Resilient Pattern Mining

Pengxin Bian[1], Panagiotis Charalampopoulos[1], Lorraine A. K. Ayad[2], Manal Mohamed[1], Solon P. Pissis[3,4], and Grigorios Loukides[1]

[1]King's College London [2]Brunel University of London [3]CWI [4]Vrije Universiteit Amsterdam
{pengxin.bian, p.charalampopoulos, grigorios.loukides}@kcl.ac.uk, lorraine.ayad@brunel.ac.uk, manalabd@gmail.com, solon.pissis@cwi.nl

Abstract*—Frequent pattern mining is a flagship problem in data mining. In its most basic form, it asks for the set of substrings of a given string S of length n that occur at least τ times in S, for some integer $\tau \in [1, n]$. We introduce a *resilient* version of this classic problem, which we term the (τ, k)-Resilient Pattern Mining (RPM) problem. Given a string S of length n and two integers $\tau, k \in [1, n]$, RPM asks for the set of substrings of S that occur at least τ times in S, even when the letters at *any* k positions of S are substituted by other letters. Unlike frequent substrings, resilient ones account for the fact that changes to string S are often expensive to handle or are unknown. We make the following contributions. First, we present RPM-DP, a simple exact $\mathcal{O}(n^3 k \log n)$-time and $\mathcal{O}(n^2)$-space algorithm for RPM that is based on an existing dynamic programming algorithm. Second, we propose RPM-ESA, an exact $\mathcal{O}(n \log n)$-time and $\mathcal{O}(n)$-space algorithm for RPM, which employs advanced data structures and combinatorial insights. Third, we conduct experiments on real large-scale datasets from different domains demonstrating that: (I) The notion of resilient substrings is useful in analyzing genomic data and fundamentally different from that of frequent substrings, as frequent substrings are often not resilient and thus do not remain frequent for long in versioned datasets; (II) RPM-ESA is *several orders of magnitude* faster and more space-efficient than RPM-DP; and (III) Clustering based on resilient substrings is effective.

I. Introduction

Given a string $S = S[0..n-1]$ of length n and an integer $\tau \in [1, n]$, the classic frequent pattern mining (FPM) problem [1]–[3] asks for the set of substrings of S that occur at least τ times in S. These substrings are referred to as τ-*frequent*. We introduce a *resilient* version of FPM, which we term the (τ, k)-Resilient Pattern Mining (RPM) problem. Informally stated, given a string S of length n and integers $\tau, k \in [1, n]$, RPM asks for the set of substrings of S that occur at least τ times in S, even when *any* k positions of S undergo letter substitutions. Clearly, RPM is a generalization of the FPM problem; it corresponds to the $(\tau, 0)$-RPM case.

Example 1. *Let* $S = \texttt{aaabaaaabbaaa}$, $\tau = 2$, *and* $k = 1$. *The output of* (τ, k)-Resilient Pattern Mining *is* $\{\texttt{aaa}, \texttt{aa}, \texttt{a}, \texttt{b}\}$. *Each such substring is* $(2, 1)$-*resilient as it has (at least) two occurrences in* S *even after the letter at any single position of* S *is substituted. For example, the substring* $\texttt{aaa}$ *of* S *is* 2-*frequent in* $S' = \underline{\texttt{aaa}}\texttt{ba}\texttt{b}\texttt{aabb}\underline{\texttt{aaa}}$, *as even when letter* $\texttt{b}$ *substitutes* $\texttt{a}$ *at position* 5 *of* S, *two occurrences of* $\texttt{aaa}$ *(those underlined in* S'*) still exist.*

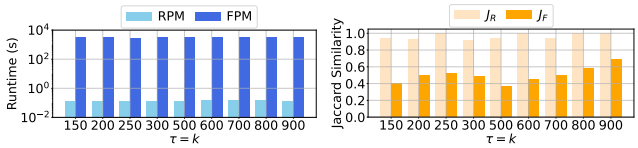
Here, we consider RPM under letter substitutions; considering other edit operations is an intriguing research direction.

Motivation. Unlike frequent substrings, resilient ones account for the fact that changes to a string are often expensive to handle (e.g., we may need to perform FPM on the new string from scratch even if it is highly similar to the original one) or are unknown (e.g., when caused by adversaries [4] or they occur in the future). RPM is motivated by applications in:

1. *Bioinformatics:* In this application domain, there are large collections of very similar strings (also referred to as *repetitive*) [5]. For example, conserved regions in DNA are stretches of nucleotides that remain relatively unchanged across different individuals, species, or time periods, indicating that these regions likely play essential biological roles [6]. However, even in conserved regions, there may be slight variations due to mutations, sequencing errors, or other factors [6]. Similarly, a pangenome is a collection of DNA sequences each corresponding to the same part of DNA of a different individual [7]. To find conserved regions in a DNA sequence collection, or motifs of interest in a pangenome, a straightforward approach is to solve FPM in each sequence of the collection and then output only the substrings that are frequent in all sequences. This, however, takes time proportional to the size of the collection, which can be in the order of TBs [8]. An alternative approach is to exploit the fact that a bound on *the number* of changes (letter substitutions) of the strings in the collection is known or can be estimated [5] and apply RPM into any of these strings. This approach is much faster and approximates the set of patterns that are frequent in all strings remarkably well, as shown in Example 2.

Example 2. *Consider the collection* $\mathcal{C}$ *of* $143,588$ *SARS-CoV-2 DNA sequences from [9] with total length* $4.176 \cdot 10^9$. *Fig. 1a shows that mining* (τ, k)-*resilient substrings from an arbitrarily selected sequence in* $\mathcal{C}$ *is more than* four orders of magnitude *faster on average than mining the* τ-*frequent substrings that appear in all sequences in* $\mathcal{C}$. *Fig. 1b shows the Jaccard Similarity [10]* J_R *between the set of* τ-*frequent substrings in all sequences in* $\mathcal{C}$ *and the set of* (τ, k)-*resilient substrings.* J_R *is* 0.96 *on average, implying that the set of* (τ, k)-*resilient substrings approximates the set of* τ-*frequent ones very well (i.e., the former substrings are frequent in almost all sequences in* $\mathcal{C}$*). Fig. 1b also shows the Jaccard Similarity* J_F *between the set of* τ-*frequent substrings in all*

sequences and the set of τ-frequent substrings in an arbitrarily selected sequence from $\mathcal{C}$. J_F is 0.5 on average and no more than 0.68, implying that the latter substrings are not τ-frequent in most of the sequences in $\mathcal{C}$.



(a) (b)

Fig. 1: (a) Running time and (b) Jaccard similarity.

2. *Text Analytics:* Text data representing webpages [11], tweets [12], or code in software repositories [13] often undergo changes due to updates, leading to slightly different versions of the same text [14]. Such updates may increase or decrease the substring frequencies. Applying RPM to a given version of the data ensures that the output substrings remain τ-frequent even after k letter substitutions may occur in later versions. This is useful when outsourcing frequent substrings [15] or using them in downstream analyses [16], as (τ, k)-resilient substrings remain truthful for longer in these datasets.

3. *Recommender Systems:* String letters are often substituted by adversaries to force a recommender that is trained on the string to favor their items [17]. Frequent substrings mined from such modified strings may be spurious and others that are indeed frequent may be missed, as letter substitutions can increase or decrease substring frequencies [18]. On the other hand, RPM ensures that spurious substrings are not mined and that the mined ones are indeed frequent, provided that k is set to a sufficiently large value.

From a technical perspective, RPM is motivated as follows: For any value of k, a brute-force algorithm for RPM considers all possible $\Omega(\binom{n}{k})$ substitutions in S, and, for each new instance S' of S, it computes the substrings occurring at least τ times in S' and outputs the substrings that are frequent in all S' instances. Clearly, this algorithm is impractical for strings of realistic length n even for constant k. Furthermore, finding frequent patterns with Hamming (or edit) distance at most k is NP-hard [19], and thus exact algorithms for this problem also have some exponential dependency on k [20]. It is thus interesting to investigate whether the RPM problem can be solved in polynomial time when k is superconstant.

Contributions and Paper Organization. In this work, we introduce RPM and make the following contributions:

1. We design a simple exact $\mathcal{O}(n^3 k \log n)$-time and $\mathcal{O}(n^2)$-space algorithm for RPM. The main idea is to efficiently find the longest (τ, k)-resilient substring starting at each position of S. These substrings form a compact $\mathcal{O}(n)$-size representation of the (τ, k)-resilient substrings. At the heart of our algorithm is a monotonicity property that allows us to apply binary search on the length of the substring starting from a certain position of S and a reduction that allows us to use a dynamic programming algorithm from [21] that solves a scheduling problem. We call our algorithm RPM-DP. See Section III.

