Shortest Unique Queries on Strings

Xiaocheng Hu[1], Jian Pei[2], and Yufei Tao[1]

[1]Chinese University of Hong Kong, New Territories, Hong Kong
[2]Simon Fraser University, Burnaby, Canada
{xchu,taoyf}@cse.cuhk.edu.hk, jpei@cs.sfu.ca

Abstract. Let D be a long input string of n characters (from an alphabet of size up to 2^w, where w is the number of bits in a machine word). Given a substring q of D, a *shortest unique query* returns a shortest unique substring of D that contains q. We present an optimal structure that consumes $O(n)$ space, can be built in $O(n)$ time, and answers a query in $O(1)$ time. We also extend our techniques to solve several variants of the problem optimally.

1 Introduction

Let D be a (long) string. Define $n = |D|$ where $|D|$ represents the length of D. Denote by $D[i]$ $(1 \leq i \leq n)$ the i-th character of D, and by $D[i : j]$ $(1 \leq i \leq j \leq n)$ the substring of D starting at $D[i]$ and ending at $D[j]$. A string is *unique* if it has only one occurrence in D; otherwise, it is *repeating*. A substring $D[i_1 : j_1]$ *contains* another $D[i_2 : j_2]$ if $i_1 \leq i_2$ and $j_1 \geq j_2$ hold at the same time.

In this paper, we study data structures on D that can efficiently answer the following query, which was recently proposed in [9], motivated by its fundamental nature in numerous applications in text retrieval and bioinformatics:

Shortest Unique Query: Given a substring $q = D[x : y]$, such a query returns a substring of D with the minimum length among all the unique substrings of D containing q.

If $x = y$, we say that the query is a *point query*; otherwise, it is an *interval query*.

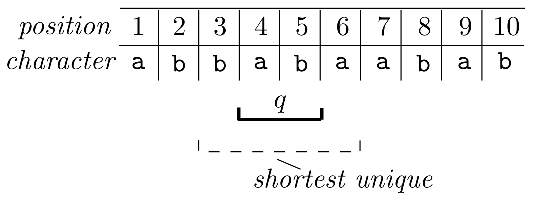

Fig. 1. An Example

Figure 1 shows a string D of length 10. Given $q = D[4 : 5] =$ ab, a shortest unique query may return $D[3 : 6] =$ baba because its length 4 is the smallest among all the unique substrings containing q. To verify this, notice that (i) baba is unique because it has only one occurrence in D, whereas (ii) $D[3:5] = \texttt{bab}$ is repeating (it occurs also at $d[8:10]$), and so is $D[4:6] = \texttt{aba}$ (see $D[7:9]$). This implies no unique string of length at most 3 contains q. Note that, in general, a query result can be output with only 2 integers, which specify its starting and ending positions in D, respectively.

We make the standard assumption that each character of D fits in a machine word. If w is the number of bits in a word, this assumption implies that the alphabet where the characters of D are drawn can have a size up to 2^w. Unless otherwise stated, the default model of computation is RAM.

Existing Results. Previous research has focused exclusively on point queries. In their initial study [9], Pei et al. showed how to construct in $O(n^2)$ time an index of $O(n)$ size that answers a query in $O(1)$ time. Soon after that, Ileri et al. [6] and Tsuruta et al. [10] independently improved the construction time to $O(n)$. It is worth mentioning that $O(n)$ size is considered optimal in the sense that D itself requires $\Omega(n)$ words to store when the alphabet is large.

Our Results. We present the first study on interval queries. Our main result is a new structure of $O(n)$ space that can be built in $O(n)$ time, and answers a query in $O(1)$ time. In other words, we achieve the optimal efficiency as with the previous work, but on more general queries.

At this point, it seems fair to delve a bit into a crucial difference between designing a structure for point and interval queries. What makes point queries easy to handle is that *there are only n of them*! Therefore, the problem of indexing is more of a one-off computation problem: how to quickly compute the answers for all those n queries. Once this is done, one can simply store these answers in an array to allow constant query time. This idea, however, no longer works for interval queries because now we have $\Theta(n^2)$ of them. Therefore, there needs to be a major shift in the indexing strategy, calling for novel ideas.

