Computing all-vs-all MEMs in run-length encoded collections of HiFi reads

Díaz-Domínguez, Diego

Springer
2022

Díaz-Domínguez , D , Puglisi , S & Salmela , L 2022 , Computing all-vs-all MEMs in run-length encoded collections of HiFi reads . in String Processing and Information Retrieval. SPIRE 2022 . Lecture Notes in Computer Science , vol. 13617 , Springer , Cham , pp. 198-213 , International Symposium on String Processing and Information Retrieval , Concepción , Chile , 08/11/2022 . https://doi.org/10.48550/arXiv.2208.14787 , https://doi.org/10.1007/978-3-031-206

http://hdl.handle.net/10138/351830
https://doi.org/10.48550/arXiv.2208.14787

Computing all-vs-all MEMs in run-length-encoded collections of HiFi reads *

Diego Díaz-Domínguez, Simon J. Puglisi, and Leena Salmela

Department of Computer Science, University of Helsinki, Finland
{diego.diaz,simon.puglisi,leena.salmela}@helsinki.fi

Abstract. We describe an algorithm to find maximal exact matches (MEMs) among HiFi reads with homopolymer errors. The main novelty in our work is that we resort to run-length compression to help deal with errors. Our method receives as input a run-length-encoded string collection containing the HiFi reads along with their reverse complements. Subsequently, it splits the encoding into two arrays, one storing the sequence of symbols for equal-symbol runs and another storing the run lengths. The purpose of the split is to get the BWT of the run symbols and reorder their lengths accordingly. We show that this special BWT, as it encodes the HiFi reads and their reverse complements, supports bi-directional queries for the HiFi reads. Then, we propose a variation of the MEM algorithm of Belazzougui et al. (2013) that exploits the run-length encoding and the implicit bi-directional property of our BWT to compute approximate MEMs. Concretely, if the algorithm finds that two substrings, $a_1 \dots a_p$ and $b_1 \dots b_p$, have a MEM, then it reports the MEM only if their corresponding length sequences, $\ell^a_1 \dots \ell^a_p$ and $\ell^b_1 \dots \ell^b_p$, do not differ beyond an input threshold. We use a simple metric to calculate the similarity of the length sequences that we call the *run-length excess*. Our technique facilitates the detection of MEMs with homopolymer errors as it does not require dynamic programming to find approximate matches where the only edits are the lengths of the equal-symbol runs. Finally, we present a method that relies on a geometric data structure to report the text occurrences of the MEMs detected by our algorithm.

Keywords: Genomics · Text indexing · Compact data structures.

1 Introduction

HiFi reads are a new type of DNA sequencing data developed by PacBio [30]. They are long overlapping strings with error rates (mismatches) comparable to those of Illumina data. They have become popular in recent years as their features improve the accuracy of biological analyses [21]. Still, mapping a collection of HiFi reads against a reference genome or computing suffix-prefix overlaps among them for *de novo* assembly remain important challenges as these tasks require approximate alignments of millions of long strings. Popular tools

* Supported by Academy of Finland Grants 323233 and 339070

that address these problems use seed-and-extend algorithms with minimizers as seeds [19] for the alignments. This technique is a cheap solution that makes the processing of HiFi reads feasible.

An alternative approach is to use *maximal exact matches* (MEMs) as seeds. A MEM is a match $S[a, b] = S'[a', b']$ between two strings S and S' that cannot be extended to the left or to the right without introducing mismatches or reaching an end in one of the sequences. MEMs are preferable over minimizers because they can capture long exact matches between the reads, thus reducing the computational costs of extending the alignments with dynamic programming. However, they are expensive to find in big collections.

A classical solution to detect MEMs among strings of a large collection is to concatenate the strings in one sequence S (separated by sentinel symbols), construct the suffix tree of S, and then traverse its topology to find MEMs in linear time [13]. Still, producing the suffix tree of a massive collection, although linear in time and space, is expensive for practical purposes. Common approaches to deal with the space overhead are sparse suffix trees [16,28], hash tables with k-mers [17,11], and Bloom filters [20].

