JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 15, Number 4, 2008

Pp. 357–377
DOI: 10.1089/cmb.2007.0105

Space Efficient Computation of Rare Maximal Exact Matches between Multiple Sequences

ENNO OHLEBUSCH[1] and STEFAN KURTZ[2]

ABSTRACT

In this article, we propose a new method for computing rare maximal exact matches between multiple sequences. A rare match between k sequences $S_1, \ldots, S_k$ is a string that occurs at most t_i-times in the sequence S_i, where the $t_i > 0$ are user-defined thresholds. First, the suffix tree of one of the sequences (the reference sequence) is built, and then the other sequences are matched separately against this suffix tree. Second, the resulting pairwise exact matches are combined to multiple exact matches. A clever implementation of this method yields a very fast and space efficient program. This program can be applied in several comparative genomics tasks, such as the identification of synteny blocks between whole genomes.

Key words: alignment, algorithms, strings, suffix trees.

1. INTRODUCTION

WHOLE GENOME COMPARISONS can be used as a first step toward solving genomic puzzles, such as determining coding regions, discovering regulatory signals, and deducing the mechanisms and history of genome evolution. One aspect that makes computational comparative genomics difficult is the fact that both local and global mutations of the DNA molecules occur during evolution. Local mutations (point mutations) consist of the substitution, insertion, or deletion of single nucleotides, while global mutations (genome rearrangements) change the DNA molecules on a large scale. In unichromosomal genomes, the most common rearrangements are inversions, where a section of the genome is excised, reversed in orientation, and re-inserted. But large-scale duplications, deletions (gene loss), insertions (horizontal gene transfer), and transpositions also play a role. In a transposition, a section of the genome is excised and inserted at a new position in the genome; this may or may not also involve an inversion. In genomes with multiple chromosomes, further genome rearrangements are translocations (in a reciprocal translocation, two non-homologous chromosomes break and exchange fragments), fusions (where two chromosomes fuse), and fissions (where a chromosome breaks into two parts).

Thus, if the organisms under consideration are closely related (that is, if no or only a few genome rearrangements have occurred) or one compares regions of conserved synteny (these are regions in two or more genomes in which orthologous genes occur in the same order), then global alignments can, for example, be used for the prediction of genes and regulatory elements. This is because coding regions

[1]Faculty of Engineering and Computer Sciences, University of Ulm, Ulm, Germany.
[2]Center for Bioinformatics, University of Hamburg, Hamburg, Germany.

are relatively well preserved, while non-coding regions tend to show varying degree of conservation. Non-coding regions that do show conservation are thought important for regulating gene expression and maintaining the structural organization of the genome; they possibly have other, yet unknown functions. Several comparative sequence approaches based on alignments have been used to analyze corresponding coding and non-coding regions from different species. These approaches are based on software tools for aligning DNA-sequences (Chain et al., 2003; Treangen and Messeguer, 2006). To cope with the shear volume of data, most of the software tools use an anchor-based method that is composed of three phases:

1. Computation of fragments (segments in the genomes that are similar).
2. Computation of a highest-scoring global chain of colinear non-overlapping fragments: these are the anchors that form the basis of the alignment.
3. Alignment of the regions between the anchors.

For diverged genomic sequences, however, a global alignment strategy is likely predestined to failure for having to align non-syntenic and unrelated regions in an end-to-end colinear approach. In this case, one must first identify syntenic regions, which then can be studied individually. Moreover, the ordering of such "synteny blocks" can then be used as input to software tools such as MGR (Bourque and Pevzner, 2002) that compute plausible rearrangement scenarios for multiple genomes. In the gene-based approach, the problem of automatically finding syntenic regions requires a priori knowledge of all genes and which of the genes are orthologous. It is safe to say that gene prediction and the accurate determination of orthologous genes are computationally difficult, but it is beyond the scope of this paper to discuss these issues in detail. Pevzner and Tesler (2003) bypassed "the difficult issues of gene annotation and ortholog identification" by using sequenced-based *synteny blocks*. In Abouelhoda and Ohlebusch (2003), it was shown that significant local chains (instead of a highest-scoring global chain as in phase (2) of the anchor-based method) of *multiMEMs* can be used to efficiently find synteny blocks in prokaryotic genomes. However, *multiMEMs* cannot be used for comparing eukaryotic genomes containing many repetitive elements. This is because the number of *multiMEMs* "explodes" in the presence of many repeats. Of course, many repetitive elements can be eliminated by repeat masking tools such as RepeatMasker (Smit and Green, 2008). However, repeat masking takes a long time, does not eliminate all repeats, and causes new problems (not discussed here). Repeat masking can be avoided by using *multiMUMs* instead of *multiMEMs* as fragments. As noted by Mau et al. (2005), however, using *multiMUMs* may fail to generate enough anchors. Consequently, something in between *multiMUMs* and *multiMEMs* is needed, and we found that rare *multiMEMs* meet the requirements. With an appropriately chosen threshold t on the allowed number of copies of a rare *multiMEM*, it is possible to generate sufficiently many anchors while at the same time avoiding an explosion of the number of multiple matches (without repeat masking). For example, when computing *multiMEMs* of minimal length 20, for a set of six *Staphylococcus aureus* genomes, we found that there are 110 times more *multiMEMs* than rare *multiMEMs* ($t = 5$). When comparing the X-chromosomes of four vertebrate genomes, we can easily compute the rare *multiMEMs*, but not the *multiMEMs* (because there are too many). See Section 8 for more details.

A method to compute rare *multiMEMs* is sketched in Abouelhoda et al. (2006).[1] This method is a modification of the following technique to find all *multiMEMs* (Höhl et al., 2002; Kurtz and Lonardi, 2004) among k genomic DNA sequences $S_1, \ldots, S_k$ (in our application, S_i is the sequence of nucleotides in one strand of the DNA double strand of a chromosome of genome G_i). First, the strings $S_1, \ldots, S_k$ are concatenated, using distinct symbols $\$_1, \ldots, \$_{k-1}$ to mark the borders between the strings. Then, one builds the suffix tree (or the enhanced suffix array) of the resulting string $S = S_1\$_1S_2\$_2, \ldots, S_{k-1}\$_{k-1}S_k$ and computes all *multiMEMs* basically by computing all repeats (satisfying some constraints) in the string S. In this way, it is possible to compare chromosomes of several species simultaneously provided that one computes rare *multiMEMs* instead of all *multiMEMs*.

However, for a comparison of multiple genomes, the method is rather time and space consuming. We will clarify this by an example. Suppose one wants to compare the genomes of human, mouse, and rat. The human genome consists of 46 chromosomes: 22 pairs of homologous chromosomes plus the X and Y chromosome (of course, females have two X chromosomes). The mouse genome has 19 pairs of homologous chromosomes, while the rat genome has 20. Therefore, $24 \cdot 21 \cdot 22$ combinations have to be

[1] To be precise, the algorithm searches for *infrequent multiMEMs*, a slight variation of rare *multiMEMs*.

dealt with in a comparison of all leading strands of the chromosomes. Because one also has to take the lagging strands into account, the number of combinations increases to $24 \cdot 42 \cdot 44 = 44{,}352$ (the lagging strands of *one* genome need not be considered because, for example, the comparison of all leading strands of the X chromosomes of human, mouse, and rat is equivalent to the comparison of all lagging strands of these chromosomes). That is, one has to build the suffix tree (or the enhanced suffix array) of 44,352 long strings, each of which is the concatenation of the leading strand of a human chromosome with the leading or lagging strands of a mouse and a rat chromosome. Although one cannot change the number of combinations to be considered, there is a better strategy that works as follows. First, one builds the suffix tree only for the leading strands of the chromosomes of a reference genome, say, the human genome. Then one separately matches the leading and lagging strands of each chromosome of the other genomes against each suffix tree. This procedure yields pairwise matches in the form of rare *MEMs*, which are stored in a suitable data structure (on file, if necessary). Finally, if one wants to compare specific chromosomes (e.g., human chromosome 17 with mouse chromosome 11 and rat chromosome 10), then one combines the pairwise rare *MEMs* between these chromosomes to rare *multiMEMs*. Obviously, this approach is much more flexible than the aforementioned. If one has a "data base" of rare *MEMs* with respect to a reference genome, then one can easily perform pairwise or multiple comparisons between the reference genome and other genomes from the data base.

As another example, we would like to mention the mapping of cDNA to multiple genomes. In his experiments, Abouelhoda (2007) found out that rare *multiMEMs* were most suitable for this task. This is because, on the one hand, the sensitivity with *multiMUMs* was too low, and on the other hand, the number of *multiMEMs* was too large to be computed.

The paper is organized as follows. After a brief discussion of related work in Section 2, we state the basic concepts in Section 3. In Section 4, we recall a method for finding all maximal exact matches between *two* sequences, and in Section 5, we show that this method can be modified (albeit with considerable effort) such that it computes *rare* maximal exact matches. This computation is generalized to multiple sequences in Section 6. Section 7 discusses implementation details, and in Section 8 we report on experimental results. The concepts and notions introduced here are illustrated by several examples.

2. RELATED WORK

As already mentioned, many software tools for aligning large DNA sequences depend on the ability to efficiently compute exact matches (either k-mers or maximal matches) (Chain et al., 2003; Treangen and Messeguer, 2006). Software tools that simultaneously compute exact matches in all sequences under consideration are MGA (Höhl et al., 2002), EMAGEN (Deogen et al., 2004), Mauve (Darling et al., 2004), and M-GCAT (Treangen and Messeguer, 2006). MGA uses maximal multiple exact matches, while the other tools use maximal multiple unique matches. As discussed in Section 1, MGA's strategy of computing maximal multiple exact matches between k strings $S_1, \ldots, S_k$ by computing repeats in the string $S = S_1\$_1S_2\$_2, \ldots, S_{k-1}\$_{k-1}S_k$ has certain disadvantages. These disadvantages can be overcome by matching $k-1$ of the sequences against the remaining (reference) sequence. The technique of matching a query string against a suffix tree in linear time by using suffix links goes back to Chang and Lawler (1994). This technique was also used in *MUMmer*2 (Delcher et al., 2002) to compute maximal unique matches between *two* sequences. *MUMmer*3 (Kurtz et al., 2004) additionally allows one to compute all maximal exact matches between *two* sequences in this manner, using an algorithm described in Kurtz and Lonardi (2004). Although no details are given in Treangen and Messeguer (2006), it seems that M-GCAT uses a similar algorithm to compute maximal *unique* matches between multiple sequences. We stress, however, that no algorithm is known that computes *rare* maximal exact matches between *two* or *multiple* sequences in this way.

3. BASIC DEFINITIONS

For $1 \leq i \leq k$, S_i denotes a string of length $n_i = |S_i|$. If $S_i = uvw$ for some (possibly empty) sequences u, v and w, then u is a *prefix* of s, v is a *substring* of s, and w is a *suffix* of s. A substring, a

prefix or a suffix of s is *proper* if it is different from s. In our application, S_i is one strand of the DNA double strand of a chromosome or genome G_i. However, the algorithms presented here work for any kind of sequence.

Let \$ be a special sentinel character that does not occur in S_i. It is appended to S_i, so that no suffix of $S_i\$$ is also a prefix of S_i (this fact is important when using suffix trees for substring matches). $S_i[h]$ denotes the *character at position h* in S_i, while $S_i[l..h]$ denotes the substring of S_i starting at position l and ending at position h. Given $t_i \in \mathbb{N}$, the substring $S_i[l..h]$ of S_i is said to be *rare in* S_i if it occurs at most t_i times in S_i.

