Shortest Unique Substring Queries on Run-Length Encoded Strings

Takuya Mieno[1], Shunsuke Inenaga[2], Hideo Bannai[3], and Masayuki Takeda[4]

1   Department of Informatics, Kyushu University, Fukuoka, Japan
    takuya.mieno@inf.kyushu-u.ac.jp
2   Department of Informatics, Kyushu University, Fukuoka, Japan
    inenaga@inf.kyushu-u.ac.jp
3   Department of Informatics, Kyushu University, Fukuoka, Japan
    bannai@inf.kyushu-u.ac.jp
4   Department of Informatics, Kyushu University, Fukuoka, Japan
    takeda@inf.kyushu-u.ac.jp

Abstract

We consider the problem of answering shortest unique substring (SUS) queries on run-length encoded strings. For a string S, a unique substring $u = S[i..j]$ is said to be a *shortest unique substring* (*SUS*) of S containing an interval $[s, t]$ ($i \leq s \leq t \leq j$) if for any $i' \leq s \leq t \leq j'$ with $j - i > j' - i'$, $S[i'..j']$ occurs at least twice in S. Given a run-length encoding of size m of a string of length N, we show that we can construct a data structure of size $O(m + \pi_s(N, m))$ in $O(m \log m + \pi_c(N, m))$ time such that queries can be answered in $O(\pi_q(N, m) + k)$ time, where k is the size of the output (the number of SUSs), and $\pi_s(N, m)$, $\pi_c(N, m)$, $\pi_q(N, m)$ are, respectively, the size, construction time, and query time for a predecessor/successor query data structure of m elements for the universe of $[1, N]$. Using the data structure by Beam and Fich (JCSS 2002), this results in a data structure of $O(m)$ space that is constructed in $O(m \log m)$ time, and answers queries in $O(\sqrt{\log m / \log \log m} + k)$ time.

1   Introduction

The *shortest unique substring* (*SUS*) problem is, given a string S of length N and query interval $[s, t] \subseteq [1, N]$ of positions in S, find the shortest substring $S[x..y]$ of S that contains $[s, t]$ (i.e., $[s, t] \subseteq [x, y]$) and is *unique* (i.e., occurs only once) in S. Finding SUSs has possible applications in various fields, including alignment free genome comparison [10], PCR primer design [17], and display of search results [17]. The problem was first introduced by Pei et al. [17], who considered only a single position (i.e., an interval of size 1) as the query input, and showed that the string can be preprocessed in $O(N^2)$ time and $O(N)$ space so that a single SUS for a query position can be returned in constant time. Again for the single position query, Tsuruta et al. [20], Ileri et al. [13], and Hon et al. [11] independently showed that the preprocessing can be improved to $O(N)$ time and space, with constant query time. Tsuruta et al. [20] and Ileri et al. [13] also showed that all SUSs that contain the query position can be answered in $O(k)$ time, where k is the number of SUSs to output. Hu et al. [12] further generalized the problem to interval queries, and showed that it can be solved

in $O(N)$ time and space, and all SUSs that contain the query interval can be answered in $O(k)$ time.

In this paper, we consider the SUS problem for interval queries in the case where the string is given in *run-length encoding* (*RLE*). The RLE of a string is a natural compressed representation where each maximal character run of a character a of length e is encoded as a^e. String processing on the compressed representation of a string without explicit decompression [1] is a heavily studied topic, and can lead to time and space efficient processing [18]. There have been many studies on efficient algorithms for processing RLE strings [7, 8, 2, 3, 16, 14, 9, 15, 4]. We show that given a run-length encoding of size m of a string, we can construct a data structure of size $O(m + \pi_s(N, m))$ in $O(m \log m + \pi_c(N, m))$ time such that all SUSs that contain the query interval can be answered in $O(\pi_q(N, m) + k)$ time, where k is the number of such SUSs and $\pi_s(N, m)$, $\pi_c(N, m)$, $\pi_q(N, m)$ are, respectively, the size, construction time, and query time for a predecessor/successor query data structure of m elements for the universe of $[1, N]$. Using the data structure by Beam and Fich [5], this results in a data structure of size $O(m)$ space that is constructed in $O(m \log m)$ time, and answers queries in $O(\sqrt{\log m / \log \log m} + k)$ time. Thus, compared to previous work [12], our algorithm allows for more time and space efficient preprocessing for RLE compressible strings, with a slight increase in query time.

Our result is an outcome of a non-trivial mixed use of combinatorial properties of RLE strings and data structures built on RLE strings: All existing solutions [17, 20, 13, 12, 11] to the SUS problem precompute *minimal unique substrings* (*MUSs*) of a given string, which are minimal substrings of S occurring exactly once in S, and store them in $\Theta(N)$ space, since, in general, there can be $\Theta(N)$ MUSs in a given string. However, using combinatorial properties of MUSs and RLE strings, we show in this paper that any string of RLE size m contains at most $2m - 1$ MUSs, enabling our space-efficient $O(m)$-size data structure for the SUS problem. This bound is indeed tight, namely, some strings contain $2m - 1$ MUSs. In our algorithm, we separately treat MUSs that are completely contained in runs, those that start at the last characters of runs, and the rest. We then show that all the MUSs can be precomputed in $O(m \log m)$ time using a special type of suffix arrays for RLE strings [19]. Finally, we show how, given all MUSs, to efficiently compute all SUSs for any given query interval.

