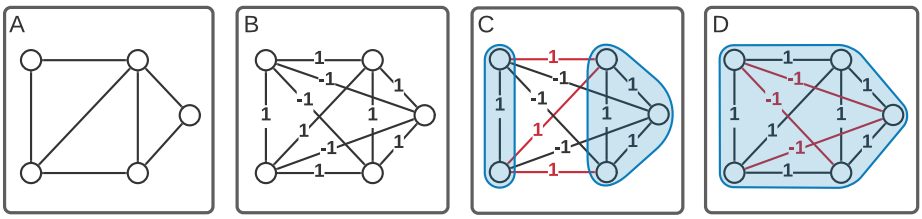

Figure 1: Examples of an unweighted graph A, the corresponding weighted graph B and two solutions to this graph C and D, both with cost 3. The clusters are shown in blue and the required edge modifications are shown in red.

number x using the corresponding polynomial $x = x^{-1} + x^{-2}$, this example gives us $x \approx 1.62$. Using this branching number and assuming every step takes constant time we can conclude that the algorithm has complexity $O(x^k)$.

Conflict Triples

A conflict triple in a weighted graph is a set of three vertices between which the edges are not transitive. This means that two of the edges have positive weight, and one edge has negative weight. If a graph contains three vertices like that it is impossible for there to be a cluster editing solution with cost zero. The opposite is not true, if there are no conflict triples it does not mean that there is a solution with cost zero. This is because zero edges can be required to be positive by one triple of vertices and negative by another triple of vertices.

Formally the set of all conflict triples is:

$$C = \{t \in \binom{V}{3} : (\exists e \in \binom{t}{2})[w(e) < 0] \wedge |\{e \in \binom{t}{2} | w(e) > 0\}| = 2\}$$

The cost of a conflict triple is equal to the smallest edge weight that is part of the triple:

$$c(t) = \min_{e \in \binom{t}{2}} |w(e)|$$

A conflict triple packing is a set of conflict triples of which the combined cost gives a lower bound on the cluster editing cost. Some examples are the *vertex disjoint packing* in which no conflict triples share a vertex. And the *edge disjoint packing* in which no conflict triples share an edge.

Kernelization

A final concept is kernelization [6]. This technique has the goal of reducing the problem size while possibly also reducing the solution cost. All forms of kernelization also allow finding the optimal solution to the original problem using the optimal solution to the reduced problem. In the context of cluster editing, the kernel size is the maximum number of vertices of the reduced graph.

For example, a kernel size of $2k$ means that $|V'| \leq 2k'$, where V' are the vertices of the reduced graph, and k' is the minimum cost of a solution to the reduced graph.

3 Algorithm

The implementation in this paper makes use of the cut and merge operations from the theory [2]. This is different from the approach used by some other implementations that only mark edges as permanent or forbidden [7]. Merging vertices has the advantage of reducing the graph size. And simple rules like adding a third permanent edge when there are two permanent edges is unnecessary. Furthermore a lot of cost can already be counted when merging vertices which improves the lower bound. The downside is that merging brings with it some complexity, because the graphs it works on have to be weighted.

Another possible difference with prior work is that this paper uses a branch and bound search instead of trying increasing values for k. This means that the algorithm starts with as upper bound the cost of a trivial solution and every time it finds a better solution the upper bound is decreased. Whenever the algorithm finds a branch for which the lower bound is equal to or greater than the upper bound it will skip that branch.

Representation

The graph is stored as an adjacency matrix, from a few tests this was faster than an adjacency list and it is also easier to implement. Of course this does mean that the algorithm requires more space, but this is not a problem for the instances tested. And we still have the same worst case space complexity of $O(v^2 + |P|)$ for the whole algorithm.

The adjacency matrix is intentionally made to store up to twice as many vertices as there are in the initial graph. This extra space is used for merged vertices such that no further allocations need to be made. For quick iteration there is an additional unsorted list that contains the indices of the active vertices.

Again, during the search no allocations are made. When cutting an edge, the weight is replaced and the original edge weight is stored on the stack. When merging two vertices, the vertex indices get removed from the active list and a new index is added that points to a new row of the adjacency matrix that gets initialized. To undo cutting an edge, the edge in the adjacency matrix is replaced by the edge on the stack. And to undo merging of two vertices, the merged vertex index is removed from the active list and the two original indices are added back. When a new optimal solution is found, the relevant part of the solution gets copied over the previous best solution.

