Fast matching statistics in small space

Djamal Belazzougui
DTISI-CERIST
Algiers, Algeria.

Fabio Cunial
MPI-CBG and CSBD
Dresden, Germany.
cunial@mpi-cbg.de

Olgert Denas
Adobe Inc.
San Jose, California, USA.

Abstract

Computing the matching statistics of a string S with respect to a string T on an alphabet of size σ is a fundamental primitive for a number of large-scale string analysis applications, including the comparison of entire genomes, for which space is a pressing issue. This paper takes from theory to practice an existing algorithm that uses just $O(|T| \log \sigma)$ bits of space, and that computes a compact encoding of the matching statistics array in $O(|S| \log \sigma)$ time. The techniques used to speed up the algorithm are of general interest, since they optimize queries on the existence of a Weiner link from a node of the suffix tree, and parent operations after unsuccessful Weiner links. Thus, they can be applied to other matching statistics algorithms, as well as to any suffix tree traversal that relies on such calls. Some of our optimizations yield a matching statistics implementation that is up to three times faster than a plain version of the algorithm, depending on the similarity between S and T. In genomic datasets of practical significance we achieve speedups of up to 1.8, but our fastest implementations take on average twice the time of an existing code based on the LCP array. The key advantage is that our implementations need between one half and one fifth of the competitor's memory, and they approach comparable running times when S and T are very similar.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis, Theory of computation → Pattern matching

Keywords and phrases Matching statistics, maximal repeat, Burrows-Wheeler transform, wavelet tree, suffix tree topology

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.17

Supplement Material Source code at `https://github.com/odenas/indexed_ms`.

Acknowledgements We thank German Tischler for help with adapting the implementation in [15]. We thank Oscar Gonzalez and Peter Steinbach for help setting up the experiments on the MPI-CBG cluster.

1 Introduction

The *matching statistics* of a string S, called the *query*, with respect to another string T, called the *text*, is an array $\mathsf{MS}_{S,T}[1..|S|]$ such that $\mathsf{MS}_{S,T}[i]$ is the length of the longest prefix of $S[i..|S|]$ that occurs in T. MS is almost as old as the suffix tree itself, with applications that include file transmission [26], the detection of sequencing errors and single-nucleotide

variations in read collections [17], the computation of substring kernels between two sequences by indexing just one of them [15, 22], and the construction of whole-genome phylogenies [5, 25] using the average value of MS as a proxy for the cross-entropy of random sources [6]. The effectiveness of matching statistics in alignment-free phylogenetics has also motivated variants that allow for a user-specified number of mismatches (see e.g. [1, 11, 18, 23, 24] and references therein). Despite the typical input being of genomic scale, however, few of these methods are explicitly designed for space efficiency.

Computing $\mathsf{MS}_{S,T}$ is a classical problem in string processing. In most practical cases, a large number of queries S are asked to a constant T, motivating the construction of an index on T. The textbook solution scans S from left to right while traversing suffix links and child links in the *suffix tree* of T, spending a total $O(|S| \log \sigma)$ time [1]. Symmetrically, S can be scanned from right to left, while taking Weiner links and parent links in the (compressed) suffix tree of T [15]. Both ways require access to the string depth of a suffix tree node, which can be encoded implicitly in $\Theta(|T| \log |T|)$ bits of total space in the *longest common prefix* (LCP) array of T, and decoded in constant time per node using range-minimum queries on the array [15]. In some practical datasets, most LCP values are small and can be encoded in less than one byte. Alternatively, each depth value can be extracted from a compressed suffix array. If such array is encoded in $O(|T| \log \sigma)$ bits, the best known time complexity for this operation is $O(\log^{\epsilon} |T|)$, which is often fast enough in practice [19]. Asymptotic complexity increases when the compressed suffix array is encoded in $|T| \log \sigma + o(|T|) + O(\mathtt{min}(\sigma, |T| \log \sigma))$ bits or in $|T| \log \sigma (1 + o(1)) + O(\mathtt{min}(\sigma, |T| \log \sigma))$ bits. A third approach does not require access to string depths at all, and achieves $O(|T| \log \sigma)$ bits of peak space while keeping running time to $O(|S| \log \sigma)$ given the indexes [3]. Such asymptotic performance is attractive for datasets, like whole genomes or collections of genomes, with large overall size and long repeats. The algorithm in [3] has an additional feature that is useful for large datasets: it does not output $\mathsf{MS}_{S,T}$ itself, which takes $|S| \log |S|$ bits, but an encoding that takes just $2|S|$ bits, and that allows one to later retrieve $\mathsf{MS}[i]$ in constant time for any i, using just $o(|S|)$ more bits.

This paper studies a number of practical variants of the algorithm described in [3], which use the same data structures as in the original paper, but improve on its speed. As customary, since indexing T is performed only once and requires standard data structures, we disregard index construction and focus just on the time for computing $\mathsf{MS}_{S,T}$ given the indexes.

2 Preliminaries

2.1 Strings

Let $\Sigma = [1..\sigma]$ be an integer alphabet, let $\# = 0 \notin \Sigma$ be a separator, and let $T = [1..\sigma]^{n-1}\#$ be a string. Given a string $W \in [1..\sigma]^k$, we call the *reverse of* W the string $\overline{W}$ obtained by reading W from right to left. For a string $W \in [1..\sigma]^k\#$ we abuse notation, denoting by $\overline{W}$ the string $\overline{W[1..k]}\#$. We call *repeat* a substring W of T that occurs more than once in T, and we call *left-extensions* (respectively, *right-extensions*) of W the set of distinct characters that precede (respectively, follow) the occurrences of W in T. A repeat W is *left-maximal* (respectively, *right-maximal*) iff it has more than one left-extension (respectively, right-extension). A *maximal repeat* of T is a repeat that is both left- and right-maximal.

[1] Here σ denotes the size of the alphabet from which strings S and T are drawn. We assume throughout this paper that σ is at most polynomial in $\mathtt{max}(|T|, |S|)$.

