On Stabbing Queries for Generalized Longest Repeat

Bojian Xu
Department of Computer Science, Eastern Washington University, WA99004, USA.
bojianxu@ewu.edu

Abstract—A longest repeat query on a string, motivated by its applications in many subfields including computational biology, asks for the longest repetitive substring(s) covering a particular string position (point query). In this paper, we extend the longest repeat query from point query to interval query, allowing the search for longest repeat(s) covering any position interval, and thus significantly improve the usability of the solution. Our method for interval query takes a different approach using the insight from a recent work on shortest unique substrings [1], as the prior work’s approach for point query becomes infeasible in the setting of interval query. Using the critical insight from [1], we propose an indexing structure, which can be constructed in the optimal $O(n)$ time and space for a string of size $n$, such that any future interval query can be answered in $O(1)$ time. Further, our solution can find all longest repeats covering any given interval using optimal $O(occ)$ time, where $occ$ is the number of longest repeats covering that given interval, whereas the prior work’s approach for point query becomes infeasible in this setting. Experiments with real-world biological data show that our proposal is competitive with prior works, both time and space wise, while providing with the new functionality of interval queries as opposed to point queries provided by prior works.

Keywords—string, repeats, longest repeats, stabbing query

I. INTRODUCTION

Repetitive structures and regularity finding in genomes and proteins is important as these structures play important roles in the biological functions of genomes and proteins [2]. One of the well-known features of DNA is its repetitive structure, especially in the genomes of eukaryotes. Examples are that overall about one-third of the whole human genome consists of repeated substrings [3]; about 10–25% of all known proteins [4] captures all the repeats of the whole string in a space-efficient manner. Maximal repeat finding over multiple strings and its duality with minimum unique substrings [6], [7], [8] is an important as these structures play important roles in the notion of maximal repeat and super maximal repeat [2], [6], [7].

II. PROBLEM STATEMENT

We consider a string $S[1..n]$, where each character $S[i]$ is drawn from an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. A substring $S[i..j]$ of $S$ represents $S[i]|S[i+1] \ldots |S[j]$ if $1 \leq i \leq j \leq n$, and is an empty string if $i > j$. String $S[i..j']$ is a proper substring of another string $S[i..j]$ if $i \leq i' \leq j' \leq j$ and $j' - i' < j - i$. The length of a non-empty substring $S[i..j]$, denoted as $|S[i..j]|$, is $j - i + 1$. We define the length of an empty string as zero. A prefix of $S$ is a substring $S[1..i]$ for some $1 \leq i \leq n$. A proper prefix $S[1..i]$ is a prefix of $S$ where $i < n$. A suffix of $S$ is a substring $S[i..n]$ for some $1 \leq i \leq n$. A proper suffix $S[i..n]$ is a suffix of $S$ where $i > 1$. We say character $S[i]$ occupies the string position $i$. We say substring $S[i..j]$ covers the position interval $[x..y]$ of $S$, if $i \leq x \leq y \leq j$. In the case $x = y$, we say substring $S[i..j]$ covers the position $x$ (or $y$) of string $S$. For two strings $A$ and $B$, we write $A = B$ (and say $A$ is equal to $B$), if $|A| = |B|$ and $A[i] = B[i]$ for $i = 1, 2, \ldots, |A|$. We say $A$ is lexicographically smaller than $B$, denoted as $A < B$, if (1) $A$ is a proper prefix of $B$, or (2) $A[1] < B[1]$, or (3) there exists an integer $k > 1$ such that $A[i] = B[i]$ for all $1 \leq i \leq k - 1$ but $A[k] < B[k]$. A substring $S[i..j]$ of $S$ is unique, if there does not exist another substring $S[i'..j']$ of $S$, such that $S[i..j] = S[i'..j']$ but $i \neq i'$. A character $S[i]$ is a singleton, if it is unique. A substring is a repeat if it is not unique.

Definition 1. A longest repeat (LR) covering string position...
interval \([x..y]\), denoted as \(LR^x_y\), is a repeat substring \(S[i..j]\), such that: (1) \(i \leq x \leq y \leq j\), and (2) there does not exist another repeat substring \(S[i'..j']\), such that \(i' \leq x \leq y \leq j'\) and \(j' - i' > j - i\).

Obviously, for any string position interval \([x..y]\), if \(S[x..y]\) is not unique, \(LR^x_y\) must exist, because at least \(S[x..y]\) itself is a repeat. Further, there might be multiple choices for \(LR^x_y\). For example, if \(S = abcabcabc\), then \(LR^1_2\) can be either \(S[1..3] = abc\) or \(S[2..4] = bca\).

Problem (generalized stabbing LR query). Given a string position interval \([x..y]\), \(1 \leq x \leq y \leq n\), find all choices of \(LR^x_y\) or the fact that it does not exist.

We call the generalized stabbing LR query as *interval query*, which includes the point query as a special case where \(x = y\). All prior works \([13, 12, 14]\) only studied point query. Our goal is to find an efficient mechanism for finding the longest repeats of every possible string position interval.

### III. Prior Work and Our Contribution

In addition to the related work discussed in Section I there were recently a sequence of work on finding shortest unique substrings (SUS) \([1\), \(2\), \(3\), \(4\), \(5\), \(6\), \(7\), \(8\)], of which Hu et al. \([1\) studied the generalized version of SUS finding: Given a string position interval \([x..y]\), \(1 \leq x \leq y \leq n\), find \(SUS^x_y\), the shortest unique substring that covers the string position interval \([x..y]\), or the fact that such \(SUS^x_y\) does not exist.

