Karlsruhe Institute of Technology

Engineering FPT-based Edge Editing Algorithms

Master Thesis of

Sven Zühlsdorf

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Michael Hamann, M. Sc.
	Dipl.-Inform. Ben Strasser

Time Period: 1st April 2017 – 30th September 2017

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 30th September 2017

Abstract

In this work we consider exact algorithms to solve the $\mathcal{F}$-free Edge Editing Problem, where $\mathcal{F}$ is a set of forbidden subgraphs. We develop the Redundant Editing Algorithm as improvement over the FPT algorithm introduced by [Cai96]. While not improving the time complexity of the algorithm, we prove and evaluate its practical benefits. We find the Redundant Editing Algorithm to be several orders of magnitude faster, increasing with higher complexity of the input graph.

Edge editing algorithms based on [Cai96] require a strategy for selecting forbidden subgraphs. We introduce *single edge editing* as an alternative to such strategies. Under specific circumstances, the Edge Editing Problem for a given $\mathcal{F}$ can be solved in $O(4^k \operatorname{poly}(n))$ using single edge editing,

In our evaluation, using $\mathcal{F} = \{P_4, C_4\}$ as set of forbidden subgraphs, we find we are able to solve three of the graphs used by [NG13] within minutes. Two of them were previously unsolved.

Deutsche Zusammenfassung

In dieser Arbeit betrachten wir Algorithmen die das $\mathcal{F}$-free Edge Editing Problem lösen, wobei $\mathcal{F}$ eine Menge an verbotenen Teilgraphen bezeichnet. Wir entwickeln den Redundant Editing Algorithm als Verbesserung des von [Cai96] vorgestellten FPT-Algorithmus. Die Zeitkomplexität des Algorithmus wird nicht verbessert, jedoch beweisen und evaluieren wir die praktischen Vorteile. Wir stellen fest, dass der Redundant Editing Algorithm mehrere Größenordnungen schneller ist. Dieser Vorteil wächst mit zunehmender Komplexität des Eingabegraphen.

Kanteneditierungsalgorithmen basierend auf [Cai96] benötigen eine Strategie zur Auswahl verbotener Teilgraphen. Wir stellen *single edge editing* als Alternative zu solchen Strategien vor. Unter bestimmten Bedingungen können wir damit das Kanteneditierungsproblem für ein gegebenes $\mathcal{F}$ in $O(4^k \operatorname{poly}(n))$ lösen.

In unserer Evaluation, in der wir $\mathcal{F} = \{P_4, C_4\}$ als Menge von verbotenen Teilgraphen verwenden, stellen wir fest, dass wir drei der von [NG13] benutzten Graphen innerhalb von Minuten lösen können. Zwei dieser Graphen waren bisher ungelöst.

Contents

1. Introduction

Given a social network, in which persons are connected if they are friends with each other, are there groups or *communities* of closely connected friends? Finding a suitable formal definition of a *community* and identifying communities based on the topology of a social network are both important and highly discussed topic in multiple disciplines, including computer science [For10, FH16].

A traditional approach for identifying communities is *cluster editing*: By inserting and removing edges, a graph is transformed into connected components that each form a single clique. A graph with that property is also called P_3-free, as it does not contain a path of three vertices as induced subgraph. The problem of finding a minimal set of edges that satisfies this property is NP-complete.

Recently however, other approaches were introduced: [NG13] defines communities as connected components that are *quasi-theshold graphs*, i.e. they contain neither P_4 nor C_4 as induced subgraph. The corresponding editing problem is NP-complete as well. They introduce a heuristic algorithm, but it is difficult to judge the quality of this heuristic, as exact solutions for most real world graphs are not known.

Other variants include *co-graph editing* (P_4-free) or only allowing deleting edges, but not inserting them. Only allowing deletions reduces the complexity of the problem as fewer choices have to be considered.

Generalising, when allowing both insertion and deletion, these problems are called $\mathcal{F}$-free edge editing, where $\mathcal{F}$ defines the set of forbidden subgraphs. There is an FPT-algorithm by [Cai96] producing exact solutions, but due to its exponential time complexity it is quite slow. [Boh15] introduces lower bounds as speedup technique and finds them to be a major improvement.

In this work we develop additional speedup techniques and evaluate them on both real world and synthetic graphs. While we do not improve the time complexity, we find that adding our techniques improves the running time by several orders of magnitude. The speedup increases with the difficulty of the input graph. As noted by [NG13], a given graph may have multiple solutions and they ask how the detected communities vary across those solutions. We are able to solve three of the graphs [NG13] used exactly and answer this question for these graphs.

We now give the outline of this work. In the remainder of this chapter we introduce the basic terms and definitions used in this work. In Chapter 2, we formally introduce the

$\mathcal{F}$-free Edge Editing Problem. We then discuss the basic branch-and-bound algorithm, followed by our improvements. Afterwards, we introduce lower bounds to quickly abort searches that cannot lead to a solution. Finally, we present various strategies to select forbidden subgraphs, before designing our own strategy. In Chapter 3, we compare the algorithms and strategies presented in Chapter 2 with each other and analyze the results. Finally, in Chapter 4, we summarize our results and give an outlook posing open questions for further work.

1.1 Preliminaries

In this section we establish basic terms used in this thesis with definitions similar to those found in literature. Derivations from these definitions in the following chapters will be motivated and explained when first needed.

We define a *graph* $G = (V, E)$ as a tuple of a set of *vertices* V and a set of *edges* E. An *edge* e is a pair of vertices $u, v \in V$. In a *directed graph* the order of these vertices is important and we write $e = (u, v) \neq (v, u)$ is an edge from u to v. In an *undirected graph* the order does not matter and we write $e = \{u, v\} = \{v, u\}$. The following definitions for *neighborhood*, *simple graph* and *path* are for undirected graphs, for directed graphs the definitions are analogous. The *neighborhood* $N(u) = \{v \in V \mid \{u, v\} \in E\}$ of a vertex $u \in V$ is the set of vertices that share an edge with u. In a *simple graph* there exists at most one edge between any two vertices ($e = \{u, v\}, f = \{u, v\} \implies e = f$) and no vertex is in its own neighborhood ($u \notin N(u), u \in V$). A *path* $p = (u_0, u_1, \ldots, u_n)$ from u_0 to u_n is a sequence of vertices $u_i \in V$ where all pairs of adjacent vertices have an edge connecting them ($\{u_i, u_{i+1}\} \in E, 0 \leq i < n$). If such a path exists, we say u_n is reachable from u_0. If the last vertex in a path is the same as the first ($u_n = u_0$) and the path contains more than a single vertex ($n > 0$), we call the path a *cycle*. A graph that contains no cycles is called an *acyclic graph*.

A *subgraph* $H = (V_H, E_H)$ of a graph $G = (V_G, E_G)$ uses a subset of G's vertices and edges ($V_H \subseteq V_G$, $E_H \subseteq (E_G \cap (V_H \times V_H))$). An *induced subgraph* $H = (S, E_H)$ of a graph $G = (V_G, E_G)$ contains all edges of G whose vertices are in S ($E_H = E_G \cap (S \times S)$) and is denoted by $G[S]$. Two graphs $G = (V_G, E_G), H = (V_H, E_H)$ are *isomorphic* if both graphs are equal except for renaming of vertices.

We call a graph $G = (V, E)$ F-free if and only if there is no subset $S \subseteq V$ of vertices whose induced subgraph $G[S]$ is isomorphic to a graph F. As a shorthand we allow $\mathcal{F}$ to be a set of graphs and say a graph G is $\mathcal{F}$-free if G is F-free for all $F \in \mathcal{F}$.

A set $L \subseteq V \times V$ of *edits* for a graph $G = (V, E)$ is a collection of edges. Applying L to G produces an edited graph $G_L = (V, E \triangle L)$, where $E \triangle L := (E \setminus L) \cup (L \setminus E)$ denotes the symmetric difference. As a shorthand we define $G \triangle L := (V, E \triangle L)$. By *editing* or *toggling* an edge e in a graph G we will refer to applying the edit $\{e\}$.

A *tree* T is a simple acyclic directed graph with a special vertex r called *root*, that, for all vertices, contains exactly one path from r to that vertex. A *subtree* T' rooted at a vertex r' is an induced subgraph of T over all vertices that are reachable from r', i.e., $T' = T[\{u \mid u \in V \wedge u \,\text{reachable from}\, r'\}]$. We call a path that starts at the root r a *branch* of T. The *height* of a tree is the length of the longest branch.

Unless explicitly noted otherwise, we will refer to a simple undirected graph when using the word graph in this thesis.

2. Editing Strategies

In this chapter, we formally introduce the $\mathcal{F}$-free Edge Editing Problem. We then discuss the basic branch-and-bound algorithm, followed by our improvements. Afterwards, we introduce lower bounds to quickly abort searches that cannot lead to a solution. Finally, we present various strategies to select forbidden subgraphs, before designing our own strategy.

2.1 Problem definition

Formally, we define the $\mathcal{F}$-Free Edge Editing Search Problem as follows:

$\mathcal{F}$-Free Edge Editing Search Problem:
Input: A graph G, a set of graphs $\mathcal{F}$
Question: Find a set L of edits so that $G \triangle L$ is $\mathcal{F}$-free and $|L|$ is minimal.

In most cases it is more convinient to talk about the corresponding decision problem, which we also define here:

$\mathcal{F}$-Free Edge Editing Decision Problem:
Input: A graph G, a set of graphs $\mathcal{F}$, a non-negative integer k
Question: Can G be edited to be $\mathcal{F}$-free using at most k edits?