For reasons of space we assume the reader to be familiar with the notion of *suffix tree* $\mathsf{ST}_T = (V, E)$ of T, which we do not define here. We denote by $\ell(v)$ the string label of node $v \in V$. It is well known that a substring W of T is right-maximal iff $W = \ell(v)$ for some internal node v of the suffix tree. It is also known that the maximal repeats of T form a subset of the internal nodes of ST_T that is closed under ancestor operation, in the sense that if the label of an internal node v is a maximal repeat of T, then the labels of all the ancestors of v are also maximal repeats of 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 $\ell(w) = W$. Here we just recall that inverting the direction of all suffix links yields the so-called *explicit Weiner links*. Given an internal node v of ST_T 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: all such left extensions of internal nodes that end in the middle of an edge or at a leaf are called *implicit Weiner links*. Note that an internal node of ST_T can have more than one outgoing Weiner link, and that all such Weiner links have distinct labels.

We assume the reader to be familiar also with the Burrows-Wheeler transform of T (denoted BWT_T in what follows), including the notions of lexicographic interval of the label of a node of ST_T, backward step, rank and select queries on bitvectors, wavelet trees, and rank queries on wavelet trees. Finally, we call *suffix tree topology* any data structure that supports the following operations on ST_T: $\texttt{parent}(\texttt{id}(v))$, which returns the identifier of the parent of a node v with identifier $\texttt{id}(v)$; $\texttt{lca}(\texttt{id}(u), \texttt{id}(v))$, which returns the identifier of the lowest common ancestor of nodes u and v; $\texttt{leftmostLeaf}(\texttt{id}(v))$ and $\texttt{rightmostLeaf}(\texttt{id}(v))$, which compute the identifier of the leftmost (respectively, rightmost) leaf in the subtree rooted at node v; $\texttt{selectLeaf}(i)$, which returns the identifier of the i-th leaf in preorder traversal; $\texttt{leafRank}(\texttt{id}(v))$, which computes the number of leaves that occur before leaf v in preorder traversal; $\texttt{isAncestor}(\texttt{id}(u), \texttt{id}(v))$, which tells whether u is an ancestor of v. For brevity, we write just v rather than $\texttt{id}(v)$ in all such operations in what follows. It is known that the topology of an ordered tree with n nodes can be represented using $2n + o(n)$ bits as a sequence of $2n$ balanced parentheses, and that $2n + o(n)$ more bits suffice to support the operations described above in constant time [12, 21]. We drop subscripts whenever the reference strings are clear from the context.

2.2 Matching statistics in small space

$\mathsf{MS}_{S,T}$ can be represented as a bitvector $\texttt{ms}$ of $2|S|$ or $2|S|-1$ bits, which is built by appending, for each $i \in [0..|S| - 1]$ in increasing order, $\mathsf{MS}_{S,T}[i] - \mathsf{MS}_{S,T}[i-1] + 1$ zeros followed by a one [3] ($\mathsf{MS}_{S,T}[-1]$ is set to one for convenience). Since the number of zeros before the i-th one in $\texttt{ms}$ equals $i + \mathsf{MS}_{S,T}[i]$, one can compute $\mathsf{MS}_{S,T}[i]$ for any $i \in [0..|S| - 1]$ using select operations on $\texttt{ms}$. The algorithm described in [3] computes $\texttt{ms}$ using both a backward and a forward scan over S, and it needs in each scan just BWT_T with rank support, and the topology of ST_T, or just $\mathsf{BWT}_{\overline{T}}$ with rank support, and the topology of $\mathsf{ST}_{\overline{T}}$. The two phases are connected via a bitvector $\texttt{runs}[1..|T| - 1]$, such that $\texttt{runs}[i] = 1$ iff $\mathsf{MS}[i] = \mathsf{MS}[i-1] - 1$, i.e. iff there is no zero between the i-th and the $(i-1)$-th ones in $\texttt{ms}$.

First, we scan S from right to left, using BWT_T with rank support, and the suffix tree topology of T, to determine the runs of consecutive ones in $\texttt{ms}$. Assume that we know the interval $[i..j]$ in BWT_T that corresponds to substring $W = S[k..k + \mathsf{MS}[k] - 1]$, as well as the identifier of the proper locus v of W in the topology of ST_T. We try to perform a backward step using character $a = S[k-1]$: if the resulting interval $[i'..j']$ is nonempty, we set $\texttt{runs}[k] = 1$ and we reset $[i..j]$ to $[i'..j']$. Otherwise, we set $\texttt{runs}[k] = 0$, we update the BWT interval to the interval of $\texttt{parent}(v)$ using the topology, and we try another backward step with character a.

In the second phase we scan S from left to right, using $\mathsf{BWT}_{\overline{T}}$ with rank support, a representation of the suffix tree topology of $\overline{T}$, and bitvector `runs`, to build `ms`. Assume that we know the interval $[i..j]$ in $\mathsf{BWT}_{\overline{T}}$ that corresponds to substring $W = S[k..h-1]$ such that $\mathsf{MS}[k-1] = h-k$ but $\mathsf{MS}[k] \geq h-k$. We try to perform a backward step with character $S[h]$: if the backward step succeeds, we continue issuing backward steps with the following characters of S, until we reach a position h^* in S such that a backward step with character $S[h^*]$ from the interval $[i^*..j^*]$ of substring $W^* = S[k..h^*-1]$ in $\mathsf{BWT}_{\overline{T}}$ fails. At this point we know that $\mathsf{MS}[k] = h^*-k$, so we append $h^*-k-\mathsf{MS}[k-1]+1 = h^*-h+1$ zeros and a one to `ms`. Moreover, we iteratively reset the current interval in $\mathsf{BWT}_{\overline{T}}$ to the interval of $\texttt{parent}(v^*)$, where v^* is the proper locus of W^* in $\mathsf{ST}_{\overline{T}}$, and we try another backward step with character $S[h^*]$, until we reach an interval $[i'..j']$ for which the backward step succeeds. Let this interval correspond to substring $W' = S[k'..h^*-1]$. Note that $\mathsf{MS}[k'] > \mathsf{MS}[k'-1]-1$ and $\mathsf{MS}[x] = \mathsf{MS}[x-1]-1$ for all $x \in [k+1..k'-1]$, thus k' is the position of the first zero to the right of position k in `runs`, and we can append $k'-k-1$ ones to `ms`. Finally, we repeat the whole process from substring $S[k'..h^*]$ and its interval in $\mathsf{BWT}_{\overline{T}}$.