To the best of our knowledge, no efficient reduction from LR finding to SUS finding is known as of now. That is, given a set of SUSes covering a set of position intervals respectively, it is not clear how to find the set of LRs that cover that same set of position intervals respectively, by only using the string \(S\), the given set of SUSes, and linear (of the set size) time cost for the reduction. The reason behind the hardness of obtaining such an efficient reduction is because simply chopping off one ending character of an SUS does not necessarily produce an LR.

For example: suppose \(S = a..aba..a\) of \(2n + 1\) characters, where every character is a except the middle one is b. Clearly, \(SUS_{n-1} = S[n-1, n+1] = aab\), whereas \(LR_{n-1}^n = S[1..n]\). Given \(SUS_{n-1}\) and \(S\) itself, it is not clear how to find \(LR_{n-1}^n = S[1..n]\) using \(O(1)\) time, without involving other auxiliary data structures (otherwise, the reduction, which is still unknown, can become so complex, making itself no better than a self-contained solution for finding LR, which is what this paper is presenting.).

Due to the overall importance of repeat finding in bioinformatics and the lack of efficient reduction from SUS finding to LR finding, it is our belief that providing and implementing a complete solution for generalized LR finding will be beneficial to the community. In summary, we make the following contributions.

1. We generalize the longest repeat query from point query to interval query, allowing the search for the longest repeat(s) covering any interval of string positions, and thus significantly improve the usability of the solution.

2. Because there are at most \(n\) point queries for a string of size \(n\), all prior works pre-compute and save the results of every possible point query, such that any future point query can be answered in \(O(1)\) time. However, in the setting of interval queries, there are \(\binom{n}{2} + n = \Theta(n^2)\) distinct intervals. It becomes impossible, under the \(O(n)\) time and space budget, to achieve the amortized \(O(1)\) query response time, by pre-computing and storing the longest repeats covering each of the \(\Theta(n^2)\) intervals. Therefore, a different approach is needed. Our approach uses the insight from the work by HU et al. \([1\) that leads us to an indexing structure, which can be constructed using optimal \(O(n)\) time and space, such that, by using this indexing structure, any future interval query can still be answered in \(O(1)\) time. The \(O(n)\) time and space costs are optimal because reading and saving the input string already needs \(O(n)\) time and space.

3. Our work can find all longest repeats covering any given interval using optimal \(O(\text{occ})\) time, where \(\text{occ}\) is the number of the longest repeats covering that interval. However, the work in \([12, 13]\) can only find the leftmost and the rightmost candidate, respectively, and only support point queries. The algorithm in \([14]\) can find all longest repeats covering a string position, but their parallelizable sequential algorithm is sub-optimal in the time cost (\(O(n^2)\), indeed) and only supports point queries as well.

4. We provide a generic implementation of our solution without assuming the alphabet size, making the software useful for the analysis of different types of strings. Experimental study with real-world biological data shows that our proposal is competitive with prior works, both time and space wise, while supporting interval queries in the meantime.

### IV. Preparation

The suffix array \(SA[1..n]\) of the string \(S\) is a permutation of \([1, 2, \ldots, n]\), such that for any \(i\) and \(j\), \(1 \leq i < j \leq n\), we have \(S[SA[i..n]] < S[SA[j..n]]\). That is, \(SA[i]\) is the start position of the \(i\)th suffix in the sorted order of all the suffixes of \(S\). The rank array \(Rank[1..n]\) is the inverse of the suffix array. That is, \(Rank[i] = j\) iff \(SA[j] = i\). The longest common prefix (lcp) array \(LCP[1..n+1]\) is an array of \(n+1\) integers, such that for \(i = 2, 3, \ldots, n\), \(LCP[i]\) is the length of the lcp of the two suffixes \(S[SA[i-1..n]]\) and \(S[SA[i..n]]\). We set \(LCP[1] = LCP[n+1] = 0\)\(1\). The following table shows the suffix array and the lcp array of an example string \(S = \text{mississippi}\).

| \(i\) | \(LCP[i]\) | \(SA[i]\) | suffixes |
|------|---------|---------|---------|
| 1    | 0       | 11      | 1       |
| 2    | 1       | 8       | lppi    |
| 3    | 1       | 5       | ississippi |
| 4    | 4       | 2       | mississippi |
| 5    | 0       | 0       | mississippi |
| 6    | 0       | 10      | pi      |
| 7    | 1       | 9       |ippi     |
| 8    | 0       | 7       | sippi   |
| 9    | 2       | 4       | mississippi |
| 10   | 1       | 6       | ssippi  |
| 11   | 3       | 3       | ssissippi |
| 12   | 0       | -       | -       |

\(^1\)In literature, the lcp array is often defined as an array of \(n\) integers. We include an extra zero at \(LCP[n+1]\) as a sentinel to simplify the description of our upcoming algorithms.
Definition 2. The left-bounded longest repeat (LLR) starting at position \( i \), denoted as \( LLR_i \), is a repeat \( S[i..j] \), such that either \( j = n \) or \( S[k..j + 1] \) is unique.

Clearly, for any string position \( k \), if \( S[k] \) is not a singleton, \( LLR_k \) must exist, because at least \( S[k] \) itself is a repeat. Further, if \( LLR_k \) does exist, it must have only one choice, because \( k \) is a fixed string position and the length of \( LLR_k \) must be as long as possible.