The complexity of the $\mathcal{F}$-Free Edge Editing Decision Problem varies depending on $\mathcal{F}$, as illustrated by the following examples: If $\mathcal{F}$ contains the path with two vertices P_2, solving the $\mathcal{F}$-Free Edge Editing Decision Problem requires m edits to remove all edges, which can be done in linear time. In case of $\mathcal{F} = \{P_3, C_3\}$ the $\mathcal{F}$-Free Edge Editing Decision Problem is equivalent to the Maximum Matching Problem solvable in polynomial time. For $\mathcal{F} = \{P_l, C_{l_1}, \dots, C_{l_x}\}$ with $l \geq l_1 > \cdots > l_x \geq 4$ and $x \in \mathbb{N}_0$ the $\mathcal{F}$-Free Edge Editing Decision Problem is NP-complete as shown in [Sch15]. If $\mathcal{F}$ is finite, [Cai96] proves the existence of a Fixed Parameter Tractable (FPT) algorithm in the number of allowed edits k.

In this work we solve the search problem by repeatedly asking the decision problem. Starting with a single allowed edit, we increase k until the $\mathcal{F}$-Free Edge Editing Decision Problem returns “true”. At that point a solution for the search problem has been implicitly constructed by the algorithms we present in the next sections.

2.2 Basic Editing Algorithm

In this section we describe the basic editing algorithm introduced by [Cai96].

Given a graph G and a number k of allowed edits for which the $\mathcal{F}$-free Edge Editing Decision Problem shall be solved, we can construct a search tree whose vertices represent graphs and whose edges represent a single edit made between a pair of vertices. The root of this search tree is the graph G. We limit the height of the search tree to $k+1$. If a graph G' represented by a vertex u in this search tree is $\mathcal{F}$-free, G can be edited to be $\mathcal{F}$-free with at most k edits. The set of edits needed to make G $\mathcal{F}$-free are represented by the path from the root of the search tree to u. Therefore, when using an approach based on exploration of this search tree, finding a solution for the $\mathcal{F}$-free Edge Editing Decision Problem implicitly leads to a solution for the $\mathcal{F}$-free Edge Editing Search Problem. We note the following:

Lemma 2.1. *At least one pair of vertices of each forbidden subgraph $G[S]$ must be modified in order to destroy that subgraph.*

Proof. Trivial. □

Thus, for each graph G' represented by a vertex in the search tree, it is sufficient to only explore editing the pairs of vertices of one forbidden subgraph $G'[S]$. The Basic Editing Algorithm shown in Algorithm 2.1 applies this idea: The algorithm first searches for an instance of a forbidden subgraph. If no forbidden subgraph is found, the algorithm terminates by returning “true” since no editing is required. Otherwise, a forbidden subgraph, induced by a set S of vertices, was found. If no further edits are allowed ($k = 0$), the algorithm terminates by returning “false”. Otherwise, the algorithm iterates over every pair of vertices $\{u, v\}$ of the subgraph $G[S]$. It edits the edge $\{u, v\}$ and calls itself recursively with the modified graph and the number of allowed edits reduced by one. If the recursion returns “true” the algorithm terminates, otherwise it reverts the edit and continues with the next pair. If the recursion returns “false” for all pairs of vertices, the algorithm also returns “false” as no solution could be found.

This algorithm runs in $O(s^k T(n))$ time, where $s = \max\{\frac{|V_F|(|V_F|-1)}{2} \mid F \in \mathcal{F}\}$ is the number of pairs of vertices in the largest forbidden subgraph and $T(n)$ is the time needed for finding an instance of a forbidden subgraph. The basic editing algorithm is Fixed Parameter Tractable in k since $T(n) \in \text{poly}(n)$ as shown by [Cai96]. As [Cai96] considers the correctness “obvious”, we refer to [Sch15, Boh15] for a formal proof.

As a trivial optimization already included in Algorithm 2.1, it is never beneficial to apply the same edit multiple times: For every two edits of the same pair of vertices, the second reverts the first one and the resulting graph can be obtained by not applying these edits, thus saving two edits [Sch15, Boh15]. We therefore prevent editing and recursing if the pair of vertices chosen by line 6 has already been edited in line 7.

For the case of the set of forbidden subgraphs $\mathcal{F}$ being $\{P_5, C_5\}$, [Sch15] show that the $\mathcal{F}$-free Edge Editing Problem can be solved in $O(9^k T(n))$ time, whereas the formula given above results in $O(10^k T(n))$ time. We give an alternate proof, generalized for any $\mathcal{F} = \{P_l, C_l\}$, with $l \geq 4$.

Theorem 2.2. *If the set of forbidden subgraphs $\mathcal{F}$ is $\{P_l, C_l\}$, with $l \geq 4$, the $\mathcal{F}$-free Edge Editing Problem can be solved in $O((\frac{l(l-1)}{2} - 1)^k T(n))$ time.*

Proof. The forbidden subgraph $G[S]$ found in a recursion is either a P_l or a C_l. We show that in both cases, there is a pair of vertices that can be ignored.

Algorithm 2.1: Basic Editing Algorithm

Input: Graph $G = (V, E)$, maximum number of edits k, set $\mathcal{F}$ of forbidden subgraphs, set L of previous edits

Output: A boolean indicating whether the $\mathcal{F}$-free editing problem can be solved for G with at most k edits

```
1  $S \leftarrow$ FindSubgraph($G$, $\mathcal{F}$)
2  if $S = \emptyset$ then
       // no forbidden subgraph was found
3      return true
4  if $S \neq \emptyset$ and $k = 0$ then
       // a forbidden subgraph was found, but no more edits allowed
5      return false
6  forall $e = \{u, v\} \in S \times S$ with $u \neq v$ do
7      if $e \notin L$ then
8          $L \leftarrow L \cup e$
9          $G$.toggle($e$)
10         if edit($G$, $k-1$, $F$, $L$) then
11             return true
12         $G$.toggle($e$)
13         $L \leftarrow L \setminus e$
14 return false
```

If $G[S]$ is a P_l, inserting the edge converting the P_l into a C_l is not useful. The resulting C_l consists of the same set of vertices S and we would have the following options: Reverting the edit we just made, which is never beneficial as shown above, or editing one of the other pairs of vertices. We could have chosen these other pairs initially, destroying the P_l but using fewer edits. Therefore, we can ignore the edit converting the P_l into a C_l. This is the same reasoning as given by [Sch15].

If $G[S]$ is a C_l, we can choose to ignore one of the existing edges. Note that, to destroy a C_l without leaving behind a P_l, we need to either insert a new edge or remove any two existing edges. After removing a single edge, in the resulting P_l, we do not consider reinserting the edge, as shown in the first part of this proof. Reinserting would also revert an edit and thus can not be beneficial. Any other edit destroys the forbidden subgraph. Thus, the edit ignored when handling the resulting P_l can never be the edit we chose to ignore when handling the C_l. Therefore, any combination of two deletions can still be achieved, regardless of which edge we chose to ignore.

We have now shown that we can ignore one pair of vertices in both cases. Thus, only $\frac{l(l-1)}{2} - 1$ choices remain in each recursion, resulting in a time complexity of $O((\frac{l(l-1)}{2} - 1)^k T(n))$ for the $\{P_l, C_l\}$-free Edge Editing Problem, with $l \geq 4$. □

2.3 Redundant Editing Algorithm

For our improved editing algorithm we expand on the optimization of never editing a pair of vertices twice.

Note the following observation: If two branches of the search tree followed by the basic algorithm apply the same set of edits, regardless of the order in which the edits were made, they result in the same graph. Further exploration of the respective subtrees will produce the same result. It would thus be advantageous to explore only one of these subtrees in order to avoid doing redundant work.

We can identify edits that lead to redundant exploration with the following observation: Suppose during one recursion we found a forbidden subgraph $G[S]$, decided to apply an edit e and its recursion returned "false". Then we know that for the remaining edits we have to try for $G[S]$ can not lead to a solution that applies e as such a solution would have been found by the recursion we did with e applied. Therefore, we can forbid applying e until we return from the recursion that found $G[S]$. Thus, no graph explored by the recursion that applied e can be found by later recursions as e will not be edited in them. For our implementation we introduce the notion of marked edits. A marked edit can not be applied by the Redundant Editing Algorithm, regardless of whether it is currently applied or not.

In detail, the Redundant Editing Algorithm differs from the Basic Editing Algorithm as follows: Instead of considering all edits for a forbidden subgraph independently of each other (an iteration of the loop in lines 6 to 13 in Algorithm 2.1 has no effect unless a solution is found), we defer clearing the mark of an edit (Algorithm 2.1, line 13) until we exit the loop (Algorithm 2.2, lines 15 and 16). Edits marked in previous recursions of this branch are entirely skipped over, in accordance with the observation above. In the Basic Editing Algorithm this corresponds to the optimization of never applying an edit twice. The effect is that all edges that were previously edited by this loop are still marked although the edits themselves were reverted. Future recursions caused by this branch are thus prevented from editing these edges, fixing them to their original state. Phrased differently, we explore a binary decision tree. For each unmarked pair of vertices in the found forbidden subgraph we ask the following question: Edit this pair of vertices? If yes, edit it and recurse, otherwise continue with the next pair. In any case, prevent further modifications of this pair by marking it. Clearing the markings in lines 15 and 16 is needed to maintain that the algorithm has no side effects when returning "false".

Before showing the correctness of the Redundant Editing Algorithm, we first note some observations:

Lemma 2.3. *Each recursion keeps at most one edit applied at a time.*

Proof. Edits are only applied in line 11 and line 14. If line 11 is executed either the edit is reverted by line 14 or the algorithm returned "true" in line 13. ☐

Lemma 2.4. *When a recursion returns "false", it had no side effects.*

Proof. The parameters k and $\mathcal{F}$ are never modified and modifications to G are already covered by Lemma 2.3. All additions to L happen in line 9 and are, with the help of M, undone in line 16. The only possibility to not reach line 16 after having reached line 9 is by returning "true" in line 13. Thus, the Redundant Editing Algorithm has no side effects when returning "false". ☐

Using these observations, we now prove that the Redundant Editing Algorithm is correct.

