Supplemental Material: Resilient Pattern Mining

Anonymous
Anonymous

I. Substituting Letters from the Alphabet

To show that we can effectively destroy the occurrences of any substring P of $S \in \Sigma^n$ by using only letters from Σ, with $|\Sigma| \geq 4$, it suffices to show that we can replace substitutions with #s with substitutions with letters from Σ in a way that does not create any new occurrences of P. Before proving this (Lemma 11), we prove a few auxiliary lemmas.

Lemma 7. *Let $T = U \cdot P \cdot V$ be a string over an alphabet Σ, such that $|\Sigma| \geq 3$, P is a non-empty string occurring only once in T, and $|U|, |V| < |P|$. There exists a string P' over Σ, such that $d_H(P, P') = 1$ and $T' = U \cdot P' \cdot V$ has no occurrence of P.*

Proof. Towards a contradiction, suppose that P occurs in T', for all strings $T' = U \cdot P' \cdot V$ such that $d_H(P, P') = 1$. Let $m = |P|$. Since $|\Sigma| \geq 3$ and we can hence choose to substitute with at least two different letters in each of the m positions of T in $[|U|, |UP|)$, there are at least $2m$ different strings T' of the form $U \cdot P' \cdot V$ such that $d_H(P, P') = 1$. We have $|T'| - m \leq 2m - 2$ possible starting positions for P in a string such a string T'. Since $2m - 2 < 2m$, by the pigeonhole principle, there are two such strings, say T'_1 and T'_2, with an occurrence of P starting at the same position, say i. This yields a contradiction, because the occurrence of P must span the substitution in each of T'_1 and T'_2 while either the position of the substitution or the substituted letter is different. □

Lemma 8. *Let $T = U \cdot V \cdot P$ be a string over an alphabet Σ, such that $|\Sigma| \geq 3$, P is an aperiodic string occurring only as a prefix and as a suffix of $V \cdot P$ in T, $|V| \in [1, |P|)$, and $|U| < |P|$. There exists a string V' over Σ, such that $d_H(V, V') = 1$ and $T' = U \cdot V' \cdot P$ has one occurrence of P.*

Proof. We will first prove that in any string $T' = U \cdot V' \cdot P$ such that V' is over Σ and $d_H(V, V') = 1$, P does not have any occurrence starting at a position in $[|U|, |U| + |V|)$. Let us fix any two such strings V' and T'. Let $X = V \cdot P$ and consider a substitution at position $j \in [0, |V|)$ of X transforming X to $X' = V' \cdot P$, such that $X'[j] = V'[j] = b$ and $X[j] = V[j] = P[j] = a$, for some letters $a \neq b$ from Σ. Further suppose that P has an occurrence in X' starting at some position $i \in [1, j]$. Due to the overlaps of the occurrences of P in each of X and X', P has periods $|V|$ and $|V| - i$. Since P occurs at position $i \in [1, |V|)$ of X' and has a period $|V|$, we have $X'[j] = X[j - i] = X[j - i + |V|] = b$. Further, since $X[j] = a$ and P also has a period $|V| - i$, we have $X[j + (|V| - i)] = X[j - i + |V|] = a$; a contradiction.

We now assume that each string $T' = U \cdot V' \cdot P$, where $V' \in \Sigma^*$ and $d_H(V, V') = 1$ has an occurrence of P starting at some position in $[0, |U|)$. Let $m = |P|$. Due to our assumption that P is aperiodic, we have $\mathsf{per}(P) > m/2$ and hence $|V| > m/2$. Since $|\Sigma| \geq 3$, we can choose to substitute with each of at least two different letters in each of the $|V|$ positions of T in $[|U|, |UV|)$ and we can thus generate at least $2|V|$ different strings T' of the form $U \cdot V' \cdot P$ such that $d_H(V, V') = 1$. However, we have $|U| < m$ possible starting positions for P in T'. Since $2|V| > m > |U|$, we have two strings T'_1 and T'_2, with an occurrence of P starting at the same position, say i. This yields a contradiction, because the occurrence of P starting at position i in each of the strings T'_1 and T'_2 must span the respective substitution while either the position of the substitution or the substituted letter is different. □

Lemma 9. *Let $T = U \cdot X \cdot V$ be a string over an alphabet Σ, such that $|\Sigma| \geq 4$, X has a substring P occurring only as a prefix and as a suffix of X, P occurs $h \geq 2$ times in T, $|X| < 2|P|$, and $|U|, |V| < |P|$. There exists a string X', such that $d_H(X, X') = 1$ and P occurs $h - 2$ times in $T' = U \cdot X' \cdot V$.*

Proof. Let $m := |P|$. Further, let $i := |UX| - m$ be the starting position of the second occurrence of P in $T[|U| \,..\, |UX| - 1]$, and $j := |U| + m - 1$ be the ending position of the first occurrence of P in $T[|U| \,..\, |UX| - 1]$. Then, the overlap of the two occurrences of P that are fully contained in $X = T[|U| \,..\, |UX| - 1]$ is precisely $F := T[i \,..\, j]$.

