String Sanitization Under Edit Distance: Improved and Generalized

Takuya Mieno*[1], Solon P. Pissis[2], Leen Stougie†[3], and Michelle Sweering‡[4]

[1]Department of Informatics, Kyushu University and Japan Society for the Promotion of Science, Japan, takuya.mieno@inf.kyushu-u.ac.jp
[2]CWI and Vrije Universiteit, Amsterdam, The Netherlands, solon.pissis@cwi.nl
[3]CWI and Vrije Universiteit, Amsterdam, The Netherlands, leen.stougie@cwi.nl
[4]CWI, Amsterdam, The Netherlands, michelle.sweering@cwi.nl

July 17, 2020

Abstract

Let W be a string of length n over an alphabet Σ, k be a positive integer, and $\mathcal{S}$ be a set of length-k substrings of W. The **ETFS** problem asks us to construct a string X_{ED} such that: (i) no string of $\mathcal{S}$ occurs in X_{ED}; (ii) the order of all other length-k substrings over Σ is the same in W and in X_{ED}; and (iii) X_{ED} has minimal edit distance to W. When W represents an individual's data and $\mathcal{S}$ represents a set of confidential patterns, the **ETFS** problem asks for transforming W to preserve its privacy and its utility [Bernardini et al., ECML PKDD 2019].

ETFS can be solved in $\mathcal{O}(n^2k)$ time [Bernardini et al., CPM 2020]. The same paper shows that **ETFS** cannot be solved in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$, unless the Strong Exponential Time Hypothesis (SETH) is false. Our main results can be summarized as follows:

- An $\mathcal{O}(n^2 \log^2 k)$-time algorithm to solve **ETFS**.
- An $\mathcal{O}(n^2 \log^2 n)$-time algorithm to solve **AETFS**, a generalization of **ETFS** in which the elements of $\mathcal{S}$ can have arbitrary lengths.

Our algorithms are thus optimal up to polylogarithmic factors, unless SETH fails. Let us also stress that our algorithms work under edit distance with arbitrary weights at no extra cost.

In order to arrive at these results, we develop new techniques for computing a variant of the standard dynamic programming (DP) table for edit distance. In particular, we simulate the DP table computation using a directed acyclic graph (DAG) in which every node is assigned to a smaller DP table. We then focus on redundancy in these DP tables and exploit a tabulation technique according to dyadic intervals to obtain an optimal alignment in $\tilde{\mathcal{O}}(n^2)$ total time[5]. As a bonus, we show how to modify some known techniques, which speed up the standard edit distance computation, to be applied to our problems. Beyond string sanitization, our techniques may inspire solutions to other problems related to regular expressions or context-free grammars.

*Supported by the JSPS KAKENHI Grant Number JP20J11983.
†Supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003.
‡Supported by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003.

[5]The notation $\tilde{\mathcal{O}}(f)$ denotes $\mathcal{O}(f \cdot \mathrm{polylog}(f))$.

1 Introduction

Let us start with an example to ensure that the reader is familiar with the basic motivation behind the computational problem investigated here. Consider a sequence W of items representing a user's on-line purchasing history. Further consider a fragment (or a subsequence) of W denoting that the user has first purchased unscented lotions and zinc/magnesium supplements and then unscented soaps and cotton balls in extra-large bags. By having access to W and to the respective domain knowledge, one can infer that the user is probably pregnant and close to the delivery date.

Data sanitization, also known as *knowledge hiding*, is a privacy-preserving data mining process aiming to prevent the mining of confidential knowledge from published datasets; it has been an active area of research for the past 25 years [14, 34, 36, 19, 37, 20, 1, 2, 18, 21, 30, 10, 7]. Informally, it is the process of disguising (hiding) confidential information in a given dataset. This process typically incurs some data utility loss that should be minimized. Thus, naturally, privacy constraints and utility objective functions lead to the formulation of combinatorial optimization problems. From a fundamental perspective, it is thus relevant to be able to establish some formal guarantees.

A string W is a sequence of letters over some alphabet Σ. Individuals' data, in domains ranging from web analytics to transportation and bioinformatics, are typically represented by strings. For example, when Σ is a set of items, W can represent a user's purchasing history [5]; when Σ is a set of locations, W can represent a user's location profile [38]; and when Σ is the DNA alphabet, W can represent a patient's genome sequence [26]. Such strings commonly fuel up a gamut of applications; in particular, frequent pattern mining applications [4]. For example, frequent pattern mining from location history data facilitates route planning [13]; frequent pattern mining from market-basket data facilitates business decision making [5]; frequent pattern mining from genome sequences facilitates clinical diagnostics [26]. To support these applications in a privacy-preserving manner, individual sequences are often being disseminated after they have been sanitized.

Towards this end, Bernardini et al. have recently formalized the following string sanitization problem under edit distance [8]. Let W be a string of length n over an alphabet Σ, k be a positive integer, and $\mathcal{S}$ be a set of length-k substrings of W. Set $\mathcal{S}$ is conceptually seen as an *antidictionary*: a set of *sensitive patterns* modelling private or confidential information. The **ETFS** problem (**E**dit distance, **T**otal order, **F**requency, **S**anitization) asks us to construct a string X_{ED} such that: (i) no string of $\mathcal{S}$ occurs in X_{ED}; (ii) the order of all other length-k substrings over Σ is the same in W and in X_{ED}; and (iii) X_{ED} has minimal edit distance to W. In order to obtain a feasible solution string, we may need to extend Σ to $\Sigma_{\#} = \Sigma \sqcup \{\#\}$, which includes a special letter $\# \notin \Sigma$.

Example 1. *Let* $W =$ `ecabaaaaabbbadf` *over alphabet* $\Sigma = \{$`a`, `b`, `c`, `d`, `e`, `f`$\}$ *be the input string. Further let* $k = 3$ *and the set of sensitive patterns be* $\mathcal{S} = \{$`aba`, `baa`, `aaa`, `aab`, `bba`$\}$*. Consider the following three feasible (sanitized) strings:* $X_{\mathrm{TR}} =$ `ecab#abb#bbb#badf`*,* $X_{\mathrm{MIN}} =$ `ecabbb#badf` *and* $X_{\mathrm{ED}} =$ `ecab#aa#abbb#badf`*. All three strings contain* no sensitive pattern *and preserve the* total order *and thus the* frequency *of all non-sensitive length-*3 *patterns of* W*:* X_{TR} *is the trivial solution of interleaving the non-sensitive length-*3 *patterns of* W *with #;* X_{MIN} *is the* shortest *possible such string; and* X_{ED} *is a string* closest *to* W *in terms of edit distance.*

A simple $\mathcal{O}(n^2k|\Sigma|)$-time solution [8] to **ETFS** can be obtained via employing approximate regular expression matching. Consider the regular expression R that encodes all feasible solution strings. The size of R is $\mathcal{O}(nk|\Sigma|)$. By aligning W and R using the standard quadratic-time algorithm [32], we obtain an optimal solution X_{ED} in $\mathcal{O}(n^2k|\Sigma|)$ time for **ETFS**. Bernardini et al. showed that this can be improved to $\mathcal{O}(n^2k)$ time [9]. Let us informally describe their algorithm. (A formal description of their algorithm follows in the preliminaries section.) We use a dynamic programming (DP) table similar to the standard edit distance algorithm. We write the letters of

the input string W on the top of the first row. Since we do not know the exact form of the output string X_{ED}, we write the non-sensitive length-k patterns to the left of the first column interleaved by special # letters. We then proceed to fill this table using recursive formulae. The formulae are more involved than the edit distance ones to account for the possibility to *merge* consecutive non-sensitive patterns (e.g., `eca` and `cab` are merged to `ecab` in Example 1) and to expand the #'s into longer gadgets that may contain up to $k-1$ letters from Σ (e.g., `#aa#` in Example 1). Once the DP table is filled, we construct an X_{ED} by tracing back an optimal alignment.

Bernardini et al. also showed, via a reduction from the weighted edit distance problem [11], that **ETFS** cannot be solved in $\mathcal{O}(n^{2-\delta})$ time, for any $\delta > 0$, unless the strong exponential time hypothesis (SETH) [22, 23] is false. We were thus also motivated to match this lower bound.

Our Results and Techniques Our first main result is the following.

Theorem 1. *The **ETFS** problem can be solved in $\mathcal{O}(n^2 \log^2 k)$ time.*

We also consider a generalized version of **ETFS**, which we denote by **AETFS** (**A**rbitrary lengths, **E**dit distance, **T**otal order, **F**requency, **S**anitization). The only difference in **AETFS** with respect to **ETFS** is that $\mathcal{S}$ can contain elements (sensitive patterns) of arbitrary lengths. This generalization is evidently more useful as it drops the restriction of fixed-length sensitive patterns; it also turns out to be algorithmically much more challenging. In both **ETFS** and **AETFS**, we make the standard assumption that substrings of W are represented as intervals over $[0, n-1]$, and thus each element in $\mathcal{S}$ has an $\mathcal{O}(1)$-sized representation. We further assume that $\mathcal{S}$ satisfies the properties of *closure* and *minimality*, which in turn ensure that $\mathcal{S}$ has an $\mathcal{O}(n)$-sized representation.

Example 2. *Consider the same input string $W =$ `ecabaaaaabbbadf` as in Example 1. Further let $k = 3$ and the set of sensitive patterns be $\mathcal{S} = \{$`aba`, `aa`, `abbba`$\}$. Then, string $Y_{\mathrm{ED}} =$ `ecab#abb#bbbadf` is a feasible string and is a closest to W in terms of edit distance. Notice that, we cannot merge all of the three consecutive non-sensitive patterns `abb`, `bbb`, and `bba` into one since it will result in an occurrence of the sensitive pattern `abbba`; we thus rather create `abb#bbba`.*

Our second main result is the following.

Theorem 2. *The **AETFS** problem can be solved in $\mathcal{O}(n^2 \log^2 n)$ time.*

Our algorithms are thus optimal up to polylogarithmic factors, unless SETH fails. Let us also stress that our algorithms work under edit distance with arbitrary weights at no extra cost.