Theorem 2.5. *The Redundant Editing Algorithm is correct.*

Proof. We show that the Redundant and the Basic Editing Algorithm always return the same result. First, we prove that for every k the Basic Editing Algorithm finds a solution for every solution found by the Redundant Editing Algorithm. For the second part, we

Algorithm 2.2: Redundant Editing Algorithm

```
Input: Graph $G = (V, E)$, maximum number of edits $k$, set $\mathcal{F}$ of forbidden
       subgraphs, set $L$ of marked edits
Output: A boolean indicating whether the $\mathcal{F}$-free editing problem can be solved
        for $G$ with at most $k$ edits
1  $S \leftarrow$ FindSubgraph$(G, \mathcal{F})$
2  if $S = \emptyset$ then
       // no forbidden subgraph was found
3      return true
4  if $S \neq \emptyset$ and $k = 0$ then
       // a forbidden subgraph was found, but no more edits allowed
5      return false
6  $M \leftarrow \emptyset$
7  forall $e = \{u, v\} \in S \times S$ with $u \neq v$ do
8      if $e \notin L$ then
9          $L \leftarrow L \cup \{e\}$
10         $M \leftarrow M \cup \{e\}$
11         $G$.toggle$(e)$
12         if edit$(G, k-1, F, L)$ then
13             return true
14         $G$.toggle$(e)$
15 forall $e \in M$ do
16     $L \leftarrow L \setminus \{e\}$
17 return false
```

reverse the roles and show that the Redundant Editing Algorithm finds a solution for every solution found by the Basic Editing Algorithm.

"$\Rightarrow$": *The Basic Editing Algorithm returns "true", if the Redundant Editing Algorithm returns "true".*

Assume the Redundant Editing Algorithm returned "true" and found a set L' of edits for graph G' and k' allowed edits. Further, assume FindSubgraph is correct. We show by induction over the length l of the currently explored path p that we can construct a path taken by the Basic Editing Algorithm that also returns "true".
Base clause: $p = (), l = |p| = 0$
On the initial call to Edit, its parameters are $G = G'$, $k = k'$ and $L = \emptyset$.
Induction hypothesis:
On a call to Edit exploring the path p with length $l = |p|$, the following relations hold: We used up l edits, i.e. l is $k' - k$, L has l members ($l = |L|$) and is a subset of L' ($L \subseteq L'$) and all pairs of vertices in L are currently edited ($G = G' \triangle L$).
Induction step: $l \rightarrow l + 1$
Let $G'' = G \overset{IH}{=} G' \triangle L$ and $L'' = L$ be the values of G and L passed to this call of Edit. Additionally, let S be the forbidden subgraph found by FindSubgraph in line 1. If $S = \emptyset$, then the Basic Editing Algorithm returns "true". This shows the claim. Otherwise, if $k = 0$ and $S \neq \emptyset$, then either L' is not a solution or FindSubgraph is incorrect, contradicting the assumption. Otherwise, we enter the loop in line 6. At least one of the pairs of vertices in S must be an edit found in L', due to Lemma 2.1. In addition, at least one of the pairs of vertices found in both S and L' cannot be in L. If they all were in L, then all edits needed to destroy the forbidden subgraph $G[S]$ would already have been

applied, as L' solves the problem and thus FINDSUBGRAPH could not have found $G[S]$. Therefore, if $((S \times S) \cap L') \setminus L = \emptyset$, either L' is not a solution or FINDSUBGRAPH is incorrect, contradicting the assumption. Thus, there must be at least one unedited pair of vertices that is present in both S and L'. If none of these pairs are iterated over, the Basic Editing Algorithm found a different solution and thus it returns "true". This shows the claim. Otherwise, let e be the first pair of vertices iterated over that is present in L'. In all prior iterations of the loop, the recursive call returns "false" and G is the same compared beginning of this function ($G = G''$) (Lemma 2.4). In the iteration that handles e, e is added to L and applied to G. At the point of the recursive call, this call applied a single additional edit, namely e, to G ($G = G''\triangle\{e\}$) and added it to L ($L = (L'' \cup \{e\})$). Thus, $|L \cap L'| = |L'' \cap L'| + 1 \overset{IH}{=} l + 1$ and $G = G''\triangle\{e\} \overset{IH}{=} G'\triangle(L'' \cap L')\triangle\{e\} = G'\triangle(L \cap L')$, which, together with the reduction of k by one, satisfies the induction hypothesis for the recursive call exploring the path $(p_1, \dots, p_n, e)$.

We have shown that the Basic Editing Algorithm finds a solution for every solution found by the Redundant Editing Algorithm. To show that the Redundant Editing Algorithm is able to find all valid solutions, we need the second part of this proof. This part is very similar to the first, except that we now have to consider the pairs of vertices marked by the Redundant Editing Algorithm.

"$\Leftarrow$": *The Redundant Editing Algorithm returns "true", if the Basic Editing Algorithm returns "true".*

Assume the Basic Editing Algorithm returned "true" and found a set L' of edits for graph G' and k' allowed edits. Further, assume FINDSUBGRAPH is correct. We show by induction over the length l of the currently explored path p that we can construct a path taken by the Redundant Editing Algorithm that also returns "true".
Base clause: $p = (), l = |p| = 0$
On the initial call to EDIT, its parameters are $G = G'$, $k = k'$ and $L = \emptyset$.
Induction hypothesis:
On a call to EDIT exploring the path p with length $l = |p|$, the following relations hold: We used up l edits, i.e. l is $k' - k$, all edited pairs of vertices in L are also in L' ($G = G'\triangle(L \cap L')$) and exactly l edits are currently applied ($l = |L \cap L'|$).
Induction step: $l \to l + 1$
Let $G'' = G \overset{IH}{=} G'\triangle(L \cap L')$ and $L'' = L$ be the values of G and L passed to this call of EDIT. Additionally, let S be the set of vertices forbidden subgraph found by FINDSUBGRAPH in line 1. If $S = \emptyset$, the Redundant Editing Algorithm returns "true". This shows the claim. Otherwise, if $k = 0$ and $S \neq \emptyset$, then either L' is not a solution or FINDSUBGRAPH is incorrect, contradicting the assumption. Otherwise, we enter the loop in line 7. At least one of the pairs of vertices in S must be an edit found in L', due to Lemma 2.1. In addition, at least one of the pairs of vertices found in both S and L' cannot be in L. If they all were in L, then all edits needed to destroy the forbidden subgraph $G[S]$ would already have been applied, as L' solves the problem and thus FINDSUBGRAPH could not have found $G[S]$. Therefore, if $((S \times S) \cap L') \setminus L = \emptyset$, either L' is not a solution or FINDSUBGRAPH is incorrect, contradicting the assumption. Thus, there must be at least one unmarked pair of vertices that is present in both S and L'. If none of these pairs is iterated over, the Redundant Editing Algorithm found a different solution and thus it returns "true". This shows the claim. Otherwise, let e be the first pair of vertices iterated over that is present in L'. In all prior iterations of the loop, the recursive call then returns "false" and G is the same compared beginning of this function ($G = G''$) (Lemma 2.4). In the iteration that handles e, e is added to L and applied to G. At the point of the recursive call, this call applied a single additional edit, namely e, to G ($G = G''\triangle\{e\}$) and L contains a single additional edit also present in L' and possibly a number of other

edits ($L \supseteq (L'' \cup \{e\})$, $(L \setminus L'') \cap L' = \{e\}$). Thus, $|L \cap L'| = |L'' \cap L'| + 1 \overset{IH}{=} l + 1$ and $G = G''\triangle\{e\} \overset{IH}{=} G'\triangle(L'' \cap L')\triangle\{e\} = G'\triangle(L \cap L')$, which, together with the reduction of k by one, satisfies the induction hypothesis for the recursive call exploring the path $(p_1, \ldots, p_n, e)$.

We have now shown that if one algorithm returns "true", the other algorithm returns "true" too, thus both algorithms are equivalent. As the Basic Editing Algorithm is correct, the Redundant Editing Algorithm therefore is correct as well. □

Now that we have shown that the Redundant Editing Algorithm is correct, we verify that we achieved our goal of avoiding redundant work.

Theorem 2.6. *The Redundant Editing Algorithm explores the subtree induced by a set of edits at most once.*

Proof. Assume the Redundant Editing Algorithm explores the subtree induced by a set of edits multiple times. Then there are at least two sequences of edits $L = (e_1, e_2, \ldots, e_n)$ and $L' = (e'_1, e'_2, \ldots, e'_n)$ consisting of the same set of edits, but in different order. Let i be the index of the first edit where the sequences diverge, i.e. $e_i \neq e'_i$ and $e_j = e'_j$ for all $j < i$. As both sequences contain the same set of edits, e_i must also be contained in L', but after e'_i, and likewise e'_i is in L, but after e_i. Then, after applying the edits $e_1, e_2, \ldots, e_{i-1}$, the algorithm found a forbidden subgraph induced by a set of vertices S containing both e_i and e'_i. While iterating over the pairs of vertices of S, assume without loss of generality e_i is handled before e'_i. At this point the algorithm branches and either marks and edits e_i and recurses or marks, but does not edit e_i. In the latter branch, and therefore especially when the algorithm eventually handles edge e'_i, e_i is already marked, which prevents the algorithm from editing e_i in this branch. Thus, e_i can not be edited after editing e'_i which was required by L' and therefore the assumption that the Redundant Editing Algorithm explores subtrees multiple times is false. This shows the claim. □

2.4 Lower bounds

To further reduce the number of explored branches we now consider calculating lower bounds of the number of edits needed to make G $\mathcal{F}$-free. Given such a lower bound l and the number of allowed edits k, we can immediately return "false" if $l > k$, preventing further exploration of this path in the search tree.

