Engineering Exact Quasi-Threshold Editing

Lars Gottesbüren
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
lars.gottesbueren@kit.edu

Michael Hamann
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
michael.hamann@kit.edu

Philipp Schoch
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany

Ben Strasser
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
academia@ben-strasser.net

Dorothea Wagner
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
dorothea.wagner@kit.edu

Sven Zühlsdorf
Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
zuehlsdorf.kit@ghostdub.de

Abstract

Quasi-threshold graphs are $\{C_4, P_4\}$-free graphs, i.e., they do not contain any cycle or path of four nodes as an induced subgraph. We study the $\{C_4, P_4\}$-free editing problem, which is the problem of finding a minimum number of edge insertions or deletions to transform an input graph into a quasi-threshold graph. This problem is NP-hard but fixed-parameter tractable (FPT) in the number of edits by using a branch-and-bound algorithm and admits a simple integer linear programming formulation (ILP). Both methods are also applicable to the general $\mathcal{F}$-free editing problem for any finite set of graphs $\mathcal{F}$. For the FPT algorithm, we introduce a fast heuristic for computing high-quality lower bounds and an improved branching strategy. For the ILP, we engineer several variants of row generation. We evaluate both methods for quasi-threshold editing on a large set of protein similarity graphs. For most instances, our optimizations speed up the FPT algorithm by one to three orders of magnitude. The running time of the ILP, that we solve using Gurobi, becomes only slightly faster. With all optimizations, the FPT algorithm is slightly faster than the ILP, even when listing all solutions. Additionally, we show that for almost all graphs, solutions of the previously proposed quasi-threshold editing heuristic QTM are close to optimal.

2012 ACM Subject Classification Information systems → Clustering; Theory of computation → Graph algorithms analysis; Theory of computation → Fixed parameter tractability; Theory of computation → Branch-and-bound

Keywords and phrases Edge Editing, Integer Linear Programming, FPT algorithm, Quasi-Threshold Editing

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.10

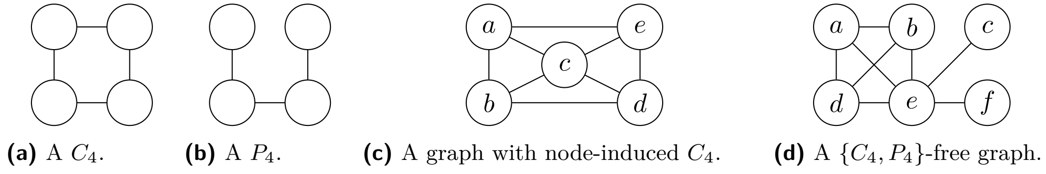
Related Version A full version of the paper is available at [13], `https://arxiv.org/abs/2003.14317`.

Supplementary Material Implementation: `https://github.com/kit-algo/fpt-editing`

Funding This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grants WA654/19-2 and WA654/22-2. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

Acknowledgements We thank James Nastos and Mark Ortmann for helpful discussions.

© Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf;
licensed under Creative Commons License CC-BY



(a) A C_4. **(b)** A P_4. **(c)** A graph with node-induced C_4. **(d)** A $\{C_4, P_4\}$-free graph.

Figure 1 A C_4, a P_4 and examples for graphs that are (not) $\{C_4, P_4\}$-free.

1 Introduction

We study graph edge editing problems. The *distance* between two graphs G and H, with the same node set, is the minimum number of edge insertions or deletions needed to transform G into H. Given a graph class $\mathcal{C}$ and a graph G, the editing problem asks for a graph $H \in \mathcal{C}$ closest to G. The corresponding decision problem asks whether k edits are sufficient to transform G into a graph $H \in \mathcal{C}$. We study algorithms to solve editing problems exactly.

A graph H is an *induced subgraph* of a graph G, if there exists an injective mapping π from the nodes of H onto the nodes of G such that there is an edge between two nodes of H, if and only if there is an edge between the corresponding nodes in G. A graph G that does not contain H as induced subgraph is *H-free*. Analogously, for a set of forbidden subgraphs $\mathcal{F}$, a graph G is $\mathcal{F}$-free, if no graph $H \in \mathcal{F}$ is an induced subgraph of G.

We denote by P_ℓ a path graph with ℓ nodes. Similarly, C_ℓ denotes a cycle graph with ℓ nodes. Figures 1a and 1b depict a C_4 and a P_4. The graph depicted in Figure 1c is not $\{C_4, P_4\}$-free as the nodes (a, b, d, e) form an induced C_4. In contrast, the graph depicted in Figure 1d is $\{C_4, P_4\}$-free. The nodes (a, b, e, c) form a P_4, however, as there is an edge between a and e, the subgraph is not an induced subgraph.

While the theoretical part of our study considers any $\mathcal{F}$-free edge editing problem for a finite set of subgraphs $\mathcal{F}$, our experimental study considers $\{C_4, P_4\}$-free graphs. These are also called *quasi-threshold* or *trivially perfect* graphs. Quasi-threshold editing has applications in detecting communities in social friendship networks. Nastos and Gao [23] detect communities in a graph G by computing a closest quasi-threshold graph H of G. Each connected component in H corresponds to a community in G.

For many choices of $\mathcal{F}$, $\mathcal{F}$-free editing is NP-hard, in particular for $\mathcal{F} = \{C_4, P_4\}$ [23]. The $\mathcal{F}$-free edge editing problem is fixed-parameter tractable (FPT) in the number of edits k [7]. This proof directly leads to a branch-and-bound algorithm, see Section 4. For $\{C_4, P_4\}$-free edge editing, it has a running time of $O(6^k \cdot (n+m))$, where n and m are the number of nodes and edges. Unfortunately, social networks typically require a large number of edits [6], which makes plain FPT algorithms impracticable. Therefore, in [23] and [6], quasi-threshold editing heuristics have been introduced for detecting communities in social friendship networks. However, as both approaches are heuristics, they might detect communities that are different from those defined by the model that assumes an optimal solution. Our goal is to improve the running time of exact $\{C_4, P_4\}$-free editing in practice in order to make it feasible at least for small networks. This allows us to study exact solutions of the community detection problem and to verify the quality of heuristics.