Let us describe the main ideas behind the new techniques we develop. As in Example 2, a sensitive pattern of length greater than k might be generated by merging multiple non-sensitive patterns. In **AETFS**, we have to consider avoiding such *invalid* merge operations. If we enumerate all *valid* combinations of merging non-sensitive patterns, and run the DP for **ETFS** for all the cases, then we can obtain an optimal solution to **AETFS**. Our main idea for reducing the time complexity is to carefully maintain a directed acyclic graph (DAG) for representing all such valid combinations. We first construct the DAG, and then plug a small DP table into each node of the DAG. This technique gives us an $\mathcal{O}(n^3)$-time solution to **AETFS**. To achieve $\tilde{\mathcal{O}}(n^2)$ time, we focus on redundancy in the DP tables. When the size of the DP tables is large, there must be multiple sub-tables corresponding to the same pair of strings. Before propagating, we precompute lookup table structures of size $\mathcal{O}(n^2 \log^2 n)$ according to dyadic intervals on $[0, n-1]$. To this end, we modify the data structure proposed in [12]. Then, we decompose the DP tables into sub-tables according to these dyadic intervals. We compute only boundaries of such sub-tables using the precomputed lookup table structures, and thus, we obtain an optimal alignment for **AETFS** in

$\mathcal{O}(n^2 \log^2 n)$ total time. By applying the same technique to **ETFS**, we obtain an $\mathcal{O}(n^2 \log^2 k)$-time solution, which improves the state of the art by a factor of $k/\log^2 k$ [9]. As a bonus, we show how to (non-trivially) modify the Four Russians [6] and the LZ78-factorization [39] techniques, which speed up the standard edit distance computation [31, 15], to be applied to **ETFS** and **AETFS**.

In a nutshell, our main technical contribution is that we manage to align a string of length n and a specific regular expression of size $\Omega(nk|\Sigma|)$ in $\tilde{\mathcal{O}}(n^2)$ time. We can also view the solution spaces of **ETFS** and **AETFS** as context-free languages. The main idea of our **AETFS** algorithm is to first preprocess a set of non-terminals N, such that we can later use them in $\mathcal{O}(n)$ time each. We then write the context-free language as a new language, which is accepted by a Deterministic Acyclic Finite State Automaton (DASFA), taking the set N as its terminals. In this paper, we develop several techniques to reduce the size of the DAFSA (cf. DAG) to $\tilde{\mathcal{O}}(n)$ and efficiently precompute the set N (cf. lookup tables) in $\tilde{\mathcal{O}}(n^2)$ time. Thus, beyond string sanitization, our techniques may inspire solutions to other problems related to regular expressions or context-free grammars.

Paper Organization Section 2 introduces the basic definitions and notation used throughout, and also provides a summary of the currently fastest algorithm for **ETFS** [9]. In Section 3, we describe our lookup table structures. In Section 4, we present the $\mathcal{O}(n^3)$-time algorithm for solving **AETFS**. This algorithm is refined to an $\tilde{\mathcal{O}}(n^2)$-time algorithm, which is described in Section 5. Along the way, in Section 5, we also describe an $\tilde{\mathcal{O}}(n^2)$-time algorithm for **ETFS**. In Section 6 we show how to utilize the Four Russians and the LZ78-factorization techniques for our problems.

2 Preliminaries

Strings An *alphabet* Σ is a finite set of elements called *letters*. Let $S = S[0]S[1]\dots S[n-1]$ be a *string* of length $|S| = n$ over an alphabet Σ of size $\sigma = |\Sigma|$. Let $\Gamma = \{\ominus, \oplus, \otimes\}$ be a set of special letters with $\Gamma \cap \Sigma = \emptyset$. By Σ^* we denote the set of all strings over Σ, and by Σ^k the set of all length-k strings over Σ. For two indices $0 \le i \le j \le n-1$, $S[i\,..\,j] = S[i]\dots S[j]$ is the *substring* of S that starts at position i and ends at position j of S. By ε we denote the *empty string* of length 0. A *prefix* of S is a substring of the form $S[0\,..\,j]$, and a *suffix* of S is a substring of the form $S[i\,..\,n-1]$. Given two strings U and V we say that U has a *suffix-prefix overlap* of length $\ell > 0$ with V if and only if the length-ℓ suffix of U is equal to the length-ℓ prefix of V, i.e., $U[|U|-\ell\,..\,|U|-1] = V[0\,..\,\ell-1]$.

We fix a string W of length n over an alphabet Σ. We assume that $\Sigma = \{1, \dots, n^{\mathcal{O}(1)}\}$. If this is not the case, we use perfect hashing [17] to hash $W[i]$, for all $i \in [1, n]$, and obtain another string over $\Sigma = \{1, \dots, n\}$ in $\mathcal{O}(n)$ time with high probability or in $\mathcal{O}(n \log n)$ time deterministically via sorting. We consider the obtained string to be W. We also fix an integer $0 < k < n$. Unless specified otherwise, we refer to a length-k string or a *pattern* interchangeably. An occurrence of a pattern is uniquely defined by its starting position. Let $\mathcal{S}_k$ be the set representing the sensitive patterns as starting positions over $\{0, \dots, n-k\}$ with the following closure property: for every $i \in \mathcal{S}_k$, any j for which $W[j\,..\,j+k-1] = W[i\,..\,i+k-1]$, must also belong to $\mathcal{S}_k$. That is, if an occurrence of a pattern is in $\mathcal{S}_k$, then all its occurrences are in $\mathcal{S}_k$. A substring $W[i\,..\,i+k-1]$ of W is called *sensitive* if and only if $i \in \mathcal{S}_k$; $\mathcal{S}_k$ is thus the complete set of occurrences of sensitive patterns. The difference set $\mathcal{I} = \{0, \dots, n-k\} \setminus \mathcal{S}_k$ is the set of occurrences of *non-sensitive* patterns.

For any substring U, we denote by $\mathcal{I}_U$ the set of occurrences in U of non-sensitive length-k strings over Σ. (We have that $\mathcal{I}_W = \mathcal{I}$.) We call an occurrence i the *t-predecessor* of another occurrence j in $\mathcal{I}_U$ if and only if i is the largest element in $\mathcal{I}_U$ that is less than j. This relation induces a *strict total order* on the occurrences in $\mathcal{I}_U$. We call a subset $\mathcal{J}$ of $\mathcal{I}_U$ a *t-chain* if for

all elements in $\mathcal{J}$ except the minimum one, their t-predecessor is also in $\mathcal{J}$. For two strings U and V, chains $\mathcal{J}_U$ and $\mathcal{J}_V$ are *equivalent*, denoted by $\mathcal{J}_U \equiv \mathcal{J}_V$, if and only if $|\mathcal{J}_U| = |\mathcal{J}_V|$ and $U[u\,..\,u+k-1] = V[v\,..\,v+k-1]$, where u is the j-th smallest element of $\mathcal{J}_U$ and v is the j-th smallest of $\mathcal{J}_V$, for all $j \leq |\mathcal{J}_U|$.

Given two strings U and V the *edit distance* $d_{\mathrm{E}}(U,V)$ is defined as the minimum number of elementary edit operations (letter insertion, deletion, or substitution) that transform one string into the other. Each edit operation can also be associated with a cost: a fixed positive value. Given two strings U and V the *weighted edit distance* $d_{\mathrm{WE}}(U,V)$ is defined as the minimal total cost of a sequence of edit operations to transform one string into the other. We assume throughout that the three edit operations all have unit weight. However, our algorithms also work for arbitrary weights at no extra cost. The standard algorithm to compute the edit distance between two strings U and V [29] works by creating a $(|U|+1)\times(|V|+1)$ DP table D with $D[i][j] = d_{\mathrm{E}}(U[0\,..\,i-1], V[0\,..\,j-1])$. The sought edit distance is thus $d_{\mathrm{E}}(U,V) = D[|U|][|V|]$. Since we compute each table entry from the entries to the left, top and top-left in $\mathcal{O}(1)$ time, the algorithm runs in $\mathcal{O}(|U|\cdot|V|)$ time. Moreover, we can find an optimal (minimum cost) alignment by tracing back through the table.

The ETFS Problem We formally define **ETFS**, one of the problems considered in this paper.

Problem 1 (**ETFS**). *Given a string W of length n, an integer $k > 1$, and a set $\mathcal{S}_k$ (and thus set $\mathcal{I}$), construct a string X_{ED} which is at minimal (weighted) edit distance from W and satisfies:*

C1 *X_{ED} does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_{X_{\mathrm{ED}}}$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_{X_{\mathrm{ED}}}$ are equivalent.*

The AETFS Problem The length of sensitive patterns in the **ETFS** setting is fixed. In what follows, we define a generalization of the **ETFS** problem which allows for arbitrary length sensitive patterns. Let $\mathcal{S}$ be a set of intervals with the two following properties (closure property and minimality property): (i) For every $[i,j] \in \mathcal{S}$, any $[i',j']$ for which $W[i'\,..\,j'] = W[i\,..\,j]$, must also belong to $\mathcal{S}$; and (ii) any proper sub-interval of $[i,j]$ is not in $\mathcal{S}$. It is easy to see that $|\mathcal{S}| \leq n$ from its minimality. Now, we redefine notions of sensitive and non-sensitive patterns as follows: A sensitive pattern is an *arbitrary length* substring $W[i\,..\,j]$ of W for each $[i,j] \in \mathcal{S}$. For a fixed k, a non-sensitive pattern is a length-k substring of W containing no sensitive pattern as a substring.