The rest of the paper is organized as follows. In Section 2, we will clarify some basic facts relevant to this study. Then, Section 3 will present our structure for interval queries. Section 4 further demonstrates the usefulness of the proposed techniques by extending them (i) to answer queries with additional constraints, and (ii) to support interval queries in external memory optimally.

2 Basic Definitions and Properties

In this section, we pave the way for our subsequent discussion by defining several concepts related to minimal unique substrings and explaining some of their fundamental properties.

Definition 1. *Each integer $p \in [1, n]$ defines a* **left-fixed minimal unique substring $MUS_{leftfix}(p)$** *as follows:*

- *$MUS_{leftfix}(p) = nil$, if $D[p:n]$ is repeating;*
- *otherwise, $MUS_{leftfix}(p) = D[p:z]$, where z is the smallest integer in $[p, n]$ such that $D[p:z]$ is unique.*

p	1	2	3	4	5	6	7	8	9	10
$MUS_{\mathit{leftfix}}(p)$	D[1:3] =abb	D[2:3] =bb	D[3:6] =baba	D[4:7] =abaa	D[5:7] =baa	D[6:7] =aa	D[7:10] =abab	*nil*	*nil*	*nil*
$MUS_{\mathit{rightfix}}(p)$	*nil*	*nil*	D[2:3] =bb	D[2:4] =bba	D[2:5] =bbab	D[3:6] =baba	D[6:7] =aa	D[6:8] =aab	D[6:9] =aaba	D[7:10] =abab

Fig. 2. The left-fixed and right-fixed minimal unique substrings in Figure 1

In other words, $MUS_{\mathit{leftfix}}(p)$ is the shortest unique substring of D starting at $D[p]$. In the example of Figure 1, $MUS_{\mathit{leftfix}}(4)$, for instance, is $D[4:7] = \texttt{abaa}$. Notice that $D[4:6]$ is repeating; and thus, $D[4:7]$ cannot be shortened on the right while still being unique. Viewed in another way, $D[4:7]$, $D[4:8]$ ..., $D[4:10]$ are all the unique substrings starting at $D[4]$; among them, $MUS_{\mathit{leftfix}}(4)$ is the shortest. See Figure 2 for the $MUS_{\mathit{leftfix}}(p)$ of all $p \in [1, 10]$.

The next definition is symmetric:

Definition 2. *Each integer $p \in [1, n]$ defines a* **right-fixed minimal unique substring $MUS_{\mathit{rightfix}}(p)$** *as follows:*

- *$MUS_{\mathit{rightfix}}(p) = nil$, if $D[1:p]$ is repeating;*
- *otherwise, $MUS_{\mathit{rightfix}}(p) = D[z:p]$, where z is the largest integer in $[1, p]$ such that $D[z:p]$ is unique.*

The last row of Figure 2 shows the $MUS_{\mathit{rightfix}}(p)$ of all $p \in [1, 10]$ for our running example. Now we are ready to define the most important concept:

Definition 3. *A substring $D[i:j]$ is a* **minimal unique substring** *(MUS) if*

$$MUS_{\mathit{leftfix}}(i) = D[i:j] \text{ and } MUS_{\mathit{rightfix}}(j) = D[i:j].$$

In other words, $D[i:j]$ is an MUS if (i) it is unique, and (ii) it can be shortened on *neither* side while still being unique. We will use $\mathcal{M}$ to denote the set of MUS's in D. From Figure 2, one can verify easily that the $\mathcal{M}$ in our example is:

$$\mathcal{M} = \{D[2:3] = \texttt{bb}, D[3:6] = \texttt{baba}, D[6:7] = \texttt{aa}, D[7:10] = \texttt{abab}\}. \quad (1)$$

$D[2:4] = \texttt{bba}$, for example, is *not* an MUS because it can be shortened on the right into bb which is still unique.

