A Framework for Space-Efficient String Kernels

Djamal Belazzougui[1,2] and Fabio Cunial[1,2](✉)

[1] Department of Computer Science, University of Helsinki, Helsinki, Finland
[2] Helsinki Institute for Information Technology, Helsinki, Finland
fabio.cunial@cs.helsinki.fi

Abstract. String kernels are typically used to compare genome-scale sequences whose length makes alignment impractical, yet their computation is based on data structures that are either space-inefficient, or incur large slowdowns. We show that a number of exact string kernels, like the k-mer kernel, the substrings kernels, a number of length-weighted kernels, the minimal absent words kernel, and kernels with Markovian corrections, can all be computed in $O(nd)$ time and in $o(n)$ bits of space in addition to the input, using just a `rangeDistinct` data structure on the Burrows-Wheeler transform of the input strings that takes $O(d)$ time per element in its output. The same bounds hold for a number of measures of compositional complexity based on multiple values of k, like the k-mer profile and the k-th order empirical entropy, and for calibrating the value of k using the data.

1 Introduction

Given two strings T^1 and T^2, a *kernel* is a function that simultaneously converts T^1 and T^2 into vectors $\mathbf{T}^1$ and $\mathbf{T}^2$ in $\mathbb{R}^n$ for some $n > 0$, and computes a similarity or a distance measure between $\mathbf{T}^1$ and $\mathbf{T}^2$, *without building and storing* $\mathbf{T}^i$ *explicitly* [14]. Kernels are often the method of choice for comparing extremely long strings, like genomes, read sets, and metagenomic samples, whose size makes alignment infeasible, yet their computation is typically based on space-inefficient data structures, like (truncated) suffix trees, or on space-efficient data structures with $O(\log^\epsilon n)$ slowdowns, like compressed suffix trees (see e.g. [1,9] and references therein). The (possibly infinite) dimensions of $\mathbf{T}^i$ are, for example, all strings of a specific family on the alphabet of T^1 and T^2, and the value assigned to vector $\mathbf{T}^i$ along dimension W corresponds to the number of occurrences of string W in T^i, often rescaled and corrected in domain-specific ways. $\mathbf{T}^i$ is often called *composition vector*, and a large number of its components can be zero in practice. In this paper we focus on space- and time-efficient algorithms for computing the *cosine of the angle between two composition vectors* $\mathbf{T}^1$ and $\mathbf{T}^2$, i.e. on computing the kernel $\kappa(\mathbf{T}^1, \mathbf{T}^2) = N/\sqrt{D^1 D^2} \in [-1..1]$, where $N = \sum_W \mathbf{T}^1[W]\mathbf{T}^2[W]$ and $D^i = \sum_W \mathbf{T}^i[W]^2$. This measure of similarity can be converted into a distance $d(\mathbf{T}^1, \mathbf{T}^2) = (1 - \kappa(\mathbf{T}^1, \mathbf{T}^2))/2 \in [0..1]$, and the

This work was partially supported by Academy of Finland under grant 284598 (Center of Excellence in Cancer Genetics Research).

F. Cicalese et al. (Eds.): CPM 2015, LNCS 9133, pp. 13–25, 2015.
DOI: 10.1007/978-3-319-19929-0_2

algorithms we describe can be applied to compute norms of vector $\mathbf{T}^1 - \mathbf{T}^2$, like the p-norm and the infinity norm. When $\mathbf{T}^1$ and $\mathbf{T}^2$ are bitvectors, we are more interested in interpreting them as sets and in computing the Jaccard distance $J(\mathbf{T}^1, \mathbf{T}^2) = ||\mathbf{T}^1 \wedge \mathbf{T}^2||/||\mathbf{T}^1 \vee \mathbf{T}^2|| = ||\mathbf{T}^1 \wedge \mathbf{T}^2||/(||\mathbf{T}^1|| + ||\mathbf{T}^2|| - ||\mathbf{T}^1 \wedge \mathbf{T}^2||)$, where $\wedge$ and $\vee$ are the bitwise AND and OR operators, and where $||\cdot||$ measures the number of ones in a bitvector.

Given a data structure that supports `rangeDistinct` queries on the Burrows-Wheeler transform of each string in input, we show that a number of popular string kernels, like the k-mer kernel, the substrings kernels, a number of length-weighted kernels, the minimal absent words kernel, and kernels with Markovian corrections, can all be computed in $O(nd)$ time and in $o(n)$ bits of space in addition to the input, *all in a single pass over the BWTs of the input strings*, where d is the time taken by the `rangeDistinct` query per element in its output. The same bounds hold for computing a number of measures of compositional complexity *for multiple values of k at the same time*, like the k-mer profile and the k-th order empirical entropy, and for choosing the value of k used in k-mer kernels from the data. All these algorithms become $O(n)$ using the `rangeDistinct` data structure described in [4], and concatenating this setup to the BWT construction algorithm described in [3], we can compute all such kernels and complexity measures *from the input strings* in randomized $O(n)$ time and in $O(n \log \sigma)$ bits of space in addition to the input. Finally, we show that measures of expectation based on Markov models are related to the left and right extensions of maximal repeats.

2 Preliminaries

2.1 Strings

Let $\Sigma = [1..\sigma]$ be an integer alphabet, let $\# = 0$, $\#_1 = -1$ and $\#_2 = -2$ be distinct separators not in Σ, and let $T = [1..\sigma]^{n-1}\#$ be a string. We assume $\sigma \in o(\sqrt{n}/\log n)$ throughout the paper. A *k-mer* is any string $W \in [1..\sigma]$ of length $k > 0$. We denote by $f_T(W)$ the number of (possibly overlapping) occurrences of a string W in the circular version of T, and we use the shorthand $p_T(W) = f_T(W)/(n - |W|)$ to denote an approximation of the *empirical probability* of observing W in T, assuming that all positions of T except the last $|W|$ ones are equally probable starting positions for W. A *repeat* W is a string that satisfies $f_T(W) > 1$. We denote by $\Sigma^\ell_T(W)$ the set of characters $\{a \in [0..\sigma] : f_T(aW) > 0\}$ and by $\Sigma^r_T(W)$ the set of characters $\{b \in [0..\sigma] : f_T(Wb) > 0\}$. A repeat W is *right-maximal* (respectively, *left-maximal*) iff $|\Sigma^r_T(W)| > 1$ (respectively, iff $|\Sigma^\ell_T(W)| > 1$). It is well known that T can have at most $n - 1$ right-maximal substrings and at most $n - 1$ left-maximal substrings. A *maximal repeat* of T is a repeat that is both left- and right-maximal.

