On String and Graph Sanitization

Giulia Bernardini
University of Milan, Italy

Huiping Chen
University of Birmingham, UK

Grigorios Loukides
King's College London, UK

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Data sanitization is a process that conceals sensitive patterns from a given dataset. A secondary goal is to not severely harm the utility of the underlying data along this process. We survey some recent advancements on two related data sanitization topics: string and graph sanitization. In particular, we highlight the important contributions of our friend Prof. Roberto Grossi along this journey.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases data privacy, data sanitization, string algorithms, graph algorithms

Digital Object Identifier 10.4230/OASIcs.Grossi.2025.9

Category Research

Funding *Giulia Bernardini*: Member of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).
Solon P. Pissis: Supported in part by the PANGAIA and ALPACA projects that have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively.

Acknowledgements We want to thank Prof. Roberto Grossi for this journey (and more). We would also like to thank the editors for providing us with the opportunity and pleasure to write this paper.

1 Introduction

Disseminating string data is often performed to support applications such as location-based service provision or DNA sequence analysis. However, this dissemination may reveal sensitive patterns that represent some form of private or confidential knowledge (e.g., trips to STD clinics from a string representing a user's visited location or a genetic disorder from a string representing an individual's DNA sequence). To prevent sensitive patterns from being exposed, string data must often undergo *sanitization* [3, 5], a process that conceals sensitive patterns from the data, without, however, severely harming the utility of the data.

Another fundamental data type is graphs. Graphs represent data in domains such as social networks, communication networks, or the Web. In these domains, users form dense or cohesive groups, referred to as *communities*. For example, many of the users who belong to the same political group may also be connected with each other on a social network, creating a community in the graph representing this social network. There is much work in the literature on mining communities from a graph [10]. Our work in [8] studied how to break such communities from a given graph without, however, severely harming the utility of the graph. This process can also be viewed as sanitization, in the sense that communities can convey private or confidential information, such as membership in a political group.

The authors feel extremely privileged to have worked with Prof. Grossi on string and graph sanitization. Our collaboration with Prof. Grossi on these two areas started in December 2018, when the first author was a Ph.D. student at the University of Milano-Bicocca, the second author was a Ph.D. student supervised by the third author at King's College London, and the last author had recently moved from the latter institution to CWI.

Prof. Grossi has made key contributions in the topics of string and graph sanitization. In addition, he shaped not only the definitions of these problems, the hardness results, and the algorithms to solve them but also the practical implementations of the algorithms. In the following, we summarize the most important of these contributions.

In the first step of string sanitization (sanitizing sensitive patterns), Prof. Grossi had the idea of using a special letter to conceal the occurrences of the sensitive patterns in the input string. In particular, this led to a novel string transformation that preserves the order and frequencies of all k-mers of the string. This transformation is very different from existing approaches that deleted [12] or permuted [13] letters of the string, and led to the first approach that provided utility guarantees. Prof. Grossi also played a key role in developing optimal algorithms that employ the aforementioned transformation. In the second step of string sanitization (replacing the occurrences of the special letter), Prof. Grossi came up with an ingenious NP-hardness proof of the problem. This constituted an important step in our work on string sanitization as it opened the way to design efficient fixed-parameter tractable algorithms to tackle the problem; see Section 2.

In graph sanitization, Prof. Grossi had an innovative idea that helped us evaluate the effectiveness of our heuristic algorithms on larger datasets compared to those that our exact algorithm could handle. Specifically, he suggested using a lower bound on the value of the optimal solution that could also be computed efficiently and then comparing the lower bound value with the value output by a heuristic. He also devised an algorithm to compute the lower bound. If the lower bound and the value output by a heuristic are reasonably close, then the heuristic is good because the optimal value lies between them; see Section 3.

Beyond his inestimable contributions to research, Prof. Grossi set an example with how he interacts with others; always with respect, kindness, and true interest and willingness to help. Also, the way Prof. Grossi cheered us up when the results of the work were not as expected was crucial to continue and improve not only the results, but also ourselves.