[Boh15] defines a *packing* Q as a set of forbidden subgraphs such that no two subgraphs share a pair of vertices and proves that this is a lower bound for the number of edits needed. While their work only concerns itself with a specific forbidden subgraph, definition and proof are trivially generalized. Their work then discusses three ways to construct such a packing. First they introduce the *vertex disjunct packing*, constructed by searching for a forbidden subgraph $G[S]$, adding it to the packing and then removing all vertices of S from the graph G. This is repeated until G no longer contains a forbidden subgraph. This will find a lower bound of at most $\lfloor \frac{n}{x} \rfloor$, where x is the number of vertices in the smallest forbidden subgraph. Second is the *vertex pair disjunct packing*, which needs a marking per pair of vertices indicating whether this pair is already contained in the packing. They search for a forbidden subgraph $G[S]$ in which no pair of vertices was marked yet, adding the forbidden subgraph to the packing and marking all pairs of vertices. Again, this is repeated until no completely unmarked forbidden subgraph can be found. The resulting lower bound will be at most $\lfloor \frac{n(n-1)}{x(x-1)} \rfloor$, where x is the number of vertices in the smallest forbidden subgraph. The third approach requires listing all forbidden subgraphs and constructs an instance of the hitting set problem, which is then solved by a heuristic.

Algorithm 2.3: LOWER BOUND

```
Input: Graph $G = (V, E)$, set $L$ of previous edits
Output: A lower bound of the minimal amount of edits needed to make $G$ $\mathcal{F}$-free
1  bound ← 0
2  $M \leftarrow \emptyset$
3  forall $S \in$ FINDALLSUBGRAPHS do
4      $b \leftarrow$ true
5      forall $e = \{u, v\} \in S \times S$ with $u \neq v$ do
6          if $e \notin L$ and $e \in M$ then
7              $b \leftarrow$ false
8      if $b$ then
9          bound ← bound + 1
10         forall $e = \{u, v\} \in S \times S$ with $u \neq v$ do
11             if $e \notin L$ then
12                 $M \leftarrow M \cup \{e\}$
13 return bound
```

This is basically a non-greedy version of the vertex pair disjunct packing, and thus has the same upper bound.

We adapted the vertex pair disjunct packing as it offers better bounds than the vertex disjunct packing while avoiding the memory problems of the hitting set approach. We are also able to improve the lower bound by considering that edges which were already edited resp. marked will not be edited again and thus cannot be used to destroy a forbidden subgraph. Therefore, multiple forbidden subgraphs, whose only overlap are pairs of vertices that are already edited or marked may all be members of the same packing. Algorithm 2.3 shows our implementation.

We also implemented a mechanism to update a previously calculated lower bound. When marking a pair of vertices, we only search for forbidden subgraphs containing that pair and test whether they can now contribute to the lower bound. When editing a pair of vertices, any forbidden subgraph containing that pair will be destroyed. The packing of the lower bound might contain one of these subgraphs. If so, we first remove that subgraph from the lower bound. We then search for new forbidden subgraphs containing at least one of the pairs of vertices of the just removed subgraph. Each of the new subgraphs is then tested whether it can contribute to the lower bound and on success added to the packing. All changes made for updating the lower bound are recorded in a stack. When an edit or marking is reversed, the corresponding changes recorded on the stack are undone. Simply using the updating algorithm again could result in a lower bound consisting of a different set of forbidden subgraphs, violating Lemma 2.4,

2.5 Subgraph selection

In this section we will discuss various ways for finding a forbidden subgraph, called FINDSUBGRAPH in the editing algorithms. The inputs are the graph G, the set $\mathcal{F}$ of forbidden subgraphs, and for our improved versions the set L of pairs of vertices in G that were already edited resp. marked. For simplicity, we will refer to L as a set of edits in this section, regardless whether it is actually the set of edits of the Basic Editing Algorithm or the set of markings of the Redundant Editing Algorithm. The output is either a set S

of vertices for which $G[S]$ is a forbidden subgraph or the empty set in case no forbidden subgraph could be found.

2.5.1 First

The basic approach for finding a forbidden subgraph is to enumerate all subsets $S \subseteq V$ and to return the first for which $G[S]$ is isomorphic to a forbidden subgraph.

2.5.2 Most edited

Instead of simply returning the first forbidden subgraph found, we now consider searching for the most edited forbidden subgraph in G, i.e. the forbidden subgraph with the most pairs of vertices present in L. For ties, we select an arbitrary forbidden subgraph.

The idea behind this approach is that reducing the number of possible edits leads to fewer recursive calls improving the overall running time. Since we know that we need to edit at least one pair of vertices in each forbidden subgraph (Lemma 2.1) and we know that editing already edited edges is not beneficial, this does not affect correctness.

2.5.3 Anti triangle

For the specific case of the set of forbidden subgraphs $\mathcal{F} = \{P_4, C_4\}$, being the path and cycle with four vertices each, we implemented a selection algorithm based on [JGGW14]. For each edge of G we calculate the number of existing and potential forbidden subgraphs containing it and update these numbers after every edit. [JGGW14] defines the number of potential P_4 of an edge as the number of P_3 and C_3 containing the edge. As this only considers edges, instead of any pair of vertices, only returning the edge with the highest ratio and using it for an adaption of the single edge editing strategy we introduce in the next section (Section 2.6) is insufficient. (This would only allow deletions, however the Edge Editing Problem allows deletions and insertions.) We therefore return a forbidden subgraph containing the edge with the highest ratio of existing to potential forbidden subgraphs. We expect that an edge with a high ratio needs to be edited and choosing it early allows the algorithm to find a solution quickly. On the other hand, while choosing to not edit this edge should provide no benefit by itself, when using the Redundant Editing Algorithm this edge will then be marked, which we expect to lead to an increased lower bound.

2.6 Single edge editing

While the anti triangle selection was designed to quickly find a solution, to prove a given k as minimal we need to show that a problem cannot be solved with $k-1$ allowed edits. Here the goal is therefore not to quickly find a solution, but instead to show that a given path does not lead to a solution. To show that a path does not lead to a solution, we calculate a lower bound on the number of edits. Calculating a higher lower bound reduces the running time, as it allows cutting paths that do not lead to a solution earlier. Obtaining a higher lower bound can be done by improving the quality of the calculated lower bound, for example by using the hitting set approach mentioned in Section 2.4. To avoid the problems of the hitting set approach, we designed another method instead: We try to find a “bad” pair of vertices that increases the calculated lower bound, regardless of whether we edit or mark it. This no longer results in a forbidden subgraph but a single pair of vertices. But in contrast to [JGGW14] which only considers edges, there is no reason to limit ourselves to specific pairs of vertices here.

The algorithm as presented in Section 2.3 would consider the pair as a forbidden subgraph and only try editing it. We therefore need to adapt the Redundant Editing Algorithm to handle single pairs of vertices. The only choices are to either edit or mark the pair, thus there are at most two recursive calls. Adapting the Basic Editing Algorithm is not useful as it does not have the concept of marked pairs, resulting in the latter option having no effect allowing infinite recursion. In detail, assume the current recursion has k allowed edits remaining and we calculated a lower bound of l edits. Let us then obtain k' and l' for these values in a recursive call. We now search for a pair of vertices for which $l' \geq l - 1$ holds after editing the pair and $l' > l$ after marking it. That is, we spend one of our allowed edits with the calculated lower bound decreasing by at most one or we mark the pair which results in increasing the calculated lower bound. If we can always find such a pair of vertices we can solve the Edge Editing Problem in $O(4^k \operatorname{poly}(n))$ as proven below. However, we cannot guarantee such a pair of vertices exists. If we cannot find one, we fall back to selecting an arbitrary forbidden subgraph and using it for an iteration of the "normal" Redundant Editing Algorithm. We therefore inherit its worst case time complexity.

Theorem 2.7. *If, in every recursion, there exists an unmarked pair of vertices for which $l' > l$ holds after marking it, the Edge Editing Problem can be solved in $O(4^k \operatorname{poly}(n))$.*

Proof. Let us consider the situation described above: There are currently k allowed edits remaining, we calculated a lower bound of l, and have selected a pair of vertices for editing.

For the recursion that marks but does not edit the pair of vertices, $l' > l$ must be true due to the theorem's condition. As we did not make an edit, the number of allowed edits remains constant ($k' = k$). Since we prune a path once the calculated lower bound exceeds the number of allowed edits ($l' > k'$), we can choose this option for at most k recursions.

In the recursion that edits the pair of vertices we reduce the number of allowed edits by one ($k' = k - 1$). Therefore, we can choose this option for at most k recursive calls, otherwise we would exceed the number of allowed edits. Taking this option does not increase the number of times we can choose the marking option: The number of times we can choose the marking option can be expressed as the difference $k - l$ of the number of allowed edits and the calculated lower bound. Thus, we need to show that $k' - l' \leq k - l$ holds after editing the pair of vertices. As we know the relation between k and k' we can simplify this to $l' \geq l - 1$. This is trivially true as we can always derive an updated lower bound of $l' := l - 1$ from the current one.

Overall, we therefore can make at most $2^{k+k} = 2^{2k} = 4^k$ recursions without exceeding either limit, resulting in a running time of $O(4^k \operatorname{poly}(n))$ for the Edge Editing Problem under this condition. □

Due to the packing approach we use for our lower bound calculations, the precondition of Theorem 2.7 can only be fulfilled by pairs of vertices currently participating in the lower bound. Marking a pair that is not contributing to the lower bound can not result in the new forbidden subgraph being added, as such a subgraph would already have been found by the greedy nature of the lower bound calculation.