Note that S is read sequentially in both phases, `runs` and `ms` are built by iteratively appending bits at one of their ends, and `runs` is read sequentially in the second phase. Thus, there is no need to keep any of these vectors fully in memory, and they can be streamed to or from disk in practice.

3 Fast matching statistics in small space

We assume that each BWT is represented as a wavelet tree, and that each suffix tree topology is represented as a sequence of balanced parentheses. Thus, a node of a suffix tree is described by two intervals: its interval in the BWT, and the pair of open and closed parentheses in the sequence of balanced parentheses. We can derive the identifier in the topology from the BWT interval, by issuing two `selectLeaf` operations followed by one `lca` operation. Vice versa, we can derive the BWT interval from the identifier in the topology, by issuing one `leftmostLeaf` and one `rightmostLeaf` operation, followed by two `leafRank` operations. As a baseline, we consider an implementation of the algorithm in Section 2.2 in which both the BWT interval and the identifier in the topology are updated at every step; in such implementation, taking a Weiner link from a node involves calling nine functions provided by the wavelet tree or the topology.

The two phases of the algorithm in Section 2.2 are symmetrical, thus the optimizations described in this section apply equally to both and have similar effects in practice.

3.1 Faster Weiner links

Following a Weiner link with label c from a node v of the suffix tree requires issuing two rank queries, for the same character, on the BWT interval $[i..j]$ of v. Such queries traverse the same nodes of the wavelet tree and, in the bitvector of each node, they access positions whose distance is at most $j-i$, and potentially decreases as the search goes deeper in the wavelet tree. Our first optimization consists in merging the two rank queries into a single `doubleRank` query, which, when the distance between its arguments is small, invokes just once some instructions (like counting the number of ones in some memory words) that would be executed twice by two distinct rank calls.

Note that the algorithm uses rank queries on the BWT interval of v also to decide whether a Weiner link with character c starts from v or not. The other advantage of merging the two rank queries is that we can detect whether the current interval is empty at each level

of the wavelet tree, and if so we can quit the traversal immediately. We call this further optimization `doubleRankAndFail`.

If node v has indeed a Weiner link labeled by c, we need to derive, from the BWT interval that results from taking such Weiner link, the identifier of the corresponding node v' in the suffix tree topology. However, the identifier of v' in the topology is used by our algorithm only if, in the following iteration, we need to move to the parent of v', and in turn this happens only when taking a Weiner link from v' fails. In other words, inside a maximal sequence of successful Weiner links, we need just to update one BWT interval per iteration, and this costs just one `doubleRank` operation per iteration; we call this optimization `lazyWeinerLink`.

Knowing the repeat structure of T allows one to optimize Weiner links even further. Specifically, if v is not a maximal repeat of T, its BWT interval contains exactly one distinct character, thus it suffices to check whether e.g. the last position of the interval is equal to c, and if so to issue just one rank operation. We call this optimization `maxrepWeinerLink`. Even more aggressively, we can implement a `rankAndCheck` query which, in a single operation, checks whether the last position of the BWT interval matches c, stops the traversal of the wavelet tree as soon as it detects an empty interval, and returns the rank in case of success. The same optimization can be applied to intervals of size one. We detect whether a node is a maximal repeat by building a bitvector `marked`$[1..|T|]$, where we set `marked`$[i] = 1$ and `marked`$[j] = 1$ for every interval $[i..j]$ of a maximal repeat of T. We build `marked` by traversing the suffix tree topology depth-first, avoiding to explore the subtree of a node that is not a maximal repeat since it cannot contain other maximal repeats. To determine whether a given interval $[i..j]$ with $j > i$ corresponds to a maximal repeat, we just check whether `marked`$[i] = 1$ and `marked`$[j] = 1$. Recall that, if a node of the suffix tree is a maximal repeat, then all its ancestors in the suffix tree are also maximal repeats. This implies that the interval of a node that is not a maximal repeat is properly contained inside the interval of a maximal repeat, and it does not contain the interval of any maximal repeat, thus it must have either its first bit or its last bit equal to zero. If $i = j$, the interval is not a repeat.

3.2 Faster parent operations

Recall that, when a Weiner link labeled by character c fails from a node v, we iteratively issue a parent operation on the topology, we convert the node identifier in the topology to its BWT interval, and we check whether c occurs inside such interval. Knowing whether a node is a maximal repeat or not enables once again a number of optimizations. First, if v is not a maximal repeat, then none of its ancestors that are not themselves maximal repeats have a Weiner link labeled with character c, so we can avoid trying the Weiner link from an ancestor of v that is not a maximal repeat. Moreover, as soon as an ancestor u of v is a maximal repeat, every one of the ancestors of u is a maximal repeat as well, so we do not need to check whether the current node is a maximal repeat after the following parent operations, if any.

Even more aggressively, when a Weiner link fails, we could directly move from node v with BWT interval $[i..j]$, to its lowest ancestor that is a maximal repeat, by taking the LCA between the rightmost one up to i in the `marked` bitvector, and the leftmost one starting from j. A further step along this line consists in moving directly to the lowest ancestor v_c of v whose BWT interval contains c: such ancestor, if any, is necessarily a maximal repeat. This technique, which we call `parentShortcut`, can be implemented as follows. We move to the rightmost occurrence of c before the interval of v (if any), by issuing a rank and a select query on the BWT, then we move to the corresponding leaf w of the suffix tree by issuing a `selectLeaf` query on the topology, and we compute the identifier of the ancestor

$p = \texttt{lca}(v, w)$ of v. We do the same for the leftmost occurrence of c to the right of the interval of v (if any), computing the identifier of an ancestor q of v. Finally, we set v_c to the lowest of p and q, by issuing one `isAncestor` query provided by the topology. The implementation of `parentShortcut` can be further optimized in practice. For example, if the interval of p ends at the same position as the interval of v, we don't even need to compute the leftmost occurrence of c after the interval of v. And if the interval of p does not contain the leftmost occurrence of c after the interval of v, we can just return p.