For reasons of space we assume the reader to be familiar with the notion of *suffix tree* ST_T of a string T, and with the notion of *generalized suffix tree* of two strings, which we do not define here. We denote by $\ell(v)$ the string label of a node v in a suffix tree. It is well known that a substring W of T is right-maximal

iff $W = \ell(v)$ for some internal node v of ST_T. We assume the reader to be familiar with the notion of *suffix link* connecting a node v with $\ell(v) = aW$ for some $a \in [0..\sigma]$ to a node w with $\ell(w) = W$: we say that $w = \texttt{suffixLink}(v)$ in this case. Here we just recall that suffix links and internal nodes of ST_T form a tree, called the *suffix-link tree* of T and denoted by SLT_T, and that inverting the direction of all suffix links yields the so-called *explicit Weiner links*. Given an internal node v and a symbol $a \in [0..\sigma]$, it might happen that string $a\ell(v)$ does occur in T, but that it is not right-maximal, i.e. it is not the label of any internal node of ST_T: all such left extensions of internal nodes that end in the middle of an edge are called *implicit Weiner links*. An internal node v of ST_T can have more than one outgoing Weiner link, and all such Weiner links have distinct labels: in this case, $\ell(v)$ is a maximal repeat. It is known that the number of suffix links (or, equivalently, of explicit Weiner links) is upper-bounded by $2n-2$, and that the number of implicit Weiner links can be upper-bounded by $2n-2$ as well.

2.2 Enumerating Right-Maximal Substrings and Maximal Repeats

For reasons of space we assume the reader to be familiar with the notion and uses of the Burrows-Wheeler transform of T, including the C array, the `rank` function, and backward searching. In this paper we use BWT_T to denote the BWT of T, we use $\texttt{range}(W) = [\texttt{sp}(W)..\texttt{ep}(W)]$ to denote the lexicographic interval of a string W in a BWT that is implicit from the context, and we use $\Sigma_{i,j}$ to denote the set of distinct characters that occur inside interval $[i..j]$ of a string that is implicit from the context. We also denote by $\texttt{rangeDistinct}(i, j)$ the function that returns the set of tuples $\{(c, \texttt{rank}(c, p_c), \texttt{rank}(c, q_c)) : c \in \Sigma_{i,j}\}$, *in any order*, where p_c and q_c are the first and the last occurrence of c inside interval $[i..j]$, respectively. Here we focus on a specific application of BWT_T: enumerating all the right-maximal substrings of T, or equivalently all the internal nodes of ST_T. In particular, we use the algorithm described in [3] (Sect. 4.1), which we sketch here for completeness.

Given a substring W of T, let $b_1 < b_2 < \cdots < b_k$ be the sorted sequence of all the distinct characters in $\Sigma^r_T(W)$, and let $a_1, a_2, \ldots, a_h$ be the list of all the characters in $\Sigma^\ell_T(W)$, not necessarily sorted. Assume that we represent a substring W of T as a pair $\texttt{repr}(W) = (\texttt{chars}[1..k], \texttt{first}[1..k+1])$, where $\texttt{chars}[i] = b_i$, $\texttt{range}(Wb_i) = [\texttt{first}[i]..\texttt{first}[i+1]-1]$ for $i \in [1..k]$, and `range`() refers to BWT_T. Note that $\texttt{range}(W) = [\texttt{first}[1]..\texttt{first}[k+1]-1]$, since it coincides with the concatenation of the intervals of the right extensions of W in lexicographic order. If W is not right-maximal, array `chars` in $\texttt{repr}(W)$ has length one. Given a data structure that supports `rangeDistinct` queries on BWT_T, and given the C array of T, there is an algorithm that converts $\texttt{repr}(W)$ into the sequence $a_1, \ldots, a_h$ and into the corresponding sequence $\texttt{repr}(a_1W), \ldots, \texttt{repr}(a_hW)$, in $O(de)$ time and $O(\sigma^2 \log n)$ bits of space in addition to the input and the output [3], where d is the time taken by the `rangeDistinct` operation per element in its output, and e is the number of distinct strings a_iWb_j that occur in the circular

version of T, where $i \in [1..h]$ and $j \in [1..k]$. We encapsulate this algorithm into a function that we call `extendLeft`.

If a_iW is right-maximal, i.e. if array `chars` in $\texttt{repr}(a_iW)$ has length greater than one, we push pair $(\texttt{repr}(a_iW), |W|+1)$ onto a stack S. In the next iteration we pop the representation of a string from the stack and we repeat the process, until the stack itself becomes empty. This process is equivalent to following all the explicit Weiner links from the node v of ST_T with $\ell(v) = W$, not necessarily in lexicographic order. Thus, running the algorithm from a stack initialized with $\texttt{repr}(\varepsilon)$ is equivalent to performing a preorder depth-first traversal of the suffix-link tree of T (with children explored in arbitrary order), which guarantees to enumerate all the right-maximal substrings of T. Every operation performed by the algorithm can be charged to a distinct node or Weiner link of ST_T, thus the algorithm runs in $O(nd)$ time. The depth of the stack is $O(\log n)$ rather than $O(n)$, since at every iteration we push the pair $(\texttt{repr}(a_iW), |a_iW|)$ with largest $\texttt{range}(a_iW)$ first. Every suffix-link tree level in the stack contains at most σ pairs, and each pair takes at most $\sigma \log n$ bits of space, thus the total space used by the stack is $O(\sigma^2 \log^2 n)$ bits. The following theorem follows from our assumption that $\sigma \in o(\sqrt{n}/\log n)$:

Theorem 1 ([3]). *Let $T \in [1..\sigma]^{n-1}\#$ be a string. Given a data structure that supports* `rangeDistinct` *queries on BWT_T, we can enumerate all the right-maximal substrings W of T, and for each of them we can return $|W|$, $\texttt{repr}(W)$, the sequence $a_1, a_2, \ldots, a_h$ of all characters in $\Sigma^\ell_T(W)$ (not necessarily sorted), and the sequence $\texttt{repr}(a_1W), \ldots, \texttt{repr}(a_hW)$, in $O(nd)$ time and in $o(n)$ bits of space in addition to the input and the output, where d is the time taken by the* `rangeDistinct` *operation per element in its output.*