Fact 2. *For any position $x \in [i \,..\, j]$ of T, there exist $b, c \in \Sigma$ such that the string T' obtained via a substitution at position x of T with either of b or c satisfies $\mathsf{occ}_{T'}(P) \cap [|U|, i) = \emptyset$.*

Proof. Consider a string $T' = T[0 \,..\, i-1] \cdot F' \cdot T[j+1 \,..\, |T|-1]$ with $d_H(F, F') = 1$ and $T[x] = a \neq T'[x] \in \Sigma$. Suppose that P has an occurrence at a position $y \in [|U|, i)$ of T'. Then, the assumed occurrence of P in T' and the occurrences of P in X imply that:

- $T'[|U| \,..\, |U| + x - y - 1] = P[0 \,..\, x - y - 1] = T'[y \,..\, x - 1]$;
- $T'[i + x - y + 1 \,..\, |UX| - 1] = P[x - y + 1 \,..\, m - 1] = T'[x + 1 \,..\, y + m - 1]$.

See Fig. 1 for an illustration.

Without loss of generality, let us assume that $x - y \leq m - (x - y + 1)$, that is, that the string in the second bullet point is longer than the string in the first bullet point (this case is depicted in Fig. 1). Let $Y = T'[x + 1 \,..\, y + m - 1] = P[x - y + 1 \,..\, m - 1]$. The equalities above also imply that Y

is a suffix of $T'[x+1\,..\,|UX|-1] = P[x-i+1\,..\,m-1]$. Therefore, Y is both a prefix and a suffix of

$$Z := T[x+1\,..\,|UX|-1] = P[x-i+1\,..\,m-1],$$

implying that Z has a period $i-y$.

Now, $T'[x\,..\,y+m-1]$ is a suffix of P and also a suffix of Z since $y+m-x \le i-1+m-x = |Z|$. Then, due to $x \ge i$ and our assumption that $x-y \le m-(x-y+1)$, we have $i-y \le x-y \le y+m-x-1$ and hence $T'[x\,..\,y+m-1]$ has a non-trivial period $i-y$. This is only possible if $T'[x] = T[x+(i-y)]$. Thus, since $\Sigma \ge 4$, the claim follows. □

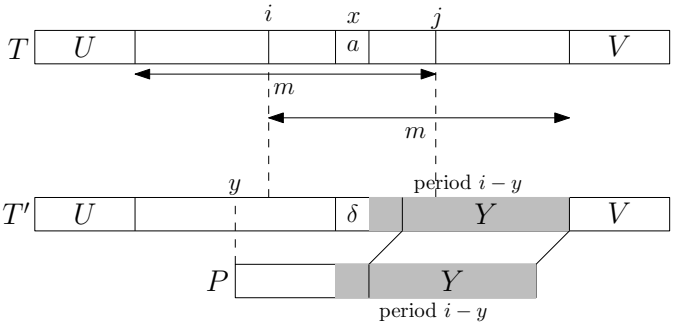

Fig. 1: The setting in Fact 2, where fragments equal to Z (with period $i-y$) are shaded. In this setting, P can only occur at position y of T' if letter a at position x is substituted with letter $\delta = T[x+(i-y)]$.

Fact 3. *There exists a string $T' = T[0\,..\,i-1]\cdot F'\cdot T[j+1\,..\,|T|-1]$ such that $d_H(F,F') = 1$, $\mathsf{occ}_{T'}(P) \cap [0,|UX|) \subset \mathsf{occ}_T(P) \cap [0,|UX|)$.*

Proof. Towards a contradiction, suppose that this is not the case. Since $|\Sigma| \ge 4$, due to Fact 2 we can choose at least two different letters to substitute each of the $|F|$ positions of T in $[i,j]$ without creating a new occurrence of P starting at a position in $[|U|,i)$. We can generate at least $2|F|$ different strings T' of the form specified in the claim's statement without any occurrence of P starting at a position in $[|U|,i)$.

In each of the constructed strings T', each new occurrence of P must then start at a position in $[|U|-|F|,|U|)\cup(i,j]$ of T' and span the substitution in F'. By the pigeonhole principle, we have two such strings T'_1 and T'_2, with an occurrence of P starting at the same position, say y, in $[|U|-|F|,|U|)\cup(i,j]$, since the total size of this set is $2|F|-1$. This yields a contradiction because the occurrences of P at positions y of T'_1 and T'_2 must span the substitution in each of T'_1 and T'_2 while either the position of the substitution or the substituted letter is different. □

To complete the proof, it suffices to note that a string T', as specified in Fact 3, has $h-2$ occurrences of P since the substitution yielding string T' from T destroys the two occurrences of P that are fully contained in $T[|U|\,..\,|UX|-1]$, it does not destroy any other occurrence due to Lemma 1, and it does not create any occurrence of P due to Fact 3. □

Lemma 10. *Let T and T' be two strings in $(\Sigma\cup\{\#\})^*$; where Σ is an alphabet, $|\Sigma| \ge 3$ and $\# \notin \Sigma$, such that $d_H(T,T') = 1$, $T[i] \ne T'[i] = \#$ for some $i \in [0,|T|)$, $|\mathsf{occ}_T(P)| - |\mathsf{occ}_{T'}(P)| = y > 0$ for a periodic string P of length m, and, further, $|\mathsf{occ}_{T'[i-m+1\,..\,i+m-1]}(\#)| = 1$. We can substitute $T'[i]$ with a letter from Σ without increasing the frequency of P.*

Proof. We perform the substitution of $\#$ in T' with any letter in $\Sigma\setminus\{T[i-\mathsf{per}(P)], T[i+\mathsf{per}(P)]\}$; let us denote the obtained string by T''. This is possible because $|\Sigma| \ge 3$. Any element $j \in \mathsf{occ}_{T''}(P)\setminus\mathsf{occ}_{T'}(P)$ must contain position i and at least one of $i+\mathsf{per}(P)$ and $i-\mathsf{per}(P)$ (since $\mathsf{per}(P) \le m/2$), and hence the period of $T''[j\,..\,j+m-1]$ does not divide $\mathsf{per}(P)$; a contradiction. □

We are now ready to put these lemmas together.

