Programming by Optimisation Meets Parameterised Algorithmics: A Case Study for Cluster Editing

Sepp Hartung[1]* and Holger H. Hoos[2]

[1]Institut für Softwaretechnik und Theoretische Informatik, TU Berlin, Berlin, Germany
sepp.hartung@tu-berlin.de
[2]Department of Computer Science, University of British Columbia, Vancouver, Canada
hoos@cs.ubc.ca

Abstract. Inspired by methods and theoretical results from parameterised algorithmics, we improve the state of the art in solving Cluster Editing, a prominent NP-hard clustering problem with applications in computational biology and beyond. In particular, we demonstrate that an extension of a certain preprocessing algorithm, called the $(k+1)$-data reduction rule in parameterised algorithmics, embedded in a sophisticated branch-&-bound algorithm, improves over the performance of existing algorithms based on Integer Linear Programming (ILP) and branch-&-bound. Furthermore, our version of the $(k+1)$-rule outperforms the theoretically most effective preprocessing algorithm, which yields a $2k$-vertex kernel. Notably, this $2k$-vertex kernel is analysed empirically for the first time here. Our new algorithm was developed by integrating Programming by Optimisation into the classical algorithm engineering cycle – an approach which we expect to be successful in many other contexts.

1 Introduction

Cluster Editing is a prominent NP-hard combinatorial problem with important applications in computational biology, e.g. to cluster proteins or genes (see the recent survey by Böcker and Baumbach [6]). In machine learning and data mining, weighted variants of Cluster Editing are known as Correlation Clustering [4] and have been the subject of several recent studies (see, e.g., [8, 12]). Here, we study the unweighted variant of the problem, with the goal of improving the state of the art in empirically solving it. Formally, as a decision problem it reads as follows:

Cluster Editing
Input: An undirected graph $G = (V, E)$ and a positive integer $k \in \mathbb{N}$.
Question: Is there a set of at most k edge insertions and deletions that transform G into a cluster graph, that is, a graph in which each connected component is a complete graph?

* Major parts of this work were done during a research visit of SH at the University of British Columbia in Vancouver (Canada), supported by a "DFG Forschungsstipendium" (HA 7296/1-1).

Cluster Editing corresponds to the basic clustering setting in which pairwise similarities between the entities represented by the vertices in G are expressed by unweighted edges, and the objective is to find a pure clustering, in the form of a cluster graph, by modifying as few pairwise similarities as possible, i.e., by removing or adding a minimal number of edges. Notably, this clustering task requires neither the number of clusters to be specified, nor their sizes to be bounded.

Related Work. The Cluster Editing problem is known to be APX-hard [10] but can be approximated in polynomial time within a factor of 2.5 [25]. Furthermore, various efficient implementations of exact and heuristic solvers have been proposed and experimentally evaluated (see the references in [6]). These methods can be divided into exact algorithms, which are guaranteed to find optimal solutions to any instance of Cluster Editing, given sufficient time, and inexact algorithms, which provide no such guarantees, but can be very efficient in practice. State-of-the-art exact Cluster Editing algorithms are based on integer linear programming (ILP) or specialised branch-&-bound methods (i.e., search tree) [6, 7]. Theoretically, the currently best *fixed-parameter* algorithm runs in $\mathcal{O}(1.62^k + |G|)$ time and it is based on a sophisticated search tree method [5].

Our work on practical exact algorithms for Cluster Editing makes use of so-called *data reduction rules* [11, 16, 17, 19] – preprocessing techniques from parameterised algorithmics that are applied to a given instance with the goal of shrinking it before attempting to solve it. Furthermore, when solving the problem by a branch-&-bound search, these data reduction rules can be "interleaved" [23], meaning that they can be again applied within each recursive step. If after the exhaustive application of data reduction rules the size of the remaining instance can be guaranteed to respect certain upper bounds, those instances are called *problem kernels* [14, 23]. Starting with an $\mathcal{O}(k^2)$-vertex problem kernel [17], the best state-of-the-art kernel for Cluster Editing contains at most $2k$-vertices [11].

Our Contribution. Starting from a search tree procedure originally developed for a more general problem called M-Hierarchical Tree Clustering (M-Tree Clustering) [20], and making heavy use of data reduction rules, we developed a competitive state-of-the-art exact solver for (unweighted) Cluster Editing.[1]

To achieve this goal, and to study the practical utility of data reduction rules for Cluster Editing, we employed multiple rounds of an algorithm engineering cycle [24] that made use of the *Programming by Optimisation (PbO)* paradigm [21]. In a nutshell, PbO is based on the idea to consider and expose design choices during algorithm development and implementation, and to use automated methods to make those choices in a way that optimises empirical performance for given use contexts, characterised by representative sets of input data.

We show that, using a clever implementation of a well-known (from a theoretical point of view, out-dated) reduction rule, called $(k+1)$-Rule, we can achieve improvements over existing state-of-the-art exact solvers for Cluster Editing on challenging real-world and synthetic instances. For example, for the synthetic

[1] Notably, our implementation is still able to solve M-Tree Clustering. However, here our focus is on improving over state-of-the-art exact solvers for Cluster Editing.

data with a timeout of 300 s our so-called Hier solver times out only on 8 % of the 1476 instances, while the best previously known solver has a rate of 22 %. Furthermore, we demonstrate that on the hardest instances the $(k+1)$-Rule dominates on aggregate all other data reduction rules we considered, and that using the best known data reduction rules [9, 11] (yielding the best known kernel of size $2k$) does not yield further significant improvements.

Achieving these results involved multiple rounds of optimizing the implementation of the $(k+1)$-Rule as well as the use of automated algorithm configuration tools in conjunction with a new method for selecting the sets of training instances used in this context. It is based on the coefficient of variation of the running time observed in preliminary runs, which we developed in the context of this work, but believe to be more broadly useful.