The space requirement for the whole search is $O(v^2 + |P|)$. Of which $O(v^2)$ space to store the graph that is being edited. $O(v)$ space to store the solution. $O(v^2)$ space on the stack to store the modifications made to the graph so that

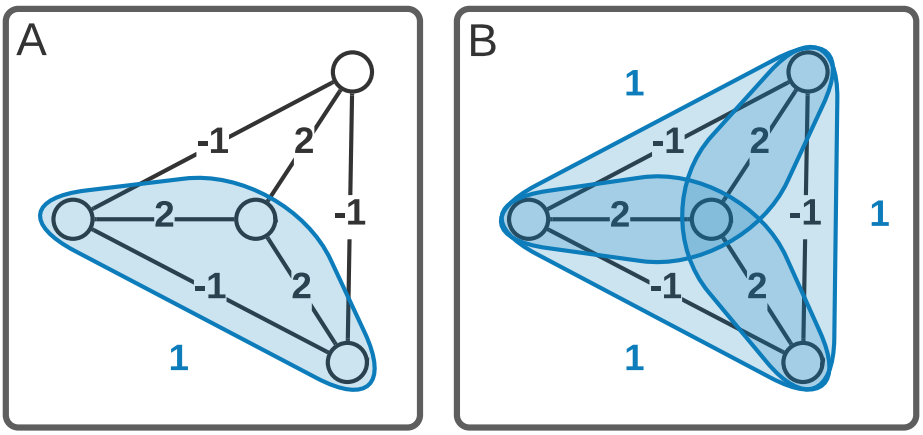

Figure 2: A: *Edge disjoint conflict triple packing* can not reuse edges and the maximum packing has a cost of one. B: *Cost disjoint conflict triple packing* can reuse the edges with weight two and the maximum packing has a cost of three.

they can be undone. And $O(|P|)$ space to store the conflict triple packing, $|P|$ is at most v^3.

Cost Disjoint Packing as Lower Bound

Edge disjoint conflict triple packing is an effective method to get a lower bound for the cost of cluster editing unweighted graphs [5]. The same lower bound can be calculate on weighted graphs by summing the cost of the conflict triples in the packing. However for weighted graphs a better lower bound is available. It is possible to generalize unweighted *edge disjoint packing* to weighted *conflict disjoint packing*, which can give a better lower bound as can be seen in Figure 2.

The idea is that an edge with weight $w(e)$ can be part of multiple conflict triples t from a packing P if the total cost of those triples $c(t)$ is less than or equal to the absolute weight of the edge:

$$(\forall_{e \in E})[|w(e)| \geq \sum_{t \in P : e \in \binom{t}{2}} c(t)]$$

The sum of the cost of the conflict triples in this packing is guaranteed to give a lower bound, because it is impossible to resolve multiple conflicts in this packing without using at least their combined cost:

$$lower = \sum_{t \in P} c(t)$$

One important observation is that the order in which conflict triples are selected can have influence on the lower bound and thus changes the performance.

To quickly know which edges have already been used and how much, there is an extra integer stored for every edge.

Deletion Triples as Upper bound

In order to get an upper bound we need to have a solution. One way to get a solution is to separate all vertices that have an edge between them with weight zero or less. We can give an upper bound on the cost of this by looking at *deletion triples*. These are very similar to conflict triples except they also include triples where two edges are positive and one edge is zero:

$$D = \{t \in \binom{V}{3} : |\{e \in \binom{t}{2} : w(e) > 0\}| = 2\}$$

The cost of a deletion triple is the minimum of the positive edges:

$$d(t) = \min_{e \in \binom{t}{2} : w(e) > 0} w(e)$$

The upper bound is then given by the sum of the cost of all deletion triples:

$$upper = \sum_{t \in D} d(t)$$

This upper bound is not proven correct, but for the rest of the paper this is not necessary. What is important is that if there are no deletion triples, the graph has been solved. This can be seen because in that case there is a trivial solution. All edges inside the clusters will have positive weight and all other edges will have zero or negative weight.

Heuristic Edge Selection

The choice of which edge to branch on is extremely important for the performance of the implementation. Two things to consider are which edge brings the lower and upper bound closest together and which edge is most likely to require an edit. In the implementation is made use of three heuristics that try to bring the upper bound down. However these heuristics also turn out to select edges that are likely to require editing.