Problem 2 (**AETFS**). *Given a string W of length n, an integer $k > 1$, and a set $\mathcal{S}$ (and thus set $\mathcal{I}$), construct a string Y_{ED} which is at minimal (weighted) edit distance from W and satisfies:*

C1 *Y_{ED} does not contain any sensitive pattern.*
P1 *$\mathcal{I}_W \equiv \mathcal{I}_{Y_{\mathrm{ED}}}$, i.e., the t-chains $\mathcal{I}_W$ and $\mathcal{I}_{Y_{\mathrm{ED}}}$ are equivalent.*

The ETFS-DP Algorithm. For independent reading we describe here ETFS-DP, the algorithm from [9] that solves the **ETFS** problem in $\mathcal{O}(n^2k)$ time. The output string X_{ED} is a string that contains all non-sensitive patterns in the same order as in W. For each pair of consecutive non-sensitive patterns, their occurrences in X_{ED} are either (i) overlapping by $k-1$ letters (e.g., `eca` and `cab` in Example 1) or (ii) delimited by a string over $\Sigma \cup \{\#\}$ which contains no length-k string over Σ (e.g., `#aa#` in Example 1). We call such strings *gadgets*. For case (ii), we use the following regular expressions:

$$\Sigma^{<k} = (a_1|a_2|\ldots|a_{|\Sigma|}|\varepsilon)^{k-1},$$

where $\Sigma = \{a_1, a_2, \ldots, a_{|\Sigma|}\}$. Also, the special letters $\ominus, \oplus, \otimes \in \Gamma$ correspond to regular expressions $(\Sigma^{<k}\#)^*$, $\#(\Sigma^{<k}\#)^*$, and $(\#\Sigma^{<k})^*$, respectively. Let $N_0, N_1, \ldots, N_{|\mathcal{I}|-1}$ be the sequence of non-sensitive patterns sorted in the order in which they occur in W. In what follows, we fix string $T = \ominus N_0 \oplus N_1 \oplus \cdots \oplus N_{|\mathcal{I}|-1} \otimes$ of length $(k+1)|\mathcal{I}| + 1$. String T corresponds to the regular expression R that represents the set of all feasible solutions (feasible strings) in which all non-sensitive patterns in the string are delimited by stings over $\Sigma \cup \{\#\}$. Moreover, we need to consider feasible strings in which a non-sensitive pattern overlaps the next one. Let M be a binary array of length $|\mathcal{I}|$ such that for each $0 \leq i \leq |\mathcal{I}| - 1$, $M[i] = 1$ if $i > 0$ and N_{i-1} has a suffix-prefix overlap of length $k-1$ with N_i, and $M[i] = 0$ otherwise. Namely, $M[i] = 1$ implies that N_{i-1} and N_i can be merged for $0 < i \leq |\mathcal{I}| - 1$.

Let E be a table of size $((k+1)|\mathcal{I}|+1) \times (n+1)$. The rows of E correspond to string T defined above and the columns to string W. Note that the leftmost column corresponds to the empty string ε as in the standard edit distance DP table. Each cell $E[i][j]$ contains the edit distance between the regular expression corresponding to $T[0\,..\,i]$ and $W[0\,..\,j-1]$. We classify the rows of E into three categories: *gadget rows*; *possibly mergeable rows*; and *ordinary rows*. We call every row corresponding to a special letter in Γ a gadget row. Namely, rows with index $i \equiv 0 \bmod (k+1)$ are gadget rows. Also, we call every row corresponding to the last letter of a non-sensitive pattern a possibly mergeable row. Namely, rows with index $i \equiv -1 \bmod (k+1)$ are possibly mergeable rows. All the other rows are called ordinary rows. The recursive formula of ordinary rows is the same as in the standard edit distance solution:

$$E[i][j] = \min \begin{cases} E[i-1][j] + 1, & \text{(insert)} \\ E[i][j-1] + 1, & \text{(delete)} \\ E[i-1][j-1] + I[T[i] \neq W[j-1]], & \text{(match or substitute)}, \end{cases}$$

where I is an indicator function: $I[T[i] \neq W[j-1]] = 1$ if $T[i] \neq W[j-1]$, and 0 otherwise. Next, consider a possibly mergeable row $E[i][\cdot]$ which is the last row of the non-sensitive pattern N_h. If $M[h] = 0$, then the recursive formula is the same as that of ordinary rows. Otherwise ($M[h] = 1$), N_{h-1} and N_h can be merged. Merging them means that the values in the previous mergeable row $E[i-k-1][\cdot]$ will be propagated to $E[i][\cdot]$ directly without considering the k rows below. Thus, the recursive formula is:

$$E[i][j] = \min \begin{cases} E[i-1][j] + 1, & \text{(insert)} \\ E[i][j-1] + 1, & \text{(delete)} \\ E[i-1][j-1] + I[T[i] \neq W[j-1]], & \text{(match or substitute)} \\ E[i-k-1][j] + 1, & \text{if } M[h] = 1 \text{ (insert and merge)} \\ E[i-k-1][j-1] + I[T[i] \neq W[j-1]], & \text{if } M[h] = 1 \text{ (match or sub. and merge)}. \end{cases}$$

Next, consider a gadget row $E[i][\cdot]$ which corresponds to a special letter in Γ. Because of the form of regular expressions corresponding to special letters, a # can either be inserted or substituted directly after a non-sensitive pattern, or be preceded by another # no more than k positions earlier. This results in the following recursive formula:

$$E[i][j] = \min \begin{cases} E[i-1][j] + 1, & \text{(insert)} \\ E[i-1][j-1] + 1, & \text{(substitute)} \\ E[i][j-1] + 1, \ldots, E[i][\max\{0, j-k\}] + 1, & \text{(delete or extend gadget)}. \end{cases}$$

For completeness, we write down the recursive formula for initializing the leftmost column:

$$E[i][0] = \begin{cases} E[i-k-1][0]+1, & \text{if } i \equiv -1 \bmod (k+1) \wedge M[h]=1 \text{ (merge)} \\ E[i-1][0]+1, & \text{otherwise (no merge).} \end{cases}$$

Unlike in the standard setting [29], the edit distance between W and any string matching the regular expression R is not necessarily found in its bottom-right entry $E[|\mathcal{I}|(k+1)][|W|]$. Instead, it is found among the rightmost k entries of the last row (in case X_{ED} ends with a string in $\otimes$), and the rightmost entry of the second-last row (when X_{ED} ends with the last letter of the last non-sensitive pattern). After computing the edit distance value, we construct an X_{ED}. To do so, when computing each entry $E[i][j]$, we memorize a backward-pointer to an entry from which the minimum value for $E[i][j]$ was obtained. We then construct X_{ED} from right to left with respect to the sequence of edit operations corresponding to an optimal alignment obtained by the backward-pointers.

3 Compact Lookup Table Structure for Squared Blocks

In this section we consider the standard edit distance table, and propose a data structure which can answer some queries on a $b \times b$ sub-table of the DP table, which we call a *block*, corresponding to two strings of the same length. Our data structure is similar to the one proposed in [12], tailored, however, to our needs. We next provide some further definitions about blocks. Let B be a $b \times b$ block to be processed. The top (resp. bottom) row of B is called the input (resp. output) row of B. Similarly, the leftmost (resp. rightmost) column of B is called the input (resp. output) column of B. A cell in the input (resp. output) row or column is called an input (resp. output) cell.

In the following, we propose a lookup table for $b \times b$ blocks that compute all output cell values of a block in $\mathcal{O}(b)$ time for any given block and input cell values of the block. We modify the following known result to enhance it with a trace-back functionality.

Theorem 3 (Theorem 1 in [12])**.** *Given two strings both of length b corresponding to a $b \times b$ block, we can construct a data structure of size $\mathcal{O}(b^2)$ in $\mathcal{O}(b^2 \log b)$ time such that given any values for the input row and column of the block, the data structure can compute the output row and column of the block in $\mathcal{O}(b)$ time.*

In [12], the authors did not refer to tracing back, i.e., it is not clear how to obtain an optimal alignment using Theorem 3. We prove that we can trace back a shortest path to an output cell in a $b \times b$ block in $\mathcal{O}(b)$ time using $\mathcal{O}(b^2 \log b)$ additional space. This yields an optimal alignment. We next briefly describe the data structure of Theorem 3, and explain how we modify it.

3.1 Constructing a Data Structure for a Pair of Strings

For constructing the data structure of Theorem 3, Brubach and Ghurye [12] utilize the result of [35]. Instead, we use the following result by Klein [25], which is more general.

Theorem 4 ([25])**.** *Given an N-node planar graph with non-negative edge labels, we can construct a data structure of size $\mathcal{O}(N \log N)$ in $\mathcal{O}(N \log N)$ time such that given a node s in the graph and another node t on the boundary of the infinite face, the data structure can compute the maximum (or minimum) distance from s to t in $\mathcal{O}(\log N)$ time. Also, if the graph has constant degree, then we can compute the shortest s-t path in time linear in the length of the path.*

The lookup table structure for a block is constructed as follows (see [12] for details). Let B be a $b \times b$ block to be preprocessed. First, we regard B as a grid-graph of size $b \times b$. Namely, each node corresponds to a cell in the block, and each edge corresponds to an edit operation. Also, each edge is labeled by the weight of its corresponding edit operation. Then, we construct the data structure of Theorem 4 for the grid-graph. We denote this data structure by $\mathcal{D}_B$. Next, for each input cell u and each output cell v, we compute the weight of the shortest path from u to v, and store them to table M_B of size $(2b-1) \times (2b-1)$. Each row (resp. column) of M_B corresponds to each output (resp. input) cell of B. A table is called *monotone* if each row's minimum value occurs in a column which is equal to or greater than the column of the previous row's minimum. It is *totally monotone* if the same property is true for every sub-table defined by an arbitrary subset of the rows and columns of the given table. It is known that we can construct M_B so that it is totally monotone [15]. We thus construct $\mathcal{D}_B$ and M_B in $\mathcal{O}(b^2 \log b)$ time and space.

By Theorem 3, the size of the final data structure (that depends on the size of M_B) is $\mathcal{O}(b^2)$. However, $\mathcal{O}(b^2 \log b)$ working space is used for constructing $\mathcal{D}_B$. In our algorithm, we also use table M_B and keep the temporary data structure $\mathcal{D}_B$ to support tracing back operations efficiently.