1.1 Related Work

For the special case of $\{C_4, P_4\}$-free edge deletion, where only edge deletion operations are allowed, optimized branching rules have been proposed that reduce the running time of the trivial algorithm from $O(4^k \cdot (n + m))$ to $O(2.42^k \cdot (n + m))$ [19]. To the best of our

knowledge, for $\{C_4, P_4\}$-free editing, no improved branching rules have been proposed so far. A polynomial kernel of size $O(k^7)$ for $\{C_4, P_4\}$-free graphs has been proposed [11], which is too large for most practical applications.

A frequently considered problem is $\{P_3\}$-free editing, better known as cluster editing [2]. Early approaches for cluster editing include a linear programming formulation with cutting planes that are incrementally added (in batches of a few hundred constraints) [14]. Later, exact algorithms based on integer linear programming as well as kernelization and more efficient FPT algorithms have been considered [4]. In [16], the authors combine the FPT algorithm with kernelization as well as upper and lower bounds. Editing to $\{P_4\}$-free graphs has been considered in phylogenomics [17] using a simple ILP-based approach.

In a bachelor thesis [5], $\{P_5\}$-free editing has been considered for community detection. They apply lower bounds, data reduction rules and rules for disallowing certain edits.

1.2 Our Contribution

In this paper, we compare two different methods for solving $\mathcal{F}$-free editing problems. The first is a branch-and-bound FPT algorithm while the second is an ILP. For the FPT algorithm, we propose a novel lower bound algorithm based on local search heuristics for independent sets as well as an improved branching strategy. Additionally, we parallelize our implementation. For the ILP, we engineer several variants of row generation. We assess the running time improvements of the different optimizations for quasi-threshold editing on a large benchmark set of 716 graphs that are connected components of a protein similarity graph. This benchmark set has previously been used to evaluate cluster editing algorithms [24, 3]. On 75% of the instances, our improved bounds and optimized branching choices yield speedups of one to three orders of magnitude for the FPT algorithm. For the ILP, we are only able to achieve small speedups. With all optimizations, in the median, the FPT algorithm is twice as fast as the ILP, even when enumerating all possible optimal solutions exactly once. Compared with the parallel execution of Gurobi [15], the FPT algorithm achieves better speedups. Additionally, we evaluate an LP relaxation as lower bound. We prove that its bounds are at least as good as our local search bounds. In our experiments, however, it is too slow to be competitive.

Further, we compare our exact solutions with heuristic solutions found by QTM [6]. It turns out that many heuristic solutions are exact and all but one of them are close to the exact solution. Additionally, we are able to solve four out of the five social networks considered in [23], of which only one was solved previously [21].

1.3 Outline

We start by introducing the preliminaries in Section 2. We describe the ILP formulation and the optimizations we apply to it in Section 3. In Section 4, we then introduce the branch-and-bound FPT algorithm including existing and novel optimizations. In Section 5, we present our experimental setup and evaluation. We conclude in Section 6.

2 Preliminaries

All graphs in this paper are undirected, unweighted, and finite. Further, no graph has self-loops or multi-edges. A graph $G = (V_G, E_G)$ consists of $n := |V_G|$ nodes and $m := |E_G|$ undirected edges. By $\overline{E_G}$, we denote the complement of the edges. In the following, k denotes the maximum number of edits.

3 Integer Linear Programming

In this section, we describe an ILP formulation for $\mathcal{F}$-free editing that is based on an existing formulation for cluster editing [14]. Further, we introduce our optimizations based on row generation and modified constraints to make the ILP practical for small instances.

For every node pair $u, v \in \binom{V_G}{2}$ we introduce a variable $x_{uv} \in \{0, 1\}$ which is 1 if the node pair is an edge in the edited graph and 0 otherwise. We add constraints to ensure that no forbidden subgraph $H \in \mathcal{F}$ can be induced in G via an injective node mapping π:

$$\forall H \in \mathcal{F}, \forall \pi\colon V_H \hookrightarrow V_G : \sum_{\{u,v\} \in E_H} (1 - x_{\pi(u)\pi(v)}) + \sum_{\{u,v\} \in \overline{E_H}} x_{\pi(u)\pi(v)} \geq 1 \tag{1}$$

The objective minimizes the number of edits:

$$\min \sum_{\{u,v\} \in E_G} (1 - x_{uv}) + \sum_{\{u,v\} \in \overline{E_G}} x_{uv} \tag{2}$$

3.1 Row Generation

Generating all of the above-mentioned constraints is infeasible, even for small instances. Row generation (also called lazy constraints) aims to speed up ILP solvers by starting with a small subset of the constraints and subsequently adding constraints that are violated in intermediate solutions. We start with constraints for forbidden subgraphs in the input graph. In our experiments, we consider two options to add constraints violated in an intermediate solution: adding either all violated constraints or only one.

