JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 23, Number 6, 2016

Pp. 472–482
DOI: 10.1089/cmb.2015.0235

A Provably Efficient Algorithm for the *k*-Mismatch Average Common Substring Problem

SHARMA V. THANKACHAN, ALBERTO APOSTOLICO, and SRINIVAS ALURU

ABSTRACT

Alignment-free sequence comparison methods are attracting persistent interest, driven by data-intensive applications in genome-wide molecular taxonomy and phylogenetic reconstruction. Among all the methods based on substring composition, the *average common substring* (ACS) measure admits a straightforward linear time sequence comparison algorithm, while yielding impressive results in multiple applications. An important direction of this research is to extend the approach to permit a bounded edit/hamming distance between substrings, so as to reflect more accurately the evolutionary process. To date, however, algorithms designed to incorporate $k \geq 1$ mismatches have $O(n^2)$ worst-case time complexity, where n is the total length of the input sequences. On the other hand, accounting for mismatches has shown to lead to much improved classification, while heuristics can improve practical performance. In this article, we close the gap by presenting the first provably efficient algorithm for the *k-mismatch average common string* (ACS_k) problem that takes $O(n)$ space and $O(n \log^k n)$ time in the worst case for any constant k. Our method extends the generalized suffix tree model to incorporate a carefully selected bounded set of perturbed suffixes, and can be applied to other complex approximate sequence matching problems.

Key words: suffix trees, alignment free methods, phylogenetic reconstruction, sequence similarity, evolutionary distance.

1. INTRODUCTION

TREATING BIOSEQUENCES AS ''DOCUMENTS OF EVOLUTION'' (Zuckerkandl and Pauling, 1965) is one of the oldest ideas of molecular biology. In this vein, measures of sequence similarity and distance primarily emerged in coding theory have provided the natural backbone in the development of tools for deriving taxonomies and phylogenies from systematic biosequence comparison. Beginning with Fitch and Margoliash (Fitch et al., 1967), the established methods for automated inference of phylogeny have been based on sequence alignment of protein orthologues or their genes, tRNAs, 16S rRNAs, 23S rRNAs, etc.. In 1994, the DIMACS special year on computational biology devoted one of its four main workshops to sequence comparison. By the end of which, more than 1200 noteworthy publications were counted on this subject (Apostolico and Giancarlo, 1998).

College of Computing, Georgia Institute of Technology, Atlanta, Georgia.
A preliminary version of this article appeared in Aluru et al., 2015.

In subsequent years, the context has drastically evolved from the original condition of scarcity of data, and a plethora of new problems have emerged. To begin with, dithering the parameters involved in alignment proved not a simple task. Common orthologues are difficult to identify for prokaryotes due to their wide genetic diversity. Lateral transfer of proteins may lead to completely wrong results, for example the entire ribosomal operon (5S + 16S + 23S) in *E. coli* may be artificially replaced by that from other species (Asai et al., 1999). The resulting trees are prone to intrinsic biases brought about by the specific data being analyzed and so on. To summarize, classical alignment distances become both computationally expensive and scarcely significant when they are applied to entire genomes and are being complemented or even entirely supplanted by global (algebraic) similarity measures that refer, implicitly or explicitly, to the subword composition of sequences, sometimes collectively referred to as ''alignment-free'' comparisons (see, e.g., Apostolico and Cunial, 2011; Apostolico and Denas, 2008; Apostolico et al., 2010; Blaisdell, 1986; Clift et al., 1986; Cunial and Apostolico, 2012; Edgar, 2004; Ferragina et al., 2007; Gatlin, 1972; Hao and Qi, 2004; Höhl and Ragan, 2007; Höhl et al., 2006; Li et al., 2004; Morgenstern et al., 2014; Otu and Sayood, 2003; Qi et al., 2004; Ulitsky et al., 2006; Van Helden, 2004; Vinga and Almeida, 2003; Wu et al., 1997).

The *average common substring* measure proposed by Burstein et al. and Ulitsky et al. (Burstein et al., 2005; Ulitsky et al., 2006) is a simple alignment-free sequence comparison method that nevertheless achieved high accuracy in large-scale phylogenetic reconstruction. Formally, let X and Y denote two sequences over the alphabet Σ. We use the notation $|\mathsf{X}|$ to denote the length of X, $\mathsf{X}[i]$ $(1 \le i \le |\mathsf{X}|)$ to denote its ith leftmost character, and $\mathsf{X}[i \ldots j]$ to denote the substring $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j]$. Let X_i denote the suffix of X starting at i^{th} position, that is, $\mathsf{X}_i = \mathsf{X}[i \ldots |\mathsf{X}|]$. Let

$$\lambda(\mathsf{X}_i, \mathsf{Y}) = \max_{1 \le j \le |\mathsf{Y}|} |\mathsf{LCP}(\mathsf{X}_i, \mathsf{Y}_j)|,$$

where $\mathrm{LCP}(\mathsf{X}_i, \mathsf{Y}_j)$ denote the longest common prefix between suffixes X_i and Y_j. The *average common substring*, $\mathsf{ACS}(\mathsf{X}, \mathsf{Y})$, is defined as:

$$\mathsf{ACS}(\mathsf{X}, \mathsf{Y}) = \frac{1}{|\mathsf{X}|} \sum_{i=1}^{|\mathsf{X}|} \lambda(\mathsf{X}_i, \mathsf{Y})$$

Therefore, $\mathsf{ACS}(\mathsf{X}, \mathsf{Y})$ is the average of the length of the longest prefix of a suffix of X occurring in Y. One then takes $\mathsf{ACS}(\mathsf{X}, \mathsf{Y})/\log |\mathsf{Y}|$ to normalize with respect to the length of Y. A distance metric based on ACS (Ulitsky et al., 2006) is obtained by first taking the inverse of the *similarity* measure $\mathsf{ACS}(\mathsf{X}, \mathsf{Y})/\log |\mathsf{Y}|$ and then subtracting a term to guarantee the condition $\mathsf{Dist}(\mathsf{X}, \mathsf{X}) = 0$. Specifically, this yields

$$\mathsf{Dist}'(\mathsf{X}, \mathsf{Y}) = \frac{\log |\mathsf{Y}|}{\mathsf{ACS}(\mathsf{X}, \mathsf{Y})} - \frac{\log |\mathsf{X}|}{\mathsf{ACS}(\mathsf{X}, \mathsf{X})}$$

where the correction term $\log |\mathsf{X}|/\mathsf{ACS}(\mathsf{X}, \mathsf{X}) = 2\log |\mathsf{X}|/|\mathsf{X}|$ vanishes as $|\mathsf{X}| \to \infty$. Following this, one compensates for symmetry by taking

$$\mathsf{Dist}(\mathsf{X}, \mathsf{Y}) = \frac{\mathsf{Dist}'(\mathsf{X}, \mathsf{Y}) + \mathsf{Dist}'(\mathsf{Y}, \mathsf{X})}{2}$$

as the final distance. We can easily compute ACS (and then Dist' and Dist) using a (generalized) suffix tree–based algorithm in time and space linear in the total length of X and Y.

