Shortest Unique Substring Queries on Run-Length Encoded Strings

Takuya Mieno\textsuperscript{1}, Shunsuke Inenaga\textsuperscript{2}, Hideo Bannai\textsuperscript{3}, and Masayuki Takeda\textsuperscript{4}

\textsuperscript{1} Department of Informatics, Kyushu University, Fukuoka, Japan
takuya.mieno@inf.kyushu-u.ac.jp
\textsuperscript{2} Department of Informatics, Kyushu University, Fukuoka, Japan
inenaga@inf.kyushu-u.ac.jp
\textsuperscript{3} Department of Informatics, Kyushu University, Fukuoka, Japan
bannai@inf.kyushu-u.ac.jp
\textsuperscript{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 \(e\) of length \(i\) is encoded as \(a^i\). 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 \(O(m \log m + \pi_s(N, m))\) 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 \(\pi_s(N, m), \pi_e(N, m), \pi_q(N, m)\) 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 \(\Sigma^*\) is called a string. The length of a string \(S\) is denoted by \(|S|\). The empty string \(\epsilon\) is the string of length 0, namely, \(|\epsilon| = 0\). 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 \(S[i..i] = \epsilon\) 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 \(\text{suf}_S(i) = S[i..N]\). For any strings \(X\) and \(Y\), let \(\text{lcp}(X, Y)\) 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 \(\text{lcs}_S(i, j) = \text{lcp}(\text{suf}_S(i), \text{suf}_S(j))\). If a string \(X\) is lexicographically smaller than another string \(Y\), then we write \(X < Y\) or \(Y > X\).
1 ≤ \text{run-length encoding}

\text{Return: All SUSs of Query: An interval}

\text{Preprocess: } RLE(S) = a_1^e_1 \cdots a_m^e_m \text{ of size } m \text{ of string } S \text{ of length } N.

\text{Query: An interval } [s, t] \in [1, N].

\text{Return: All SUSs of } S \text{ containing the query interval } [s, t].
Shortest Unique Substring Queries on Run-Length Encoded Strings

The truncated RLE suffix array

These arrays were first introduced in [19]. Let

\[ \text{tRLESA}_S \]

of words (rather than bits).

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 \( 

\[ \text{SSA}_B \]

of a string \( S \) w.r.t. \( B \) is an array of size \( |B| \) such that \( \text{SSA}_B[i] \in B \) for all \( 1 \leq i \leq |B| \) and \( \text{suf}_S(\text{SSA}_B[i]) < \text{suf}_S(\text{SSA}_B[i+1]) \) for all \( 1 \leq i < N \).

We will use the following arrays in our algorithm for computing SUSs on RLE strings. These arrays were first introduced in [19]. Let \( m = |\text{RLE}(S)| \) and \( E = \{ \text{epos}_S(i) \mid 1 \leq i \leq m \} \).

The truncated RLE suffix array for \( \text{RLE}(S) \), denoted \( \text{tRLESA}_S \), is the sparse suffix array of \( S \) w.r.t. \( E \). Namely, for any \( 1 \leq i \leq m \), \( \text{tRLESA}_S[i] = j \) 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 \( \text{tRLESA}_S^{-1} \) be an array of size \( m \) such that \( \text{tRLESA}_S[\text{tRLESA}_S^{-1}[i]] = \text{epos}_S(i) \) for all \( 1 \leq i \leq m \). Let \( \text{tRLELCP}_S \) be an array of size \( m+1 \) such that \( \text{tRLELCP}_S[1] = \text{tRLELCP}_S[m+1] = 0 \) and \( \text{tRLELCP}_S[i] = \text{lces}(\text{tRLESA}_S[i-1], \text{tRLESA}_S[i]) \) for all \( 2 \leq i \leq m \). Also, let \( \text{EXP}_S \) be an array of size \( m \) such that \( \text{EXP}_S[i] = \text{exp}_S(k) \) where \( \text{tRLESA}_S[i] = \text{epos}_S(k) \) for all \( 1 \leq i \leq m \), namely, \( \text{EXP}_S[i] \) stores the ignored exponent of the first run of the \( i \)th suffix in \( \text{tRLESA}_S \). See Figure 3 for concrete examples of these arrays.

Lemma 2 ([19]). Given \( \text{RLE}(S) \) of size \( m \), \( \text{tRLESA}_S \), \( \text{tRLESA}_S^{-1} \), \( \text{tRLELCP}_S \), and \( \text{EXP}_S \) 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
\[
l = \max\{\text{tRLELCP}_S[p], \text{tRLELCP}_S[p + 1]\},
\]
where \(p = \text{tRLESA}_S^{-1}[i]\). If \(l \neq 0\), then \(l\) is the length of the longest repeat of \(S\) that starts at \(\text{epos}_S(i)\).