2 Preliminaries

2.1 Notations

Let $\Sigma = \{1, \ldots, \sigma\}$ be an *alphabet*. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is the string of length 0, namely, $|\varepsilon| = 0$. The i-th character of a string S of length N is denoted by $S[i]$ for $1 \leq i \leq N$. For $1 \leq i \leq j \leq N$, let $S[i..j] = S[i] \cdots S[j]$, i.e., $S[i..j]$ is the *substring* of S starting at position i and ending at position j in S. For convenience, let $S[i..j] = \varepsilon$ if $j < i$. For any $1 \leq i \leq N$, non-empty strings $S[1..i]$ and $S[i..N]$ are respectively called *prefixes* and *suffixes* of S. Let $suf_S(i) = S[i..N]$. For any strings X and Y, let $lcp(X, Y)$ denote the length of the *longest common prefix* of X and Y. For any string S of length n and any $1 \leq i \leq j \leq N$, let $lce_S(i, j) = lcp(suf_S(i), suf_S(j))$. If a string X is lexicographically smaller than another string Y, then we write $X \prec Y$ or $Y \succ X$.
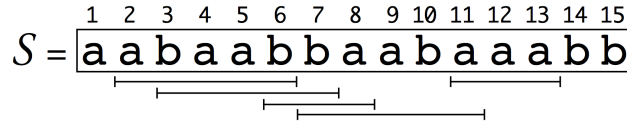
Figure 1 All the MUSs of string $S = $ `aabaabbaabaaabb` are $S[2..6] = $ `abaab`, $S[3..7] = $ `baabb`, $S[6..8] = $ `bba`, $S[7..11] = $ `baaba`, and $S[11..13] = $ `aaa`.
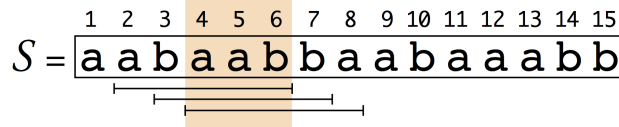
Figure 2 Consider the same string $S = $ `aabaabbaabaaabb` as in Figure 1. All the SUSs that contain a query interval $[4, 6]$ are $S[2..6] = $ `abaab`, $S[3..7] = $ `baabb`, and $S[4..8] = $ `aabba`. The first two SUSs are also MUSs of S, while the last SUS is obtained by taking the beginning position 4 of the query interval $[4, 6]$ and the ending position of a MUS $S[6..8]$ that overlaps the query interval.

2.2 MUSs and SUSs

For any non-empty strings S and w, let $occ_S(w)$ denote the set of occurrences of w in S, namely, $occ_S(w) = \{i \mid 1 \le i \le |S| - |w| + 1, w = S[i..i + |w| - 1]\}$. A substring w of a string S is called a *unique substring* (resp. a *repeat*) of S if $|occ_S(w)| = 1$ (resp. $|occ_S(w)| \ge 2$). In the sequel, we will identify each unique substring w of S with its corresponding (unique) interval $[i, j]$ in S such that $w = S[i..j]$. A unique substring $u = S[i..j]$ is said to be *right minimal unique* if for any $i \le j' < j$, $S[i..j']$ is a repeat of S. A unique substring $u = S[i..j]$ is said to be *left minimal unique* if for any $i < i' \le j$, $S[i'..j]$ is a repeat of S. A substring $u = S[i..j]$ is said to be a *minimal unique substring* (*MUS*) of S if u is right minimal unique and left minimal unique. Let $\mathcal{M}_S$ denote the set of all MUSs of S. Also, a unique substring $u = S[i..j]$ is said to be a *shortest unique substring* (*SUS*) of S containing an interval $[s, t]$ $(i \le s \le t \le j)$ if for any $i' \le s \le t \le j'$ with $j - i > j' - i'$, $S[i'..j']$ is a repeat of S. Hu et al. [12] showed that precomputing all MUSs $\mathcal{M}_S$ in a given string S, later allows to efficiently answer all SUSs that contain any query range $[s, t]$. See Figures 1 and 2 for examples of MUSs and SUSs of a string.

2.3 Run-length encodings and our problem

The *run-length encoding* (*RLE*) of string S of length N is a compact representation of S which encodes each maximal character run $S[i..i + e - 1]$ by a^e, if (1) $S[j] = a$ for all $i \le j \le i + e - 1$, (2) $S[i - 1] \ne S[i]$ or $i = 1$, and (3) $S[i + e - 1] \ne S[i + e]$ or $i + e - 1 = N$. E.g., $RLE(\texttt{aabbbbcccaaa\$}) = \texttt{a}^2\texttt{b}^4\texttt{c}^3\texttt{a}^3\texttt{\$}^1$. The *size* of $RLE(S) = a_1^{e_1} \cdots a_m^{e_m}$ is the number m of maximal character runs in S and is denoted by $|RLE(S)|$. For any $1 \le i \le m$, let $bpos_S(i)$, $epos_S(i)$, and $exp_S(i)$ respectively denote the beginning position, ending position, and exponent of the ith run of $RLE(S)$ in the original string S; namely, $bpos_S(i) = 1 + \sum_{k=1}^{i-1} e_k$, $epos_S(i) = \sum_{k=1}^{i} e_k$, and $exp_S(i) = e_i$.