We briefly note that, in addition to being useful in practice, the `parentShortcut` technique implies also an on-line algorithm for matching statistics:

▶ **Lemma 1.** *There is an online algorithm that computes* $\mathsf{MS}_{S,T}$ *in* $O(\log \sigma)$ *time per character of* S*, and in* $|T| \log \sigma(1 + o(1)) + O(\texttt{min}(\sigma, |T| \log \sigma))$ *bits of space, by scanning* S *from right to left.*

Proof. The algorithm has the same structure as the one described in [15], but it spends a bounded amount of time per character of S. Recall that rank queries can be implemented in $O(\log \sigma)$ time with a wavelet tree, that select queries can be implemented in $O(\log \sigma)$ time as well using additional $|T|o(\log \sigma)$ bits, and that `parentShortcut` takes constant time in addition to select queries. Since the node reached by `parentShortcut` is a maximal repeat, we just need to compute the string depth of maximal repeats throughout the algorithm. This can be done in constant time, by storing also the topology of the suffix-link tree of $\overline{T}$ and suitable bitvectors to commute between maximal repeats in the two topologies, as described in [4, Section 6]. ◀

3.2.1 Selecting the next occurrence of a character

Selecting the previous or the next occurrence of a character from a given position of the BWT of T lies at the core of the `parentShortcut` operation of Section 3.2, and it is a primitive of independent interest. We implement a generalization $\texttt{selectNext}_T(i, j, c)$, which returns the j-th occurrence of character c strictly to the right of position i, in a string T that is assumed to be represented as a wavelet tree. For brevity we focus here just on the case in which $T[i] = c$, since the case in which $T[i] \neq c$ can be handled similarly.

We move from the root of the wavelet tree to a leaf, as in the rank operation at position i, remembering the position i_k that we reach in the bitvector of each level k. We invoke $j_k = \texttt{selectNext}(i_k, j, c_k)$ on the bitvector of the leaf, where c_k is the k-th bit in the binary representation of c, then we move up to the bitvector of the parent and we invoke $j_{k-1} = \texttt{selectNext}(i_{k-1}, j_k - i_k, c_{k-1})$; we continue in this way until we reach the root. A version of `selectNext` for bitvectors that uses a sequential, word-based scan, was already described in [9]. We take a similar approach, issuing a select operation on the bits of the computer word that contains position i_k, and on the following word if necessary (using class `bits` in SDSL). In case such selects fail, we issue a standard select operation on the whole bitvector. Note that the number of words that we check explicitly, in addition to the one containing i_k, could be bigger for bitvectors that are closer to the root of the wavelet tree: we leave such variant to the full version of the paper.

A more advanced strategy could consist in dividing the bitvector of length $|T|$ into blocks of size B each, keeping an auxiliary bitvector $\texttt{pure}[1..|T|/B]$ to mark the blocks that contain just one distinct bit. Given a position i in the original bitvector, we could compute the block b that contains it, and we could check $\texttt{pure}[b]$ to see if such block is pure. If this is the case we could return $i + 1$, or keep scanning $\texttt{pure}[b + 1..|T|/B]$ until we find either a pure block with the correct value, or an impure block. If block b is impure, we could scan it

starting from position i and, in case of failure, we could continue as above. To speed up the search even further, we could introduce block clusters of size B^2, storing for each cluster the position of the next pure cluster and of the next impure cluster. We leave the study of these and other variants to the full version of the paper.

Recall that we use `selectNext` when a Weiner link fails from a BWT interval $[i..j]$. Thus, one could create a `doubleRankAndSelectNext` operation that reuses the work performed by `doubleRank` when going down the wavelet tree in `selectNext`. Specifically, the only overhead of this operation, compared to a successful `doubleRankAndFail`, is saving the positions to which i and j are projected in each bitvector. In case of failure, the procedure does not abort but keeps going down to a leaf, from which it finally moves up, selecting both the next occurrence (from j) and the previous occurrence (from i). As with `doubleRank`, performing both selections at each level might save time by executing some instructions common to them only once. In case of success, the bottom-up phase is not executed. For reasons of time, we leave a full experimental study of `doubleRankAndSelectNext` to the full version of the paper.

Finally, note that a fast implementation of `selectNext` can be useful for answering *range matching statistics queries* on bitvector `ms`, i.e. queries that ask for all $\mathsf{MS}[k]$ values for k in a user-specified $[i..j]$, or for the average or maximum of such values, since one could replace $j - i + 1$ select operations with one select operation and $j - i$ `selectNext` operations. We leave the experimental study of range MS queries to the full version of the paper.

4 Implementation

Our C++ implementation, available at `https://github.com/odenas/indexed_ms`, is based on the SDSL library [8], which we adapt to our purposes. Specifically, we modify the following parts of SDSL: (1) the rank data structure `rank_support_v`; (2) the implementation `wt_pc` of the wavelet tree for byte sequences (using other wavelet tree variants, e.g. Huffman-shaped, is beyond the scope of this paper); (3) the implementation `csa_wt` of the compressed suffix array based on a wavelet tree; (4) the code for select operations on bitvectors `select_support_mcl`. We build the BWT with `dbwt` [16], and the other parts of the index by using `csa_wt` to represent a compressed suffix array without samples, and by stripping down the compressed suffix tree code in `cst_sct3` (among other things, by removing the LCP array). We represent the suffix tree topology explicitly with balanced parentheses, using a simplified version of `bp_support_sada` that supports just the `parent` operation. Recall that `bp_support_sada` is based on [20, 21], and that SDSL provides other representations `bp_support_gg` and `bp_support_g` based instead on [7, 14]. We use `bp_support_sada` because it turned out to be the fastest in preliminary experiments. We use `cst_sct3`, based on [13], rather than the other compressed suffix tree representation provided by SDSL, because its topology takes less space ($3|T|$ bits, rather than $4|T|$ bits of e.g. `cst_sada`), at the cost of slower `parent` operations. One could use more efficient algorithms to build the indexes, for example [2] for the suffix tree topology; we do not try to optimize construction in this paper.