Theorem 1 does not specify the order in which the right-maximal substrings must be enumerated, nor the order in which the left extensions of a right-maximal substring must be returned. The algorithm we just described can be adapted to return all the maximal repeats of T, with the same bounds, by outputting a right-maximal string W iff $|\texttt{rangeDistinct}(\texttt{sp}(W), \texttt{ep}(W))| > 1$. A version of the same algorithm can also enumerate all the internal nodes of the *generalized suffix tree* of two string T^1 and T^2, using BWT_{T^1} and BWT_{T^2}: in this case, a string W is represented as a quadruple $\texttt{repr}'(W) = (\texttt{chars}_1[1..k_1], \texttt{first}_1[1..k_1+1], \texttt{chars}_2[1..k_2], \texttt{first}_2[1..k_2+1])$, and we assume that $\texttt{first}_i[1] = 0$ iff W does not occur in T^i. We call $\texttt{extendLeft}'$ the function that maps $\texttt{repr}'(W)$ to the list of its left extensions $\texttt{repr}'(a_iW)$.

Theorem 2 ([3]). *Let $T^1 \in [1..\sigma]^{n_1-1}\#_1$ and $T^2 \in [1..\sigma]^{n_2-1}\#_2$ be two strings. Given two data structures that support* `rangeDistinct` *queries on BWT_{T^1} and on BWT_{T^2}, respectively, we can enumerate all the right-maximal substrings W of $T = T^1T^2$, and for each of them we can return $|W|$, $\texttt{repr}'(W)$, the sequence $a_1, a_2, \ldots, a_h$ of all characters in $\Sigma^\ell_{T^1T^2}(W)$ (not necessarily sorted), and the sequence $\texttt{repr}'(a_1W), \ldots, \texttt{repr}'(a_hW)$, in $O(nd)$ time and in $o(n)$ bits of space in addition to the input and the output, where $n = n_1 + n_2$ and d is the time taken by the* `rangeDistinct` *operation per element in its output.*

For reasons of space, we assume throughout the paper that d is the time per element in the output of a `rangeDistinct` data structure that is implicit from the context. We also replace T^i by i in subscripts, or we waive subscripts completely whenever they are clear from the context.

3 Kernels and Complexity Measures on k-mers

Given a string $T \in [1..\sigma]^{n-1}\#$ and a length $k > 0$, let vector $\mathbf{T}_k[1..\sigma^k]$ be such that $\mathbf{T}_k[W] = f_T(W)$ for every $W \in [1..\sigma]^k$. The *k-mer complexity* $\mathsf{C}_k(T)$ of string T is the number of nonzero components of $\mathbf{T}_k$. The *k-mer kernel* of two strings T^1 and T^2 is $\kappa(\mathbf{T}^1_k, \mathbf{T}^2_k)$. Recall that Theorems 1 and 2 enumerate all nodes of a suffix tree in no specific order. In this section we describe algorithms to compute $\mathsf{C}_k(T)$ and $\kappa(\mathbf{T}^1_k, \mathbf{T}^2_k)$ in a way that does not depend on the order in which the nodes of a suffix tree are enumerated: we can thus implement such algorithms on top of Theorems 1 and 2. The main idea behind our approach is a telescoping strategy that works by adding and subtracting terms in a sum, as described below:

Theorem 3. *Let $T \in [1..\sigma]^{n-1}\#$ be a string. Given an integer k and a data structure that supports* `rangeDistinct` *queries on BWT_T, we can compute $\mathsf{C}_k(T)$ in $O(nd)$ time and in $o(n)$ bits of space in addition to the input.*

Proof. A k-mer of T can either be the label of a node of ST_T, or it could end in the middle of an edge (u, v) of ST. In the latter case, we assume that the k-mer is represented by its locus v, which might be a leaf. Let $\mathsf{C}_k(T)$ be initialized to $n - k$, i.e. to the number of leaves that correspond to suffixes of T of length at least $k + 1$. We enumerate the internal nodes of ST using Theorem 1, and every time we enumerate a node v we proceed as follows: if $|\ell(v)| < k$ we leave $\mathsf{C}_k(T)$ unaltered, otherwise we increment $\mathsf{C}_k(T)$ by one and we decrement $\mathsf{C}_k(T)$ by the number of children of v in ST, which is the length of array `chars` in $\texttt{repr}(\ell(v))$. In this way, every internal node v of ST that is located at string depth at least k and that is not the locus of a k-mer is both added to $\mathsf{C}_k(T)$ (when the algorithm visits v) and subtracted from $\mathsf{C}_k(T)$ (when the algorithm visits $\texttt{parent}(v)$). Leaves at depth at least $k + 1$ that are not the locus of a k-mer are added by the initialization of $\mathsf{C}_k(T)$, and they are subtracted during the enumeration. Conversely, every locus v of a k-mer of T (including leaves) is just added to $\mathsf{C}_k(T)$, since $|\ell(\texttt{parent}(v))| < k$.

We can apply the same telescoping strategy to compute $\kappa(\mathbf{T}^1_k, \mathbf{T}^2_k)$:

Theorem 4. *Let $T^1 \in [1..\sigma]^{n_1-1}\#_1$ and $T^2 \in [1..\sigma]^{n_2-1}\#_2$ be strings. Given an integer k and two data structures that support* `rangeDistinct` *queries on BWT_{T^1} and on BWT_{T^2}, respectively, we can compute $\kappa(\mathbf{T}^1_k, \mathbf{T}^2_k)$ in $O(nd)$ time and in $o(n)$ bits of space in addition to the input, where $n = n_1 + n_2$.*

Proof. Recall that $\kappa(\mathbf{T}^1_k, \mathbf{T}^2_k) = N/\sqrt{D^1 D^2}$, where $N = \sum_W \mathbf{T}^1_k[W]\mathbf{T}^2_k[W]$, $D^i = \sum_W \mathbf{T}^i_k[W]^2$, and $W \in [1..\sigma]^k$. We initially set $N = 0$ and $D^i = n_i - k$, since these are the contributions of all the leaves at depth at least $k+1$ in the generalized suffix tree of T^1 and T^2. Then, we enumerate every internal node u of the generalized suffix tree, using Theorem 2: if $|\ell(u)| < k$ we keep all variables unchanged, otherwise we set N to $N + f_1(\ell(u)) \cdot f_2(\ell(u)) - \sum_v f_1(\ell(v)) \cdot f_2(\ell(v))$ and we set D^i to $D^i + f_i(\ell(u))^2 - \sum_v f_i(\ell(v))^2$, where v ranges over all children of u in the generalized suffix tree. Clearly $f_i(\ell(u)) = \texttt{first}_i[k_i + 1] - \texttt{first}_i[1]$ where k_i is the size of array $\texttt{chars}_i$ in $\texttt{repr}'(\ell(u))$, and $f_i(\ell(v)) = f_i(\ell(u)b_j) = \texttt{first}_i[j+1] - \texttt{first}_i[j]$ for some $j \in [1..k_i]$. In analogy to Theorem 3, the contribution of the loci of the distinct k-mers of T^1, of T^2, or of both, is added to the three temporary variables and never subtracted, while the contribution of every other node u at depth at least k in the generalized suffix tree is both added (when the algorithm visits u, or when N and D^i are initialized) and subtracted (when the algorithm visits $\texttt{parent}(u)$).