The first heuristic selects the edge e that is part of the most deletion triples.

$$heur_1 = max_e |\{t \in D : e \in \binom{t}{2}\}|$$

The second heuristic selects the edge e that has the highest contribution to the deletion triple upper bound.

$$heur_2 = max_e \sum_{t \in D : e \in \binom{t}{2}} d(t)$$

The third heuristic selects the edge e that is part of the most deletion triples where the other two edges are expensive.

$$heur_3 = max_e \sum_{t \in D : e \in \binom{t}{2}} min_{f \in \binom{t}{2} : f \neq e} |w(f)|$$

For all these heuristics if not a single edge gets a value more than zero, that must mean that the graph has been solved. If at least one edge gets a value greater than zero, then the edge with the highest value is merged if its edge weight is less or equal to zero and cut when its weight is more than zero. This makes sense intuitively because this resolves the conflicts that the heuristic indicates.

Interesting to note is that in case the upper bound is correct, it is also possible to subtract the lower bound contribution in a packing P of each edge from the upper bound as a heuristic.

$$max_e \sum_{t \in D : e \in \binom{t}{2}} d(t) - \sum_{t \in P : e \in \binom{t}{2}} c(t)$$

However, this does not improve the performance on the tested instances. Most likely this is because removing an edge that was contributing to the lower bound does not necessarily reduce the lower bound. The cost could be moved to another edge.

Connected components

Connected components can be found with a simple flood fill in time $O(v^2)$. Because the implementation uses a list of active vertex indices it is very easy to deactivate part of the graph and solve a sub-graph. This has no performance penalty because there is no need for a copy of the graph components and the active list was already needed to deactivate merged vertices.

Kernelization and Reduction Rules

The 2k kernel from [6] was implemented and evaluated on the input graphs from PACE, but this did not reduce most of the instances. This is most likely because most PACE instances have less than 2k vertices. Also a few simple reduction rules on conflict triples from [11] did not reduce most instances. Using these reduction rules during the search was not tested.

Some other reduction rules that can be used for cluster editing:

On unweighted graphs:

1. Vertices that have at most one common neighbour do not have to be connected [10].
2. Vertices that have the same closed neighborhood have to be connected [1].

On weighted graphs:

1. If cutting out one of the vertices next to a negative edge costs less or the same as adding the edge, then the two vertices never have to be connected [3].

Interestingly, none of these reduction rules seem to have any effect on the performance of the algorithm. This is most likely because the reduction rules are somewhat implied by the heuristic edge selection. For example the rules on the unweighted graphs apply to vertices with one or zero conflicts. These will almost never be selected by the heuristic and are thus not modified in compliance with the rules.

The parameter dependent $(k+1)$ reduction rule from [7] is applied implicitly because the algorithm selects edges that are part of many conflicts. If adding or removing the selected edge brings the lower bound up to the current upper bound, then the branch is skipped in compliance with the $(k+1)$ rule.

Runtime complexity

For reference, basic operations have the following complexities, where v is the number of vertices. The time complexities for undoing these operations are the same.

- Merge two vertices: $O(v)$
- Cut edge $O(1)$
- Calculate conflict triple packing $O(v^3)$
- Update conflict triple packing $O(v^2)$
- Select edge to branch on $O(v^2)$
- Split into connected components $O(v^2)$

Notice that calculating the conflict triple packing only needs to be done at the start of the search. After that the conflict triple packing can always be kept synchronised by updating. Thus all operations that are required during the search are $O(v^2)$.

4 Experiments

In this section we will show the results of some experiments with alternative implementations and to show the significance of the different features.

The experiment in Figure 5 was executed on an *Intel Core i7-3770* and includes the non exponential parts of the algorithm like loading the graph. All other experiments were run on an *AMD Ryzen 5 5600X* and the times in those experiments only include the exponential part of the algorithm.