2 String Sanitization

2.1 The Model: Combinatorial String Dissemination

The Combinatorial String Dissemination (CSD) model was introduced by Bernardini et al [3]. In this model, we have a string W that we would like to disseminate for analysis. Toward effective dissemination, a dissemination that achieves a *good trade-off* between *privacy* and *utility*, we need to specify a set of privacy-related constraints and a set of utility-related properties, to then determine the best possible sequence of edit operations to be applied to W so that the utility-related properties are satisfied subject to the privacy-related constraints.

A specific CSD setting that has been considered by Bernardini et al [3] is the following. The set of constraints (C1) is defined using an integer $k > 0$ and a set of length-k strings, known as the set of *sensitive patterns*; in particular, it consists in strictly forbidding the occurrence of any sensitive pattern in W', the string to be constructed from W. The set of properties is defined using two widely used string properties. The first one (P1) says that the *order* of occurrence of the length-k non-sensitive patterns is preserved in W'; the second (P2) says that the *frequency* of the length-k non-sensitive patterns is also preserved in W'.

This model leads to interesting combinatorial problems on strings [4, 3, 15, 6, 9, 5, 16, 2]. We define a few such problems in the following subsections and briefly describe optimal algorithms for solving them or solid evidence as to why they are unlikely to be solved exactly.

2.2 Sanitizing Sensitive Patterns

In the TFS (Total order, Frequency, Sanitization) problem, we are given a string $W = W[1] \dots W[n]$ of length n over an alphabet Σ, and we are asked to output a *shortest* string $X := W'$ that satisfies C1, P1, and P2. The following example shows a TFS instance.

▶ **Example 1.** Let $W =$ `aabaaaababbbaab`, $k = 4$, and the set of sensitive patterns be $\{\texttt{aaaa}, \texttt{baaa}, \texttt{bbaa}\}$. Then, $X =$ `aabaa#aaababbba#baab`, for some letter `#` $\notin \Sigma$.

(Note that since `#` $\notin \Sigma$, one could use the occurrences of `#` in X to learn about potential occurrences of sensitive patterns in W; see Section 2.3.) The following result is known.

▶ **Theorem 2** ([3]). *The length of X is in $\Theta(kn)$. Given the set of occurrences of sensitive patterns in W, there exists an algorithm that solves TFS in the optimal $\mathcal{O}(kn)$ time.*

The algorithm proceeds by reading W from left to right and constructs X, which is initially empty. When the length-k substring S read is non-sensitive, it merges it with X. Otherwise, it applies two rules. In the first rule (R1), given S, it constructs the string:

$$S' = S[1] \dots S[k-1]\texttt{\#}S[2] \dots S[k].$$

Note that the letter `#` is a special letter that does not belong to the alphabet Σ of W.

▶ **Example 3.** For $S =$ `baaa` and $k = 4$, R1 constructs string $S' =$ `baa#aaa`.

In the second rule (R2), given S' of length $2(k-1)+1$ obtained from R1, the algorithm tries to check if a shorter string S' is possible. In particular, when the $k-1$ letters before `#` are the same as the $k-1$ letters after it, we remove the `#` and the $k-1$ subsequent letters. One can easily notice that the above edit operations on S' will violate neither P1 nor P2.

▶ **Example 4.** Let $S =$ `aaaa` and $k = 4$. R1 will construct string $S' =$ `aaa#aaa`. By applying R2, we get $S' =$ `aaa`, which is a string of length $k - 1 = 3$.

Instead of S, the string S', produced by rules R1 and R2, is merged with X.