Lemma 1. *The strings in $\mathcal{M}$ have distinct left endpoints, and distinct right endpoints.*

Proof. Suppose $D[i_1 : j_1]$ and $D[i_2 : j_2]$ are two different strings in $\mathcal{M}$ but $i_1 = i_2$. This means that they are both $MUS_{\mathit{leftfix}}(i_1)$. But only one string can be $MUS_{\mathit{leftfix}}(i_1)$, thus giving a contradiction. Similarly, it must hold that $j_1 \neq j_2$. □

It has been shown [10] that all the substrings defined earlier can be computed efficiently:

Lemma 2 ([10]). *All the left-fixed MUS's, right-fixed MUS's, and MUS's can be computed from D in $O(n)$ time.*

In general, a substring $D[i : j]$ requires only two integers to represent: integers i and j. Therefore, all the left-fixed MUS's, right-fixed MUS's, and MUS's can be stored in $O(n)$ words. This leads to the following useful fact:

Corollary 1. *In $O(n)$ time, we can compute a structure of $O(n)$ size that, given any substring $D[i : j]$, we can check whether it is unique in D in $O(1)$ time.*

Proof. Simply compute all the left-fixed MUS's using Lemma 2. Then, given a substring $D[i : j]$, declare that it is unique if and only if $j \geq z$, where z is such that $MUS_{\mathit{leftfix}}(i) = D[i : z]$. □

3 A Data Structure for Interval Queries

This section serves as a proof for our main result:

Theorem 1. *Given a data string of length n, we can pre-compute in $O(n)$ time an index structure that consumes $O(n)$ space, and answers any shortest unique query in $O(1)$ time.*

3.1 A 4-Candidate Lemma

Lemma 3. *The answer of the shortest unique query with substring $q = D[x : y]$ must be the shortest of the following 4 candidates:*

1. *$D[x : y]$ if it is unique*
2. *$MUS_{\mathit{leftfix}}(x)$*
3. *$MUS_{\mathit{rightfix}}(y)$*
4. *the shortest MUS containing q (breaking length ties arbitrarily). No such candidate exists if no MUS contains q.*

Proof. First of all, if $D[x : y]$ is unique, then clearly $D[x : y]$ is the answer because no string containing q can be any shorter. The following discussion focuses on the scenario where $D[x : y]$ is repeating.

Let $D[x' : y']$ be an answer to the query. If $x' = x$, then it must hold that $MUS_{\mathit{leftfix}}(x) = D[x' : y']$; otherwise, either $MUS_{\mathit{leftfix}}(x)$ or $D[x' : y']$ can be shortened on the right end while still being unique, which contradicts their definitions. Likewise, if $y' = y$, then $MUS_{\mathit{rightfix}}(y) = D[x' : y']$.

In the remaining scenario, $x' < x$ and $y' > y$. Suppose that $D[x' : y']$ was not an MUS, namely, it can be still be shortened either on the left or right while still being unique. However, as both $D[x' + 1 : y']$ and $D[x' : y' - 1]$ contain q, we have found a unique string containing q that is even shorter than $D[x' : y']$, which contradicts the definition of $D[x' : y']$. □

Whether Candidate 1—namely $D[x : y]$—is unique can be checked in constant time using an $O(n)$-space structure (see Corollary 1). Also, Candidates 2 and 3 can be obtained in constant time using an $O(n)$-space structure (see Lemma 2). It thus remains to give a structure for finding Candidate 4.

As before, let $\mathcal{M}$ be the set of MUS's of D. For each MUS $D[i:j]$ in $\mathcal{M}$, create an interval $[i,j]$. Denote by $\mathcal{I}$ the set of all the intervals created this way. For the example of Figure 1, we know from Equation 1 that

$$\mathcal{I} = \{[2,3],[3,6],[6,7],[7,10]\} \tag{2}$$

Lemma 4. *No two intervals in $\mathcal{I}$ can contain each other.*

Proof. Suppose, on the contrary, that $[i_1, j_1]$ and $[i_2, j_2]$ are two different intervals in I such that $[i_1, j_1]$ contains $[i_2, j_2]$. Recall that $D[i_1 : j_1]$ and $D[i_2 : j_2]$ are both MUS's of D. However, that $[i_1, j_1]$ contains $[i_2, j_2]$ indicates that we can shorten $D[i_1 : j_1]$ to $D[i_2 : j_2]$ which is still unique. This violates the definition of MUS. $\square$

It is not hard to see that the problem ahead of us can be restated as:

> **Containment Min.** Let $\mathcal{I}$ be a set of at most n intervals in the domain $[1,n]$ such that no two intervals contain each other (a requirement inherited from Lemma 4). Given an interval $[x,y]$ in the domain $[1,n]$, a *containment min query* returns the shortest one (breaking ties arbitrarily) among all the intervals in $\mathcal{I}$ containing $[x,y]$. We want to store $\mathcal{I}$ in a data structure to answer such queries efficiently.