Overall, our work demonstrates that the adoption of the Programming by Optimisation paradigm, and in particular, the use of automated algorithm configuration methods can substantially enhance the "classical" algorithm engineering cycle and aid substantially in developing state-of-the-art solvers for hard combinatorial problems, such as Cluster Editing. We note that a similar approach has been taken by de Oca et al. [13] to optimise a particle swarm optimization algorithm.

2 Preliminaries

We use standard graph-theoretic notations. All studied graphs are undirected and simple without self-loops and multi-edges. For a given graph $G = (V, E)$ with vertex set V and edge set E, a set consisting of edge deletions and additions over V is called an *edge modification set*. For a given Cluster Editing-instance (G, k) an edge modification set S over V is called a *solution*, if it is of size at most k and transforms G into a cluster graph, which we denote by $G \otimes S$. For convenience, if two vertices $\{u, v\}$ are not adjacent, we call $\{u, v\}$ a *non-edge*.

It is well-known that a graph $G = (V, E)$ is a cluster graph if, and only if, it is *conflict-free*, where three vertices $\{u, v, w\} \subseteq V$ form a *conflict* if $\{u, v\}, \{v, w\} \in E$, but $\{u, w\} \notin E$ – in other words, a conflict consists of three vertices with two edges and one non-edge. We denote by $\mathcal{C}(G)$ the set of all conflicts of G. Branching into either deleting one of the two edges in a conflict or adding the missing edge is a straight-forward search tree-strategy that results in a $\mathcal{O}(3^k + |V|^3)$ algorithm to decide an instance $((V, E), k)$ [17]. This algorithm can be generalised to M-Tree Clustering [20] and is the basic algorithm implemented in our Hier solver.

Parametrised Algorithmics. Since our algorithm makes use of data reduction rules known from parametrised algorithmics, and Cluster Editing has been intensely studied in this context, we briefly review some concepts from this research area (see [14, 23]). A problem is *fixed-parameter tractable* (FPT) with respect to a parameter k if there is a computable function f such that any instance (I, k), consisting of the "classical" problem instance I and parameter k, can be exactly solved in $f(k) \cdot |I|^{O(1)}$ time. In this work k always refers to the "standard" parameter solution size.

The term *problem kernel* formalizes the notion of effective and (provably) efficient preprocessing. A *kernelization algorithm* reduces any given instance (I, k) in polynomial time to an equivalent instance (I', k') with $|I'| \leq g(k)$ and $k' \leq g(k)$ for some computable function g. Here, equivalent means that (I, k) is a yes-instance if, and only if, (I', k') is a yes-instance. The instance (I', k') is called *problem kernel* of size g. For example, the smallest problem kernel for Cluster Editing consists of at most $2k$ vertices [11]. A common way to derive a problem kernel is by the exhaustive application of *data reduction rules*. A data reduction rule is a polynomial-time algorithm which computes for each instance (I, k) an equivalent *reduced* instance (I', k') and it has been applied *exhaustively* if applying it once more would not change the instance.

PbO and Automated Algorithm Configuration. Programming by Optimisation (PbO) is a software design approach that emphasises and exploits choices encountered at all levels of design, ranging from high-level algorithmic choices to implementation details [21]. PbO makes use of powerful machine learning and optimisation techniques to find instantiations of these choices that achieve high performance in a given application situation, where application situations are characterised by representative sets of input data, here: instances of the Cluster Editing problem. In the simplest case, all design choices are exposed as algorithm parameters and then optimised for a given set of training instances using an automated algorithm configurator. In this work, we use SMAC [22] (in version 2.08.00), one of the best-performing general-purpose algorithm configurators currently available. SMAC is based on sequential model-based optimisation, a technique that iteratively builds a model relating parameter settings to empirical performance of a given (implementation of a) target algorithm $\mathcal{A}$, here: our Cluster Editing solver Hier, and uses this model to select promising algorithm parameter configurations to be evaluated by running $\mathcal{A}$ on training instances.

By following a PbO-based approach, using algorithm configurators such as SMAC, algorithm designers and implementers no longer have to make *ad-hoc* decisions about heuristic mechanisms or settings of certain parameters. Furthermore, to adapt a target algorithm to a different application context, it is sufficient to re-run the algorithm configurator, using a set of training instances from the new context.

We note that the algorithm parameters considered in the context of automated configuration are different from the problem instance features considered in parameterised algorithmics, where these features are also called parameters.

3 Our Algorithm

Basic Algorithm Design. The algorithm framework underlying our Hier solver is outlined in Algorithm 2; the actual implementation has several refinements of this three-step approach, and many of them are exposed as algorithm parameters (in total: 49) to be automatically configured using SMAC.

Given a graph G as input for the optimization variant of Cluster Editing, we maintain a lower and upper bound, called k_{LB} and k_{UB}, on the size of an