Since its introduction, the ACS measure has gained considerable attention (Apostolico, 2010; Bonham-Carter et al., 2013; Chang and Wang, 2011; Comin et al., 2012; Domazet-Lošo and Haubold, 2009; Fracasso, 2014; Guyon et al., 2009). An important direction of research is to improve the measure by allowing a bounded number k of mismatches in sequence comparison. Define $\lambda_k(\mathsf{X}_i, \mathsf{Y})$ as:

$$\lambda_k(\mathsf{X}_i, \mathsf{Y}) = \max_{1 \le j \le |\mathsf{Y}|} |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|,$$

where $\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)$ denotes the longest common prefix between suffixes X_i and Y_j with allowance for up to k mismatches. The *k-mismatch average common substring* of X and Y, denoted $\mathsf{ACS}_k(\mathsf{X}, \mathsf{Y})$, is defined as:

$$\mathsf{ACS}_k(\mathsf{X}, \mathsf{Y}) = \frac{1}{|\mathsf{X}|} \sum_{i=1}^{|\mathsf{X}|} \lambda_k(\mathsf{X}_i, \mathsf{Y})$$

An algorithm quite recently proposed by Leimeister and Morgenstern (2014) for computing $\mathsf{ACS}_k(\mathsf{X}, \mathsf{Y})$ requires $O(|\mathsf{X}|\times|\mathsf{Y}|\times k)$ worst-case run-time, which is quadratic when $|\mathsf{X}|$ and $|\mathsf{Y}|$ are of the same order, even for $k=1$ (also see Pizzi, 2015). However, it is amply documented in Horwege et al. (2014) and Leimeister and Morgenstern (2014) that the $\mathsf{ACS}_k(\mathsf{X}, \mathsf{Y})$ paradigm does yield better phylogenetic classification than both the exact version of the problem and classical approaches using multiple sequence alignment and maximum likelihood. Although Leimeister and Morgenstern (2014) (also see the recent work by Thankachan et al., 2015) proposed faster heuristics to approximate ACS_k, designing a provably efficient algorithm remained open. Our goal in this work is to compute $\mathsf{ACS}_k(\mathsf{X}, \mathsf{Y})$ (hence also the naturally associated $\mathsf{Dist}_k(\mathsf{X}, \mathsf{Y})$) in time and space that is as close to linear in $n=|\mathsf{X}| + |\mathsf{Y}|$ as possible. Our main contribution is summarized in the following theorem:

Theorem 1 *Let* X *and* Y *be two sequences of n characters in total. Let* X_i *be the suffix of* X *starting at location i, and* $\lambda_k(\mathsf{X}_i, \mathsf{Y})$ *be the length of the longest prefix of* X_i *that appears in* Y *with at most k mismatches. Then* $\lambda_k(\mathsf{X}_i, \mathsf{Y})$ *can be computed for all values of i in* $O(n \log^k n)$ *overall time using* $O(n)$ *working space for any constant k.*

Using Theorem 1, we can compute $\mathsf{ACS}_k(\mathsf{X}, \mathsf{Y})$ in $O(n \log^k n)$ time. As a byproduct, we also improved upon the best-known results for the k-mismatch longest common substring problem (LCS_k), which previously took linear space and $O(n \log n)$ time for $k=1$, but takes almost quadratic time otherwise (Babenko and Starikovskaya, 2008; Flouri et al., 2015; Grabowski, 2015). Notice that $\mathrm{LCS}_k(\mathsf{X}, \mathsf{Y})$ is the substring corresponding to $\max\{\lambda_k(\mathsf{X}_i, \mathsf{Y}) \mid 1 \le i \le |\mathsf{X}|\}$, hence it can be computed within the same space-time complexity of Theorem 1.

Theorem 2 **(k-mismatch longest common substring)** *Given two sequences of total length n, their longest common substring with at most k mismatches can be computed in* $O(n \log^k n)$ *time using* $O(n)$ *space for any constant k.*

2. KEY CONCEPTS AND PROPERTIES

A generalized suffix tree of X and Y (from now onward called GST) is a compact trie that stores all (nonempty) suffixes of X and Y. It takes $O(n)$ space and can be constructed in $O(n)$ space and time (McCreight, 1976; Weiner, 1973), where $n=|\mathsf{X}| + |\mathsf{Y}|$. Using GST, we can compute $|\mathsf{LCP}(\mathsf{X}_i, \mathsf{Y}_j)|$ for any (i, j) pair in constant time. Specifically, $|\mathsf{LCP}(\mathsf{X}_i, \mathsf{Y}_j)|$ is the *string depth* of the lowest common ancestor (LCA) of leaves representing suffixes X_i and Y_j in the GST. Then, $|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|$ for any (i, j) pair can be computed in $O(k)$ time using the following recursion.

$$|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)| = \begin{cases} z, \text{ where } z = |\mathsf{LCP}(\mathsf{X}_i, \mathsf{Y}_j)| & \text{if } k=0 \\ z+1+|\mathsf{LCP}_{k-1}(\mathsf{X}_{i+z+1}, \mathsf{Y}_{j+z+1})| & \text{if } k>0 \end{cases}$$

Therefore, $\mathsf{ACS}_k(\cdot,\cdot)$ can be computed in $O(n^2 k)$ time. Clearly, this approach is computationally expensive. To describe our faster approach, we first introduce some definitions and prove resulting useful properties. Throughout this article, we treat k as a **constant**.

2.1. Modified suffix

Definition 1 *Let* Δ *be a set of (position, character) pairs. Then, for any string* X*, a modified suffix* X_i^{Δ} *of* X *is the string obtained by changing characters in* X_i *as specified by* Δ*. Specifically,* $\forall (p, \sigma) \in \Delta$*, p-th character in* X_i *(i.e.,* $\mathsf{X}_i[p] = \mathsf{X}[i + p - 1]$*) is changed to* σ*.*

For example, if $\mathsf{X}[1\ldots7] = GATATTT$, then $\mathsf{X}_3 = TATTT$ and $\mathsf{X}_3^{\{(2, G), (4, C)\}} = TGTCT$.