If, in any given recursion, there are multiple pairs of vertices satisfying the precondition of Theorem 2.7, we prioritize as follows: First, we consider having only one direct recursion to be preferable to having both. Second, we minimize the number of potential child recursions caused, either directly or indirectly, by this recursion. In detail, we test if there are pairs for which one branch of the recursion would immediately be cut due to the lower bound exceeding the number of allowed edits. If there are, we choose among those pairs the one with the highest increase of the lower bound in the other branch. If there are none, we consider, for each pair of vertices, the branch with the smaller increase to the lower bound and choose the highest increase among them. If this selects multiple pairs, we pick one of the pairs with the highest increase to the lower bound in either branch from these.

3. Evaluation

In this chapter, we first describe the setup used for our experiments. Using the path and the cycle with four vertices each as the set of forbidden subgraphs ($\mathcal{F} = \{P_4, C_4\}$), we then compare the Basic Editing Algorithm with the Redundant Editing Algorithm. We then compare the forbidden subgraph selection strategies with each other using the Redundant Editing Algorithm. Afterwards we test the behavior of the best forbidden subgraph selection strategy in a multithreaded environment. Finally, we analyze the solutions of the graphs we were able to solve.

3.1 Experiment Setup

We now introduce the graphs and hardware we used in our experiments and mention some implementation details optimizing the running time.

3.1.1 Used Graphs

For our evaluation we use the same graphs as [NG13]. We introduce these graphs in more detail now.

Zachary's karate club[Zac77], which we will call "karate" for short, documents observations on a university's karate club. Members are represented by vertices and are connected when members interacted with each other outside of club activities. Due to a conflict between the administrator and the instructor, the club split into two. Except for one member, Zachary correctly predicted the split.

[Knu93] constructed a graph based on the novel "Les Misérables" by Victor Hugo. The characters of the novel are represented by vertices and two characters are connected when there is a chapter in which they appear together. We will call this graph "lesmis".

"Dolphins" was created by [Lus03] from observations of a group of dolphins. Edges connect dolphins that were seen together significantly more often than what would be expected over the observation period.

[DHC95] depicts the food chain between various grassland species. We will call this graph "grassweb".

[GN02] constructs a graph off a season of United States college football. Two teams are connected if they played against each other during the season. While, according to [GN02], the graph represents the 2000 season, [Eva10] notes it is more likely the 2001 season. We will call this graph “football”.

In addition to the real world graphs, we generated synthetic graphs as described in [BHSW15]. For each graph, first several connected components are created. The sizes of the components are controlled by a power law sequence with minimum 10, maximum $0.2n$ and exponent -1, where n is the number of vertices in the final graph. Each of these components is transformed into a quasi-threshold graph. For each vertex v of a component, in order, a parent p in $\{0, \dots, v-1\}$ is randomly chosen. Then edges are added from v to p and every vertex in the parent’s neighborhood. All connected components of a graph are then connected by editing some pairs of vertices. For a target number of k required edits, $0.2k$ edges are deleted and $0.8k$ new edges are inserted. The resulting graphs typically requires slightly fewer edits than targeted, as the created forbidden subgraphs might overlap. We generated groups with ten graphs each for all combinations of 100, 200, 400, . . . , 1000 vertices and 20, 40, . . . , 100 targeted edits.

3.1.2 Hardware

All experiments were run on a server with two Intel Xeon E5-2670 CPUs for a total of 16 cores clocked at 2.6GHz. Except for the multithreading experiments in Section 3.4, the experiments only used a single core. The server has 64GiB of RAM and runs openSUSE 42.2. We implemented our algorithms in C++14 and compiled them using GCC version 5.3.1 with full optimizations and disabled assertions (i.e. `-O3 -DNDEBUG -march=native`).

3.1.3 Implementation details

As the sizes of the used graphs are relatively small, we use adjacency matrices as graph representation.

When utilizing a lower bound, its calculation is incorporated into the code of the forbidden subgraph selection algorithm. For the most edited selection strategy this is beneficial as both, forbidden subgraph selection and calculation of a lower bound, require enumerating all forbidden subgraphs. The anti triangle and the single edge selection strategy are designed around using a lower bound and still allow sharing some work.

When calculating the lower bound, whenever given a choice of pairs of vertices to iterate over, we choose the already edited or marked ones first. Due to the greedy nature of our lower bounds calculation, this aims to reserve fewer pairs of vertices for a given forbidden subgraph participating in the lower bound. Remember that forbidden subgraphs may overlap in already edited or marked pairs of vertices, as these pairs can not be used to destroy these subgraphs.

In cases where no pair of vertices satisfies the precondition of Theorem 2.7 for the single edge editing strategy, we fall back to selecting the most edited forbidden subgraph.

We repeat that the set of forbidden subgraphs used for the evaluation was the path and the cycle with four vertices each ($\mathcal{F} = \{P_4, C_4\}$). While most algorithms presented in this work are usable for any choice of $\mathcal{F}$, our implementation is specialized for the case of $\mathcal{F} = \{P_4, C_4\}$.

3.2 Editing Algorithms

We first confirm that the Redundant Editing Algorithm is actually an improvement over the Basic Editing Algorithm. We use “karate” as an example here, as it can be solved with a low number of edits, while still showing the typical behaviour we observed on other

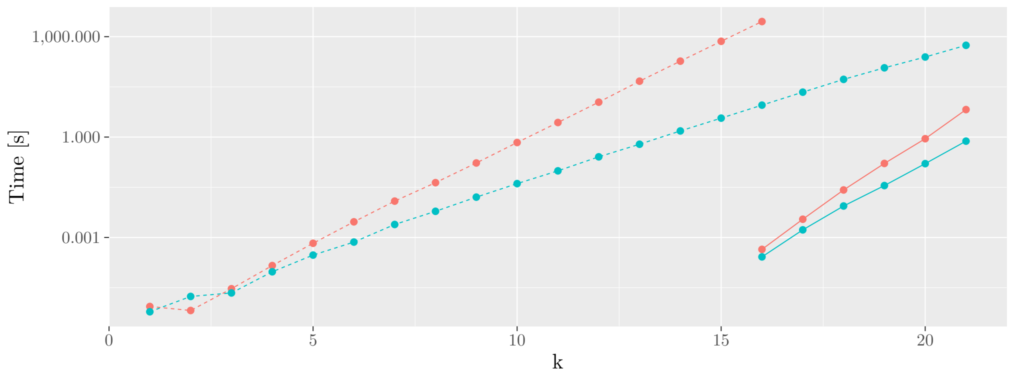


Figure 3.1: Running time of the Basic Editing Algorithm (red) and the Redundant Editing Algorithm (blue), in terms of the number of allowed edits (k). The experiment is run on the “karate” graph. The solid lines were obtained by utilizing lower bounds, while for the dashed lines lower bounds were not used.

graphs. In Figure 3.1, we show the running times of both algorithms on the “karate” graph. Note that the running time is displayed on a logarithmic scale. We start with allowing only a single edit, i.e. $k = 1$ and increase the number of allowed edits until either a solution is found ($k = 21$) or the editing algorithm does not terminate within three hours. While the Basic Editing Algorithm gets cut off after $k = 16$ due to exceeding the allowed running time, the Redundant Editing Algorithm finds a solution in about nine minutes. We observe that the speedup increases with increasing k. This is because with increasing k, more identical graphs are explored. This is not suprising as with a larger number of edits there are more ways to explore them in a different order – there are $l!$ possibilities to order l elements in a set. The Redundant algorithm, however, is guaranteed to explore each set at most once, as shown in Theorem 2.6.

The Redundant Editing Algorithm with lower bounds is significantly faster than the Basic Editing Algorithm. The relative difference is much smaller compared to not using lower bounds. We omit the values for $k \leq 15$ as the initially calculated lower bound for the graph is 16, causing both algorithms to immediately return “false”.

For the other graphs we obtained similar results. For visualization, we refer to Figure A.1.

We confirm the results of [Boh15] concerning lower bounds and find them to be essential for solving the $\mathcal{F}$-free Edge Editing Problem in a timely manner. We also find that the theoretical advantage of the Redundant Editing Algorithm over the Basic Editing Algorithm (Theorem 2.6) results in a significant reduction in running time. For “karate” we report a difference of an order of magnitude. The difference increases for the other graphs.

3.3 Forbidden subgraph selection

In this section, we compare the algorithms for selecting forbidden subgraphs we introduced in Sections 2.5 and 2.6. In all experiments we use the Redundant Editing Algorithm and calculate lower bounds. As in the previous section, we omit the cases where k is lower than the lower bound computed on the input graph.