ALGORITHM 2: Pseudo code of our Hier solver.

```
1  Algorithm Hier ()
       Input: Graph G.
       Output: The size k_OPT of a minimum edge modification set S such that G ⊗ S is
               a cluster graph.
3      Compute a lower bound k_LB ≤ k_OPT
5      Compute an upper bound k_OPT ≤ k_UB
7      while k_LB < k_UB do
8          if decisionSolver(G, k_LB)=YES then
9              return k_LB
10         else
11             increase k_LB //details are subject to two algorithm parameters
12         end
13     end
1  Procedure decisionSolver(G, k)
       Input: Graph G and integer k.
       Output: YES/NO whether there is a size-at-most-k edge modification set for G.
2      (G, k) ← Apply data reduction rules to (G, k)
3      if LP-based lower bound on modification cost for G > k then return NO
4      {u, v, w} ← a conflict in G
5      if {u, v} is unmarked ∧ decisionSolver (G − uv, k − 1) =YES then return YES
6      else Mark edge {u, v} unmodifiable
7      if {v, w} is unmarked ∧ decisionSolver (G − vw, k − 1) =YES then return
       YES
8      else Mark edge {v, w} unmodifiable
9      if decisionSolver(G + uw, k − 1) =YES then return YES
10     else return NO
```

optimal solution for G. As long as lower and upper bound are not equal, we call our branch-&-bound search procedure (Line 8) to decide whether (G, k_{LB}) is a yes-instance. At the heart of our solver lies the following recursive procedure for solving the (decision variant) Cluster Editing-instance (G, k_{LB}). First, a set of data reduction rules is applied to the given instance (see Line 2 in `decisionSolver`). Next, a lower bound is computed on the size of a minimum solution using our LP-based lower bound algorithm. If this lower bound is larger than k, then we abort this branch, otherwise we proceed with the search. Afterwards, if there are still conflicts in the resulting graph, one of these is chosen, say $\{u, v, w\}$. Then the algorithm branches into the three possibilities to resolve the conflict: Delete the edge $\{u, v\}$, delete $\{v, w\}$, or add the edge $\{u, w\}$.

On top of this, if the branch of deleting edge $\{u, v\}$ has been completely explored without having found any solution, then in all other branches this edge can be marked as unmodifiable (the branch for deleting $\{v, w\}$ is handled analogously). Moreover, in all three recursive steps, the (non-)edge that was introduced to solve the conflict $\{u, v, w\}$ gets marked as unmodifiable. Furthermore, the choice of the conflict to resolve prefers conflicts involving unmodifiable (non-)edges, since this reduces the number of recursive calls by one or, in the best case, completely determines how to resolve the conflict. Combining this with

solving "isolated" conflicts is known to reduce the (theoretical) time complexity from $\mathcal{O}(3^k + |V|^3)$ to $\mathcal{O}(2.27^k + |V|^3)$ [17]. Our empirical investigation revealed that this improvement is also effective in practice.

Data Reduction Rules. In total, we considered seven data reduction rules and implemented them such that each of them can be individually enabled or disabled via an algorithm parameter. We first describe three rather simple data reduction rules. First, there is a rule (Rule 2 in Hier) that deletes all vertices not involved in any conflict (see [20] for the correctness). A second simple rule (Rule 4 in Hier) checks all sets of three vertices forming a triangle, and in case two of the edges between them are already marked as unmodifiable it also marks the third one (deleting this edge would result in a unresolvable conflict). The last simple rule (Rule 6 in Hier) checks each conflict and resolves it in case of there is only one way to do this as a result of already marked (non-)edges.

We describe the remaining "sophisticated" data reduction rules in chronological order of their invention. Each of it either directly yields or is the main data reduction rule of a problem kernel.

$(k+1)$-Rule: Gramm et al. [17] provide a problem kernel of size $\mathcal{O}(k^3)$ that can be computed in $\mathcal{O}(n^3)$ time. More specifically, the kernel consists of at most $2k^2 + k$ vertices and at most $2k^3 + k^2$ edges. At the heart of this kernel lies the following so-called $(k+1)$-Rule (Rule 1 in [17]):

> Given a Cluster Editing-instance (G, k), if there are two vertices $\{u, v\}$ in G that are contained in at least $k+1$ conflicts in $\mathcal{C}(G)$, then in case of $\{u, v\} \notin E$ add the edge $\{u, v\}$ and otherwise delete the edge $\{u, v\}$.

The $(k+1)$-Rule is correct, since a solution that is not changing the (non-)edge $\{u, v\}$ has to resolve all the $\geq k+1$ conflicts containing $\{u, v\}$ by pairwise disjoint edge modifications; however, this cannot be afforded with a "budget" of k.

We heuristically improved the effectiveness of the $(k+1)$-Rule by the following considerations: For a graph G denote by $\mathcal{C}(\{u, v\}) \subseteq \mathcal{C}(G)$ all conflicts containing $\{u, v\}$. If $|\mathcal{C}(\{u, v\})| \geq k+1$, then the $(k+1)$-Rule is applicable. Otherwise, let $\mathcal{C}_{\overline{u,v}}(G) \subseteq \mathcal{C}(G) \setminus \mathcal{C}(\{u, v\})$ be all conflicts that are (non-)edge-disjoint with $\mathcal{C}(\{u, v\})$, meaning that any pair of vertices occurring in a conflict in $\mathcal{C}_{\overline{u,v}}(G)$ does not occur in a conflict in $\mathcal{C}(\{u, v\})$. By the same argument as for the correctness of the $(k+1)$-Rule, it follows that if any lower bound on the number of edge modifications needed to solve all conflicts in $\mathcal{C}_{\overline{u,v}}(G)$ plus $|\mathcal{C}(\{u, v\})|$ exceeds k, then the (non-)edge $\{u, v\}$ needs to be changed (all these conflicts require pairwise disjoint edge modifications). We use our heuristic algorithm described below to compute a (heuristic) lower bound on the modification cost of $\mathcal{C}_{\overline{u,v}}(G)$.

As our experimental analysis reveals, the heuristically improved version of the $(k+1)$-Rule is the most successful one in Hier. Its operational details are configurable by three algorithm parameters (not counting the parameters to enable/disable it), and we implemented two different versions of it (Rule 0 & 1 in Hier). These versions differ in their "laziness": Often it is too time consuming to exhaustively apply the $(k+1)$-Rule, as any edge modification requires an update on the lower bound for $\mathcal{C}_{\overline{u,v}}(G)$. In addition to various heuristic techniques, we

implemented a priority queue that (heuristically) delivers the (non-)edges that are most likely reducible by the $(k+1)$-Rule.