In this paper, we will tackle the following problem:

▶ **Problem 1** (SUSs on RLE strings).
Preprocess: $RLE(S) = a_1^{e_1} \cdots a_m^{e_m}$ *of size m of string S of length N.*
Query: *An interval $[s, t] \in [1, N]$.*
Return: *All SUSs of S containing the query interval $[s, t]$.*

	tRLESA_S^{-1}	EXP_S	tRLESA_S	tRLELCP_S	
1	3	2	10	0	$a^{(2)}\ b^2\ c^3\ \$^1$
2	6	1	6	1	$a^{(1)}\ c^2\ a^2\ b^2\ c^3\ \$^1$
3	2	3	3	4	$a^{(3)}\ c^2\ a^1\ c^2\ a^2\ b^2\ c^3\ \$^1$
4	5	2	12	0	$b^{(2)}\ c^3\ \$^1$
5	1	2	8	0	$c^{(2)}\ a^2\ b^2\ c^3\ \$^1$
6	4	2	5	2	$c^{(2)}\ a^1\ c^2\ a^2\ b^2\ c^3\ \$^1$
7	7	3	15	1	$c^{(3)}\ \$^1$
8	8	1	16	0	$\$^{(1)}$
9				0	

Figure 3 tRLESA_S, tRLESA_S^{-1}, tRLELCP_S, and EXP_S for $RLE(S) = a^3 c^2 a^1 c^2 a^2 b^2 c^3 \$^1$ with $m = 8$ and $N = |S| = 16$. We remark that the exponents of the first runs in parentheses are all regarded as 1. For instance, consider the suffixes of lexicographical ranks 2 and 3. Although $a^1 c^2 a^2 b^2 c^3 \$^1$ is lexicographically greater than $a^3 c^2 a^1 c^2 a^2 b^2 c^3 \$^1$, $a^1 c^2 a^2 b^2 c^3 \$^1$ is lexicographically smaller than $a^1 c^2 a^1 c^2 a^2 b^2 c^3 \$^1$, and tRLESA_S builds on the latter ordering.

Our model of computation is a standard word RAM with machine word size $\Omega(\log N)$. The space complexity of our algorithm to solve Problem 1 will be evaluated by the number of words (rather than bits).

3 Tools

In this section, we list some data structure which we use to solve Problem 1.

3.1 Suffix arrays and related arrays for RLE strings

Let S be a string of length N and let $B \subseteq [1, N]$ be any subset of positions in S called sampled positions. The sparse suffix array SSA_B of a string S w.r.t. B is an array of size $|B|$ such that $\text{SSA}_B[i] \in B$ for all $1 \le i \le |B|$ and $suf_S(\text{SSA}_B[i]) \prec suf_S(\text{SSA}_B[i+1])$ for all $1 \le i < N$.

We will use the following arrays in our algorithm for computing SUSs on RLE strings. These arrays were first introduced in [19]. Let $m = |RLE(S)|$ and $E = \{epos_S(i) \mid 1 \le i \le m\}$. The *truncated RLE suffix array* for $RLE(S)$, denoted tRLESA_S, is the sparse suffix array of S w.r.t. E. Namely, for any $1 \le i \le m$, $\text{tRLESA}_S[i] = j$ iff $j \in E$ and the lexicographical rank of the suffix $S[j..N]$ is i among all suffixes of S that begin with positions in E. Let tRLESA_S^{-1} be an array of size m such that $\text{tRLESA}_S[\text{tRLESA}_S^{-1}[i]] = epos_S(i)$ for all $1 \le i \le m$. Let tRLELCP_S be an array of size $m+1$ such that $\text{tRLELCP}_S[1] = \text{tRLELCP}_S[m+1] = 0$ and $\text{tRLELCP}_S[i] = lce_S(\text{tRLESA}_S[i-1], \text{tRLESA}_S[i]) = lcp(suf_S(\text{tRLESA}_S[i-1]), suf_S(\text{tRLESA}_S[i]))$ for all $2 \le i \le m$. Also, let EXP_S be an array of size m such that $\text{EXP}_S[i] = exp_S(k)$ where $\text{tRLESA}_S[i] = epos_S(k)$ for all $1 \le i \le m$, namely, $\text{EXP}_S[i]$ stores the ignored exponent of the first run of the ith suffix in tRLESA_S. See Figure 3 for concrete examples of these arrays.

▶ **Lemma 2** ([19]). *Given $RLE(S)$ of size m, tRLESA_S, tRLESA_S^{-1}, tRLELCP_S, and EXP_S can be computed in a total of $O(m \log m)$ time with $O(m)$ working space.*

The following is a simple observation of these arrays we will exploit.

▶ **Observation 3.** *For any $1 \leq i \leq m$, let*

$$l = \max\{\mathsf{tRLELCP}_S[p], \mathsf{tRLELCP}_S[p+1]\},$$

where $p = \mathsf{tRLESA}_S^{-1}[i]$. If $l \neq 0$, then l is the length of the longest repeat of S that starts at $epos_S(i)$.

For example of Observation 3, see Figure 3. There, for position $i = 3$, we have $p = \mathsf{tRLESA}_S^{-1}[3] = 2$. Then, observe that $l = \max\{1, 4\} = 4$ is the length of the longest repeat $\mathtt{ac^2a}$ that starts at position $epos_S(3) = 6$. On the other hand, for position $i = 6$, we have $p = \mathsf{tRLESA}_S^{-1}[6] = 4$. Then, $l = \max\{0, 0\} = 0$, but this is not equal to the length 1 of the longest repeat $\mathtt{b}$ that starts at position $epos_S(6) = 12$. In our algorithm, we will use Observation 3 only the case where $l \neq 0$.

3.2 Range minimum/maximum query data structure

Let A be an integer array of size m. Give a query range $[i, j] \in [1, m]$, a *range minimum query* $\mathsf{RmQ}_A(i, j)$ returns the index of a minimum element in the subarray $A[i, j]$, namely, it returns one of $\arg\min_{i \leq k \leq j}\{A[k]\}$. Similarly, *range maximum query* $\mathsf{RMQ}_A(i, j)$ returns the index of a maximum element in the subarray $A[i, j]$, namely, it returns one of $\arg\max_{i \leq k \leq j}\{A[k]\}$. It is well-known (see e.g. [6]) that after an $O(m)$-time preprocessing over the input array A, $\mathsf{RmQ}_A(i, j)$ and $\mathsf{RMQ}_A(i, j)$ can be answered in $O(1)$ time for any query range $[i, j]$, using $O(m)$ space.