For example of Observation 3, see Figure 3. There, for position \(i = 3\), we have \(p = \text{tRLESA}_S^{-1}[3] = 2\). Then, observe that \(l = \max\{1, 4\} = 4\) is the length of the longest repeat \(ac^2a\) that starts at position \(\text{epos}_S(3) = 6\). On the other hand, for position \(i = 6\), we have \(p = \text{tRLESA}_S^{-1}[6] = 4\). Then, \(l = \max\{0, 0\} = 0\), but this is not equal to the length 1 of the longest repeat \(b\) that starts at position \(\text{epos}_S(6) = 12\). 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 \([i, j] \subseteq [1, m]\), a range minimum query \(\text{RmQ}_A(i, j)\) returns the index of a minimum element in the subarray \(A[i, j]\), namely, it returns one of \(\arg\min_{k \leq j} \{A[k]\}\). Similarly, range maximum query \(\text{RMQ}_A(i, j)\) returns the index of a maximum element in the subarray \(A[i, j]\), namely, it returns one of \(\arg\max_{k \leq j} \{A[k]\}\).

It is well-known (see e.g. [6]) that after an \(O(m)\)-time preprocessing over the input array \(A\), \(\text{RmQ}_A(i, j)\) and \(\text{RMQ}_A(i, j)\) can be answered in \(O(1)\) time for any query range \([i, j]\), using \(O(m)\) space.

3.3 Some functions related to \text{tRLESA}

In this subsection, we introduce some functions related to \(\text{tRLESA}_S\) and the other arrays, which will be used in our algorithm to compute SUSs on RLE strings.

Consider \(\text{RLE}(S)\) of size \(m\). For any pair \((i, j) \subseteq [1, m] \times [1, m]\), let \(\text{tRLE}_\text{lc}_S(i, j) = \text{lc}_S(\text{tRLESA}_S[i], \text{tRLESA}_S[j])\). Since
\[
\text{trle}_\text{lc}_S(i, j) = \begin{cases} 
\text{RmQ}_{\text{tRLELCP}_S}(i + 1, j) & \text{if } i < j, \\
\text{RmQ}_{\text{tRLELCP}_S}(j + 1, i) & \text{otherwise},
\end{cases}
\]
after a linear-time preprocessing on \(\text{tRLELCP}_S\), we can answer \(\text{trle}_\text{lc}_S(i, j)\) in \(O(1)\) time for any given pair \((i, j)\).

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

Lemma 4. Given \(\text{EXP}_S\) for \(\text{RLE}(S)\) of size \(m\), we can preprocess \(\text{EXP}_S\) in \(O(m)\) time so that subsequent \(\text{exp}_\text{pos}(q, e)\) 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 \(\text{EXP}_S\) in \(O(m)\) time. Since lexicographically close strings share a longer prefix, \(\text{exp}_\text{pos}(q, e)\) is one of the two closest neighbours of \(q\) in \(\text{EXP}_S\) that stores an exponent at least \(e\), corresponding to a run of the same character. Thus, we can compute \(\text{exp}_\text{pos}(q, e)\) using two binary searches on \(\text{EXP}\), 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 \(\text{EXP}_S\) is \(m\) and each RMQ query takes \(O(1)\) time, it takes \(O(\log m)\) time to locate \(\text{exp}_\text{pos}(q, e)\).
Lemma 5. Given \( tRLELCP_S \) for \( RLE(S) \) of size \( m \), we can preprocess \( tRLELCP_S \) 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 \( tRLELCP_S \). 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 \( tRLELCP_S \) using \( RmQ \) and obtain the maximal range \( [q_p, q_u] \) such that \( trle_{lce}(q, q') \geq \ell \) if and only if \( q'' \in [q_p, q_u] \). After finding the range, the larger of the two \( RMQ \) queries for the ranges \( [q_p, q - 1] \) and \( [q + 1, q_u] \) 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

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

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

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 \( 1 \leq d \leq N \), \( \text{Pred}_A(d) \) and \( \text{Succ}_A(d) \) 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 \( M_S \) 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 \( M_S \)

We begin with the analysis of the size of \( M_S \) in terms of \( m = |RLE(S)| \). Let

\[
M^{(1)} = \{ [x, y] \in M_S \mid bpos_S(i) \leq x \leq y \leq epos_S(i) \text{ for some } 1 \leq i \leq m \},
\]

\[
M^{(2)} = \{ [x, y] \in M_S \mid x = epos_S(i) < y \text{ for some } 1 \leq i < m \}, \quad \text{and}
\]

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

Clearly, \( M_S = M^{(1)} \cup M^{(2)} \cup M^{(3)} \). For example, for the same string \( S = \text{aaacaccaabbcce}$\) as in Figure 3, \( M_S = \{ [1, 3], [2, 4], [5, 7], [8, 10], [10, 11], [11, 12], [12, 13], [13, 15], [16, 16] \} = \{ \text{aaa, aac, cac, caa, ab, bb, bc, ccc, }$\}, \( M^{(1)} = \{ \text{aa, bb, ccc, }$\}, \( M^{(2)} = \{ \text{cac, caa, ab, bc} \}, \) and \( M^{(3)} = \{ \text{aac} \).
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 \(|M^{(2)}| \leq m - 1\).