Lemma 1 shows that, by using the rank array and the lcp array of the string \( S \), it is easy to calculate any \( LLR_i \) if it exists or to detect the fact that it does not exist.

**Lemma 1 (\cite{12}).** For \( i = 1, 2, \ldots, n \):
\[
LLR_i = \begin{cases} 
S[i..i + L_i - 1], & \text{if } L_i > 0 \\
\text{does not exist}, & \text{if } L_i = 0
\end{cases}
\]
where \( L_i = \max\{LCP[Rank[i]], LCP[Rank[i] + 1]\} \).

Observe that an LLR can be a substring (proper suffix, therefore not a singleton, \( LLR_i = S[i..j] \), which is a substring of \( S\)). For example, suppose \( S = ababab \). Formally, the neighboring LLRs have the following relationship.

**Lemma 2 (\cite{13}).** \( |LLR_i| \leq |LLR_{i+1}| + 1 \)

**Definition 3.** We say an LLR is useless if it is a substring of another LLR; otherwise, it is useful.

**Lemma 3.** Any existing longest repeat \( LLR_x \), \( 1 \leq x \leq y \leq n \), must be a useful LLR.

**Proof:** (1) We first prove \( LLR_y \) must be an LLR. Assume that \( LLR_y = S[i..j] \) is not an LLR. Note that \( S[i..j] \) is a repeat starting from position \( i \). If \( S[i..j] \) is not an LLR, it means \( S[i..j] \) can be extended to some position \( j' > j \), so that \( S[i..j'] \) is still a repeat and also covers the position interval \( x.y \).

Clearly, for any string position \( k \), if \( S[k] \) is not a singleton, \( LLR_k \) must exist, because at least \( S[k] \) itself is a repeat. Further, if \( LLR_k \) does exist, it must have only one choice, because \( k \) is a fixed string position and the length of \( LLR_k \) must be as long as possible.

**Lemma 2 (\cite{13}).** \( |LLR_i| \leq |LLR_{i+1}| + 1 \)

**Definition 3.** We say an LLR is useless if it is a substring of another LLR; otherwise, it is useful.

**Lemma 3.** Any existing longest repeat \( LLR_x \), \( 1 \leq x \leq y \leq n \), must be a useful LLR.

**Proof:** (1) We first prove \( LLR_y \) must be an LLR. Assume that \( LLR_y = S[i..j] \) is not an LLR. Note that \( S[i..j] \) is a repeat starting from position \( i \). If \( S[i..j] \) is not an LLR, it means \( S[i..j] \) can be extended to some position \( j' > j \), so that \( S[i..j'] \) is still a repeat and also covers the position interval \( x.y \).

That is, \( S[i..j'] \) is already the longest repeat covering the position interval \( x..y \). (2) Further, \( LLR_y \) must be a useful LLR, because if it is a useless LLR, it means there exists another LLR that covers the position interval \( x..y \) but is longer than \( LLR_y \), which contradicts the fact that \( LLR_y \) is the longest repeat that covers the interval \( x..y \).

V. LR FINDING FOR ONE INTERVAL

In this section, we propose an algorithm that takes as input a string position interval and returns the LR(s) covering that interval. The algorithm spends \( O(n) \) time and space per query but does not need any indexing data structure. We present this algorithm here in case the practitioners have only a small number of interval queries of their interest and thus this lightweight algorithm will suffice. We start with the finding of the leftmost LR covering the given interval and will give a trivial extension in the end for finding all LRs covering the given interval.

**Lemma 4.** For any \( i, j, x, \) and \( y, 1 \leq i < j \leq x \leq y \leq n \): If \( LLR_i \) does not exist or exists but does not cover the interval \( x..y \), \( LLR_j \) does not exist or does not cover \( x..y \).

**Algorithm 1:** Find the leftmost \( LLR_x \) covering a given string position interval \( [x..y] \).

**Input:** (1) Two integers \( x \) and \( y \), \( 1 \leq x \leq y \leq n \), representing a string position interval \( [x..y] \). (2) The rank array and the lcp array of the string \( S \).

**Output:** The leftmost \( LLR_x \) or the fact that \( LLR_x \) does not exist.

1. \( \text{start} \leftarrow -1; \text{end} \leftarrow -1; // \text{start and end positions of } LLR_x \)
2. for \( i = x \) down to 1 do
3. \( \text{end} \leftarrow \max\{\text{LCP}[\text{Rank}[i]], \text{LCP}[\text{Rank}[i] + 1]\}; // |LLR_i| \)
4. if \( \text{L} = 0 \) or \( i + \text{L} < \text{y} \) then break; // Early stop
5. else if \( \text{L} \geq \text{end} - \text{start} + 1 \) then // Pick the leftmost one
6. \( \text{start} \leftarrow i; \text{end} \leftarrow i + \text{L} - 1 \)
7. \( \text{return } LLR_x \leftarrow (\text{start}, \text{length}) \).

**Proof:** We prove the lemma by contradiction. (1) Assume it is possible that when \( LLR_i \) does not cover the interval \( [x..y] \), \( LLR_i \) can still cover \( [x..y] \). Say, \( LLR_i = S[i..k] \) for some \( k \geq y \). It follows that \( S[j..k] \) is also a repeat and covers \( [x..y] \), which is a contradiction, because \( LLR_j \), the longest repeat starting from string location \( j \), does not cover \( [x..y] \). (2) Assume it is possible that when \( LLR_i \) does not exist, \( LLR_i \) can still cover \( [x..y] \). Say, \( LLR_i = S[i..k] \) for some \( k \geq y \). It follows that \( S[j..k] \) is also a repeat and covers \( [x..y] \), which is a contradiction, because \( LLR_j \) does not exist at all, i.e., \( S[j] \) is a singleton.