An even more specific notion of compositional complexity is $\mathsf{C}_{k,f}(T)$, the number of distinct k-mers that occur exactly f times in T. In the *k-mer profiling* problem [6,7] we are given a string T, an interval $[k_1..k_2]$ of lengths and an interval $[f_1..f_2]$ of frequencies, and we are asked to compute the matrix $\texttt{profile}[k_1..k_2, f_1..f_2]$ defined as follows: $\texttt{profile}[i,j] = \mathsf{C}_{i,j}(T)$ if $j < f_2$, and $\texttt{profile}[i,j] = \sum_{h \geq j} \mathsf{C}_{i,h}(T)$ if $j = f_2$. Note that the jth column of $\texttt{profile}$ can have nonzero cells only if f_j is the frequency of some internal node of ST_T. In practice $\texttt{profile}$ is often computed by running a k-mer extraction algorithm $k_2 - k_1 + 1$ times, and by scanning the output of all such runs (see e.g. [6] and references therein). The following lemma shows that we can compute $\texttt{profile}$ in just one pass over the BWT of the input string, and in linear time in the size of $\texttt{profile}$:

Theorem 5. *Let $T \in [1..\sigma]^{n-1}\#$ be a string. Given ranges $[k_1..k_2]$ and $[f_1..f_2]$, and given a data structure that supports* $\texttt{rangeDistinct}$ *queries on BWT_T, we can compute matrix $\texttt{profile}[k_1..k_2, f_1..f_2]$ in $O(nd + (k_2 - k_1)(f_2 - f_1))$ time and in $o(n)$ bits of space in addition to the input and the output.*

Proof. We use Theorem 1 again. Assume that, for every internal node u of ST_T with string depth at least k_1 and with frequency at least f_1, and for every $k \in [k_1..\min\{|\ell(u)|, k_2\}]$, we increment $\texttt{profile}[k, \min\{f(u), f_2\}]$ by one and we decrement $\texttt{profile}[k, \min\{f(v), f_2\}]$ by one for every child v of u in ST such that $f(v) \geq f_1$. This would take $O(n^2)$ total updates to $\texttt{profile}$. However, we can perform all of these updates in batch, as follows: for every node u of ST with $f(u) \geq f_1$ and with $|\ell(u)| \geq k_1$, we just increment $\texttt{profile}[\min\{|\ell(u)|, k_2\}, \min\{f(u), f_2\}]$ by one, and we just decrement $\texttt{profile}[\min\{|\ell(u)|, k_2\}, \min\{f(v), f_2\}]$ by one for every child v of u in ST such that $f(v) \geq f_1$. After having traversed all the internal nodes of ST, we scan $\texttt{profile}$ as follows: for every $j \in [f_1..f_2]$, we traverse all values of i in the decreasing order $k_2 - 1, \ldots, k_1$, and we set $\texttt{profile}[i,j] = \texttt{profile}[i,j] + \texttt{profile}[i+1, j]$. If $f_1 = 1$, at the end of this process the first column of $\texttt{profile}$ contains

negative numbers, since Theorem 1 does not enumerate the leaves of ST. Thus, before returning, we add to $\texttt{profile}[i, 1]$ the number of leaves with string depth at least $k_i + 1$, i.e. value $n - k_i$, for all $i \in [k_1..k_2]$.

A similar algorithm allows computing $\kappa(\mathbf{T}^1_k, \mathbf{T}^2_k)$ for all k in a user-specified range $[k_1..k_2]$ in $O(nd + k_2 - k_1)$ time. Matrix $\texttt{profile}$ can be used to determine a range of values of k to be used in k-mer kernels. The smallest number in this range is typically the value of k that maximizes the number of distinct k-mers that occur at least twice in T [15]. The largest number in the range is typically determined using some measure of expectation: we cover this computation in Sect. 5.

A related notion of compositional complexity is the *k-th order empirical entropy* of T, defined as $\mathsf{H}_k(T) = (1/|T|) \cdot \sum_W \sum_{a \in \Sigma^r(W)} f_T(Wa) \cdot \log(f_T(W)/f_T(Wa))$, where W ranges over all strings in $[1..\sigma]^k$. Clearly only the internal nodes of ST_T contribute to some $\mathsf{H}_k(T)$ [9], thus our methods allow computing $\mathsf{H}_k(T)$ for a user-specified *range of lengths* $[k_1..k_2]$ in $O(nd + k_2 - k_1)$ time, using just one pass over BWT_T.

4 Kernels and Complexity Measures on All Substrings

Given a string $T \in [1..\sigma]^{n-1}\#$, consider the infinite-dimensional vector $\mathbf{T}_\infty$ indexed by all distinct substrings $W \in [1..\sigma]^+$, such that $\mathbf{T}_\infty[W] = f_T(W)$. The *substring complexity* $\mathsf{C}_\infty(T)$ of T is the number of nonzero components of $\mathbf{T}_\infty$. The *substring kernel* of two strings T^1 and T^2 is the cosine of composition vectors $\mathbf{T}^1_\infty$ and $\mathbf{T}^2_\infty$. Computing substring complexity and substring kernel amounts to applying the same telescoping strategy described in Theorems 3 and 4, but with different contributions:

Corollary 1. *Let $T \in [1..\sigma]^{n-1}\#$ be a string. Given a data structure that supports* $\texttt{rangeDistinct}$ *queries on BWT_T, we can compute $\mathsf{C}_\infty(T)$ in $O(nd)$ time and in $o(n)$ bits of space in addition to the input.*

Proof. The substring complexity of T coincides with the number of characters in $[1..\sigma]$ that occur on all edges of ST_T. We can thus proceed as in Theorem 3, initializing $\mathsf{C}_\infty(T)$ to $(n-1)n/2$, or equivalently to the sum of the lengths of all suffixes of $T[1..n-1]$. Whenever we visit a node v of ST, we add to $\mathsf{C}_\infty(T)$ the quantity $|\ell(v)|$, and we subtract from $\mathsf{C}_\infty(T)$ the quantity $|\ell(v)| \cdot |\texttt{children}(v)|$. The net effect of all such operations coincides with summing the lengths of all edges of ST, discarding all occurrences of character #. Note that $|\ell(u)|$ is provided by Theorem 1, and $|\texttt{children}(v)|$ is the size of array $\texttt{chars}$ in $\texttt{repr}(\ell(v))$.