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

Lemma 6. For any \([x, y] \in M^{(3)}\) and \(p \in \text{occ}_S(S[x+1..y]) \setminus \{x+1\}\), 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 \(\text{occ}_S(S[x+1..y]) \setminus \{x+1\}\) 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 \(|M^{(3)} \cup M^{(1)}| \leq m\). Let \(R = \{bpos_S(i) | 1 \leq i \leq m\}\). From Lemma 6, we can define a function \(f : M^{(3)} \cup M^{(1)} \to R\) as follows:

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

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

If \([x_1, y_1], [x_2, y_2] \in M^{(3)}\), 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 \(M^{(3)}\), 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 \([x_1, y_1] \in M^{(3)}\) and \([x_2, y_2] \in M^{(1)}\), 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 \([x_1, y_1], [x_2, y_2] \in M^{(1)}\), \(p = f([x_1, y_1]) = f([x_2, y_2])\) implies that \(p = x_1 = x_2\) contradicting that \(x_1 \neq x_2\). Thus, \(f\) must be an injective function. Therefore, \(|M^{(3)} \cup M^{(1)}| \leq |R| = m|\).

From the above arguments, we have:

Lemma 7. \(|M_S| \leq 2m - 1|\).

We note that the upper bound of Lemma 7 is tight, and there exists a string \(S\) such that \(|M_S| = 2m - 1|\). 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 \(M_S\)

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

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

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

Lemma 8. For any string \(S\) where \(\text{RLE}(S) = a_1^{e_1} \cdots a_m^{e_m}\), an interval \([x, y] \in M^{(1)}\) 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 \(e_j < e_i\).
To compute, $M(1)$ 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 $M(1)$. 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 $M(2)$

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

Let $r = \text{tRLESA}_S^{-1}[i]$. By Observation 3, we have that $l = \max\{t_{\text{RLELCP}}_S[r], t_{\text{RLELCP}}_S[r+1]\}$ is the length of the longest repeat of $S$ that starts at $e_{pos}(i)$. This implies that $S[e_{pos}(i)\cdot e_{pos}(i) + l]$ is right minimal unique. Thus, given the $t_{\text{RLELCP}}_S$ array, $y = e_{pos}(i) + l$ can be computed in constant time. Next, to determine whether $S[e_{pos}(i)\cdot y]$ is unique or not, we compute $y'$ such that $S[e_{pos}(i)\cdot y']$ is right minimal unique. Then, $e_{pos}(i) + y'$ is unique if $y' \leq y$. Noticing that $e_{pos}(i) + 1 = b_{pos}(i) + 1$, we can compute $y'$ as follows. Let $q = t_{\text{RLESA}}_S^{-1}[i] + 1$ and $x = \text{EXP}_S[q]$. We compute $l' = x - 1 + \text{t_rle_lcs}(q, q')$, where $q' = \text{exp_pos}(q, x)$. By definition, we have that $l'$ is the length of the longest repeat of $S$ that starts at $b_{pos}(i) + 1$. Thus, $y' = b_{pos}(i + 1) + l'$. 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 $M(3)$