3.1. Suffix trees

A *suffix tree* $ST(S_1\$)$ for the string $S_1\$$ is a rooted directed tree with exactly $n_1 + 1$ leaves numbered 0 to n_1. Each internal node, other than the root, has at least two children, and each edge is labeled with a nonempty substring of $S_1\$$. No two edges out of a node can have edge labels beginning with the same character. The key feature of the suffix tree is that for any leaf j, the concatenation of the edge labels on the path from the root to leaf j exactly spells out $S_1[j..n_1-1]\$$, the j-th nonempty suffix of the string $S_1\$$. Figure 1 shows the suffix tree for the string $S_1\$ = acaaacatat\$$.

For $ST(S_1\$)$ we also use the abbreviation ST. As already mentioned, the leaves in ST are numbered such that leaf j represents the jth suffix of $S_1\$$. For convenience, let the interior nodes of ST also be numbered. That is, an interior node gets a number $n_1 < j \leq n'$, where $n' \leq 2n_1 - 1$ is the number of nodes in ST (Fig. 2). For each node $j \neq root$, let $parent(j)$ denote the parent node of j in ST. In the following, we denote a node j in the suffix tree by $\overline{u}$ if and only if the concatenation of the edge labels on the path from the root to node j spells out the string u. $|u|$ is the *depth* of node $\overline{u}$. It is a property of suffix trees that for any internal node $\overline{au}$, where a is some character, there is also an internal node $\overline{u}$. A pointer from $\overline{au}$ to $\overline{u}$ is called a *suffix link* (Fig. 1).

A suffix tree can be built in linear time and space (Weiner, 1973).

3.2. Exact matches

Definition 1. An *exact match* between two strings S_1 and S_2 is a triple (l, p_1, p_2) such that $S_1[p_1..p_1+l-1] = S_2[p_2..p_2+l-1]$. An exact match is called *right maximal* (*RMEM*) if $p_1+1 = n_1$ or $p_2+1 = n_2$ or $S_1[p_1 + l] \neq S_2[p_2 + l]$. It is called *left maximal* if $p_1 = 0$ or $p_2 = 0$ or $S_1[p_1 - 1] \neq S_2[p_2 - 1]$. A left and right maximal exact match is called *maximal exact match* (*MEM*).

Using user-defined thresholds t_1 and t_2 on the number of allowed copies of a *MEM* in the strings S_1 and S_2 yields the notion of rare *MEM*.

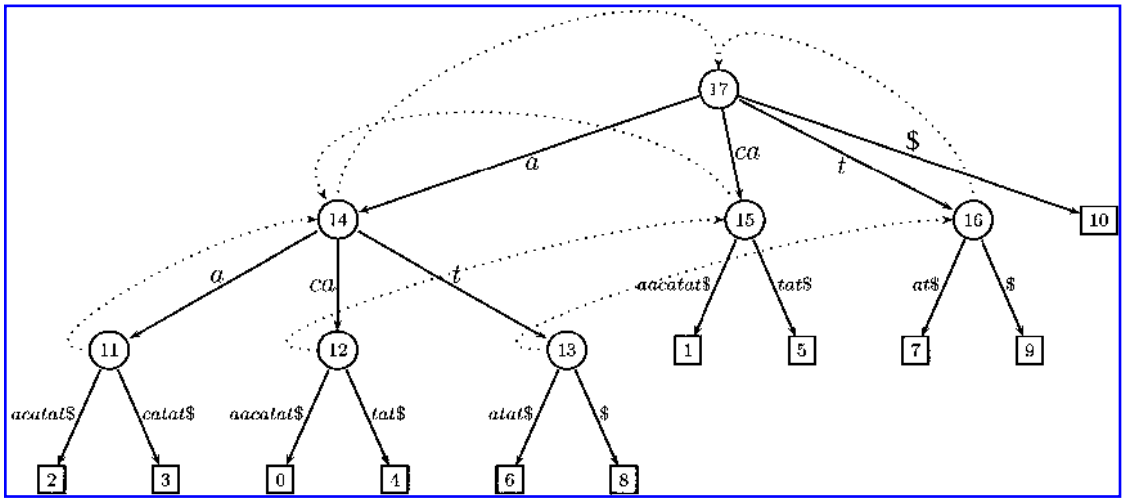

FIG. 1. The suffix tree for $S_1\$ = acaaacatat\$$. Suffix links for the interior nodes are drawn as dotted arrows. The leaves and interior nodes are numbered as described in the text.

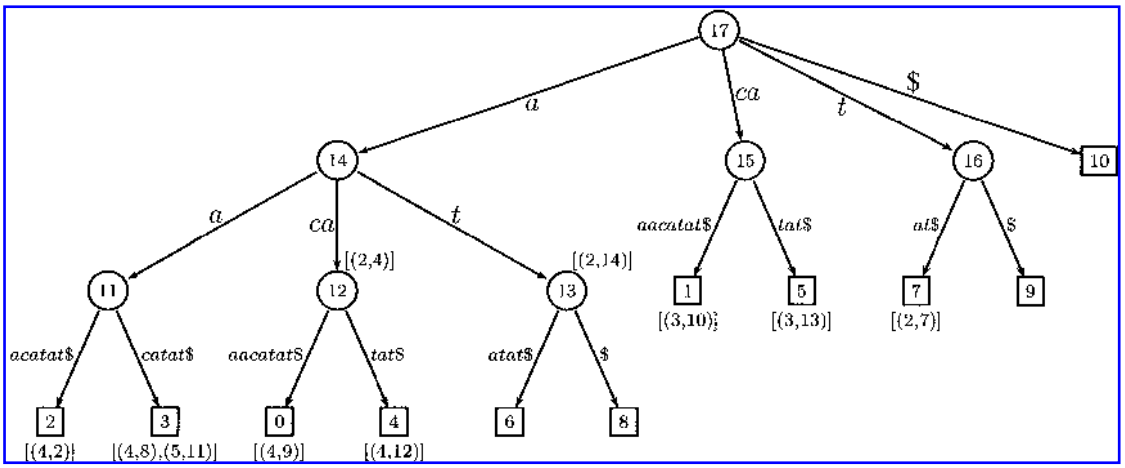

FIG. 2. The suffix tree for $S_1\$ = acaaacatat\$$ annotated with matches with respect to $S_2 = aaaaacttaacaacat$. (Note that the element $(2, 4)$ at node 12 will be deleted from L[12] in the subsequent deletion phase.)

Definition 2. Given t_1 and t_2, a *MEM* or *RMEM* (l, p_1, p_2) is called *rare in* S_i if the string $S_1[p_1..p_1 + l - 1] = S_2[p_2..p_2 + l - 1]$ is rare in S_i, that is, if it occurs at most t_i times in S_i. A *MEM* or *RMEM* (l, p_1, p_2) is called *rare* if it is rare in S_1 and S_2. A maximal unique match (*MUM*) is a rare *MEM* with respect to the thresholds $t_1 = 1 = t_2$. In other words, the number of allowed copies of that string equals 1.

4. FINDING ALL *MEM*s BETWEEN TWO STRINGS

In this section, we will recall how one can compute all *MEMs* between two genomic sequences S_1 and S_2 by using only the suffix tree *ST* of the sequence $S_1\$$ (Kurtz and Lonardi, 2004). Our exposition follows Gusfield (1997), which contains a description of Chang and Lawler's (1994) technique of matching a query sequence S_2 against a suffix tree *ST* in linear time.

The naive way to compute all *MEMs* (l, p_1, p_2), where p_2 is a fixed position in S_2, is to match the initial characters of $S_2[p_2..n_2 - 1]$ against *ST* by following the unique path of character matches until no further matches are possible. If there are l matches until a mismatch occurs, and the mismatch occurs on the edge label $\overline{u} \to \overline{uv}$, then $S_2[p_2..p_2 + l - 1] = uw$ for some proper prefix w of v ($w = \varepsilon$ is possible), but $S_2[p_2..p_2 + l]$ does not match any substring of S_1. If $w = \varepsilon$, then we say that the matching substring $S_2[p_2..p_2 + l - 1]$ *ends* at node $\overline{u}$. If $w \neq \varepsilon$, then we say that the matching substring $S_2[p_2..p_2 + l - 1]$ *ends* at node $\overline{uv}$. We can find out at what positions the string uw occurs in S_1 by considering the leaf numbers of the subtree rooted at $\overline{uv}$ if $w \neq \varepsilon$ (at $\overline{u}$ if $w = \varepsilon$, respectively). Let *leafset*$(\overline{uv})$ be the leaf set for $\overline{uv}$, that is, the set of leaf numbers in the subtree below node $\overline{uv}$. For each $j \in$ *leafset*$(\overline{uv})$, the concatenation of the edge labels on the path from the root to leaf j exactly spells out the string $S_1[j..n_1 - 1]\$$ (i.e., the jth suffix of $S_1\$$). Consequently, for each $p_1 \in$ *leafset*$(\overline{uv})$, the triple (l, p_1, p_2) is an *RMEM*. It is a *MEM* if it is also left-maximal. The test for left-maximality (check whether either $p_1 = 0$ or $p_2 = 0$ or $S_1[p_1 - 1] \neq S_2[p_2 - 1]$ holds) takes only constant time. On the one hand, repeating this procedure for every p_2, $0 \leq p_2 \leq n_2 - 1$ would not yield a linear time algorithm. On the other hand, if we would supply every node $\overline{u}$ in *ST* with a counter that is incremented whenever $\overline{u}$ is visited, then we could count how often the string u occurs in S_2.

To compute all *MEMs* (l, p_1, p_2) for $p_2 = 0$, we use the naive method described above, that is, we match the characters of string S_2 against *ST* by following the unique matching path of $S_2[0..n_2 - 1]$ starting from the root of *ST*. Now suppose in general that the algorithm has just followed a matching path for some position p_2 in S_2 ending at node j. More precisely, let $S_2[p_2..p_2 + l - 1]$ be the length l substring of S_2 that starts at position p_2 and matches auw, where au is the string corresponding to node $\overline{au} = parent(j)$ and w is a non-empty prefix of the label v ($w = v$ is possible) of the edge $parent(j) \to j$, but $S_2[p_2..p_2 + l]$ does not match a substring of S_1. To match the next suffix $S_2[p_2 + 1..n_2 - 1]$ against *ST*, one follows the

suffix link $\overline{au} \rightarrow \overline{u}$. Because auw is a prefix of $S_2[p_2..p_2+l-1]$, uw is a prefix of $S_2[p_2+1..p_2+l-1]$. That is, the search for matches can start at node $\overline{u}$. Moreover, instead of traversing the path labeled uw by examining every character on it, the algorithm uses the skip and count trick of Ukkonen's (1995) suffix tree construction algorithm. This trick works as follows: Let $\overline{u} \rightarrow \overline{uw_1} \rightarrow \overline{uw_1w_2} \rightarrow \cdots \rightarrow \overline{uw_1w_2, \ldots, w_m}$ be the path in ST such that $w = w_1w_2, \ldots, w_mw_{m+1}$ and there is an edge outgoing from $\overline{uw_1w_2, \ldots, w_m}$ such that w_{m+1} is a proper prefix of the edge label ($w_{m+1} = \varepsilon$ is possible). For each node $\overline{uw_1w_2, \ldots, w_i}$, one follows the edge whose first character (of the label) coincides with the first character of w_{i+1}. In this manner, the correct child node $\overline{uw_1w_2, \ldots, w_iw_{i+1}}$ is reached in constant time. Finally, the matching phase continues by following the unique matching path of $S_2[|uw|..n_2-1]$ starting from the $|w_{m+1}|+1$-th character of the label of the outgoing edge of $\overline{uw_1w_2, \ldots, w_m}$ whose first character coincides with the first character of w_{m+1}.