Lemma 11. *Let S be a string of length n over Σ, with $|\Sigma| \ge 4$, and P be a substring of S. If we can reduce the frequency of P in S by $y \in \mathbb{Z}_+$ using k substitutions with a letter $\# \notin \Sigma$, then we can also reduce the frequency of P in S by at least y using k substitutions with letters from Σ.*

Proof. Let $m = |P|$ and $\kappa \le k$ be the *smallest* integer such that we can use κ letters $\# \notin \Sigma$ to destroy y occurrences of P in S. Then, there exist $\kappa \le k$ positions in S where we can place $\#$s such that the distance between any pair of $\#$s is at least m (and hence no two of them touch the same occurrence of P). Consider the collection $\mathcal{U}$ of all the sets of positions that satisfy the above property. For a set $U \in \mathcal{U}$, let B_U be the bit-string of length n such that $B_U[i]$ is set if and only if $i \in U$. Let $Y := \arg\min_{U\in\mathcal{U}} B_U = \{i_1,\ldots,i_\kappa\}$, where $i_1 < i_2 < \cdots < i_\kappa$, and denote the set of starting positions of occurrences of P in S that the letter $\#$ at position i_j destroys by $I_j := \mathsf{occ}_S(P) \cap (i_j-|P|, i_j]$.

We perform κ substitutions to S with letters from Σ, such that the j-the substitution destroys the occurrences of P at all positions in I_j (and no others), while it does not create any new occurrences of P. By the minimality of κ and B_Y, for each i_j, if P is aperiodic, we do not try to destroy a single occurrence of P that overlaps with an occurrence of P in both sides. Hence, for each position i_j, we want to destroy one of the following:

- one occurrence of P that does not overlap with any other occurrence of P (this can be done due to Lemma 7); or
- if P is aperiodic, one occurrence of P that overlaps with another occurrence of P only on one of its two sides (this can be done due to Lemma 8); or
- two overlapping occurrences of P (this can be done due to Lemma 9); or
- if P is periodic, some number of overlapping occurrences of P (this can be done due to Lemma 10).

For the remaining $k-\kappa$ substitutions that we can perform (if any), we substitute the letters of S from left to right by any letter other than $P[0]$ – we do not decrease the budget if we substitute a letter at a position in Y. This guarantees that

these remaining substitutions do not create a new occurrence of P starting within the updated prefix. □

II. Proof of Fact 1

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.*

Proof. We can prove Fact 1 by contradiction. Assume Z is (τ, k)-resilient and either X or Y is not (τ, k)-resilient. Assume that X is not (τ, k)-resilient (the proof for Y is analogous). Then there exists a string S' that is obtained from S after substitution k letters with $\#$ such that X is not τ-frequent in S'. However, Z is by definition τ-resilient in S' and each occurrence of Z in S' implies at least one occurrence of X in S' because X is a prefix of Z, this is a contradiction. □

III. Listing the (τ, k)-Resilient Substrings

Assume that we have access to $\mathsf{SA}(S)$, the LCP array of S, and the OUTPUT array. We traverse the $\mathsf{SA}(S)$ from top to bottom and output the set of (τ, k)-resilient substrings explicitly in $\mathcal{O}(n)$ time plus time proportional to their total number. In particular, if a (τ, k)-resilient substring occurs in multiple positions of S, the latter positions will form a *range* (sub-array) of $\mathsf{SA}(S)$. Thus we can avoid reporting the substring again and again by traversing $\mathsf{SA}(S)$ from top to bottom and checking the corresponding LCP values.

IV. The DP-based Algorithm

We represent $\mathsf{occ}_S(Z)$ as a set of intervals $\mathcal{I} = \{(i_0, i_0 + |Z| - 1, (i_1, i_1 + |Z| - 1), \ldots, (i_{f-1}, i_{f-1} + |Z| - 1)\}$. We assume without a loss of generality that $i_0 < i_1 < \ldots < i_{f-1}$. The dynamic programming algorithm [1] underlying Theorem 1 has three steps:

1) We compute a matrix W of size $f \times f$, such that for every $i_a, i_b \in occ_S(Z)$, the element $W[a][b]$ stores the number of intervals that do not contain position $i_a + |Z| - 1$ but contain position $i_b + |Z| - 1$, where $i_a < i_b$.
2) We compute a matrix T of size $f \times k$ as follows. The first column of T (i.e., the element $T[b][0]$, for all $b = 0, 1 \ldots, f - 1$) is initialized with the number of intervals that contain $i_b + |Z| - 1$. The remaining of the first row of T (i.e., the element $T[0][h]$, for all $h = 1, \ldots, k - 1$) is initialized with the number of intervals that contain $i_0 + |Z| - 1$. Every other element $T[b][h]$ is then computed using $\max_{a<b}\{T[a][h-1] + W[a][b]\}$.
3) We output $\max_b\{T[b][k-1]\}$, as the maximum number of intervals that can be hit by a subset of k points.