2. We design an exact $\mathcal{O}(n \log n)$-time and $\mathcal{O}(n)$-space algorithm for RPM which employs advanced data structures and combinatorial insights. The algorithm outputs the same $\mathcal{O}(n)$-size representation as RPM-DP but is theoretically and practically more time- and space-efficient. Surprisingly, the complexity of our algorithm is independent of τ and k. Its efficiency stems from: (I) the use of an index of S to compactly represent and efficiently explore the space of solutions; and (II) novel algorithms based on combinatorial insights for efficiently checking whether a *periodic* (see Section II for a definition) or an *aperiodic* substring is resilient. Interestingly, these checks are performed efficiently based on counting arguments; and not by modifying the string and then inferring the τ-frequent substrings. We provide two implementations of this algorithm: one called RPM-ST that is based on a suffix tree [22]; and another one called RPM-ESA that is based on an enhanced suffix array [23]. See Section IV.

3. We present experiments on *seven real datasets* from different domains showing that: (I) Resilient substrings are a fundamentally different notion than frequent substrings, as frequent substrings are often not resilient and, unlike resilient substrings, they do not remain frequent for long in versioned datasets. (II) RPM-ST and RPM-ESA significantly outperform RPM-DP in terms of running time and space consumption and, as expected, their running time is not affected by τ or k. For instance, RPM-ST is more than *two orders of magnitude* faster and more space-efficient than RPM-DP. Also, RPM-ESA is up to about an order of magnitude faster than RPM-ST, and it uses about *three times* less space. (III) Clustering based on resilient substrings is effective, as it outperforms a widely-used clustering algorithm [24] and a clustering algorithm [25] based on τ-frequent patterns, achieving results very close to the ground truth. See Section VI.

Related work is discussed in Section V.

II. Preliminaries

Strings. An *alphabet* Σ is a finite set of elements (e.g., integers, characters, etc.), which we call *letters*. A *string* $S = S[0\,..\,n-1]$ of *length* $|S| = n$ is a sequence of n letters from Σ, where $S[i]$ denotes the i-th letter of the sequence. We refer to each $i \in [0, n)$ as a *position* of S. We consider throughout an alphabet $\Sigma = [0, \sigma)$ with $\sigma = n^{\mathcal{O}(1)}$, which captures virtually any realistic scenario. A substring R of S may occur multiple times in S. The set of the starting positions of its *occurrences* in S is denoted by $\mathsf{occ}_S(R)$; we define the *frequency* of R to be $|\mathsf{occ}_S(R)|$ and may omit the subscript S when it is clear from the context. An occurrence of R in S starting at position i is referred to as a *fragment* of S and is denoted by $R = S[i\,..\,i + |R| - 1]$. Thus, different fragments may correspond to different occurrences of the same substring. A *prefix* of S is a substring of the form $S[0\,..\,j]$, and a *suffix* of S is a substring of the form $S[i\,..\,n-1]$. A prefix $S[0\,..\,j]$ (resp., suffix $S[i\,..\,n-1]$) is *proper* if $j < n-1$ (resp., $i > 0$). Thus any fragment of S is a prefix of some suffix of S. The concatenation of two strings S and S' is denoted by $S \cdot S'$ and the concatenation of k copies of S is denoted by S^k. The

Hamming distance $d_H(S, S')$ of two strings $S, S' \in \Sigma^n$ of equal length n is $|\{i \in [0, n) : S[i] \neq S'[i]\}|$.

Definition 1 (Period)**.** *An integer $p > 0$ is a* period *of a string P if $P[i] = P[i+p]$ for all $i \in [0, |P|-p)$. The smallest period of P is referred to as* the period *of P and is denoted by $\mathsf{per}(P)$. String P is called* periodic *if and only if $\mathsf{per}(P) \leq |P|/2$ and* aperiodic *otherwise.*

Example 3. *For $P =$ abaaabaaabaaaba, $\mathsf{per}(P) = 4$ as $P[i] = P[i+4]$ for all $i \in [0, 15-4)$; $p = 8$ is also a period of P. Since $\mathsf{per}(P) = 4 \leq |P|/2 = 15/2$, P is periodic.*

Definition 2 (Run)**.** *A fragment F of a string S is a* run *of S if and only if $\mathsf{per}(F) \leq |F|/2$ and F cannot be extended in either direction with its period remaining the same.*

Example 4. *Consider $S =$ abaaabaaabaaabab. The* underlined *fragment $F = S[0\,..\,14]$ is a run, as its period $\mathsf{per}(F) = 4 \leq |F|/2 = 15/2$ and it cannot be extended to the right by one letter with its period remaining the same, as then it would become equal to S and $\mathsf{per}(S) = 14 \neq 4$.*

Definition 3 (Lyndon Root)**.** *The* Lyndon root *of a run F is the lexicographically smallest rotation of $F[0\,..\,\mathsf{per}(F)-1]$.*

Definition 4. *A run F with Lyndon root L has a unique* canonical representation *$L_1 \cdot L^k \cdot L_2$, where L_1 is a proper suffix of L, L_2 is a proper prefix of L, and $k \geq 0$ is a nonnegative integer; either of L_1 and L_2 may be the empty string.*

Example 5. *The run $F =$ abaaabaaabaaaba has Lyndon root aaab, as the rotations of $F[0\,..\,4-1]$ are abaa, baaa, aaab, and aaba, and the lexicographically smallest one is aaab. The canonical representation of F is $\mathtt{ab} \cdot (\mathtt{aaab})^3 \cdot \mathtt{a}$.*

Indexes. For a set $\mathcal{S}$ of strings, the *trie* $\mathsf{T}(\mathcal{S})$ is a rooted tree whose nodes are in one-to-one correspondence with the prefixes of the strings in $\mathcal{S}$ [22]. The edges of $\mathsf{T}(\mathcal{S})$ are labeled with letters from Σ; the prefix corresponding to node v is denoted by $\mathsf{str}(v)$ and equals the concatenation of the letters labeling the edges on the root-to-v path. The node v is called the *locus* of $\mathsf{str}(v)$. The order on Σ induces an order on the edges outgoing from any node of $\mathsf{T}(\mathcal{S})$. A node v is *branching* if it has at least two children and *terminal* if $\mathsf{str}(v) \in \mathcal{S}$.

A *compacted trie* is obtained from $\mathsf{T}(\mathcal{S})$ by dissolving all nodes except the root, the branching, and the terminal nodes. The dissolved nodes are called *implicit*; the preserved ones are called *explicit*. The edges of the compacted trie are labeled by strings. The *string depth* $\mathsf{sd}(v) = |\mathsf{str}(v)|$ of node v is the length of the string it represents, i.e., the total length of the strings labeling the path from the root to v. The compacted trie takes $\mathcal{O}(|\mathcal{S}|)$ space provided that edge labels are stored as fragments of strings in $\mathcal{S}$ (a fragment $S[i\,..\,j]$ can be stored in $\mathcal{O}(1)$ space [22]). Given the lexicographic order on $\mathcal{S}$ along with the lengths of the longest common prefixes between any two consecutive (in this order) elements of $\mathcal{S}$, one can construct the compacted trie of $\mathcal{S}$ in $\mathcal{O}(|\mathcal{S}|)$ time [26].

The *suffix tree* $\mathsf{ST}(S)$ of a string S is the compacted trie of the set of all suffixes of S; see Fig. 2a. Each terminal node v of ST is labeled by $n - \mathsf{sd}(v)$, i.e., the starting position of the suffix it represents. $\mathsf{ST}(S)$ can be constructed in $\mathcal{O}(n)$ time for any string S of length n over Σ [22]. After an $\mathcal{O}(n)$-time preprocessing of $\mathsf{ST}(S)$, we can compute the length of the longest common prefix of any two suffixes in $\mathcal{O}(1)$ time. We denote the length of the longest common prefix of $S[i\,..\,n-1]$ and $S[j\,..\,n-1]$ by $\mathsf{lcp}(i, j)$. The *suffix array* $\mathsf{SA}(S)$ of S [27] is the permutation of $[0, n)$ such that $\mathsf{SA}[i]$ is the starting position of the i-th lexicographically smallest suffix of S; see Fig. 2b and Fig. 2c. It can be constructed in $\mathcal{O}(n)$ time for any string S of length n over Σ [22]. The $\mathsf{LCP}(S)$ array [27] of S stores the length of the longest common prefix of lexicographically adjacent suffixes. For $j > 0$, $\mathsf{LCP}[j] = \mathsf{lcp}(j-1, j)$ and $\mathsf{LCP}[0] = 0$; see Fig. 2d. Given $\mathsf{SA}(S)$, $\mathsf{LCP}(S)$ can be constructed in $\mathcal{O}(n)$ time [26].

Problem Definition. We define the main problem as follows.