5 Experimental results

5.1 Artificial strings

We use a number of artificial strings to evaluate the effect of the optimizations in Section 3. We resort to artificial strings because our optimizations are not designed for a particular class of inputs, and because the magnitude of their speedups might be affected by the similarity

between S and T, by the repetitiveness of T, and by the alphabet size, over which we need control. In Section 5.2 we study performance on a dataset that approximates a real use case in phylogeny reconstruction by average matching statistics. Since our artificial strings do not cover the space of all possible input classes, the speedups we report should not be interpreted as upper bounds.

To measure the effect of the Weiner link optimizations in Section 3.1, we consider four types of (S, T) pairs, corresponding to repetitive or random T, and to a query S that is either similar to T or random (by random we mean a string generated by a source that assigns uniform probability to every character of the alphabet). A repetitive string is generated as follows: we take an initial random string W of length 10^4, and we concatenate it with 10^4 copies of itself, containing each 10 random positions that mismatch with W. We create a string S that is similar to T by introducing mismatches in $5 \cdot 10^3$ random positions of a copy of T, and by taking a prefix of such copy of length $5 \cdot 10^5$, and we experiment with alphabet size 4 and 20. We measure the total time to compute `ms` and `runs` on a machine with one Intel Core i7-6600U processor with two cores. Its L3 cache is 4MB, thus T does not fit in cache. Throughout the section we call *speedup* the ratio between the time to build the `ms` and `runs` bitvectors with a baseline implementation that is clear from the context, and the time to build the same vectors with an implementation with specific optimizations turned on.

The `doubleRank` optimization tends to give median speedups smaller than 1.05 with respect to an implementation that uses two rank operations per Weiner link, both for small and for large alphabet, and it rarely degrades performance. Using `doubleRankAndFail` gives further speedups between 1.01 and 1.02 with respect to `doubleRank`, mostly when S is not similar to T; when S is similar to T instead, performance tends to degrade by similar amounts. This is expected, since this optimization targets cases in which Weiner link calls are unsuccessful, and this is more likely to happen when S is not similar to T. If a Weiner link is successful, `doubleRankAndFail` is actually introducing an overhead. The early failure strategy might also be useful for representations in which the wavelet tree is not full, e.g. when it has a Huffman shape: we leave this study to the full version of the paper.

As expected, `lazyWeinerLink` is most effective when S is similar to T, since it optimizes successful Weiner links. When the alphabet is small, we observe median speedups of 3.2 for repetitive T and of 2.4 for random T (Figure 1). For large alphabets the median speedups become approximately 2.4 and 1.6, respectively. When S is not similar to T, the speedups are approximately equal to 1.1 and never smaller than one.

Conversely, it turns out that exploiting maximal repeat information in Weiner links is beneficial when S is not similar to T, and especially so when the alphabet is large. `maxrepWeinerLink` provides median speedups of 2.1 and 2.6 when T is repetitive (depending on alphabet size), and of 1.7 and 2.2 when T is random (Figure 2), compared to an implementation with just `doubleRankAndFail`. If S is similar to T, `maxrepWeinerLink` tends to degrade performance, and the median speedup can drop down to approximately 0.8. With respect to `maxrepWeinerLink`, `rankAndCheck` provides median speedups of approximately 1.01 when T is repetitive, and of 1.04 and 1.07 when T is random, depending on alphabet size, but only when S is not similar to T. The fact that maximal repeats are not useful when S is similar to T might be explained by the fact that the overhead of checking whether a node is a maximal repeat or not is compensated by a faster fail when a Weiner link is not successful, but it is not compensated in case of a successful Weiner link. One might try to reduce the cost of accessing the `marked` bitvector by interleaving it with the bitvector of the root of the wavelet tree: we leave such extension to the full version of the paper.

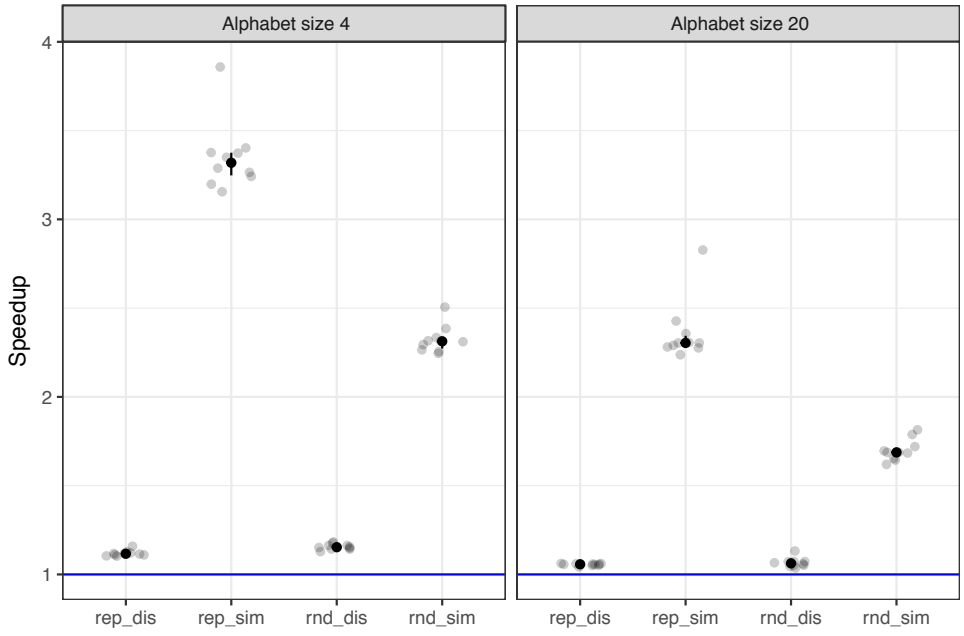

Figure 1 Speedup of `lazyWeinerLink` with respect to an implementation with no optimization, for different input types (horizontal axis). Gray points are single measurements. Black points are medians. Black bars indicate the first and third quartiles of ten experiments.