Lemma 1 *Given two modified suffixes* X_i^{Δ} *and* $\mathsf{Y}_j^{\Delta'}$*, we can compute* $|\mathsf{LCP}(\mathsf{X}_i^{\Delta}, \mathsf{Y}_j^{\Delta'})|$ *in* $O(|\Delta \cup \Delta'|)$ *time using a generalized suffix tree of* X *and* Y*.*

Proof Let $t = |\Delta \cup \Delta'|$. Consider breaking each suffix X_i and Y_j into at most $(t+1)$ segments by using the t position values in $\Delta \cup \Delta'$ as breakpoints. Note the substrings of X_i^{Δ} and $\mathsf{Y}_j^{\Delta'}$ between any two consecutive breakpoints are substrings of X and Y, respectively. By preprocessing GST of X and Y to answer lowest common ancestor queries in constant time, $|\mathsf{LCP}(\mathsf{X}_i^{\Delta}, \mathsf{Y}_j^{\Delta'})|$ can be computed using at most $(t+1)$ such queries. We remark that the result holds true even if both modified suffixes are from the same string.

Lemma 2 *Given a set $\mathcal{S}$ of modified suffixes, where the number of modifications in each suffix is a constant, we can lexicographically sort them all in $O(|\mathcal{S}| \log |\mathcal{S}|)$ time.*

Proof Using Lemma 1, we can compute the $|\mathsf{LCP}(\cdot,\cdot)|$ between any two strings in $\mathcal{S}$ in constant time. Therefore, by comparing their next characters, we can determine the lexicographic ordering between them. Since two strings can be compared in constant time, an efficient comparison-based sorting (e.g., merge sort) can be used. ■

2.2. $(i, j)_h$-*Maxpair*

Definition 2 *Let $l_1 < l_2 < \ldots < l_h$ be the first h positions in which suffixes X_i and Y_j differ, i.e., $\mathsf{X}_i[l_m] \neq \mathsf{Y}_j[l_m]$ (or equivalently, $\mathsf{X}[i + l_m - 1] \neq \mathsf{Y}[j + l_m - 1]$) for all $1 \leq m \leq h$. If the modifications specified in Δ and Δ' are performed only at positions $l_1, l_2, \ldots l_h$ in such a way that $\forall m$, $\mathsf{X}_i^{\Delta}[l_m] = \mathsf{Y}_j^{\Delta'}[l_m]$, then we term X_i^{Δ} and $\mathsf{Y}_j^{\Delta'}$ an $(i, j)_h$-*maxpair, and the following is true: $|\mathsf{LCP}_h(\mathsf{X}_i, \mathsf{Y}_j)| = |\mathsf{LCP}(\mathsf{X}_i^{\Delta}, \mathsf{Y}_j^{\Delta'})|$.

Let $q = |\mathsf{LCP}_h(\mathrm{X}_i, \mathsf{Y}_j)|$. Then the prefixes $\mathsf{X}[i \ldots i + q - 1]$ and $\mathsf{Y}[j \ldots j + q - 1]$ of X_i and Y_j respectively, differ in (at most) h positions. An $(i, j)_h$-maxpair contains modifications to X_i and Y_j so that the characters in these positions no longer differ. As there are $|\Sigma|$ possible ways of effecting this change for each position, the total number of potential $(i, j)_h$-maxpairs is $|\Sigma|^h$. For example, if $\mathsf{X}_i = AATT\ldots$, $\mathsf{Y}_j = AAAT\ldots$, and $\Sigma = \{A, C, G, T\}$, then $(i, j)_1$-maxpair's are $(\mathsf{X}_i^{\{(3, A)\}}, \mathsf{Y}_j)$, $(\mathsf{X}_i, \mathsf{Y}_j^{\{(3, T)\}})$, $\mathsf{X}_i^{\{(3, C)\}}, \mathsf{Y}_j^{\{(3, C)\}})$, and $(\mathsf{X}_i^{\{(3, G)\}}, \mathsf{Y}_j^{\{(3, G)\}})$.

3. AN OVERVIEW OF OUR ALGORITHM

Definition 3 *For a fixed k, a* **universe** (*denoted by $\mathcal{U}$*) *is a set of modified suffixes, such that for any $1 \leq i \leq |\mathsf{X}|$ and $1 \leq j \leq |\mathsf{Y}|$, $\mathcal{U}$ contains two modified suffixes that together form an $(i, j)_k$-*maxpair. *A* **partition** *of $\mathcal{U}$ is a collection of its subsets $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_t$ such that $\forall i, j$, $\exists$ an $(i, j)_k$-*maxpair *in at least one $\mathcal{P}_f$, $1 \leq f \leq t$.*

The key intuition behind our algorithm is that by processing a given $\mathcal{U}$ in time and space linear (or near-linear) to its size, we can compute ACS_k. Partitioning $\mathcal{U}$ provides a more convenient way to compute ACS_k by splitting the potentially large $\mathcal{U}$ into multiple smaller sets and processing them independently. Clearly, there exists a universe of size $O(n^2)$. However, our algorithm is based on the existence of a partition as summarized in the following lemma.

Lemma 3 *There exists a universe $\mathcal{U}$ and its partition $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_t\}$, such that*

$$\max_{1 \leq f \leq t} |\mathcal{P}_f| = O(n) \quad \text{and} \quad \sum_{1 \leq f \leq t} |\mathcal{P}_f| = O(n \log^k n)$$

Our algorithm can be viewed as consisting of two phases. In the first phase, we generate members of the partition, one at a time. The second phase takes each member $\mathcal{P}_f$ separately and extracts information necessary for computing ACS_k. We show that using $O(n)$ working space, members can be generated one after the other in overall $O(n \log^k n)$ time. Similarly, using $O(n)$ working space, members can be processed one after the other in overall $O(n \log^k n)$ time. Therefore, by interleaving phase 1 and phase 2 (i.e., as soon

as a member is generated, process it and then discard it from the working space), we achieve the space–time complexities in Theorem 1. We now present the phases in detail.