Example 1. Let $S_1 = acaaacatat$. The suffix tree for $S_1\$$ is depicted in Figure 1. Suppose that in the computation of all maximal exact matches between S_1 and the string $S_2 = aaaaacttaacaacat$, the algorithm has followed the matching path of the suffix cat of S_2. In order to find the matching path for at, the algorithm first follows the suffix link from node 15 ($\overline{ca}$) to node 14 ($\overline{a}$), and then follows the edge whose label has t as the first letter.

Matching query sequence S_2 against the suffix tree ST takes linear time. To be precise, the construction of ST takes $O(n_1)$ time, and the matching phase takes $O(n_2)$ time. In contrast to the naive algorithm, however, if we would supply every node $\overline{u}$ in ST with a counter that is incremented whenever $\overline{u}$ is visited, then the value of this counter does not generally give the number of occurrences of the string u in S_2. This is because the algorithm takes the suffix link shortcuts.

The overall time complexity for the computation of all *MEMs* in this way is $O(n_1 + n_2 + r)$, where r is the number of *RMEMs*. Note that the space consumption does not depend on n_2.

5. COMPUTATION OF RARE *MEM*s

Here we will show how all rare *MEMs* of S_1 and S_2 that exceed a length threshold ℓ can be computed space efficiently by matching S_2 against the suffix tree ST of $S_1\$$. A node j in ST is called *relevant* w.r.t. S_2 if there is a rare *RMEM* between S_1 and S_2 of length $\geq \ell$ ending at j. Our algorithm consists of three phases:

1. Matching phase, in which we identify (a) nodes in ST that are potentially relevant and (b) strings corresponding to potentially rare *RMEMs* (these are stored in lists).
2. Deletion phase, in which we (a) determine whether a potentially relevant node is really relevant or not and (b) delete strings corresponding to *RMEMs* that are not rare (from the lists).
3. Output phase, in which rare *MEMs* are generated and output.

5.1. Pairwise matching phase

Each node j in ST satisfying (1) j has string depth $\geq \ell$, and (2) the subtree of ST with root j has at most t_1 leaves, is potentially relevant. Any other node $\overline{u}$ is irrelevant because if (1) is not satisfied, then u is too short and if (2) is not satisfied, then u is not rare in S_1. A potentially relevant node j is associated with a value $\mathsf{min}[j]$ and a list $\mathsf{L}[j]$. Initially, $\mathsf{min}[j] := \ell$ and $\mathsf{L}[j] := [\,]$, where $[\,]$ denotes the empty list. $\mathsf{L}[j]$ has at most t_2 entries of the form $(length, position)$ in decreasing order w.r.t. the first component. The following invariants are maintained in each step of the matching phase:

- Each entry $(length, position)$ in $\mathsf{L}[j]$ satisfies $length \geq \mathsf{min}[j]$.
- If $\mathsf{min}[j] = \ell$, then at most t_2 matches of length ℓ ending at node j have been detected so far.
- If $\mathsf{min}[j] > \ell$, then the prefix of length $\mathsf{min}[j]-1$ of u occurs more than t_2 times in S_2, where u is the string corresponding to node j.

At the beginning, as long as $\mathsf{L}[j]$ has less than t_2 members (i.e., $|\mathsf{L}[j]| < t_2$), we insert exact matches ending at node j into $\mathsf{L}[j]$ provided that the length of the match is greater than or equal to $\mathsf{min}[j] = \ell$.

At the point at which $\mathsf{L}[j]$ has exactly t_2 members and there is another exact match ending at node j, we have to update $\mathsf{L}[j]$ such that it does not contain exact matches which we already know are not rare in S_2. From that point on, the list may shrink and expand.

Let us turn to the details of this procedure. Suppose that the algorithm has just followed a matching path for some position p_2 in S_2 ending at node j. More precisely, let $S_2[p_2..p_2 + l - 1]$ be the length l substring of S_2 that starts at position p_2 and matches uw, where u is the string corresponding to node $parent(j)$ and w is a non-empty prefix of the label v ($w = v$ is possible) of the edge $parent(j) \to j$, but $S_2[p_2..p_2 + l]$ does not match a substring of S_1. If $l < \mathsf{min}[j]$, there is nothing to do. Otherwise, if $l \geq \mathsf{min}[j]$, we further proceed by case analysis.

- If $|\mathsf{L}[j]| < t_2$, then the capacity of $\mathsf{L}[j]$ is not exceeded and so the pair (l, p_2) is inserted into $\mathsf{L}[j]$.
- Suppose $|\mathsf{L}[j]| = t_2$ and let len be the smallest length value in $\mathsf{L}[j]$.
  - Let $l \geq \mathsf{len}$. Since $\mathsf{len} \geq \mathsf{min}[j]$, we have $l \geq \mathsf{min}[j]$. Hence, there are more than t_2 occurrences of $S_2[p_2..p_2 + \mathsf{len} - 1]$ in S_2. We delete all elements with length value len from $\mathsf{L}[j]$ and set $\mathsf{min}[j] := \mathsf{len} + 1$. Note that at least one element was deleted from $\mathsf{L}[j]$. So (l, p_2) is inserted into $\mathsf{L}[j]$ if $l > \mathsf{len}$. After deleting the elements and one possible insertion (if $l > \mathsf{len}$), we have $|\mathsf{L}[j]| \leq t_2$ and it is easy to see that the invariants specified above are satisfied.
  - If $l < \mathsf{len}$, then we know that $S_2[p_2..p_2 + l - 1]$ occurs more than t_2 times in S_2. Hence, we set $\mathsf{min}[j] := l + 1$. Since all elements of $\mathsf{L}[j]$ are of length at least $\mathsf{len} \geq l + 1 = \mathsf{min}[j]$, the invariants specified above are satisfied.

Example 2. Figure 2 depicts the annotated suffix tree of $S_1\$ = acaaacatat\$$ after matching $S_2 = aaaaacttaacaacat$ with threshold parameters $\ell = 2$ and $t_1 = t_2 = 2$ against it. Note that the nodes 10, 14, 16, and 17 are irrelevant because their string depth is smaller than ℓ. Moreover, the subtrees with root node 14 and 17 have more than t_1 nodes.

To show how our method works, consider the list $\mathsf{L}[2]$ of leaf 2. It contains all matches of substrings of S_2 with prefixes of the suffix $aaacatat$ that start at position 2 in S_1. When the algorithm detects that the first three characters of the suffix of S_2 starting at position 0 and those of $aaacatat$ match, it inserts the pair $(3, 0)$ into the initially empty list $\mathsf{L}[2]$. Analogously, the pair $(3, 1)$ is inserted into $\mathsf{L}[2]$ because the capacity of $\mathsf{L}[2]$ is not exceeded yet. However, when the match of the length 4 prefix of $aaacatat$ and $S_2[2..15]$ is detected, the list $\mathsf{L}[2]$ already contains $t_2 = 2$ elements. Consequently, the elements $(3, 0)$ and $(3, 1)$ are deleted from the list (they have the smallest length value $\mathsf{len} = 3$), the value $\mathsf{min}[2]$ is set to $\mathsf{len} + 1 = 4$, and the pair $(4, 2)$ is inserted into $\mathsf{L}[2]$.

5.2. Deletion phase

Because of the nature of the matching procedure (described in Section 4), it cannot be used to count how often a string u occurs in S_2. That is the reason why the suffix tree may be annotated with more lists and list elements than necessary. Thus, before we compute rare *MEMs* by a traversal of *ST*, we get rid of unnecessary nodes and list-elements. More precisely, we first want to identify nodes that are irrelevant. Second, for the remaining relevant nodes j, we want to delete elements from list $\mathsf{L}[j]$ that do not correspond to rare *RMEMs*.

Let us turn to the details of this procedure. In a bottom-up traversal of *ST*, for any node j, the following value is computed:

$$elem\_in\_sublists[j] := \begin{cases} 0 & \text{if } j \text{ is a leaf} \\ \displaystyle\sum_{j' : j = parent(j')} (elem\_in\_sublists[j'] + |\mathsf{L}[j']|) & \text{otherwise} \end{cases}$$

In words, $elem\_in\_sublists[j]$ denotes the total number of elements in the L-lists of all nodes in the subtrees strictly below j. That is, the string u corresponding to j occurs $elem\_in\_sublists[j]$ times as a *proper* substring of matches between S_1 and S_2.

If j satisfies $\mathsf{min}[j] > \ell$, then by the third invariant the prefix of length $\mathsf{min}[j] - 1$ of u occurs more than t_2 times in S_2. In other words, every ancestor of j is irrelevant because its corresponding string occurs too many times. Consequently, there is no need to continue the bottom-up traversal of the annotated suffix tree

with the ancestors of j. In our example, node 2 satisfies $\mathsf{min}[2] = 4 > \ell = 2$ and hence every ancestor of node 2 is irrelevant (in particular, node 11 is irrelevant).

From now on, we assume that the node j under consideration was not identified to be irrelevant yet. Hence every successor j' of j satisfies $\mathsf{min}[j'] = \ell$. That is, nothing was deleted from the list $\mathsf{L}[j']$.

Theorem 1. *Node j is irrelevant if and only if at least one of the following conditions holds (where u is the string corresponding to node j).*

(1) $\mathsf{L}[j] = [\,]$ *and* $\mathit{elem\_in\_sublists}[j] = 0$.
(2) $\mathsf{min}[j] > |u|$.
(3) $\mathit{elem\_in\_sublists}[j] + |\{(r, q) \in \mathsf{L}[j] \mid r = |u|\}| > t_2$.

Proof. "**if**" If j satisfies (1), then there is no rare *RMEM* ending at j. If j satisfies (2), then we have $\mathsf{min}[j] > |u| \geq \ell$ and hence u is of length $\leq \mathsf{min}[j] - 1$ and thus occurs more than t_2 times in S_2 (see third invariant above). If condition (3) is satisfied, then the strings represented by the elements in $\mathsf{L}[j]$ occur more than t_2 times in S_2. This argumentation shows that each node j satisfying one of the conditions (1)–(3) is irrelevant.

"**only if**" Conversely, assume that none of the above-mentioned conditions holds for node j. Because conditions (2) and (3) are not satisfied, it follows that the string u occurs at most t_2 times in S_2. In order to show that j is relevant, we proceed by case analysis. If j is a leaf, then $\mathit{elem\_in\_sublists}[j] = 0$. Since condition (1) is not true, it follows that list $\mathsf{L}[j]$ is not empty, and it is readily verified that every element in $\mathsf{L}[j]$ corresponds to a rare *RMEM*. Hence j is relevant. Now suppose that j is an internal node. If $\mathit{elem\_in\_sublists}[j] \neq 0$, then there is at least one successor j' of j such that $(l, p_2) \in \mathsf{L}[j']$. Because j is an internal node and every internal node is branching, there is a suffix $S_1[p_1, \ldots, n_1]\$$ of $S_1\$$ in the subtree rooted at j, which is not in the subtree rooted at j'. Therefore, $(|u|, p_1, p_2)$ is a rare *RMEM*. In other words, node j is relevant. Otherwise, $\mathit{elem\_in\_sublists}[j] = 0$ and $\mathsf{L}[j] \neq [\,]$. Again, it is readily seen that every element in $\mathsf{L}[j]$ corresponds to a rare *RMEM*. Thus, j is also relevant in this case. ■

In phase (2a) of our algorithm, we mark all nodes that are irrelevant according to the preceding theorem. In our example, none of the nodes is marked, but if, for example, $\mathsf{L}[15]$ would contain an entry $(\mathit{length}, \mathit{position})$ with $\mathit{length} = 2$, then node 15 would be marked because of condition (3).