3.2 Answering Queries and Tracing Back

Given a query input row and column, we can compute the output row and column in $\mathcal{O}(b)$ time using the SMAWK algorithm [3] for finding the minimum value in each row of an implicitly-defined totally monotone table, since M_B is totally monotone [12]. Note that, for each output cell v of B, we can also obtain an input cell s_v which is the starting cell of a shortest path ending at v from the result of SMAWK algorithm. Thus, by using data structure $\mathcal{D}_B$, we can obtain a shortest s_v-v path in time linear in the length of the path. To summarize, we obtain the following lemma.

Lemma 1. *Given two strings both of length b corresponding to a $b \times b$ block, we can construct a data structure of size $\mathcal{O}(b^2 \log b)$ in $\mathcal{O}(b^2 \log b)$ time such that given any values for the input row and column of the block, the data structure can compute the output row and column of the block in $\mathcal{O}(b)$ time. Furthermore, given an output cell v and any other cell u in the block, we can compute a shortest u-v path in time linear in the length of the path.*

Note that this data structure works under edit distance with arbitrary weights at no extra cost.

4 Sensitive Patterns of Arbitrary Lengths

In this section we propose a data structure with which we can solve the **AETFS** problem in time $\mathcal{O}(n^3)$. First, let us consider whether ETFS-DP can be applied directly to the **AETFS** problem. The **AETFS** problem is a generalization of the **ETFS** problem, and there are some differences between them: if there exists a *long sensitive pattern* of length longer than k, then we cannot apply the same logic for the possibly mergeable rows to the **AETFS** problem. This is because merging multiple non-sensitive patterns of length k may create a long sensitive pattern, while this sensitive pattern must be hidden. In contrast, if there exists a *short sensitive pattern* of length less than k, then we cannot apply the same logic for the gadget rows to the **AETFS** problem, since this may introduce a short sensitive pattern in a gadget. Thus **AETFS** is much more challenging.

Let $L = \mathcal{O}(n^2)$ denote the total length of long sensitive patterns. As a first step towards our main result, we prove the following lemma.

Lemma 2. *The **AETFS** problem can be solved in $\mathcal{O}(k|\mathcal{I}|n + Ln)$ time.*

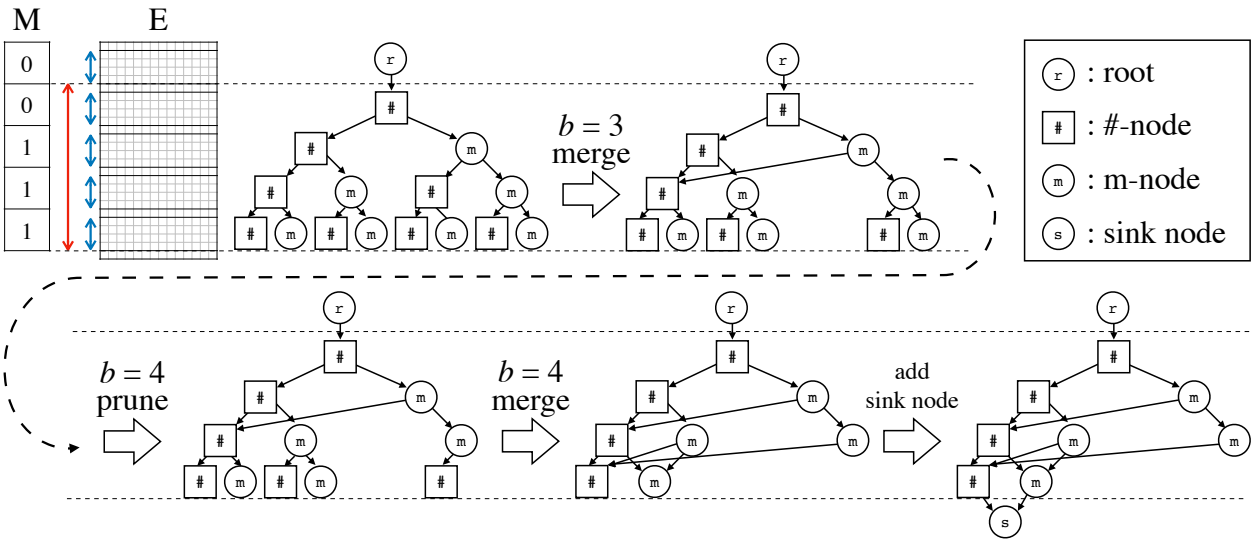

Figure 1: An example for pruning and merging a decision tree. The red arrow represents a long sensitive pattern, and blue arrows represent non-sensitive patterns. The tree on the top-left represents the decision tree after pruning or merging operations for depths $b = 0, 1, 2$. For $b = 3$, we merge two #-nodes by Rule 3. For $b = 4$, we prune an m-node by Rule 2, merge three #-nodes by Rule 3, and merge two m-nodes by Rule 4. Finally, we add the sink node at the bottom.

Note that Lemma 2 yields $\mathcal{O}(n^2k)$ time for **ETFS** because in this case $L = 0$. Lemma 2 thus generalizes Theorem 2 in [9]. In what follows, we propose a new data structure for solving the **AETFS** problem and prove Lemma 2. The main idea is to use multiple DP tables and link them under specific rules. Interestingly, our data structure is shaped as a DAG consisting of DP tables.

4.1 Long Sensitive Patterns

If there is a long sensitive pattern, we need to consider the case where multiple non-sensitive patterns are contained in a single sensitive pattern. (Recall that all non-sensitive patterns have fixed length k.) In this case, we cannot apply the ETFS-DP algorithm from [9] directly.

Let us consider the situation in which we have just finished computing a possibly mergeable row. We may be able to choose the next move from two candidates: either go down to the next (gadget) row or jump to the next possibly mergeable row if possible. We consider a *decision tree* $\mathcal{T}$ that represents all combinations of such choices at all possibly mergeable rows (inspect Figure 1). We regard $\mathcal{T}$ as a *tree of tables*, i.e., each node of $\mathcal{T}$ represents a small DP table. Let $E[0\,..\,(k+1)|\mathcal{I}|][0\,..\,n]$ be the DP table of the ETFS-DP algorithm described in Section 2. There are three types of nodes in $\mathcal{T}$: *root*, *#-node*, and *m-node*. The root represents sub-table $E[0\,..\,k][0\,..\,n]$. For each depth b with $1 \leq b \leq |\mathcal{I}| - 1$, the #-node at depth b represents sub-table $E[b(k+1)\,..\,(b+1)(k+1)-1][0\,..\,n]$, and each m-node at depth b represents a copy of possibly mergeable row $E[(b+1)(k+1)-1][0\,..\,n]$. Each edge (u, v) of $\mathcal{T}$ means that the bottom row values of u will be propagated to the top row of v. If there are multiple incoming edges $(u_1, v), (u_2, v), \ldots, (u_p, v)$ of a single node v, then we virtually consider a row $r[0\,..\,n]$ as the previous possibly mergeable row of v such that $r[j]$ is the minimum value between all j-th values in the last rows of $u_1, u_2, \ldots, u_p$ for each $0 \leq j \leq n$. We call a path that consists of only m-nodes an *m-path*.

We can solve the **AETFS** problem if we can simulate all valid combinations of merge operations represented by $\mathcal{T}$. However, we do not have to check all combinations explicitly. This is due to the

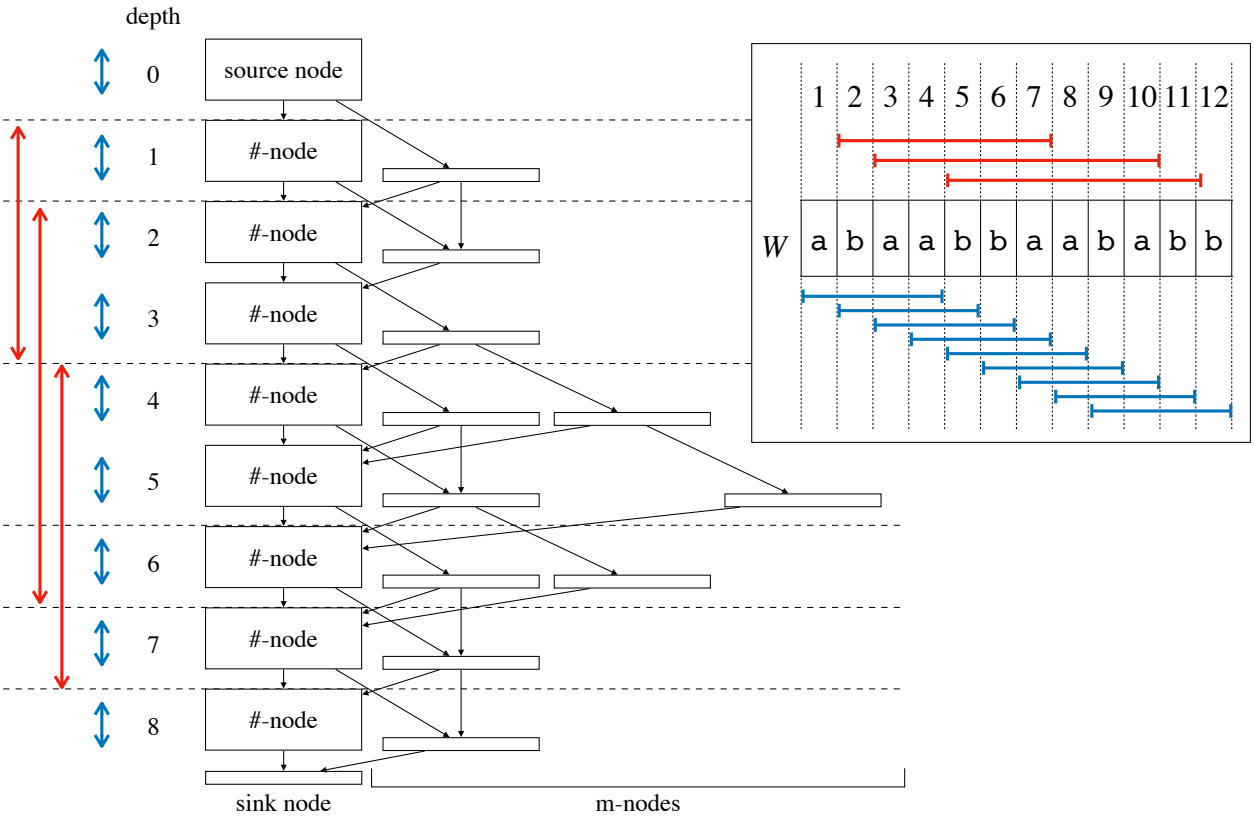

Figure 2: The decision DAG for $W = \texttt{abaabbaababb}$, $k = 4$, and $\mathcal{S} = \{[2, 7], [3, 10], [5, 11]\}$.

fact that we can *prune* branches and *merge* nodes in the decision tree $\mathcal{T}$ as follows[6].