Definition 5 (τ-frequent substring)**.** *A substring X of S is* τ-frequent *in S if it occurs at least τ times in S.*

Definition 6 ((τ, k)-resilient substring)**.** *A substring X of $S \in \Sigma^n$ is* (τ, k)-resilient *in S if X is τ-frequent in all strings $S' \in \Sigma^n$ that satisfy $d_H(S, S') \leq k$.*

(τ, k)-Resilient Pattern Mining (RPM)
Input: A string $S \in \Sigma^n$ and two integers $\tau, k \in [1, n]$.
Output: An array OUTPUT of size n such that $\mathsf{OUTPUT}[i]$ is the length of the longest prefix of $S[i\,..\,n-1]$ that is (τ, k)-resilient.

Parameter k can be set based on domain knowledge (e.g., in bioinformatics, we can use the substitution rate [28]). For conceptual simplicity, we first present our algorithms under the assumption that all substitutions in S are done using a letter $\# \notin \Sigma$. In §I in [29], we generalize our solution, waiving the assumption for alphabets of size at least 4. The array OUTPUT is a *compact representation* of the set of (τ, k)-resilient substrings of S. It is based on Fact 1; see §II in [29] for the proof of this fact.

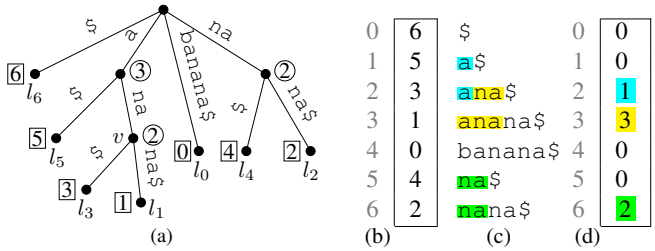
Fact 1 (Monotonicity)**.** *If string $Z = X \cdot Y$ is (τ, k)-resilient in S, where X, Y are strings, then so are X and Y.*

In §III in [29], we explain how to explicitly construct the set of (τ, k)-resilient substrings of S using the OUTPUT array in $\mathcal{O}(n)$ time plus time linear in the number of these substrings.

III. A DP-based Polynomial Algorithm for RPM

We design RPM-DP, a dynamic programming (DP) based polynomial-time algorithm for (τ, k)-Resilient Pattern Mining, for any integer $\tau, k \in [1, n]$. RPM-DP is founded on the following result.

Theorem 1 ([21], Section 3.1)**.** *Given a family $\mathcal{F}$ of f arbitrary intervals on the real line and a positive integer $k \leq f$, we can find a set of at most k points on the line that intersect a maximum number of intervals from $\mathcal{F}$ in $\mathcal{O}(kf^2)$ time and $\mathcal{O}(f^2)$ space.*



(a) (b) (c) (d)

Fig. 2: $S = \texttt{banana\$}$: (a) $\mathsf{ST}(S)$; each branching non-root node is labeled by its frequency (circle) and each terminal node l_i by its associated starting position i (square). Node v is the locus of $\mathsf{str}(v) = \texttt{ana}$ and $\mathsf{sd}(v) = |\mathsf{str}(v)| = 3$. (b) $\mathsf{SA}(S)$. (c) The lexicographically sorted suffixes of S. (d) $\mathsf{LCP}(S)$.

By Fact 1, we can binary search on the length of the substring starting at position i, for all $i \in [0, n)$, to find the longest substring $S[i\,..\,j]$ that is (τ, k)-resilient. Thus, it suffices to check $\mathcal{O}(n \log n)$ substrings of S.

Let us explain how we check whether a fixed substring Z of S is (τ, k)-resilient. We first find $\mathsf{occ}_S(Z)$, the set of occurrences of Z in S, in $\mathcal{O}(n)$ time using the KMP algorithm [30]. If $|\mathsf{occ}_S(Z)|$ is at least τ we apply the DP algorithm from [21]; we also describe it in §IV in [29]. This DP algorithm underlies Theorem 1; it finds a set of at most k points on the line that intersect a maximum number M of intervals in a given family of f intervals, in $\mathcal{O}(kf^2)$ time and $\mathcal{O}(f^2)$ space. Specifically, for each occurrence $i \in \mathsf{occ}_S(Z)$, the interval $[i, i + |Z| - 1]$ represents the fragment $S[i\,..\,i + |Z| - 1]$ and the points that intersect the intervals represent the letters of the fragments that, if substituted by $\# \notin \Sigma$, change the frequency of Z as much as possible. Hence, $f = |\mathsf{occ}_S(Z)|$ and Z is *not* (τ, k)-resilient if and only if $f - M < \tau$ (in which case Z would cease being τ-frequent after the letter substitutions).

Since we check $\mathcal{O}(n \log n)$ substrings, and each check takes $\mathcal{O}(kn^2)$ time and $\mathcal{O}(n^2)$ space, as $f = |\mathsf{occ}_S(Z)| \leq n$, we obtain an $\mathcal{O}(n^3 k \log n)$-time and $\mathcal{O}(n^2)$-space algorithm for RPM. The algorithm returns the OUTPUT array as required.

Proposition 1. *For any string $S \in \Sigma^n$, with $|\Sigma| \geq 4$, and integers $\tau, k \in [1, n]$, the (τ, k)-Resilient Pattern Mining problem can be solved in $\mathcal{O}(n^3 k \log n)$ time and $\mathcal{O}(n^2)$ space.*

IV. An Efficient Algorithm for RPM

High-Level Idea. The main idea is to *implicitly* consider all substrings of S. Each substring is either *aperiodic* or *periodic*: (I) The occurrences of an aperiodic substring R of S do not overlap a lot, so we can use a simple greedy algorithm to count the maximal number of occurrences of R that may be lost after k substitutions in S. If the frequency of R in S minus this number is at least τ, then R is (τ, k)-resilient; otherwise, it is not. (II) The occurrences of a periodic substring P of S can overlap a lot but have a predictable structure: they form compact clusters of occurrences of P and the clusters do not overlap much. The technical difficulty is to retrieve these clusters efficiently, in time proportional to their number and *not* to the total number of occurrences of P. After retrieving these clusters, we use a more involved version of the greedy algorithm to check whether P is (τ, k)-resilient in S or not using counting arguments similarly to the aperiodic case.

The total number of substrings of S is $\Theta(n^2)$. By Fact 1, it suffices to check $\mathcal{O}(n \log n)$ substrings: $\mathcal{O}(\log n)$ for each starting position of S in a binary-search fashion. This way we obtain a compact representation of the set of (τ, k)-resilient substrings of S in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(n)$ space.

Roadmap. The main algorithm is described in Section IV-B. It relies on two combinatorial lemmas (Section IV-A) and three functions (Section IV-C). We present the algorithm using a suffix tree [22], for simplicity. It is well-known that the suffix tree functionality (e.g., bottom-up traversal) can be simulated using an enhanced suffix array [23] (see §V.A in [29]), resulting in time and space improvements in practice.

A. Combinatorial Lemmas

We give two crucial lemmas for our algorithm's efficiency; they are folklore but we give their proofs in §V.B in [29]. Lemma 1 is useful for handling the aperiodic substrings of S. Intuitively, it guarantees that any position of S is contained in at most two occurrences of an aperiodic substring R.

Lemma 1. *Let S be a string and R be an aperiodic substring of S. For any three distinct occurrences of R in S at positions x, y, and z with $x < y < z$, we have $z > x + |R|$.*

Lemma 2 is useful for handling the periodic substrings of S. Intuitively, it guarantees that any position of S is contained in at most two runs with the same period.

Lemma 2. *Let S be a string. Suppose that for positions $i < j < k$ there are runs $S[i\,..\,i']$, $S[j\,..\,j']$, and $S[k\,..\,k']$ in S that have the same period. We then have $k > i'$.*

B. Main Algorithm

The main algorithm has two phases and returns OUTPUT, a compact representation of the (τ, k)-resilient substrings.

Preprocessing. We construct the suffix tree $\mathsf{ST}(S)$ of S in $\mathcal{O}(n)$ time and space. By construction (inspect Figs. 2a and 2b), the leaves of $\mathsf{ST}(S)$ of S, read from left to right (e.g., using a post-order traversal), precisely form the suffix array $\mathsf{SA}(S)$ of S. Using such a traversal, for each branching node u in $\mathsf{ST}(S)$, we record the SA indices of the leftmost and rightmost leaves in the subtree rooted at u, denoted by l_u and r_u, respectively. Then, u stores interval $[l_u, r_u]$ where, $\mathsf{str}(u) = S[\mathsf{SA}[l_u]\,..\,\mathsf{SA}[l_u] + \mathsf{sd}(u) - 1] = \ldots = S[\mathsf{SA}[r_u]\,..\,\mathsf{SA}[r_u] + \mathsf{sd}(u) - 1]$, and $r_u - l_u + 1$ is the frequency of $\mathsf{str}(u)$. Lastly, we associate each node of $\mathsf{ST}(S)$ with a flag initialized to FAIL. This preprosessing takes linear time in the size of $\mathsf{ST}(S)$, i.e., $\mathcal{O}(n)$ time.