▶ **Example 5.** Consider the fragment $F =$ `baaaa` of $W =$ `aa`$\underline{\texttt{baaaa}}$`babbbaab`, with $k = 4$, and the set of sensitive patterns $\{\texttt{aaaa}, \texttt{baaa}, \texttt{bbaa}\}$. For $S =$ `baaa`, R1 will construct string $S' =$ `baa#aaa` resulting in fragment $F' =$ `baa#aaaa`. We will then need to deal with $S =$ `aaaa`. R1 will construct string $S' =$ `aaa#aaa` to replace it. However, applying R2, we get $S' =$ `aaa`, which is shorter. Thus $F =$ `baaaa` will be replaced by fragment $F'' =$ `baa#aaa`.

By carefully implementing R1 and R2, we can construct string X in $\mathcal{O}(kn)$ time. As for the length of X in the worst case, it suffices to construct the de Bruijn sequence of order $k-1$ as the input string W, and assign every other consecutive length-k substring to be a sensitive pattern. Recall that a *de Bruijn sequence* of order k' over an alphabet Σ is a string in which every possible length-k' string over Σ occurs exactly once as a substring. Thus, for such an input string W, X must be of length $\Omega(kn)$ because R2 cannot be applied.

The question that now arises naturally is whether we can hope for a shorter string W' by relaxing the constraints of the TFS problem. In response, Bernardini et al. [3] introduced the following related problem. In the PFS (Partial order, Frequency, Sanitization) problem, we are given a string W of length n, and we are asked to output a *shortest* string $Y := W'$ that satisfies C1, Π1, and P2, where the Π1 property is a relaxation of the P1 property: it replaces the total order by a *partial* order saying that the order of occurrence of the length-k non-sensitive patterns in maximal fragments with no # occurring remains the same.

▶ **Example 6.** Let $W =$ aabaaaababbbaab, $k = 4$, and let the set of sensitive patterns be {aaaa, baaa, bbaa}. Then, $Y =$ aaababbba#aabaab, for some letter # $\neq$ a and # $\neq$ b.

The following result is known.

▶ **Theorem 7** ([3]). *Given the set of occurrences of sensitive patterns in W, there exists an algorithm that solves PFS in the optimal $\mathcal{O}(n + |Y|)$ time.*

(Note that the $|Y|$ term in Theorem 7 accounts for the fact that Y can be longer than W.) We briefly explain how we can arrive at Theorem 7. Consider that we have (a representation of) string X obtained via TFS. If any two maximal #-free fragments of X, called *blocks*, overlap by $k-1$ letters, then we can further apply R2 while still satisfying Π1 and P2.

▶ **Example 8.** Recall that for $W =$ aabaaaababbbaab, $k = 4$, and the set of sensitive patterns {aaaa, baaa, bbaa}, the TFS problem outputs $X =$ aa$\underline{\texttt{baa}}$#aaababbba#$\underline{\texttt{baa}}$b. Observe that blocks aa$\underline{\texttt{baa}}$ and $\underline{\texttt{baa}}$b overlap by $k-1$ letters, so we can apply R2 further.

Now, it might seem that we must solve the famously NP-hard Shortest Common Superstring (SCS) problem [11] on the set of blocks originating from X to construct Y. Fortunately, this is not the case as the *allowed overlaps are of fixed length* $k-1$. To solve this problem, we map the prefix and suffix of length $k-1$ of every block to an integer identifier. We can then ignore the middle part of each block, thus transforming every block to a string of length 2. After this reduction, we can solve SCS on a collection of two-letter strings in linear time [3].

2.3 Hide and Mine

Say we have completed the task in Section 2.2: sanitizing the occurrences of all sensitive patterns by means of a special letter that we represent by #. Although sensitive patterns are no longer present in the sanitized sequence, the occurrences of # reveal the locations where they used to occur. It is not hard to imagine that a malicious adversary could use this information to try and reconstruct the concealed information from the context surrounding the #s. This is because, although the adversary cannot know precisely which special letter has been used in the sanitization process, they know that it is (by definition) not part of the original alphabet, and thus it can be relatively easily identified. Imagine that the dataset consists of a sequence of locations: # could be a non-existing location or a location far away from those in the original sequence. In a sequence of online purchases, # could be an item not sold in a certain online store; in a genomic sequence, # is a letter that does not correspond to any nucleotide base. In all such examples, it is not hard for an attacker to identify # with a simple scan of the database and thus partially retrieve the concealed information.