For incremental $b = 1, 2, \ldots, |\mathcal{I}| - 1$, we edit $\mathcal{T}$ according to the following four rules:

Rule 1. If $M[b] = 0$, then prune all edges to m-nodes at depth b.

Rule 2. If there is a path $(v_1, v_2, \ldots, v_p)$ such that the depth of v_p is b, $(v_2, \ldots, v_p)$ is an m-path, and v_1 and v_p respectively corresponds to the length-k prefix and the length-k suffix of the same sensitive pattern, then prune the edge (v_{p-1}, v_p).

Rule 3. If there are multiple #-nodes at depth b, then merge all of them into a single #-node.

Rule 4. If there are multiple m-nodes $\{v_1, v_2, \ldots\}$ at depth b such that each u_i does not correspond to the length-k prefix of any sensitive pattern, where u_i is the parent of the starting node of the longest m-path ending at v_i, then merge such m-nodes into a single m-node.

Finally, we add the *sink node* under the decision tree such that the sink node corresponds to the bottom row $E[|\mathcal{I}|(k+1)][0\,..\,n]$, and each node at depth $|\mathcal{I}| - 1$ has only one outgoing edge to the sink node. We also rename the root to the *source node*.

After executing all pruning and merging operations, the decision tree becomes a DAG whose all source-to-sink paths represent all valid choices (inspect Figure 2).

[6] Once the merge operation is applied to the decision tree, it is no longer a tree. However, we continue calling it the decision "tree" for convenience.

We call such DAG the *decision DAG*, and we denote it by $\mathcal{G}$. Although the size of $\mathcal{T}$ can be exponentially large, we can directly construct $\mathcal{G}$ in a top-down fashion from an instance of **AETFS** in $\mathcal{O}(|\mathcal{G}|)$ time.

Correctness We show that no valid path is eliminated and all invalid paths are eliminated while constructing $\mathcal{G}$ from $\mathcal{T}$. Clearly, crossing #-nodes creates no invalid path. In what follows, we mainly focus on m-nodes that can create invalid paths.

It is easy to see that a path $(v_1, v_2, \ldots, v_p)$ is invalid if and only if (i) there is a sub-path $(v_s, \ldots, v_t)$ where v_s and v_t respectively correspond to the length-k prefix and suffix of the same sensitive pattern, and $v_{s+1}, \cdots, v_t$ are all m-nodes or (ii) there is an edge (v_{i-1}, v_i) such that $M[d_i] = 0$ where d_i is the depth of v_i. By Rule 1, we delete all invalid paths which satisfy condition (ii), and do not delete any valid path. By Rule 2, we delete an invalid path which satisfies condition (i), and do not delete any valid path. By Rule 3, we merge #-nodes, however, it does not matter since this operation does not cause deleting or creating any path. By Rule 4, we may merge m-nodes, however, the m-nodes to be merged are carefully chosen to not interfere with Rule 2. Thus, this also does not cause deleting or creating any path. Therefore, $\mathcal{G}$ is constructed correctly.

The Size of the DAG We next analyze the size of $\mathcal{G}$. Clearly, the total number of #-nodes is equal to $|\mathcal{I}| - 1$. Also, the source node and the sink node are unique. The number of m-nodes, each of which is a child of some #-node, is equal to the number of #-nodes, i.e., $|\mathcal{I}| - 1$. The number of the rest of m-nodes is at most L. Also, each node has at most two outgoing edges.

Each #-node and the source node represent a sub-table of size $(k+1) \times (n+1)$. Each m-node represents a possibly mergeable row, and the sink node represents the last gadget row. Therefore, the total size of $\mathcal{G}$ is $\mathcal{O}(|\mathcal{I}|kn + |\mathcal{I}|n + Ln) = \mathcal{O}(k|\mathcal{I}|n + Ln)$.

Time Complexity The decision DAG $\mathcal{G}$ is computed in $\mathcal{O}(k|\mathcal{I}| + L)$ time without creating the original decision tree $\mathcal{T}$ by applying the above four rules for incremental $b = 1, 2, \ldots, |\mathcal{I}| - 1$. Also, we can compute each cell in $\mathcal{G}$ in amortized constant time [9]. Thus, the total time is $\mathcal{O}(k|\mathcal{I}|n + Ln)$.

4.2 Short Sensitive Patterns

Running the ETFS-DP algorithm may introduce short sensitive patterns in its gadgets. We explain how to modify the recursive formulae of the gadget row to account for short sensitive patterns. We first prove that w.l.o.g. all gadgets are either a single # or can be optimally aligned such that:

1. All #'s in gadgets are substituted by letters in W;
2. All letters in gadgets are matched with letters in W; and
3. No further letters are inserted between letters of the same gadget.

If some extra inserted letters of W are aligned with a gadget, we can add some extra #'s to change them into substitutions without increasing the cost, changing the number of non-sensitive patterns or increasing the number of sensitive patterns. Similarly, if some letters of the gadget are not matched with the same letters in W, these gadget letters can be replaced by #. Finally, if some #'s in the gadget are not aligned with any letter in W, we can either remove them or move them to the place of an adjacent gadget letter while deleting that letter. Inspect the following example.

Example 3. *Let the following optimal alignment from Example 1 with cost 4. Gadgets are in red.*

```
e c a b - a a a a a b b - b a d f
e c a b # a a # a b b b # b a d f
```

We transform it to another optimal alignment of the same cost that respects the above conditions:

```
e c a b a a a a a b b - b a d f
e c a b # a # a b b b # b a d f
```

Let us first consider the leftmost gadget: (1) #'s are substituted by letters in W; (2) all letters are matched with letters in W; and (3) no further letters are inserted between the gadget's letters. Note that the rightmost gadget is a single # and so the modified alignment satisfies all conditions above.

A single # cannot introduce any sensitive pattern, so just as in the ETFS-DP algorithm we can get a cost of $E[i-1][j]+1$ corresponding to the case that a single # is inserted after $W[j-1]$ or a cost of $E[i-1][j-1]+1$ corresponding to the case that a single # is aligned with $W[j-1]$. For longer gadgets the possibilities are a bit more restricted than in the ETFS-DP algorithm. Assuming the gadget to have the structure described above, it follows that the previous # cannot be aligned before $W[F[j-1]]$, where $F[j]$ is defined to be the largest integer such that $W[F[j]\,..\,j-1]$ contains a sensitive or non-sensitive pattern (if it exists). More formally:

$$F[j] = \max(\{i < j \mid W[i\,..\,j-1] \text{ contains a sensitive pattern}\} \cup \{j-k\} \cup \{0\}).$$

F can be computed in $\mathcal{O}(kn)$ time. We denote the point-wise minimum of the copies of the preceding merge row with r; in the case of **ETFS** this is just the previous merge row. This gives us the following formula for the gadget rows. For all $0 \le i \le (k+1)|\mathcal{I}|$ with $i \equiv 0 \mod k+1$,

$$E[i][j] \;=\; \min \begin{cases} r[j]+1 \\ r[j-1]+1 \\ E[i][j-1]+1, E[i][j-2]+1, \ldots, E[i][F[j-1]+1]+1. \end{cases}$$

(Notice that a string position and its corresponding table index differ by one.)

To conclude, we also need to consider the range in which the edit distance value lies. Since the last row corresponds to $\otimes$, the value stored in $E[|\mathcal{I}|(k+1)][j]$, for all $0 \le j \le n$, is the cost of an optimal alignment between $W[0\,..\,j+e_j-1]$ and a string in the regular expression whose length-(e_j+1) suffix is $\#W[j\,..\,j+e_j-1]$, where $e_j = \min(\max\{e \mid W[j\,..\,j+e-1] \text{ does not contain any sensitive or non-sensitive pattern}\} \cup \{n-j\})$. The edit distance between W and any string matching the regular expression is found among the rightmost $n-F[n]$ entries of the last row or the rightmost entry of the second-last row. Thus, we obtain:

$$d_{\mathrm{E}}(Y_{\mathrm{ED}}, W) \;=\; \min \begin{cases} E[|\mathcal{I}|(k+1)-1][n], \\ E[|\mathcal{I}|(k+1)][n], E[|\mathcal{I}|(k+1)][n-1], \ldots, E[|\mathcal{I}|(k+1)][F[n]+1]. \end{cases}$$

For each $E[i][j]$ and $r[j]$ we store a pointer to an entry which leaded to this minimum value. We can then trace back as in ETFS-DP, taking the minimizing entry of the above equation as a starting point, and obtain Y_{ED} in an additional $\mathcal{O}(kn)$ time. Therefore the total time complexity of **AETFS** is $\mathcal{O}(k|\mathcal{I}|n + Ln)$ and we arrive at Lemma 2.

5 $\tilde{\mathcal{O}}(n^2)$-Time Algorithms using Dyadic Intervals

In this section we improve ETFS-DP and the algorithm of Lemma 2. We first show an algorithm to compute *gadget rows* in amortized constant time per cell in the rows. Secondly, we focus on the

redundancy in the computation of *ordinary rows*, and propose an algorithm to compute them by using the lookup table structure of Section 3 according to dyadic intervals. These two improvements yield an $\tilde{\mathcal{O}}(n^2)$-time algorithm for **ETFS**. Finally, we employ a similar lookup table technique to contract m-paths in the decision DAG, which yields an $\tilde{\mathcal{O}}(n^2)$-time algorithm for **AETFS** as well.