4. PHASE 1: CONSTRUCTING THE PARTITION

Our method for constructing the partition borrows from ideas in Cole et al. (2004) on string indexing for approximate pattern-matching queries, which itself relies on the classic heavy path decomposition strategy invented by Sleator and Tarjan (1981). The procedure is recursive, which can be best explained as constructing a tree (TREE) as follows:

Definition 4 TREE *is of depth k and each node x in* TREE *is associated with a set $\mathcal{S}_x$ of modified suffixes. The set associated with root consists of all n suffixes of* X *and* Y. *Then, starting with x = root, the sets associated with the children of any internal node x are generated from $\mathcal{S}_x$ using a recursive procedure maintaining the following properties:*

1. *The size of the set associated with any child node of x is $O(|\mathcal{S}_x|)$.*
2. *The sum of sizes of sets associated with all children of x is $O(|\mathcal{S}_x| \log |\mathcal{S}_x|)$.*
3. *If there exists an $(i, j)_{h-1}$-*maxpair *in $\boldsymbol{\mathcal{S}}_x$, where x is at depth $(h-1)$, then the set associated with one of the children of x must contain an $(i, j)_h$-*maxpair.

By unfolding the recursion k times, we have the following bounds for any **constant** k.

$$\max_{x \in \text{ NODES}} |\boldsymbol{\mathcal{S}}_x| = O(n) \text{ and } \sum_{x \in \text{NODES}} |\boldsymbol{\mathcal{S}}_x| = O(n \log^k n)$$

Here NODES is the set of nodes in TREE. Additionally, the collection of sets associated with the leaves of TREE form the partition in Lemma 3. We now present details of the recursion.

4.1. Details of recursion

Let $\boldsymbol{\mathcal{S}}_x$ be the set associated with a node x at depth $(h-1)$ in TREE. Then, we create its children and the sets associated with them as follows: construct a patricia trie $\mathcal{T}$ over the strings in $\boldsymbol{\mathcal{S}}_x$ (Gusfield, 1997), which consists of $|\boldsymbol{\mathcal{S}}_x|$ leaves and at most $(|\boldsymbol{\mathcal{S}}_x| - 1)$ internal nodes such that the degree of any internal node is at least two. Corresponding to each string in $\boldsymbol{\mathcal{S}}_x$, there will be a unique leaf node in $\mathcal{T}$ and vice versa.* We then classify the nodes in $\mathcal{T}$ as either **light** or **heavy** using the following rule: the root is always light and exactly one child of every internal node is heavy. The heavy child of any internal node is the one with the maximum number of leaves in its subtree (break ties arbitrarily). Essentially we are doing a decomposition of the tree based on heavy and light nodes (a.k.a heavy path decomposition). For any node w in $\mathcal{T}$, let $\mathsf{str}(w)$ be the set of strings in $\boldsymbol{\mathcal{S}}_x$ corresponding to all the leaves in the subtree of w. The following result is by Sleator and Tarjan (1981).

Lemma 4 *The number of light ancestors of any node in $\mathcal{T}$ is $\leq \log |\boldsymbol{\mathcal{S}}_x|$.*

Proof Note that for any light node w, with w' being its heavy sibling and v being their parent, $|\mathsf{str}(v)| \geq |\mathsf{str}(w)| + |\mathsf{str}(w')| \geq 2 \times |\mathsf{str}(w)|$. If the number of light ancestors exceeds $\log |\boldsymbol{\mathcal{S}}_x|$, then $|\mathsf{str}(root)| > |\boldsymbol{\mathcal{S}}_x|$, which is a contradiction.

A path, starting from w to a unique leaf in its subtree, where all nodes on this path are heavy is called the heavy path of w. We use $\mathsf{heavypath}(w)$ to denote the string corresponding to that unique leaf node. Corresponding to each **light internal node** w in $\mathcal{T}$, we create a child for x (say x') in TREE and the set $\mathcal{S}_{x'}$ is obtained as follows (see Fig. 1 for an illustration):

1. For each string $\alpha \in \mathsf{str}(w)$ except $\mathsf{heavypath}(w)$, find $p = 1 + |\mathsf{LCP}(\alpha, \mathsf{heavypath}(w))|$, that is, the position where the first mismatch occurs while matching the prefixes of α and $\mathsf{heavypath}(w)$.

*Specifically, the mth leftmost leaf node corresponds to the mth smallest string in lexicographic order.

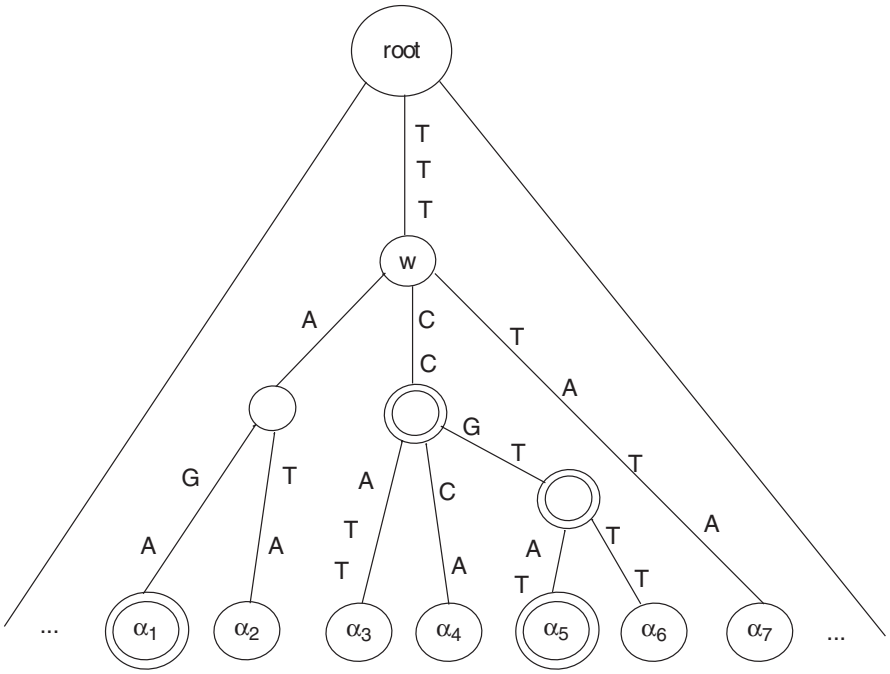

FIG. 1. The heavy nodes are drawn as double circles. Here $\alpha_1 = TTTAGA$, $\alpha_2 = TTTATA$, $\alpha_3 = TTTCCATT$, etc. $\mathsf{str}(w) = \{\alpha_1, \ldots, \alpha_7\}$, and $\mathsf{heavypath}(w) = \alpha_5 = TTTCCGTAT$. Therefore, set corresponding to w is $\mathsf{str}(w) \cup \{\alpha'_1, \alpha'_2, \alpha'_3, \alpha'_4, \alpha'_6, \alpha'_7\}$, where $\alpha'_1 = TTTCGA$, $\alpha'_2 = TTTCTA$, $\alpha'_3 = TTTCCGTT$, $\alpha'_4 = TTTCCGA$, $\alpha'_6 = TTTCCGTAT$, and $\alpha'_7 = TTTCATA$.