Therefore, a complete sanitization pipeline should eventually replace the occurrences of # with letters of the original alphabet. This immediately poses a problem: each replacement introduces k new "spurious" occurrences of length-k patterns over the original alphabet, which might influence downstream analysis results based on the frequency of length-k substrings. This kind of analysis extracts length-k substrings that appear at least τ times, for some fixed integer τ, assuming that these τ-frequent patterns carry important information.

The natural problem arising is to decide whether it is possible to replace every occurrence of # with some letter from the original alphabet such that: (i) no sensitive pattern is reintroduced; and (ii) the set of τ-frequent non-sensitive patterns before and after sanitization is the same. We term this problem *Hide and Mine* (HM). Intuitively, HM is hard because replacements are not independent of one another: making a choice at one position could prevent feasible choices from being available at other positions, while this would not have been the case when making a different choice in the first place. But how does one prove its hardness?

The answer came from a brilliant idea by Prof. Grossi: we should reduce from the famous *Bin Packing* [14] problem! More precisely, we should reduce from a variant of the problem called *Unique-Weights Bin Packing* (UWBP). The reduction is far from being trivial: we will provide a friendly, informal description of the proof in the rest of this section.

Reducing UWBP to Hide and Mine

The UWBP problem asks us to decide whether N items can be packed into M bins, each bin with a fixed capacity B that cannot be exceeded: each item i has a given weight w_i, and no two items have the same weight.

▶ **Example 9.** Consider the instance $\mathcal{I}^+$ of UWBP consisting of $M = 3$ bins, each with capacity $B = 7$, and the $N = 5$ items with weights $w_1 = 2, w_2 = 3, w_3 = 4, w_4 = 5, w_5 = 6$. The answer to $\mathcal{I}^+$ is positive: it suffices, for instance, to pack item 5 in one bin, items 2 and 3 in another bin, and items 1 and 4 in a third bin. Now consider a second instance $\mathcal{I}^-$ with the same bins and the same number of items as $\mathcal{I}^+$, but this time the weights are $w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6, w_5 = 7$. The answer to $\mathcal{I}^-$ is negative: there is no way to distribute all the 5 items into the 3 bins without exceeding the capacity of any bin.

The problem UWBP is *strongly* NP-complete, that is, it remains NP-complete even when all of its parameter values (i.e., the bin capacity and the item weights) are polynomially bounded in N, the number of items.

Prof. Grossi's intuition was that bins could be modelled by unique alphabet letters, string gadgets could be constructed to model the event "item j is packed into bin i" and to force each item to be packed in at least one bin, the frequency threshold τ could be tuned to encode the bin capacity B, and the set of sensitive patterns could be chosen to avoid situations in the HM instance that do not correspond to any scenario in the original UWBP instance. Let us now present this idea in detail (but informally) using instance $\mathcal{I}^+$ of Example 9. The alphabet of the HM instance we construct from $\mathcal{I}^+$ consists of the three letters x, y, \$, and a unique letter for each bin: we will denote b_1, b_2 and b_3 the letters corresponding to bin number 1, 2, and 3, respectively, and use # to denote the special character to be replaced in the input string. The value of k (length of patterns) of HM is chosen to be the maximum weight of the input items plus 3: in the case of $\mathcal{I}^+$, we have $k = 6 + 3 = 9$.