$\mathcal{O}(M \cdot k)$-vertex kernel: There is a generalisation of Cluster Editing called M-Tree Clustering, in which the input data is clustered on M levels [2]. The parametrised complexity of M-Tree Clustering has been first examined by Guo et al. [20], who introduced a $(2k \cdot (M+2))$-vertex kernel which is computable in $\mathcal{O}(M \cdot n^3)$ time. This kernel basically corresponds to a careful and level-wise application of the $4k$-vertex kernel by Guo [19] for Cluster Editing. The underlying technique is based on so-called *critical cliques* – complete subgraphs that have the same neighbourhood outside and never get split in an optimal Cluster Editing-solution. We refer to Guo et al. [20] for a detailed description of the implemented $\mathcal{O}(M \cdot k)$ kernel (Rule 3 in Hier).

$2k$-vertex kernel: The state-of-the-art problem kernel for Cluster Editing has at most $2k$-vertices and is based on so-called *edge-cuts* [11]. In a nutshell, for the closed neighbourhood N_v of each vertex v, the cost of completing it to a complete graph (adding all missing edges into N_v) and cutting it out of the graph (removing all edges between a vertex in N_v and a vertex not in N_v) is accumulated. If this cost is less than the size of N_v, then N_v is completed and cut out. This kernel has been generalised to M-Tree Clustering without any increase in the worst-case asymptotic size bound [9]. We implemented this kernel in its generalized form for M-Tree Clustering (Rule 7), but omitted a rule that basically merges N_v after it has been completed and cut out of the graph; although this rule is necessary for the bound on the kernel size, as it removes vertices from the graph, Hier will not deal with these vertices again and thus simply ignores them.

Lower- and Upper-Bound Computation. We implemented two lower-bound algorithms (LP-based and heuristic) and one upper-bound heuristic. Our preliminary experiments revealed that high-quality lower- and upper-bound algorithms are a key ingredient for obtaining strong performance in our Cluster Editing solver. In total, these algorithms expose twenty-two algorithm parameters that influence their application and behaviour.

LP-based lower bound computation: We implemented the ILP-formulation for M-Tree Clustering proposed by Ailon and Charikar [3], which corresponds to the "classical ILP-formulation" for Cluster Editing in case of $M = 1$ [6]. The formulation involves a 0/1-variable for each vertex of the graph and a cubic number of constraints. Our LP-based lower bound algorithm simply solves the relaxed LP-formulation where all variables take real values from the interval $[0, 1]$, which provides a lower bound on any ILP-solution. If after having solved the relaxed LP-formulation the time limit (set via an algorithm parameter) has not been exceeded, then we require a small fraction of the variables to be 0/1-integers and try to solve the resulting mixed-integer-linear-program (MIP) again. Surprisingly, to obtain optimal integer solutions, in many cases, one only needs to require a small fraction of the variables ($\approx 10\%$) to be 0/1-integers. Using this mechanism, we are frequently able to provide optimal bounds on the solution size, especially for small instances where the LP-formulation can be solved quickly.

Heuristic lower bound computation: Given a set of conflicts $\mathcal{C}$ (not necessarily all, as in the application of the $(k+1)$-Rule), our second lower bound algorithm heuristically determines a maximum-size set of independent conflicts based on the following observation. Consider the conflict graph for $\mathcal{C}$, which contains a vertex for each conflict in $\mathcal{C}$ and an edge between two conflicts if they have a (non-)edge in common. A subset of vertices is an *independent set* if there is no edge between any two vertices in it. Similarly to the correctness argument for the $(k+1)$-Rule, it follows that the size of an independent set in the conflict graph of $\mathcal{C}$ is a lower bound on the number of edge modifications needed to resolve all conflicts in $\mathcal{C}$. Computing a maximum-size independent set in a graph is a classical NP-hard problem, and we thus implemented the commonly known "small-degree heuristic" to solve it: As long as the graph is not empty, choose one of the vertices with smallest degree, put it into the independent set and delete it and all its neighbours. We apply this small-degree heuristic multiple times with small (random) perturbations on the order in which the vertices get chosen (not necessarily a smallest degree vertex is chosen, but only one with small degree). In total, there are four algorithm parameters which determine the precise way in which the order is perturbed and how often the heuristic is applied.

Heuristic upper bound computation: Given a graph G and the set of conflicts $\mathcal{C}(G)$ in G, we use the following heuristic algorithm to compute an upper bound on the minimum modification cost for G. The *score* of an (non-)edge is the number of its occurrences in $\mathcal{C}(G)$, and the score of a conflict is simply the maximum over the scores of all its modifiable (non-)edges. The algorithm proceeds as follows: While there are still conflicts in $\mathcal{C}(G)$, choose a conflict with highest *score* in $\mathcal{C}(G)$ and among the modifiable (non-)edges change (delete if it is an edge otherwise add) one of those with highest score. Furthermore, mark the corresponding (non-)edge as unmodifiable. Before solving the next conflict, we exhaustively apply Rule 6, which solves all conflicts for which two of its (non)-edges have been marked as unmodifiable.

In our implementation, the score of an edge is randomly perturbed, and thus we run the algorithm described above multiple times and return the minimum over all these runs. The time limit for this computation as well as the maximum number of rounds are exposed as algorithm parameters.