The ILP solver uses LP relaxations to prune its search. These can be strengthened by adding constraints from Equation 1 that are violated by the LP relaxation. We generate constraints in three steps. First, we consider each node pair $\{u, v\}$ for which the relaxation solution has a value different from the input graph. We edit it, then enumerate the forbidden subgraph embeddings containing u and v, add the constraint that is most violated (i.e., whose left side is furthest below 1) and then revert the edit. Ties are broken uniformly at random. Second, we apply the same procedure to the best heuristic solution found so far. Third, we round the LP solution, i.e., an edge exists iff the corresponding variable is greater than 0.5. We then list forbidden subgraph embeddings in this rounded solution and add the corresponding most violated constraint if there is any. The listing skips forbidden subgraphs for which the corresponding constraint has already been added.

3.2 Optimizing Constraints for $\{C_4, P_4\}$-free Editing

If one forbidden subgraph can be transformed into another by a single edit, we can omit a node pair from the constraint for this subgraph. This is similar to the optimization described in Section 4.2. For a P_4, this is the node pair consisting of the two degree-one nodes. For a C_4, we can omit any one of its four edges. We always consider all four possibilities, and in the initial constraint generation as well as the basic row generation variant we add all of them. With this optimization, the constraints for C_4s and P_4s are identical.

We can also formulate a constraint for a C_4 that explicitly models that two deletions or one insertion are required:

$$\forall (u_1, u_2, u_3, u_4) \in V_G^4 : \; 0.5 \cdot \sum_{i=1}^{4} (1 - x_{u_i u_{i+1}}) + x_{u_1 u_3} + x_{u_2 u_4} \geq 1 \tag{3}$$

4 The FPT Branch-and-Bound Algorithm

The FPT algorithm [7] is a branch-and-bound algorithm. For a given maximum number of edits k, it either reports that no solution exists or returns a set of k edits. It works as follows: Find a forbidden subgraph H and branch on all possible edits in H. As H is induced, only edits in H can destroy it and thus one of these edits must be part of the solution. The algorithm is then recursively called for each branch with $k-1$ remaining edits.

Denote by p the maximum number of nodes in a forbidden subgraph. Finding H can be done trivially in time $O(n^p)$ by enumerating all subgraphs of the required size. For specific sets of forbidden subgraphs, such as $\{C_4, P_4\}$, this can be improved to $O(n+m)$ [8, 6].

Every pair of nodes in H is a valid edit. The branching factor is therefore $p \cdot (p-1)/2$. The depth of the recursion is bounded by the maximum number of edits k. The total running time is therefore in $O(p^{2k} \cdot n^p)$ for general families of forbidden subgraphs. For quasi-threshold editing the running time is $O(6^k \cdot (n+m))$. This can be improved to $O(5^k \cdot (n+m))$ by applying the optimization described in Section 4.2.

For finding the minimum number of edits k_{opt}, the algorithm needs to be executed for increasing values of k until a solution is returned. For a branching factor of 2 or larger, the running time of all $k < k_{\text{opt}}$ together is at most the running time for k_{opt}. Thus the total running time is dominated by the running time for k_{opt}.

In the following, we describe several optimizations to reduce the number of explored branches in practice. We describe existing techniques for avoiding redundant exploration of branches (Section 4.1), for skipping certain branches (Section 4.2) as well as lower bounds (Section 4.3). We introduce a novel local search lower bound (Section 4.4), optimized branching choices (Section 4.5), early pruning of branches (Section 4.6) and a simple parallelization (Section 4.7). The appendix of the full version [13] provides in-depth implementation details.

4.1 Avoiding Redundancy

Damaschke [9] proposes to block node pairs to list every solution exactly once. When spawning a search tree node x through editing a node pair, it is neither useful to undo that edit in the sub-search-tree rooted at x, nor is it useful to perform the edit in sibling search trees. While this has been introduced for cluster editing, the technique can be applied to arbitrary $\mathcal{F}$-free editing problems. We explain this technique in detail in the full version [13].

4.2 Skip Forbidden Subgraph Conversion

▶ **Lemma 1.** *If each forbidden subgraph $A \in \mathcal{F}$ can be transformed into another one $B \in \mathcal{F}$ by one edit, the branching factor of the FPT algorithm can be reduced from $\binom{p}{2}$ to $\binom{p}{2} - 1$.*

There is an edit that transforms a P_4 into a C_4. Clearly, this edit can be skipped. Further, there are four edge deletions that transform a C_4 into a P_4. One of these can be skipped [22]. We can choose which one, but as any pair of two edge deletions eliminates the forbidden subgraph, skipping more than one of them might eliminate a necessary branch. Since the branching factor is reduced, this decreases the worst-case running time from $O(6^k \cdot (n+m))$ to $O(5^k \cdot (n+m))$ for quasi-threshold editing.

4.3 Existing Lower Bound Approaches

At each branching node, we have a certain number k of edits left. If we can show that the graph needs at least $k+1$ edits, we do not need to explore further branches below that node. Lower bounds have been used for cluster editing [4, 16] and $\{P_5\}$-free editing [5]. Commonly, they are based on an LP relaxation of the ILP [16], or on a disjoint packing argument [5, 16].

Subgraph Packing. A *node-pair disjoint subgraph packing* P is a set of induced forbidden subgraphs that do not share a node pair. As no edit can eliminate more than one subgraph, $|P|$ is a lower bound on the number of edits required. Taking the previously mentioned optimizations into account, we can include more subgraphs in P by allowing to share blocked node pairs, as they cannot be edited. Further, for each forbidden subgraph a node pair that transforms it into another forbidden subgraph may be shared. In the case of $\mathcal{F} = \{C_4, P_4\}$, the pair of degree-1 nodes of an induced P_4 can be shared. For C_4, we can choose any edge to share, but it remains the same as long as the C_4 is in the packing.