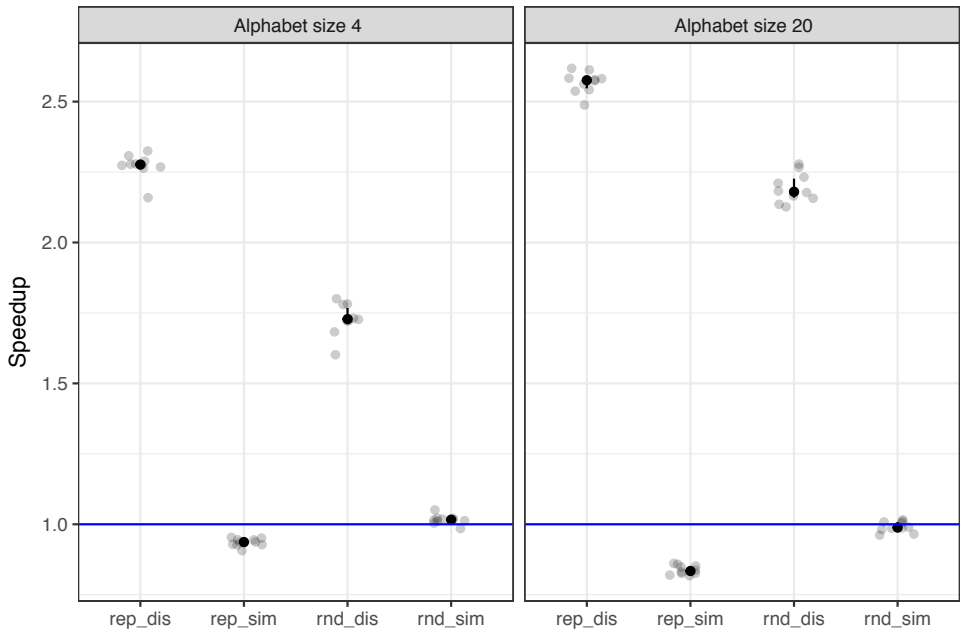

Figure 2 Speedup of `doubleRankAndFail` and `maxrepWeinerLink`, with respect to an implementation with just `doubleRankAndFail`. The plots follow the conventions of Figure 1.

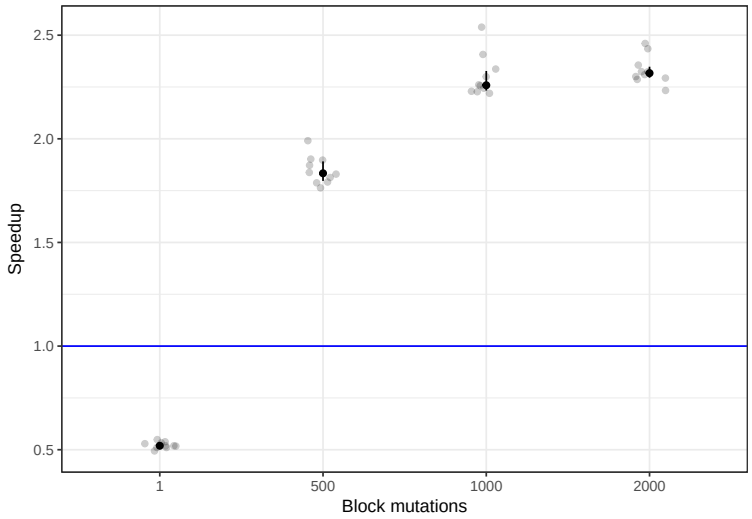

Figure 3 Speedup of `parentShortcut` with respect to a baseline with all optimizations turned off (vertical axis), for different settings of k (horizontal axis). The plot follows the conventions of Figure 1.

To measure the effect of the `parentShortcut` optimization of Section 3.2, we create pairs (S, T) in which the baseline algorithm is forced to issue long sequences of `parent` operations. Specifically, we build a repetitive T as described above, starting from a random string W of length 200 on alphabet size 6, and concatenating to it $5000 - 1$ copies of itself, containing each k random positions that mismatch with W. The resulting T takes 10MB and does not fit in cache. We build T repetitive to have nodes with large tree depth in ST_T and $\mathsf{ST}_{\overline{T}}$. We build S by concatenating labels of nodes of ST_T with large tree depth (at least 10) but not too large string length (at most 170), separated by a character that does not occur in T, to enforce enough parent operations per successful Weiner link. We experiment with values of k ranging from one to 200, and at low values of k a large fraction of parent sequences has indeed large length. Using `parentShortcut` gives overall speedups between 1.8 and 2.3, depending on the value of k, with respect to an implementation with no optimization (Figure 3).

5.2 Biological strings

We use a dataset of 24 eukaryotic species with relatively large genomes that has been used for whole-genome phylogeny reconstruction by average matching statistics before [5]. The dataset contains both pairs of very similar and of very different genomes[2]. We experiment with both genomes and proteomes to study the effect of alphabet size (4 and 20, respectively).

[2] We download the latest assemblies from NCBI, and we concatenate all sequences that belong to the genome or proteome of the same species using a separator character not in the alphabet, replacing runs of undetermined characters with a single occurrence of the separator. We run our experiments on a machine with 256GB of RAM and with one Intel Xeon E5-2680v3 processor, which has two sockets and 12 cores per socket. We pin our sequential program to a single socket using `numactl -N0 -m0`. The L3 cache of the processor is 30MB, and its L2 cache is 256KB, thus the indexes for genomes and bacteria do not fit in cache, the indexes for proteomes do not fit in the L2 cache, and the indexes of some proteomes can fit in the L3 cache. We measure wall clock time with standard C++ methods, and we measure peak memory by reading the maximum resident set size reported by `/usr/bin/time`.