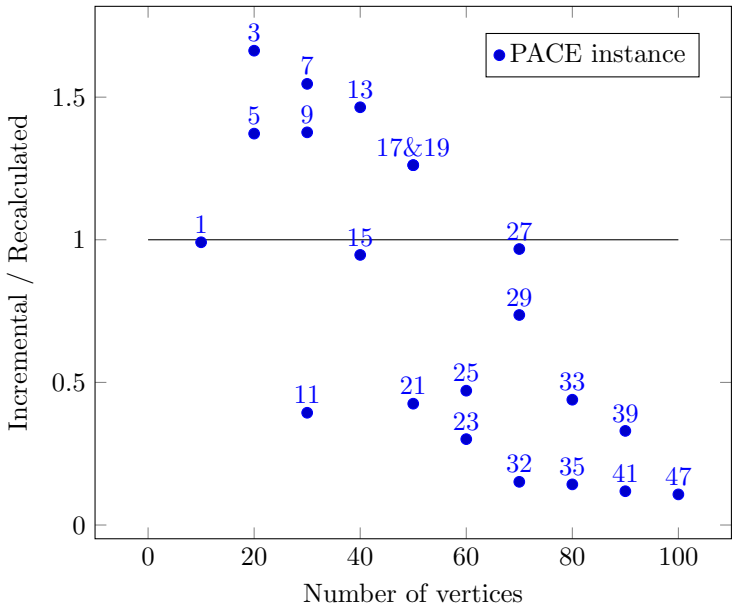

Figure 3: Algorithm run-time using incremental lower bounds relative to recalculating lower bounds.

Incremental Lower Bound

There are two implementations of the lower bound calculation. The first version just recalculates the lower bound for every branch, this takes time $O(v^3)$. And then there is the incremental version that updates the previous lower bound, which takes time $O(v^2)$. Figure 3 shows that while recalculating the lower bound is faster for instances with low vertex count, the incremental calculation is faster for larger graphs.

Another interesting observation that can be made about Figure 3 is that for a fixed number of vertices, higher instance numbers seem to benefit more from incremental lower bound updates. This is possible because PACE instances are sorted first by vertex count and then by edge count. It is not clear why instances with higher edge count would benefit more from incremental updates.

No Lower Bound

We also conducted a small test to see what the performance without lower bound was like. In that case not a single branch of the search tree is skipped, the result of this can be seen in Figure 4. Only the first six instances are shown because the algorithm without lower bound timed out for the bigger graphs.

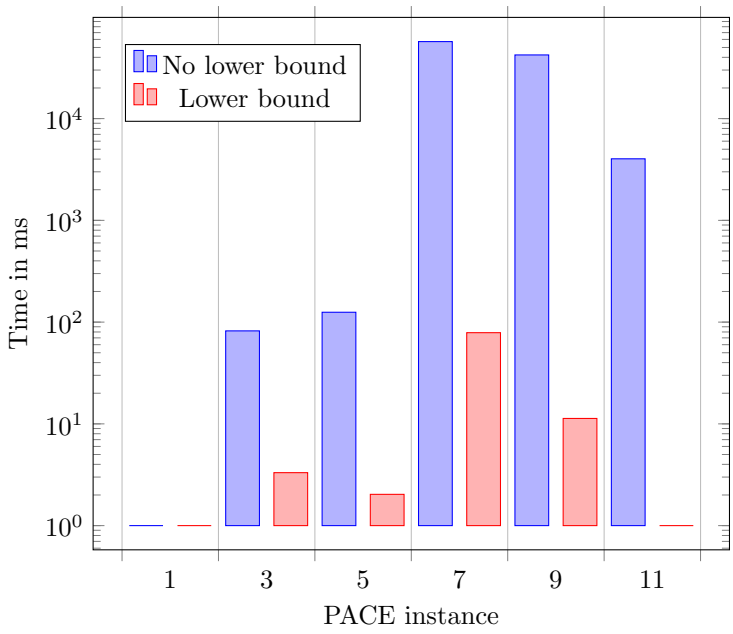

Figure 4: Performance without lower bound.

Comparison of heuristics

The three implemented heuristics were compared in performance. $heur1$ performs pretty good overall and is what is used in all the other benchmarks. $heur_2$ is slightly worse than $heur_1$ for every instance except instance 33, where it is much worse. Finally $heur_3$ is again slightly worse for most instances, but it is much better than the two other heuristics on instance 21. It also performs the best on instance 33 where $heur_2$ failed. Because all three heuristics have such similar performance, Figure 6 has been placed in the appendix.

Connected components

The implementation can split the input graph in connected components only at the start or for every branch. Splitting the graph into connected components for every branch visited does not significantly change the performance. This could mean that splitting is really fast or that the speed-up from the smaller search tree counteracts the overhead. Because the two approaches are so similar in performance, Figure 7 has been placed in the appendix.