5.1 Speeding Up Gadget Rows Computation

First, we show how to speedup the gadget rows computation. For each #-node u in the decision DAG $\mathcal{G}$, we denote by d_u the in-degree of u. Let $G_u[0\,..\,n]$ be the gadget row in u. For each $0 \le i \le d_u - 1$, let $M_u^i[0\,..\,n]$ be a possibly mergeable row of a node which has an edge pointing to u. The recursive formula for $G_u[i]$ is as follows: $G_u[0] = \min\{M_u^0[0]+1, \ldots, M_u^{d-1}[0]+1\}$, and

$$G_u[j] = \min \begin{cases} M_u^0[j-1]+1, \ldots, M_u^{d-1}[j-1]+1, \\ M_u^0[j]+1, \ldots, M_u^{d-1}[j]+1, \\ G_u[j-1]+1, \ldots, G_u[F[j-1]+1]+1, \end{cases}$$

for $1 \le j \le n$. We assume that $M_u^0, \ldots, M_u^{d-1}$ and F are given. It costs $\mathcal{O}(n(k+d_u))$ time to compute G_u naïvely. The next lemma states that we can actually compute G_u in $\mathcal{O}(nd_u)$ time.

Lemma 3. *Given $M_u^0, \ldots, M_u^{d-1}$ and F, we can compute every $G_u[i]$ in $\mathcal{O}(d_u)$ time for incremental $i = 0, \ldots, n$.*

Proof. Let us fix an arbitrary #-node u and omit subscripts related to u. Let r_j be the index such that $G[r_j]$ is the rightmost minimum value in the range $G[F[j-1]+1\,..\,j]$, and let $m_j = G[r_j]$ be that minimum value. Then, it can be seen that $G[p] = m_j + 1$, for any $r_j < p \le j$, since $G[p] > G[r_j]$ and $G[p] \le G[r_j]+1$ by the recursive formula. Clearly, $r_0 = 0$. We assume that r_{j-1} is known before computing $G[j]$. If $r_{j-1} < F[j-1]+1$, then $G[j-1] = m_{j-1}+1$ is the minimum in $G[F[j-1]+1\,..\,j-1]$, and $r_j = \arg\min\{G[j-1], G[j]\}$. Otherwise, $F[r_{j-1}] = m_j$ is the minimum in $G[F[j-1]+1\,..\,j-1]$, and $r_j = \arg\min\{G[r_{j-1}], G[j]\}$. Note that F is a non-decreasing array, i.e., $F[j] \ge F[j-1]$. Thus, we can compute $G[j]$ and r_j in $\mathcal{O}(d)$ time. □

By Lemma 3, we can compute all gadget rows in a total of $\mathcal{O}(n\sum_{u\in\mathcal{G}} d_u) = \mathcal{O}(n|\mathcal{I}| + nL)$ time.

5.2 ETFS in $\mathcal{O}(n^2 \log^2 k)$ Time

In this section we describe an algorithm which solves **ETFS** in $\mathcal{O}(n^2 \log^2 k)$ time. The key to losing the factor k is the fact that the string T on the left is highly repetitive and only consists of substrings of the length-n string W (interleaved by some letters in Γ). Therefore we can compute the DP table efficiently using only few precomputed sub-tables as in the Four Russians method [6].

First, we partition W into substrings of length 2^i (or shorter if $2^i \nmid |W|$) for each $i \in \{0, 1, 2, \ldots, \lfloor \log k \rfloor\}$. This gives a set $\mathcal{A}$ of at most $2n$ different strings. Moreover, note that each length-k pattern in W can be written as the concatenation of at most $2\lfloor \log k \rfloor + 2$ such strings.

For every pair of strings in $(w_1, w_2) \in \mathcal{A}^2$ with $|w_1| = |w_2|$, we precompute the lookup table for the strings w_1 and w_2 according to Lemma 1. Now we can compute the non-merge case of each possibly mergeable row using at most $2 \cdot \lceil n/2^i \rceil$ precomputed lookup tables of size $2^i \times 2^i$ (or smaller) for each $i \in \{0, 1, 2, \ldots, \lfloor \log k \rfloor\}$.

Time Complexity Precomputing a lookup table for two strings of length up to 2^i takes $\mathcal{O}(2^{2i}i)$ time. In total this gives a precomputation time of

$$\mathcal{O}\left(\sum_{i=0}^{\lfloor \log k \rfloor} \frac{n}{2^i} \cdot \frac{n}{2^i} \cdot 2^{2i} i\right) = \mathcal{O}(n^2 \log^2 k).$$

Each possibly mergeable row can now be computed in $\mathcal{O}(n \log k)$ time from the previous merge and gadget row, since each non-sensitive length-k pattern can be partitioned into at most $2\lfloor \log k \rfloor + 2$ precomputed strings. Gadget rows can be computed in $\mathcal{O}(n)$ time each from the preceding possibly mergeable rows using the technique described by Lemma 3.

For the traceback, note that $|X_{\mathrm{ED}}| = \mathcal{O}(kn)$, i.e., the length of an optimal alignment path over E is $\mathcal{O}(kn)$. We do not know how the path behaves inside each block. However we can compute the sub-path inside a block in time linear in the path's length by using Lemma 1. The gadget rows can be traced back in a further $\mathcal{O}(n)$ time. Thus, we can trace back in a total time of $\mathcal{O}(kn)$. Therefore the total time complexity is $\mathcal{O}(n^2 \log^2 k)$. We arrive at the following result.

Theorem 1. *The **ETFS** problem can be solved in $\mathcal{O}(n^2 \log^2 k)$ time.*

5.3 AETFS in $\mathcal{O}(n^2 \log^2 n)$ Time

In this section we describe how to further reduce the decision DAG $\mathcal{G}$ from Section 4 by precomputing parallel m-paths. First, we give some observations for m-paths. An m-path is said to be *maximal* if the m-path cannot be extended either forward or backward. Any two maximal m-paths do not share any nodes, since every m-node in $\mathcal{G}$ has at most one outgoing edge to m-nodes and at most one incoming edge from m-nodes. Also, the number of maximal m-paths is at most $|\mathcal{I}|$ since the parent of the first m-node of each maximal m-path is a different #-node or the source node. In what follows, suppose $\mathcal{G}$ contains a total of p maximal m-paths of length $\ell_1, \ldots, \ell_p$ with $\sum_{i=1}^{p} \ell_i \leq n + \ell n$, where ℓ is the length of the longest sensitive pattern. Recall that an m-node represents a possibly mergeable row of size $1 \times (n+1)$, and thus, we will identify an m-path of length x with a DP table of size $x \times (n+1)$.

Let us now describe our DAG reduction. An example of the DAG reduction is demonstrated in Figure 3. In the prepocessing phase, we first construct a lookup table structure for all possible m-paths corresponding to dyadic intervals of lengths at most ℓ over the range $[1, |\mathcal{I}| - 1]$ of the depths of m-paths, in a similar way as in Section 5.2. Next, for each maximal m-path in $\mathcal{G}$, we decompose it into shorter m-paths according to dyadic intervals. We then contract each such m-path into a single node named *j-node* consisting of a single row, which *jumps* from the beginning of the m-path to the end and represent consecutive merges. Also, we have to take into account edges leaving the m-path. Note that paths that leave the m-path early always leave to a #-node, so we do not have to worry about introducing any sensitive patterns. We therefore create a new *copy* of the m-path preceded by an additional node named *c-node* consisting of a single row, which takes the point-wise minimum of the parent nodes of the m-paths.

After finishing the DAG reduction, we fill the DP tables from top to bottom: all j-nodes are computed by using the lookup table; all new m-paths are computed in the original fashion, including all outgoing edges; and all the other nodes are computed as in Section 5.2. Also, we can trace back and find the solution to **AETFS** by storing appropriate backward-pointers and the data structures of Lemma 1 just as in Section 5.2.

Time Complexity Constructing the lookup tables takes $\mathcal{O}(n^2 \log^2 \ell)$ time since there are at most $\lfloor \log \ell \rfloor$ different path lengths, and for each $i \in \{0, 1, \ldots, \lfloor \log \ell \rfloor\}$, we preprocess at most

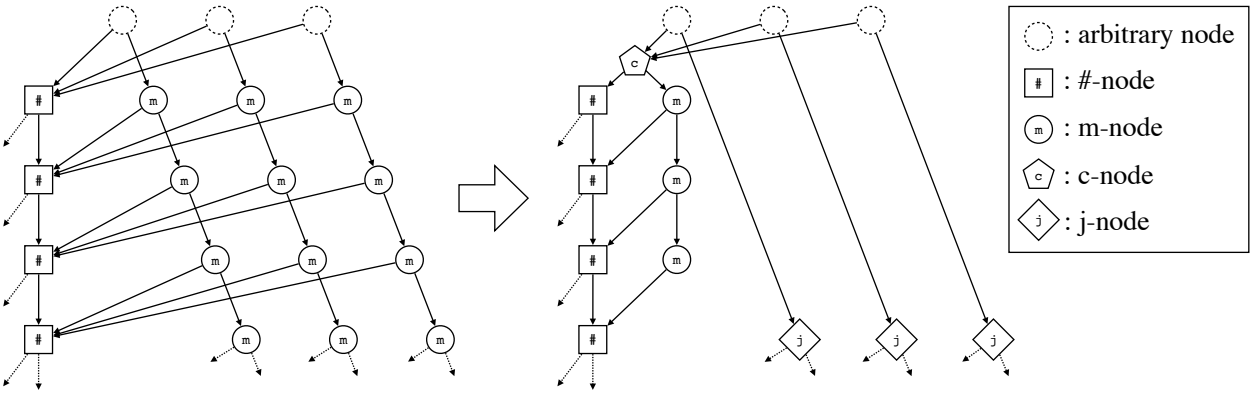

Figure 3: An example of contracting a part of the decision DAG. Before contracting, there are three parallel m-paths each of length $2^2 = 4$. We contract each m-path to a j-node corresponding to the case in which four consecutive non-sensitive patterns are merged. Also, we create an m-path of length 3 preceded by a c-node corresponding to the case in which at least one gadget is inserted.