3.2 The Proposed Structure

In this subsection, we will present a structure of $O(n)$ space that answers a containment min query in $O(1)$ time, which will complete our proof of Theorem 1.

Idea. Let $m = |\mathcal{I}|$. From now on, we will view $\mathcal{I}$ as an ordered set

$$\{I_1 = [i_1, j_1], I_2 = [i_2, j_2], ..., I_m = [i_m, j_m]\}$$

where $i_1 < i_2 < ... < i_m$, and therefore $j_1 < j_2 < ... < j_m$[1]. For any $a < b$, we say that I_a is on the *left* of I_b, and conversely, I_b is on the *right* of I_a. Given a subset $S \subseteq \mathcal{I}$, we say that it is a *consecutive subset* if $S = \{I_a, I_{a+1}, ...I_b\}$ for some a, b satisfying $1 \le a \le b \le m$. We also regard the empty set $\emptyset$ as a consecutive subset.

For example, given the $\mathcal{I}$ in Equation 2, we have:

$$\{I_1 = [2,3], I_2 = [3,6], I_3 = [6,7], I_4 = [7,10]\}. \tag{3}$$

$\{I_3\}$ and $\{I_2, I_3, I_4\}$ are consecutive subsets, while $\{I_2, I_4\}$ is not. We observe:

Lemma 5. *For any $[x,y]$ in the domain $[1,n]$, the set of intervals of $\mathcal{I}$ containing $[x,y]$ must be a consecutive subset.*

Proof. Let a be the smallest integer such that $[i_a, j_a]$ contains $[x,y]$, and b be the largest integer such that $[i_b, j_b]$ contains $[x,y]$. For any integer $c \in [a,b]$, it holds that $i_c \le i_b \le x$ and $y \le j_a \le j_c$. In other words, $[i_c, j_c]$ contains $[x,y]$ as well. $\square$

[1] Otherwise, there must be an interval containing another, which violates Lemma 4.

y	1	2	3	4	5	6	7	8	9	10
$\alpha(y)$	1	1	1	2	2	2	3	4	4	4

x	1	2	3	4	5	6	7	8	9	10
$\beta(x)$	*nil*	1	2	2	2	3	4	4	4	4

Fig. 3. Arrays α and β on the $\mathcal{I}$ in Equation 3

Algorithm 1: COMPUTING-α-ARRAY

```
Input: A set $\mathcal{I}$ of $m$ intervals $I_1 = [i_1, j_1]$, ..., $I_m = [i_m, j_m]$, sorted in ascending order
       of left point. The domain is $[1, n]$.
Output: Array $\alpha$.
1 $z \leftarrow 1$
2 for $y = 1$ to $n$ do
3     while $j_z < y$ and $z \leq m$ do
4         $z \leftarrow z + 1$
5     if $z \leq m$ then
6         $\alpha(y) = z$
7     else
8         $\alpha(y) = nil$
9 return $\alpha$
```

The above lemma motivates the following strategy for solving the containment min query. Given a query interval $[x, y]$, we will find the leftmost interval I_a in $\mathcal{I}$ containing $[x, y]$, and the rightmost interval I_b in $\mathcal{I}$ containing $[x, y]$. Then, the remaining task is to find the shortest interval among the consecutive subset $\{I_a, I_{a+1}, ..., I_b\}$, which is nothing but a standard *range min query* (RMQ)! We can index $\mathcal{I}$ using an RMQ structure [4, 5] which uses $O(m) = O(n)$ space, can be constructed in $O(m)$ time, and answers an RMQ in $O(1)$ time.

Structure. It remains to explain how to design an index so that, given any $[x, y]$, we can derive the corresponding a and b in constant time. We resolve this issue with another key observation: a depends *only* on y! Formally, given a value $y \in [1, n]$, let us define $\alpha(y)$ as

– the smallest integer $z \in [1, m]$ such that $j_z \geq y$, if such a z exists;
– nil, otherwise.