Finding such a packing can be modeled as an independent set problem [16]. The forbidden subgraphs are nodes and every pair of forbidden subgraphs that shares a non-shareable node pair is connected by an edge. A natural greedy heuristic for independent sets is to iteratively add the node that has the smallest degree and then remove all its neighbors from the graph. This can be implemented in linear time by splitting nodes into buckets according to their degree (see e.g. [1]). This heuristic has also been used to calculate lower bounds for cluster editing [16]. We are not aware of complexity results of the independent set problem on this special graph class.

In our experiments, we evaluate three bounds based on subgraph packing: 1) A *basic* bound that iteratively adds subgraphs to the packing as they are found. 2) An incremental version of 1) that *updates* the packing as the graph is modified in the branch-and-bound algorithm. After applying an edit, we remove the subgraph that contains the edited node pair. After both editing and blocking, we enumerate and add subgraphs to the bound until it is maximal. 3) A greedy bound based on the *minimum degree* heuristic. In contrast to the first two, this requires storing all forbidden subgraphs. To avoid this in trivial cases, we first apply 2) to the previous bound and only compute a new packing if this fails to prune the branch.

LP relaxation. The optimal solution of the LP relaxation provided in Section 3 is an upper bound for the node-pair-disjoint packing problem. This can be shown by considering an LP with just the constraints that correspond to the subgraphs in a packing. Each subgraph in the packing is a node-induced subgraph of G. Therefore, the terms on the left side of its corresponding constraint appear in the objective function exactly as they appear in the constraint, confer Equations 1 and 2. Each term in the objective function is at least 0, and each group of terms corresponding to a fulfilled constraint sums to at least 1. Since the packing is node-pair disjoint, the constraints do not share any variables and thus groups do not overlap. Therefore, the objective value is at least the number of subgraphs in the packing. Adding more constraints can only increase the objective and thus improve the bound. We can also model blocked node pairs by replacing the corresponding variable by its value. The variables in the constraints are then disjoint again and thus the same argument applies.

4.4 Local Search Lower Bound

We propose a lower bound based on a subgraph packing that is computed using an adaptation of the 2-improvements local search heuristic [1] for independent sets. Our local search starts with an initial packing and works in rounds. In each round, it iterates over all forbidden subgraphs in the packing and tries to replace one by two forbidden subgraphs. If this is not possible, it tries to replace one by one. Preliminary experiments have shown that choosing this replacement from those candidates which cover the fewest other forbidden subgraphs leads to significantly higher bounds than considering all candidates. We also found that using this strategy only 70% of the time and otherwise choosing a random replacement

is even better. We repeat this procedure until in five consecutive rounds only one-by-one replacements were found. We also terminate the search if the packing remains completely unchanged in a round, or if the packing is large enough to prune the current branch in the search tree. To make this efficient, we approximate the number of forbidden subgraphs that are covered by a certain forbidden subgraph H, by adding up the number of forbidden subgraphs each node pair of H is part of. For the latter we can efficiently maintain counters.

The initial packing is computed with the basic greedy bound. For recursive calls, we update the previous bound as discussed above, before employing local search.

4.5 Branch on Most Useful Node Pairs

We can choose any forbidden subgraph for branching on its possible edits, e.g., the first we find. If there is a forbidden subgraph with only one non-blocked node pair, we choose it, as this will lead to just one recursive call. Otherwise, the first node pair we try to edit should ideally lead to a solution, or blocking the edit should prune the search. We propose to prefer forbidden subgraphs whose non-blocked node pairs are part of many other forbidden subgraphs. Then, a single edit can eliminate many forbidden subgraphs (possibly leading to a solution) and blocking the node pairs allows adding many subgraphs to the lower bound. For each forbidden subgraph, we sort its non-blocked node pairs in decreasing order by the number of forbidden subgraphs that contain the respective node pair. The edits of the selected forbidden subgraph are also tried in this order. We select the subgraph to branch on using a lexicographical ordering on these counts. The last node pair is excluded, as there are no branches left to prune. Additionally, if two subgraphs have identical count sequences (up to the length of the shorter one), we prefer the subgraph with the shorter sequence.

4.6 Prune Branches Early

Normally, we attempt to prune a branch after applying an edit and descending into recursion. With the optimization from Section 4.1, the edited node pair of a recursive call remains blocked after returning from recursion. We update the lower bound to consider this blocked node pair. If the new lower bound already exceeds the remaining number of edits, we can directly prune all subsequent recursive calls, instead of pruning them individually. There are two cases for which we skip the bound update to save running time: If there is only one subsequent recursive call, as we would only prune a single branch, and if the blocked node pair is only part of a single forbidden subgraph, as it cannot yield a better lower bound.

4.7 Parallelization

The algorithm can be parallelized by letting different cores explore different branches. Due to our optimizations, not every branch needs the same running time. Therefore, just executing the first branches in parallel is not scalable. Instead, we use a simple work stealing algorithm. Whenever a thread has fully explored its branch, it steals a branch on the highest available level from another thread and further explores it.

5 Experimental Evaluation

In the appendix of the full version [13] we discuss implementation details. The C++ source code[1] of all discussed variants is available online. We use the C++ interface of Gurobi [15] to solve ILPs and LPs. We evaluate our algorithms on a set of 3964 graphs that are connected components of the COG protein similarity data[2] that has already been used for the evaluation of cluster editing algorithms [24, 3]. The dataset consists of a similarity matrix for each graph. We treat all non-negative scores as edges. Unless stated otherwise, we restrict our evaluation to the 716 graphs that require at least 20 edits. On the 3248 excluded graphs, the maximum running time is less than 0.43 seconds for the FPT algorithm using our local search lower bound. Of these graphs, 1666 require no edits at all. Further, we evaluate our algorithms on a set of 5 small social networks that were already considered by Nastos and Gao [23], namely `karate` [25], `grass_web` [10], `lesmis` [18], `dolphins` [20], and `football` [12].