Item-in-bin Gadgets

The first class of gadgets consists of a string t_{ij} for each bin i and each item j, containing a single occurrence of #. Each of these string gadgets is paired with a set of sensitive patterns designed in such a way that only two replacements are possible: the first one models the event "pack item j into bin i" by introducing w_j occurrences of a length-k substring specific to bin i; the second one naturally corresponds to item j not being packed in bin i. These string gadgets each have a length of $2k - 1$ and are of the following form:

$$t_{ij} = b_i \underbrace{x \dots x}_{k-1-w_j} \underbrace{b_i \dots b_i}_{w_j - 1} \# \underbrace{b_i \dots b_i}_{k-1}$$

▶ **Example 10.** Consider instance $\mathcal{I}^+$ of Example 9. The gadget t_{13}, modelling the possibility of packing item 3 in the first bin, is the string $t_{13} = b_1xxxxb_1b_1b_1\texttt{\#}b_1b_1b_1b_1b_1b_1b_1b_1$. Four length-$k$ strings, with $k = 9$, are added to the set of sensitive patterns to rule out the possibility of replacing # with $b_2, b_3, \$$ or y in t_{13}; namely, $b_2b_1b_1b_1b_1b_1b_1b_1b_1$, $b_3b_1b_1b_1b_1b_1b_1b_1b_1$, $b_1\$b_1b_1b_1b_1b_1b_1b_1$, and $b_1yb_1b_1b_1b_1b_1b_1b_1$. It should be clear that there are only two replacement options: replacing # by x, corresponding to the choice not to pack the item 3 in the first bin, and replacing it by b_1, corresponding to the opposite choice. Replacement of # with b_1 introduces $w_3 = 4$ occurrence of the string $b_1b_1b_1b_1b_1b_1b_1b_1b_1$, specific to bin 1. The gadget t_{11}, modelling the possibility of packing item 1 in the first bin, is the string $t_{11} = b_1xxxxxxb_1\texttt{\#}b_1b_1b_1b_1b_1b_1b_1b_1$; the associated sensitive patterns are the same as t_{13}. Replacing # with b_1 introduces $w_1 = 2$ occurrences of $b_1b_1b_1b_1b_1b_1b_1b_1b_1$. The gadget t_{23}, modelling the possibility of packing item 3 in the second bin, has the same form as t_{13}, except b_1 is replaced by b_2; the associated sensitive patterns are $b_1b_2b_2b_2b_2b_2b_2b_2b_2$, $b_3b_2b_2b_2b_2b_2b_2b_2b_2$, $b_2\$b_2b_2b_2b_2b_2b_2b_2$ and $b_2yb_2b_2b_2b_2b_2b_2b_2$, leaving b_2 and x as the only possible replacements.

The other gadgets are obtained similarly to those in Example 10; see [5] for the formal definitions.

Each-item-in-some-bin Gadgets

The second class of gadgets consists of a string u_{ij} for each bin i and each item j: these string gadgets also contain an occurrence of # each and are paired with some sensitive patterns. The role of these gadgets is to ensure that each item $\bar{\jmath}$ is packed in some bin, corresponding to replacing # with b_i in at least one of the gadgets $t_{i\bar{\jmath}}$. The latter is enforced by designing $u_{i\bar{\jmath}}$ and the corresponding sensitive patterns so that: (i) # can only be replaced by either x or y; (ii) replacing # by x in $u_{i\bar{\jmath}}$ introduces an occurrence of a length-k string that is also introduced when replacing # by x in $t_{i\bar{\jmath}}$; and choosing the frequency threshold τ in such a way that (iii) if # is replaced by x in $t_{i\bar{\jmath}}$, then # in $u_{i\bar{\jmath}}$ must be replaced by y; and (iv) # cannot be replaced by y in $u_{i\bar{\jmath}}$ for all i, as otherwise, the frequency threshold is exceeded. These string gadgets each have a length of $2k - 1$ and are of the following form:

$$u_{ij} = b_i\, \underbrace{x \ldots x}_{k-1-w_j}\, \underbrace{b_i \ldots b_i}_{w_j - 1}\, \texttt{\#}\, \underbrace{y \ldots y}_{w_j}\, \underbrace{x \ldots x}_{k-w_j-2}\, y$$