For each $1 \leq i < m$, we will compute the elements of $M(3)$ that start in the $i$th run. Let $s = b_{pos}(i)$ and we repeat the following while $s < e_{pos}(i)$. First, compute $y$ such that $S[s\cdot y]$ is right minimal unique. If such $y$ does not exists, i.e., $S[s\cdot y]$ is not unique, then we are done. If $y$ does exist, $y \geq e_{pos}(i)$ since, as noted earlier, no proper substring of a run can be unique. If $y = e_{pos}(i)$, we must have that $s = b_{pos}(i)$ and $[s, y]$ is a MUS in $M(1)$ and not in $M(1)$, thus we simply increment $s$ by 1 and repeat the process. Otherwise, if $y > e_{pos}(i)$, we try to find $x$ such that $S[x\cdot y]$ is left minimal unique. Then, by definition, $[x, y]$ is a MUS. If $x < e_{pos}(i)$, then we have that $[x, y]$ is a MUS in $M(3)$, 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 $x \geq e_{pos}(i)$, then $[x, y]$ is either a MUS in $M(2)$ or does not start in the $i$th 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 $y = e_{pos}(i)$ or not can be determined by checking if $[s, e_{pos}(i)]$ is a MUS in $M(1)$ as described in Section 4.2.1. Next, we assume $y \geq e_{pos}(i) + 1 = b_{pos}(i + 1)$. Let $q = t_{\text{RLESA}}_S^{-1}[i]$, $q' = \text{exp_pos}(q, e_{pos}(i) - s + 1)$. If $q'$ is nil, this implies that no run other than the $i$th one contains a run of character $S[e_{pos}(i)]$ with length at least $e_{pos}(i) - s + 1$. Since $y > e_{pos}(i)$, we have that $S[s..e_{pos}(i)]$ is not unique but $S[s..b_{pos}(i + 1)]$ is unique and thus, $y = b_{pos}(i + 1)$. Otherwise, if $q'$ is not nil, then, we have that $e_{pos}(i) - s + 1$, where
\( l = \text{trle}_{\text{lce}}(q, q') \), is the length of the longest repeat of \( S \) that starts at \( s \). Therefore, we have \( y = \text{epos}_S(i) + l \). From the above arguments and Lemma 4, \( y \) can be determined in \( O(\log m) \) time. Whether \( x \geq \text{epos}_S(i) \) or not can be determined by the arguments for checking whether \( \text{epos}_S(i, y) \) is a repeat. Let \( q = \text{tRLESA}_S^{-1}(i) \), \( q' = \text{lce}_{\text{pos}}(q, y - \text{epos}_S(i) + 1) \). From the definition of \( \text{lce}_{\text{pos}} \), we have \( x = \text{epos}_S(i) - \text{EXP}_{S}[q'] + 1 \). 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.

\[ \text{Lemma 9.} \text{ For any string } S, \text{ the set } M_S \text{ can be computed from } \text{RLE}(S) \text{ in } O(m \log m) \text{ time using } O(m) \text{ space, where } m = |\text{RLE}(S)|. \]

5 Solution to the SUS Problem

5.1 Data structure

Our data structure consists of three arrays: \( X_S, Y_S, \) and \( \text{MUSlen}_S \). Arrays \( X_S \) and \( Y_S \) are arrays of size \( |M_S| \) such that for any \( 1 \leq i \leq |M_S|, [X_S[i],Y_S[i]] \) is the \( i \)th MUS in order of their start position in \( S \). Also, let the array \( \text{MUSlen}_S[i] = Y_S[i] - X_S[i] + 1 \) hold the length of each MUS. Arrays \( X_S \) and \( Y_S \) are preprocessed for \( \text{Succ} \) and \( \text{Pred} \) queries, and \( \text{MUSlen}_S \) is preprocessed for \( \text{RmQ} \) 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 \( \text{cover}([s, t], [x, y]) \) be the smallest interval that contains both \([s, t], [x, y]\), i.e., \( \text{cover}([s, t], [x, y]) = [\min(s, x), \max(t, y)] \).

Given a query interval \([s, t]\), let \( i = \text{Pred}_{X_S}(t) \) and \( j = \text{Succ}_{X_S}(s) \). Clearly, all SUSs that contain interval \([s, t]\) are contained in the set \( \{\text{cover}([s, t], [X_S[r], Y_S[r]]) | i \leq r \leq j\} \). Thus, it suffices to find the intervals of smallest size in this set, i.e., if \( \rho \in \arg \min\{\text{cover}([s, t], [X[r], Y[r]]) | i \leq r \leq j\} \), then \( \text{cover}([s, t], [X_S[\rho], Y_S[\rho]]) \) is a SUS. Notice that for all \( i < r < j \), we have \( \text{cover}([s, t], [X_S[r], Y_S[r]]) = [X_S[r], Y_S[r]] \). Thus, the shortest of these can be found by considering \( \text{cover}([s, t], [X_S[i], Y_S[i]]) \), \( \text{cover}([s, t], [X_S[j], Y_S[j]]) \), and performing an \( \text{RmQ} \) query on \( \text{MUSlen}_S \). An example is shown in Figure 4. For finding a single SUS, the query time is dominated by the \( \text{Pred} \) and \( \text{Succ} \) queries, and thus is \( O(\pi_q(N, m)) \) time.

To output all SUSs that contain \([s, t]\), recursive \( \text{RmQ} \) on sub-intervals of \( \text{MUSlen}_S \) 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:

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

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

\[ \text{Corollary 11.} \text{ Given } \text{RLE}(S) \text{ of size } m \text{ representing a string } S \text{ of length } N, \text{ we can compute in } O(m \log m) \text{ time a data structure of size } O(m) \text{ which answers SUSs queries for any interval } [s, t] \subseteq [1, N] \text{ in } O(\sqrt{\log m / \log \log m} + k) \text{ time, where } k \text{ is the number of such SUSs}. \]
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.

Acknowledgements. The authors especially thank an anonymous reviewer for invaluable comments which helped improve the paper. SI, HB, MT were partly supported by JSPS KAKENHI Grant Numbers 26280003, 16H02783, 25240003.

References