4 Experimental Results

Algorithms and Datasets. We compare our solver, Hier, with two other exact solvers for (weighted) Cluster Editing: The Peace solver by Böcker et al. [7] applies a sophisticated branching strategy based on merging edges, which yields a search tree of size at most $\mathcal{O}(1.82^k)$. This search tree algorithm is further enhanced by a set of data reduction rules that are applied in advance and during branching. Böcker et al. [7] compared the empirical performance of Peace against that obtained by solving an ILP-formulation (due to Grötschel and Wakabayashi [18]) using the commercial CPLEX solver 9.03. In August 2013, a new version 2.0 of this ILP-based approach has become available, which now directly combines

data reduction rules with an ILP-formulation. We refer to this solver as Yoshiko[2] (developed by G. Klau and E. Laude, VU University Amsterdam).

We compare our algorithm to Peace and Yoshiko (version 2.0) on the synthetic and biological datasets provided by Böcker et al. [7]. The (unweighted) synthetic dataset consists of 1475 instances that are generated from randomly disturbed cluster graphs with 30-1040 vertices (median: 540) and densities of 11-99%. These instances have been observed to be substantially harder than the biological dataset, which consists of 3964 instances that have been obtained from a protein similarity network.[3] The number of vertices in the biological dataset range from 3 to 3387, but the median is only 10, and thus, most instances are rather easy. Since the biological instances are weighted Cluster Editing-instances and Hier is restricted to unweighted Cluster Editing (as a result of its ability to solve the general M-Tree Clustering problem), we transformed them into unweighted instances by setting edges only for the $c\,\%$ of the pairs with highest weight (corresponds to highest similarity). Using three different values of $c = 33, 50,$ and 66, we obtained 11 889 biological instances in total.

Implementation and Execution Environment. All our experiments were run on an Intel Xeon E5-1620 3.6 Ghz machine (4 Cores + Hyper-Threading) with 64 GB memory under the Debian GNU/Linux 6.0 operating system, with a time limit of 300 s per problem instance. Our Hier solver was implemented in Java and is run under the OpenJDK runtime environment in version 1.7.0_25 with 8 GB heap space. We use the commercial Gurobi MIP solver in version 5.62 to compute our LP-based lower bound [1]. The source code along with the scenario file used for configuration with SMAC is freely available.[4] For Yoshiko, we used the binary provided by the authors, and we compiled Peace using the provided Make file with gcc, version 4.7.2. Our Hier solver sets up parallel threads for computing the lower and upper bounds, but otherwise runs in only one thread. Peace uses a single thread, while Yoshiko makes extensive use of the parallel processing capabilities of the CPU (according to its output, Yoshiko sets up 8 threads). All running times were measured in wall-clock seconds.

Results for Synthetic Dataset. Table 1 and the scatter plots in Figure 1 provide an overview of our experimental findings on the synthetic dataset. Hier-Opt$_S$ refers to Hier with the best configuration found by SMAC. Before discussing how we obtained this configuration we first discuss the performance of Hier's default configuration (always referred to simply as Hier) to that of Yoshiko and Peace.

As can be seen from these results, Hier clearly outperforms both Yoshiko and Peace (see columns 4-6 in Table 1). Furthermore, it seems that search-tree based algorithms, such as Peace and Hier, generally perform better than the ILP-based Yoshiko-solver. We suspect that this is mainly due to the instance sizes which are considerably larger than for the biological dataset. As can be seen in the top left scatter plot in Figure 1, Peace is on average faster than Hier for instances

[2] http://www.mi.fu-berlin.de/w/LiSA/YoshikoCharles

[3] We removed the largest instance with 8836 vertices from the dataset. It is more than two times larger than the second largest instance and could not be solved.

[4] http://fpt.akt.tu-berlin.de/cluEdit/

Table 1: Running time (wall time in s) comparison of four different solvers on the synthetic dataset (performance on disjoint training and test instances).

	training (#=196)		test (#=953)				
	Hier	Hier-Opts	Peace	Yoshiko	Hier	Hier-Opts	Hier-Opts-Rule7
Par-10	187.4	127.2	662.2	904.1	255.2	252.7	265.8
Mean	49.7	30.7	92.8	142.0	45.6	40.2	42.0
Median	28.4	7.5	26.4	96.6	18.6	10.1	9.6
% Timeouts	5.1%	3.6%	21.1 %	28.2 %	7.8 %	7.9 %	8.3 %

solvable within ≤ 25 s by both solvers. However, the higher the time required by both solvers, the more Hier starts to dominate on average, and, of course, its overall success is heavily due to the smaller timeout-rate of 7.8 % (Peace: 21 %).

The bottom two scatter plots in Figure 1 show that Hier-Opts clearly dominates Yoshiko and Peace on most instances (also on instances solvable in a couple of seconds). We obtained Hier-Opts by using SMAC; however, not by a single "shot", but rather by using SMAC repeatedly within an algorithm engineering cycle. This means that we performed multiple rounds of tweaking the implementation, testing it, and analysing it on our experimental data. Therein, in each round we performed multiple SMAC runs in order to analyse not only the default configuration of our current solver but also its optimized variant. We then used an ablation analysis [15] to further pinpoint the crucial parameter adjustments made by SMAC. This was important, because it revealed which algorithm parameters – and thus, which parts of the algorithm – are particularly relevant for the overall performance of our solver. For example, we learned that by allowing more time for the application of our original implementation of the $(k+1)$-Rule, we can reduce the number of timeouts. We thus spent serious effort on tweaking the implementation of the $(k+1)$-Rule and making more of its details accessible to get optimized by SMAC. Of course, if one parameter setting clearly had been identified by SMAC to be beneficial, then we adjusted the default values of this parameter for the next round. This is the main reason why the final default configuration of Hier is already quite competitive (for example, we started with a version of Hier that had more than 30 % timeouts on the synthetic data).