By Lemma 3 we know any LR must be an LLR, so we can find \( LLR_x \) covering a given interval \( [x..y] \) by simply checking each \( LLR_i, i \leq x \), and picking the longest one that covers the interval \( [x..y] \). Ties are resolved by picking the leftmost choice. Because of Lemma 3 early stop is possible to make the procedure faster in practice by checking every \( LLR_i \) in the decreasing order of the value of \( i = x, x - 1, \ldots, 1 \); the search will stop whenever we see an \( LLR_i \) that does not cover the interval \( [x..y] \) or does not exist at all. Algorithm 1 shows the pseudocode, which returns \((\text{start}, \text{end})\), representing the start and ending positions of \( LLR_x \), respectively. If \( LLR_x \) does not exist, \((-1, -1)\) is returned.

**Lemma 5.** Given the rank array and the lcp array of the string \( S \), for any string position interval \( [x..y] \), Algorithm 1 can find \( LLR_x \) or the fact that it does not exist, using \( O(x) \) time and \( O(n) \) space. If there are multiple choices for \( LLR_x \), the leftmost one is returned.

**Proof:** The algorithm clearly has no more than \( x \) iterations and each iteration takes \( O(1) \) time, so it costs \( O(x) \) time. The space cost is primarily from the rank array and the lcp array, which altogether is \( O(n) \), assuming each integer in these arrays costs a constant number of memory words. If multiple LRs cover position interval \( [x..y] \), the leftmost LR will be returned, as is guaranteed by Line 5 of Algorithm 1.

**Theorem 1.** For any position interval \( [x..y] \) in the string \( S \), we can find \( LLR_x \) or the fact that it does not exist using \( O(n) \) time and space. If there are multiple choices for \( LLR_x \), the leftmost one is returned.

**Proof:** The suffix array of \( S \) can be constructed by existing algorithms using \( O(n) \) time and space (e.g., \cite{19}). After the suffix array is constructed, the rank array can be trivially created using another \( O(n) \) time and space. We can then use the suffix array and the rank array to construct the
Lemma 6. Given the Lcp and rank array of the string S, we can compute the useful LLRs in O(n) time and space.

Proof. By Lemma 5, we know if LLR does not exist, the right boundary of LLR is on or can be computed as the right boundary of the Lcp array using the Lcp array and the Lcp array, the time cost of Algorithm 1 is easy to compute.

In this section, we present a geometric perspective of the useful LLRs and the LR queries. This perspective is inspired by the ideas presented in [11, 12], which applies the intuition behind the algorithms in Sections VII and VIII to solve the problem effectively. We refer to the same spirit of the algorithm in Lemma 1 for a quick and easy way.

Algorithm 1. Find all LLRs covering a given position interval.

Output: All LLRs that cover the position interval [i, j] or the fact that such LLRs do not exist.

Input: Two integers i and j, 1 ≤ i ≤ j ≤ n, representing a string of length n and a position interval [i, j].

1. Find the length of LLR.
2. Compute the rank array Rank[1...n] of the string S.
3. Compute the length of the Lcp array Lcp[1...n] of the string S.
4. Compute the Lcp array Lcp[1...n] of the string S.
5. Find all LLRs that cover position interval [i, j].
6. Return the result.

Algorithm 2. Find all LLRs that cover a given string.

Output: All LLRs that cover the string S.

Input: A string S of length n.

1. Compute the rank array Rank[1...n] of the string S.
2. Compute the length of the Lcp array Lcp[1...n] of the string S.
3. Find all LLRs that cover the string S.
4. Return the result.

Algorithm 3. The calculation of LLR, the array of useful LLRs.

Output: The array of useful LLRs.

Input: A string S of length n.

1. Compute the rank array Rank[1...n] of the string S.
2. Compute the length of the Lcp array Lcp[1...n] of the string S.
3. Find all LLRs that cover the string S.
4. Return the result.

Definition 4. A LR is an array of useful LLR, which are sorted in the ascending order of their start position.

Algorithm 4 shows the procedure for the LLR array and the array of useful LLRs. For each useful LR, we have the following fact.

Fact 1. All elements in the LLR array have their start position as the x coordinate and their end position as the y coordinate.

Fig. 1. The 2d geometric perspective on the useful LLRs of string S = ababababababababababababababababababababababababababababababababababababababababababababababababababababababababababababababababababababababababababa.
that by using the indexing structure any future generalized LR queries can be answered in $O(\log n)$ time. If there exist multiple choices for the LR of interest, ties are resolved arbitrarily.

Proof: (1) The suffix array of $S$ can be constructed by existing algorithms using $O(n)$ time and space (e.g., [19]). After the suffix array is constructed, the rank array can be trivially created using $O(n)$ time and space. We can then use the suffix array and the rank array to construct the lcp array using another $O(n)$ time and space ([20]). (2) Given the suffix array and the lcp array, we can construct the $LLRc$ array of useful LLRs using $O(n)$ time and space (Lemma 6 and Algorithm 3). (3) We then create the indexing structure for the $LLRc$ array elements for 2d DMQ, using $O(n \log n)$ time and $O(n)$ space (e.g., [21]). By using this index, we can answer any future generalized LR query in $O(\log n)$ time and ties are resolved arbitrarily.