We now explain phase (2b) of our algorithm. Let j be a relevant node. For each element $(l, p_2) \in \mathsf{L}[j]$, the string $S_2[p_2..p_2 + l - 1]$ of length l occurs as a substring of S_1. We say that this string is represented by (l, p_2). Since $\mathit{elem\_in\_sublists}[j] \leq t_2$, the value $t_2 - \mathit{elem\_in\_sublists}[j]$ is an upper bound on the number of rare strings on the path from the root of *ST* to node j. We distinguish between the cases $|\mathsf{L}[j]| \leq t_2 - \mathit{elem\_in\_sublists}[j]$ and $|\mathsf{L}[j]| > t_2 - \mathit{elem\_in\_sublists}[j]$:

- Suppose $|\mathsf{L}[j]| \leq t_2 - \mathit{elem\_in\_sublists}[j]$. Then each element in $\mathsf{L}[j]$ represents a string that corresponds to rare *RMEMs*. Therefore, we keep the complete list $\mathsf{L}[j]$.
- Suppose $|\mathsf{L}[j]| > t_2 - \mathit{elem\_in\_sublists}[j]$. This means $\mathit{elem\_in\_sublists}[j] + |\mathsf{L}[j]| > t_2$. Hence we have to delete elements from $\mathsf{L}[j]$. We iteratively delete *all* elements from $\mathsf{L}[j]$ with minimum length component,[2] until we arrive at a list $\mathsf{L}[j]$ such that $\mathit{elem\_in\_sublists}[j] + |\mathsf{L}[j]|$ does not exceed t_2. The deletion of these elements does no harm because the strings corresponding to the elements occur more than t_2 times in S_2.

Note that in case we delete elements from $\mathsf{L}[j]$, we have to update $\mathsf{min}[j]$ appropriately. To make this more precise, suppose the original size of $\mathsf{L}[j]$ was q and the new size after deleting the elements is $q' < q$. Let (l, p_2) be the element at index $q' + 1$ in the original list. Then all elements in the new list are of length at least $l + 1$, and the prefix of u of length l occurs more than t_2 times in S_2. As a consequence, we set $\mathsf{min}[j]$ to $l + 1$.

To illustrate the deletion phase (2b), we continue Example 2. The element $(2, 4)$ at node 12 will be deleted from $\mathsf{L}[12]$ and $\mathsf{min}[12]$ is set to 3. Although the L-list of node 12 is empty after the deletion phase, the node is relevant w.r.t. S_2 (leading to type 2 match candidate specifications).

[2] After each deletion step, the resulting list is again called $\mathsf{L}[j]$.

Although the computation of the *elem_in_sublists*-values and the deletion phases (2a) and (2b) have been described as separate steps, we would like to stress that all these computations are actually done in a single bottom-up traversal of the suffix tree.

5.3. Output phase

In this section, we assume that j is a relevant node for which $\mathsf{min}[j]$ and $\mathsf{L}[j]$ have been computed by the matching and deletion procedures of Sections 5.1 and 5.2. It is clear from the previous sections that all rare *RMEMs* between S_1 and S_2 are contained in the L-lists of relevant nodes. To be precise, given a relevant node j, the matches contained in $\mathsf{L}[j]$ immediately yield rare *RMEMs* ending at j. Moreover, every match contained in $\mathsf{L}[j']$, where j' is a successor of j gives rise to at least one rare *RMEM* ending at j.

In view of the more general case of rare *multiRMEMs*, we now introduce the abstract concept of *match candidate specifications* for j. In the pairwise case, each match candidate specification leads to at least one rare *RMEM* as we shall see below.

5.3.1. Type 1 match candidate specifications. Let j be a relevant node w.r.t. S_2. For each (l, p_2) in $\mathsf{L}[j]$, the triple (l, P_1, p_2) is called a *type 1 match candidate specification* for j and S_2, where $P_1 = leafset(j)$.

In the pairwise case, a type 1 match candidate specification directly leads to at least one rare *RMEM*. That is, for each match candidate specification (l, P_1, p_2) and each $p_1 \in P_1$, the triple (l, p_1, p_2) is a rare right-maximal exact match between S_1 and S_2. Clearly, if (l, p_1, p_2) is also left-maximal, then it is a rare *MEM*. To check for left-maximality, we have to verify that either $p_1 = 0$ or $p_2 = 0$ or $S_1[p_1 - 1] \neq S_2[p_2 - 1]$.

Continuing the previous example, we have the following type 1 match candidate specifications: $(4, \{2\}, 2)$, $(4, \{3\}, 8)$, $(5, \{3\}, 11)$, $(4, \{0\}, 9)$, $(4, \{4\}, 12)$, $(2, \{6, 8\}, 14)$, $(3, \{1\}, 10)$, $(3, \{5\}, 13)$, and $(2, \{7\}, 7)$. The match candidate specification $(2, \{6, 8\}, 14)$ specifies the right-maximal exact matches $(2, 6, 14)$ and $(2, 8, 14)$. Since $S_1[5] = c = S_2[13]$, it follows that $(2, 6, 14)$ is not left-maximal, whereas $S_1[7] = t \neq c = S_2[13]$ shows that $(2, 8, 14)$ is left-maximal. Hence, $(2, 8, 14)$ is a rare *MEM*.

5.3.2. Type 2 match candidate specifications. Suppose there is a pair of relevant nodes j and j', where j' is a successor of j. Furthermore, let j'' be the direct successor of j on the path from j to j' (note that $j' = j''$ is possible). For each (l, p_2) in $\mathsf{L}[j']$, the triple (l', P_1, p_2) is a type 2 match candidate specification for j and S_2, where l' is the string depth of j and $P_1 = leafset(j) \setminus leafset(j'')$. We exclude $leafset(j'')$ from P_1 because the leaves in the subtree below j'' have already been considered when computing other match candidate specifications.

Continuing our example, we have the following type 2 match candidate specifications: $(3, \{0\}, 12)$, $(3, \{4\}, 9)$, $(2, \{1\}, 13)$, and $(2, \{5\}, 10)$.

In the pairwise case, a type 2 match candidate specification directly leads to at least one rare *RMEM*. That is, for each match candidate specification (l', P_1, p_2) and each $p_1 \in P_1$, the triple (l', p_1, p_2) is a rare right-maximal exact match between S_1 and S_2. Clearly, (l', p_1, p_2) is a rare *MEM* if and only if it is left-maximal.

5.4. Time and space complexity

As already discussed in Section 4, the construction of the suffix tree *ST* takes $O(n_1)$ time, and matching S_2 against *ST* takes $O(n_2)$ time. Here, however, L-lists have to be updated during the matching phase. Because the length of any L-list is limited to t_2, the matching phase requires $O(n_2 t_2)$ time.

In the deletion and output phases, we traverse *ST* in a bottom-up fashion. Such a bottom-up traversal of *ST* requires $O(n_1)$ time. At each node, we check in $O(t_2)$ time whether it is relevant or not. If so, superfluous elements from its L-list are deleted in $O(t_2)$ time. At each relevant node, the number of length/position pairs leading to match candidate specifications is bounded by $O(t_2)$. To obtain type 2 match candidate specifications, one further collects sets of relevant nodes j' during the traversal, combines them with relevant predecessor nodes j, and derives type 2 match candidate specifications as described above. The size of the node sets collected during the traversal is bounded by $O(t_1)$. Because left-maximality of a rare *RMEM* can be verified in constant time and there are at most $t_1 t_2$ rare *RMEMs* ending at a node j,

all type 1 and 2 match candidate specifications can be computed in $O(n_1 t_1 t_2)$ time. Therefore, the overall time complexity is $O(n_2 t_2 + n_1 t_1 t_2)$.

We would like to point out that the space complexity does not depend on n_2. The space consumption for the suffix tree ST of the sequence S_1, as well as for the min-values and the L-lists stored at each node of ST is $O(n_1 t_2)$.

6. THE GENERALIZATION TO MORE THAN TWO GENOMES

The following definition naturally extends the notion of "rare maximal exact match between two sequences" to multiple sequences.

Definition 3. A *multiple exact match* between k strings $S_1, \ldots, S_k$ is a $(k+1)$–tuple $(l, p_1, \ldots, p_k)$ such that $S_1[p_1..p_1 + l - 1] = S_2[p_2..p_2 + l - 1] = \cdots = S_k[p_k..p_k + l - 1]$. A multiple exact match is *right (left) maximal* if it cannot be simultaneously extended to the right (left) in each sequence S_i, $1 \le i \le k$. We use the term *multiRMEM* as a shorthand for right maximal multiple exact match. A left and right maximal multiple exact match is called maximal multiple exact match (*multiMEM*). Given $t_1, \ldots, t_k \in \mathbb{N}$, a *multiMEM* or *multiRMEM* $(l, p_1, \ldots, p_k)$ is called *rare* if it is rare in each S_i, that is, the string $S_1[p_1..p_1 + l - 1] = S_i[p_i..p_i + l - 1]$ occurs at most t_i times in S_i for all $1 \le i \le k$. A maximal multiple unique match (*multiMUM*) is a rare *multiMEM* with respect to the thresholds $t_1 = \cdots = t_k = 1$. In other words, the number of allowed copies of that string equals 1.

6.1. The multiple matching and deletion phases

In order to compute rare *multiMEMs*, we match each S_i, $2 \le i \le k$, separately against ST and compute $\mathsf{min}_i[j]$ and $\mathsf{L}_i[j]$ as described in Sections 5.1 and 5.2. The advantage of this strategy is that the values can be computed in parallel.

We would like to stress, however, that it is also possible to compute $\mathsf{min}_i[j]$ and $\mathsf{L}_i[j]$ incrementally by sequentially matching $S_2, \ldots, S_k$ against ST. That is, first S_2 is matched against ST as described in Section 5. Then S_3 is matched against ST and so on. This strategy has the following advantage: If for some $i \ge 3$, node j is not relevant w.r.t. S_{i-1}, then it is not necessary to compute $\mathsf{L}_i[j]$ because there cannot be a rare *multiMEM* ending at node j. This strategy may considerably reduce the number of matches to be stored. Furthermore, if node j is relevant for all S_h with $2 \le h < i$, then $\mathsf{min}_i[j]$ can be initialized with $\mathsf{min}_{i-1}[j]$ instead of ℓ.

6.2. A characterization of rare right-maximal multiple exact matches

We now show how to combine the values from the pairwise matching and deletion phases to compute rare *multiRMEMs*. In essence, we combine pairwise rare *RMEMs*. As we shall see, however, the length l_i of such a pairwise rare *RMEM* (l_i, p_1, p_i) between S_1 and S_i usually must be shortened to some $l^* < l_i$. The next lemma provides a criterion for testing whether the string $S_1[p_1..p_i + l^* - 1] = S_i[p_i..p_i + l^* - 1]$ is rare in S_i.