2. Obtain a new string α' from α by changing the character of α at position p to the character of $\mathsf{heavypath}(w)$ at the same position p. Specifically, $\alpha' = \alpha^{\{(p, \mathsf{heavypath}(w)[p])\}}$. Therefore, $|\mathsf{LCP1}(\alpha, \mathsf{heavypath}(w))| = |\mathsf{LCP}(\alpha', \mathsf{heavypath}(w))|$.
3. Then $\mathcal{S}_{x'}$ is the union of $\mathsf{str}(w)$ and the set of $(|\mathsf{str}(w)|-1)$ newly created strings with respect to $\mathsf{heavypath}(w)$.

Clearly, $|\mathcal{S}_{x'}| = (2|\mathsf{str}(w)| - 1) = O(|\mathcal{S}_x|)$. The number of sets (associated with the children of x) in which a particular string in $\mathcal{S}_x$ or a modified version of it belongs to is bounded by the maximum number of light ancestors of any node in $\mathcal{T}$, which is $\log |\mathcal{S}_x|$. This gives an upper bound $2|\mathcal{S}_x| \log |\mathcal{S}_x|$ on the sum of sizes of the sets associated with the children of x. Finally, let X_i^{Δ} and $\mathsf{Y}_j^{\Delta'}$ be two modified suffixes in $\mathcal{S}_x$ that form an $(i, j)_{h-1}$-maxpair, and let u (resp., w) be the lowest common ancestor (resp., light ancestor) of the leaves corresponding to them in $\mathcal{T}$. Then, our method creates a set corresponding to w, such that it contains two modified suffixes $\mathsf{X}_i^{\Delta''}$ and $\mathsf{Y}_j^{\Delta'''}$ that form an $(i, j)_h$-maxpair. Specifically, Δ'' and Δ''' are as follows (see Fig. 2 for an illustration):

(a) If the heavy child of u, say u^*, is an ancestor of the leaf corresponding to X_i^{Δ}, then $\Delta'' = \Delta$ and $\Delta''' = \Delta' \cup \{(p, \mathsf{X}_i[p])\}$, where $p = 1$ plus string depth of u.[†]
(b) If u^* is an ancestor of the leaf corresponding to $\mathsf{Y}_j^{\Delta'}$, then $\Delta'' = \Delta \cup \{(p, \mathsf{Y}_j[p])\}$ and $\Delta''' = \Delta'$.
(c) If neither of the above is true, then we have $\Delta'' = \Delta \cup \{(p, \mathsf{heavypath}(w)[p])\}$ and $\Delta''' = \Delta' \cup \{(p, \mathsf{heavypath}(w)[p])\}$.

4.1.1. Time and space complexity. The above recursive step can be executed as follows: First, construct $\mathcal{T}$ using the following steps.

1. Lexicographically sort all modified suffixes in $O(|\mathcal{S}_x|)$. We shall use the result in Lemma 2 and perform this step in $O(|\mathcal{S}_x| \log |\mathcal{S}_x|)$ time.

[†]Notice that $p = 1 + |\mathsf{LCP}(\mathsf{X}_i^{\Delta}, \mathsf{Y}_j^{\Delta'})| = 1 + |\mathsf{LCP}_{h-1}(\mathsf{X}_i, \mathsf{Y}_j)|$.

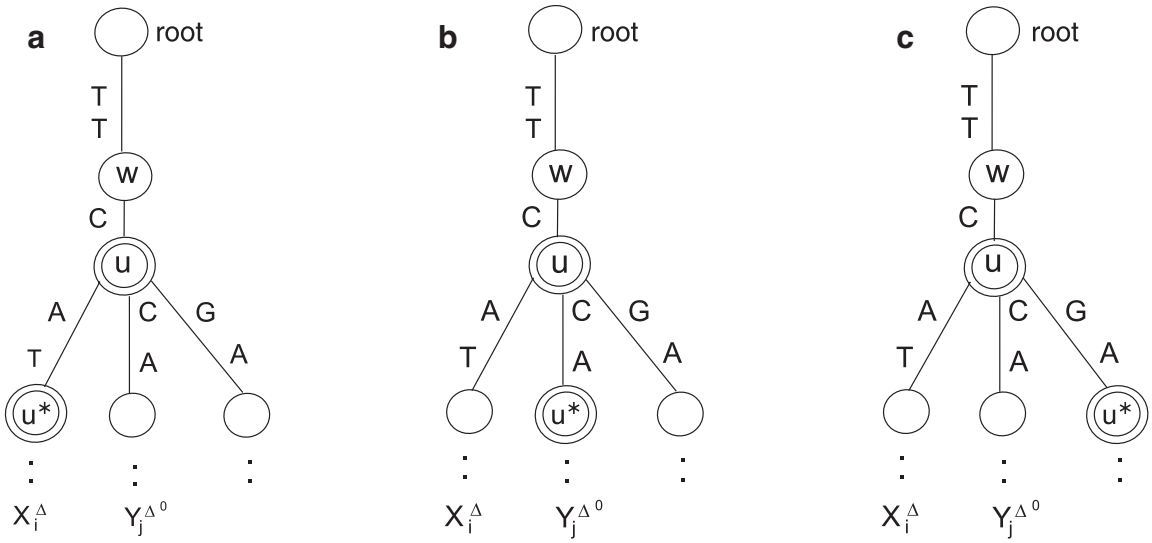

FIG. 2. Heavy nodes are drawn as double circles. In this example, u is heavy, the string depth of u is 3, and $p=4$. Then, $\Delta''' = \Delta' \cup \{[4, A]\}$ in case(a), $\Delta'' = \Delta \cup \{[4, C]\}$ in case(b), and $\Delta'' = \Delta \cup \{[4, G]\}$, $\Delta''' = \Delta' \cup \{[4, G]\}$ in case(c).

2. Compute $|\mathsf{LCP}(\cdot,\cdot)|$ between all consecutive pairs of modified suffixes in $\mathcal{S}_x$. Using GST, we can execute this step in $O(|\mathcal{S}_x|)$ time using Lemma 1.
3. Finally, to construct $\mathcal{T}$ using the results from Step (1) and Step (2), we borrow well-known techniques from the suffix tree construction literature [Farach-Colton et al., 2000): a patricia tree over a set of strings can be constructed in time linear in the number of strings, given the lexicographical ordering of all strings and length of the longest common prefix of all pairs of consecutive leaves of the tree. Therefore, time complexity is $O(|\mathcal{S}_x|)$.