Phase 1. We perform a bottom-up (DFS) traversal of $\mathsf{ST}(S)$ to find a *cut* of $\mathsf{ST}(S)$ (i.e., a set of nodes that partitions the suffix tree into disjoint subtrees) where the nodes that form the cut and their ancestors are exactly the nodes that represent (τ, k)-resilient substrings. When some node u is explored during the traversal, we check its flag. If the flag value is FAIL, then we read the SA interval $[l_u, r_u]$ stored

at u, check whether $r_u - l_u + 1 \geq \tau + k$, and, if so, apply the IsPeriodic function to $\mathsf{str}(u)$, which determines whether $\mathsf{str}(u)$ is periodic or not. If IsPeriodic answers YES, we apply the PeriodicResilient function to $\mathsf{str}(u)$; otherwise, we apply the AperiodicResilient function. If the applied function answers YES, $\mathsf{str}(u)$ is (τ, k)-resilient, and the flag of u is changed to SUCCESS; otherwise, $\mathsf{str}(u)$ is not (τ, k)-resilient and the flag of u is kept unchanged as FAIL. Whenever a node u has its flag changed to SUCCESS, all the ancestors of u should also have their flags changed to SUCCESS; we thus propagate the SUCCESS flag upward before proceeding with our traversal. The end of the traversal gives a *preliminary cut*. The upper part including the cut is composed of SUCCESS nodes and the lower part is composed of FAIL nodes.

Phase 2. We use Fact 1 to refine the preliminary cut from Phase 1. We try to extend the (τ, k)-resilient substrings we found in Phase 1 as much as possible to compute the final output. For each edge (u, v) in $\mathsf{ST}(S)$, such that: (a) $\mathsf{str}(u)$ is (τ, k)-resilient (i.e., the flag of u is equal to SUCCESS and u is on the cut); (b) $\mathsf{str}(v)$ is not (τ, k)-resilient (the flag of v is equal to FAIL and v is below the cut); and (c) $\mathsf{str}(v)$ has at least $\tau + k$ occurrences in S, we perform binary search on the interval $[\mathsf{sd}(u), \mathsf{sd}(v)]$ to compute *the longest prefix of* $\mathsf{str}(v)$ *that is* (τ, k)*-resilient*. (Note that if condition (c) does not hold, we can already conclude that the sought prefix of $\mathsf{str}(v)$ is $\mathsf{str}(u)$.) In particular, in each iteration of the binary search, we first call IsPeriodic and then either PeriodicResilient or AperiodicResilient (similarly to Phase 1). In the end, we obtain our *refined final cut*. All in all, for each edge (u, v) such that u is on the preliminary cut and v is not, we have computed the longest prefix of $\mathsf{str}(v)$ that is (τ, k)-resilient; let us denote the length of this prefix by $m(u, v)$. Then, for each such edge (u, v), we read the SA interval $[l_v, r_v]$ from v, and set $\mathsf{OUTPUT}[\mathsf{SA}[i]] := m(u, v)$, for all $i \in [l_v, r_v]$, in an initially all-zeroes array $\mathsf{OUTPUT}[0\,..\,n-1]$.

As $\mathsf{ST}(S)$ has $\mathcal{O}(n/(\tau+k))$ nodes, each with at least $\tau + k$ leaf descendants, that call the 3 functions, we *directly* obtain:

Lemma 3. *The (τ, k)-Resilient Pattern Mining problem can be reduced in $\mathcal{O}(n \log n)$ time, using $\mathcal{O}(n)$ space, to $\mathcal{O}(n \log n/(\tau+k))$ calls to IsPeriodic, PeriodicResilient, and AperiodicResilient.*

Remark 1. *Conceptually, one can consider a simpler algorithm that performs binary search to compute, for each node u of $\mathsf{ST}(S)$ with at least $\tau + k$ leaf descendants, the longest prefix of $\mathsf{str}(u)$ that is (τ, k)-resilient. The problem is that in order to do this, we would need random access to the sorted list of occurrences of $\mathsf{str}(v)$ at each node v of $\mathsf{ST}(S)$, which is costly. However, as we show below, we can maintain a sorted list of occurrences during a DFS traversal.*

C. The Functions of the Algorithm

In this section, we present the theoretically efficient implementations of the IsPeriodic, AperiodicResilient, and PeriodicResilient functions. For each function, we also discuss our practical implementation when it differs from the theoretically efficient one. These functions assume that, during the bottom-up (DFS) traversal of $\mathsf{ST}(S)$, at each node v, we can retrieve a sorted list of the occurrences of $\mathsf{str}(v)$ in S. Any such list is the *sorted* union of the disjoint lists of the children of v. By implementing these lists using mergeable AVL trees, we can compute all of them in $\mathcal{O}(n \log n)$ total time. This is rather standard but we also describe it in §V.C in [29].

IsPeriodic Function. For any string S of length n and a fragment F of S, a *periodic extension* query [31] decides whether F is periodic, and, if so, returns $\mathsf{per}(F)$. We rely on the following result.

Theorem 2 ([31])**.** *For any string of length n, after an $\mathcal{O}(n)$-time preprocessing, we can answer periodic extension queries in $\mathcal{O}(1)$ time.*

Unfortunately, there is no available implementation of the data structure underlying Theorem 2, and it seems that an implementation of it would require heavy engineering in order to be practically efficient; Theorem 2 relies on an intricate string sampling scheme and other sophisticated (auxiliary) data structures. We thus propose a practical implementation that relies on first computing the runs of S [32] and then indexing them using an interval tree [33].

We first compute the runs of S in $\mathcal{O}(n)$ time using the algorithm of [32]. Since there are $\mathcal{O}(n)$ runs [34], each represented by a subinterval of $[0, n)$, we index them in an interval tree in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space [33]. Given a substring $S[i\,..\,j]$, we return all m runs that contain $S[i\,..\,j]$ as a substring in $\mathcal{O}(m + \log n)$ time using the interval tree. By iterating over these runs, we check if $S[i\,..\,j]$ is periodic (Definition 1) and, if so, find its smallest period. The query time is thus $\mathcal{O}(m+\log n)$; in real-world strings, each position is expected to be in a few runs, so m is expected to be small.

AperiodicResilient Function. Let R be an aperiodic substring of S, such that $|\mathsf{occ}_S(R)| \geq \tau + k$. Function AperiodicResilient uses a greedy algorithm (see Algorithm 1) to compute the maximal number of occurrences of R that may be lost if k letters of S are substituted with a letter $\# \notin \Sigma$. Lemma 1 proves that, for any aperiodic substring R of S, such a substitution could result in the loss of at most two occurrences of R. We refer to each of the k substitutions, with $\# \notin \Sigma$, in our budget as a *token* and we say that a token *destroys* one (or more) occurrence(s) of R if the frequency of R is reduced by one (or more) as a result of (using) this token. Our algorithm is greedy in the sense that it first computes the tokens needed for the destruction of as many pairs of overlapping occurrences as possible spending one token per pair of overlapping occurrences (Lines 4-8). Note that an occurrence of R may overlap with at most two other occurrences of R (the one that precedes it and the one that succeeds it) and in this case only one pair of overlapping occurrences can be destroyed by a single token (Line 7). The algorithm then uses the remaining tokens to destroy single occurrences (Line 9). Clearly, Algorithm 1 takes linear time in the number of occurrences, as they are given sorted.

Algorithm 1 Compute the number of occurrences of a length-λ string that can be destroyed with t tokens in a sorted list $\mathcal{O}$ of occurrences

```
1: function Greedy(O, t, λ)
2:     k'' ← 0  ▷ Max num. of tokens that can each destroy 2 distinct occs
3:     i ← 0
4:     while i < |O| and k'' < t do          ▷ Iterate over all occurrences
5:         if i ≤ |O| − 2 and O[i + 1] − O[i] ≤ λ − 1
6:             k'' ← k'' + 1
7:             i ← i + 1                  ▷ Skip the (i + 1)-st occurrence of O
8:         i ← i + 1
9:     k' ← t − k''                                  ▷ Remaining number of tokens
10:    return k' + 2k''          ▷ Max number of occs that can be destroyed
```

Algorithm 2 presents our implementation of PeriodicResilient. For an aperiodic substring $R = S[i\,..\,j]$, we compute its frequency $|\mathsf{occ}_S(R)| = r_u - l_u + 1$ (Line 2), where $[l_u, r_u]$ is the SA interval stored for node u with $\mathsf{str}(u) = R$. Note that $S[i\,..\,j]$ is only considered if its frequency is greater than or equal to $\tau + k$ (see Phase 1). If the frequency is greater than or equal to $\tau + 2k$ then R is (τ, k)-resilient (Lines 3-4); this is because at most $2k$ occurrences of R can be destroyed by k tokens (Lemma 1), and hence at least τ occurrences survive. Otherwise (Lines 5-8), we employ the greedy algorithm (Algorithm 1) after retrieving the sorted list of the occurrences of R in S (Line 6). Recall that we maintain AVL trees that allow us to access the sorted list of occurrences for each node u considered by the DFS on $\mathsf{ST}(S)$. Finally, we check whether the remaining number of occurrences (after Algorithm 1 has been applied) suffices for R to be (τ, k)-resilient and return YES if that is the case and NO otherwise (Lines 7-8). Applying the greedy algorithm (Algorithm 1) in Line 6 costs an additional $\mathcal{O}(\tau + k)$ time.