All experiments were performed on systems with two 8-core Intel Xeon E5-2670 (Sandy Bridge) processors and 64 GB RAM. We set a global time limit of 1000 seconds. Experiments comparing just FPT variants were executed on 16 different node orders, running 16 node orders in parallel. Due to the memory requirements of Gurobi, this is not feasible for the ILP and the LP bound. For these variants, we run just one instance at a time. For experiments involving ILP variants, we also limit the experiments to 4 node orders, and, for better comparability, we run one instance at a time also for the FPT comparison runs in Figure 4. By default, all algorithms terminate at the first found solution, as the ILP is unable to enumerate solutions. Variants with the suffix -All enumerate all solutions. Further, variants with the suffix -MT are parallelized using 16 cores.

5.1 Variants of the FPT Algorithm

The baseline branching strategy -F uses the first found forbidden subgraph. Our `Most` branching strategy from Section 4.5 is denoted by -M, additional early pruning by -MP. The basic greedy bound is denoted by -G, the incremental update bound by -U, the min-degree heuristic by -MD, our local search lower bound by -LS, and LP relaxations by -LP. The comparison includes the nine variants FPT-G-F-All, FPT-G-MP-All, FPT-U-MP-All, FPT-MD-F-All, FPT-MD-MP-All, FPT-LP-MP-All, FPT-LS-F-All, FPT-LS-M-All and FPT-LS-MP-All.

Figure 2 shows how many of the COG dataset instances can be solved within a certain time limit and with a certain number of recursive calls – added over all k's. Additional lower bound calls due to -MP count extra. An instance is a single node id permutation of a graph, i.e., every graph is counted 16 times. Of the 716 graphs we are able to solve 547 within the 1000 second time limit. Below, we also compare calls and running times per instances.

For comparing branching strategies, we fix the local search algorithm -LS as the lower bound. The median factor of additional calls needed by -M over -MP is 1.9 and by -F over -MP is 3.36, restricted to instances solved by both algorithms. While the median speedup of -MP over -F is 3.11, it is just 1.06 over -M. On 5% of the instances, the speedup is at least 56.62 and 1.24, respectively. This shows that for -M the improvement in the number of calls directly leads to similar running time improvements, while early pruning just reduces calls.

For comparing lower bound algorithms, we fix -MP as the branching strategy. There is an inherent trade-off between the number of recursive calls and the time spent per call, with

[1] `https://github.com/kit-algo/fpt-editing`
[2] `https://bio.informatik.uni-jena.de/data/#cluster_editing_data`

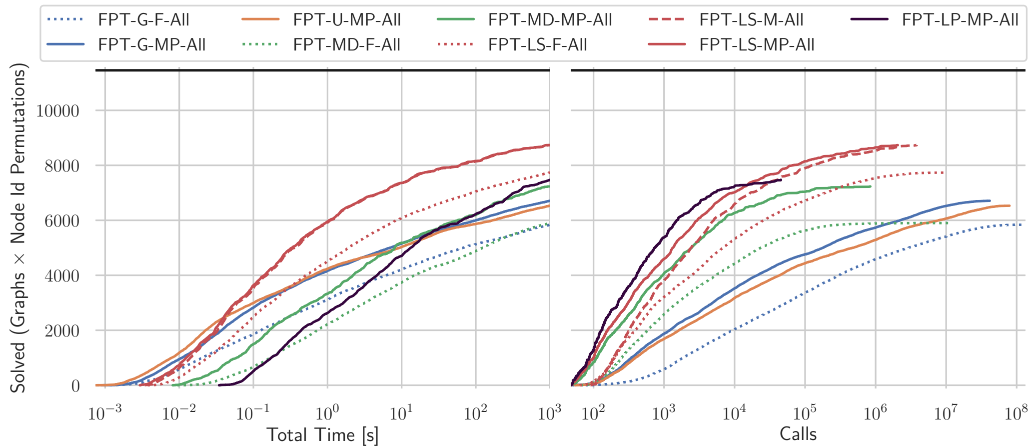


Figure 2 Number of permutations of graphs of the COG dataset that require at least 20 edits and can be solved within a certain total running time / with a certain number of recursive calls (and extra lower bound updates for -MP). The horizontal black line indicates the total number of graphs and node permutations that require 20 or more edits, including unsolved instances.

a sweet spot that gives the best overall running time. The basic greedy bounds need 10 to 24 times as many calls as the other bounds in the median. The recomputed greedy bound -G is slightly better than the updated one -U, -LP is the best, followed by -LS and -MD.

Nonetheless, for very small time limits, -U solves the highest number of instances. For larger time limits, reducing the number of calls pays off, though not at any cost. The median speedup of min-degree over the LP is 2.16 while needing 47% more calls in the median. Local search avoids their substantial memory overhead and spends significantly less time per call than both. It needs 83% of the calls of -MD while being a factor of 12.36 faster in the median. It is never slower, and on 5% of the instances even more than 137 times faster than -MD.

Comparing the state-of-the-art FPT-MD-F-All algorithm to our FPT-LS-MP-All algorithm, we need 4.33 times less calls and are 46.06 times faster in the median. We are never slower, on 75% of the instances more than 16 times and on 5% of the instances more than 1044 times faster. In conclusion, our local search lower bound gives high-quality bounds while being fast. Our branching rules reduce the running time by another small factor while early pruning mainly reduces the number of calls. Overall, we achieve a speedup of one to three orders of magnitude over the state-of-the-art.