On “karate”, all strategies need less than a second to solve the problem. Simply selecting the first forbidden subgraph found is the slowest strategy, while using anti triangle and selecting the most edited forbidden subgraph are the fastest. When investigating the number of recursive calls caused by each selection strategy, we see that single edge editing causes the least amount of recursive calls. This advantage does not play out however,

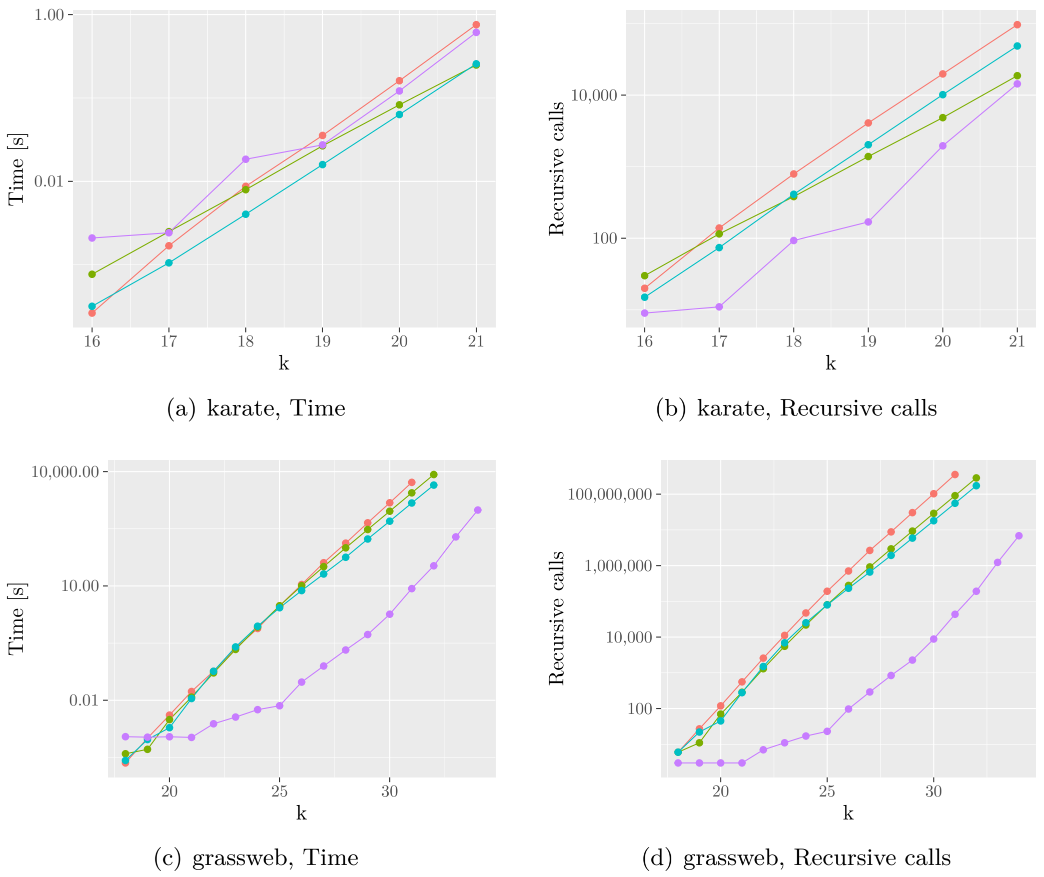


Figure 3.2: Comparison of the forbidden subgraph selection strategies when using the Redundant Editing Algorithm, in terms of the number of allowed edits (k). The colors used for the strategies are: First: red, most edited: blue, anti triangle: green, single edge editing: purple.

due to the higher complexity of the selection strategy and the small size of “karate”. See Figure 3.2(a) and (b) for a visualization.

On “grassweb” and “football”, the outcome is different: Single edge editing still causes fewer recursions, but here this results in significantly less running time compared to the other strategies. It is the only strategy that is able to solve “grassweb” within the maximum running time of three hours, requiring about 16 minutes. Among the other strategies selecting the most edited forbidden subgraph is better than simply selecting the first one, the anti triangle strategy falls between these two. The results on “grassweb” are shown in Figure 3.2(c) and (d), for “football” we refer to Figure A.2.

On “lesmis” and “dolphins”, we observe the same patterns as on “grassweb” and “football”, but with minor differences. On “dolphins”, the smaller number of recursions for single edge editing does not result in a faster running time, causing it to be slightly slower than selecting the most edited forbidden subgraph. On “lesmis”, the anti triangle strategy falls behind selecting the first forbidden subgraph found, in both time and recursive calls. Single edge editing is able to solve “lesmis” within three hours, requiring 61 seconds. Visualizations can be found in Figure A.2.

Overall, we find that single edge editing is either the best strategy or at least among the best strategies for selecting a forbidden subgraph. It is also the only strategey that is able to solve graphs other than “karate” within the time limit. Considering the precondition of

Theorem 2.7 we find it to be satisfied frequently. The worst case is “lesmis” where we need to fall back other subgraph selection strategies in 2.1% of all recursions.

3.4 Multithreading

We developed a simple work sharing strategy for the Redundant Editing Algorithm. All threads have access to a single work sharing queue, and monitor its length. Initially this queue contains a single task representing the whole problem. A thread that was just created or finished its current task, removes the first element of the queue and calls the editing algorithm with it. Should the queue be empty the thread waits for other threads to insert new tasks. If all threads are waiting, the problem is finished and the algorithm terminates.

Every thread checks the length of the queue once during each recursive call. If the length falls below a threshold (we use the number of threads), any thread noticing this interrupts its exploration and splits its current task: In the outermost recursion of the editing algorithm, all remaining iterations of the loop surrounding the recursive call (Lines 7 to 14 in Algorithm 2.2) are executed immediately. Instead of the recursive call, a copy of the current state is created and inserted into the work queue. The execution of the loop then continues as if the recursion returned “false”. The outermost recursion is then removed from the call stack, making the next recursion along the path the thread is currently exploring the new outermost recursion. The thread then resumes it normal exploration.

We evaluate the work sharing strategy on the “dolphin” and “football” graphs using the single edge editing strategy. Using more threads allows the algorithm to finish faster, although the relative efficiency decreases with an increased number of threads. We explain this decrease with the increased contention of the mutex guarding access to the work sharing queue. See Figures 3.3(a) and (b) for a visualization.

On “dolphins”, we observe superlinear speedup (Figure 3.3(d)). This behavior is due to an implementation detail. When a task is inserted into the work sharing queue, the information which forbidden subgraphs are part of the currently used lower bound is lost. This forces a recalculation of the lower bound a soon as a thread starts working on this task. After correcting this inconsistency by recalculating the lower bound in any case, the observed efficiency on “dolphins” is similar to the one on “football” (Figure 3.3(e)).

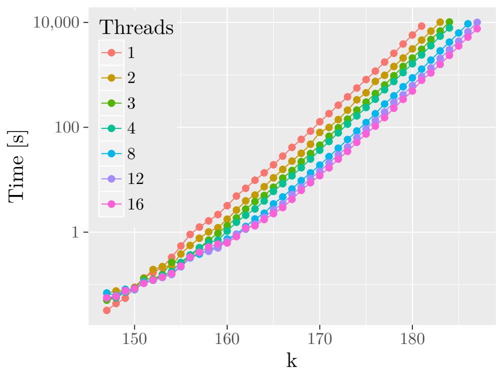
We find that, regardless of the loss of efficiency, using multithreading reduces the running time. However, the $\mathcal{F}$-free Edge Editing Problem scales exponentially, limiting the achievable effect to an increase in k by a few steps, before exhausting the time limit.

3.5 Summary

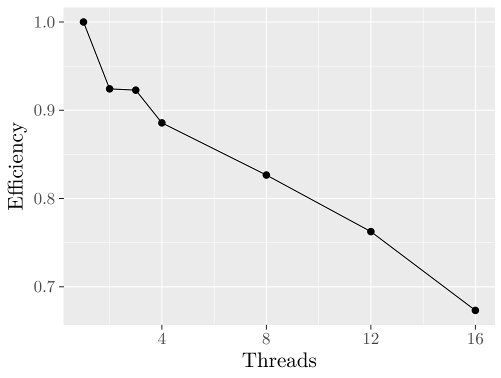
A summary of the best results achieved for each real world graph can be seen in Table 3.1. On “karate”, which is the only graph that has been solved prior to this work, we improve the running time by an order of magnitude compared to [Boh15]. For “lesmis” and “grassweb”, the number of edits needed are identical to upper bounds discovered by heuristic approaches presented in [NG13, BHSW15]. This suggests these heuristics produce good results, although, with only three solved graphs it is too early to draw a definite conclusion.

3.6 Synthetic graphs

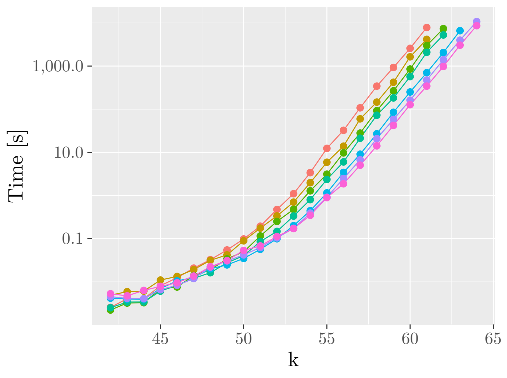
For a broader evaluation of the behavior of the Redundant Editing Algorithm using single edge editing, we use the synthetic graphs. The results are shown in Figure 3.4. We find the size of the graph has only a minor influence on the running time, instead the number



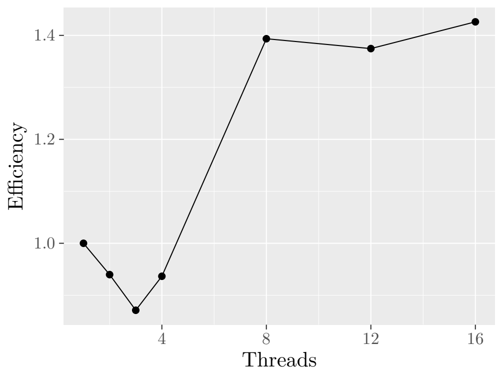
(a) football, Time



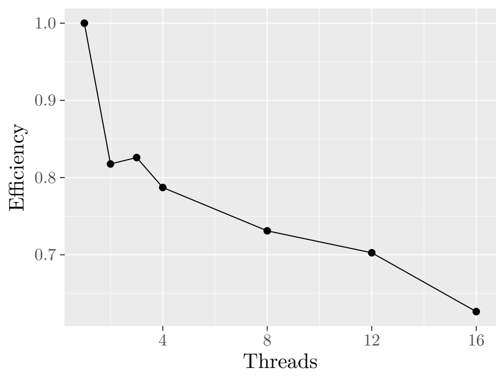
(b) football, Efficiency



(c) dolphins, Time



(d) dolphins, Efficiency



(e) dolphins, Efficiency, corrected

Figure 3.3: Running time and efficiency of the multithreaded Redundant Editing Algorithm. For the efficiency diagrams we used the data points from the highest number of allowed edits that the singlethreaded algorithm could finish within three hours (football: $k = 181$, dolphins: $k = 61$).