In each round of the algorithm engineering cycle, we performed at least five independent SMAC runs, each with a wall-clock time limit of 36 hours and a cut-off time of 300 s per run of Hier. In each SMAC run about 160–200 configurations were evaluated and about 1200–1500 runs of Hier have been performed. We not only started SMAC from the default configuration, but also with the best configuration that had been obtained in previous runs (we obtained our final best configuration from one of these runs). We chose a validation set of 368 instances uniformly at random from the entire synthetic dataset, and we selected the best configurations from multiple SMAC runs based on their performance on this set. Our training set was initially also chosen uniformly at random from the entire synthetic data set. However, we found that SMAC found better configurations when selecting the training set as follows: We had, from multiple rounds of the algorithm engineering cycle, multiple performance evaluations for

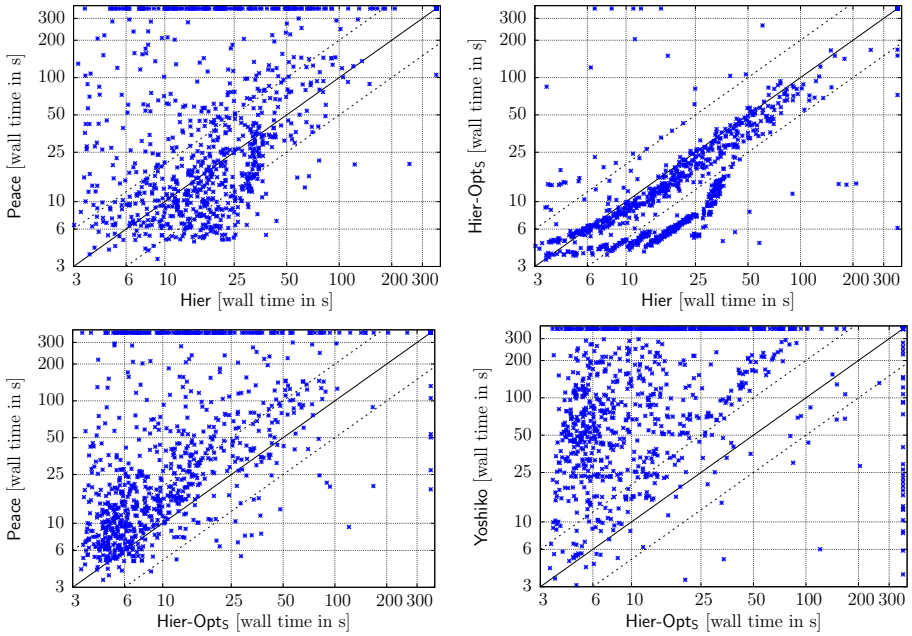

Fig. 1: Scatter plots of the running time of all solvers on the test instances of the synthetic dataset (full synthetic set minus training and validation instances). Timeouts ($> 300\,\mathrm{s}$) are plotted at $360\,\mathrm{s}$.

default and optimised configurations, and we observed that on many instances, these running times did not vary. More specifically, there were many instances whose solving times only seemed to improve due to some general improvements (e. g. parallelizing the lower and upper bound computation) but appeared to be almost entirely uncorrelated with algorithm parameters. Surprisingly, this was true not only for rather quickly solvable instances, where one would expect only minor differences, but also for harder instances. For example, we found instances that were almost completely unaffected by the data reduction rules and that were solved by exploring a (more less constant) number of search-tree nodes. In light of this observation, we computed for each instance the coefficient of variation (standard deviation divided by the mean) of the running times measured for different configurations we had run on it. We then selected only the instances with the highest coefficient of variation into a training set of size 196.

As can be seen in the top right scatter plot in Figure 1, the configuration Hier-Opts clearly dominates Hier on average. Furthermore, according to columns 6 and 7 in Table 1, although Hier-Opts improves the timeout-rate only slightly from 5.1 % to 3.6 % (on training data), the mean and PAR-10 running times are considerable smaller and the median is less than half.[5] Notably, Hier-Opts enables the $(k+1)$-Rule but disables all other data reduction rules. While this was already observed for Rule 3 (computing the $\mathcal{O}(M \cdot k)$ kernel) in

[5] PAR-10 is the average with timeouts counted as 10 times the cut-off time.

Table 2: Running time (wall time in s) comparison of five solvers on the biological dataset with different "density" parameters c. The median of all solvers is less than 0.2 s.

	Peace			Hier			Hier-Opt$_B$			Yoshiko			Yoshiko & Hier-Opt$_B$		
c	33	50	66	33	50	66	33	50	66	33	50	66	33	50	66
Par-10	109	124	126	101	94.9	84.4	78.8	78.1	65.8	72.8	82.1	66.4	68.7	68.8	53.0
Mean	11.9	13.9	14	11.2	11.1	10.1	9.3	9.3	8.6	8.8	9.9	8.5	8.1	8.2	6.7
Timeouts	142	161	164	132	123	109	102	101	84	94	106	85	89	89	68

previous studies [20], this was surprising for Rule 7, which computes the $2k$-vertex kernel [9]. The last column in Table 1 provides the results for Hier-Opt$_S$ with Rule 7 enabled. Interestingly, while it slightly decreases the running time (mean and PAR-10) due to slightly more timeouts, the median is even lower than for Hier-Opt$_S$. This shows that Rule 7, in principle, reduces the running time on many instances, but the cost of applying it is overall not amortised by its benefits.

Results for Biological Dataset. Our experimental findings for the biological dataset are summarized in Table 2 and in the scatter plots in Figure 2.