3.3 Some functions related to tRLESA

In this subsection, we introduce some functions related to tRLESA_S and the other arrays, which will be used in our algorithm to compute SUSs on RLE strings.

Consider $RLE(S)$ of size m. For any pair $(i, j) \in [1, m] \times [1, m]$, let $trle\_lce_S(i, j) = lce_S(\mathsf{tRLESA}_S[i], \mathsf{tRLESA}_S[j])$. Since

$$trle\_lce_S(i, j) = \begin{cases} \mathsf{RmQ}_{\mathsf{tRLELCP}_S}(i+1, j) & \text{if } i < j, \\ \mathsf{RmQ}_{\mathsf{tRLELCP}_S}(j+1, i) & \text{otherwise}, \end{cases}$$

after a linear-time preprocessing on $\mathsf{tRLELCP}_S$, we can answer $trle\_lce_S(i, j)$ in $O(1)$ time for any given pair (i, j).

For any $1 \leq q \leq m$ and $e \geq 1$, let $exp\_pos(q, e)$ denote a query which returns a position $q' \neq q$, if it exists, that satisfies $\mathsf{EXP}_S[q'] \geq e$ and $S[\mathsf{tRLESA}_S[q']] = S[\mathsf{tRLESA}_S[q]]$ while maximizing $trle\_lce(q, q')$, and *nil* otherwise. Thus, with $q' = exp\_pos(q, e)$, we can obtain the length of the longest repeating substring starting at position $\mathsf{tRLESA}_S[q] - e + 1$ as $e - 1 + trle\_lce(q, q')$.

▶ **Lemma 4.** *Given EXP_S for $RLE(S)$ of size m, we can preprocess EXP_S in $O(m)$ time so that subsequent $exp\_pos(q, e)$ queries can be answered in $O(\log m)$ time for any $1 \leq q \leq m$ and $e \geq 1$.*

Proof. We construct an RMQ data structure for EXP_S in $O(m)$ time. Since lexicographically close strings share a longer prefix, $exp\_pos(q, e)$ is one of the two closest neighbours of q in EXP_S that stores an exponent at least e, corresponding to a run of the same character. Thus, we can compute $exp\_pos(q, e)$ using two binary searches on EXP, by comparing e with the answer of the RMQ queries, starting with the initial range $[1, q - 1]$ and $[q + 1, m]$. Since the size of EXP_S is m and each RMQ query takes $O(1)$ time, it takes $O(\log m)$ time to locate $exp\_pos(q, e)$. ◀

For any $1 \leq q \leq m$ and $\ell \geq 0$, let $lce\_pos(q, \ell)$ denote a query which returns a position $q' \neq q$, if it exists, such that $trle\_lce(q, q') \geq \ell$ while maximizing $\mathsf{EXP}_S[q']$, and nil otherwise. In other words, $lce\_pos(q, \ell)$ corresponds to a suffix that has the maximum exponent out of suffixes which, have a common prefix of length ℓ with the suffix corresponding to q. Note that if $\ell > \max\{\mathsf{tRLELCP}[q], \mathsf{tRLELCP}[q+1]\}$, $lce\_pos(q, \ell) = nil$.

▶ **Lemma 5.** *Given* $\mathsf{tRLELCP}_S$ *for* $RLE(S)$ *of size* m, *we can preprocess* $\mathsf{tRLELCP}_S$ *in* $O(m)$ *time so that subsequent* $lce\_pos(q, \ell)$ *queries can be answered in* $O(\log m)$ *time for any* $1 \leq q \leq m$ *and* $\ell \geq 0$.

Proof. We construct an RmQ data structure on $\mathsf{tRLELCP}_S$. Since, as noted previously, lexicographically close strings share a longer prefix, values of $trle\_lce(q, q'')$ are larger when q'' is closer to q. Thus, similar to Lemma 4, we can conduct two binary searches on $\mathsf{tRLELCP}_S$ using RmQ and obtain the maximal range $[q_p, q_n]$ such that $trle\_lce(q, q'') \geq \ell$ if and only if $q'' \in [q_p, q_n]$. After finding the range, the larger of the two RMQ queries for the ranges $[q_p, q-1]$ and $[q+1, q_n]$ on EXP_S gives the answer. ◀

3.4 Predecessor/successor query data structure

Let $A[1..m]$ be an array containing positive integers less than or equal to N, in increasing order. The predecessor and successor queries on A are defined for any $1 \leq d \leq N$ as

$$\mathsf{Pred}_A(d) \;=\; \begin{cases} \max\{i \mid A[i] \leq d\} & \text{if it exists,} \\ 0 & \text{otherwise.} \end{cases}$$

$$\mathsf{Succ}_A(d) \;=\; \begin{cases} \min\{i \mid A[i] \geq d\} & \text{if it exists,} \\ N+1 & \text{otherwise.} \end{cases}$$

There exists a data structure of $O(m)$ space that can be built in $O(m\sqrt{\log m / \log\log m})$ time, such that later, for any given $1 \leq d \leq N$, $\mathsf{Pred}_A(d)$ and $\mathsf{Succ}_A(d)$ can be answered in $O(\sqrt{\log m / \log\log m})$ time [5].

4   Computing MUSs from RLE strings

In this section we show how we can compute $\mathcal{M}_S$ given $RLE(S)$, which is the main part of our preprocessing. As will be seen in Section 4.1, we partition MUSs into three disjoint groups; those that are completely contained in runs, those that start at the last characters of runs, and the rest.