V. Details of the Main Algorithm

A. Simulating $\mathsf{ST}(S)$

Note that instead of using $\mathsf{ST}(S)$, we can apply the algorithm of Abouelhoda et al. [2, Algorithm 4.4], which simulates a bottom-up traversal of $\mathsf{ST}(S)$ using $\mathsf{SA}(S)$ and the LCP array of S in $\mathcal{O}(n)$ time. This data structure is known as the *enhanced suffix array* (ESA). Simulating the $\mathsf{ST}(S)$ using ESA is usually faster, due to cache-friendly operations, and more space efficient in practice.

B. Proofs of Combinatorial Lemmas

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 $z > x + |R|$.*

Proof. If two occurrences of R in S started $p \leq |R|/2$ positions apart, then $\mathsf{per}(R) \leq p \leq |R|/2$ and hence R would be periodic, a contradiction. Thus, we have $z > y + |R|/2 > x + |R|$. □

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 $k > i'$.*

Proof. Let the common period of the considered runs be p. First, note that $j' \geq j + 2p - 1$ since, for any run R, we have $|R| \geq 2\mathsf{per}(R)$. Now, two runs with the same period p cannot overlap by $x \geq p$ positions; if this were the case, the portion of S spanned by these runs would have period p and this would contradict the maximality of at least one of the considered runs. Due to the overlap constraint, we have $k > j' - p + 1$ and $j > i' - p + 1$. Putting everything together, we obtain $k > j' - p + 1 \geq (j + 2p - 1) - p + 1 \geq (i' - p + 1) + p = i'$. □

C. AVL Trees for Maintaining Sorted Lists

The following lemmas related to the merge operations of two AVL trees are utilized:

Lemma 12 ([3])**.** *Two AVL trees of size at most n_1 and n_2 can be merged in time $\mathcal{O}(\log \binom{n_1+n_2}{n_1})$.*

Using the *smaller-half trick*, the sum over all nodes of an arbitrary binary tree of the terms stated in Lemma 12 is given as follows:

Lemma 13 ([3])**.** *Let T be an arbitrary binary tree with n leaves. The sum over all internal nodes v in T of terms $\log \binom{n_1+n_2}{n_1}$, where n_1 and n_2 are the numbers of leaves in the subtrees rooted at the two children of v and $n_1 \leq n_2$, is $\mathcal{O}(n \log n)$.*

The suffix tree $\mathsf{ST}(S)$ can be converted into a binary tree via the addition of dummy nodes in $\mathcal{O}(n)$ time and space, therefore, maintaining the sorted list of occurrences of $\mathsf{str}(v)$ for each node v during a DFS traversal can be done in $\mathcal{O}(n \log n)$ time.

D. Proof of Lemma 5

Lemma 14 encapsulates a restricted variant of priority search trees [4], which we use in the proof of Lemma 5.

Lemma 14 ([4])**.** *Given n points in $[1, n] \times [1, n]$, we can construct in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space, a data structure that supports the following type of queries (known as 3-sided range reporting queries) in $\mathcal{O}(\log n + |\mathsf{output}|)$ time: given $a, b, c, d \in \mathbb{Z} \cup \{-\infty, \infty\}$, such that $\{a, b, c, d\} \cap \{-\infty, \infty\} \neq \emptyset$, report all the points in $(a, b) \times (c, d)$.*

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.*

Proof. We first describe the preprocessing and then the query.

a) Preprocessing.

We compute all runs of S in $\mathcal{O}(n)$ time using the algorithm of [5]. We then group the runs by Lyndon root in $\mathcal{O}(n)$ time [6, Theorem 6]. Each group is further divided into subgroups according to the number of Lyndon root occurrences in the runs; these subgroups are stored in descending order with respect to this number. Creating these subgroups and sorting them can be done in $\mathcal{O}(n)$ time using radix sort since the numbers to be sorted are from $[1, n]$. A subgroup containing runs with Lyndon root V such that V occurs t times in each run is labeled with (V, t). For each subgroup (V, t), we create a two-dimensional grid $\mathcal{G}_{(V,t)}$ where we insert, for each run $V_1 \cdot V^t \cdot V_2$ in (V, t), the point $(|V_1|, |V_2|)$ — note that we may have a multiset of points. Further, for each subgroup (V, t), after reducing to rank space, we insert each point of $\mathcal{G}_{(V,t)}$ into a priority search tree $\mathcal{T}_{(V,t)}$ (cf. Lemma 14). Since we have no more than n runs in S in total [7], we can construct all such trees [4] in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(n)$ space. Each tree supports 3-sided range reporting queries in $\mathcal{O}(\log n + f)$ time, where f is the number of points in the output [4]. We also preprocess S, in $\mathcal{O}(n)$ time for constant-time periodic extension queries (cf. Theorem 2) and constant-time minimal rotation queries [8] (that return the lexicographically smallest rotation of an arbitrary fragment of S). The preprocessing time is $\mathcal{O}(n \log n)$ and the space usage is $\mathcal{O}(n)$.

b) Querying.