Lemma 1. *Let j be a relevant node w.r.t. S_i such that the string $S_i[p_i..p_i + l^* - 1]$ with $l^* \ge \ell$ ends at node j or at a successor node of j. Then it is rare in S_i if and only if $l^* \ge \mathsf{min}_i[j]$.*

Proof. If $l^* < \mathsf{min}_i[j]$, then $S_i[p_i..p_i + l^* - 1]$ is not rare in S_i according to the invariants maintained in the matching phase. To show the other direction, suppose that $S_i[p_i..p_i + l^* - 1]$ is not rare in S_i. As usual, let u be the string corresponding to node j. If $l^* > |u|$, then j would be irrelevant. Thus $l^* \le |u|$ must hold. If $S_i[p_i..p_i + l^* - 1]$ was not inserted into $\mathsf{L}[j]$ in the matching phase, then this was because $l^* < \mathsf{min}_i[j]$. If it was inserted into $\mathsf{L}[j]$ but afterwards deleted from $\mathsf{L}[j]$ in the matching phase, then we also have $l^* < \mathsf{min}_i[j]$. Finally, if $S_i[p_i..p_i + l^* - 1]$ was deleted from $\mathsf{L}[j]$ in the deletion phase, then $l^* \le \mathsf{min}_i[j] + 1$ (hence $l^* < \mathsf{min}_i[j]$) because $\mathsf{min}_i[j]$ was set to $l + 1$, where $l^* \le l$. ■

The next theorem provides a characterization of rare *multiRMEMs* that can be used to compute all rare *multiRMEMs* efficiently.

Theorem 2. *Suppose that the string $S_1[p_1..p_1 + l^* - 1]$ ends at node j in the suffix tree of S_1. Then $(l^*, p_1, \ldots, p_k)$ is a rare multiRMEM if and only if the following holds:*

- *for all $2 \leq i \leq k$, node j is relevant w.r.t. S_i, and*
- *for all $2 \leq i \leq k$ there is a RMEM (l_i, p_1, p_i) such that*
  - *(l_i, p_1, p_i) is rare in S_1 and S_i,*
  - *$l^* = \min\{l_i \mid 2 \leq i \leq k\}$, and*
  - *$l^* \geq \mathsf{maxmin}[j]$, where $\mathsf{maxmin}[j] = \max\{\mathsf{min}_i[j] \mid 2 \leq i \leq k\}$.*

Proof. "**only if**" Let $(l^*, p_1, \ldots, p_k)$ be a rare *multiRMEM*. That is, the string $S_1[p_1..p_1 + l^* - 1] = S_2[p_2..p_2 + l^* - 1] = \cdots = S_k[p_k..p_k + l^* - 1]$ is rare and there is at least one i such that $S_1[p_1 + l^*] \neq S_i[p_i + l^*]$. For each $2 \leq i \leq k$, let l_i be the number such that $S_1[p_1..p_1 + l_i - 1] = S_i[p_i..p_i + l_i - 1]$ and $S_1[p_1 + l_i] \neq S_i[p_i + l_i]$. Obviously, (l_i, p_1, p_i) is a *RMEM* and $l_i \geq l^*$. The former implies that j is relevant w.r.t. S_i. Moreover, it follows from the latter that (l_i, p_1, p_i) is rare in S_i (otherwise (l^*, p_1, p_i) would not be rare in S_i) and that $l^* = \min\{l_i \mid 2 \leq i \leq k\}$ (recall that at least for one i, we have $l_i = l^*$). For an indirect proof of $l^* \geq \mathsf{maxmin}[j]$ suppose that $l^* < \mathsf{maxmin}[j]$. Let i be an index such that $\mathsf{min}_i[j] = \mathsf{maxmin}[j]$. It follows from $l^* < \mathsf{min}_i[j]$ that the string $S_1[p_1..p_1 + l^* - 1] = S_i[p_i..p_i + l_i - 1]$ is not rare in S_i. This is a contradiction to the fact that this string is rare (in each S_i).

"**if**" Now suppose that for all $2 \leq i \leq k$ node j is relevant w.r.t. S_i and there is a *RMEM* (l_i, p_1, p_i). Furthermore, $l^* = \min\{l_i \mid 2 \leq i \leq k\}$ satisfies $l^* \geq \mathsf{maxmin}[j]$. Clearly, $S_1[p_1..p_1 + l^* - 1] = S_2[p_2..p_2 + l^* - 1] = \cdots = S_k[p_k..p_k + l^* - 1]$. Because there is at least one i such that $S_1[p_1 + l^*] \neq S_i[p_i + l^*]$, the tuple $(l^*, p_1, \ldots, p_k)$ is a *multiRMEM*. It must still be shown that the string $S_1[p_1..p_1 + l^* - 1]$ is rare in each S_i, where $2 \leq i \leq k$. It follows from $l^* \geq \mathsf{maxmin}[j]$ that $l^* \geq \mathsf{min}_i[j]$ for all $2 \leq i \leq k$. According to Lemma 1, for any $2 \leq i \leq k$, the string $S_1[p_1..p_1 + l^* - 1] = S_i[p_i..p_i + l^* - 1]$ is rare in S_i if and only if $l^* \geq \mathsf{min}_i[j]$. Hence, we conclude that $(l^*, p_1, \ldots, p_k)$ is a rare *multiRMEM*. ■

6.3. The multiple output phase

With the help of the preceding characterization of rare *multiRMEMs*, we immediately obtain an algorithm that computes all rare *multiRMEMs*. At first, all relevant nodes j are computed, that is, for all $2 \leq i \leq k$, node j must be relevant w.r.t. S_i. For each such node j, the following steps are performed:

(1) For each $i, 2 \leq i \leq k$, compute the set of all type 1, type 2, and type 3 match candidate specifications for j and S_i. Type 3 match candidate specifications are defined as follows: Enumerate all nodes j' in the subtree below j. For each $(l, p_2) \in \mathsf{L}_i[j']$, $(l, \mathit{leafset}(j'), p_2)$ is a type 3 match candidate specification for j and S_i.
(2) Compute $\mathsf{maxmin}[j] = \max\{\mathsf{min}_i[j] \mid 2 \leq i \leq k\}$.
(3) Discard match candidate specifications (l_i, P_1, p_i) satisfying $l_i < \mathsf{maxmin}[j]$. For each $i, 2 \leq i \leq k$, let $\mathsf{MCS}_i[j]$ denote the set of all remaining match candidate specifications for j and S_i.
(4) Enumerate the elements of the Cartesian product

$$\mathsf{MCS}^*[j] := \mathsf{MCS}_2[j] \times \mathsf{MCS}_3[j] \times \cdots \times \mathsf{MCS}_{k-1}[j] \times \mathsf{MCS}_k[j]$$

and for each element $((l_2, P_1^2, p_2), (l_3, P_1^3, p_3), \ldots, (l_k, P_1^k, p_k))$ do the following:
(a) Discard it if it solely consists of match candidate specifications of type 3.
(b)
- Compute $P_1^* = \bigcap_{i=2}^{k} P_1^i$.
- Compute $l^* = \min\{l_i \mid 2 \leq i \leq k\}$.
- For all $p_1 \in P_1^*$, $(l^*, p_1, p_2, p_3, \ldots, p_k)$ is a rare *multiRMEM*. If $p_i = 0$ for at least one i with $1 \leq i \leq k$, or the set $\{S_1[p_1 - 1]\} \cup \{S_i[p_i - 1] \mid 2 \leq i \leq k\}$ is not singleton, then it is also left-maximal and thus a rare *multiMEM*.

The correctness of our algorithm is a direct consequence of Theorem 2. According to this theorem, it suffices to solely consider relevant nodes. Furthermore, for each relevant node j, a rare *multiRMEM* ending at j is the combination of rare *RMEMs* (l_i, p_1, p_i), $2 \leq i \leq k$, ending at j or at a successor of j (note that this is the reason why we have to add type 3 match specifications). Moreover, the theorem implies that none of the rare *RMEMs* (l_i, p_1, p_i) is allowed to have a length smaller than $\mathsf{maxmin}[j]$. That is the reason why such rare *RMEMs* are excluded in our algorithm. Finally, all elements of $\mathsf{MCS}^*[j]$ that solely

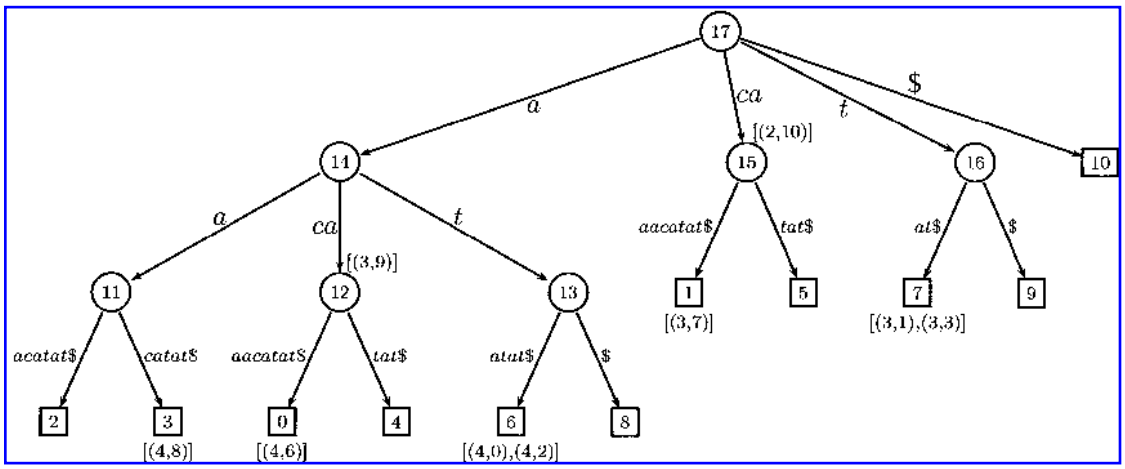

FIG. 3. The suffix tree for $S_1\$ = acaaacatat\$$ annotated with matches w.r.t. $S_3 = atatatacaaca$.

consist of type 3 match candidate specifications correspond to rare *multiRMEMs* ending at a successor node of j and thus have already been taken into account.

Example 3. As before, let $S_1 = acaaacatat$ and $S_2 = aaaaacttaacaacat$. Additionally, let $S_3 = atatatacaaca$. Matching S_3 against the suffix tree of $S_1\$$ yields the annotated suffix tree depicted in Figure 3.

To illustrate how our algorithm works, we consider the internal node $j = 12$. For this node and S_2, there is no type 1 match specification, the type 2 match specifications are $(3, \{0\}, 12)$ and $(3, \{4\}, 9)$, while the type 3 match specifications are $(4, \{0\}, 9)$ and $(4, \{4\}, 12)$. With respect to node $j = 12$ and S_3, there is one type 1 match specification $(3, \{0, 4\}, 9)$, one type 2 match specification $(3, \{4\}, 6)$, and one type 3 match specification $(4, \{0\}, 6)$. The combination of these match specifications leads to the following rare *multiRMEMs*: $(3, 0, 12, 9)$, $(3, 0, 12, 6)$, $(3, 4, 9, 9)$, $(3, 4, 9, 6)$, $(3, 0, 9, 9)$, $(3, 4, 12, 9)$, and $(3, 4, 12, 6)$. Note that the combination of the type 3 match specifications $(4, \{0\}, 9)$ and $(4, \{0\}, 6)$ is discarded at node $j = 12$ because the corresponding rare *multiRMEM* $(4, 0, 9, 6)$ has already been computed at leaf node $j = 0$. Out of these seven rare *multiRMEMs*, only $(3, 4, 9, 9)$ and $(3, 4, 12, 9)$ are not rare *multiMEMs* because $S_1[3] = S_2[8] = S_3[8]$ and $S_1[3] = S_2[11] = S_3[8]$.

6.4. Time and space complexity

Again, the construction time of the suffix tree ST of $S_1\$$ is $O(n_1)$. Recall from Section 5.4 that for each i, $2 \le i \le k$, the matching phase takes $O(n_i t_i)$ time, the deletion phase takes $O(n_1 t_i)$ time, and the computation of all match candidate specifications w.r.t. S_i requires $O(n_1 t_1 t_i)$ time. Thus, the time complexity up to step (1) of Section 6.3 is $O(\sum_{i=2}^{k} n_i t_i + n_1 t_1 \sum_{i=2}^{k} t_i)$.