In other words, I_z is the leftmost interval in $\mathcal{I}$ whose right endpoint is at least y. If such an interval exists, then $\alpha(y) = z$; otherwise, $\alpha(y) = nil$. The next lemma states the aforementioned observation formally:

Lemma 6. *Fix an integer $y \in [1, n]$. For any $x \in [1, y]$, all the following are true:*

1. *If $\alpha(y) = nil$, then $\mathcal{I}$ has no interval containing $[x, y]$.*
2. *If $I_{\alpha(y)}$ does not contain $[x, y]$, then $\mathcal{I}$ has no interval containing $[x, y]$.*
3. *If $I_{\alpha(y)}$ contains $[x, y]$, then it is the leftmost interval in $\mathcal{I}$ containing $[x, y]$.*

Proof. Statement 1 holds because when $\alpha(y) = nil$, all the intervals of $\mathcal{I}$ end strictly to the left of y.

Algorithm 2: CONTAINMENT-MIN

```
  Input: A query interval $[x, y]$.
  Output: The shortest interval in $\mathcal{I}$ containing $[x, y]$.
1 $a \leftarrow \alpha(y)$
2 $b \leftarrow \beta(x)$
3 if $a = nil$ or $b = nil$ then
4     return $nil$
5 if $I_a$ does not contain $[x, y]$ then
6     return $nil$
7 perform an RMQ to retrieve the shortest interval among $I_a, I_{a+1}, ..., I_b$
8 return the above interval
```

To prove Statement 2, suppose on the contrary that there was an interval $[x_c, y_c]$ in $\mathcal{I}$ that contains $[x, y]$. It follows from the definition of $\alpha(y)$ that $c > \alpha(y)$. This means that $x_{\alpha(y)} < x_c \leq x$. On the other hand, from how $\alpha(y)$ is defined we know that $y_{\alpha(y)} \geq y$. Therefore, $[x_{\alpha(y)}, y_{\alpha(y)}]$ contains $[x, y]$, which contradicts the if-condition of the statement.

To prove Statement 3, suppose on the contrary that there was an interval $[x_c, y_c]$ in $\mathcal{I}$ containing $[x, y]$, and that this interval is on the left of $[x_{\alpha(y)}, y_{\alpha(y)}]$. Then, it follows that $y \leq y_c < y_{\alpha(y)}$, which contradicts the definition of $\alpha(y)$. □

A similar observation holds on b—it depends only on x. Formally, given a value $x \in [1, n]$, define $\beta(x)$ as:

– the largest integer $z \in [1, m]$ such that $i_z \leq x$, if such a z exists;
– nil, otherwise.

In other words, $I_{\beta(x)}$ (if exists) is the rightmost interval in $\mathcal{I}$ whose left endpoint is at most x. Then, we have:

Lemma 7. *Fix an integer $x \in [1, n]$. For any $y \in [x, n]$, all the following are true:*

1. *If $\beta(x) = nil$, then $\mathcal{I}$ has no interval containing $[x, y]$.*
2. *If $I_{\beta(x)}$ does not contain $[x, y]$, then $\mathcal{I}$ has no interval containing $[x, y]$.*
3. *If $I_{\beta(x)}$ contains $[x, y]$, then it is the rightmost interval in $\mathcal{I}$ containing $[x, y]$.*

Proof. Symmetric to the proof of Lemma 6. □

Figure 3 demonstrates all the $\alpha(y)$ and $\beta(x)$ values for the $\mathcal{I}$ of our running example shown in Equation 3. Using the two arrays, we can figure out in $O(1)$ time the values of a and b for any $[x, y]$ (recall that I_a and I_b are the leftmost and rightmost intervals of $\mathcal{I}$ containing $[x, y]$, respectively) using the previous two lemmas. Consider, for example, $x = 4$ and $y = 5$. Probing the α array gives us $\alpha(y) = 2$. Since $I_2 = [3, 6]$ contains $[x, y]$, we conclude from Lemma 6 that $a = 2$. Probing the β array gives us $\beta(x) = 2$. We thus conclude from Lemma 7 that $b = 2$.

Arrays α and β are all we need to complete our structure. Their space consumption is clearly $O(n)$. Furthermore, it is fundamental to compute them in $O(n)$ time.

Algorithm 1 elaborates on the computation of α, whereas we omit the algorithm for β due to symmetry.

The above discussion results in our final query algorithm as shown in Algorithm 2. It is easy to see that the query time is $O(1)$.

4 Extensions

In this section, we discuss several extensional issues. First, in Sections 4.1 and 4.2, we will explain how to use the structure of Theorem 1 (without any modification) to answer two other useful queries, thus further demonstrating the power of our techniques. Then, Section 4.3 will present the I/O-efficient counterpart of Theorem 1.