Graph	Solved	Edits	Time [s]	Threads
karate	yes	21	0.25	1
lesmis	yes	60	61.28	1
grassweb	yes	34	980.03	1
dolphins	no	$64 < x \leq 72$	8632.32	16
football	no	$187 < x \leq 251$	7594.89	16

Table 3.1: Summary of the results. For each graph we report whether we solved it, the number of edits required, the running time of the fastest selection strategy to produce this result and the number of threads it used. For the graphs we did not solve, we report the highest number of edits for which we were able to show that it does not suffice to solve the graph. The upper bounds are taken from [BHSW15].

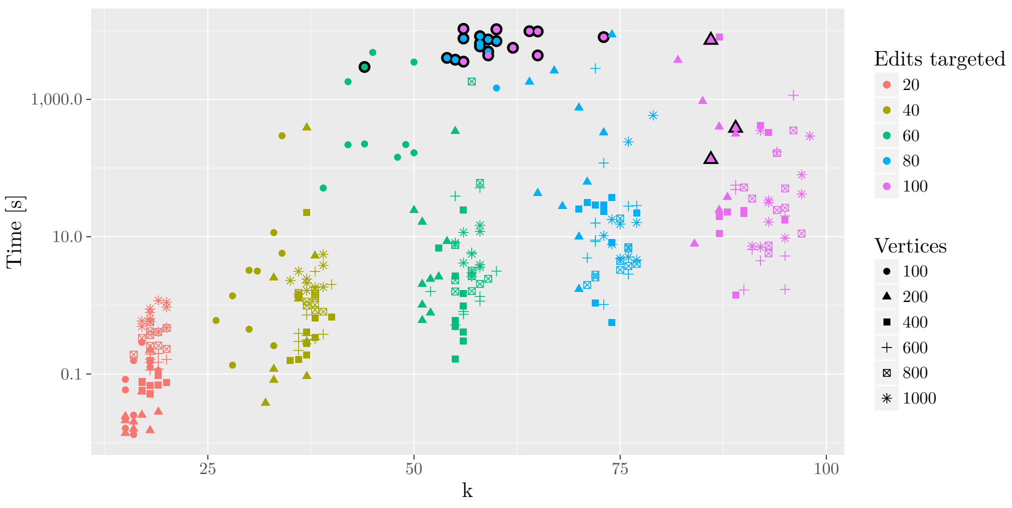


Figure 3.4: Running times of the Redundant Editing Algorithm using single edge editing on the synthetic graphs. The highlighted points indicate graphs that could not be solved within three hours. For these, we show the highest number of edits for which we were able to show that it does not suffice to solve the graph.

of edits needed is the major factor. We observe that the graphs which could not be solved within three hours are those that only have few vertices but targeted a high number of edits. This is because we only calculate a low lower bound for these graphs. Remember that the forbidden subgraphs contributing to the packing used for the lower bound may not overlap. In contrast, in these graph the majority of the created forbidden subgraphs overlap. With increasing graph size the overlap decreases, resulting in better lower bounds and allowing the graphs to be solved within the time limit.

3.7 Analyzing solutions

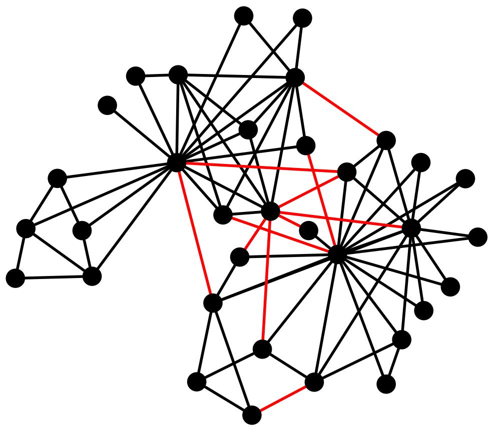
For the real world graphs we are able to solve, we modified our algorithm such that it generates all solutions for a given number of allowed edits and analyzed the solutions.

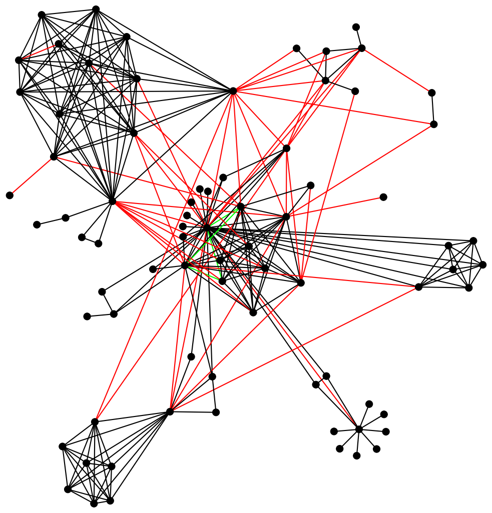
For “karate” our algorithm finds 896 solutions, which are all distinct from each other. (Having nondistinct solutions would disprove Theorem 2.6.) When analyzing the connected components induced by these solutions, we find them largely stable. The vertices are split into two groups that, across all solutions, only form connected components among themselves. These groups are identical to the ones [Zac77] found. In about a quarter of all solutions, these two groups make up one connected component each. In the other solutions, up to five specific vertices split from their respective group's connected component to form their own connected components. Overall there are twelve distinct sets of connected components. We also find that all solutions have eleven edits in common, all of which are deletions. Figure 3.5(a) shows the graph, indicating the common edits. Ten of these deletions are used to split the graph into the aforementioned groups. The remaining deletion and ten additional edits are then used to eliminate the forbidden subgraphs within these groups. Overall, all solutions use 40 distinct edits.

For "lesmis" we find 384 solutions. In contrast to "karate", where we find only a few distinct sets of connected components, we count 192 distinct sets for "lesmis". Each of these sets appears either once or trice and consists of eight to twelve connected components. Analyzing the solutions, we find 73 distinct edits, of which 44 deletions and four insertions are made by all solutions. These 48 common edits split the graph into six connected components. Figure 3.5(b) illustrates the graph, indicating the common edits.

The results for "grassweb" are similar to "lesmis". There are 3006 solutions, inducing 2250 distinct sets of connected components, each appearing once or twice and consisting of eleven to 14 connected components. All solutions have eleven deletions and one insertion in common and use 56 distinct edits. The graph is shown in Figure 3.5(c), indicating the common edits.

In the real world graphs we are able to solve, we find that the number of distinct edits used overall is within the same order of magnitude than the number of edits needed to solve the graph. A significant part of these edits are present in every solution. The remaining edits are then used to split components still containing multiple communities and to destroy forbidden subgraphs within them.


(a) karate


(b) lesmis

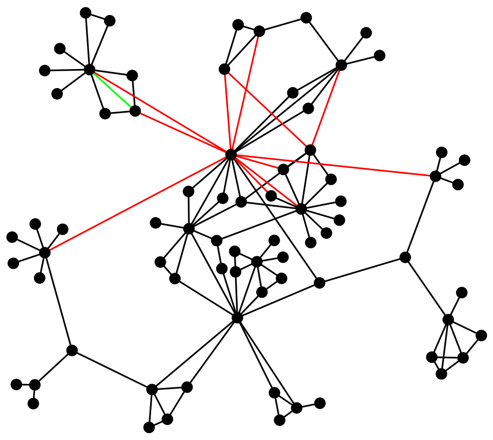

(c) grassweb

Figure 3.5: Real world graphs we are able to solve. The colored edges indicate edits common to all solutions: Red edges are always deleted, green edges are always inserted.

4. Conclusion

In this work, we introduced the Redundant Editing Algorithm and proved its advantage over the Basic Editing Algorithm. We also proposed single edge editing as alternative to the various subgraph selection strategies. In the experimental evaluation, we have shown that both perform significantly better than their respective counterparts.

Depending on the complexity of the input graph, we improved the running time by several orders of magnitude. As result, we are able to solve some previously unsolved graphs exactly and analyze their solutions.

However, our evaluation only concerned the case of the set of forbidden subgraphs $\mathcal{F}$ being $\{P_4, C_4\}$. A different set of forbidden subgraphs should not affect the benefits of the Redundant Editing Algorithm. However, the single edge editing strategy is dependent on the presence of pairs of vertices satisfying its precondition. Our experiments indicate that for $\mathcal{F} = \{P_4, C_4\}$ in most cases there is such a pair and the fallback is only seldom needed. It is not clear if other sets of forbidden subgraphs show similar behavior. This might also depend on the algorithm used for updating the lower bound. In particular, an interesting question is also if there is a lower bound algorithm that guarantees the presence of pairs of vertices satisfying the precondition for a given set of forbidden subgraphs $\mathcal{F}$. If yes, the Edge Editing Problem for that $\mathcal{F}$ can be solved in $O(4^k \operatorname{poly}(n))$.

Due to an implementation detail in the multithreaded version of the editing algorithm, we observed that choosing which forbidden subgraphs participate in a lower bound has a significant impact on the running time of the single edge editing strategy. This indicates that lower bounds provide further possibilities for improving the running time.

Multithreading speeds up the algorithm, although efficiency decreases with more threads. We believe the efficiency can be improved with a better work sharing algorithm that further reduces the communication overhead.

Identifying the edits common to all solutions of a given graph appears to be an interesting problem for heuristics.

Bibliography

[BHSW15] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. Fast quasi-threshold editing. In *Algorithms-ESA 2015*, pages 251–262. Springer, 2015.

[Boh15] Bohlmann, Felix. Graphclustern durch Zerstören langer induzierter Pfade. Bachelor thesis, Technische Universität Berlin, 2015.