Corollary 2. *Let $T^1 \in [1..\sigma]^{n_1-1}\#_1$ and $T^2 \in [1..\sigma]^{n_2-1}\#_2$ be strings. Given data structures that support* $\texttt{rangeDistinct}$ *queries on BWT_{T^1} and on BWT_{T^2}, respectively, we can compute $\kappa(\mathbf{T}^1_\infty, \mathbf{T}^2_\infty)$ in $O(nd)$ time and in $o(n)$ bits of space in addition to the input, where $n = n_1 + n_2$.*

Proof. We proceed as in Theorem 4, setting again $N = 0$ and $D^i = (n_i - 1)n_i/2$ at the beginning of the algorithm. When we visit a node u of the generalized suffix tree of T^1 and T^2, we set N to $N + |\ell(u)| \cdot (f_1(\ell(u))f_2(\ell(u)) - \sum_v f_1(\ell(v))f_2(\ell(v)))$ and we set D^i to $D^i + |\ell(u)| \cdot (f_i(\ell(u))^2 - \sum_v f_i(\ell(v))^2)$, where v ranges over all children of u in the generalized suffix tree.

In a substring kernel it is common to weight a substring W by a user-specified function of its length: typical choices are $\epsilon^{|W|}$ for a given constant ϵ, or indicators that select only substrings within a specific range of lengths [16]. We denote by $\mathbf{T}^i_{\infty,g}$ a weighted version of the infinite-dimensional vector $\mathbf{T}^i_\infty$ such that $\mathbf{T}^i_{\infty,g}[W] = g(|W|) \cdot \mathbf{T}^i_\infty[W]$, where g is any user-specified function. We assume that the number of bits required to represent the output of g with sufficient precision is $O(\log n)$. It is easy to adapt Corollary 2 to support this type of composition vector:

Corollary 3. *Let $T^1 \in [1..\sigma]^{n_1-1}\#_1$ and $T^2 \in [1..\sigma]^{n_2-1}\#_2$ be strings. Given a function $g(k)$ that can be evaluated in constant time, and given data structures that support* `rangeDistinct` *queries on BWT_{T^1} and on BWT_{T^2}, respectively, we can compute $\kappa(\mathbf{T}^1_{\infty,g}, \mathbf{T}^2_{\infty,g})$ in $O(nd)$ time and in $o(n)$ bits of space in addition to the input, where $n = n_1 + n_2$.*

Proof. We modify Corollary 2 as follows. Assume that we are processing an internal node v of the generalized suffix tree, let $\ell(v) = W$, and assume that we have computed $\texttt{repr}'(aW)$ for all the left extensions aW of W. In addition to pushing $\texttt{repr}'(aW)$ onto the stack, we also push value $\texttt{prefixSum}(aW) = \sum_{i=1}^{|W|+1} g(i)^2$ with it, where $\texttt{prefixSum}(aW) = \texttt{prefixSum}(W) + g(|W| + 1)^2$. When we pop $\texttt{repr}'(aW)$, we compute its contributions to N and D^i as described in Corollary 2, but replacing $|aW|$ by $\texttt{prefixSum}(aW)$. We initialize D^i to $\sum_{j=1}^{n_i-1} g(j)^2$.

Corollary 3 can clearly support distinct weight functions for T^1 and T^2. For some functions, like $\epsilon^{|W|}$, prefix sums can be computed in closed form [16], thus there is no need to push `prefixSum` values on the stack. Another frequent weighting scheme for a string W associates a score $q(c)$ to every character c of W, and it weights W by e.g. $q(W) = \prod_{i=1}^{|W|} q(W[i])$. In this case we could just push $\texttt{prefixSum}(V) = \sum_{i=1}^{|V|} \prod_{j=1}^{i} q(V[j])^2$ onto the stack, where $V = aW$ and $\texttt{prefixSum}(V) = q(a)^2 \cdot (1 + \texttt{prefixSum}(W))$. A similar weighting scheme can be used for k-mers as well. Let $\mathbf{T}_{k,q}$ be a version of $\mathbf{T}_k$ such that $\mathbf{T}_{k,q}[W] = f_T(W) - (|T| - |W|)q(W)$ for every $W \in [1..\sigma]^k$, and consider the following distances defined in [13]:

$$D_2^s(\mathbf{T}^1_{k,q}, \mathbf{T}^2_{k,q}) = \sum_W \mathbf{T}^1_{k,q}[W]\mathbf{T}^2_{k,q}[W] / \sqrt{(\mathbf{T}^1_{k,q}[W])^2 + (\mathbf{T}^2_{k,q}[W])^2}$$

$$D_2^*(\mathbf{T}^1_{k,q}, \mathbf{T}^2_{k,q}) = \sum_W \mathbf{T}^1_{k,q}[W]\mathbf{T}^2_{k,q}[W] / \left(\sqrt{(n_1 - k)(n_2 - k)} \cdot q(W)\right)$$

where W ranges over all strings in $[1..\sigma]^k$. We can compute such distances using just a minor modification to Theorem 4:

Corollary 4. *Let $T^1 \in [1..\sigma]^{n_1-1}\#_1$ and $T^2 \in [1..\sigma]^{n_2-1}\#_2$ be strings. Given an integer k and data structures that support* `rangeDistinct` *queries on BWT_{T^1} and on BWT_{T^2}, respectively, we can compute $D_2^s(\mathbf{T}^1_{k,p}, \mathbf{T}^2_{k,p})$ and $D_2^*(\mathbf{T}^1_{k,p}, \mathbf{T}^2_{k,p})$ in $O(nd)$ time and in $\lambda \log \sigma + o(n)$ bits of space in addition to the input, where $n = n_1 + n_2$ and λ is the length of the longest repeat in T^1T^2.*

Proof. We proceed as in Theorem 4, pushing on the stack value $q(W,k) = \prod_{j=1}^{k} q(W[j])$ in addition to $\texttt{repr}'(W)$, and maintaining a separate stack of characters to represent the string we are processing during the depth-first traversal of the generalized suffix-link tree. We set $q(aW,k) = q(a) \cdot q(W,k)/q(b)$, where b is the kth character from the top of the character stack when we are processing W.