A. Find all choices of any LR

We know $LR^w_y$ are the heaviest dots in $S_{x,y}$, if $S_{x,y}$ is not empty; otherwise, $LR^w_y$ does not exist. Upon receiving a query for $LR^w_y$, we first perform a 2d DMQ, which returns one of the heaviest dots in $S_{x,y}$. If no such a dot is returned, then $LR^w_y$ does not exist. Otherwise, suppose $(x',y')$ is the dot returned, then $(x',y')$ is one of the choices for $LR^w_y$.

Because all the dots representing the $LLRc$ array elements have their both $x$ and $y$ coordinates strictly increase (Fact 1), all other choices (if existing) of $LR^w_y$ must be existing in the union of $S_{x'-y,y}$ and $S_{x,y'+1}$. Therefore, we can find other choices of $LR^w_y$ by the following two recursive searches: one will find one of the heaviest dots in $S_{x'-y,y}$, the other will find one of the heaviest dots in $S_{x,y'+1}$. Each of these two recursive searches is again a 2d DMQ.

For each recursive search: (1) If the weight of the heaviest dot it finds is equal to $y' - x' + 1$, the length of $LR^w_y$, it will return the found dot as another choice of $LR^w_y$ and will then launch its own two new recursive searches, similar to what its caller has done in order to find other choices for $LR^w_y$; (2) otherwise, it stops and returns to its caller.

Function $QueryAll2d$ in Algorithm 4 shows the pseudocode for finding all choices of $LR^w_y$.

Example 1 (Figure 1). Search $A$ is for $LR^w_{11,12}$. That is to find all heaviest dots in $S_{11,12}$, which include dot $(7,13)$ and dot $(11,17)$. Suppose the 2d DMQ launched by search $A$ returns dot $(7,13)$, which has a weight of 7 and is one choice for $LR^w_{11,12}$. The next two recursive searches launched by search $A$ will be search $B$ looking for one of the heaviest dots in $S_{11,13}$ and search $C$ looking for one of the heaviest dots in $S_{6,12}$.

Search $B$ will return the heaviest dot $(11,17)$ from $S_{11,12}$, whose weight is equal to 7, so the dot $(11,17)$ is another choice of $LR^w_{11,12}$. Search $B$ will then launch its own two new recursive searches for one heaviest dot in each of $S_{10,14}$ and $S_{11,18}$. (These two searches are not shown in Figure 1 for concision). The search in $S_{10,14}$ returns dot $(10,14)$ whose weight is less than 7, so the search stops and returns to its caller. The search in $S_{11,18}$ finds nothing, so it stops and returns to its caller. After all its recursive searches return, search $B$ returns to its caller, which is search $A$.
Search \( \mathcal{C} \) finds nothing in \( S_{6,12} \), so it stops and returns to its caller, which is search \( \mathcal{A} \).

At this point, all the work of search \( \mathcal{A} \) is finished, and we have found all the choices, which are \([S_{7,13}] \) and \([S_{11,17}] \) (or \( LLRc[3] \) and \( LLRc[5] \), equivalently), for \( LR_{12}^{11} \).

Clearly, the same 2d DMQ index is used in finding all choices of an LR query, and there are no more than \( 2 \cdot occ + 1 \) instances of 2d DMQ, in the finding of all choices of an LR, where \( occ \) is the number of choices of the LR. Because each 2d DMQ takes \( O(\log n) \) time, we get the following theorem.

**Theorem 4.** We can construct an indexing structure for a string \( S \) of size \( n \) using \( O(n \log n) \) time and \( O(n) \) space, such that by using the indexing structure, we can find all choices of any LR in \( O(occ \cdot \log n) \) time, where \( occ \) is the number of choices of the LR being queried for.

**VIII. AN INDEX OF \( O(occ) \) QUERY TIME**

In this section, we present the optimal indexing structure for generalized LR finding. It is again based on the intuition derived from the geometric perspective on the relationship between useful LLRs and LR queries (Section VI).

Recall that the answer for an \( LR_y^x \) query is the heaviest dot(s) from \( S_{x,y} \) if \( S_{x,y} \) is not empty. Due to Fact 1, \( S_{x,y} \) corresponds to a continuous chunk of the \( LLRc \) array, if \( S_{x,y} \) is not empty. Therefore, searching for one heaviest dot in \( S_{x,y} \) becomes searching for one heaviest element within a continuous chunk of the \( LLRc \) array, which is nothing but the range minimum query on the array \( LLRc \)

**Range minimum query (RMQ).** Given an array \( A[i..n] \) of \( n \) comparable elements, find the index of the smallest element within \( A[i..j] \), for any given \( i \) and \( j \), \( 1 \leq i \leq j \leq n \). If there are multiple choices, ties are resolved arbitrarily.

There exist indexing structures (e.g., [22], [23]) that can be constructed on top of the array \( A \) using \( O(n) \) time and space, such that any future RMQ can be answered in \( O(1) \) time.