Comparison with MaxSAT solver

The implementation from this paper was compared to a MaxSAT based solver from [9]. The results are shown in Figure 5. While the implementation from this

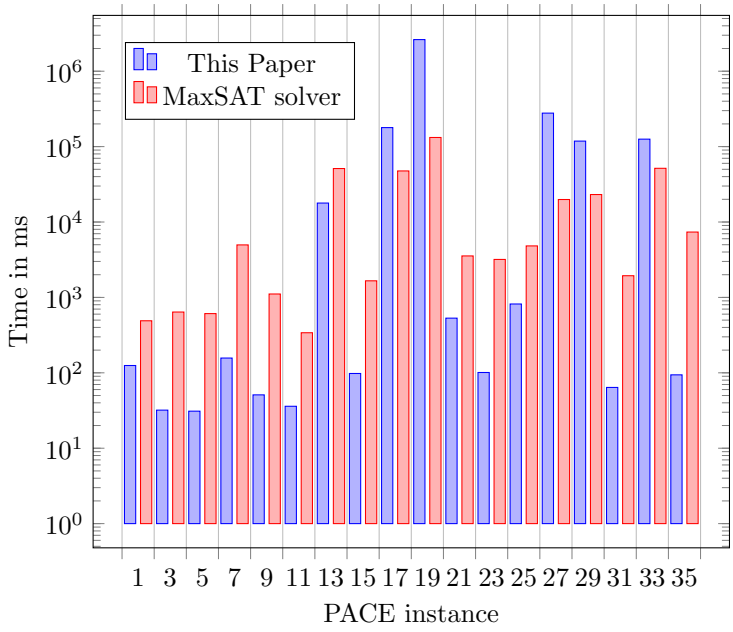

Figure 5: Performance comparison with MaxSat based solver.

paper is much faster for simple graphs. The algorithm as presented does not seem to scale well to harder instances when compared to the MaxSAT solver, which is much faster on hard instances. The MaxSAT based solver can also solve more instances, this is not shown in the graph.

The first reason for the difference in performance is most likely that the MaxSAT solver needs more time to start and thus performs worse on easy graphs. However some hard graphs like instance 13 are solved faster by the implementation from this paper. This might be because the reductions used by the MaxSAT solver are not effective on this instance and the implementation in this paper is really fast at traversing the search tree.

5 Responsible Research

This work should be fully reproducible by downloading the implementation source from GitHub[1] and the exact track public problem instances from the PACE website[2]. The source can be build with Rust 1.53.0. The MaxSAT solver that is compared to is from [9].

[1] https://github.com/LHolten/Cluster-Editing
[2] https://pacechallenge.org/2021

6 Conclusion and Future Research

The presented data-structure for vertex merging and accompanying cluster editing algorithm can be useful to implement other fast algorithms for cluster editing. This paper furthermore shows a powerful generalization of a lower bound to weighted graphs. It is shown that a lower bound is extremely important for the search. The choice of heuristic for choosing an edge to branch on is not trivial, and there are multiple heuristics that work. Further connected components can be applied during the search, but this does not seem to make a difference. Finally the source code is available on GitHub and can be used as reference material for future research.

Future research might be directed at finding the differences between the graphs and improving the heuristic or lower bound. A good candidate for the lower bound would be to use the edge deletion algorithm from [8] on a relaxed version of the graph. Another option is to try to improve the MaxSAT solver to be as fast as the implementation from this paper on all graphs.

It would also be interesting to see if it is possible to exploit the similarity between the branches that are searched in a dynamic programming fashion. This could improve the performance substantially because searching those branches repeatedly is the time consuming part of the algorithm.

A Appendix

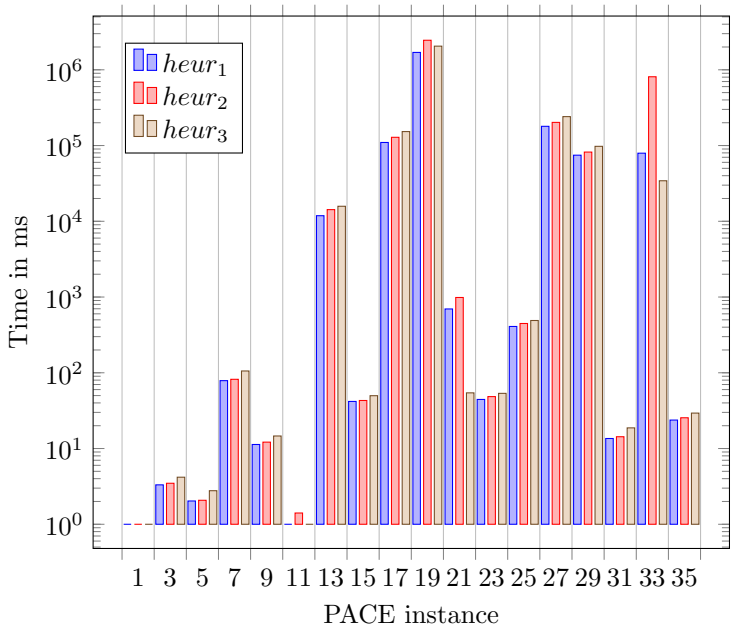

Figure 6: Performance of the three heuristics.