Once $\mathcal{T}$ is constructed, all light internal nodes can be identified by a linear time tree traversal. Each light node w is then visited and the set corresponding to it is constructed in $O(|\mathsf{str}(w)|)$ time. The total time is proportional to the $\mathcal{S}_x$ plus the sum of sizes of all newly created sets, that is, $O(|\mathcal{S}_x| \log |\mathcal{S}_x|)$. Therefore, time for constructing TREE is $\sum_{x \in \text{NODES}} |\mathcal{S}_x| = O(n \log^k n)$.

Since we are not interested in storing the members of the partition, but just create, process, and discard them, the nodes in TREE can be created in the order of depth first search, so that at any point in time, we need to maintain nodes (and associated sets) in only one root to leaf path. Therefore, $O(k \times \max_{x \in \text{NODES}} |\mathcal{S}_x|) = O(n)$ working space is sufficient. This completes phase 1 of our algorithm.

5. PHASE 2: PROCESSING THE PARTITION

In this section, we describe how to process each member $\mathcal{P}_f$ of the partition independently. Let $\Lambda[1, |\mathsf{X}|]$ be an integer array of length $|\mathsf{X}|$ with all its entries initialized to 0. While processing the members, we update this array in such a way that after processing all members in the partition, we obtain $\lambda_k(\mathsf{X}_i, \mathsf{Y}) = \Lambda[i]$. To process a particular $\mathcal{P}_f$, we follow the steps below:

1. Create all nonempty subsets (called pi-sets) derived from $\mathcal{P}_f$ as defined below:

$$\Pi_{(\delta, x, y)} = \{\mathsf{X}_i^{\Delta} \in \mathcal{P}_f | \delta \subseteq \Delta \text{ and } |\Delta| = x\} \cup \{\mathsf{Y}_j^{\Delta'} \in \mathcal{P}_f | \delta \subseteq \Delta' \text{ and } |\Delta'| = y\}$$

   Here δ is a set of (*position, character*) pairs and x, y are integers in $[0, k]$.
2. Then process each $\Pi_{(\delta, x, y)}$ as follows: For each $\mathsf{X}_i^{\Delta} \in \Pi_{(\delta, x, y)}$, find the $\mathsf{Y}_j^{\Delta'} \in \Pi_{(\delta, x, y)}$, that maximizes the *length of their longest common prefix*. Then update

$$\Lambda[i] \leftarrow \max\{\Lambda[i], |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_j)|\}$$

In other words, first sort all modified suffixes in $\Pi_{(\delta, x, y)}$. Then for each $\mathsf{X}_i^{\Delta} \in \Pi_{(\delta, x, y)}$, find the closest modified suffixes of Y (say, $\mathsf{Y}_{j_l}^{\Delta_l}$ and $\mathsf{Y}_{j_r}^{\Delta_r}$) toward, the left as well as the right side of X_i^{Δ}, then update $\Lambda[i] \leftarrow \max\{\Lambda[i], |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_{j_l})|, |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_{j_r})|\}$.

Correctness: To prove that $\lambda_k(\mathsf{X}_i, \mathsf{Y}) = \Lambda[i]$ after processing all members, consider a fixed i, where $\lambda_k(\mathsf{X}_i, \mathsf{Y}) = |\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_{j^*})|$. While processing the member that contains an (i, j^*) $_k$-maxpair (say $\mathsf{X}_i^{\Delta_1}$ and $\mathsf{Y}_{j^*}^{\Delta_2}$), a pi-set $\Pi_{(\Delta_1 \cap \Delta_2, |\Delta_1|, |\Delta_2|)}$ will be created, and while processing this particular pi-set, $\Lambda[i]$ will be updated to $|\mathsf{LCP}_k(\mathsf{X}_i, \mathsf{Y}_{j^*})|$.

Analysis: Step 1 can be implemented as follows.

1. For each $\mathsf{X}_i^{\Delta} \in \mathcal{P}_f$, generate a (set, *string*) pair of the form $((\delta, \mathsf{X}_i^{\Delta}))$ for each $\delta \subseteq \Delta$. Similarly, for each $\mathsf{Y}_j^{\Delta'} \in \mathcal{P}_f$, generate pair $((\delta, \mathsf{Y}_j^{\Delta'}))$ for each $\delta \subseteq \Delta'$. Therefore, the number of pairs is $\leq 2^k |\mathcal{P}_f|$.
2. Partition pairs into buckets, such that all pairs of the form $(\delta, \cdot)$ are in the same bucket, say B_δ. This step can be reduced to sorting by first defining a total ordering among sets. For example, pairs can be first sorted based on the size of the set (smaller set first). Then, among the sets of the form $\{(l_1, \sigma_1), (l_2, \sigma_2) \ldots, (l_m, \sigma_m)\}$, $l_1 < l_2 < \cdots < l_m$, where m is fixed, first sort them with respect to l_1, break ties with respect to σ_1, break further ties with respect to l_2, and so on. This way, pairs within the same bucket form a contiguous region in the sorted list, hence can be easily separated.
3. Now creating pi-sets $\Pi_{(\delta, x, y)}$ from B_δ for all $0 \leq x, y \leq k$ is straightforward. Note that the sum of sizes of all pi-sets created is $\leq (k+1)2^k |\mathcal{P}_f| = O(|\mathcal{P}_f|)$

Essentially, step 1 involves sorting and scanning several sets of integers of total cardinality $O(|\mathcal{P}_f|)$. The key operations involved in step 2 are also the same. Therefore, a particular $\mathcal{P}_f$ can be processed in $O(|\mathcal{P}_f| \log |\mathcal{P}_f|)$ time using $O(n + |\mathcal{P}_f|)$ working space. Therefore, we can implement phase 2 in $O(n + \max_f |\mathcal{P}_f|) = O(n)$ space and $O(n \log^k n + \sum_f |\mathcal{P}_f| \log |\mathcal{P}_f|) = O(n \log^{k+1} n)$ time.

5.1. Improving the run-time complexity

In this section, we show how to improve the run time of phase 2 (thus overall time) to $O(n \log^k n)$. Following is a key result (proof will be detailed later).

Lemma 5 *The sorting of strings within a member $\mathcal{P}_f$ can be reduced to the sorting and scanning of several sets of integers in $[0, O(n)]$ of total cardinality $O(|\mathcal{P}_f|)$.*

Notice that the processing of $\mathcal{P}_f$ essentially consists of sorting and scanning of several sets of integers or strings of total cardinality $O(|\mathcal{P}_f|)$. However, by applying Lemma 5 once, we can associate each string in $\mathcal{P}_f$ with its lexicographic rank in $\mathcal{P}_f$, and all time-consuming steps in the processing of $\mathcal{P}_f$ can be reduced to integer sorting as follows.