Unlike for the synthetic dataset, the ILP-based solver Yoshiko clearly outperforms Peace and Hier. However, comparing results for the latter two revealed that Hier is still better than Peace (see the upper-right plot in Figure 2), especially, on harder instances. In general, since the median of the running times is pretty small (for Hier ≤ 0.18 s and for Peace and Yoshiko even ≤ 0.01 s), we suspect that our Hier solver suffers from the fact that on extremely easy instances the initialization cost of the Java VM dominates the running time.

While the default configuration of Hier is not competitive with Yoshiko, our SMAC-optimized configuration, called Hier-Opt$_B$, considerably closes this gap. Although, being greatly slower for density value $c = 33$, Hier-Opt$_B$ clearly beats Yoshiko for $c = 50$ and even slightly for $c = 66$. The bottom right plot in Figure 2 stresses this point by clearly demonstrating that starting from instances that require at least 10 s on both solvers, Hier-Opt$_B$ begins to dominate on average. This behaviour goes together with the observations that can be made from directly comparing Hier-Opt$_B$ with Hier (see the bottom-left plot in Figure 2): For instances up to 1 s, Hier and Hier-Opt$_B$ roughly exhibit the same performance, but the higher the running times get, the clearer Hier-Opt$_B$ is dominating on average. We suspect that this is mainly caused by an algorithm parameter adjustments made in Hier-Opt$_B$ that heavily increases the time fraction spend to compute the initial lower bound. While easy instances do not largely benefit from computing a slightly better lower bound, on large instances this might save expensive calls of the search-tree solver for the decision variant. Even better performance can be obtained by running Hier-Opt$_B$ and Yoshiko in parallel on the same instances, as evident from the bottom right plot of Figure 2. To demonstrate the potential of this approach, the last column in Table 2 shows the running times of a virtual solver that takes the minimum of Yoshiko and Hier-Opt$_B$ for each instance.

To obtain Hier-Opt$_B$, SMAC was used in the same way as for the synthetic data, but could typically perform about 7500 algorithm runs and evaluate 3500

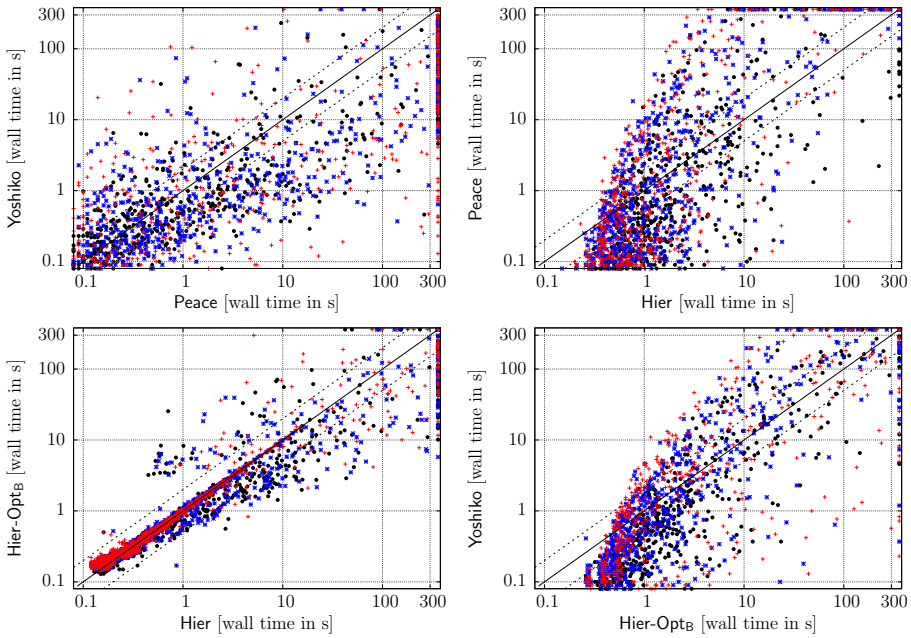

Fig. 2: Scatter plots of the running time of all solvers on the biological dataset (point colour/value for c: black/33, blue/50, red/66). Timeouts (> 300 s) are plotted at 360 s.

different configurations, because the instances tend to be easier. Due to the small median running time, we once again selected the training set based on the coefficient of variation but only among those instances, where at least one previous run needed at least 0.5 s. On the 327 training instances, the PAR-10 running time value of Hier is 850 s and could be improved to 149 s for Hier-Opt$_B$. This improvement was mainly due to a reduction in the number of timeouts from 90 down to 12.

We note that Hier-Opt$_B$ enables all data reduction rules, except the two simple Rules 4 & 6. However, Rule 7 (computing the $2k$-vertex kernel) is also almost disabled, since it is applied only in every 88th recursive step (adjusted by an algorithm parameter) of the search tree. For all other enabled rules, this "interleaving constant" is at most 13. Overall, having a more heterogeneous set of data reductions seems to be important on the biological dataset, but not for synthetic data, where only the $(k+1)$-Rule was enabled. Our default Hier enables all rules except Rules 4 and 7.

Finally, to investigate to which extent the difference in the use of parallel processing capabilities of our CPU between Yoshiko and Hier affect our results, we conducted the following experiment: For the biological dataset and $c = 33$ (where Yoshiko performed better than Hier-Opt$_B$) we computed for each instance that could be solved by both solvers the maximum of their running times. According to these, we then sorted the instances in descending order and performed on the instances with number 1-100 and 301-400 another run of Yoshiko and Hier-Opt$_B$,

Table 3: Running time (wall time in s) comparison of Yoshiko and Hier-Opt$_B$ on biological data for multi- vs single-threaded execution on our multi-core CPU.

	multi-threaded		single thread	
	Hier-Opt$_B$	Yoshiko	Hier-Opt$_B$	Yoshiko
Mean running time, instances 1-100	39.9	44.9	76.1	70.0
Mean running time, instances 301-400	1.7	0.4	3.7	0.8
Timeouts	0	0	7	4

were we restricted the CPU to run in single-threaded mode. Table 3 shows the results of this experiment. To our surprise, despite of the different ways the solvers explicitly use parallel resources, their performance slows down only by a factor of less than two when restricted to sequential execution. The reasons for this unexpected result, especially for the CPLEX-based Yoshiko solver, are somewhat unclear and invite further investigation.