4.1   Size of $\mathcal{M}_S$

We begin with the analysis of the size of $\mathcal{M}_S$ in terms of $m = |RLE(S)|$. Let

$$\mathcal{M}^{(1)} = \{[x, y] \in \mathcal{M}_S \mid bpos_S(i) \leq x \leq y \leq epos_S(i) \text{ for some } 1 \leq i \leq m\},$$
$$\mathcal{M}^{(2)} = \{[x, y] \in \mathcal{M}_S \mid x = epos_S(i) < y \text{ for some } 1 \leq i < m\}, \text{ and}$$
$$\mathcal{M}^{(3)} = \{[x, y] \in \mathcal{M}_S \mid bpos_S(i) \leq x < epos_S(i) < y \text{ for some } 1 \leq i < m\}.$$

Clearly, $\mathcal{M}_S = \mathcal{M}^{(1)} \cup \mathcal{M}^{(2)} \cup \mathcal{M}^{(3)}$. For example, for the same string $S = \mathtt{aaaccaccaabbccc\$}$ as in Figure 3, $\mathcal{M}_S = \{[1,3], [2,4], [5,7], [8,10], [10,11], [11,12], [12,13], [13,15], [16,16]\} = \{\mathtt{aaa}, \mathtt{aac}, \mathtt{cac}, \mathtt{caa}, \mathtt{ab}, \mathtt{bb}, \mathtt{bc}, \mathtt{ccc}, \$\}$, $\mathcal{M}^{(1)} = \{\mathtt{aaa}, \mathtt{bb}, \mathtt{ccc}, \$\}$, $\mathcal{M}^{(2)} = \{\mathtt{cac}, \mathtt{caa}, \mathtt{ab}, \mathtt{bc}\}$, and $\mathcal{M}^{(3)} = \{\mathtt{aac}\}$.

Since, by definition, a MUS cannot be a proper substring of another MUS, there can be at most one MUS that starts at any given position. Thus, it follows that $|\mathcal{M}^{(2)}| \leq m - 1$.

For $|\mathcal{M}^{(3)}|$, we have the following lemma.

▶ **Lemma 6.** *For any $[x, y] \in \mathcal{M}^{(3)}$ and $p \in occ_S(S[x + 1..y]) \setminus \{x + 1\}$, we have that $p = bpos_S(i)$ for some $1 \leq i < m$.*

Proof. Since $S[x..y]$ is a MUS, $S[x + 1..y]$ is not unique and thus $occ_S(S[x + 1..y]) \setminus \{x + 1\}$ is not empty. If $p \neq bpos_S(i)$ for any $1 \leq i < m$, then $S[p - 1] = S[p]$ and thus gives another occurrence of $S[x..y]$ contradicting that it is unique. ◀

We now show $|\mathcal{M}^{(3)} \cup \mathcal{M}^{(1)}| \leq m$. Let $\mathcal{R} = \{bpos_S(i) \mid 1 \leq i \leq m\}$. From Lemma 6, we can define a function $f : \mathcal{M}^{(3)} \cup \mathcal{M}^{(1)} \to \mathcal{R}$ as follows:

$$f([x, y]) = \begin{cases} \min(occ_S(S[x + 1..y]) \setminus \{x + 1\}) & \text{if } [x, y] \in \mathcal{M}^{(3)} \\ x & \text{if } [x, y] \in \mathcal{M}^{(1)} \end{cases}$$

Suppose f is not an injective function, i.e., there exist distinct intervals $[x_1, y_1], [x_2, y_2] \in \mathcal{M}^{(3)} \cup \mathcal{M}^{(1)}$ such that $x_1 \neq x_2$ and $p = f([x_1, y_1]) = f([x_2, y_2])$. Note that by definition, $\mathcal{M}^{(3)} \cap \mathcal{M}^{(1)} = \emptyset$.

If $[x_1, y_1], [x_2, y_2] \in \mathcal{M}^{(3)}$, assume w.l.o.g. $y_1 - x_1 \leq y_2 - x_2$. By definition of f, we have $S[x_1 + 1..y_1] = S[p..p + y_1 - x_1 - 1]$ and $S[x_2 + 1..y_2] = S[p..p + y_2 - x_2 - 1]$. Also, from the definition of $\mathcal{M}^{(3)}$, we have $S[x_1] = S[x_1 + 1] = S[p] = S[x_2 + 1] = S[x_2]$. It follows that $S[x_1..y_1]$ is a prefix of $S[x_2..y_2]$, contradicting that $S[x_1..y_1]$ is unique. If $[x_1, y_1] \in \mathcal{M}^{(3)}$ and $[x_2, y_2] \in \mathcal{M}^{(1)}$, this implies that $S[x_2..y_2]$ is a prefix of $S[x_1 + 1..y_1]$ which is not unique, thus contradicting that $S[x_2..y_2]$ is unique. Finally, if $[x_1, y_1], [x_2, y_2] \in \mathcal{M}^{(1)}$, $p = f([x_1, y_1]) = f([x_2, y_2])$ implies that $p = x_1 = x_2$ contradicting that $x_1 \neq x_2$. Thus, f must be an injective function. Therefore, $|\mathcal{M}^{(3)} \cup \mathcal{M}^{(1)}| \leq |\mathcal{R}| = m$.

From the above arguments, we have:

▶ **Lemma 7.** $|\mathcal{M}_S| \leq 2m - 1$.

We note that the upper bound of Lemma 7 is tight, and there exists a string S such that $|\mathcal{M}_S| = 2m - 1$. Consider $S = a_1^{e_1} a_2^{e_2} \cdots a_m^{e_m}$ such that for any $1 \leq i, j \leq m$, $e_i \geq 2$, and $a_i \neq a_j$ when $i \neq j$. Clearly, $a_i^{e_i}$ is a MUS for all $1 \leq i \leq m$, and $a_i a_{i+1}$ is a MUS for all $1 \leq i < m$, giving $2m - 1$ MUSs.