An orthogonal way to measure the similarity between T^1 and T^2 consists in comparing the repertoire of all strings that *do not appear* in T^1 and in T^2. Given a string T and two frequency thresholds $\tau_1 < \tau_2$, a string W is a *minimal rare word* of T if $\tau_1 \leq f_T(W) < \tau_2$ and if $f_T(V) \geq \tau_2$ for every proper substring V of W. Setting $\tau_1 = 0$ and $\tau_2 = 1$ gives the well-known *minimal absent words* (see e.g. [5,10] and references therein), whose total number can be $\Theta(\sigma n)$ [8]. Setting $\tau_1 = 1$ and $\tau_2 = 2$ gives the so-called *minimal unique substrings* (see e.g. [11] and references therein), whose total number is $O(n)$, like the number of strings obtained by any other setting of $\tau_1 \geq 1$. In what follows we focus on minimal absent words, but our algorithms can be generalized to other settings of the thresholds.

To decide whether aWb is a minimal absent word of T, where a and b are characters, it clearly suffices to check whether $f_T(aWb) = 0$ and whether both $f_T(aW) \geq 1$ and $f_T(Wb) \geq 1$. It is well known that only a maximal repeat of T can be the infix W of a minimal absent word aWb, and this applies to any setting of τ_1 and τ_2. To enumerate all the minimal absent words, for example to count their total number $\mathsf{C}_{-}(T)$, we can thus iterate over all nodes of ST_T associated with maximal repeats, as described below:

Theorem 6. *Let $T \in [1..\sigma]^{n-1}\#$ be a string. Given a data structure that supports* `rangeDistinct` *queries on BWT_T, we can compute $\mathsf{C}_{-}(T)$ in $O(nd)$ time and in $o(n)$ bits of space in addition to the input.*

Proof. For clarity, we first describe how to *enumerate* all the distinct minimal absent words of T: we specialize this algorithm to counting at the end of the proof. We use Theorem 1 to enumerate all nodes v of ST_T associated with maximal repeats, as described in Sect. 2.2. Let $\{a_1, \ldots, a_h\}$ be the set of distinct left extensions of string $\ell(v)$ in T returned by operation $\texttt{extendLeft}(\texttt{repr}(\ell(v)))$, let $\texttt{extensions}[1..\sigma+1, 0..\sigma]$ be a boolean matrix initialized to all zeros, and let $\texttt{leftExtensions}[1..\sigma+1]$ be an array initialized to all zeros. Let h' be a pointer initialized to one. Operation `extendLeft` allows following all the Weiner links from v, not necessarily in lexicographic order: for every string $a_i\ell(v)$ obtained in this way, we set $\texttt{leftExtensions}[h'] = a_i$, we enumerate its right extensions $\{c_1, \ldots, c_{k'}\}$ using array `chars` of $\texttt{repr}(a_i\ell(v))$, we set $\texttt{extensions}[h', c_j] = 1$

for all $j \in [1..k']$, and we finally increment h' by one. Note that only the columns of `extensions` that correspond to the right extensions of $\ell(v)$ are updated by this procedure. Then, we enumerate all the right extensions $\{b_1, \ldots, b_k\}$ of $\ell(v)$ using array `chars` of $\texttt{repr}(\ell(v))$, and for every such extension b_j we report all pairs (a_i, b_j) such that $a_i = \texttt{chars}[x]$, $x \in [1..h']$, and $\texttt{extensions}[x, b_j] = 0$. This process takes time proportional to the number of Weiner links from v, plus the number of children of v, plus the number of Weiner links from v multiplied by σ. When applied to all nodes of ST, this takes in total $O(n\sigma)$ time, which is optimal in the size of the output. The matrices and vectors used by this process can be reset to all zeros after processing each node: the total time spent in such reinitializations in $O(n)$.

If we just need $\mathsf{C}_-(T)$, rather than storing the temporary matrices `extensions` and `leftExtensions`, we store just a number `area` which we initialize to hk before processing node v. Whenever we observe a right extension c_j of a string $a_i\ell(v)$, we decrease `area` by one. Before moving to the next node, we increment $\mathsf{C}_-(T)$ by `area`.

Let $\mathbf{T}_-$ be the infinite-dimensional vector indexed by all distinct substrings $W \in [1..\sigma]^+$, such that $\mathbf{T}_-[W] = 1$ iff W is a minimal absent word of T. Theorem 6 can be adapted to compute the Jaccard distance between the composition vectors of two strings:

Corollary 5. *Let $T^1 \in [1..\sigma]^{n_1-1}\#_1$ and $T^2 \in [1..\sigma]^{n_2-1}\#_2$ be strings. Given data structures that support* `rangeDistinct` *queries on BWT_{T^1} and on BWT_{T^2}, respectively, we can compute $J(\mathbf{T}^1_-, \mathbf{T}^2_-)$ in $O(nd)$ time and in $o(n)$ bits of space in addition to the input, where $n = n_1 + n_2$.*

Proof. We apply the strategy of Theorem 6 to the internal nodes of the generalized suffix tree of T^1 and T^2 whose label is a maximal repeat of T^1 and a maximal repeat of T^2: such strings are clearly maximal repeats of T^1T^2 as well. We enumerate such nodes as described in Sect. 2.2. We keep a global variable `intersection` and a bitvector $\texttt{sharedRight}[1..\sigma]$. For every node v that corresponds to a maximal repeat of T^1 and of T^2, we merge the sorted arrays $\texttt{chars}_1$ and $\texttt{chars}_2$ of $\texttt{repr}'(\ell(v))$, we set $\texttt{sharedRight}[c] = 1$ for every character c that belongs to the intersection of the two arrays, and we cumulate in a variable k' the number of ones in `sharedRight`. Then, we scan every left extension a_i provided by $\texttt{extendLeft}'$, we determine in constant time whether it occurs in both T^1 and T^2, and if so we increment a variable h' by one. Finally, we initialize a variable `area` to $h'k'$, and we process again every left extension a_i provided by $\texttt{extendLeft}'$: if $a_i\ell(v)$ occurs in both T^1 and T^2, we compute the union of arrays $\texttt{chars}_1$ and $\texttt{chars}_2$ of $\texttt{repr}'(a_i\ell(v))$, and for every character c in the union such that $\texttt{sharedRight}[c] = 1$, we decrement `area` by one. At the end of this process, we add `area` to the global variable `intersection`. To compute $||\mathbf{T}^1_- \vee \mathbf{T}^2_-||$ we apply Theorem 6 to T^1 and T^2 separately.