Another way to deal with the space overhead is to find MEMs on top of a compact suffix tree [27,23]. For instance, Ohlebusch et al. [24] described a method that computes MEMs between two strings via matching statistics [6]. Their technique requires only one of the strings to be indexed using a compact suffix tree while the other is kept in plain format. Other more recent methods [26,3,25] follow an approach similar to that of Ohlebusch et al., but they use the r-index [9] instead of the compact suffix tree.

The problem with the algorithms that rely on matching statistics is that they consider input collections with two strings (one indexed and the other in plain format). It is not clear how to generalize these techniques to compute all-vs-all MEMs in a collection with an arbitrary number of sequences. A simple solution would be to implement classical MEM algorithms on top of the compact suffix tree. Still, producing a full compact suffix tree is expensive for genomic applications as it requires producing a sampled version of the suffix array [22], the Burrows–Wheeler transform [4], and the longest common prefix array [15]. Although it is possible to obtain these composite data structures in linear time and space, in practice, they might require an amount of working memory that is several times the input size. In this regard, Belazzougui et al. [2] proposed a MEM algorithm that only uses the bi-directional BWT of the text, although their idea reports the sequences for the MEM, not their occurrences in the text.

Besides the input size, there is another relevant issue when computing MEMs in HiFi data: homopolymer errors. Concretely, if a segment of the DNA being sequenced has an equal-symbol run of length ℓ, then the PacBio sequencer might spell imprecise copies of that run in the reads that overlap the segment. These copies have a correct[1] DNA symbol, but the value ℓ might be incorrect. In general, homopolymer errors shorten the alignment seeds, which means that the

[1] The symbol correctly represents the nucleotide that was read from the DNA molecule.

pattern matching algorithm will spend more time performing dynamic programming operations to extend the alignments. In this work, we study the problem of finding MEMs in HiFi reads efficiently. Our strategy is to use run-length encoding to remove the homopolymer errors, and then try to filter out the matches between different sequences that, by chance, were compressed to the same run-length encoded string.

Our contribution. We present a set of techniques to compute all-vs-all MEMs in a collection of HiFi reads of n symbols. We build on the MEM algorithm of Belazzougui et al. [2] that uses the bi-directional BWT, a versatile succinct text representation that uses $2n \log \sigma + o(2n \log \sigma)$ bits of space. Strings in a DNA collection have two complementary sequences that we need to consider for the matches, meaning that we need to create a copy of the input with the complementary strings and merge all in one collection $\mathcal{R}$. We describe a framework that exploits the properties of these DNA complementary sequences to produce an implicit bi-directional BWT for $\mathcal{R}$ without increasing the input size by a factor of 4x. In addition, we define parameters to detect MEMs in a run-length-encoded representation of $\mathcal{R}$. Concretely, we propose the concept of run-length excess, which we use to differentiate homopolymer errors from sporadic matches generated by the run-length compression. Finally, we describe our variation of the algorithm of Belazzougui et al. [2] for computing MEMs using our implicit bi-directional BWT constructed on a run-length-encoded version of $\mathcal{R}$, denoted $\mathcal{R}^h$. Let S be a sequence of length $d = |S|$ that has x occurrences in $\mathcal{R}^h$, with $l \leq x$ of them having MEMs with other positions of $\mathcal{R}^h$. Once our algorithm detects S, it can report its MEMs in $O(\sigma^2 \log \sigma + x^2 d)$ time, where σ is the alphabet of $\mathcal{R}^h$. We also propose an alternative solution that uses a geometric data structure, and report the MEMs of S in $O((x + \sigma)(1 + \log n_h / \log \log n_h) + l^2 d)$ time, where n_h is the number of symbols in $\mathcal{R}^h$.

2 Preliminaries

Rank and select data structures. Given a sequence $B[1,n]$ of symbols over the alphabet $\Sigma = [1,\sigma]$, the operation $\mathsf{rank}_a(B,i)$, with $i \in [1,n]$ and $a \in \Sigma$, returns the number of times the symbol a occurs in the prefix $B[1,i]$. On the other hand, the operation $\mathsf{select}_a(B,r)$ returns the position of the rth occurrence of a in B. For binary alphabets, B can be represented in $n + o(n)$ bits so that it is possible to solve rank_a and select_a, with $a \in \{0,1\}$, in constant time [14,7].