The resulting genome files range from approximately 87 million to 5.8 billion characters, and proteome files range from approximately 4 million to 74 million characters. For reasons of time, we compute the matching statistics just between the genome of a fixed species, chosen arbitrarily (*Oryzias latipes*), and every other genome, and just between the proteome of a fixed species, chosen arbitrarily (*Homo sapiens*), and every other proteome. We call this setup *Experiment 1*. We also study the performance of our algorithms when T is repetitive and S is similar to T. Specifically, we download the genomes and proteomes of all the 8658 bacteria currently in NCBI: we build a query that consists of the concatenation of the genomes (respectively, proteomes) of 839 bacteria randomly sampled from the set, and a text that consists of the concatenation of the genomes (respectively, proteomes) of all remaining bacteria. The resulting text files contain approximately 30 and 8.5 billion characters, respectively, and the resulting query files contain approximately 3 billion and 980 million characters, respectively. We call this setup *Experiment 2*. As customary, we measure peak memory and running time just for computing matching statistics given the indexes. We define speedup as in Section 5.1, with respect to a version of our code with no optimization.

It turns out that `lazyWeinerLink` is the most effective optimization in both Experiment 1 and Experiment 2, providing approximately a 1.25 speedup for both genomes and proteomes in the first case, and a 1.75 speedup for genomes and a 1.5 speedup for proteomes in the second case. Using `doubleRankAndFail` in addition to `lazyWeinerLink` yields minor improvements and the fastest implementation. Speedups are greater when S is similar to T, with the largest speedups observed in Experiment 2 and for the pair of proteomes *Homo sapiens* and *Pan troglodytes* (see Figure 4).

We compare our implementations to the one described in [15], which builds essentially the same compressed suffix tree as SDSL's `cst_sct3`, with `csa_wt` representing the compressed suffix array, but which includes a compressed LCP array[3] storing all values that are at most 254 in one byte, and longer values in $\log |T|$ bits [10]. As expected, our implementation is more space-efficient, taking from approximately half to one-fifth of the space of the competitor in both genomes and proteomes (Figure 5). Not surprisingly, however, our implementation is also slower than the competitor, taking e.g. approximately twice its time for proteomes (Figure 5). Recall that the fact that the algorithm in [15] performs just one pass over S might contribute to this slowdown. However, for approximately half of the pairs of genomes in Experiment 1, and for the pair of genomes in Experiment 2, our implementation is faster than the competitor, with speedups up to 1.4.

The eight pairs of sibling leaves[4] in the phylogenetic tree of [5, Figure 1] correspond to similar species with short divergence times, and we observed before that our implementation gives larger speedups for similar strings. We call *Experiment 3* the comparison of our implementation with `doubleRankAndFail` and `lazyWeinerLink` enabled, to the one in [15], limited to such pairs. It turns out that our implementation is faster than or as fast as the competitor for all but one such pairs of genomes, with speedups up to two, while still taking between one fourth and one fifth of the competitor's space (Figure 5). For pairs of proteomes, however, our implementation remains slower.

[3] The index contains also (unused) samples of the suffix array of T and of $\overline{T}$, but the sampling rate (10^4) makes their space negligible in practice. The code of [15] does not work with long strings, thus we take prefixes of length 800 million of the strings it cannot handle. We use a version of the two implementations in which they do not write any output. For each (S, T) pair, we compare the fastest of our implementations to the competitor.

[4] *Arabidopsis lyrata-Arabidopsis thaliana*; *Zea mays-Oryza sativa*; *Rattus norvegicus-Mus musculus*; *Gallus gallus-Taeniopygia guttata*; *Brugia malayi-Caenorhabditis elegans*; *Drosophila melanogaster-Anopheles gambiae*.

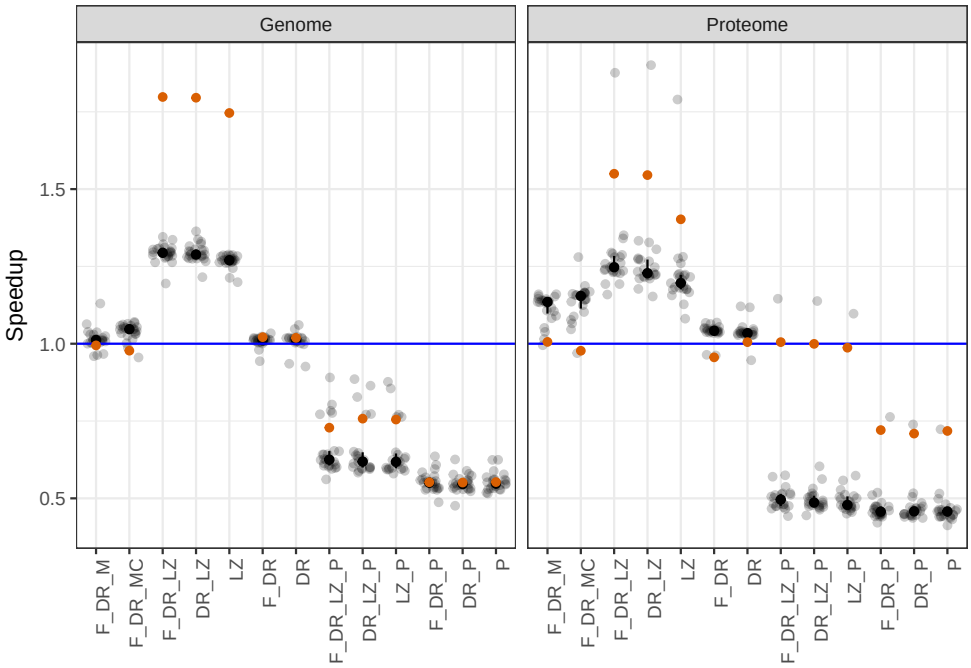

Figure 4 Effect of our optimizations (horizontal axis) on Experiment 1 (gray circles) and on Experiment 2 (red circles). DR: `doubleRank`, F_DR: `doubleRankAndFail`, M: `maxrepWeinerLink`, MC: `rankAndCheck`, LZ: `lazyWeinerLink`, P: `parentShortcut`. The largest speedups in the plot on the right correspond to pair *Homo sapiens* and *Pan troglodytes*.