The computation of $\mathsf{maxmin}[j]$ in step (2) can be done in $O(k)$ time for every node j in ST, resulting in an $O(n_1 k)$ time complexity for this step.

Step (3) requires $O(n_1 t_1 \sum_{i=2}^{k} t_i)$ time because there are $O(t_1 t_i)$ match candidate specifications at each node j in ST.

For every node j in ST, the set $\mathsf{MCS}^*[j]$ has $O(\prod_{i=2}^{k}(t_1 t_i))$ elements. Therefore, the time needed in step (4) to compute the Cartesian products at all relevant nodes would be $O(n_1 t_1^{k-1} \prod_{i=2}^{k} t_i)$. In order to avoid the factor t_1^{k-1}, the Cartesian product can be built incrementally and the intersection of the position sets is taken into account. More precisely, at each node j we first build $\mathsf{MCS}_2[j] \times \mathsf{MCS}_3[j]$ and delete all elements with $P_1^2 \cap P_1^3 = \emptyset$. This takes $O((t_1 \cdot t_2) \cdot (t_1 \cdot t_3)) = O(t_1^2 \cdot t_2 \cdot t_3)$ time. The resulting set $\mathsf{MCS}_{2,3}[j]$ has at most $t_1 \cdot t_2 \cdot t_3$ elements. Then we build $\mathsf{MCS}_{2,3}[j] \times \mathsf{MCS}_4[j]$ and delete all elements with $P_1^2 \cap P_1^3 \cap P_1^4 = \emptyset$. Similarly, this takes $O(t_1^2 \cdot t_2 \cdot t_3 \cdot t_4)$ time, and the resulting set has at most $t_1 \cdot t_2 \cdot t_3 \cdot t_4$ elements.

This process is continued until we get

$$\{((l_2, P_1^2, p_2), (l_3, P_1^3, p_3), \ldots, (l_k, P_1^k, p_k)) \in \mathsf{MCS}^*[j] \mid P_1^* = \bigcap_{i=2}^{k} P_1^i \neq \emptyset\}$$

in time $O(k\, t_1^2 \prod_{i=2}^{k} t_i)$. In this way, step (4) can be implemented in $O(n_1 k\, t_1^2 \prod_{i=2}^{k} t_i)$ time.

Consequently, the overall time complexity of our algorithm is $O(\sum_{i=2}^{k} n_i t_i + n_1 k\, t_1^2 \prod_{i=2}^{k} t_i)$.

Again, note that the space complexity is independent of the sequence lengths $n_2, \ldots, n_k$. The space consumption is $O(\max\{n_1 t_i \mid 2 \leq i \leq k\})$ because after matching S_i against ST, the resulting pairwise rare matches can be stored in secondary memory.

7. IMPLEMENTATION

An implementation of our algorithms requires a representation of leaf sets. As suggested in Stoye and Gusfield (2002), leaf sets can efficiently be represented by two additional integers for each interior node of the suffix tree. These integers refer to an interval of an array storing all leaf numbers of ST in depth first order. This additional array is the well-known suffix array as introduced by Manber and Myers (1993). Of course, the two integers need only be added to all nodes j with string depth $\geq \ell$ and $|leafset(j)| \leq t_1$. Nevertheless, the representation requires considerable space in addition to the suffix tree representation.

Instead of suffix trees, our implementation employs enhanced suffix arrays. This is for the following reasons:

- As shown in Abouelhoda et al. (2004), the enhanced suffix array is as powerful as the suffix tree. Any algorithm on the suffix tree can be implemented on the enhanced suffix array, without sacrificing optimal asymptotic running times. Moreover, enhanced suffix arrays require much less space than suffix trees.
- Enhanced suffix arrays already represent leaf sets.
- The software tool *Vmatch*[3] provides a comprehensive code base implementing enhanced suffix arrays. In fact, our implementation extensively uses this code base.

Before we describe our implementation in detail, let us briefly introduce the notion of enhanced suffix arrays in its simplest form:

Let $T_j = S_1[j..n_1 - 1]\$$ denote the suffix of $S_1\$$ beginning with jth position. The *suffix array* suftab of the string $S_1\$$ is an array of integers in the range 0 to n_1, specifying the lexicographic ordering of the $n_1 + 1$ suffixes of the string $S_1\$$. That is, $T_{\mathsf{suftab}[0]}, T_{\mathsf{suftab}[1]}, \ldots, T_{\mathsf{suftab}[n_1]}$ is the sequence of suffixes of $S_1\$$ in ascending lexicographic order. The *lcp-table* lcptab is an array of integers in the range 1 to n_1. We define $\mathsf{lcptab}[j]$ to be the length of the longest common prefix of $T_{\mathsf{suftab}[j-1]}$ and $T_{\mathsf{suftab}[j]}$, for $1 \leq j \leq n_1$. The enhanced suffix array for S_1 consists of the tables suftab and lcptab. An interval $[l..r]$, $0 \leq l \leq r \leq n_1$, is an *lcp-interval of lcp-value q* (or q-interval, for short) if the following holds:

1. if $l < r$, then
   - $l = 0$ or $\mathsf{lcptab}[l] < q$,
   - $\mathsf{lcptab}[s] \geq q$ for all s, $l + 1 \leq s \leq r$,
   - $\mathsf{lcptab}[s] = q$ for at least one s, $l + 1 \leq s \leq r$,
   - $r = n$ or $\mathsf{lcptab}[r + 1] < q$.
2. if $l = r$, then $q = n_1 + 1 - \mathsf{suftab}[l]$.

Note that this definition of lcp-intervals also includes singleton intervals. By contrast, the original definition of lcp-intervals does not include singleton intervals (Abouelhoda et al., 2004). We will also use the shorthand *lcp-interval* for an lcp-interval $[l..r]$ of unknown lcp-value. The size of an lcp-interval $[l..r]$ is $r - l + 1$. A q-interval $[l..r]$ is said to be *embedded* in a q'-interval $[l'..r']$ if it is a subinterval of $[l'..r']$ (i.e., $l' \leq l \leq r \leq r'$) and $q > q'$. The lcp-interval $[l'..r']$ is then called the interval *enclosing* $[l..r]$. If

[3] *www.vmatch.de.*

$[l'..r']$ encloses $[l..r]$ and there is no interval embedded in $[l'..r']$ that also encloses $[l..r]$, then $[l..r]$ is called a *child interval* of $[l'..r']$.

7.1. Representing suffix tree nodes

Lcp-intervals of size one (singleton intervals) correspond to leaves of ST. q-intervals of size greater than one correspond to interior nodes of ST of depth q. In our implementation, we represent suffix tree nodes by lcp-intervals. The parent-child relationship of lcp-intervals constitutes a conceptual (or virtual) tree which we call the *lcp-interval tree*. The lcp-interval tree is equivalent to the suffix tree. Hence, in the following, we reuse several suffix tree notions for the lcp-interval tree.

Example 4. Figure 4 shows the enhanced suffix array for the sequence $S_1\$ = acaaacatat\$$, and Figure 5 shows the corresponding lcp-interval tree.

The *Vmatch* code base provides a function which delivers right maximal matches of S_2 in the enhanced suffix array of S_1. It reports the lcp-interval where the matching substring ends. This is exactly what we need for computing rare matches. The corresponding algorithm, described in Kurtz and Lonardi (2004) on the basis of suffix trees, runs in $O(n_2 + z)$ time, where z is the number of reported matches, provided that the enhanced suffix array for S_1 is already precomputed.

7.2. Representing leaf sets

Each leaf set occurring in our algorithms is represented by an interval $\langle l, r\rangle$, where $0 \le l \le r \le n_1$. $\langle l, r\rangle$ represents the leaf set $\{\mathsf{suftab}[s] \mid l \le s \le r\}$. Note that there are leaf sets $\langle l, r\rangle$ such that $[l..r]$ is not an lcp-interval. The leaf sets generated by our algorithms are subject to difference and intersection operations. The difference operation $leafset(j') \setminus leafset(j'')$ (see Section 5.3.2) means to subtract $\langle l', r'\rangle$

i	suftab	lcptab	$S_{1\mathsf{suftab}[i]}$
0	2		aaacatat$
1	3	2	aacatat$
2	0	1	acaaacatat$
3	4	3	acatat$
4	6	1	atat$
5	8	2	at$
6	1	0	caaacatat$
7	5	2	catat$
8	7	0	tat$
9	9	1	t$
10	10	0	$

FIG. 4. The enhanced suffix array for $S_1\$ = acaaacatat\$$.

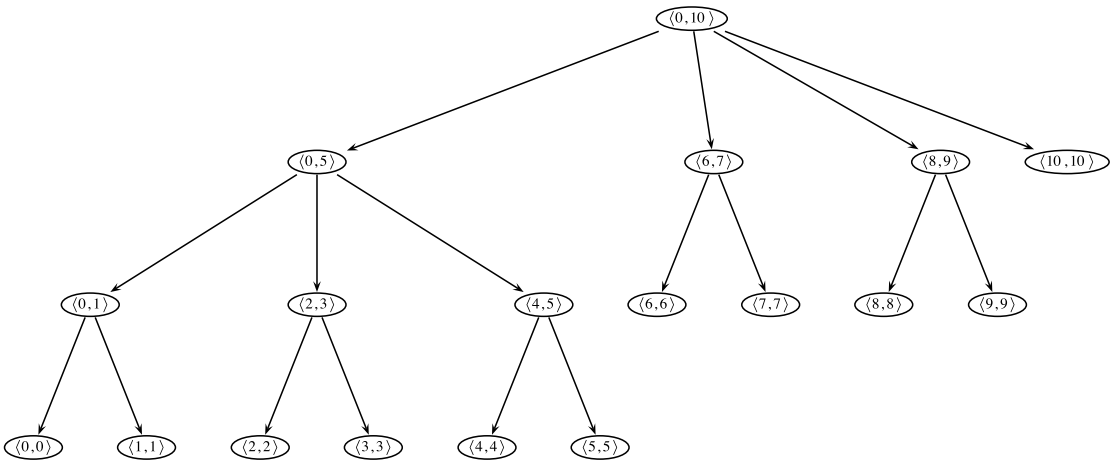

FIG. 5. The lcp-interval tree for $S_1\$ = acaaacatat\$$. The corresponding suffix array is $[2, 3, 0, 4, 6, 8, 1, 5, 7, 9, 10]$ (see also Fig. 4).

and $\langle l'', r''\rangle$ representing *leafset*(j') and *leafset*(j''), respectively. Since the node j'' is in the subtree below j', we have $l' \le l'' \le r'' \le r'$. Thus, the difference operation delivers two leaf sets, $\langle l', l''-1\rangle$ and $\langle r''+1, r'\rangle$, one of which is empty if $l' = l''$ or $r'' = r'$. The split of a leaf set leads to two match candidate specifications, rather than one as described in Section 5.3.2. This however does not affect the correctness or efficiency of our algorithms. The intersection on k leaf sets (see Section 6.3, step (4)) can be computed in $O(k)$ steps. Each step computes the intersection $\langle \max\{l, l'\}, \min\{r, r'\}\rangle$ of two leaf sets $\langle l, r\rangle$ and $\langle l', r'\rangle$.

7.3. Storing min-values and L-lists

In the pairwise matching and deletion phase, we have to store, lookup, and update min-values and L-lists for a subset of all possible lcp-intervals. Since we expect only a small fraction of all possible lcp-intervals to be annotated, we use a balanced binary search tree (i.e., red-black tree) to store, lookup, and update these values during the pairwise matching and deletion phase. Since an lcp-interval is uniquely determined by its left and right boundaries l and r, we use the pair (l, r) as search key. Figure 6 shows the binary search tree obtained when matching $S_2 = aaaaacttaacaacat$ against the enhanced suffix array of S_1.