▶ **Example 11.** Consider again instance $\mathcal{I}^+$ of Example 9. For bin 1 and item 1 we have $u_{11} = b_1xxxxxxb_1\texttt{\#}yyxxxxxy$ and four associated sensitive patterns: $b_1yyxxxxxy$, $b_2yyxxxxxy$, $b_3yyxxxxxy$, and $b_1\$yyxxxxx$, forbidding replacing # with any letter but letters x or y. Note that replacing # by x in u_{11} introduces an occurrence of the length-k string $b_1xxxxxxb_1x$, which is also introduced when replacing # by x in t_{11} (thus not packing item 1 in bin 1: see Example 10). For bin 2 and item 3 we have $u_{23} = b_2xxxxb_2b_2b_2\texttt{\#}yyyyxxxy$ and the associated sensitive patterns: $b_1yyyyxxxy$, $b_2yyyyxxxy$, $b_3yyyyxxxy$, and $b_2\$yyyyxxx$.

The other gadgets u_{ij} and their associated sensitive patterns can be obtained similarly to those in Example 11; see [5] for the formal definitions.

Frequency Threshold and Final Construction

It would be tempting to set the frequency threshold τ to the bin capacity B, since when # is replaced by b_i in a gadget t_{ij} it creates exactly w_j occurrences of $b_i \cdots b_i$; exceeding the capacity of bin i should correspond to introducing more than B occurrences of $b_i \cdots b_i$. However, to ensure that the only feasible solutions to HM correspond to packing each item in some bin, we need to bound the number of allowed occurrences of other non-sensitive patterns like those introduced by replacing # in gadgets u_{ij}, which is linked to the number M of bins rather than to their capacity B. For this reason, in the reduction, we set $\tau = \max\{B, M\} + 1$: to maintain the correspondence between the number of occurrences of length-k strings $b_i \cdots b_i$ and bin capacity, we append to the final string constructed by the reduction $\tau - B - 1$ occurrences of $b_i \cdots b_i$ for each i.

The final string is obtained by concatenating all gadgets t_{ij} and u_{ij} interleaved with \$\$, followed by an appropriate number of occurrences of some of the strings introduced by replacements in the gadgets. For a formal proof of this reduction, we refer the reader to [5].

▶ **Theorem 12** ([5]). *The HM problem is strongly NP-complete.*

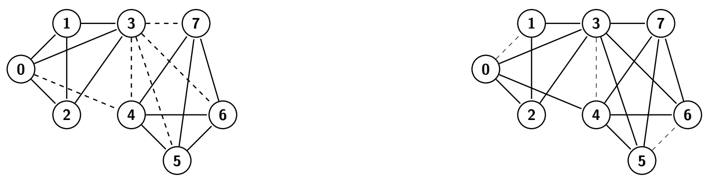
3 Graph Sanitization

3.1 The Community Breaking Problem

The community structure is a fundamental property of a graph, and understanding how this structure can be maintained or disrupted is crucial. In our graph sanitization work, we introduced the following general *Community Breaking* (CB) problem: Given an undirected graph $G(V, E)$, a set of nodes $U \subseteq V$, and a notion of community, identify a smallest subset E' of E, so that no community in $G' = G(V, E \setminus E')$ contains a node in U.

In [8], we considered a specific community notion, namely the k-truss. A *k-truss* is a subgraph of a graph in which every edge is part of at least $k - 2$ triangles formed entirely within the subgraph; see Fig. 1a for an example.

Building on the CB problem and the k-truss notion, we defined *MIN-k-TBS* (Minimum k-Truss Breaking Set): Given an undirected graph $G(V, E)$ and a parameter k, find a smallest subset E' of E such that $G(V, E \setminus E')$ contains no k-truss. MIN-k-TBS is obtained from the CB problem by considering communities based on the notion of k-truss and $U = V$.



(a) 4-truss. **(b)** 4-truss-free graph.