4.2 Computing $\mathcal{M}_S$

We now show how to obtain $\mathcal{M}_S$ in $O(m \log m)$ time and $O(m)$ space, by computing the sets $\mathcal{M}^{(1)}, \mathcal{M}^{(2)}, \mathcal{M}^{(3)}$ as defined in Section 4.1.

4.2.1 Computing $\mathcal{M}^{(1)}$

To compute $\mathcal{M}^{(1)}$, we first show a necessary and sufficient condition for an interval $[x, y]$ to be in $\mathcal{M}^{(1)}$.

▶ **Lemma 8.** *For any string S where $RLE(S) = a_1^{e_1} \cdots a_m^{e_m}$, an interval $[x, y] \in \mathcal{M}^{(1)}$ if and only if there exists some $1 \leq i \leq m$ such that $bpos_S(i) = x$, $epos_S(i) = y$, and for any $j \in [1, m] \setminus \{i\}$, either $a_i \neq a_j$ or $e_j < e_i$.*

Proof. ($\Rightarrow$) Since $[x, y]$ is a MUS and any proper substring of $[x, y]$ is not unique, it must be that $x = bpos_S(i), y = epos_S(i)$ for some $1 \leq i \leq m$. Furthermore, it must be that $a_i \neq a_j$ or $e_j < e_i$ for any $j \in [1, m] \setminus \{i\}$, since otherwise, $[x, y]$ will not be unique. ($\Leftarrow$) The condition implies that $S[x..y]$ is the longest run of character a_i in S and is unique. Since any proper substring of $S[x..y]$ is not unique, $[x, y]$ is a MUS and is thus in $\mathcal{M}^{(1)}$. ◀

Let Σ_S be the subset of Σ consisting of letters occurring in S. Using Lemma 8, we can compute $\mathcal{M}^{(1)}$ by simply checking for each character $a \in \Sigma_S$, whether there exists a run of character a with a unique (w.r.t. runs of character a) maximum exponent, and if so, include the interval corresponding to the run in $\mathcal{M}^{(1)}$. Since $|\Sigma_S| \leq m$, this can be done in $O(m \log m)$ time and $O(m)$ space using any standard sorting algorithm.

4.2.2   Computing $\mathcal{M}^{(2)}$

To compute, $\mathcal{M}^{(2)}$, we check for each $1 \leq i \leq m - 1$, whether there exists a MUS that starts at $epos_S(i)$ and insert it in $\mathcal{M}^{(2)}$ if there is. More specifically, we first compute y such that $S[epos_S(i)..y]$ is right minimal unique. Next, we check whether $S[epos_S(i) + 1..y]$ is unique or not, and if not, we have that $[epos_S(i), y]$ is also left minimal unique and thus is a MUS.

Let $r = \mathsf{tRLESA}_S^{-1}[i]$. By Observation 3, we have that $l = \max\{\mathsf{tRLELCP}_S[r], \mathsf{tRLELCP}_S[r + 1]\}$ is the length of the longest repeat of S that starts at $epos_S(i)$. This implies that $S[epos_S(i)..epos_S(i) + l]$ is right minimal unique. Thus, given the $\mathsf{tRLELCP}_S$ array, $y = epos_S(i) + l$ can be computed in constant time. Next, to determine whether $S[epos_S(i) + 1..y]$ is unique or not, we compute y' such that $S[epos_S(i) + 1..y']$ is right minimal unique. Then, $[epos_S(i) + 1, y]$ is unique iff $y' \leq y$. Noticing that $epos_S(i) + 1 = bpos_S(i + 1)$, we can compute y' as follows. Let $q = \mathsf{tRLESA}_S^{-1}[i + 1]$ and $x = \mathsf{EXP}_S[q]$. We compute $l' = x - 1 + trle\_lce_S(q, q')$, where $q' = exp\_pos(q, x)$. By definition, we have that l' is the length of the longest repeat of S that starts at $bpos_S(i + 1)$. Thus, $y' = bpos_S(i + 1) + l'$. By Lemma 4, this can be computed in $O(m \log m)$ total time and $O(m)$ space for all i.

4.2.3   Computing $\mathcal{M}^{(3)}$

For each $1 \leq i < m$, we will compute the elements of $\mathcal{M}^{(3)}$ that start in the ith run. Let $s = bpos_S(i)$ and we repeat the following while $s < epos_S(i)$. First, compute y such that $S[s..y]$ is right minimal unique. If such y does not exists, i.e., $S[s..|S|]$ is not unique, then we are done. If y does exist, $y \geq epos_S(i)$ since, as noted earlier, no proper substring of a run can be unique. If $y = epos_S(i)$, we must have that $s = bpos_S(i)$ and $[s, y]$ is a MUS in $\mathcal{M}^{(1)}$ and not in $\mathcal{M}^{(1)}$; thus we simply increment s by 1 and repeat the process. Otherwise, if $y > epos_S(i)$, we try to find x such that $S[x..y]$ is left minimal unique. Then, by definition, $[x, y]$ is a MUS. If $x < epos_S(i)$, then we have that $[x, y]$ is a MUS in $\mathcal{M}^{(3)}$, and since there can be no other MUS that starts in the interval $[s, x]$, we set $s = x + 1$ and repeat the process. Otherwise, if $x \geq epos_S(i)$, then $[x, y]$ is either a MUS in $\mathcal{M}^{(2)}$ or does not start in the ith run, so we are finished for the current value of i. Because we obtain one distinct MUS each time we determine y and x, the above process is repeated for a total of $O(m)$ times for all i by Lemma 7. What remains is how to determine y and x.