It is easy to extend Corollary 5 to compute $\kappa(\mathbf{T}^1_-, \mathbf{T}^2_-)$, as well as to support weighting schemes based on the length and on the characters of minimal absent words.

5 Markovian Corrections

In some applications it is desirable to assign to component $W \in [1..\sigma]^k$ of composition vector $\mathbf{T}_\infty$ an estimate of the *statistical significance* of observing $f_T(W)$ occurrences of W in T: intuitively, strings whose frequency departs from its expected value are more likely to carry "information", and they should be weighted more [12]. Assume that T is generated by a Markov random process of order $k-2$ or smaller, that produces strings on alphabet $[1..\sigma]$ according to a probability distribution $\mathbb{P}$. It is well known that the probability of observing W in a string generated by such a random process is $\mathbb{P}(W) = \mathbb{P}(W[1..k-1]) \cdot \mathbb{P}(W[2..k])/\mathbb{P}(W[2..k-1])$. We can estimate $\mathbb{P}(W)$ using the empirical probability $p_T(W)$, obtaining the following approximation for $\mathbb{P}(W)$: $\tilde{p}_T(W) = p_T(W[1..k-1]) \cdot p_T(W[2..k])/p_T(W[2..k-1])$ if $p_T(W[2..k-1]) \neq 0$, and $\tilde{p}_T(W) = 0$ otherwise. We can thus estimate the significance of the event that substring W has empirical probability $p_T(W)$ in string T using the following score: $z_T(W) = (p_T(W) - \tilde{p}_T(W))/\tilde{p}_T(W)$ if $\tilde{p}_T(W) \neq 0$, and $z_T(W) = 0$ if $\tilde{p}_T(W) = 0$ [12]. After elementary manipulations [2], $z_T(W)$ becomes:

$$z_T(W) = g(n,k) \cdot \frac{f_T(W) \cdot f_T(W[2..k-1])}{f_T(W[1..k-1]) \cdot f_T(W[2..k])} - 1$$
$$g(x,y) = (x-y+2)^2/(x-y+1)(x-y+3)$$

Since $g(x,y) \in [1..1.125]$, we temporarily assume $g(x,y) = 1$ in what follows, removing this assumption later.

Let $\mathbf{T}_z$ be a version of the infinite-dimensional vector $\mathbf{T}_\infty$ in which $\mathbf{T}_z[W] = z_T(W)$. Among all strings that occur in T, only strings aWb such that a and b are characters in $[0..\sigma]$ and such that W is a maximal repeat of T can have $\mathbf{T}_z[aWb] \neq 0$. Similarly, among all strings that *do not occur* in T, only the minimal absent words of T have a nonzero component in $\mathbf{T}_z$: specifically, $\mathbf{T}_z[aWb] = -1$ for all minimal absent words aWb of T, where a and b are characters in $[0..\sigma]$ [2]. Given two strings T^1 and T^2, we can thus compute $\kappa(\mathbf{T}^1_z, \mathbf{T}^2_z)$ using the same strategy as in Corollary 5:

Theorem 7. *Let $T^1 \in [1..\sigma]^{n_1-1}\#_1$ and $T^2 \in [1..\sigma]^{n_2-1}\#_2$ be strings. Given data structures that support* `rangeDistinct` *queries on* BWT_{T^1} *and on* BWT_{T^2}*, respectively, and assuming $g(x,y) = 1$ for all settings of x and y, we can compute $\kappa(\mathbf{T}^1_z, \mathbf{T}^2_z)$ in $O(nd)$ time and in $o(n)$ bits of space in addition to the input, where $n = n_1 + n_2$.*

Proof. We focus here on computing component N of $\kappa(\mathbf{T}^1_z, \mathbf{T}^2_z)$: computing D^i follows a similar algorithm on BWT_{T^i}. We keep again a bitvector `sharedRight`$[1..\sigma]$, and we enumerate all the internal nodes of the generalized suffix tree of T^1 and T^2 whose label is a maximal repeat of T^1 and a maximal repeat of T^2, as described in Corollary 5. For every such node v, we merge the sorted arrays $\texttt{chars}_1$ and $\texttt{chars}_2$ of $\texttt{repr}'(\ell(v))$, we set `sharedRight`$[c] = 1$ for every character c that belongs to the intersection of the two arrays, and we cumulate in a variable k' the number of ones in `sharedRight`. Then, we scan every left

extension a_i provided by `extendLeft'`, we determine in constant time whether it occurs in both T^1 and T^2, and if so we increment a variable h' by one. Finally, we initialize a variable `area` to $h'k'$, and we process again every left extension a_i provided by `extendLeft'`. If $a_i\ell(v)$ occurs in both T^1 and T^2, we merge arrays $\texttt{chars}_1$ and $\texttt{chars}_2$ of $\texttt{repr}'(a_i\ell(v))$: for every character b in the intersection of $\texttt{chars}_1$ and $\texttt{chars}_2$, we add to N value $z_1(a_i\ell(v)b) \cdot z_2(a_i\ell(v)b)$, retrieving the corresponding frequencies from $\texttt{repr}'(a_i\ell(v))$ and from $\texttt{repr}'(\ell(v))$, and we decrement `area` by one. For every character b that occurs only in $\texttt{chars}_1$, we test whether $\texttt{sharedRight}[b] = 1$: if so, a_iWb is a minimal absent word of T^2 that occurs in T^1, thus we decrement `area` by one and we add to N value $-z_1(a_i\ell(v)b)$. We proceed symmetrically if b occurs only in $\texttt{chars}_2$. At the end of this process, `area` counts the number of minimal absent words with infix $\ell(v)$ that are shared by T^1 and T^2: thus, we add `area` to N.