Lemma 6 *Processing of $\mathcal{P}_f$ can be reduced to the sorting and scanning of several sets of integers in $[0, O(n)]$ of total cardinality $O(|\mathcal{P}_f|)$.*

Using the above lemma, we have the following result: processing of several $\mathcal{P}_f$'s of total cardinality $O(n)$ can be reduced to sorting and scanning of several sets of integers in $[0, O(n)]$ of total cardinality $O(n)$, and solved in $O(n)$ space and time. Using this result as a black box, we can process all members in the partition in $O(n)$ space and $O(n \log^k n)$ time.[§] This completes phase 2 of our algorithm.

In summary, both Phase 1 and Phase 2 can be executed in $O(n \log^k n)$ time using $O(n)$ working space. This completes the proof of Theorem 1.

5.1.1. Proof of Lemma 5. From section 4.1, recall that each node x in TREE is associated with a set $\mathcal{S}_x$, and it corresponds to a member in the partition if x is a leaf node. Therefore, it suffices to prove the result for an arbitrary $\mathcal{S}_x$. Let $\mathcal{T}$ be the trie over the strings in $\mathcal{S}_x$ and *root* be its root node, then

[§] Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_r$ be the sets, then each $e \in \mathcal{C}_i$ can be mapped to a pair (*key, value*) = (i, e). Then, sort all pairs using count sort with respect to *value* followed by a stable sort with respect to *key*. By scanning the result once, we obtain the desired output. Space and time complexity is $O(n)$.

1. For any string $\alpha \in \mathcal{S}_x$, the string obtained by deleting the $|\mathsf{LCP}(\alpha, \mathsf{heavypath}(root))|$-long prefix of α is a proper suffix (i.e., suffix without any modification) of X or Y. The result can be proven using mathematical induction: the base case (i.e., x is *root*) is trivially true. For any other node x, the result easily follows from the construction procedure detailed in section 4.1, provided that the result holds true for the set associated with the parent of x.
2. For two strings α, β, we denote $\alpha < \beta$ if α comes before β in lexicographic order. Then, the set $\mathcal{S}_x$ can be partitioned into sets $BEFORE(d)$'s and $AFTER(d)$'s for several values of d, where

$$BEFORE(d) = \{\alpha \in \mathcal{S}_x || \mathsf{LCP}(\alpha, \mathsf{heavypath}(root))| = d, \alpha < \mathsf{heavypath}(root)\}$$
$$AFTER(d) = \{\alpha \in \mathcal{S}_x || \mathsf{LCP}(\alpha, \mathsf{heavypath}(root))| = d, \alpha > \mathsf{heavypath}(root)\}$$

   Each proper suffix can be uniquely mapped to its lexicographic rank among all suffixes of X and Y. Therefore, a collection of proper suffixes can be sorted via integer sorting. Additionally, all strings within a particular $BEFORE(d)$ [resp., $AFTER(d)$] share the same d-long prefix, and its removal will result in a set of proper suffixes, which can be sorted via integer sorting. In summary, we can categorize the strings in $\mathcal{P}_f$ into sets $BEFORE(\cdot)$'s and $AFTER(\cdot)$'s with the strings within it sorted.
3. Finally, observe the following:
   (a) If $\alpha \in BEFORE(\cdot)$ and $\beta \in AFTER(\cdot)$, then $\alpha < \beta$,
   (b) If $\alpha \in BEFORE(d)$ and $\beta \in BEFORE(d')$, then $\alpha < \beta$ iff $d < d'$, and
   (c) If $\alpha \in AFTER(d)$ and $\beta \in AFTER(d')$, then $\alpha < \beta$ iff $d > d'$.

In light of the above three observations, we can obtain a sorted $\mathcal{P}_f$ by concatenating sorted $BEFORE(d)$'s in the ascending order of d's followed by sorted $AFTER(d)$'s in the descending order of d's.

This completes the proof of Lemma 5.

6. CONCLUSIONS

We present an efficient algorithm for the k-mismatch average common substring problem that runs in $O(n \log^k n)$ time and $O(n)$ space. This constitutes a significant improvement over the $O(n^2)$ worst case runtime of the prior state of the art. In the absence of the proposed space-saving technique, our algorithm would have taken $O(n \log^k n)$ space, which would be impractical for even small values of k. Our algorithm achieves linear working space by decomposing the set of modified suffixes so that they can be processed and discarded independently, while preserving the ability to bring pairs of modified suffixes together. Note that the biological applications of ACS_k are well documented in Leimeister and Morgenstern (2014). The vastly improved efficiency in computing ACS_k by our algorithm should naturally impact these applications positively.

ACKNOWLEDGMENT

This research is supported in part by the U.S. National Science Foundation under IIS-1416259.

AUTHOR DISCLOSURE STATEMENT

No competing financial interests exist.

REFERENCES

Aluru, S., Apostolico, A., and Thankachan, S.V. 2015. Efficient alignment free sequence comparison with bounded mismatches, 1–12. Proceedings of Research in Computational Molecular Biology—19th Annual International Conference, RECOMB 2015, Warsaw, Poland, April 12–15, 2015.

Apostolico A. 2010. Maximal words in sequence comparisons based on subword composition, 34–44. In *Algorithms and Applications*. Springer, New York.

Apostolico, A., and Cunial, F. 2011. Sequence similarity by gapped lzw, 343–352. Data Compression Conference (DCC), 2011. IEEE, New York.

Apostolico, A., and Denas, O. 2008. Fast algorithms for computing sequence distances by exhaustive substring composition. *Algorithms Mol. Biol.* 3, 1756.

Apostolico, A., and Giancarlo, R. 1998. Sequence alignment in molecular biology. *J. Comput. Biol.* 5, 173–196.

Apostolico, A., Denas, O., and Dress, A. 2010. Efficient tools for comparative substring analysis. *J. Biotechnol.* 149, 120–126.

Asai, T., Zaporojets, D., Squires, C., and Squires, C.L. 1999. An *Escherichia coli* strain with all chromosomal rRNA operons inactivated: Complete exchange of rRNA genes between bacteria. *Proc. Natl. Acad. Sci.* 96, 1971–1976.

Babenko, M.A., and Starikovskaya, T.A. 2008. Computing longest common substrings via suffix arrays, 64–75. In *Computer Science—Theory and Applications*. Springer, New York.