Algorithm 2 Check if an aperiodic substring of S is (τ, k)-resilient, given access to its sorted list $\mathcal{O}$ of occurrences in S

```
1: function AperiodicResilient(i, j, τ, k, SA, l_u, r_u, O)
2:     |occ_S(S[i..j])| ← r_u − l_u + 1
3:     if |occ_S(S[i..j])| ≥ τ + 2k
4:         return YES
5:     else                          ▷ In this case, |occ_S(S[i..j])| = O(τ + k)
6:         if |occ_S(S[i..j])| − Greedy(O, k, j − i + 1) ≥ τ
7:             return YES
8:         return NO
```

We have thus shown the following result.

Lemma 4 (AperiodicResilient)**.** *Given the occurrences of an aperiodic substring P of a string S sorted, we can check whether P is (τ, k)-resilient in $\mathcal{O}(\tau + k)$ time.*

PeriodicResilient Function. Let P be a periodic substring of S. We say that an occurrence $S[i\,..\,j]$ of P *belongs* to a run $R = S[x\,..\,y]$ of S when $\mathsf{per}(R) = \mathsf{per}(S)$ and $x \le i \le j \le y$. By Lemma 2, we know that a single token can destroy occurrences of P that belong to at most two different runs of S. Hence, PeriodicResilient relies on an efficient non-trivial preprocessing of S to retrieve (at most) $\tau + 2k$ runs of S with period $\mathsf{per}(P)$ that have P as a substring. If the number of such runs is greater than or equal to $\tau + 2k$, then we cannot destroy sufficiently many occurrences of P, and thus P is (τ, k)-resilient; otherwise, we employ a more involved version of the greedy algorithm discussed before to check whether P is (τ, k)-resilient. We start by describing this preprocessing (and our practical implementation of it) before providing the full implementation of the PeriodicResilient function.

Preprocessing. Let a periodic substring P of S with fewer than $\tau + 2k$ runs to which its occurrences belong. The set $\mathcal{H}_P$ is defined as the set of all such runs. For any periodic substring Q that does not satisfy this condition, we set $\mathcal{H}_Q := \varnothing$. Recall that any run of S, being a fragment of S (see Definition 2), can be represented in $\mathcal{O}(1)$ space by its endpoints. We can thus represent $\mathcal{H}_P$ in $\mathcal{O}(\tau + k)$ space. The following lemma encapsulates a data structure that, given a periodic substring P of S, efficiently computes $\mathcal{H}_P$. The proof is in §V.D in [29].

Lemma 5 (ComputeHP(i, j, τ, k))**.** *For any string S of length n and integers $\tau, k \in [1, n]$, we can construct, in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(n)$ space, a data structure that, for any periodic substring P of S, computes the set $\mathcal{H}_P$ in $\mathcal{O}(\log n + \tau + k)$ time.*

The implementation of Lemma 5 is highly unlikely to be efficient in practice; see our previous discussion on Theorem 2. In practice, however, we can settle for a *simple* $\mathcal{O}((\tau + k) \log n)$-time algorithm as the total number of times we call the function PeriodicResilient in real-world strings is very small. In our experiments, this number was about $n/7000$ even for the smallest tested τ and k values; specifically, it was about $29 \cdot 10^3$ for $n = 2 \cdot 10^8$.

To realize our simple algorithm, we will view each run in $\mathcal{H}_P$ as a cluster of occurrences of P, and from now on we will be abusing the definition of $\mathcal{H}_P$ as follows: each run in $\mathcal{H}_P$ is represented by a pair (f, m), where f is the first occurrence of P in the run and $m \ge 1$ is the total number of occurrences of P in this run. Hence, if $(f, m) \in \mathcal{H}_P$ then there is a run in S with period $\mathsf{per}(P)$ such that the set of the starting positions of occurrences of P in S that lie within this run is $\{f + i \cdot \mathsf{per}(P) : i \in [0, m)\}$. The pseudocode of our simple algorithm for constructing $\mathcal{H}_P$ is in §V.E in [29] as Algorithm 3. It considers the occurrences of P in S in the left-to-right order and efficiently merges them into runs.

We give a high-level description of function PeriodicResilient; see §V.F in [29] for pseudocode. PeriodicResilient first computes $\mathcal{H}_P$ ($|\mathcal{H}_P| < \tau + 2k$) either by Lemma 5 or by our simple algorithm (Algorithm 3 in [29]). We start with two key concepts:

1) The maximum number of occurrences of P that can be destroyed using a single token, denoted by $\alpha := \lceil \frac{|P|}{\mathsf{per}(P)} \rceil$.
   Example 6. *Let $P =$ `aabaaba` and the run `aabaabaabaaba` that contains 3 occurrences of P. We have $\alpha = \lceil \frac{|P|}{\mathsf{per}(P)} \rceil = 3$ because all occurrences can be destroyed with a single token:* `aabaab#abaaba`.
2) The subset $\mathcal{H}_P^{(1,2)}$ of $\mathcal{H}_P$ defined as follows:

$$\mathcal{H}_P^{(1,2)} = \{(f, m) \in \mathcal{H}_P : m \bmod \alpha = 1 \text{ OR } 2\}.$$

These runs are treated specially by PeriodicResilient.

Function PeriodicResilient maintains two counters t and d, where t is the number of available tokens (initially $t := k$), and d is the maximum number of occurrences (initially $d := 0$) that may be destroyed using $k - t$ tokens. The algorithm has three main stages:

In the first stage, we only want to spend a token if it results in the destruction of α occurrences of P. We process the set of runs $\mathcal{H}_P$ in the left-to-right order and, for each run $(f, m) \in \mathcal{H}_P$, we compute $\lfloor m/\alpha \rfloor$, which equals the maximum number of tokens that we can use on this run such that each token destroys α (unique) occurrences of P. Specifically, while we have tokens available, we process (f, m) as follows: first, we use $y := \min\{t, \lfloor m/\alpha \rfloor\}$ tokens on this run, destroying a total of $y \cdot \alpha$ occurrences of P. The counters d and t are updated accordingly: $d := d + y \cdot \alpha$ and $t := t - y$.

1) If $m \bmod \alpha \geq 3$, $m \bmod \alpha$ occurrences of P are yet to be destroyed. Their number is inserted into a global max-heap $\mathcal{R}$ along with its counter that is incremented every time the number is encountered. These remaining occurrences are to be processed in the second stage.
2) If $1 \leq m \bmod \alpha \leq 2$, we set $\mathcal{H}_P^{(1,2)} := \mathcal{H}_P^{(1,2)} \cup \{(f, m)\}$ ($\mathcal{H}_P^{(1,2)}$ is initially empty). The remaining at most two occurrences of P are processed in the third stage.

After the first stage is finished, if $\mathsf{occ}_S(P) - d \geq \tau$ (that is, P is still τ-frequent) and $t > 0$ (that is, we still have tokens available), we proceed to the second stage. In this stage, we use the max-heap $\mathcal{R}$ in order to maximize the number of occurrences destroyed by each token. In particular, if the maximum element in $\mathcal{R}$ is r with counter c_r, we use $\min\{t, c_r\}$ tokens to destroy r occurrences of P with each of them. (Observe that by construction $r < \alpha$.)

We next proceed to the third stage if $\mathsf{occ}_S(P) - d \geq \tau$ and $t > 0$. In this stage, we process the runs of $\mathcal{H}_P^{(1,2)}$; each of them has one or two surviving occurrences of P. Due to Lemma 2, any position in S is contained in at most two runs of $\mathcal{H}_P$. We refer to a run $(f, m) \in \mathcal{H}_P^{(1,2)}$ as type one or type two depending on the value of $m \bmod \alpha$. We can assume that the runs in $\mathcal{H}_P^{(1,2)}$ are sorted with respect to starting position (due to how they are computed). We process the remaining occurrences of P, which all belong to runs in $\mathcal{H}_P^{(1,2)}$, using Algorithm 1: this algorithm first destroys pairs of overlapping occurrences and then non-overlapping singleton occurrences, similarly to the aperiodic case.