[Cai96] Cai, Leizhen. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171–176, 1996.

[DHC95] Hassan Ali Dawah, Bradford A Hawkins, and Michael F Claridge. Structure of the parasitoid communities of grass-feeding chalcid wasps. *Journal of animal ecology*, pages 708–720, 1995.

[Eva10] Tim S Evans. Clique graphs and overlapping communities. *Journal of Statistical Mechanics: Theory and Experiment*, 2010(12):P12037, 2010.

[FH16] Santo Fortunato and Darko Hric. Community detection in networks: A user guide. *Physics Reports*, 659:1–44, 2016.

[For10] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3–5):75–174, 2010.

[GN02] 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.

[JGGW14] Songwei Jia, Lin Gao, Yong Gao, and Haiyang Wang. Anti-triangle centrality-based community detection in complex networks. *IET systems biology*, 8(3):116–125, 2014.

[Knu93] Donald E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing*. Addison-Wesley, 1993.

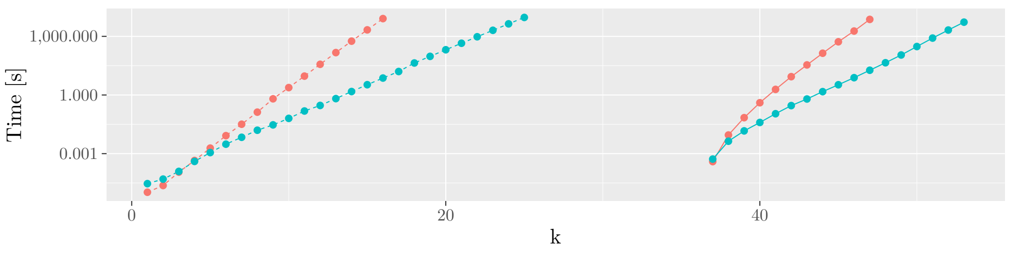
[Lus03] David Lusseau. The emergent properties of a dolphin social network. *Proceedings of the Royal Society of London B: Biological Sciences*, 270(Suppl 2):S186–S188, 2003.

[NG13] James Nastos and Yong Gao. Familial groups in social networks. *Social Networks*, 35(3):439–450, 2013.

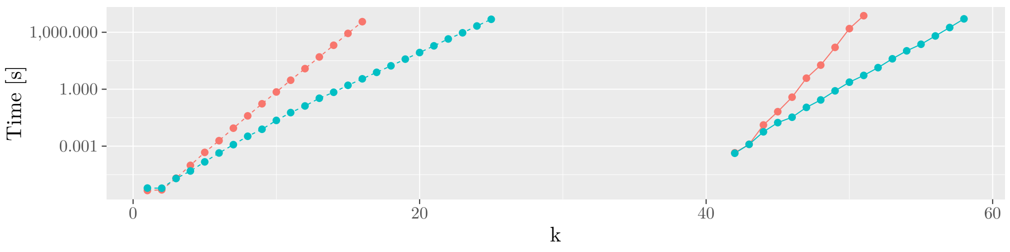
[Sch15] Schoch, Philipp. Editing to (P5, C5)-free Graphs - a Model for Community Detection? Bachelor thesis, Karlsruhe Institute of Technology, October 2015.

[Zac77] Wayne W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33:452–473, 1977.

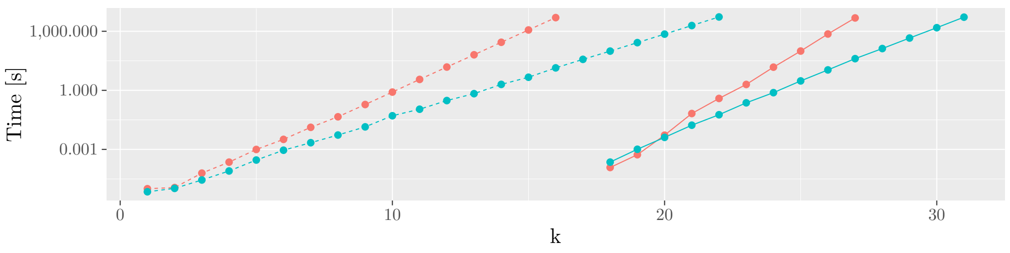
Appendix



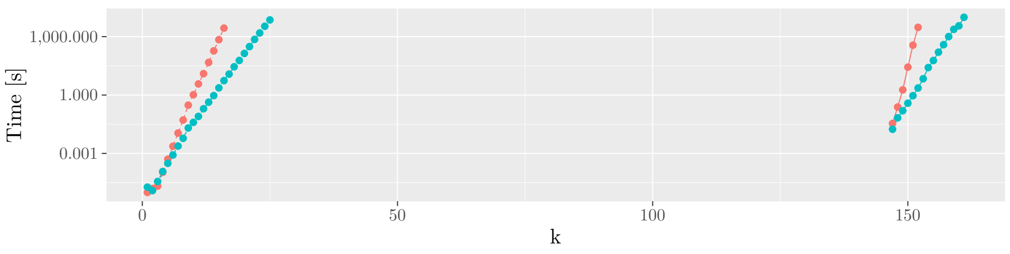
(a) lesmis



(b) dolphins

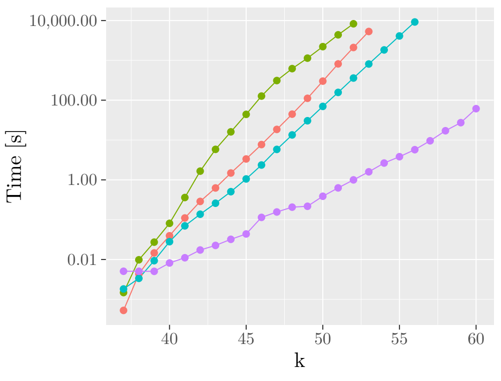


(c) grassweb

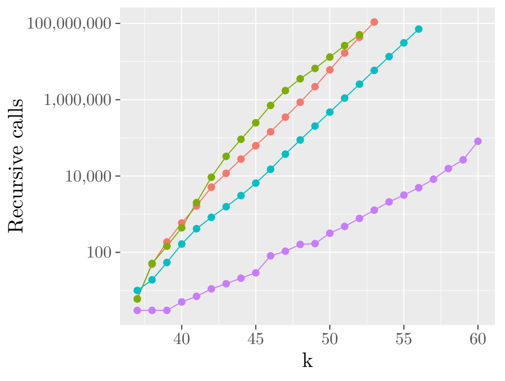


(d) football

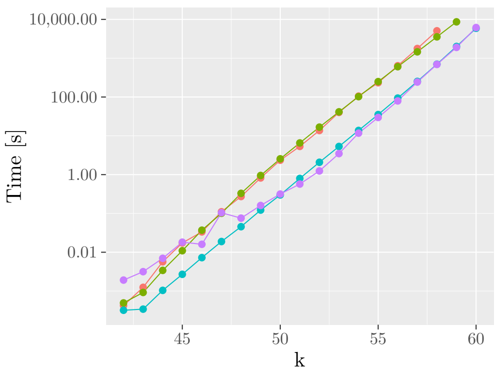
Figure A.1: Running time of the Basic Editing Algorithm (red) and the Redundant Editing Algorithm (blue), in terms of the number of allowed edits (k). The solid lines were obtained by utilizing lower bounds, while for the dashed lines lower bounds were not used.



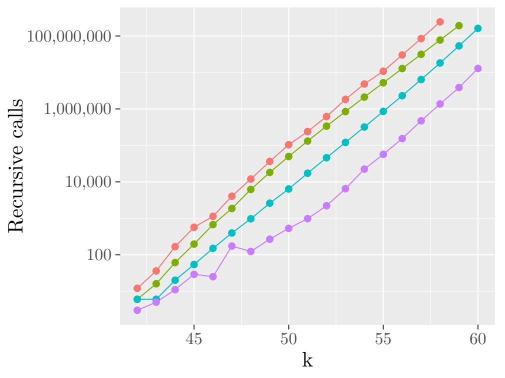
(a) lesmis, Time



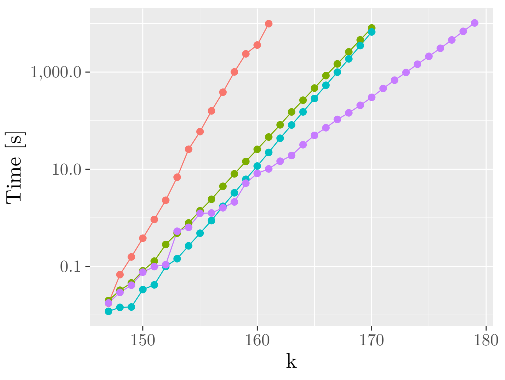
(b) lesmis, Recursive calls



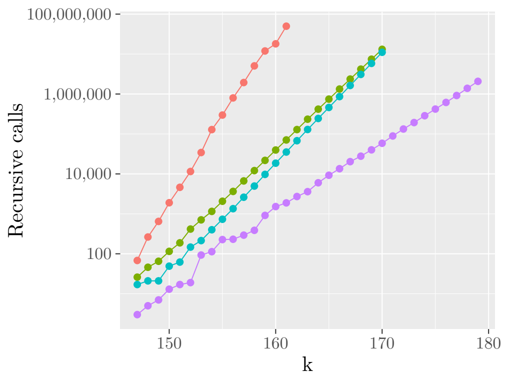
(c) dolphins, Time



(d) dolphins, Recursive calls



(e) football, Time



(f) football, Recursive calls

Figure A.2: Comparison of the forbidden subgraph selection strategies when using the Redundant Editing Algorithm, in terms of the number of allowed edits (k). The colors used for the strategies are: First: red, most edited: blue, anti triangle: green, single edge editing: purple.