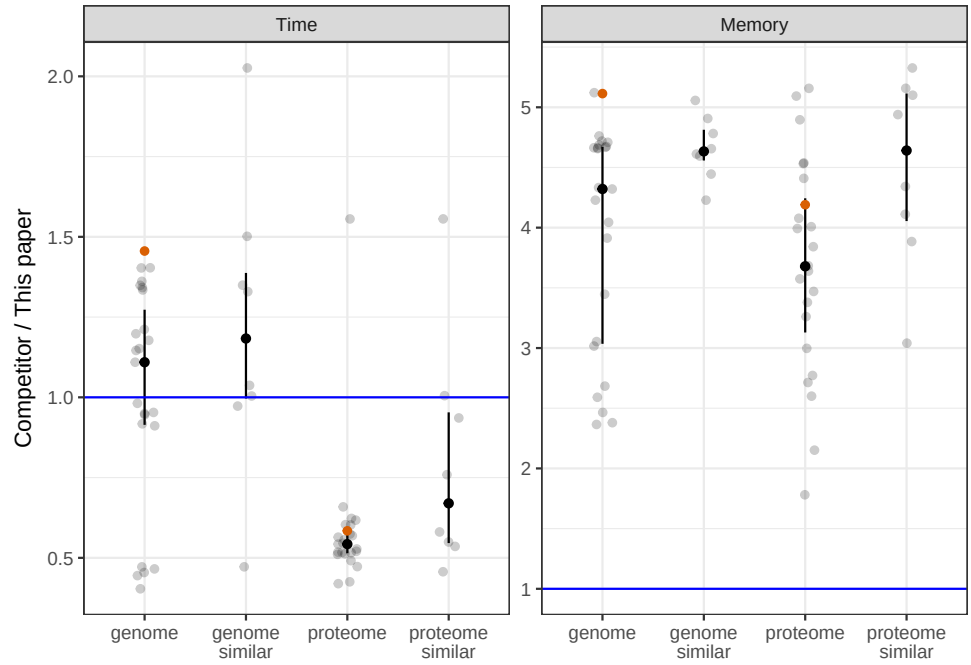

Figure 5 The fastest of our implementations, compared to the implementation in [15]. "Genome", "Proteome": Experiment 1 (gray circles) and Experiment 2 (red circles). "Genome similar", "Proteome similar": Experiment 3. The plot follows the conventions of Figure 1.

References

1 Alberto Apostolico, Concettina Guerra, Gad M. Landau, and Cinzia Pizzi. Sequence similarity measures based on bounded Hamming distance. *Theoretical Computer Science*, 638:76–90, 2016.

2 Uwe Baier, Timo Beller, and Enno Ohlebusch. Space-efficient parallel construction of succinct representations of suffix tree topologies. *Journal of Experimental Algorithmics (JEA)*, 22:1–1, 2017.

3 Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *International Symposium on String Processing and Information Retrieval*, pages 179–190. Springer, 2014.

4 Djamal Belazzougui and Fabio Cunial. A framework for space-efficient string kernels. *Algorithmica*, 79(3):857–883, 2017.

5 Eyal Cohen and Benny Chor. Detecting phylogenetic signals in eukaryotic whole genome sequences. *Journal of Computational Biology*, 19(8):945–956, 2012.

6 Martin Farach, Michiel Noordewier, Serap Savari, Larry Shepp, Abraham Wyner, and Jacob Ziv. On the entropy of DNA: algorithms and measurements based on memory and rapid convergence. In *Proceedings of the sixth annual ACM-SIAM symposium on discrete algorithms*, pages 48–57, 1995.

7 Richard F Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

8 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

9 R. González, G. Navarro, and H. Ferrada. Locally compressed suffix arrays. *ACM Journal of Experimental Algorithmics*, 19(1):article 1, 2014.

10 Stefan Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 29(13):1149–71, 1999.

11 Chris-Andre Leimeister and Burkhard Morgenstern. Kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 2014.

12 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):16, 2014.

13 Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *International Symposium on String Processing and Information Retrieval*, pages 322–333. Springer, 2010.

14 Enno Ohlebusch and Simon Gog. A compressed enhanced suffix array supporting fast string matching. In *International Symposium on String Processing and Information Retrieval*, pages 51–62. Springer, 2009.

15 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *SPIRE*, pages 347–358, 2010.

16 Daisuke Okanohara and Kunihiko Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *International Symposium on String Processing and Information Retrieval*, pages 90–101. Springer, 2009.

17 Nicolas Philippe, Mikaël Salson, Thérèse Commes, and Eric Rivals. CRAC: an integrated approach to the analysis of RNA-seq reads. *Genome Biology*, 14(3):R30, 2013.

18 Cinzia Pizzi. Missmax: alignment-free sequence comparison with mismatches through filtering and heuristics. *Algorithms for Molecular Biology*, 11(1):6, 2016.

19 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

20 Kunihiko Sadakane. The ultimate balanced parentheses. Technical report, Kyushu University, 2008.

21 Kunihiko Sadakane and Gonzalo Navarro. Fully-functional succinct trees. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 134–149. SIAM, 2010.

22 Choon Hui Teo and S.V.N. Vishwanathan. Fast and space efficient string kernels using suffix arrays. In *Proceedings of the 23rd international conference on Machine learning*, pages 929–936. ACM, 2006.

23 Sharma V Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016.

24 Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Ambujam Krishnan, and Srinivas Aluru. A greedy alignment-free distance estimator for phylogenetic inference. *BMC bioinformatics*, 18(8):238, 2017.

25 Igor Ulitsky, David Burstein, Tamir Tuller, and Benny Chor. The average common substring approach to phylogenomic reconstruction. *Journal of Computational Biology*, 13(2):336–350, 2006.

26 Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973*, pages 1–11. IEEE, 1973.