4.1 Position Constrained Queries

In our current definition, the result of a shortest unique query can start and end anywhere in the data string D. Next, we formulate a variant where a query can specify the permissible ranges for the endpoints of its result:

Position Constrained Query. Such a query specifies (i) a substring $q = D[x : y]$, and (ii) two ranges $r_{start} = [s_1, s_2]$ and $r_{end} = [e_1, e_2]$ both in the domain $[1, n]$. It returns (if exists) a substring $D[i : j]$ with the minimum length such that

- $D[i : j]$ is unique
- $D[i : j]$ contains q
- $i \in [s_1, s_2]$ and $j \in [e_1, e_2]$.

Since $i \leq x$ and $j \geq y$ must always hold, it suffices to consider that $s_2 \leq x$ and $e_1 \geq y$.

For example, in Figure 1, consider a query with $q = D[4 : 5]$ (as shown) and $r_{start} = [3, 4]$ and $r_{end} = [8, 9]$. Then, $D[3 : 6]$ is no longer a legal answer because its right endpoint is not in r_{end}. Instead, the query should return $D[4 : 8] = \texttt{abaab}$.

Queries with $s_2 = x$ and $e_1 = y$. Let us first consider a special class of position constrained queries, where s_2 and e_1 always equal x and y, respectively. Interestingly, any query outside the class actually has the same result as a query inside the class, as explained later. Thus, solving this class of queries is the key.

Lemma 8. *Consider a position constrained query with $q = D[x : y]$, $r_{start} = [s, x]$, and $r_{end} = [y, e]$. Then:*

– *If $D[s : e]$ is repeating, the query has no result.*
– *Otherwise, the result is the shortest of the following 4 candidates:*
  1. *$D[x : y]$ if it is unique;*
  2. *$MUS_{leftfix}(x)$ if its right endpoint is in r_{end};*
  3. *$MUS_{rightfix}(y)$ if its left endpoint is in r_{start};*
  4. *The shortest MUS (breaking ties arbitrarily) that (i) contains q, (ii) has its left endpoint in r_{start}, and (iii) has its right endpoint in r_{end}. No such candidate exists if no MUS satisfies these conditions.*

Proof. The lemma's correctness follows from an argument almost identical to the one we used to prove Lemma 3. □

With our experience with Lemma 3, it should be quite clear that we only need to clarify how to find Candidate 4, because all the other candidates and the necessary uniqueness checking can be done in $O(1)$ time under the $O(n)$ space budget. Furthermore, it is easy to see that the task of finding Candidate 4 boils down to the following problem:

> **Position Constrained Containment Min (PCCM).** Let $\mathcal{I}$ be a set of $m \leq n$ intervals in the domain $[1, n]$ such that no two intervals contain each other (a requirement inherited from Lemma 4). Given intervals $[x, y]$, $[s, x]$, $[y, e]$ all in the domain $[1, n]$, a PCCM query returns the shortest interval in $\mathcal{I}$ (breaking ties arbitrarily) that (i) contains $[x, y]$, (ii) has its left endpoint in $[s, x]$, and (iii) has its right endpoint in $[y, e]$. We want to store $\mathcal{I}$ in a data structure to answer such queries efficiently.

The structure we need is exactly the one described in Section 3.2 for solving the containment min query, namely, the α and β arrays, and an RMQ index. A PCCM query is also answered by a single RMQ, which fetches the shortest interval in $\{I_a, I_{a+1}, ..., I_b\}$ for a pair of a and b carefully chosen as follows[2]:

$$a = \begin{cases} \alpha(y) & \text{if } s = 1 \text{ or } \beta(s-1) = \mathit{nil} \\ \mathit{nil} & \text{if } \beta(s-1) = m \\ \max\{\beta(s-1)+1, \alpha(y)\} & \text{otherwise} \end{cases}$$

$$b = \begin{cases} \beta(x) & \text{if } e = n \text{ or } \alpha(e+1) = \mathit{nil} \\ \mathit{nil} & \text{if } \alpha(e+1) = 1 \\ \min\{\alpha(e+1)-1, \beta(x)\} & \text{otherwise} \end{cases}$$

These values ensure that

- if $a = \mathit{nil}$ or I_a does not cover $[x, y]$, then the PCCM query has no answer;
- otherwise, $\{I_a, I_{a+1}, ..., I_b\}$ includes all and only the intervals of $\mathcal{I}$ containing $[x, y]$ whose left and right endpoints fall in $[s, x]$ and $[y, e]$, respectively.