Suppose that we are given a periodic substring $P = V_1 \cdot V^h \cdot V_2$ of S, where V is the lexicographically smallest rotation of $P[0\,..\,\mathsf{per}(P) - 1]$ and $|V_1|, |V_2| \leq \mathsf{per}(P)$ (see Definition 4). The period of P of S can be computed in $\mathcal{O}(1)$ time using a periodic extension query. The Lyndon root V of P can then be computed in $\mathcal{O}(1)$ time using a minimal rotation query.

Recall that we would like to retrieve $\mathcal{H}_P$: a set of (less than $\tau + 2k$) runs such that each contains some occurrence of P and has period $\mathsf{per}(P)$. If, during the course of the query procedure, we compute $\tau + 2k$ such runs, we terminate the algorithm and return $\emptyset$. In the remainder of the proof, we assume that this is not the case. We first retrieve all runs in subgroups (V, t) with $t \geq h + 2$. Each of these runs contains at least one occurrence of P.

Example 8. *Let $P = \mathtt{bc}(\mathtt{abc})^2\mathtt{ab}$ with Lyndon root $V = \mathtt{abc}$ and $h = 2$. The run $(\mathtt{abc})^3$ does not have any occurrence of P. The run $(\mathtt{abc})^4$ has an occurrence of P at position 1.*

We retrieve all f runs $U_1 \cdot V^{h+1} \cdot U_2$ from $(V, h+1)$ such that either $|V_1| \leq |U_1|$ or $|V_2| \leq |U_2|$. These runs are retrieved by computing the points of $\mathcal{G}_{(V,h+1)}$ in the union of $[|V_1|, \infty) \times (-\infty, \infty)$ and $(-\infty, |V_1| - 1] \times [|V_2|, \infty)$; these points can be reported using $\mathcal{T}_{(V,h+1)}$ in time $\mathcal{O}(\log n + f)$.

Finally, we retrieve all f' runs from (V, h) such that $|V_1| \leq |U_1|$ and $|V_2| \leq |U_2|$. We employ a range reporting query using $\mathcal{T}_{(V,h)}$ to compute the points that lie in the quarterplane $[|V_1|, \infty) \times [|V_2|, \infty)$ in $\mathcal{G}_{(V,h)}$. This takes $\mathcal{O}(\log n + f')$ time.

The total query time is $\mathcal{O}(\log n + \tau + k)$. □

E. Practical Alternative to Lemma 5

Algorithm 3 presents our practical implementation of the preprocessing. During the bottom-up traversal of the suffix tree $\mathsf{ST}(S)$ and at node u such that $\mathsf{str}(u) = P$ and P is periodic, if $P = S[i\,..\,j]$ then Algorithm 3 takes as input the set $\mathcal{O}$ of occurrences of P in S sorted from the left to the right. The set $\mathcal{O}$ is then partitioned in inclusion-maximal sequences of occurrences such that the occurrences in each sequence are at distance $p = \mathsf{per}(S[i\,..\,j])$ or there exists a single occurrence of P (Line 2). We only need to compute at most $\tau + 2k$ such sequences. If retrieving more sequences is possible, then P is (τ, k)-resilient in S due to Lemma 2. Otherwise, we construct $\mathcal{H}_P$ (Line 6). This can be done in $\mathcal{O}((\tau + k) \log n)$ time. Before explaining this time bound, observe that, if P occurs at position i' then all occurrences of P that start in $[i', i' + \mathsf{lcp}(i', i' + \mathsf{per}(P))]$ belong to one run. We achieve the stated time bound by iterating over the occurrences in $\mathcal{O}$ in increasing order as follows, starting from the first such occurrence. We process the i-th occurrence (if we have to) as follows; let f be the corresponding starting position. We compute $m = \lfloor \mathsf{lcp}(f, f + \mathsf{per}(P))/\mathsf{per}(P) \rfloor + 1$ and insert (f, m) to $\mathcal{H}_P$. The computed m occurrences of P belong to one run and can be skipped. We thus proceed to processing the $(i + m)$-th occurrence of P. Retrieving the starting position of the j-th occurrence for any j can be done in $\mathcal{O}(\log n)$ time from the sorted list $\mathcal{O}$ (maintained as an AVL tree). The described process is repeated $\mathcal{O}(\tau + k)$ times. Therefore the time complexity is $\mathcal{O}((\tau + k) \log n)$.

Algorithm 3 Create clusters for periodic fragment $P = S[i\,..\,j]$, given random access to P's sorted list $\mathcal{O}$ of occurrences in S

```
1: function CreateClusters(i, j, p, k, τ, O)
2:     Construct a partition P of O, such that |P| < τ + 2k, in
       every set C ∈ P, |C| = 1 or any two consecutive elements of C
       are at distance p = per(S[i..j]) and C is inclusion-maximal.
3:     if ∑_{C∈P} |C| < |occ_S(S[i..j])|
4:         return H_P = ∅
5:     else
6:         return H_P = {(C[0], |C|) for all C ∈ P}
```

F. Details of PERIODICRESILIENT

In this subsection we prove the following lemma.