Blaisdell, B.E. 1986. A measure of the similarity of sets of sequences not requiring sequence alignment. *Proc. Natl. Acad. Sci.* 83, 5155–5159.

Bonham-Carter, O., Steele, J., and Bastola, D. 2013. Alignment-free genetic sequence comparisons: A review of recent approaches by word analysis. *Brief. Bioinform.* 15, 890–905.

Burstein, D., Ulitsky, I., Tuller, T., and Chor, B. 2005. Information theoretic approaches to whole genome phylogenies, 283–295. Research in Computational Molecular Biology, 9th Annual International Conference, RECOMB 2005.

Chang, G., and Wang, T. 2011. Phylogenetic analysis of protein sequences based on distribution of length about common substring. *Protein J.* 30, 167–172.

Clift, B., Haussler, D., and McConnell, R., et al. 1986. Sequence landscapes. *Nucleic Acids Res.* 14, 141–158.

Cole, R., Gottlieb, L.-A., and Lewenstein, M. 2004. Dictionary matching and indexing with errors and don't cares, 91–100. STOC '04: Proceedings of the Annual ACM Symposium on the Theory of Computing.

Comin, M., Verzotto, D., et al. 2012. Alignment-free phylogeny of whole genomes using underlying subwords. *Algorithms Mol. Biol.* 7, 34.

Cunial, F., and Apostolico, A. 2012. Phylogeny construction with rigid gapped motifs. *J. Comput. Biol.* 19, 911–927.

Domazet-Loso, M., and Haubold, B. 2009. Efficient estimation of pairwise distances between genomes. *Bioinformatics* 25, 3221–3227.

Edgar, R.C. 2004. Local homology recognition and distance measures in linear time using compressed amino acid alphabets. *Nucleic Acids Res.* 32, 380–385.

Farach-Colton, M., Ferragina, P., and Muthukrishnan, S. 2000. On the sorting-complexity of suffix tree construction. *J. ACM* 47, 987–1011.

Ferragina, P., Giancarlo, R., Greco, V., et al. 2007. Compression-based classification of biological sequences and structures via the universal similarity metric: Experimental assessment. *BMC Bioinform.* 8, 252.

Fitch, W.M., Margoliash, E., et al. 1967. Construction of phylogenetic trees. *Science* 155, 279–284.

Flouri, T., Giaquinta, E., Kobert, K., and Ukkonen, E. 2015. Longest common substrings with k mismatches. *Inf. Process. Lett.* 115, 643–647.

Fracasso, G. 1972. Genome composition with mismatches and its application to phylogeny. 2014.

Gatlin, L.L. 1972. Information theory and the living system.

Grabowski, S. 2015. A note on the longest common substring with k-mismatches problem. *Inf. Process. Lett.* 115, 640–642.

Gusfield, D. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, UK.

Guyon, F., Brochier-Armanet, C., and Guénoche, A. 2009. Comparison of alignment free string distances for complete genome phylogeny. *Adv. Data Anal. Classif.* 3, 95–108.

Hao, B., and Qi, J. 2004. Prokaryote phylogeny without sequence alignment: From avoidance signature to composition distance. *J. Bioinform. Comput. Biol.* 2, 1–19.

Höhl, M., and Ragan, M.A. 2007. Is multiple-sequence alignment required for accurate inference of phylogeny? *Syst. Biol.* 56, 206–221.

Höhl, M., Rigoutsos, I., and Ragan, M.A. 2006. Pattern-based phylogenetic distance estimation and tree reconstruction. *Evol. Bioinform. Online* 2, 359.

Horwege, S., Lindner, S., Boden, M., et al. 2014. Spaced words and kmacs: Fast alignmentfree sequence comparison based on inexact word matches. *Nucleic Acids Res.* 42, W7–W11.

Leimeister, C.-A., and Morgenstern, B. 2014. kmacs: The k-mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics* 30, 2000–2008.

Li, M., Chen, X., Li, X., et al. 2004. The similarity metric. *IEEE Trans. Inform. Theory.* 50, 3250–3264.

McCreight, E.M. 1976. A space-economical suffix tree construction algorithm. *J. ACM* 23, 262–272.

Morgenstern, B., Zhu, B., Horwege, S., and Leimeister, C.-A. 2014. Estimating evolutionary distances from spaced-word matches, 161–173. Algorithms in Bioinformatics—14th International Workshop, WABI.

Otu, H.H., and Sayood, K. 2003. A new sequence distance measure for phylogenetic tree construction. *Bioinformatics* 19, 2122–2130.

Pizzi, C. 2015. A Filtering approach for alignment-free biosequences comparison with mismatches, 231–242. Proceedings of Algorithms in Bioinformatics—15th International Workshop, WABI 2015, Atlanta, GA, September 10–12, 2015.

Qi, J., Wang, B., and Hao, B.-I. 2004. Whole proteome prokaryote phylogeny without sequence alignment: A k-string composition approach. *J. Mol. Evol.* 58, 1–11.

Sleator, D.D., and Tarjan, R.E. 1981. A data structure for dynamic trees, 114–122. STOC: Proceedings of the Annual ACM Symposium on the Theory of Computing.

Thankachan, S.V., Chockalingam, S., Liu, Y., et al. 2015. A greedy alignment-free distance estimator for phylogenetic inference. 5th IEEE International Conference on Computational Advances in Bio and Medical Sciences, 2015.

Ulitsky, I., Burstein, D., Tuller, T., and Chor, B. 2006. The average common substring approach to phylogenomic reconstruction. *J. Comput. Biol.* 13, 336–350.

Van Helden, J. 2004. Metrics for comparing regulatory sequences on the basis of pattern counts. *Bioinformatics* 20, 399–406.

Vinga, S., and Almeida, J. 2003. Alignment-free sequence comparison—A review. *Bioinformatics* 19, 513–523.

Weiner, P. 1973. Linear pattern matching algorithms, 1–11. 14th Annual IEEE Symposium on Switching and Automata Theory.

Wu, T.-J., Burke, J.P., and Davison, D.B. 1997. A measure of DNA sequence dissimilarity based on mahalanobis distance between frequencies of words. *Biometrics* 53, 1431–1439.

Zuckerkandl, E., and Pauling, L. 1965. Molecules as documents of evolutionary history. *J. Theor. Biol.* 8, 357–366.

Address correspondence to:
Dr. Srinivas Aluru
College of Computing
Georgia Institute of Technology
1336 Klaus Advanced Computing Building
266 Ferst Drive
Atlanta, GA 30332

E-mail: aluru@cc.gatech.edu