Finally, the function outputs whether P is (τ, k)-resilient.

Example 7. *Consider $\alpha = 10$ and that for all runs $(f, m) \in \mathcal{H}_P$, $m \in \{61, 60, 43, 23, 12, 7\}$. The max-heap $\mathcal{R}$ contains $[7, 1], [3, 2]$: 1 occurrence as the remainder from 7 and 2 occurrences as the remainder from 43 and 23. The remainders from 61 and 12 are processed by Algorithm 1.*

$\mathcal{H}_P$ is computed in $\mathcal{O}(\log n + \tau + k)$ time (Lemma 5) or $\mathcal{O}((\tau + k) \log n)$ time (Algorithm 3 in [29]). For each of the $\mathcal{O}(\tau + k)$ runs in $\mathcal{H}_P$, we spend $\mathcal{O}(\log n)$ time for a constant number of max-heap operations. Moreover, $\mathcal{O}((\tau + k) \log n)$ time is needed for retrieving and deleting elements from $\mathcal{R}$ and $\mathcal{O}(\tau + k)$ time analogously to Algorithm 1. We obtain the following result; see §V.F in [29] for the proof.

Lemma 6 (PeriodicResilient)**.** *After an $\mathcal{O}(n \log n)$-time preprocessing of a string S of length n, we can check whether any given periodic substring P of S is (τ, k)-resilient in $\mathcal{O}((\tau + k) \log n)$ time.*

Wrapping up. We obtain our main result by plugging Theorem 2 and Lemmas 4 and 6 to Lemma 3. However, to shave a $\log n$ factor, we replace the max-heap implementation in the proof of Lemma 6 with *global sorting* and *avoid recomputing the set $\mathcal{H}_P$* in each step of a binary search along an edge when PeriodicResilient is called; see §V.G in [29] for the proof.

Theorem 3. *For any string $S \in \Sigma^n$, with $|\Sigma| \geq 4$, and integers $\tau, k \in [1, n]$, the (τ, k)-Resilient Pattern Mining problem can be solved in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(n)$ space.*

V. Related Work

There has been much interest in resilience notions for many fundamental problems. One such notion is *sensitivity* (and average sensitivity) which, informally stated, measures the difference between the output of an algorithm after its input is perturbed by adding or removing *a single* element. This notion has been employed on graph [35], clustering [36], and learning [37] problems and several algorithms based on it have been proposed. There are also resilience notions specifically developed for clustering [38]–[40]. For example, *stability* in [38] is applied to graph clustering and requires the interesting instances to be only those for which small perturbations in the data do not change the optimal partition of the graph. *γ-Resilience* in [39] is applied to k-clustering [39] and, informally, requires an algorithm to return *similar* solutions on any two inputs that are *close*. Our resilience notion in Definition 6 differs from all existing ones in that it: (1) applies to substrings of a string; and (2) it considers a given number of possible changes to the string. Also, it differs from *differential privacy* [41] which imposes a bound on how much the probability of an algorithm's output can change when applied to any two datasets that differ by an element. Our work is somewhat related to works for mining frequent approximate sequential patterns [42]–[44] and fault-tolerant patterns [45]. The former works consider changes to the patterns while we consider changes to the input string. The latter work mines *dense* sets from set-valued data, while we mine substrings using a different resilience notion.

VI. Experimental Evaluation

Algorithms. As no existing algorithm can deal with RPM, we compared the suffix tree and enhanced suffix array implementations of the algorithm in Section IV, RPM-ST and RPM-ESA, to RPM-DP an implementation of the algorithm in Section III. Also, in our clustering case study, we compared a clustering algorithm [25] using (τ, k)-resilient substrings, mined by any of our algorithms for RPM, as features to: (I)

TABLE I: Dataset characteristics and values of parameters used. The default values are in bold within brackets.

Dataset	Length n	Alphabet Size $\|\Sigma\|$	Frequency τ	Number of Substitutions k
DNA	209,715,200	16	[10^1,10^6] (**10^4**)	[10^1,10^6] (**10^2**)
ENGLISH	209,671,447	225	[10^1,10^6] (**10^4**)	[10^1,10^6] (**10^2**)
PROTEINS	209,715,200	25	[10^1,10^6] (**10^4**)	[10^1,10^6] (**10^2**)
SOURCES	209,714,417	230	[10^1,10^6] (**10^4**)	[10^1,10^6] (**10^2**)
XML	209,715,200	96	[10^1,10^6] (**10^4**)	[10^1,10^6] (**10^2**)
BOOST	190, 898, 317	88	[4,50] (**4**)	[4,50] (**4**)
WIKI	629, 145, 600	194	[20,70] (**50**)	[20,70] (**50**)

the same clustering algorithm that uses τ-frequent substrings as features instead; and (II) a widely-used frequency-based clustering algorithm [24], which we will refer to as FC.

Datasets. We used 5 real-world, benchmark datasets from [46]; see Table I. DNA contains genomic data, ENGLISH contains English text, PROTEINS protein data, SOURCES source code, and XML bibliographic information on major computer science venues. We also considered two popular versioned datasets following [47], which were also used in [48]: BOOST, which consists of $10,000$ versions of the boost library in GitHub; and WIKI which consists of $9,271$ versions of Wikipedia's English Einstein page. Each version was treated as a different string. As expected by its time complexity, RPM-DP was not able to handle these datasets. Thus, we report results on prefixes of DNA (the results on prefixes of the other datasets were analogous). We also used two genomic datasets, EBOL and COR, from [49] in our clustering experiments. Each of these datasets is a collection of viral genomes with a ground truth clustering.

Setup. We report running time and peak memory usage. We also report two quality measures. The first is the ratio between the size of the set of the (τ, k)-resilient substrings and that of the set of the τ-frequent substrings. We refer to this ratio as RFR (for Resilient to Frequent Ratio). Since any (τ, k)-resilient substring is also τ-frequent for the same τ, RFR measures the ratio of the τ-frequent substrings that are also (τ, k)-resilient. The second measure is used for versioned datasets and is denoted by LR. For an arbitrary set Y of τ-frequent substrings (note, (τ, k)-resilient substrings are also τ-frequent) and a dataset version V, we define:

$$LR(Y, V) := |\{F \in Y : F \text{ is not } \tau\text{-frequent in } V\}|/|Y|.$$

All experiments run on an AMD EPYC 7282 CPU with 252 GB RAM. All methods were implemented in C++. *Our code is available at* https://github.com/PXBian/robust_mining.

RFR. Fig. 3 shows that RFR is at most 0.08 for $\tau = 10$ and $k = 100$; thus most 10-frequent substrings are *not* $(10, 100)$-resilient. RFR increases with τ since, due to the relatively small k value we used, more τ-frequent substrings remain τ-frequent (and thus are $(\tau, 100)$-resilient) when τ increases. Fig. 4 shows that RFR decreases (and approaches or becomes 0) as k increases. This is because, as τ is fixed, a larger k allows more letter substitutions and thus most or all substrings that are (τ, k)-resilient for one k are no longer resilient for a larger one. Thus, the sets of (τ, k)-resilient substrings and τ-frequent substrings for the same τ can be very different, which motivates the notion of (τ, k)-resilient substrings.

Resilience in Versioned Datasets. Fig. 5a (and 5b) shows that the set of τ-frequent substrings in version 1 of the BOOST (and WIKI dataset), denoted by FREQ, changes in the next version of the dataset in most cases. For example, in BOOST, the earliest version in which it changes is 2. On the other hand, the set of (τ, k)-resilient substrings for the same τ and k as FREQ, which is denoted by RESI, changes much later. For example, the earliest version in which RESI changes is $2,412$ in BOOST and 828 in WIKI.

Figs. 5c and 5d show that LR for the set of τ-frequent substrings is much larger compared to the LR for the set of (τ, k)-resilient substrings, for any version V (on average, by more than 2 orders of magnitude in BOOST and by more than 1 order of magnitude in WIKI). Thus, only a very small number of (τ, k)-resilient substrings stop being τ-frequent as more versions of the dataset are considered, and this happens after a very large number of versions (e.g., after version $2,000$ in BOOST). As expected, LR increases with V.

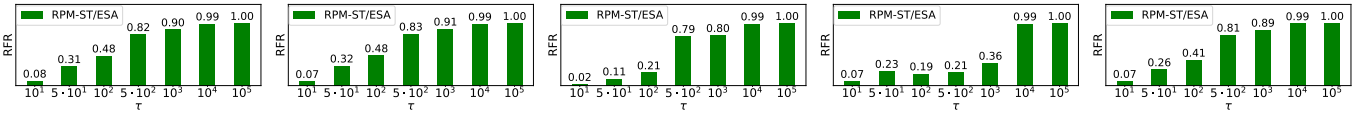
Thus, (τ, k)-resilient substrings remain truthful for longer, which is useful in applications, as mentioned in Introduction.