5.2 Parallelization

The left part of Figure 3 reports the speedup of FPT-LS-MP-All-MT over its sequential counterpart FPT-LS-MP-All, the sequentially fastest variant on the COG dataset. We show the speedup with 1, 2, 4, 8 and 16 cores in comparison to the number of recursive calls and lower bound calculations. For each graph and permutation, we plot the speedup on the last value of k for which the sequential version of the algorithm terminated within the time limit. With only few recursive calls we cannot expect a good speedup. For a high number of recursive calls, FPT-LS-MP-All-MT achieves almost perfect speedup for all numbers of cores on many graphs. As the algorithm is executed with increasing values of k, for some graphs only the last value of k needs a high number of calls and thus the overall speedup is not perfect even though in sum the number of calls is high.

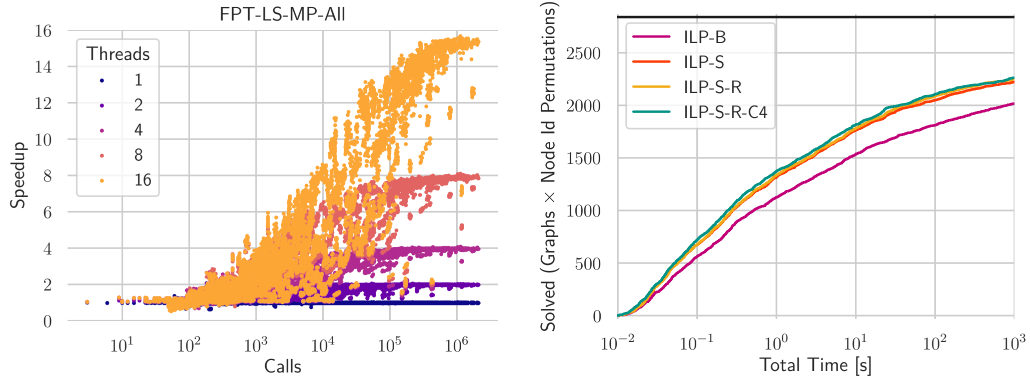


Figure 3 Speedup of FPT-LS-M-All-MT and comparison of the different ILP variants on 16 (left) and 4 (right) node id permutations of the 716 COG graphs that require at least 20 edits.

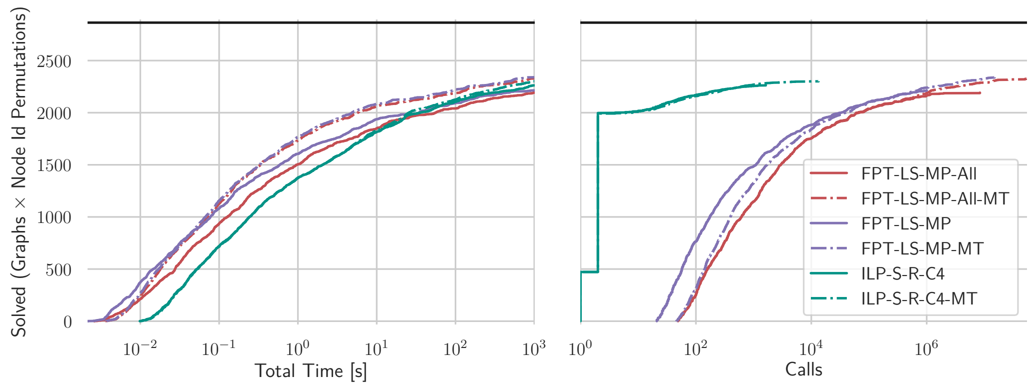


Figure 4 Comparison of the ILP to the FPT algorithm on 4 node id permutations of the 716 COG graphs that require at least 20 edits.

5.3 Variants of the ILP

Figure 3 (right) shows the impact of the different optimizations on the ILP, when enabled one after another. We denote adding only one violated constraint by -S, adding constraints during relaxations by -R, and specialized C_4 constraints by -C4. The baseline is ILP-B, where row generation always adds all violated constraints for intermediate solutions.

In the median, ILP-S is just 5% faster than ILP-B. While on 95% of the instances it is at most 20% slower, it is more than 44 times faster on 5% of them, which explains the gap in Figure 3. ILP-S-R is not faster in the median, but on 95% of the instances at most 12% slower and on 5% it is at least 73% faster. The C_4 constraints make the ILP 12% faster in the median, at most 26% slower on 95% and at least 95% faster on 5% of the instances. With all optimizations, the ILP solves 568 graphs. We also tried providing a heuristic solution from QTM [6] to Gurobi, but the improvement was even smaller and disappeared in parallel.

Figure 4 compares the best ILP and FPT algorithms with and without -MT in terms of running time and recursive calls. For the FPT algorithm, stopping at the first solution is not slower on 95%, more than 52% faster on 50% and more than 3 times faster on 5% of the instances. Multi-threading incurs a measurable overhead. Compared to FPT-LS-MP-All,

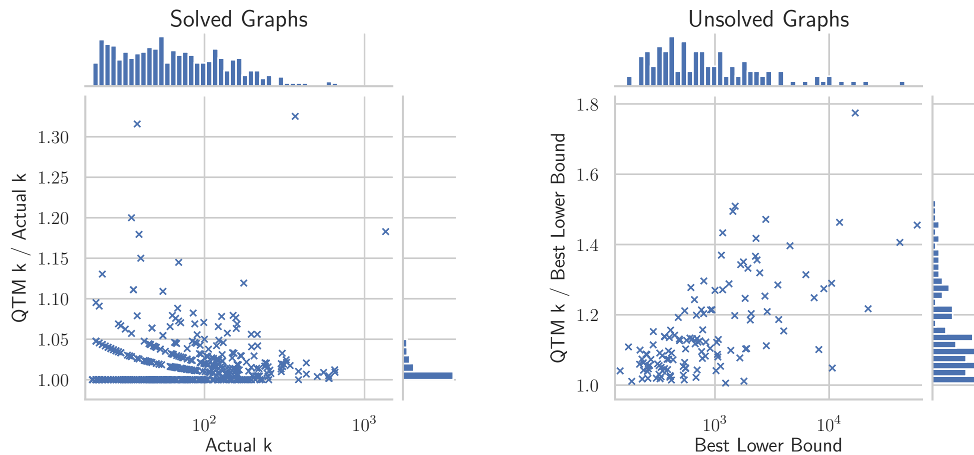


Figure 5 Comparison of heuristic solutions of QTM and the exact number of edits k for solved graphs (left) or the best lower bound for unsolved graphs (right) achieved by the FPT algorithm or the ILP. For readability, we exclude one solved graph at $k = 64$, where QTM needed 202 edits.