The next issue is: Upon receiving a query for \( LR_y^x \), for some \( x \) and \( y \), \( 1 \leq x \leq y \leq n \), how to find the left and right boundaries of the continuous chunk of \( LLRc \), over which we will perform an RMQ? Due to Fact 1 and with the aid of the geometric perspective of the useful LLRs, we can observe that the left boundary of the chunk only depends on the value of \( y \), whereas the right boundary of the chunk only depends on the value of \( x \). Intuitively, if one sweeps a horizontal line starting from position \( y \) (inclusive) toward the up direction, the \( LLRc \) array index of the first dot that the line hits is the left boundary of the RMQ’s range. Similarly, if one sweeps a vertical line starting from position \( x \) (inclusive) toward the left direction, the \( LLRc \) array index of the first dot that the line hits is the right boundary of the RMQ’s range. The range for RMQ is invalid, if any one of the following three possibilities happens: 1) No dot is hit by the horizontal line; 2) No dot is hit by the vertical line; 3) The index of the left boundary of the range is larger than the index of the right boundary of the range. An invalid RMQ range means that \( LR_y^x \) does not exist. See Figure 1 for examples.

More precisely, given the values of \( x \) and \( y \) from the query for \( LR_y^x \), the left boundary \( L_y \) and the right boundary \( R_x \) of the range for RMQ can be determined as follows:

\[
L_y=\begin{align*}
\min\{i & \mid LLRc[i].end \geq y\}, & & \text{if } \{i \mid LLRc[i].end \geq y\} \neq \emptyset \\
-1, & & \text{otherwise}
\end{align*}

\[
R_x=\begin{align*}
\max\{i & \mid LLRc[i].start \leq x\}, & & \text{if } \{i \mid LLRc[i].start \leq x\} \neq \emptyset \\
-1, & & \text{otherwise}
\end{align*}
\]

Further, we can pre-compute \( L_y \) and \( R_x \), for every \( x = 1, 2, \ldots, n \) and \( y = 1, 2, \ldots, n \), and save the results for future references. Algorithm 5 shows the procedure for computing the \( L \) and \( R \) arrays, which clearly uses \( O(n) \) time and space.

**Lemma 7.** Algorithm 5 computes \( L_1, L_2, \ldots, L_n \) and \( R_1, R_2, \ldots, R_n \) using \( O(n) \) time and space.

Now we are ready to present the algorithm for finding one choice of a generalized LR query. Algorithm 6 (through Line 7) gives the pseudocode. After array \( LLRc \) is created, we will compute the \( L \) and \( R \) arrays using the \( LLRc \) array (Algorithm 5). Then we will create the RMQ structure for the \( LLRc \) array, where the weight of each array element is defined as the length of the corresponding LLR (or, from the geometric perspective, is the weight of the 2d dot representing that LLR), as the length of the corresponding LLR (or, from the geometric perspective, is the weight of the 2d dot representing that LLR), as the length of the corresponding LLR (or, from the geometric perspective, is the weight of the 2d dot representing that LLR), as the length of the corresponding LLR (or, from the geometric perspective, is the weight of the 2d dot representing that LLR).

**Theorem 5.** We can construct an indexing structure for a string \( S \) of size \( n \) using \( O(n) \) time and space, such that any future generalized LR query can be answered in \( O(1) \) time. If there exist multiple choices for the LR being queried for, ties are resolved arbitrarily.

**Proof:** (1) The suffix array of \( S \) can be constructed by existing algorithms using \( O(n) \) time and space (e.g., [19]). After the suffix array is constructed, the rank array can be trivially created using \( O(n) \) time and space. We can then use the suffix array and the rank array to construct the lcp array using another \( O(n) \) time and space (20). (2) Given the rank

---

\[1\] We should actually perform range maximum query, which however can be trivially reduced to RMQ by viewing each array element as the negative of its actual value.

---

**Algorithm 5: Compute \( L_i \) and \( R_i \) for \( i = 1, 2, \ldots, n \).**

**Input:** The \( LLRc \) array.

**Output:** The \( L \) and \( R \) arrays.

1. for 1 = 1, 2, ..., n do \( L_i \leftarrow -1 \); \( R_i \leftarrow -1 \); // Initialization.
2. \( i \leftarrow 1 \);
3. for \( y = 1 \ldots n \) do
4. if \( y = LLRc[1] \).end then \( L_y \leftarrow i \);
5. else if \( i < LLRc[size] \) then \( i \leftarrow i + 1 \); \( L_y \leftarrow i \);
6. else break;
7. \( i \leftarrow LLRc.size \);
8. for \( x = n \ldots 1 \) do
9. if \( x \geq LLRc[x]. \)start then \( R_x \leftarrow i \);
10. else if \( i > 1 \) then \( i \leftarrow i - 1 \); \( R_x \leftarrow i \);
11. else break;
3. Construct the RMQ structure for the LRC array: // Algorithm 2

/* Find one choice of LR

4. Query one RMQ(x, y): if L \neq 1 and R \neq 1 and L_y \leq R_x then

6. return LRC[RMQ(LRC[L_y..R_x])] // Recursion exits.

7. else return (-1, -1); // LR^u_y does not exist.

8. QueryAllRMQ(x, y): if L \neq 1 and R \neq 1 and L_x \leq R_y then

10. m ← RMQ(LRC[L_x..R_y]);
11. weight ← LRC[m].end − LRC[m].start + 1; // |LR^u_y|
12. FindAllRMQ(L_y, R_x, weight): // Recursive searches start.
13. else return (-1, -1); // LR^u_y does not exist.

14. FindAllRMQ(ℓ, r, weight): // Helper function
15. m ← RMQ(LRC[ℓ, r]);
16. if LRC[m].end − LRC[m].start + 1 < weight then
17. return; // Recursion exits.
18. Print LRC[m]; // One choice of LR^u_y is found.
19. if ℓ < m − 1 then
20. FindAllRMQ(ℓ, m − 1, weight); // New recursive search.
21. if r > m + 1 then
22. FindAllRMQ(m + 1, r, weight); // New recursive search.