Running Time. Both RPM-ST and RPM-ESA have the same time complexity. As expected, both τ and k play an insignificant role in the running time of our algorithms; see Figs. 6 and 7. It is important to note that RPM-ESA is at least 9 and up to 12 times faster than RPM-ST in the experiment of Fig. 6 and an order of magnitude faster in the experiment of Fig. 7. This is because the enhanced suffix array takes less memory and supports faster cache-friendly operations compared to the suffix tree [23]. Fig. 8a shows that both algorithms scale close to linearly in n, as expected by their time complexity. RPM-ESA is faster (by 10.2 times on average) as expected.

Space. Fig. 8b shows that the space occupied by both RPM-ST and RPM-ESA increases linearly with n, as expected by their $\mathcal{O}(n)$ space complexity. The space is thus not affected by τ or k; see Figs. 9a and 9b for the XML dataset (the results for the other datasets are analogous and omitted). RPM-ESA occupies much less space than RPM-ST (e.g., 3.2 times less on average in the experiments of Figs. 9a and 9b), due to the use of the enhanced suffix array, which occupies less space than the suffix tree in practice [23].

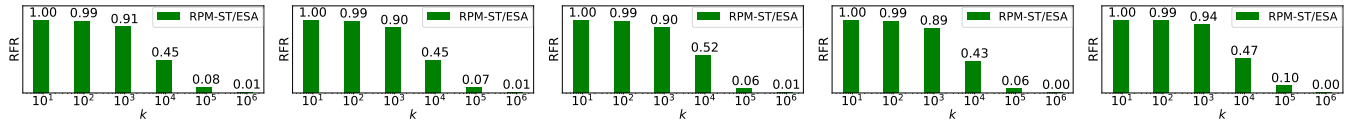
Comparison to RPM-DP. Figs. 10a and 10b show the impact of τ and k, respectively, for a prefix of DNA with $n = 10,000$. The running time of RPM-DP increases with both τ and k. This is because a larger τ requires more binary search iterations to locate the resilient substrings, and also slows down the DP checks due to the increased number of substring occurrences. The increase due to k is expected by the $\mathcal{O}(kn^3 \log n)$ time complexity. However, the running time of RPM-ST and RPM-ESA does not increase, similarly to Figs. 6 and 7. As expected by their time complexity, both RPM-ST and RPM-ESA substantially outperform RPM-DP (RPM-ST is more than 2 orders of magnitude faster and RPM-ESA is around 4 orders of magnitude faster). Figs. 11a and 11b show that both the running time and the memory usage for RPM-DP increase quadratically with n, while those for RPM-ST and RPM-ESA increase linearly with n; in this experiment, we have set $\tau = 50$ and $k = 10$.

Clustering Case Study. We show the benefit of clustering



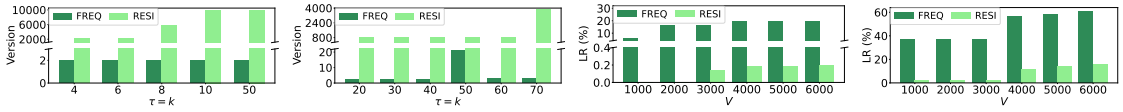
(a) DNA (b) ENGLISH (c) PROTEINS (d) SOURCES (e) XML

Fig. 3: RFR for varying τ. The RFR values are on the top of the bars.



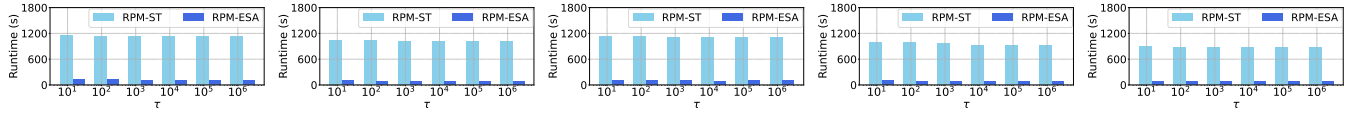
(a) DNA (b) ENGLISH (c) PROTEINS (d) SOURCES (e) XML

Fig. 4: RFR for varying k. The RFR values are on the top of the bars.



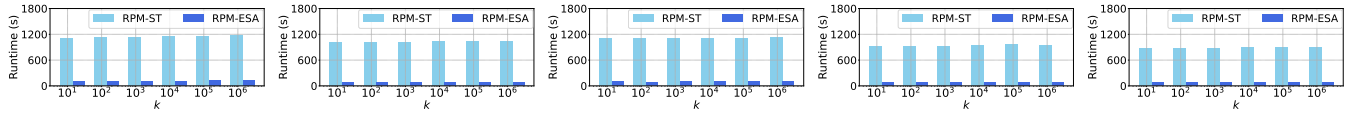
(a) BOOST (b) WIKI (c) BOOST (d) WIKI

Fig. 5: The earliest version in which at least one of the τ-frequent substrings in set FREQ or of the (τ, k)-resilient substrings in set RESI that is mined in Version 1 is no longer τ-frequent in (a) BOOST and (b) WIKI. (c, d) LR for varying V.



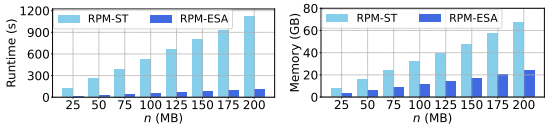
(a) DNA (b) ENGLISH (c) PROTEINS (d) SOURCES (e) XML

Fig. 6: Running time for varying τ.



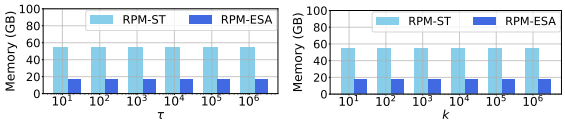
(a) DNA (b) ENGLISH (c) PROTEINS (d) SOURCES (e) XML

Fig. 7: Running time for varying k.



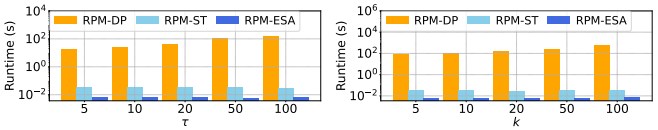
(a) Prefixes of DNA (b) Prefixes of DNA

Fig. 8: (a) Running time and (b) space for varying n.



(a) XML (b) XML

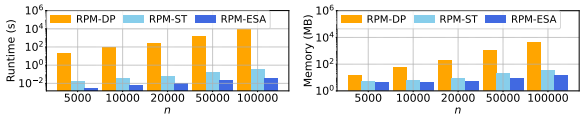
Fig. 9: Space for varying (a) τ and (b) k.



(a) DNA prefix with $n = 10,000$ (b) DNA prefix with $n = 10,000$

Fig. 10: Running time of RPM-DP, RPM-ST and RPM-ESA for varying (a) τ and (b) k.

based on (τ, k)-resilient substrings following the methodology of [24], [25]. We perform pattern-based clustering of a dataset comprised of multiple strings: the top-K most frequent (τ, k)-resilient substrings are the features, the frequency of a substring in a string is the feature value, and the k-means algorithm is used to cluster the feature matrix. We set $\mathrm{k} = 5$ in k-means, as the ground truth clustering has 5 clusters.



(a) Prefixes of DNA (b) Prefixes of DNA

Fig. 11: (a) Running time and (b) space of RPM-DP, RPM-ST and RPM-ESA for varying n.

We evaluated how similar is the clustering obtained from the clustering algorithm based on (τ, k)-resilient substrings to the ground truth; see Table IIa. We used two well-known measures: Normalized Mutual Information (NMI) and Rand Index (RI) (see §VI in [29] for details), which take values in $[0, 1]$ (higher values are preferred and 1 means same clustering as the ground truth clustering). The results are the maximum and average value of these measures over all combinations of τ, k, and K such that $\tau = k \in \{60, 70, 80, 90, 100\}$ and $K \in$

TABLE II: (a) Clustering with (τ, k)-resilient substrings vs ground truth clustering. (b, c) Percentage increase $\frac{M_R - M_C}{M_C} \cdot 100\%$, where, for a clustering quality measure (max/avg NMI and max/avg ARI), M_R is the measure for our approach and M_C is the same measure for (b) the clustering algorithm using τ-frequent substrings and (c) the FC clustering algorithm in [24].