Whether $y = epos_S(i)$ or not can be determined by checking if $[s, epos_S(i)]$ is a MUS in $\mathcal{M}^{(1)}$ as described in Section 4.2.1. Next, we assume $y \geq epos_S(i) + 1 = bpos_S(i + 1)$. Let $q = \mathsf{tRLESA}_S^{-1}[i]$, $q' = exp\_pos(q, epos_S(i) - s + 1)$. If q' is nil, this implies that no run other than the ith one contains a run of character $S[epos_S(i)]$ with length at least $epos_S(i) - s + 1$. Since $y > epos_S(i)$, we have that $S[s..epos_S(i)]$ is not unique but $S[s..bpos_S(i + 1)]$ is unique and thus, $y = bpos_S(i + 1)$. Otherwise, if q' is not nil, then, we have that $epos_S(i) - s + l$, where

$l = trle\_lce_S(q, q')$, is the length of the longest repeat of S that starts at s. Therefore, we have $y = epos_S(i) + l$. From the above arguments and Lemma 4, y can be determined in $O(\log m)$ time. Whether $x \geq epos_S(i)$ or not can be determined by the arguments for checking whether $[epos_S(i), y]$ is a MUS in $\mathcal{M}^{(2)}$, as described in Section 4.2.2. Next, we assume $x < epos_S(i)$. Then, $S[epos_S(i)..y]$ is a repeat. Let $q = \mathsf{tRLESA}_S^{-1}(i)$, $q' = lce\_pos(q, y - epos_S(i) + 1)$. From the definition of lce_pos, we have $x = epos_S(i) - \mathsf{EXP}_S[q'] + 1$. Thus, from the above arguments and Lemma 5, x can be determined in $O(\log m)$ time.

The arguments from Sections 4.2.1-4.2.3 lead to the following lemma.

▶ **Lemma 9.** *For any string S, the set $\mathcal{M}_S$ can be computed from $RLE(S)$ in $O(m \log m)$ time using $O(m)$ space, where $m = |RLE(S)|$.*

5 Solution to the SUS Problem

5.1 Data structure

Our data structure consists of three arrays: X_S, Y_S, and MUSlen_S. Arrays X_S and Y_S are arrays of size $|\mathcal{M}_S|$ such that for any $1 \leq i \leq |\mathcal{M}_S|$, $[\mathsf{X}_S[i], \mathsf{Y}_S[i]]$ is the ith MUS in order of their start position in S. Also, let the array $\mathsf{MUSlen}_S[i] = \mathsf{Y}_S[i] - \mathsf{X}_S[i] + 1$ hold the length of each MUS. Arrays X_S and Y_S are preprocessed for Succ and Pred queries, and MUSlen_S is preprocessed for RmQ queries. From arguments in previous sections, the preprocessing can clearly be done in a total of $O(m \log m)$ time and $O(m)$ space.

5.2 Answering queries

For any two intervals $[s, t]$ and $[x, y]$, let $cover([s, t], [x, y])$ be the smallest interval that contains both $[s, t], [x, y]$, i.e., $cover([s, t], [x, y]) = [\min\{s, x\}, \max\{t, y\}]$.

Given a query interval $[s, t]$, let $i = \mathsf{Pred}_{\mathsf{Y}_S}(t)$ and $j = \mathsf{Succ}_{\mathsf{X}_S}(s)$. Clearly, all SUSs that contain interval $[s, t]$ are contained in the set $\{|cover([s, t], [\mathsf{X}_S[r], \mathsf{Y}_S[r]])| \mid i \leq r \leq j\}$. Thus, it suffices to find the intervals of smallest size in this set, i.e., if $p \in \arg\min\{|cover([s, t], [\mathsf{X}[r], \mathsf{Y}[r]])| \mid i \leq r \leq j\}$, then $cover([s, t], [\mathsf{X}_S[p], \mathsf{Y}_S[p]])$ is a SUS. Notice that for all $i < r < j$, we have that $cover([s, t], [\mathsf{X}_S[r], \mathsf{Y}_S[r]]) = [\mathsf{X}_S[r], \mathsf{Y}_S[r]]$. Thus, the shortest of these can be found by considering $cover([s, t], [\mathsf{X}_S[i], \mathsf{Y}_S[i]])$, $cover([s, t], [\mathsf{X}_S[j], \mathsf{Y}_S[j]])$, and performing an RmQ query on MUSlen_S. An example is shown in Figure 4. For finding a single SUS, the query time is dominated by the Pred and Succ queries, and thus is $O(\pi_q(N, m))$ time. To output all SUSs that contain $[s, t]$, recursive RmQ on sub-intervals of MUSlen_S can be conducted in constant time per output, in order to find all the shortest intervals in the range $[i, j]$. Thus, the total query time is $O(\pi_q(N, m) + k)$, where k is the total number of SUSs that are output.

Putting everything together, we have proved the following theorem:

▶ **Theorem 10.** *Given $RLE(S)$ of size m representing a string S of length N, we can compute in $O(m \log m + \pi_c(N, m))$ time a data structure of size $O(m + \pi_s(N, m))$ which answers SUS queries for any interval $[s, t] \subseteq [1, N]$ in $O(\pi_q(N, m) + k)$ time, where k is the number of SUSs to output.*

Using known results for predecessor/successor queries [5], we obtain the following corollary.

▶ **Corollary 11.** *Given $RLE(S)$ of size m representing a string S of length N, we can compute in $O(m \log m)$ time a data structure of size $O(m)$ which answers SUSs queries for any interval $[s, t] \subseteq [1, N]$ in $O(\sqrt{\log m / \log \log m} + k)$ time, where k is the number of such SUSs.*
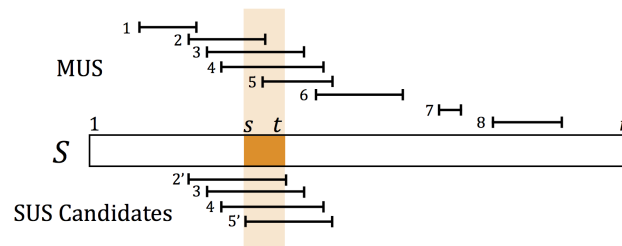
Figure 4 Finding SUSs that contains query interval $[s, t]$. The SUS must be either a MUS that completely contains $[s, t]$ (MUS 3,4), or, it must be an interval that covers both $[s, t]$ and the preceding MUS (MUS 2) or succeeding MUS (MUS 5). Of these, the intervals with shortest length are the SUSs that contain $[s, t]$.