FPT-LS-MP-All-MT is at most 16% slower on 95%, 78% faster in the median and more than 12 times faster on 5% of the instances. When stopping at the first solution, this decreases to 24% slower, 1% faster and 10 times faster, as more branches that do not lead to a solution are explored in multi-threaded mode. FPT-LS-MP-MT is still 4% faster than FPT-LS-MP-All-MT in the median, at most 3% slower on 95% and at least 68% faster on 5% of the instances.

The parallel ILP is at most 5% slower on 95%, as fast in the median and more than 52% faster on 5% of the instances than the sequential ILP. Thus, the parallelization helps the FPT algorithm more than the ILP. A likely cause is that Gurobi needs much less search nodes than the FPT algorithm which offer less potential for parallelism – on 50% of the instances at least 185 times less, and on many graphs even just one or two, see Figure 4.

The speedup of FPT-LS-MP over ILP-S-R-C4 is at least 0.59 on 95%, 3.25 in the median and at least 10.72 on 5% of the instances. For FPT-LS-MP-All, this decreases to 0.29, 2.10 and 7.02. In parallel, the speedups are 1.09, 3.41 and 16.45 for all solutions, and 1.34, 3.67 and 18.14 for the first solution. Single-threaded, the ILP solves more instances within 1000 seconds than the FPT algorithm, indicating that for difficult instances better bounds are more important. Overall, the FPT algorithm is often faster than the ILP, in particular in parallel and even when listing all solutions.

5.4 Comparison to QTM

Figure 5 compares the results of the heuristic Quasi-Threshold Mover (QTM) [6] with exact results for solved and the best lower bounds for unsolved graphs. We use the maximum value of k achieved for any permutation by FPT-LS-MP-MT and by ILP-S-R-C4 with and without -MT. If any of the algorithms solved the graph, we list it in the left part, otherwise in the right part. For QTM, we report the minimum k that QTM found over 16 runs. Again, the plot excludes 3248 graphs that require less than 20 edits. Of those, QTM solved 3172 exactly, 56 with offset 1, 15 with offset 2 and 5 with offset 3. Of the remaining graphs, 588 are solved and 128 are unsolved. Of the solved graphs, QTM solved 319 graphs exactly. For none of the unsolved graphs, QTM matches the lower bound. For 95% of the 716 graphs, QTM needs at most 1.22 times the edits of the exact solution or the lower bound.

Table 1 Overview of the social network graphs. Using the algorithms FPT-LS-MP and ILP-S-R-C4 with 1 and 16 cores, we report the maximum k that finished within 1000 seconds, and the minimum time over all permutations that is needed to find the first solution. In the case of `football`, we report the time needed to show that there is no solution with that k.

			FPT				ILP			
			1 core		16 cores		1 core		16 cores	
Graph	n	m	k	Time [s]	k	Time [s]	k	Time [s]	k	Time [s]
karate	34	78	21	0.01	21	0.01	21	0.02	21	0.03
lesmis	77	254	60	0.17	60	0.13	60	0.96	60	0.97
grass_web	75	113	34	1.81	34	0.21	34	2.91	34	2.83
dolphins	62	159	70	126.54	70	18.57	70	23.81	70	12.10
football	115	613	223	929.55	228	649.94	235	1000.01	237	1000.05

5.5 Social Network Instances

Table 1 shows an overview of the social networks with results for FPT-LS-MP and ILP-S-R-C4. Both solve `karate` and `lesmis` in less than a second, and `grass_web` within 3 seconds, with the FPT algorithm being faster. Even though `lesmis` is both larger than `grass_web` and requires 60 edits instead of 34, both algorithms are significantly slower on `grass_web`. This shows that their performance depends on the specific structure of the graph and not just the graph size and k. For `dolphins`, the ILP is faster than the FPT algorithm. For all graphs, the FPT algorithm scales better with the number of cores. None of the algorithms can solve the `football` network. We show that there is no solution for $k \leq 223$, $k \leq 228$ using the FPT algorithm with 1 or 16 cores respectively, and $k \leq 235$, $k \leq 237$ using the ILP with 1 or 16 cores respectively. The previously best known upper bound was 251, computed with QTM [6] in 2.5ms. In 1000 seconds, the ILP shows a new upper bound of 250. For the smallest three social networks, we verify that the best heuristic solutions in [6] are exact. QTM needs 72 edits on `dolphins`, whereas 70 edits are optimal. The appendix of the full version [13] contains a detailed analysis of the solution space with a focus on the community detection application.

6 Conclusion

We have introduced optimizations for two different approaches to solving any $\mathcal{F}$-free edge editing problem. We evaluate our optimizations for the special case of quasi-threshold editing on a set of 716 protein interaction graphs. For the first approach, the FPT algorithm, we show that the combination of good lower bounds with careful selection of branches allows to reduce the running time by one to three orders of magnitude for 75% of the instances. For the second approach, an ILP, we evaluate several variants of row generation and show that they achieve small speedups. We show that the FPT algorithm is slightly faster than the ILP, with a larger margin in parallel, and it can easily enumerate all optimal solutions. For the heuristic editing algorithm QTM, we show that on 95% of the instances, it needs at most 22% more edits than our exact solutions or lower bounds indicate.