The PCCM query algorithm is exactly the same as Algorithm 2 except that, at Lines 1 and 2, we should replace a and b with the ones given above.

General Queries. Now we consider position constrained queries with arbitrary $q = D[x : y]$, $r_{start} = [s_1, s_2]$, and $r_{end} = [e_1, e_2]$. As promised, each such query can be converted to one in the special class we have discussed:

Lemma 9. *To answer a positioned constrained query with $q = D[x : y]$, $r_{start} = [s_1, s_2]$, and $r_{end} = [e_1, e_2]$, we can simply return the result of the position constrained query with $q' = D[s_2 : e_1]$, $r'_{start} = [s_1, s_2]$, and $r'_{end} = [e_1, e_2]$.*

[2] We follow the convention that $\max\{v, \mathit{nil}\} = \mathit{nil}$ and $\min\{v, \mathit{nil}\} = \mathit{nil}$ for any integer v.

Proof. The lemma follows from the fact that the answer for the first query must contain $D[s_2 : e_1]$. □

We thus conclude with:

Theorem 2. *Given a data string of length n, we can pre-compute in $O(n)$ time an index structure that consumes $O(n)$ space, and answers any position constrained query in $O(1)$ time.* □

4.2 Find-All Queries

A shortest unique query may have more than one answer. For example, consider again $q = D[4 : 5] =$ `ab` in Figure 1. Besides $D[3 : 6]$, both $D[2 : 5] =$ `bbab` and $D[4 : 7] =$ `abaa` can be returned as a query result. Motivated by this, we define a new operation to retrieve all these possible results:

> **Find-All Query.** Given a substring $q = D[x : y]$, such a query returns *all* the substrings of D whose lengths are the minimum among the unique substrings of D containing q.

We will denote by k the number of substrings returned by a query (e.g., a find-all query with $q = D[4 : 5]$ returns $k = 3$ substrings). Next, we describe an algorithm that answers such a query in $O(k)$ time.

We achieve the purpose using position constrained queries. First, run a (normal) shortest unique query to get an answer string $D[i : j]$. Let $\ell = j - i + 1$ be the length of this string. The value i breaks the interval $[1, x]$ into two disjoint parts: $[1, i - 1]$ and $[i + 1, x]$. Now we can use two position constrained queries to find the next answers, if any. Due to symmetry, it suffices to explain how to do so for $[1, i-1]$. We run a position constrained query with $q' = D[x : y]$, $r_{start} = [1, i - 1]$, and $r_{end} = [y, n]$. A crucial observation is that, if this query returns a string—say $D[i' : j']$—of length *greater* than ℓ, then we can assert that the original find-all query has no result substring that starts within $D[1 : i - 1]$. On the other hand, if $D[i' : j']$ indeed has length ℓ (note that its length cannot be shorter than ℓ), we have found another answer for the find-all query, after which we use i' to break $[1, i - 1]$ into even smaller intervals for recursion.

Algorithm 3 describes the above strategy in detail. To answer a find-all query, simply call FIND-ALL($D[x : y]$, $[1, x]$, ℓ).

Lemma 10. *Our algorithm answers a find-all query in $O(k)$ time.*

Proof. Suppose that the j-th ($1 \leq j \leq k$) answer of the final-all query starts at position i_j, such that $1 \leq i_1 < i_2 < ... < i_k \leq x$. Clearly, these k positions break $[1, x]$ into at most $2k + 1$ disjoint parts: $[1, i_1 - 1]$, i_1, $[i_1 + 1, i_2 - 1]$, ..., i_k, $[i_k + 1, x]$. Our algorithm issues a position constrained query for each part. The query time then follows from Theorem 2. □

Thus we have proved:

Theorem 3. *Given a data string of length n, we can pre-compute in $O(n)$ time an index structure that consumes $O(n)$ space, and answers any find-all query in $O(k)$ time, where k is the number of substrings reported.*

```
Algorithm 3: FIND-ALL $(D[x:y], [s_1, s_2], \ell)$
  Input: $D[x:y]$ is a query substring, $[s_1, s_2]$ is an interval in the domain $[1, n]$, and $\ell$ is
         the length of the shortest unique substrings containing $D[x:y]$.
  Output: All the shortest unique substrings containing $q$ whose left endpoints are in
          $[s_1, s_2]$.
1 run a position constrained query with $q = D[x:y]$, $r_{start} = [s_1, s_2]$, and $r_{end} = [y, n]$
2 if the query returns nil then
3     return $\emptyset$
4 $D[i:j] \leftarrow$ the string returned by the query
5 if the length of $D[i:j] > \ell$ then
6     return $\emptyset$
7 $S_1 \leftarrow$ FIND-ALL$(D[x:y], [s_1, i-1], \ell)$
8 $S_2 \leftarrow$ FIND-ALL$(D[x:y], [i+1, s_2], \ell)$
9 return $\{D[i:j]\} \cup S_1 \cup S_2$
```

4.3 External Memory

The previous discussion has concentrated on the RAM model. In this section, we consider shortest unique queries in the standard *external memory* (EM) model [1]. Under this model, the machine is equipped with a disk that is formated into *blocks* of size B words, and with internal memory of $M \geq 2B$ words. An I/O exchanges a block of data between the disk and memory. The *space* of a structure is measured by the number of disk blocks it occupies, and the *time* of an algorithm is measured by the number of I/Os it performs.

The structure of Theorem 1 works *directly* in external memory. This means that one can simply store the structure by treating the disk as virtual memory. Given that the structure uses $O(n)$ words, the number of blocks it occupies is $O(n/B)$, where B is the number of words in a block. To answer a shortest unique query, one can simply apply the algorithm of Theorem 1 by again treating the disk as virtual memory. As the algorithm performs only $O(1)$ CPU calculation and probes $O(1)$ memory locations, its I/O cost is definitely bounded by $O(1)$.

Our structure can also be constructed efficiently. Remember that it has the following components:

– The MUS_{leftfix} and MUS_{rightfix} arrays (see Figure 2)
– The α and β arrays (Figure 3)
– An RMQ structure.

Both the MUS_{leftfix} and MUS_{rightfix} arrays can be built using the algorithm of [7] in $O(SORT(n))$ I/Os, provided that a *suffix array* [8] is given, where $O(SORT(n))$ is the number of I/Os needed to sort n elements. The suffix array itself can also be computed in $O(SORT(n))$ I/Os [3]. After the MUS_{leftfix} and MUS_{rightfix} arrays are ready, we can then obtain the set $\mathcal{M}$ of MUS's, sorted by left endpoint, in $O(SORT(n))$ I/Os. Then, the α and β arrays can be built using Algorithm 1 in $O(n/B)$ I/Os. An RMQ structure can also be created from $\mathcal{M}$ in $O(n/B)$ I/Os [2].

We now conclude with the last main result of this paper:

Theorem 4. *Given a data string of length n, we can pre-compute in $O(SORT(n))$ I/Os an index structure in external memory that occupies $O(n/B)$ blocks, and answers any shortest unique query in $O(1)$ I/Os.*

Acknowledgements

Xiaocheng and Yufei Tao were supported in part by projects GRF 4165/11, 4164/12, and 4168/13 from HKRGC. Jian Pei was supported by an NSERC Discovery grant and a BCIC NRAS Team project.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31(9):1116–1127, 1988.
2. E. D. Demaine, G. M. Landau, and O. Weimann. On cartesian trees and range minimum queries. In *ICALP*, pages 341–353, 2009.
3. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12, 2008.
4. J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *CPM*, pages 36–48, 2006.
5. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. of Comp.*, 13(2):338–355, 1984.
6. A. M. Ileri, M. O. Külekci, and B. Xu. Shortest unique substring query revisited. In *CPM*, pages 172–181, 2014.
7. L. Ilie and W. F. Smyth. Minimum unique substrings and maximum repeats. *Fundam. Inform.*, 110(1-4):183–195, 2011.
8. U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. of Comp.*, 22(5):935–948, 1993.
9. J. Pei, W. C.-H. Wu, and M.-Y. Yeh. On shortest unique substring queries. In *ICDE*, pages 937–948, 2013.
10. K. Tsuruta, S. Inenaga, H. Bannai, and M. Takeda. Shortest unique substrings queries in optimal time. In *SOFSEM*, pages 503–513, 2014.