(a)

Dataset	Max NMI	Avg NMI	Max RI	Avg RI
EBOL	1	0.83	1	0.87
COR	0.86	0.80	0.96	0.92

(b)

Dataset	Max NMI	Avg NMI	Max RI	Avg RI
EBOL	22.6%	1.5%	25.3%	9.2%
COR	8.5%	0.1%	2.9%	0.2%

(c)

Dataset	Max NMI	Avg NMI	Max RI	Avg RI
EBOL	22.6%	1.3%	25.3%	9.0%
COR	10.2%	1.4%	11.8%	9.4%

$\{50, 150, 250\}$. The clusters constructed by our approach are very close or the same as those in the ground truth clustering. Using the same measures and setting, we also show that our approach is better in terms of finding a clustering close to the ground truth than: (I) the same clustering algorithm based on τ-frequent substrings instead of (τ, k)-resilient substrings (Table IIb); and (II) the FC algorithm (Table IIc).

References

[1] J. Fischer, V. Heun, and S. Kramer, "Fast frequent string mining using suffix arrays," in *ICDM*, 2005, pp. 609–612.

[2] ——, "Optimal string mining under frequency constraints," in *PKDD*, 2006, pp. 139–150.

[3] J. Dhaliwal, S. J. Puglisi, and A. Turpin, "Practical efficient string mining," *TKDE*, vol. 24, no. 4, pp. 735–744, 2012.

[4] C. T. Clelland, V. Risca, and C. Bancroft, "Hiding messages in DNA microdots," *Nature*, vol. 399, no. 6736, pp. 533–534, 1999.

[5] G. Navarro, "Indexing highly repetitive string collections, Part I: Repetitiveness measures," *ACM Comput. Surv.*, vol. 54, no. 2, pp. 29:1–29:31, 2021.

[6] G. V. Kryukov, S. Schmidt, and S. Sunyaev, "Small fitness effect of mutations in highly conserved non-coding regions," *Human molecular genetics*, vol. 14, no. 15, pp. 2221–2229, 2005.

[7] The Computational Pan-Genomics Consortium, "Computational pan-genomics: status, promises and challenges," *Briefings Bioinform.*, vol. 19, no. 1, pp. 118–135, 2018.

[8] C. Boucher, T. Gagie, I. Tomohiro, D. Köppl, B. Langmead, G. Manzini, G. Navarro, A. Pacheco, and M. Rossi, "PHONI: Streamed matching statistics with multi-genome references," in *DCC*, 2021, pp. 193–202.

[9] NCBI, "NCBI genome datasets," 2025, https://www.ncbi.nlm.nih.gov/datasets/genome/.

[10] R. Buyya, R. N. Calheiros, and A. V. Dastjerdi, *Big data: principles and paradigms*. Morgan Kaufmann, 2016.

[11] M. Henzinger, "Finding near-duplicate web pages: a large-scale evaluation of algorithms," in *SIGIR*, 2006, pp. 284–291.

[12] K. Tao, F. Abel, C. Hauff, G. Houben, and U. Gadiraju, "Groundhog day: near-duplicate detection on Twitter," in *WWW*, 2013, pp. 1273–1284.

[13] C. Kapser and M. W. Godfrey, "Improved tool support for the investigation of duplication in software," in *ICSM*, 2005, pp. 305–314.

[14] P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Łącki, and P. Sankowski, "Optimal dynamic strings," in *SODA*, 2018, pp. 1509–1528.

[15] A. Liu, K. Zheng, L. Li, G. Liu, L. Zhao, and X. Zhou, "Efficient secure similarity computation on encrypted trajectory data," in *ICDE*, 2015, pp. 66–77.

[16] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann, 2011.

[17] J. Tan, S. Heinecke, Z. Liu, Y. Chen, Y. Zhang, and H. Wang, "Towards more robust and accurate sequential recommendation with cascade-guided adversarial training," in *SDM*, 2024, pp. 743–751.

[18] G. Bernardini, H. Chen, A. Conte, R. Grossi, G. Loukides, N. Pisanti, S. P. Pissis, G. Rosone, and M. Sweering, "Combinatorial algorithms for string sanitization," *TKDD*, vol. 15, no. 1, pp. 8:1–8:34, 2021.

[19] P. A. Evans, A. D. Smith, and H. T. Wareham, "On the complexity of finding common approximate substrings," *Theor. Comput. Sci.*, vol. 306, no. 1-3, pp. 407–430, 2003.

[20] N. Pisanti, A. M. Carvalho, L. Marsan, and M. Sagot, "RISOTTO: fast extraction of motifs with mismatches," in *LATIN*, 2006, pp. 757–768.

[21] M. Chrobak, M. J. Golin, T. W. Lam, and D. Nogneng, "Scheduling with gaps: new models and algorithms," *J. Sched.*, vol. 24, no. 4, pp. 381–403, 2021.

[22] M. Farach, "Optimal suffix tree construction with large alphabets," in *FOCS*, 1997, pp. 137–143.

[23] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *J. Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[24] S. Vinga and J. S. Almeida, "Alignment-free sequence comparison—a review," *Bioinformatics*, vol. 19, no. 4, pp. 513–523, 2003.

[25] Y. Wu, X. Zhao, Y. Li, L. Guo, X. Zhu, P. Fournier-Viger, and X. Wu, "OPR-Miner: Order-preserving rule mining for time series," *TKDE*, pp. 11 722–11 735, 2023.

[26] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *CPM*, 2001, pp. 181–192.

[27] U. Manber and E. W. Myers, "Suffix arrays: A new method for on-line string searches," *SICOMP*, vol. 22, no. 5, pp. 935–948, 1993.

[28] R. Lundstrom, S. Tavare, and R. H. Ward, "Estimating substitution rates from molecular data using the coalescent," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 89, no. 13, pp. 5961–5965, 1992.

[29] P. Bian *et al.*, "Supplemental Material: Resilient Pattern Mining," 2025, https://github.com/PXBian/robust_mining/.

[30] D. E. Knuth, J. H. M. Jr., and V. R. Pratt, "Fast pattern matching in strings," *SICOMP*, vol. 6, no. 2, pp. 323–350, 1977.

[31] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen, "Internal pattern matching queries in a text and applications," *SICOMP*, vol. 53, no. 5, pp. 1524–1577, 2024.

[32] J. Ellert and J. Fischer, "Linear time runs over general ordered alphabets," in *ICALP*, 2021, pp. 63:1–63:16.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[34] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta, "The "runs" theorem," *SICOMP*, vol. 46, no. 5, pp. 1501–1514, 2017.

[35] N. Varma and Y. Yoshida, "Average sensitivity of graph algorithms," *SICOMP*, vol. 52, no. 4, pp. 1039–1081, 2023.

[36] Y. Yoshida and S. Ito, "Average sensitivity of Euclidean k-clustering," in *NeurIPS*, 2022, pp. 32 487–32 498.

[37] S. Hara and Y. Yoshida, "Average sensitivity of decision tree learning," in *ICLR*, 2023.

[38] Y. Bilu and N. Linial, "Are stable instances easy?" *Comb. Probab. Comput.*, vol. 21, no. 5, pp. 643–660, 2012.

[39] S. Ahmadian, M. Bateni, H. Esfandiari, S. Lattanzi, M. Monemizadeh, and A. Norouzi-Fard, "Resilient k-clustering," in *KDD*, 2024, pp. 29–38.

[40] P. Awasthi, A. Blum, and O. Sheffet, "Center-based clustering under perturbation stability," *IPL*, vol. 112, no. 1-2, pp. 49–54, 2012.

[41] C. Dwork, F. McSherry, K. Nissim, and A. D. Smith, "Calibrating noise to sensitivity in private data analysis," in *TCC*, 2006, pp. 265–284.

[42] J. Shang, J. Peng, and J. Han, "MACFP: maximal approximate consecutive frequent pattern mining under edit distance," in *SDM*, 2016, pp. 558–566.

[43] S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich, "REPuter: the manifold applications of repeat analysis on a genomic scale," *NAR*, vol. 29, no. 22, pp. 4633–4642, 2001.

[44] F. Zhu, X. Yan, J. Han, and P. S. Yu, "Efficient discovery of frequent approximate sequential patterns," in *ICDM*, 2007, pp. 751–756.

[45] A. K. Poernomo and V. Gopalkrishnan, "Towards efficient mining of proportional fault-tolerant frequent itemsets," in *KDD*, 2009, pp. 697–706.

[46] "The Pizza and Chili Corpus," 2025, https://pizzachili.dcc.uchile.cl/texts.html.

[47] "RepCorpus," 2025, https://pizzachili.dcc.uchile.cl/repcorpus/real/.

[48] T. Gagie, G. Navarro, and N. Prezza, "Fully functional suffix trees and optimal text searching in BWT-runs bounded space," *J. ACM*, vol. 67, no. 1, pp. 2:1–2:54, 2020.

[49] Y. Li, L. He, R. L. He, and S. S.-T. Yau, "A novel fast vector method for genetic sequence comparison," *Sci Rep.*, vol. 7, no. 12226, 2017.