To save space, we do not store the lcp-value with each search key, but recompute it on the fly by evaluating $\min\{\mathsf{lcptab}[s] \mid l+1 \le s \le r\}$. Since we only consider lcp-intervals $[l..r]$ of size at most t_1, this requires $O(t_1)$ time. L-lists are stored as linked lists. We do not store the length of a list, but recompute it on the fly when required.

Whenever we find a right maximal matching substring of length $\ge \ell$ at position p_2 in S_2 ending at the interval $[l..r]$, we insert this match into the search tree. We also insert additional lcp-intervals (associated with a min-value of ℓ and an empty L-list), to guarantee that all relevant lcp-intervals are finally contained in the search tree. In particular, we insert a q'-interval $[l'..r']$, if the following holds:

- $[l'..r']$ is on the path from the root of the lcp-interval tree to $[l..r]$,
- $r' - l' + 1 \le t_1$,
- $q' \ge \ell$,
- $[l'..r']$ is not already contained in the search tree.

These additional lcp-intervals can be found in $O(t_1)$ time by looking up the appropriate values in table lcptab.

7.4. Representing parent-child relationships for lcp-intervals

To efficiently compute *elem_in_sublists*-values and in turn decide whether an lcp-interval is relevant, we have to perform a bottom-up traversal of the lcp-interval tree, restricted to the lcp-intervals contained in

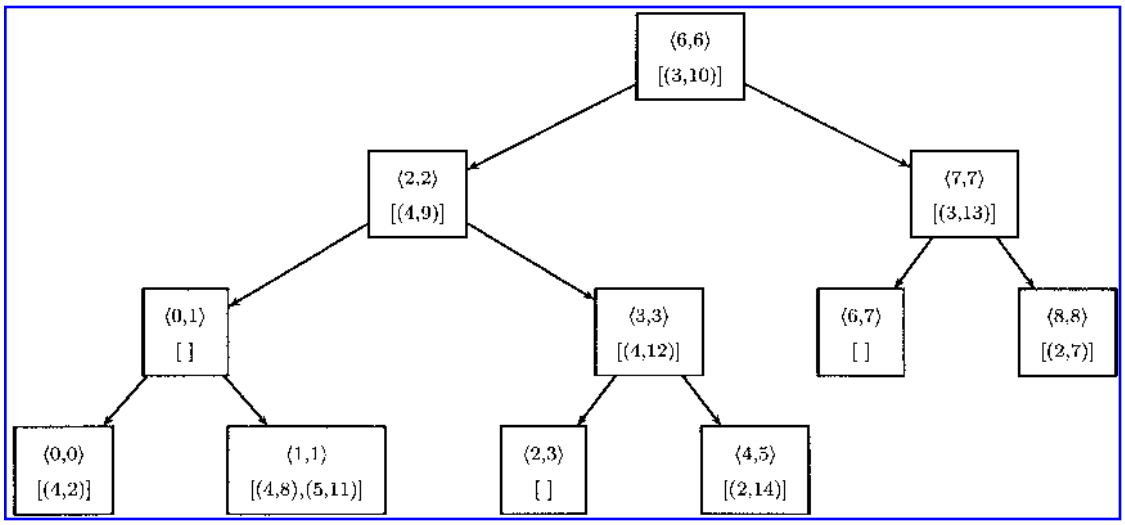

FIG. 6. The binary search-tree obtained when matching $S_2 = aaaaacttaacaacat$ against the enhanced suffix array of $S_1\$ = acaaacatat\$$. For each node of the binary search tree, we show the stored lcp-interval and the corresponding list of matches (see also Fig. 2).

the search tree. This requires information about the parent-child relationship of a subset of all lcp-intervals. This relationship is already available in the search tree, because we store the search keys according to the total order $\prec$, defined as follows: $(l', r') \prec (l, r)$ if and only if $l' < l$ or $l' = l$ and $r' > r$. This order is consistent with the parent-child relationship of the lcp-intervals in the lcp-interval tree. That is, if we enumerate the elements of the search tree in $\prec$-order, then a parent comes before all of its children. This allows one to simulate the required bottom up traversal over the suffix tree in time proportional to the size of the search tree.

While the matching and deletion phases of our algorithms both require random insert and update operations on the min-values and L-lists, the output phase (pairwise or multiple) merely needs sequential read access. Moreover, in the multiple output phase, we have to combine the results of $k-1$ matching and deletion phases. Therefore, we store the result of the matching and deletion phases as a bit stream. For an lcp-interval $[l..r]$, we store l, $r-l$, $\mathsf{min}[[l..r]]$, and all elements of L-lists in the bit stream. An additional bit for each interval is used to mark the end of the bit stream encoding for the current interval. Since we know the distribution of the values stored in the bit stream, each kind of value can be stored as a fixed number of bits. Especially for the min-value, the difference $r-l$, and the length component of the elements of the L-lists, we only need a small number of bits. A Huffman-coding would lead to even better results. The bit stream encodes all relevant intervals w.r.t. to the order $\prec$. Hence, we can simulate a bottom-up traversal over the lcp-interval tree restricted to all relevant lcp-intervals (as required to compute type 2 and type 3 match candidate specifications), without explicitly storing parent-child relationships.

7.5. Combining min-values and L-lists in the multiple output phase

In the multiple output phase, we first have to identify the lcp-intervals that are relevant w.r.t. S_i, for all i, $2 \le i \le k$. Since for each i, the relevant lcp-intervals for S_i are stored w.r.t. to the same order $\prec$, this identification task can be performed efficiently by iteratively merging the sequence of relevant lcp-intervals. That is, we first merge the sequence of relevant lcp-intervals for S_2 and the sequence of relevant lcp-intervals for S_3. This gives us the lcp-intervals which are relevant both for S_2 and S_3. For each such lcp-interval, we store the maximum of the min-values and two references to the bit streams encoding the relevant lcp-intervals. The combined sequence is then merged with the sequence of relevant intervals for S_4, resulting in a sequence of intervals that are relevant w.r.t. all S_i, for $2 \le i \le 4$. This process is continued until the identification task is completed. As a result, we obtain a sequence of lcp-intervals. For each such interval, say $[l..r]$, we store $\max\{\mathsf{min}_i[[l..r]] \mid 2 \le i \le k\}$ and $k-1$ references $r_2, \ldots, r_k$, where r_i refers to the bit streams encoding the lcp-interval $[l..r]$ relevant for S_i.

There are two motivations for using the bit stream. At first, it reduces the size of the representation. To see this, let us compare the size of the bit stream to the size of the random access representation of the binary search tree. Assuming a pointer size of 4 bytes, each node in this binary search tree requires $32 + 12(m - 1)$ bytes, where m is the number of matches it stores. In contrast, the simplest form of bit stream encodes all relevant values as 4 bytes integers. It requires $8 + 8m$ bytes for a singleton interval and $12 + 8m$ bytes for any other interval. Thus depending on the number of matches and kinds of intervals to store, we obtain a space improvement by a factor of about 2–3 times. Second, and more important than the space reduction, is the fact that in the multiple output phase the bit stream can be processed sequentially. That is, instead of keeping several binary search trees resulting from the match/deletion phases in memory, we sequentially read the bit streams, keeping only a small part of them in main memory.

8. EXPERIMENTAL RESULTS

The algorithms described here were implemented in C. The resulting program *ramaco* (rare match computation) will be made available free of charge for academia. The goal of this section is to show that the described algorithms are of practical use concerning their time and space requirements.

For our first experiment, the sequences from the *MOSAIC* project (Chiapello et al., 2005) were used. This project aims to provide high-quality pairwise and multiple alignments of groups of related bacterial genomes (*http://genome.jouy.inra.fr/mosaic/*). The *MOSAIC* sequences consist of 24 groups of genomes (12 groups of two genomes, 6 groups of three genomes, 4 groups of four genomes, 2 groups of six genomes). The *MOSAIC* sequences were downloaded from Genbank, and *ramaco* was applied to each group. More precisely, the enhanced suffix array of the first genome of each group was constructed, and the remaining sequences were used as queries (in forward direction). For each combination of index and queries, three runs where performed:

- In the first run, rare maximal matches were computed in a single program call (*all*-run).
- In the second run, only the matching and deletion phases for the index and each query were performed. The result of this phase was stored on files (*search*-run).
- In the third run, only the output phase was performed, using the result from the appropriate search-runs (*output*-run).

Table 1 shows the result when computing rare matches for four *MOSAIC* groups with $\ell = 14$ and $t_i = 5$. Each of the chosen groups was the largest (in terms of base pairs [bp]) among all groups with the same number of sequences. The results for the other groups do not significantly differ from the results of the groups shown here. The first column of Table 1 gives the name of the group, the number of rare *multiMEMs*, and the number of *multiMEMs* in the group. We additionally show how many more *multiMEMs* than rare *multiMEMs* exist. The second column gives the Genbank accession number of the genomes making up the group. In the third column, the size of the genome (in bp) is reported. The fourth column shows for which kind of run (*all*, *search*, or *output*) the results are given. Column 5 gives the running time (in sec) on a Pentium 4 computer (2.4 GHz CPU, 4 GB RAM) running Linux. Column 6 reports the overall space peak (in megabytes) of the corresponding run. Column 7 shows the "rare match space peak" (abbreviated RM-space), that is, the space peak minus the space for the index and the query sequence. While the space for the index and the query sequence only depends on the input size, the rare match space peak depends on the number of pairwise matches. This is because our algorithm has to store the pairwise matches, until their rareness can finally be verified. As a consequence, the rare match space peak gives the additional space required for handling the rareness constraints. To allow a comparison between the groups of different sizes, column 8 reports the rare match space peak per rare match (in bytes).

Table 1 shows that the ratio of the number of *multiMEMs* and rare *multiMEMs* (rareness ratio) significantly differs. In the first two groups, almost all *multiMEMs* are rare. In the last two groups, with four and six genomes, less than 2% and less than 1% are rare, respectively. That is, for the last two groups, the rareness constraint represents a very effective filter for matches. Concerning the required resources, it is obvious that all sets of genomes can easily be handled by *ramaco*, although the genomes inside one group are very similar. The running times are in a range of a few seconds to about 3 min for the set with six genomes. The running times for the separate *search* and *output* runs approximately sum up to

TABLE 1. RUNNING TIME AND SPACE RESULTS WHEN APPLYING *ramaco* TO FOUR *MOSAIC* GROUPS WITH $\ell = 14$ AND $t_i = 5$

Group	*Genomes*	*Length (bp)*	*Phase*	*Time (sec)*	*Space (MB)*	*RM-space (MB)*	*Per match (bytes)*
$\ell = 14, t_i = 5$, matches on forward strand							
Pseudomonas syringae	NC_005773	5,928,787	All	13.00	119.08	70.90	135
[549036 rare MEMs,	NC_007005	6,093,698	Search	13.53	95.08	46.90	90
627290 MEMs (1.1×)]			Output	0.66	64.74	24.21	46
Bacillus cereus [896291 rare	NC_003909	5,224,283	All	53.14	309.54	262.12	307
MEMs, 1308576 MEMs	NC_004722	5,411,809	Search	31.30	163.60	121.24	142
(1.5×)]	NC_006274	5,300,915	Output	19.53	212.22	171.79	201
Escherichia coli [1198345	NC_004431	5,231,428	All	108.93	543.50	491.31	430
rare MEMs, 75520384	NC_000913	4,639,675	Search	53.22	247.02	204.50	179
MEMs (63.0×)]	NC_002655	5,528,445	Output	49.58	405.01	359.81	315
	NC_002695	5,498,450					
Staphylococcus aureus	NC_002758	2,878,040	All	184.39	643.94	609.02	62
[10260430 rare MEMs,	NC_003923	2,820,462	Search	61.14	293.02	268.82	27
1140814012 MEMs	NC_002745	2,814,816	Output	114.12	434.35	404.17	41
(111.2×)]	NC_002951	2,809,422					
	NC_002952	2,902,619					
	NC_002953	2,799,802					

ramaco, Rare match computation.

the time for the *all* run. More important, the separation of the computation into two program calls leads to a considerably reduced space requirement. The last column of Table 1 shows that the space per match considerably differs between the different groups, ranging from 27 to 430 bytes. This large variation is due to the fact that the rare match space (RM-space) is mainly dependent on number of pairwise matches, and to a lesser extent on the number of *multiMEMs*. But given the usual context that the entire set of matches are stored (on file or in RAM) to be further processed (i.e., by chaining methods), the per match space seems acceptable in practice.