5 Conclusions & Future Work

We have shown how, by combining data reduction rules known from parameterised algorithmics with a heuristically enhanced branch-&-bound procedure, we can solve the NP-hard (unweighted) Cluster Editing problem more efficiently in practice than the best known approaches known from the literature. This success was enabled by integrating Programming by Optimisation into the classical algorithm engineering cycle and, as a side effect, lead to a new method for assembling training sets for effective automated algorithm configuration.

It would be interesting to see to which extent further improvements could be obtained by automatically configuring the LP solver used in our algorithm, or the MIP solver used by Yoshiko. Furthermore, we see potential for leveraging the complementary strengths of the three algorithms studied here, either by means of per-instance algorithm selection techniques, or by deeper integration of mechanisms gleaned from each solver. We also suggest to study more sophisticated methods, such as multi-armed bandit algorithms, to more fine-grainely determine in which depths of the search tree a data reduction rule should be applied. Finally, we see considerable value in extending our solver to weighted Cluster Editing, and in optimising it for the general M-Hierarchical Tree Clustering problem.

Acknowledgement. We thank Tomasz Przedmojski who provided, as part of his bachelor thesis, an accelerated implementation of the $\mathcal{O}(M \cdot k)$ kernel [20].

References

[1] Gurobi 5.62. Software, 2014.

[2] R. Agarwala, V. Bafna, M. Farach, B. Narayanan, M. Paterson, and M. Thorup. On the approximability of numerical taxonomy (fitting distances by tree matrices). *SIAM J. Comput.*, 28(3):1073–1085, 1999.

[3] N. Ailon and M. Charikar. Fitting tree metrics: Hierarchical clustering and phylogeny. In *Proc. 46th FOCS*, pages 73–82. IEEE Computer Society, 2005.

[4] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. *Mach. Learn.*, 56(1–3): 89–113, 2004.
[5] S. Böcker. A golden ratio parameterized algorithm for Cluster Editing. *J. Discrete Algorithms*, 16:79–89, 2012.
[6] S. Böcker and J. Baumbach. Cluster Editing. In *Proc. 9th CIE*, volume 7921 of *LNCS*, pages 33–44. Springer, 2013.
[7] S. Böcker, S. Briesemeister, and G. W. Klau. Exact algorithms for Cluster Editing: Evaluation and experiments. *Algorithmica*, 60(2):316–334, 2011.
[8] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen. Chromatic correlation clustering. In *Proc. 18th ACM SIGKDD (KDD '12)*, pages 1321–1329. ACM Press, 2012.
[9] Y. Cao and J. Chen. On parameterized and kernelization algorithms for the hierarchical clustering problem. In *Proc. 10th TAMC*, volume 7876 of *LNCS*, pages 319–330, 2013.
[10] M. Charikar, V. Guruswami, and A. Wirth. Clustering with qualitative information. *J. Comput. System Sci.*, 71(3):360–383, 2005.
[11] J. Chen and J. Meng. A $2k$ kernel for the Cluster Editing problem. *J. Comput. System Sci.*, 78(1):211–220, 2012.
[12] F. Chierichetti, N. Dalvi, and R. Kumar. Correlation clustering in MapReduce. In *Proc. 20th ACM SIGKDD (KDD '14)*, pages 641–650. ACM Press, 2014.
[13] M. A. M. de Oca, D. Aydin, and T. Stützle. An incremental particle swarm for large-scale continuous optimization problems: an example of tuning-in-the-loop (re)design of optimization algorithms. *Soft Comput.*, 15(11):2233–2255, 2011.
[14] R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013.
[15] C. Fawcett and H. H. Hoos. Analysing differences between algorithm configurations through ablation. In *Proc. 10th MIC*, pages 123–132.
[16] M. R. Fellows, M. A. Langston, F. A. Rosamond, and P. Shaw. Efficient parameterized preprocessing for Cluster Editing. In *Proc. 16th FCT*, volume 4639 of *LNCS*, pages 312–321. Springer, 2007.
[17] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Graph-modeled data clustering: Exact algorithms for clique generation. *Theory Comput. Syst.*, 38(4):373–392, 2005.
[18] M. Grötschel and Y. Wakabayashi. A cutting plane algorithm for a clustering problem. *Math. Program.*, 45(1-3):59–96, 1989.
[19] J. Guo. A more effective linear kernelization for Cluster Editing. *Theor. Comput. Sci.*, 410(8-10):718–726, 2009.
[20] J. Guo, S. Hartung, C. Komusiewicz, R. Niedermeier, and J. Uhlmann. Exact algorithms and experiments for hierarchical tree clustering. In *Proc. 24th AAAI*. AAAI Press, 2010.
[21] H. H. Hoos. Programming by optimization. *Commun. ACM*, 55(2):70–80, 2012.
[22] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proc. 5th LION*, volume 6683 of *LNCS*, pages 507–523. Springer, 2011.
[23] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.
[24] P. Sanders and D. Wagner. Algorithm engineering. *it - Information Technology*, 53(6):263–265, 2011.
[25] A. van Zuylen and D. P. Williamson. Deterministic algorithms for rank aggregation and other ranking and clustering problems. In *Proc. 5th WAOA*, volume 4927 of *LNCS*, pages 260–273. Springer, 2007.