A. Find all choices of any LR.

Upon receiving a query for LR^u_y, we first perform an RMQ over range LRC[L_y..R_x] if such range exists; otherwise, it means LR^u_y does not exist, and we stop. Suppose the range LRC[L_y..R_x] is valid and its RMQ returns m, the array index of the heaviest element in the range, then LRC[m] is one of the choices for LR^u_y and |LR^u_y| = LRC[m].end − LRC[m].start + 1. If LR^u_y has other choices, those choices must be existing in the union of the ranges LRC[L_y..m − 1] and LRC[m + 1..R_x]. We can find those choices of LR^u_y by recursively performing an RMQ on each of those two ranges. The recursion will exit, if the element returned by RMQ has a weight smaller than |LR^u_y| or the range for RMQ is invalid. The QueryAllRMQ function in Algorithm 6 shows the pseudocode of this procedure for finding all choices of an LR query.

Example 2 (Figure 1). Given the LRC array of the example string in Figure 1, Algorithm 5 computes the L and R arrays.

IX. IMPLEMENTATION AND EXPERIMENTS

We implement our proposals in C++, using the library binary of the implementation of the DMQ and RMQ structures from libdivsufsort. Our implementation is generic in that it does not assume the alphabet size of the underlying string, and thus supports LR queries over different types of strings.

We compare the performance of our proposals with the prior works including the optimal O(n) time and space solution from [22] and the suboptimal sequential algorithm presented in [14]. Note that all prior works can only answer point queries. All programs involved in the experiments use the same libdivsufsort library for the suffix array construction, and are compiled by gcc 4.7.2 with -O3 option.

We conduct our experiments on a GNU/Linux machine with kernel version 3.2.51-1. The computer is equipped with an Intel Xeon 2.40GHz E5-2609 CPU with 10MB Smart Cache and has 16GB RAM. All experiments are conducted on 3https://code.google.com/p/libdivsufsort.
real-world datasets including the DNA and Protein strings, downloaded from the Pizza&Chili Corpus. The datasets we use are the two 100MB DNA and Protein pure ASCII text files, each of which thus represents a string of $100 \times 1024 \times 1024 = 104,857,600$ characters. Any other shorter strings involved in our experiments are prefixes of certain lengths, cut from the 100MB strings.

A. Space

Here, we measure the peak memory usage of different proposals, using the Linux command `/usr/bin/time -f "%M"` that captures the maximum resident set size of a process during its lifetime. We do not save the output in the RAM in order to focus on the comparison of the memory usage of the algorithmics. It is also because practitioners often flush the outputs directly to disk files for future reuse.

Figure 2 shows the peak memory usage of different proposals that process DNA and protein strings of different sizes. It is worth noting that, by design, the memory usage of each proposal is independent from the query type, such as finding one choice vs. all choices of an LR, point query vs. interval query. We have the following main observations:

- All proposals show the linearity of their space usage over string size.
- Our DMQ-based proposal uses much more memory space than other proposals. It is mainly caused by the high space demand from the DMQ structure.
- Our RMQ-based proposal uses nearly the same amount of memory space as that of prior works, while significantly improving the usability of the technique by providing the functionality of interval queries.

B. Time

Figure 3 shows the construction time of the indexing structures used by different proposals. Note that all proposals need to construct the suffix array, rank array, and the lcp array of the given string, and our proposals further use these auxiliary arrays to construct the DMQ and RMQ structures for interval queries. The following are the main observations:

- The construction of the DMQ structure takes much more time than that of the auxiliary arrays and the RMQ structure.
– Both the auxiliary array and RMQ structure clearly show the
linearity in their construction time over string size.
– The construction of the RMQ structure takes less time than
the construction of the auxiliary arrays, making our RMQ-
based proposal practical while supporting interval queries.

Figure 4 shows the time cost of various types of query. Our
DMQ-based proposal is so slow in query response that we
do not include it in the figure. For point queries, we plot
the total time cost for all the point queries over all \( n \) string
positions, where \( n \) is the string size. For interval queries with
interval size \( \delta \), we plot the total time cost for all the interval
queries over all \( n - \delta + 1 \) intervals of the string. Note that
only point queries are involved in the experiments with the
proposals from [12] and [14], because they do not support
interval queries. The two figures on the left show the case
for finding only one choice for each LR, whereas the two on
the right show the case for finding all choices for each LR.
Because the proposal from [12] does not support the finding
of all choices, it is not included in the two figures on the right.
The following are the main observations:

– All proposals show the clear linearity of the total query time
cost, meaning the amortized \( O(1) \) time cost for each query.

– In the setting of finding one choice for each LR (the two
figures on the left of Figure 4), our RMQ-based proposal is
the fastest regarding the per-query response time, including
both point query and interval query! Further, our RMQ-based
proposal’s interval query response becomes even faster, when
interval size increases. That is because a longer interval is
covered by fewer number of repeats, reducing the search space
size for finding the LR covering the interval.

– In the setting of finding all choices for each LR (the two
figures on the right side of Figure 4):

  • For point query, our RMQ-based proposal is a little
    slower than [14] due to the following reason. On
    average, an LR point query returns more choices than
    an interval query. Our technique needs to make a
    query to the index for finding every single choice,
    whereas the technique in [14] only needs one extra
    “walk” for finding all choices for a particular LR point
    query. Even though our technique is faster than [14]
    for finding one choice (the two figures on the left),
    when a particular point query has many choice, our
    technique can become slower in finding all choices.