Figure 1 (a) The subgraph induced by the solid edges is a 4-truss because every edge of the subgraph is contained in at least $4 - 2 = 2$ triangles of the subgraph. (b) The graph obtained after deleting the set $\{(0, 1), (3, 4), (5, 6)\}$ of (dashed) edges contains no 4-truss.

▶ **Example 13.** An optimal solution to MIN-k-TBS with $k = 4$ in Fig. 1b is the set of (dashed) edges $E' = \{(0, 1), (3, 4), (5, 6)\}$: deleting these edges leads to a graph with no 4-truss and deleting any fewer edges leads to a graph that contains a 4-truss.

The MIN-k-TBS problem is *intuitively* challenging: there are up to $2^{|E|}$ edge subsets that one may consider; and k-trusses have a hierarchical structure. For example, a $(k+i)$ truss, for any k and $i \geq 0$, is also a k-truss and contains at least $\binom{k+i}{k}$ smaller k-trusses. Finding a minimum set of edges to delete to make a graph triangle-free is known to be NP-hard [17], and that is exactly the MIN-3-TBS problem. Assuming the Exponential Time Hypothesis, we cannot even solve this problem in $2^{o(|E'|)} \cdot n^{O(1)}$ time [1]. We proved that MIN-k-TBS is also NP-hard for $k > 3$ using a reduction from MIN-3-TBS.

▶ **Theorem 14** ([8])**.** *For every $k \geq 3$, MIN-k-TBS is NP-hard.*

Here is a brief sketch of our proof. We prove that for every $k \geq 3$, the MIN-k-TBS problem is NP-hard by reducing from the known NP-hard MIN-3-TBS problem, which involves making a graph triangle-free. In our reduction, for each triangle in the original graph, we add $k-3$ extra nodes to form a clique with the triangle's vertices. This construction ensures that a k-truss exists in the new graph G_k if and only if a triangle exists in the original graph. It follows that solving MIN-3-TBS for G is equivalent to solving MIN-k-TBS for G_k. Therefore, the problem MIN-k-TBS is NP-hard for every $k \geq 3$.

We also presented an exact exponential-time algorithm to solve the MIN-k-TBS problem. The algorithm first enumerates all minimal k-trusses (i.e., a k-truss for which removing any single edge would destroy the k-truss) and then computes a minimum transversal of the corresponding hypergraph. Consequently, its overall time complexity is exponential in the number $|E|$ of edges. To practically improve on the efficiency of this algorithm, we developed three heuristics, $\mathrm{MBH_S}$, $\mathrm{MBH_C}$, and SNH, which run in polynomial time; see [8] for details.

3.2 Lower Bound on the Size of OPT

Due to the exponential-time complexity of our exact algorithm, computing the optimal solution is a heavy task even for small graphs with a few hundred nodes. Prof. Grossi designed an algorithm to compute a lower bound on the size of the optimal solution, which facilitates the evaluation of our heuristic algorithms.

The main idea is to use cliques as a "proxy" for trusses. Since a k-clique (i.e., a complete subgraph consisting of k vertices, where every pair of vertices is directly connected by an edge) is a k-truss, we must at the very least make the input graph G free from k-cliques to solve MIN-k-TBS. The algorithm works in three phases:

1. Computes an *edge clique partition* of G. We partition the graph into edge-disjoint cliques. This partitioning ensures that each edge belongs to exactly one clique, allowing us to analyse dense substructures individually.
2. Applies the best available lower bound on each clique. For each clique, we estimate the minimum number of edges that must be removed to eliminate all k-trusses within it. We employ two complementary approaches: (1) a bound derived from Turan's theorem [7], which limits the maximum number of edges in a graph that avoids a clique of size k; and (2) a triangle-based approach, as each edge in a k-truss must belong to at least k -2 triangles – counting the number of triangles gives an estimate of the minimum deletions required.
3. Outputs a lower bound on the size of the optimal solution by summing the bounds of the cliques. This is possible because the cliques are all edge-wise disjoint.