Comparing the structure of exact vs. heuristic solutions might give further insights how to improve heuristics. Exact FPT algorithms could be further improved by better bounds, possibly based on LP relaxations. As the COG benchmark set actually contains edit costs, an extension of our optimizations to the weighted editing problem could be investigated.

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. Sebastian Böcker and Jan Baumbach. Cluster Editing. In *Proceedings of the 9th Conference on Computability in Europe (CiE'13)*, volume 7921 of *Lecture Notes in Computer Science*, pages 33–44. Springer, 2013. doi:10.1007/978-3-642-39053-1_5.
3. Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. A fixed-parameter approach for Weighted Cluster Editing. In *Proceedings of the 6th Asia-Pacific Bioinformatics Conference (APBC 2008)*, volume 6, pages 211–220, 2008. doi:10.1142/9781848161092_0023.
4. 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.
5. Felix Bohlmann. Graphclustern durch Zerstören langer induzierter Pfade. Bachelor thesis, TU Berlin, 2015. URL: http://fpt.akt.tu-berlin.de/publications/theses/BA-felix-bohlmann.pdf.
6. Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. Fast Quasi-Threshold Editing. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science*, pages 251–262. Springer, 2015. doi:10.1007/978-3-662-48350-3_22.
7. Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171–176, May 1996. doi:10.1016/0020-0190(96)00050-6.
8. Frank Pok Man Chu. A simple linear time certifying LBFS-based algorithm for recognizing trivially perfect graphs and their complements. *Information Processing Letters*, 107(1):7–12, June 2008. doi:10.1016/j.ipl.2007.12.009.
9. Peter Damaschke. Fixed-Parameter Enumerability of Cluster Editing and Related Problems. *Theory of Computing Systems*, 46(2):261–283, 2008. doi:10.1007/s00224-008-9130-1.
10. Hassan Ali Dawah, Bradford A. Hawkins, and Michael F. Claridge. Structure of the Parasitoid Communities of Grass-Feeding Chalcid Wasps. *Journal of Animal Ecology*, 64(6):708–720, 1995. doi:10.2307/5850.
11. Pål Grønås Drange and Michał Pilipczuk. A Polynomial Kernel for Trivially Perfect Editing. *Algorithmica*, 80(12):3481–3524, December 2017. doi:10.1007/s00453-017-0401-6.
12. Michelle Girvan and Mark E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Science of the United States of America*, 99(12):7821–7826, 2002. doi:10.1073/pnas.122653799.
13. Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. Engineering exact quasi-threshold editing. *arXiv e-prints*, 2020. arXiv:2003.14317.
14. Martin Grötschel and Yoshiko Wakabayashi. A cutting plane algorithm for a clustering problem. *Mathematical Programming*, 45(1-3):59–96, 1989. doi:10.1007/BF01589097.
15. Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2020. URL: http://www.gurobi.com.
16. Sepp Hartung and Holger H. Hoos. Programming by Optimisation Meets Parameterised Algorithmics: A Case Study for Cluster Editing. In *Proceedings of the 9th International Conference on Learning and Intelligent Optimization*, Lecture Notes in Computer Science, pages 43–58. Springer, 2015. doi:10.1007/978-3-319-19084-6_5.
17. Marc Hellmuth, Nicolas Wieseke, Marcus Lechner, Hans-Peter Lenhof, Martin Middendorf, and Peter F. Stadler. Phylogenomics with paralogs. *Proceedings of the National Academy of Science of the United States of America*, 112(7):2058–2063, 2015. doi:10.1073/pnas.1412770112.
18. Donald E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing*. Addison-Wesley, 1993.

19 Yunlong Liu, Jianxin Wang, Jie You, Jianer Chen, and Yixin Cao. Edge deletion problems: Branching facilitated by modular decomposition. *Theoretical Computer Science*, 573:63–70, 2015. doi:10.1016/j.tcs.2015.01.049.

20 David Lusseau, Karsten Schneider, Oliver Boisseau, Patti Haase, Elisabeth Slooten, and Steve Dawson. The Bottlenose Dolphin Community of Doubtful Sound Features a Large Proportion of Long-Lasting Associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405, September 2004. doi:10.1007/s00265-003-0651-y.

21 James Nastos. *Utilizing graph classes for community detection in social and complex networks*. PhD thesis, University of British Columbia, 2015. doi:10.14288/1.0074429.

22 James Nastos and Yong Gao. A Novel Branching Strategy for Parameterized Graph Modification Problems. In *Proceedings of the 4th International Conference on Combinatorial Optimization and Applications*, volume 2 of *Lecture Notes in Computer Science*, pages 332–346. Springer, 2010. doi:10.1007/978-3-642-17461-2_27.

23 James Nastos and Yong Gao. Familial groups in social networks. *Social Networks*, 35(3):439–450, July 2013. doi:10.1016/j.socnet.2013.05.001.

24 Sven Rahmann, Tobias Wittkop, Jan Baumbach, Marcel Martin, Anke Truß, and Sebastian Böcker. Exact and Heuristic Algorithms for Weighted Cluster Editing. In *Proceedings of the 6th Annual International Conference on Computational Systems Bioinformatics (CSB 2007)*, volume 6, pages 391–401, 2007. doi:10.1142/9781860948732_0040.

25 Wayne W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33:452–473, 1977. doi:10.1086/jar.33.4.3629752.