  • As interval size increases, our RMQ-based proposal
    becomes faster, because a longer interval on average
    has fewer choices for its LR, making our technique
    have fewer queries to its index. Our technique’s in-
    terval query can be even faster than the point query.
by \[\text{[14]}\] in finding all choices when interval size increases. For example, it is true, when interval size becomes \(\geq 15\) for DNA string (top-right figure) and \(\geq 5\) for protein string (bottom-right figure).

X. Conclusion

We generalized the longest repeat query on a string from point query to interval query and proposed both time and space optimal solution for interval queries. Our approach is different from prior work which can only handle point queries. Using the insight from \[\text{[1]}\], we proposed an indexing structure that can be built on top of the string using time and space linear of the string size, such that any future interval queries can be answered in \(O(1)\) time. We implemented our proposals without assuming the alphabet size of the string, making it useful for different types of strings. An interesting future work is to parallelize our proposal so as to take advantage of the modern multi-core and multi-processor computing platforms, such as the general-purpose graphics processing units.

References

[1] X. Hu, J. Pei, and Y. Tao, “Shortest unique queries on strings,” in Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE), 2014, pp. 161–172.

[2] D. Gusfield, Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press, 1997.

[3] E. H. McConkey, Human Genetics: The Molecular Revolution. Boston, MA: Jones and Bartlett, 1993.

[4] X. Liu and L. Wang, “Finding the region of pseudo-periodic tandem repeats in biological sequences,” Algorithms for Molecular Biology, vol. 1, no. 1, p. 2, 2006.

[5] H. M. Martinez, “An efficient method for finding repeats in molecular sequences,” Nucleic Acids Research, vol. 11, no. 13, pp. 4629–4634, 1983.

[6] V. Becher, A. Deymonnaz, and P. A. Heiber, “Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome,” Bioinformatics, vol. 25, no. 14, pp. 1746–1753, 2009.

[7] M. O. Kulekci, J. S. Vitter, and B. Xu, “Efficient maximal repeat finding using the burrows-wheeler transform and wavelet tree,” IEEE Transactions on Computational Biology and Bioinformatics (TCBB), vol. 9, no. 2, pp. 421–429, 2012.

[8] T. Beller, K. Berger, and E. Ohlebusch, “Space-efficient computation of maximal and supermaximal repeats in genome sequences,” in Proceedings of the 19th International Conference on String Processing and Information Retrieval (SPIRE), 2012, pp. 99–110.

[9] A. Bakalis, C. S. Iliopoulos, C. Makris, S. Sioutas, E. Theodoridis, A. K. Tsakalidis, and K. Tsichlas, “Locating maximal multirepeats in multiple strings under various constraints,” Computer Journal, vol. 50, no. 2, pp. 178–185, 2007.

[10] C. S. Iliopoulos, W. F. Smyth, and M. Yusufu, “Faster algorithms for computing maximal multirepeats in multiple sequences,” Fundamenta Informaticae, vol. 97, no. 3, pp. 311–320, 2009.

[11] L. Ilie and W. F. Smyth, “Minimum unique substrings and maximum repeats,” Fundamenta Informaticae, vol. 110, no. 1-4, pp. 183–195, 2011.

[12] A. M. Ileri, M. O. Kulekci, and B. Xu, “On longest repeat queries,” http://arxiv.org/abs/1501.06259.

[13] T. Schnattinger, E. Ohlebusch, and S. Gog, “Bidirectional search in a string with wavelet trees and bidirectional matching statistics,” Information and Computation, vol. 213, pp. 13–22, Apr. 2012.

[14] Y. Tian and B. Xu, “On longest repeat queries using GPU,” in Proceedings of the 20th International Conference on Database Systems for Advanced Applications (DASFAA), 2015, pp. 316–333.

[15] J. Pei, W. C. H. Wu, and M. Y. Yeh, “On shortest unique substring queries,” in Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE), 2013, pp. 937–948.

[16] K. Tsurtua, S. Ienaga, H. Bannai, and M. Takeda, “Shortest unique substrings queries in optimal time;,” in Proceedings of International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), 2014, pp. 503–513.

[17] A. M. Ileri, M. O. Kulekci, and B. Xu, “Shortest unique substring query revisited,” in Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM), 2014, pp. 172–181.

[18] A. M. Ileri, M. O. Kulekci, and B. Xu, “A simple yet time-optimal and linear-space algorithm for shortest unique substring queries,” Theoretical Computer Science, vol. 562, no. 0, pp. 621 – 633, 2015.

[19] P. Ko and S. Aluru, “Space efficient linear time construction of suffix arrays,” Journal of Discrete Algorithms, vol. 3, no. 2-4, pp. 143–156, 2005.

[20] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, “Linear-time longest-common-prefix computation in suffix arrays and its applications,” in Symposium on Combinatorial Pattern Matching, 2001, pp. 181–192.

[21] C. Sheng and Y. Tao, “New results on two-dimensional orthogonal range aggregation in external memory,” in Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS), 2011, pp. 129–139.

[22] J. Fischer and V. Heun, “Theoretical and practical improvements on the rmq-problem, with applications to lca and lce,” in Proceedings of the 17th Annual Conference on Combinatorial Pattern Matching (CPM), 2006, pp. 36–48.

[23] D. Harel and R. E. Tarjan, “Fast algorithms for finding nearest common ancestors,” SIAM Journal of Computing, vol. 13, no. 2, pp. 338–355, May 1984.