It is easy to remove the assumption that $g(x, y)$ is always equal to one. There are only two differences from the previous case. First, the score of the substrings W of T^i that have a maximal repeat of T^i as an infix changes, but $g(n_i, |W|)$ can be immediately computed from $|W|$, which is provided by the enumeration algorithm. Second, the score of all substrings W of T^i that do not have a maximal repeat as an infix changes from zero to $g(n_i, |W|) - 1$: we can take all such contributions into account by pushing prefix-sums to the stack, as in Corollary 3. For example, to compute component N of $\kappa(\mathbf{T}^1_z, \mathbf{T}^2_z)$, we can first assume that *all* substring W that occur both in T^1 and in T^2 have score $g(n_i, |W|) - 1$, by pushing on the stack the prefix-sums described in [2] and by enumerating only nodes v of the generalized suffix tree of T^1 and T^2 such that $\ell(v)$ occurs both in T^1 and in T^2. Then, we can run a similar algorithm as in Theorem 7, subtracting quantity $(g(n_1, |W| + 2) - 1) \cdot (g(n_2, |W| + 2) - 1)$ from the contribution to N of every string a_iWb that occurs both in T^1 and in T^2.

Finally, recall that in Sect. 3 we mentioned the problem of determining an *upper bound* on the values of k to be used in k-mer kernels. Let $\mathbf{T}_k$ be the composition vector indexed by all strings in $[1..\sigma]^k$ such that $\mathbf{T}_k[W] = p_T(W)$, and let $\tilde{\mathbf{T}}_k$ be a similar composition vector with $\tilde{\mathbf{T}}_k[W] = \tilde{p}_T(W)$, where $\tilde{p}_T(W)$ is defined as in the beginning of this section. It makes sense to disregard values of k for which $\mathbf{T}_k$ and $\tilde{\mathbf{T}}_k$ are very similar, and more formally whose Kullback-Leibler divergence $\mathsf{KL}(\mathbf{T}_k, \tilde{\mathbf{T}}_k) = \sum_W \mathbf{T}_k[W] \cdot (\log(\mathbf{T}_k[W]) - \log(\tilde{\mathbf{T}}_k[W]))$ is small, where W ranges over all strings in $[1..\sigma]^k$. Thus, we could use as an upper bound on k the minimum value k^* such that $\sum_{k'=k^*}^{\infty} \mathsf{KL}(\mathbf{T}_{k'}, \tilde{\mathbf{T}}_{k'}) < \tau$ for some user-specified threshold τ [15]. Note again that only strings aWb such that a and b are characters in $[0..\sigma]$ and W is a maximal repeat of T contribute to $\mathsf{KL}(\mathbf{T}_{|W|+2}, \tilde{\mathbf{T}}_{|W|+2})$. We can thus adapt Theorem 7 to compute the KL divergence for a user-specified *range of lengths* $[k_1..k_2]$, using just one pass over BWT_T, in $O(nd)$ time and in $o(n)$ bits of space in addition to the input and the output. The same approach can be used to compute the *KL-divergence kernel* $\kappa(\mathbf{T}^1_{KL}, \mathbf{T}^2_{KL})$, where $\mathbf{T}^i_{KL}[W] = \mathsf{KL}_{T^i}(W)$ and $\mathsf{KL}_{T^i}(W) = \sum_{a,b \in \Sigma} p_{T^i}(aWb) \cdot (\log(p_{T^i}(aWb)) - \log(\tilde{p}_{T^i}(aWb)))$.

References

1. Apostolico, A.: Maximal words in sequence comparisons based on subword composition. In: Elomaa, T., Mannila, H., Orponen, P. (eds.) Ukkonen Festschrift 2010. LNCS, vol. 6060, pp. 34–44. Springer, Heidelberg (2010)
2. Apostolico, A., Denas, O.: Fast algorithms for computing sequence distances by exhaustive substring composition. Algorithms Mol. Biol. **3**(1), 13 (2008)
3. Belazzougui, D.: Linear time construction of compressed text indices in compact space. In Symposium on Theory of Computing, STOC 2014, New York, NY, USA, 31 May–03 June, pp. 148–193 (2014)
4. Belazzougui, D., Navarro, G., Valenzuela, D.: Improved compressed indexes for full-text document retrieval. J. Discret. Algorithms **18**, 3–13 (2013)
5. Chairungsee, S., Crochemore, M.: Using minimal absent words to build phylogeny. Theoret. Comput. Sci. **450**, 109–116 (2012)
6. Chikhi, R., Medvedev, P.: Informed and automated k-mer size selection for genome assembly. Bioinformatics **30**(1), 31–37 (2014)
7. Chor, B., Horn, D., Goldman, N., Levy, Y., Massingham, T., et al.: Genomic DNA k-mer spectra: models and modalities. Genome Biol. **10**(10), R108 (2009)
8. Crochemore, M., Mignosi, F., Restivo, A.: Automata and forbidden words. Inf. Process. Lett. **67**(3), 111–117 (1998)
9. Gog, S.: Compressed suffix trees: design, construction, and applications. Ph.D. thesis, University of Ulm, Germany (2011)
10. Herold, J., Kurtz, S., Giegerich, R.: Efficient computation of absent words in genomic sequences. BMC Bioinform. **9**(1), 167 (2008)
11. İleri, A.M., Külekci, M.O., Xu, B.: Shortest unique substring query revisited. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) CPM 2014. LNCS, vol. 8486, pp. 172–181. Springer, Heidelberg (2014)
12. Qi, J., Wang, B., Hao, B.-I.: Whole proteome prokaryote phylogeny without sequence alignment: a k-string composition approach. J. Mol. Evol. **58**(1), 1–11 (2004)
13. Reinert, G., Chew, D., Sun, F., Waterman, M.S.: Alignment-free sequence comparison (I): statistics and power. J. Comput. Biol. **16**(12), 1615–1634 (2009)
14. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, Cambridge (2004)
15. Sims, G.E., Jun, S.-R., Wu, G.A., Kim, S.-H.: Alignment-free genome comparison with feature frequency profiles (FFP) and optimal resolutions. Proc. Natl. Acad. Sci. **106**(8), 2677–2682 (2009)
16. Smola, A.J., Vishwanathan, S.V.N.: Fast kernels for string and tree matching. In: Becker, S., Thrun, S., Obermayer, K. (eds.) Advances in Neural Information Processing Systems 15, pp. 585–592. MIT Press, Cambridge (2003)