6   Conclusions and open question

We considered the problem of finding all shortest unique substrings (SUSs) of a string S given as the run-length encoding (RLE) of size m. We showed that we can preprocess the RLE in $O(m \log m)$ time and $O(m)$ space so that subsequent SUS queries for S can be answered in $O(\sqrt{\log m / \log \log m} + k)$ time, where k is the number of outputs for the query interval. Notice that none of the preprocessing time, space requirement, or query time depends on the original length N of the string S. This efficiency was achieved by a non-trivial use of the suffix arrays for RLE strings and by revealing combinatorial properties of MUSs and SUSs on RLE strings.

The $\sqrt{\log m / \log \log m}$ term in our query time is due to the use of the $O(m)$-space dynamic predecessor/successor data structure by Beame and Fich [5]. They also showed that for a static set $\mathcal{A}$ of m integers from the universe $[1, N]$, any predecessor/successor data structure for $\mathcal{A}$ of polynomial size in m must use $\Omega(\sqrt{\log m / \log \log m})$ query time (Corollary 3.10 of [5]). Notice that once we build arrays X_S and Y_S, they will remain static. Hence, we cannot hope for faster SUS query time as long as we use predecessor/successor queries to find a MUS for a given interval. Thus, an interesting open question is whether there exists a data structure of size $O(m)$ that can efficiently answer SUS queries without using predecessor/successor queries.

───── **References** ─────

1   Amihood Amir, Gary Benson, and Martin Farach. Let sleeping files lie: pattern matching in z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, April 1996.

2   Alberto Apostolico, Gad M. Landau, and Steven Skiena. Matching for run-length encoded strings. *J. Complex.*, 15(1):4–16, March 1999. `doi:10.1006/jcom.1998.0493`.

3   Ora Arbell, Gad M. Landau, and Joseph S.B. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, 2002. `doi:10.1016/S0020-0190(02)00215-6`.

4   Golnaz Badkobeh, Gabriele Fici, Steve Kroon, and Zsuzsanna Lipták. Binary jumbled string matching for highly run-length compressible texts. *Information Processing Letters*, 113(17):604–608, 2013. `doi:10.1016/j.ipl.2013.05.007`.

5 Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002. `doi:10.1006/jcss.2002.1822`.

6 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, LATIN 2000, pages 88–94, 2000. URL: `http://dl.acm.org/citation.cfm?id=646388.690192`.

7 Horst Bunke and János Csirik. An algorithm for matching run-length coded strings. *Computing*, 50(4):297–314, 1993.

8 Horst Bunke and János Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Information Processing Letters*, 54(2):93–96, April 1995.

9 Kuan-Yu Chen, Ping-Hui Hsu, and Kun-Mao Chao. Efficient retrieval of approximate palindromes in a run-length encoded string. *Theoretical Computer Science*, 432:28–37, 2012. `doi:10.1016/j.tcs.2012.01.023`.

10 Bernhard Haubold, Nora Pierstorff, Friedrich Möller, and Thomas Wiehe. Genome comparison without alignment using shortest unique substrings. *BMC Bioinformatics*, 6(1):123, 2005.

11 Wing-Kai Hon, Sharma V. Thankachan, and Bojian Xu. An in-place framework for exact and approximate shortest unique substring queries. In *ISAAC 2015*, pages 755–767, 2015.

12 Xiaocheng Hu, Jian Pei, and Yufei Tao. Shortest unique queries on strings. In *Proc. SPIRE 2014*, pages 161–172, 2014.

13 Atalay Mert Ileri, M. Oguzhan Külekci, and Bojian Xu. A simple yet time-optimal and linear-space algorithm for shortest unique substring queries. *Theor. Comput. Sci.*, 562:621–633, 2015.

14 Jin Wook Kim, Amihood Amir, Gad M. Landau, and Kunsoo Park. Computing similarity of run-length encoded strings with affine gap penalty. *Theoretical Computer Science*, 395(2–3):268–282, 2008. SAIL – String Algorithms, Information and Learning: Dedicated to Professor Alberto Apostolico on the occasion of his 60th birthday. `doi:10.1016/j.tcs.2008.01.008`.

15 Jia-Jie Liu, Guan-Shieng Huang, and Yue-Li Wang. A fast algorithm for finding the positions of all squares in a run-length encoded string. *Theoretical Computer Science*, 410(38–40):3942–3948, 2009. `doi:10.1016/j.tcs.2009.05.032`.

16 Mäkinen, Ukkonen, and Navarro. Approximate matching of run-length compressed strings. *Algorithmica*, 35(4):347–369, 2003. `doi:10.1007/s00453-002-1005-2`.

17 Jian Pei, Wush Chi-Hsuan Wu, and Mi-Yen Yeh. On shortest unique substring queries. In *Proc. ICDE 2013*, pages 937–948, 2013.

18 Masayuki Takeda. *Encyclopedia of algorithms*, chapter "Compressed Pattern Matching", pages 171–174. Springer US, 2008.

19 Yuya Tamakoshi, Keisuke Goto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. An opportunistic text indexing structure based on run length encoding. In *Proc. CIAC 2015*, pages 390–402, 2015.

20 Kazuya Tsuruta, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Shortest unique substrings queries in optimal time. In *Proc. SOFSEM 2014*, pages 503–513, 2014.