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.*

A pseudocode implementation of our algorithm is provided as Algorithm 4. The only lines that do not take $\mathcal{O}(1)$ time are decorated with their time complexity. Recall that the preprocessing takes $\mathcal{O}(n \log n)$ time. Thus, the time complexity

follows from that $\mathcal{H}_P$ is computed in $\mathcal{O}(\log n + \tau + k)$ time (Lemma 5) and that each while loop is executed $\mathcal{O}(\tau + k)$ times, each of which needs $\mathcal{O}(\log n)$ time.

Algorithm 4 Check if a periodic substring of S is (τ, k)-resilient

1: **function** PERIODICRESILIENT($i, j, l_u, r_u, \tau, k, \mathsf{per}(S[i\,..\,j])$)
2: $\mathcal{H}_P \leftarrow$ COMPUTEHP(i, j, τ, k) ▷ See Lemma 5; $\mathcal{O}(\tau + k + \log n)$ time
3: **if** $\mathcal{H}_P = \varnothing$
4: **return** YES
5: $\mathsf{occ} \leftarrow r_u - l_u + 1$
6: $\alpha \leftarrow \lceil \frac{j-i+1}{\mathsf{per}(S[i\,..\,j])} \rceil$ ▷ Max no. of occs that a token can destroy
7: $\mathcal{H}_P^{(1,2)} \leftarrow \varnothing$ ▷ A subset of $\mathcal{H}_P$
8: $d \leftarrow 0$ ▷ Destroyed occurrences
9: $t \leftarrow k$ ▷ Available tokens
10: $\mathcal{R} \leftarrow$ empty max-heap of positive integers ▷ With multiplicity counters
11: $(f, m) \leftarrow$ the first element in $\mathcal{H}_P$
12: **while** $(\mathsf{occ} - d \geq \tau)$ and $(t > 0)$ and $(f, m) \neq nil$ **do** ▷ Iterate over $\mathcal{H}_P$
13: $y \leftarrow \min\{t, \lfloor m/\alpha \rfloor\}$
14: $d \leftarrow d + y \cdot \alpha$
15: $t \leftarrow t - y$
16: **if** $m \bmod \alpha \geq 3$
17: **if** $m \bmod \alpha \notin \mathcal{R}$
18: Insert $[m \bmod \alpha, 1]$ into $\mathcal{R}$ ▷ $\mathcal{O}(\log n)$ time
19: **else** Increment the counter of $m \bmod \alpha$ in $\mathcal{R}$ by one ▷ $\mathcal{O}(\log n)$ time
20: **else if** $m \bmod \alpha = 1$ or $m \bmod \alpha = 2$
21: Insert (f, m) into $\mathcal{H}_P^{(1,2)}$
22: $(f, m) \leftarrow$ the next element of $\mathcal{H}_P$ ▷ nil if we have processed all of $\mathcal{H}_P$
23: **if** $\mathsf{occ} - d \geq \tau$ and $t > 0$
24: **while** $\mathcal{R} \neq \varnothing$ and $\mathsf{occ} - d \geq \tau$ and $t > 0$ **do** ▷ Iterate over $\mathcal{R}$
25: $[r, c_r] \leftarrow \mathcal{R}.\mathsf{top}()$
26: Pop $\mathcal{R}.\mathsf{top}()$ ▷ $\mathcal{O}(\log n)$ time
27: $y \leftarrow \min\{t, c_r\}$
28: $d \leftarrow d + y \cdot r$
29: $t \leftarrow t - y$
30: $\kappa \leftarrow 0$ ▷ Max number of tokens that can each destroy two distinct occs
31: **if** $\mathsf{occ} - d \geq \tau$ and $t > 0$
32: $z \leftarrow -\infty$ ▷ Position of rightmost processed and surviving occ
33: **for all** $(f, m) \in \mathcal{H}_P^{(1,2)}$ **do** ▷ In increasing order
34: **if** $f - z < j - i + 1$ ▷ Overlap between two runs
35: $\kappa \leftarrow \kappa + 1$ ▷ Destroy a pair of occs
36: **if** $m \bmod \alpha = 2$
37: $z \leftarrow f + (m-1)\mathsf{per}(S[i\,..\,j])$
38: **else if** $m \bmod \alpha = 2$
39: $\kappa \leftarrow \kappa + 1$ ▷ Destroy the remaining pair of occs of (f, m)
40: **else**
41: $z \leftarrow f + (m-1)\mathsf{per}(S[i\,..\,j])$
42: **if** $\mathsf{occ} - d - \min\{\kappa, t\} - \min\{\mathsf{occ} - d, t\} \geq \tau$
43: **return** YES
44: **return** NO

The rest of the proof is for correctness. Our algorithm first constructs $\mathcal{H}_P$ and returns YES if it is $\varnothing$. In the remainder of this section, we assume that this is not the case.