$(n/2^i)^2$ blocks of size $2^i \times 2^i$ each in $\mathcal{O}(2^{2i}i)$ time. We can also easily contract $\mathcal{G}$ in $\mathcal{O}(n^2)$ time by traversing the DAG. Note that the number of nodes in the original DAG is $\mathcal{O}(n^2)$ (Section 4.1). We partition each path of length ℓ_i into at most $2(\log \ell + \ell_i/\ell)$ precomputed paths: at most $\ell_i/2^{\lfloor \log \ell \rfloor}$ paths of length $2^{\lfloor \log \ell \rfloor}$ and at most 2 of each shorter length. Therefore the j-nodes can be computed in $\mathcal{O}(n \cdot \sum_{i=1}^{p} 2(\log \ell + \ell_i/\ell)) = \mathcal{O}(n^2 \log \ell)$ time. The c-nodes and the following m-nodes can be computed in $\mathcal{O}(n^2 \log \ell)$ time, because there is at most one c- or m-node per depth and per precomputed path length. The #-nodes can each be computed in $\mathcal{O}(n \log^2 k)$ time using the method described in Section 5.2. Finally, tracing back takes only $\mathcal{O}(kn)$ time by using the backward-pointers and the data structures of Lemma 1.

Summarizing this section, we have shown that the **AETFS** problem can be solved in time $\mathcal{O}(n^2 \log^2 k + \min\{n^2 \log^2 \ell, Ln\})$. We arrive at the following result.

Theorem 2. *The **AETFS** problem can be solved in $\mathcal{O}(n^2 \log^2 n)$ time.*

6 Applying Four Russians and LZ78-Factorization

In this section we show how to apply the Four Russians [6] and the LZ78-factorization [39] techniques, which speed up the standard edit distance computation [31, 15], to the **ETFS** and **AETFS** problems. The recursive formula of the ordinary rows is the same as that of edit distance. Thus, it is easy to see that we can partially apply the Four Russians technique to our problems. On the contrary, applying LZ78-factorization to our problems is nontrivial, since LZ78-factorization does not take into account gadget rows or possibly mergeable rows. We define a new factorization, which is an extension of LZ78, and show that we can modify the technique of [15] to **ETFS** and **AETFS**.

6.1 Applying the Four-Russians Method

We apply the Four Russians method to the **ETFS** problem. Let $b \leq k$ be the length of one side of a squared block. First, we create a lookup table for all possible σ^{2b} pairs of length-b strings over Σ. Each lookup entry is a data structure of Lemma 1 whose size is $\mathcal{O}(b^2 \log b)$, and it can be constructed in $\mathcal{O}(b^2 \log b)$ time. The total time to create the lookup table is $\mathcal{O}(\sigma^{2b} b^2 \log b)$. Next, we decompose the DP table into $b \times b$-sized blocks except for possibly mergeable rows and

gadget rows. Since the number of blocks is $\mathcal{O}(k|\mathcal{I}|n/b^2)$ and the computation time for propagation is $\mathcal{O}(b)$ per block, the total time for computing all boundaries of all blocks is $\mathcal{O}(k|\mathcal{I}|n/b)$. Unlike the original Four Russians method [31], we compute all possibly mergeable rows and gadget rows consisting of $\mathcal{O}(|\mathcal{I}|n)$ cells. We can compute these rows in a total of $\mathcal{O}(|\mathcal{I}|n)$ time by Lemma 3.

Theorem 5. *The **ETFS** problem can be solved in $\mathcal{O}(n^2 + \frac{n^2 k}{\log_\sigma n})$ time.*

Proof. From the above discussion, we can solve the **ETFS** problem in $\mathcal{O}(kn|\mathcal{I}|/b+\sigma^{2b}b^2 \log b+|\mathcal{I}|n)$ time for $b \leq k$. Then $\sigma^{2b}b^2 \log b \in \mathcal{O}(n^2)$ for $b = \lfloor(\log_\sigma n)/2\rfloor$ yields the statement. □

For the **AETFS** setting, we can apply the Four Russians method above to each #-node and the source node in the decision DAG. Thus, we obtain the following corollary.

Corollary 1. *The **AETFS** problem can be solved in $\mathcal{O}(n^2 + \min\{n^2 \log^2 k, \frac{n^2 k}{\log_\sigma n}\} + \min\{n^2 \log^2 \ell, nL\})$ time.*

6.2 Further Speedup for Highly Repetitive Strings

For highly repetitive strings, we can obtain a faster algorithm than Theorem 5 by modifying the method of [15]. We define the two following factorizations for a given string. A sequence of strings $(s_0, \ldots, s_{t-1})$ is called a *factorization* of string S if and only if $s_0 \cdots s_{t-1} = S$.

LZ78-factorization ([39]) A factorization $LZ(W) = (f_0 = \varepsilon, f_1, \ldots, f_z)$ of string W is called the *LZ78-factorization* of W if and only if each f_i with $1 \leq i \leq z$ is the longest prefix of $W[|f_0 \ldots f_{i-1}| \,..\, |W| - 1]$ such that $f_i = f_j c$ for some $j < i$ and letter c. The *LZ78-trie* of W is a rooted tree such that each node represents a factor f_i, and there is an edge (f_j, f_i) with label c if $f_i = f_j c$. Given a string W of length n, $LZ(W)$ and the LZ78-trie of S can be computed in $\mathcal{O}(n)$ time [33]. See Figure 4 for examples of LZ78-factorization and LZ78-trie.

G-factorization We define a new factorization based on the LZ78-factorization and the special letters Γ as follows. Let $LZ(S) = (f_0, \ldots, f_z)$ be the LZ78-factorization of string $S \in (\Sigma \cup \Gamma)^*$. For every factor f_i we check if $f_i = xcy$ for $x \in \Sigma^*$, $c \in \Gamma$, and $y \in (\Sigma \cup \Gamma)^*$. If so, we further factorize it into three factors (x, c, y), and continue checking y recursively. After that, we rename the factors by $(g_0, \ldots, g_\ell)$. We call such a factorization the *G-factorization* of S, and we denote it by $G(S)$. We define the *G-trie*, which is a variant of LZ78-trie, w.r.t. G-factorization. The G-trie of S is a rooted tree such that (1) each node u represents a multiset of factors $\{\{g_{u_1}, \ldots, g_{u_t}\}\}$ with $g_{u_1} = \cdots = g_{u_t}$, (2) $g_{u_1} \neq g_{v_1}$ if $u \neq v$, and (3) there is an edge (v, u) with label $c \in \Sigma \cup \Gamma$ if $g_{u_1} = g_{v_1} c$. Notice that, if $S \in \Sigma^*$, then $LZ(S) = G(S)$ and its G-trie is the same as its LZ78-trie. It should also be clear that $|G(S)| \leq |LZ(S)| + 2\sum_{c\in\Gamma} |occ_S(c)|$, where $occ_S(c)$ denotes the total number of occurrences of letter c in S. See Figure 4 for examples of G-factorization and G-trie.

Also, given $LZ(S)$, we can compute $G(S)$ in $\mathcal{O}(|S|)$ time. Finally, we show that we can construct the G-trie of S over $\Sigma \cup \Gamma$ from $G(S)$ in $\mathcal{O}(|S|)$ time. We first construct the suffix tree of S in $\mathcal{O}(|S|)$ time [16]. Next, we compute the loci of all G-factors in $G(S)$ in the suffix tree, and make them explicit in $\mathcal{O}(|S| + |G(S)|)$ time [27], and then, we remove all the other nodes except for the loci from the suffix tree. Thus, we can construct G-trie of S in $\mathcal{O}(|S|)$ time.

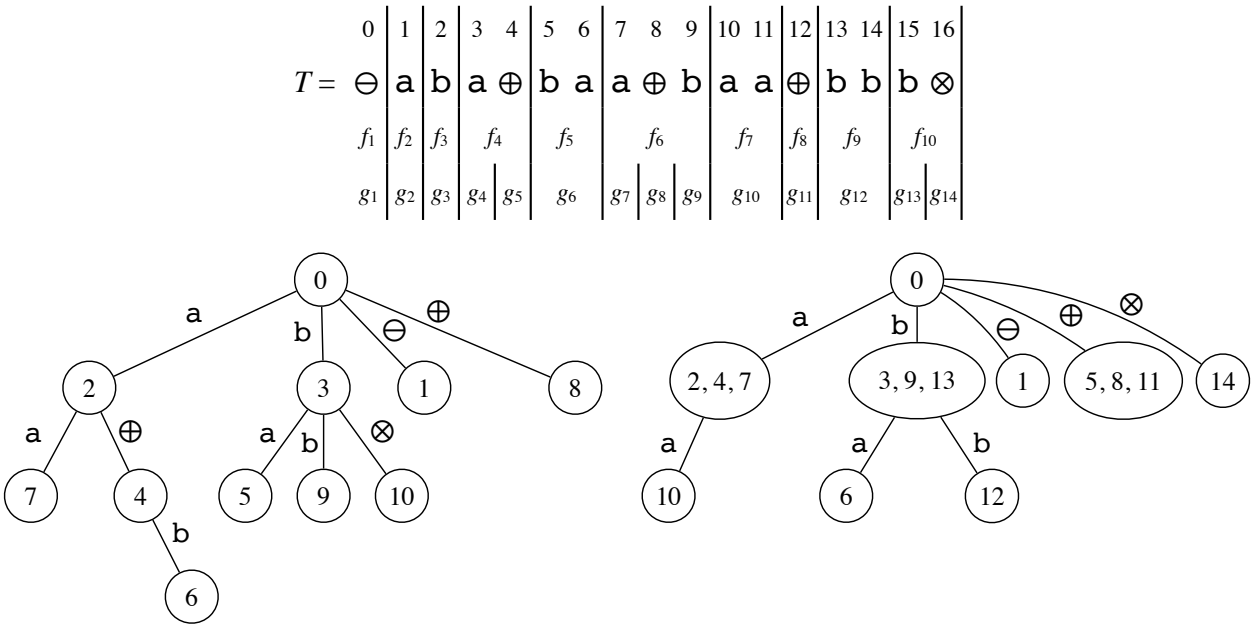

Figure 4: Examples for LZ78-factorization, LZ78-trie, G-factorization, and G-trie of string $T = \ominus\texttt{aba} \oplus \texttt{baa} \oplus \texttt{baa} \oplus \texttt{bbb}\otimes$. Factorizations $(f_0 = \varepsilon, f_1, \ldots, f_{10})$ and $(g_0 = \varepsilon, g_1, \ldots, g_{14})$ of T are respectively the LZ78-factorization and the G-factorization of T. The tree on the left is the LZ78-trie of T and the tree in the right is the G-trie of T.