To study the effect of the length threshold, we ran *ramaco* for the same *MOSAIC* groups and with $\ell = 20$ and $t_i = 5$ (Table 2).

Interestingly, the number of rare *multiMEMs* decreases more than the number of *multiMEMs*. As a consequence, the rareness ratio increases, except for the *S. aureus*-group, where it hardly changes. This means that rareness filtering is also relevant for larger length thresholds. The running times only slightly decrease by a factor of 1.2–1.5, compared to the runs reported in Table 1. The decrease in the space requirement is by a factor of 1.1–2.0. Unfortunately, the per match space increases by a factor of up to 10, compared to the run with $\ell = 14$. Again, this can be explained by the fact that there are many pairwise matches, which do not lead to rare *multiMEMs*.

Our second experiment deals with the computation of rare MEMs for two groups of X-chromosomes from different vertebrate genomes. More precisely, as a first group, the X-chromosomes of *Homo sapiens* (human), of *Mus musculus* (mouse), and of *Rattus norvegicus* (rat) were used. The second group consists of the members of the first group plus the X-chromosome of *Canis familiaris* (dog). The same procedure as in the first experiment was applied. Additionally, rare *multiMEMs* on the reverse strands were computed. The results are shown in Table 3; see above for the explanation of the values given in the different columns. *ramaco* efficiently computed all rare MEMs of length at least 20 with rareness threshold $t_i = 10$.

While the program is able to compute *multiMEMs* (without any rareness constraints), it failed to do so for the X-chromosomes because there are simply too many *multiMEMs*. Table 3 shows that *ramaco* can handle large sequence sets of about 600 million bp. The running time (for the *all*-runs) is about 9–13 min, and the space requirement is on the order of 2 gigabytes. The additional dog genome led to an increased number of rare *multiMEMs* by a factor of 2.7. The number of matches on the forward strand only slightly differs from the number of matches on the reverse strand. This shows that it is necessary to

TABLE 2. RUNNING TIME AND SPACE RESULTS WHEN APPLYING *ramaco* TO FOUR *MOSAIC* GROUPS WITH $\ell = 20$ AND $t_i = 5$

Group	*Genomes*	*Length (bp)*	*Phase*	*Time (sec)*	*Space (MB)*	*RM-space (MB)*	*Per match (bytes)*
$\ell = 20, t_i = 5$							
Pseudomonas syringae	NC_005773	5,928,787	All	9.77	82.35	34.17	699
[51274 rare MEMs, 75308	NC_007005	6,093,698	Search	9.93	70.99	22.81	466
MEMs (1.47×)]			Output	0.37	52.11	11.58	237
Bacillus cereus [55581 rare	NC_003909	5,224,283	All	36.59	209.59	162.17	3060
MEMs, 230193 MEMs	NC_004722	5,411,809	Search	25.29	126.06	83.70	1579
(4.14×)]	NC_006274	5,300,915	Output	11.70	141.69	101.26	1910
Escherichia coli [103631 rare	NC_004431	5,231,428	All	91.02	450.38	398.19	4029
MEMs, 10211804 MEMs	NC_000913	4,639,675	Search	51.58	214.15	171.63	1737
(98.54×)]	NC_002655	5,528,445	Output	43.03	332.42	287.22	2906
	NC_002695	5,498,450					
Staphylococcus aureus	NC_002758	2,878,040	All	146.83	565.17	530.25	1464
[379856 rare MEMs,							
42011419 MEMs	NC_003923	2,820,462	Search	67.78	273.84	249.64	689
(110.60×)]	NC_002745	2,814,816	Output	78.54	372.19	342.01	944
	NC_002951	2,809,422					
	NC_002952	2,902,619					
	NC_002953	2,799,802					

ramaco, Rare match computation.

TABLE 3. RUNNING TIME AND SPACE RESULTS WHEN APPLYING *ramaco* TO TWO COLLECTIONS OF X-CHROMOSOMES WITH $\ell = 20$ AND $t_i = 10$

Group	*Genomes*	*Length (bp)*	*Phase*	*Time (sec)*	*Space (MB)*	*RM-space (MB)*	*Per match (bytes)*
$\ell = 20, t_i = 10$, forward strands							
X-chromosome	NC_000023	154,824,264	All	511.24	1676.69	348.17	58
human/mouse/rat	NC_000086	164,906,252	Search	497.84	1409.61	234.34	39
[6295682 rare MEMs]	NC_005120	160,699,376	Output	18.28	1336.19	283.32	47
X-chromosome	NC_000023	154,824,264	All	744.19	1846.88	397.35	24
human/dog/mouse/rat	NC_006621	126,883,977	Search	713.70	1468.98	293.71	18
[17111495 rare MEMs]	NC_000086	164,906,252	Output	117.48	1423.84	249.96	15
	NC_005120	160,699,376					
$\ell = 20, t_i = 10$, reverse strands							
X-chromosome	NC_000023	154,824,264	All	517.72	1972.00	643.48	108
human/mouse/rat	NC_000086	164,906,252	Search	505.82	1556.79	381.52	64
[6256893 rare MEMs]	NC_005120	160,699,376	Output	18.17	1326.25	273.38	46
X-chromosome	NC_000023	154,824,264	All	743.23	2232.21	782.68	48
human/dog/mouse/rat	NC_006621	126,883,977	Search	714.16	1596.77	421.50	26
[17102466 rare MEMs]	NC_000086	164,906,252	Output	36.69	1413.41	239.53	15
	NC_005120	160,699,376					

ramaco, Rare match computation.

compute matches on both strands. This is done by a script appropriately calling *ramaco* for each possible combination of strands. For the first set of X-chromosomes, this gives a running time of about 30 min. For the second set of X-chromosomes, this gives a running time of about 100 min. Note that the per match space requirement given in Table 3 is generally smaller and has less variance than the per match space requirement for the bacterial sequences of the first experiment. This may be explained by the fact that the X-chromosomes are so similar that pairwise matches lead to *multiMEMs* with high probability.

9. CONCLUSION

We have devised an efficient algorithm for solving an important sequence comparison problem, viz. finding rare maximal exact matches between multiple sequences. Apart from that, we have also presented an efficient implementation technique. This allowed us to implement the algorithm with a space complexity that is independent of the sizes of the sequences. Our experiments show that the rareness constraints significantly reduce the number of matches, and their computation is feasible.

Because *multiMUMs* are special rare *multiMEMs*, any algorithm for finding rare *multiMEMs* can also be used to search for *multiMUMs*. If one is solely interested in *multiMUMs*, however, then a more efficient implementation can be obtained by exploiting the properties of this special case. We refrained from giving the details of how our algorithm can be specialized for finding *multiMUMs* because *multiMUMs* are not particularly suitable for the genome comparisons we have in mind.

DISCLOSURE STATEMENT

No competing financial interests exist.

REFERENCES

Abouelhoda, M. 2007. A chaining algorithm for mapping cDNA sequences to multiple genomic sequences. *Lect. Notes Comp. Sci.*, vol. 4726, 1–13. Springer-Verlag, Berlin.

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. 2004. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms* 2, 53–86.

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. 2006. Enhanced suffix arrays and applications. In: Aluru, S., ed., *Handbook of Computational Molecular Biology*, 7-1–7-21. Chapman & Hall, New York.

Abouelhoda, M., and Ohlebusch, E. 2003. A local chaining algorithm and its applications in comparative genomics. *Lect. Notes Bioinform.* 2812, 1–16. Berlin.

Bourque, B., and Pevzner, P. 2002. Genome-scale evolution: reconstructing gene orders in the ancestral species. *Genome Res.* 12, 26–36.

Chain, P., Kurtz, S., Ohlebusch, E., et al. 2003. An applications-focused review of comparative genomics tools: capabilities, limitations and future challenges. *Brief. Bioinform.* 4, 105–123.

Chang, W., and Lawler, E. (1994). Sublinear approximate string matching and biological applications. *Algorithmica* 12, 327–344.

Chiapello, H., Bourgait, I., Sourivong, F., et al. 2005. Systematic determination of the mosaic structure of bacterial genomes: species backbone versus strain-specific loops. *BMC Bioinform.*, 6, 171.

Darling, A., Mau, B., Blattner, F., et al. 2004. Mauve: multiple alignment of conserved genomic sequence with rearrangements. *Genome Res.* 14, 1394–1403.

Delcher, A., Phillippy, A., Carlton, J., et al. 2002. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res.* 30, 2478–2483.

Deogen, J., Yang, J., and Ma, F. 2004. EMAGEN: an efficient approach to multiple genome alignment. *Proc. 2nd Asia Pacific Bioinform. Conf.* 113–122.

Gusfield, D. 1997. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York.

Höhl, M., Kurtz, S., and Ohlebusch, E. 2002. Efficient multiple genome alignment. *Bioinformatics* 18, S312–S320.

Kurtz, S., and Lonardi, S. 2004. Computational biology. In: Mehta, D., and Sahni, S., eds., *Handbook on Data Structures and Applications*, 58-1–58-17. CRC Press, Boca Raton, FL.

Kurtz, S., Phillippy, A., Delcher, A., et al. 2004. Versatile and open software for comparing large genomes. *Genome Biol.* 5, R12.

Manber, U., and Myers, E. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* 22, 935–948.

Mau, B., Darling, A., and Perna, N. 2005. Identifying evolutionarily conserved segments among multiple divergent and rearranged genomes. *Lect. Notes Bioinform.* 3388, 72–84.

Pevzner, P., and Tesler, G. 2003. Genome rearrangements in mammalian evolution: lessons from human and mouse genomic sequences. *Genome Res.* 13, 3–26.

Smit, A., and Green, P. 2008. RepeatMasker. Available at: *www.repeatmasker.org/*. Accessed February 27, 2008.

Stoye, J., and Gusfield, D. 2002. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoret. Comput. Sci.* 270, 843–856.

Treangen, T., and Messeguer, X. 2006. M-GCAT: interactively and efficiently constructing large-scale multiple genome comparison frameworks in closely related species. *BMC Bioinform.* 7, 433.

Ukkonen, E. 1995. On-line construction of suffix-trees. *Algorithmica* 14, 249–260.

Weiner, P. 1973. Linear pattern matching algorithms. *Proc. 14th IEEE Ann. Symp. Switching Autom. Theory*, 1–11.

Address reprint requests to:
Dr. Enno Ohlebusch
Faculty of Engineering and Computer Sciences
University of Ulm
89069 Ulm, Germany

E-mail: enno.ohlebusch@uni-ulm.de