Our algorithm maintains the number t of available tokens and a counter d of the maximum number of occurrences of P that can be destroyed with the number of tokens that have been used so far. In Lines 5-22, we use tokens to destroy α occurrences of P as long as this is possible, while populating max-heap $\mathcal{R}$ and $\mathcal{H}_P^{(1,2)}$. Then, we start popping elements from $\mathcal{R}$ (Lines 24-29). If $\mathcal{R}.\mathsf{top}() = [r, c_r]$, we use $y = \min\{t, c_r\}$ tokens to destroy a total of $y \cdot c_r$ occurrences of P; the values of d and t are updated accordingly (Lines 28-29). Elements are popped from $\mathcal{R}$ as long as $|\mathsf{occ}_S(P)| - d \geq \tau$ and $t > 0$ (Line 24). If $\mathcal{R}$ is empty, $|\mathsf{occ}_S(P)| - d \geq \tau$, and $t > 0$, then the remaining (one or two) occurrences of P in each of the runs of type one and two are processed with a greedy procedure, analogously to Algorithm 1. This is done as follows (Lines 30-41). We process the elements of $\mathcal{H}_P^{(1,2)}$ in a left-to-right manner, and maintain a counter κ for the maximum number of pairs of remaining occurrences that can be destroyed. When we process a run (f, m), we check whether this run overlaps with an occurrence of P in an already processed run of $\mathcal{H}_P^{(1,2)}$ that has not been destroyed yet:

- If that is the case, we increment κ (Lines 34-35). Additionally, if $m \bmod \alpha = 2$, we set the starting position of the rightmost occurrence that has been processed but has not been destroyed to $f + (m-1)\mathsf{per}(S[i\,..\,j])$ (Lines 36-37).
- Otherwise, when $m \bmod \alpha = 2$, we destroy the two remaining occurrences of (f, m) (Lines 38-39); in the remaining case when $m \bmod \alpha = 1$, we set the starting position of the rightmost occurrence that has been processed but has not been destroyed to $f + (m-1)\mathsf{per}(S[i\,..\,j])$ (Lines 40-41).

Observe, that as we have $\mathsf{occ} - d$ occurrences remaining when the greedy procedure commences (if it does), the total number of occurrences that we can destroy with the t available tokens is $\min\{\kappa, t\} + \min\{\mathsf{occ} - d, t\}$. Thus, the test of Line 42 allows us to conclude whether P is (τ, k)-resilient.

The correctness of PERIODICRESILIENT is based on the fact that it first destroys the batches of occurrences of P of size α; then it processes $\mathcal{R}$, destroying numbers of occurrences of P greater than 2 (and less than α) *in decreasing order*, and then it processes the remaining occurrences (from $\mathcal{H}_P^{(1,2)}$) using a variant of Algorithm 1; the correctness of the latter follows from Lemma 2.

G. Proof of Theorem 3

In Phase 1, we perform a DFS of $\mathsf{ST}(S)$, using AVL trees to maintain the sorted list of occurrences of $\mathsf{str}(u)$ for each node u of $\mathsf{ST}(S)$; this requires $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space; see Section V-C. For each node of $\mathsf{ST}(S)$ with at least $\tau + k$ descendants, we first call ISPERIODIC which costs $\mathcal{O}(1)$ time (Theorem 2) and then call either APERIODICRESILIENT or PERIODICRESILIENT. Each call to APERIODICRESILIENT costs $\mathcal{O}(\tau + k)$ time (Lemma 4), while each call to function PERIODICRESILIENT costs $\mathcal{O}((\tau + k) \log n)$ time (Lemma 6). Therefore, the total time for Phase 1 is $\mathcal{O}(n \log n)$.

In Phase 2, we perform a DFS of $\mathsf{ST}(S)$ maintaining AVL trees as in Phase 1. For each node v, such that edge (u, v) is on the preliminary cut computed in Phase 1, we compute the sorted list of occurrences of $\mathsf{str}(v)$ and then we perform binary search for this edge. In total, we perform $\mathcal{O}(n \log n/(\tau + k))$ calls to each of the functions ISPERIODIC, APERIODICRESILIENT, and PERIODICRESILIENT. All calls

to the first two functions take $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space due to Theorem 2 and Lemma 4. We show that all calls to the function PERIODICRESILIENT cost $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. If we simply called PERIODICRESILIENT each time during the binary searches, we would obtain an algorithm running in $\mathcal{O}(n \log^2 n)$ time (see Lemma 6). We achieve a better running time by (i) employing the following combinatorial lemma; and (ii) avoiding the usage of heaps and rather sorting globally.

Lemma 15. *Let (u, v) be an edge of $\mathsf{ST}(S)$, $\ell \in [\mathsf{sd}(u), \mathsf{sd}(v) - 1)$ be an integer, and $P = \mathsf{str}(v)[0\,..\,\ell]$. If P is periodic and $\mathcal{H}_P$ does not only contain elements of the form $(\star, 1)$, then $\mathcal{H}_P = \mathcal{H}_{\mathsf{str}(v)}$.*

Proof. If $\mathsf{per}(P) < \mathsf{per}(\mathsf{str}(v))$, then P does not have any two occurrences in S at distance $\mathsf{per}(P)$ as otherwise $\mathsf{per}(\mathsf{str}(v))$ would be at most $\mathsf{per}(P)$. Hence, all elements of $\mathcal{H}_P$ are of the form $(\star, 1)$.