The overall time complexity of our LB algorithm is $O(q\Delta^2|E|)$, where q, Δ, and $|E|$ are the size of the largest clique, the highest degree, and the number of edges in G, respectively.

Interestingly, the lower bound produced by the above algorithm helped us demonstrate that our heuristics were very competitive to the exact algorithm on large instances in terms of solution quality, even if we were unable to run the exact algorithm on these instances!

References

1 N. R. Aravind, R. B. Sandeep, and Naveen Sivadasan. Dichotomy results on the hardness of H-free edge modification problems. *SIAM Journal on Discrete Mathematics*, 31(1):542–561, 2017. doi:10.1137/16M1055797.

2 Giulia Bernardini, Philip Bille, Inge Li Gørtz, and Teresa Anna Steiner. Differentially private substring and document counting. *Proc. ACM Manag. Data*, 3(2):95:1–95:27, 2025. doi:10.1145/3725232.

3 Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giovanna Rosone, and Michelle Sweering. Combinatorial algorithms for string sanitization. *ACM Trans. Knowl. Discov. Data*, 15(1):8:1–8:34, 2021. doi:10.1145/3418683.

4 Giulia Bernardini, Huiping Chen, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Leen Stougie, and Michelle Sweering. String sanitization under edit distance. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 161 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CPM.2020.7.

5 Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering. Hide and mine in strings: Hardness, algorithms, and experiments. *IEEE Trans. Knowl. Data Eng.*, 35(6):5948–5963, 2023. doi:10.1109/TKDE.2022.3158063.

6 Giulia Bernardini, Alberto Marchetti-Spaccamela, Solon P. Pissis, Leen Stougie, and Michelle Sweering. Constructing strings avoiding forbidden substrings. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 191 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.9.

7 B. Bollobás. *Extremal graph theory.* Courier Corporation, 2004.

8 Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, and Michelle Sweering. On breaking truss-based and core-based communities. *ACM Trans. Knowl. Discov. Data*, 18(6):135:1–135:43, 2024. doi:10.1145/3644077.

9 Huiping Chen, Changyu Dong, Liyue Fan, Grigorios Loukides, Solon P. Pissis, and Leen Stougie. Differentially private string sanitization for frequency-based mining tasks. In *IEEE International Conference on Data Mining (ICDM)*, pages 41–50. IEEE, 2021. doi:10.1109/ICDM51629.2021.00014.

10 Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. A survey of community search over big graphs. *VLDB J.*, 29(1):353–392, 2020. doi:10.1007/S00778-019-00556-X.

11 John Gallant, David Maier, and James A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980. doi:10.1016/0022-0000(80)90004-5.

12 Aris Gkoulalas-Divanis and Grigorios Loukides. Revisiting sequential pattern hiding to enhance utility. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1316–1324, 2011. doi:10.1145/2020408.2020605.

13 Robert Gwadera, Aris Gkoulalas-Divanis, and Grigorios Loukides. Permutation-based sequential pattern hiding. In *IEEE 13th International Conference on Data Mining*, pages 241–250, 2013. doi:10.1109/ICDM.2013.57.

14 Edward G. Coffman Jr., Gábor Galambos, Silvano Martello, and Daniele Vigo. Bin packing approximation algorithms: Combinatorial analysis. In Ding-Zhu Du and Panos M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 151–207. Springer, 1999. doi:10.1007/978-1-4757-3023-4_3.

15 Takuya Mieno, Solon P. Pissis, Leen Stougie, and Michelle Sweering. String sanitization under edit distance: Improved and generalized. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 191 of *LIPIcs*, pages 19:1–19:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.19.

16 Teresa Anna Steiner. Differentially private approximate pattern matching. In *15th Innovations in Theoretical Computer Science Conference, (ITCS)*, volume 287 of *LIPIcs*, pages 94:1–94:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ITCS.2024.94.

17 M. Yannakakis. Edge-deletion problems. *SIAM Journal on Computing*, 10(2):297–309, 1981. doi:10.1137/0210021.