Utilizing G-factorization In the rest of this section, we prove the following theorem.

Theorem 6. *The **ETFS** problem can be solved in time* $\mathcal{O}(n^2 + |LZ(W)|kn + |LZ(T)|n) \subseteq \mathcal{O}(n^2 + \frac{n^2 k}{\log_\sigma n})$.

The following corollary immediately holds.

Corollary 2. *The **AETFS** problem can be solved in time* $\mathcal{O}(n^2 + \min\{n^2 \log^2 k, |LZ(W)|kn + |LZ(T)|n\} + \min\{n^2 \log^2 \ell, nL\}) \subseteq \mathcal{O}(n^2 + \min\{n^2 \log^2 k, \frac{n^2 k}{\log_\sigma n}\} + \min\{n^2 \log^2 \ell, nL\})$.

First, we provide an overview of the algorithm of [15] for the standard edit distance problem. Let A and B be the input strings both of length n, and let z_a and z_b be respectively the size of the LZ78-factorization of A and B. They first partition the DP table into $z_a \times z_b$ blocks each corresponding to a comparison of an LZ78 factor of A with an LZ78 factor of B. Each block implicitly holds the *DIST* table, which stores the minimal weights from all input cells of the block to all output cells of the block. It is known that *DIST* is totally monotone [15], and thus, given values of input cells we can compute all row and column maxima of *DIST* in time linear in the length of the boundary of the block by using the SMAWK algorithm [3]. The total size of *DIST* tables for all blocks is $\mathcal{O}(n^2)$; however, each block actually stores only one column of *DIST* and $\mathcal{O}(bnd)$ pointers to previous blocks which store all the other columns, where *bnd* is the length of the boundary of the block. The total time and space complexity is $\mathcal{O}((z_a + z_b)n) \subseteq \mathcal{O}(\frac{n^2}{\log_\sigma n})$. The correctness of their algorithm can be proved by using the following property of the LZ78-factorization.

Property 1. *Let $LZ(S) = (f_0, f_1, \ldots, f_z)$ be the LZ78-factorization of string S. For each LZ78-factor f_i with $0 < i \leq z$, there exists a factor f_j such that $j < i$ and $f_i = f_j c$ for a letter c. Also, when f_i is given, we can obtain such f_j and c in constant time by using the LZ78-trie of S.*

Since the G-factorization holds the same properties as the above, we can apply their method even if we use G-factorization instead. We consider the LZ78-factorization of W and the G-factorization of T, and then, apply their method to **ETFS**. Therefore, we obtain Theorem 6.

Note that Theorem 6 is at least as good as Theorem 5; and Corollary 2 is at least as good as Corollary 1. Actually, the size of the LZ78-factorization of a string of length n is in $\mathcal{O}(hn/\log_\sigma n)$ for most strings (i.e. strings that are compressible), where $0 < h \leq 1$ is the entropy of the string [28, 39, 24, 15]. As a consequence, the **ETFS** problem can be solved in $\mathcal{O}(n^2 + \frac{(h_W+h_T)n^2k}{\log_\sigma n})$ time for such strings, where h_W and h_T are, respectively, the entropy of W and the entropy of T.

Acknowledgments We wish to thank Grigorios Loukides (King's College London) for useful discussions about improving the presentation of this manuscript.

References

[1] Osman Abul, Francesco Bonchi, and Fosca Giannotti. Hiding sequential and spatiotemporal patterns. *IEEE Trans. Knowl. Data Eng.*, 22(12):1709–1723, 2010.

[2] Osman Abul and Harun Gökçe. Knowledge hiding from tree and graph databases. *Data Knowl. Eng.*, 72:148–171, 2012.

[3] Alok Aggarwal, Maria M Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1-4):195–208, 1987.

[4] Charu C. Aggarwal. Applications of frequent pattern mining. In Charu C. Aggarwal and Jiawei Han, editors, *Frequent Pattern Mining*, pages 443–467. Springer, 2014.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 3–14. IEEE Computer Society, 1995.

[6] Vladimir L. Arlazarov, Yefim A. Dinitz, MA Kronrod, and Igor A. Faradzhev. On economical construction of the transitive closure of an oriented graph. *Doklady Akademii Nauk*, 194(3):487–488, 1970.

[7] Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. String sanitization: A combinatorial approach. In Ulf Brefeld, Élisa Fromont, Andreas Hotho, Arno J. Knobbe, Marloes H. Maathuis, and Céline Robardet, editors, *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2019, Würzburg, Germany, September 16-20, 2019, Proceedings, Part I*, volume 11906 of *Lecture Notes in Computer Science*, pages 627–644. Springer, 2019.

[8] Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giovanna Rosone, and Michelle Sweering. String sanitization: A combinatorial approach. *CoRR*, abs/1906.11030, 2019.

[9] Giulia Bernardini, Huiping Chen, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Leen Stougie, and Michelle Sweering. String Sanitization Under Edit Distance. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:14, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[10] Luca Bonomi, Liyue Fan, and Hongxia Jin. An information-theoretic approach to individual sequential data sanitization. In Paul N. Bennett, Vanja Josifovski, Jennifer Neville, and Filip Radlinski, editors, *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, San Francisco, CA, USA, February 22-25, 2016*, pages 337–346. ACM, 2016.

[11] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97. IEEE Computer Society, 2015.

[12] Brian Brubach and Jay Ghurye. A succinct four Russians speedup for edit distance computation and one-against-many banded alignment. In *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[13] Meng Chen, Xiaohui Yu, and Yang Liu. Mining moving patterns for predicting next location. *Inf. Syst.*, 54:156–168, 2015.

[14] Chris Clifton and Don Marks. Security and privacy implications of data mining. In *In ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 15–19, 1996.

[15] Maxime Crochemore, Gad M Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM journal on computing*, 32(6):1654–1673, 2003.

[16] Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM (JACM)*, 47(6):987–1011, 2000.

[17] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[18] Aris Gkoulalas-Divanis and Grigorios Loukides. Revisiting sequential pattern hiding to enhance utility. In Chid Apté, Joydeep Ghosh, and Padhraic Smyth, editors, *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 1316–1324. ACM, 2011.

[19] Aris Gkoulalas-Divanis and Vassilios S. Verykios. An integer programming approach for frequent itemset hiding. In Philip S. Yu, Vassilis J. Tsotras, Edward A. Fox, and Bing Liu, editors, *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management, Arlington, Virginia, USA, November 6-11, 2006*, pages 748–757. ACM, 2006.

[20] Aris Gkoulalas-Divanis and Vassilios S. Verykios. Exact knowledge hiding through database extension. *IEEE Trans. Knowl. Data Eng.*, 21(5):699–713, 2009.

[21] Robert Gwadera, Aris Gkoulalas-Divanis, and Grigorios Loukides. Permutation-based sequential pattern hiding. In Hui Xiong, George Karypis, Bhavani M. Thuraisingham, Diane J. Cook, and Xindong Wu, editors, *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*, pages 241–250. IEEE Computer Society, 2013.

[22] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001.

[23] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

[24] Philippe Jacquet and Wojciech Szpankowski. Asymptotic behavior of the lempel-ziv parsing scheme and digital search trees. *Theoretical Computer Science*, 144(1-2):161–197, 1995.

[25] Philip N Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 146–155. Society for Industrial and Applied Mathematics, 2005.

[26] Daniel C. Koboldt, Karyn M. Steinberg, David E. Larson, Richard K. Wilson, and Elaine R. Mardis. The next-generation sequencing revolution and its impact on genomics. *Cell*, 155(1):27–38, 2013.

[27] Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear-time algorithm for seeds computation. *ACM Trans. Algorithms*, 16(2):27:1–27:23, 2020.

[28] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 1976.

[29] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.

[30] Grigorios Loukides and Robert Gwadera. Optimal event sequence sanitization. In Suresh Venkatasubramanian and Jieping Ye, editors, *Proceedings of the 2015 SIAM International Conference on Data Mining, Vancouver, BC, Canada, April 30 - May 2, 2015*, pages 775–783. SIAM, 2015.

[31] William J Masek and Michael S Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System sciences*, 20(1):18–31, 1980.

[32] Eugene W. Myers and Webb Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1989.

[33] Yuto Nakashima, I Tomohiro, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Constructing LZ78 tries and position heaps in linear time for large alphabets. *Information Processing Letters*, 115(9):655–659, 2015.

[34] Stanley R. M. Oliveira and Osmar R. Zaïane. Protecting sensitive knowledge by data sanitization. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003), 19-22 December 2003, Melbourne, Florida, USA*, pages 613–616. IEEE Computer Society, 2003.

[35] Jeanette P Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

[36] Vassilios S. Verykios, Ahmed K. Elmagarmid, Elisa Bertino, Yücel Saygin, and Elena Dasseni. Association rule hiding. *IEEE Trans. Knowl. Data Eng.*, 16(4):434–447, 2004.

[37] Yi-Hung Wu, Chia-Ming Chiang, and Arbee L. P. Chen. Hiding sensitive association rules with limited side effects. *IEEE Trans. Knowl. Data Eng.*, 19(1):29–42, 2007.

[38] Josh Jia-Ching Ying, Wang-Chien Lee, Tz-Chiao Weng, and Vincent S. Tseng. Semantic trajectory mining for location prediction. In Isabel F. Cruz, Divyakant Agrawal, Christian S. Jensen, Eyal Ofek, and Egemen Tanin, editors, *19th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2011, November 1-4, 2011, Chicago, IL, USA, Proceedings*, pages 34–43. ACM, 2011.

[39] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.