In the remaining case, we have $\mathsf{per}(P) = \mathsf{per}(\mathsf{str}(v))$. Note that any occurrence of $\mathsf{str}(v)$ implies an occurrence of P at the same position. Since $\mathsf{occ}_S(P) = \mathsf{occ}_S(\mathsf{str}(v))$, the statement follows. □

We are now ready to explain how we can perform all calls to PERIODICRESILIENT during the binary searches of Phase 2 more efficiently than by simply employing Lemma 6. Note that Algorithm 4 (underlying Lemma 6) only incurs logarithmic factors due to the call to Lemma 5 and to inserting/popping elements to/from a max-heap. The $\log n$ factor from each heap's operation can be avoided by sorting all elements to be inserted to heaps (in each step of the binary search) over all edges of the cut as a batch in $\mathcal{O}(n+(\tau+k)\cdot n/(\tau+k)) = \mathcal{O}(n)$ time in total using bucket sort. As we have $\mathcal{O}(\log n)$ batches (one for each step of the binary searches), the total time required for sorting is $\mathcal{O}(n \log n)$. Moreover, due to Lemma 15, we need to call Lemma 5 at most once for each considered edge (u, v) of $\mathsf{ST}(S)$ as we can reuse the computed runs in $\mathcal{H}_{\mathsf{str}(v)}$ (if $\mathsf{str}(v)$ is periodic) for all prefixes of $\mathrm{str}(v)$ of lengths more than $\mathsf{sd}(u)$ and with period equal to $\mathsf{per}(\mathrm{str}(v))$. Although there might be a prefix that is periodic with period $q < \mathsf{per}(\mathrm{str}(v))$, as such a prefix does not have any two occurrences at positions that are q positions apart, we can treat it as an aperiodic substring using Algorithm 2. We thus obtain the main result of this work.

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.*

VI. EVALUATION MEASURES FOR CLUSTERING

In the following, we discuss the two well-known measures of clustering quality we used in the clustering case study, namely Normalized Mutual Information (NMI) [9] and the Rand Index (RI) [10].

A clustering is a partition of a collection of strings into sets called clusters. We consider two clusterings C, C' with sizes $|C|$ and $|C'|$, respectively, and a collection of N strings. C is the ground truth clustering. NMI and RI quantify how similar is C' to the ground truth clustering. The values in all these measures are in $[0, 1]$, where a higher value indicates that C' is closer to the ground truth clustering.

The NMI between C and C' is defined as:

$$\mathrm{NMI}(C, C') = \frac{2 \cdot \sum_{i=1}^{|C|} \sum_{j=1}^{|C'|} \frac{n_{ij}}{N} \log\left(\frac{n_{ij/N}}{n_i \cdot \hat{n}_j / N^2}\right)}{-\sum_{i=1}^{|C|} \frac{n_i}{N} \log \frac{n_i}{N} - \sum_{j=1}^{|C'|} \frac{\hat{n}_j}{N} \log \frac{\hat{n}_j}{N}},$$

where n_i denotes the number of strings in the ith cluster in C, $\hat{n}_j$ denotes the number of strings in the jth cluster in C', and n_{ij} denotes the number of clusters belonging both in the ith cluster in C and in the jth cluster in C'.

The Rand Index (RI) measures similarity between the clustering C' and the ground truth clustering C by counting the number of pairs of strings that are placed in the same clusters in C and C', in different clusters in C and C', or in the same clusters in one of the C and C' and in different clusters in the other. $\mathrm{RI}(C, C')$ is defined as follows:

$$\mathrm{RI}(C, C') = \frac{a + b}{a + b + c + d},$$

where:

- a is the number of pairs of strings that are in the same cluster in C and C'.
- b is the number of pairs of strings that are in different clusters in both C and C'.
- c is is the number of pairs of strings that are in the same cluster in C but in different clusters in C'.
- d is is the number of pairs of strings that are in different clusters in C but in the same cluster in C'.

REFERENCES

[1] P. Damaschke, "Refined algorithms for hitting many intervals," *Inf. Process. Lett.*, vol. 118, pp. 117–122, 2017.

[2] 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. [Online]. Available: https://doi.org/10.1016/S1570-8667(03)00065-0

[3] M. Brown and R. Tarjan, "A fast merging algorithm," *Journal of the ACM (JACM)*, vol. 26, no. 2, pp. 211–226, Apr. 1979.

[4] E. M. McCreight, "Priority search trees," *SIAM J. Comput.*, vol. 14, no. 2, pp. 257–276, 1985.

[5] J. Ellert and J. Fischer, "Linear time runs over general ordered alphabets," in *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, 2021, pp. 63:1–63:16.

[6] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen, "Extracting powers and periods in a word from its runs structure," *Theor. Comput. Sci.*, vol. 521, pp. 29–41, 2014. [Online]. Available: https://doi.org/10.1016/j.tcs.2013.11.018

[7] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta, "The "runs" theorem," *SIAM J. Comput.*, vol. 46, no. 5, pp. 1501–1514, 2017. [Online]. Available: https://doi.org/10.1137/15M1011032

[8] T. Kociumaka, "Minimal suffix and rotation of a substring in optimal time," in *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, 2016, pp. 28:1–28:12.

[9] X. V. Nguyen, J. Epps, and J. Bailey, "Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance," *J. Mach. Learn. Res.*, vol. 11, pp. 2837–2854, 2010. [Online]. Available: https://dl.acm.org/doi/10.5555/1756006.1953024

[10] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *Journal of the American Statistical association*, vol. 66, no. 336, pp. 846–850, 1971.