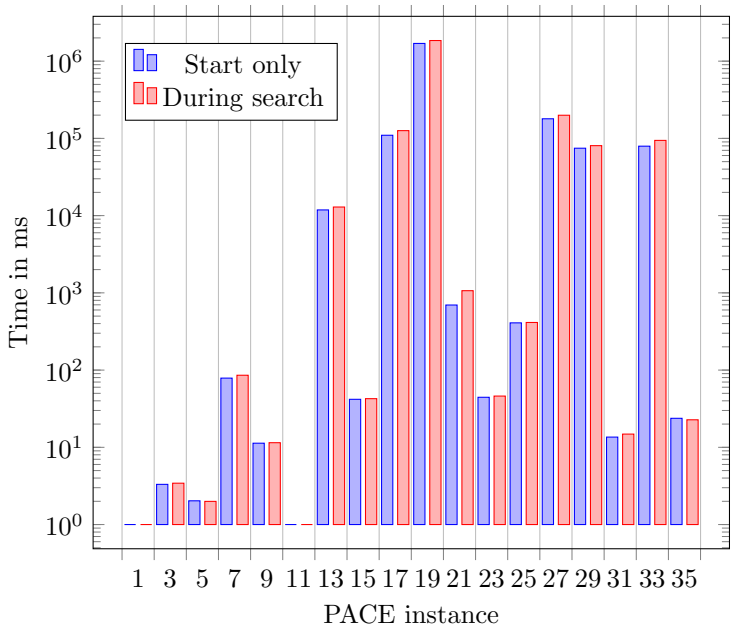

Figure 7: Performance of finding connected components only at the start or also during search.

References

[1] L. Bastos, L. S. Ochi, F. Protti, A. Subramanian, I. C. Martins, and R. G. S. Pinheiro, *Efficient algorithms for cluster editing*, Journal of Combinatorial Optimization, 31 (2016), pp. 347–371.

[2] S. Böcker, *A golden ratio parameterized algorithm for Cluster Editing*, in Journal of Discrete Algorithms, vol. 16, Elsevier, 10 2012, pp. 79–89.

[3] S. Böcker and J. Baumbach, *Cluster Editing*, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7921 LNCS, Springer, Berlin, Heidelberg, 2013, pp. 33–44.

[4] S. BÖCKER, S. BRIESEMEISTER, Q. B. A. BUI, and A. TRUB, *A FIXED-PARAMETER APPROACH FOR WEIGHTED CLUSTER EDITING*, in Proceedings of the 6th Asia-Pacific Bioinformatics Conference, PUBLISHED BY IMPERIAL COLLEGE PRESS AND DISTRIBUTED BY WORLD SCIENTIFIC PUBLISHING CO., 12 2007, pp. 211–220.

[5] S. Böcker, S. Briesemeister, and G. W. Klau, *Exact Algorithms for Cluster Editing: Evaluation and Experiments*, Algorithmica, 60 (2011), pp. 316–334.

[6] Y. Cao and J. Chen, *Cluster Editing: Kernelization Based on Edge Cuts*, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 6478 LNCS, Springer, Berlin, Heidelberg, 12 2010, pp. 60–71.

[7] S. Hartung and H. H. Hoos, *Programming by optimisation meets parameterised algorithmics: A case study for cluster editing*, in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 8994, Springer Verlag, 2015, pp. 43–58.

[8] S. Malek and W. Naanaa, *A new approximate cluster deletion algorithm for diamond-free graphs*, Journal of Combinatorial Optimization, 39 (2020), pp. 385–411.

[9] I. Marijnissen, *Solving Cluster Editing Using MaxSAT-based Techniques*, bachelor research project, Delft University of Technology, 2021.

[10] D. Rhebergen, *Cluster Editing with Diamond-free Vertices*, bachelor research project, Delft University of Technology, 2021.

[11] R. van Bevern, V. Froese, and C. Komusiewicz, *Parameterizing Edge Modification Problems Above Lower Bounds*, Theory of Computing Systems, 62 (2018), pp. 739–770.