Wavelet trees. Let $S[1,n]$ be a string of length n over the alphabet $\Sigma = [1,\sigma]$. A wavelet tree [12] is a tree data structure W that encodes S in $n \log \sigma + o(n \log \sigma)$ bits of space and supports several queries in $O(\log \sigma)$ time. Among them, the following are of interest for this work:

- $\mathsf{access}(W,i)$: retrieves the symbol at position $T[i]$

- $\mathsf{rank}_a(W, i)$: number of symbols a in the prefix $T[1, i]$
- $\mathsf{select}_a(W, r)$: position j where the rth symbol a lies in S

The wavelet tree can also answer more elaborated queries efficiently [10]. From them, the following are relevant:

- $\mathsf{rangeList}(W, i, j)$: the list of all triplets (c, r_i^c, r_j^c) such that c is one of the distinct symbols within $S[i, j]$, r_i^c is the rank of c in $S[1, i-1]$, and r_j^c is the rank of c in $S[1, j]$.
- $\mathsf{rangeCount}(W, i, j, l, r)$: number of symbols $y \in S[i, j]$ such that $l \leq y \leq r$.

It is possible to answer rangeList in $O(u \log \frac{\sigma}{u})$ time, where u is the number of distinct symbols in $S[i, j]$, and rangeCount in $O(\log \sigma)$ time.

Suffix arrays and suffix trees. Consider a string $S[1, n-1]$ over alphabet $\Sigma[2, \sigma]$, and the sentinel symbol $\Sigma[1] = \$$, which we insert at $S[n]$. The *suffix array* [22] of S is a permutation $SA[1, n]$ that enumerates the suffixes $S[i, n]$ of S in increasing lexicographic order, $S[SA[i], n] < S[SA[i+1], n]$, for $i \in [1, n-1]$.

The suffix trie [8] is the trie T induced by the suffixes of S. For every $S[i, n]$, there is a path $U = v_1, v_2, \ldots, v_p$ of length $p = n - i + 2$ in T, where v_1 is the root and v_p is a leaf. Each edge (v_j, v_{j+1}) in U is labeled with a symbol in Σ, and concatenating the edge labels from v_1 to v_p produces $S[i, n]$. The child nodes of each internal node v are sorted from left to right according to the ranks of the labels in the edges that connect them to v. Further, when two or more suffixes of S have the same j-prefix, their paths in T share the first $j+1$ nodes.

It is possible to compact T by discarding each unary path $U = v_i, \ldots, v_j$ where every node $v_i, v_{i-1}, \ldots, v_{j-1}$ has exactly one child. The procedure consists of removing the subpath $U' = v_{i+1}, \ldots, v_{j-1}$ and connect v_i with v_j by an edge labeled with the concatenation of the labels in U'. The result is a compact trie of n leaves and less than n internal nodes called the *suffix tree* [29].

The suffix tree can contain other special edges that connect nodes from different parts of the tree, not necessarily a parent with its children. These edges are called suffix and Weiner links. Let us denote $\mathsf{label}(v)$ the string that labels the path starting at the root and ending at v. Two nodes u and v are connected by a suffix link (u, v) if $\mathsf{label}(u) = aW$ and $\mathsf{label}(v) = W$. Similarly, an explicit Weiner link (u, v) labeled a occurs if $\mathsf{label}(u) = W$ and $\mathsf{label}(v) = aW$. A Weiner link is implicit when, for $\mathsf{label}(u) = W$, the sequence aW matches a proper prefix of a node label (i.e., there is no node labeled aW). The suffix and Weiner links along with the suffix tree nodes yield another tree called the suffix link tree.

The Burrows–Wheeler transform. The *Burrows–Wheeler transform* (BWT) [4] is a reversible string transformation that stores in $BWT[i]$ the symbol that precedes the ith suffix of S in lexicographical order, i.e., $BWT[i] = S[SA[i] - 1]$ (assuming $S[0] = S[n] = \$$).

The mechanism to revert the transformation is the so-called LF mapping. Given an input position $BWT[j]$ that maps a symbol $S[i]$, $\mathsf{LF}(j) = j'$ returns the index j' such that $BWT[j'] = S[i-1]$ maps the preceding symbol of $S[i]$. Thus, spelling S reduces to continuously applying LF from $BWT[1]$, the symbol to the left of $T[n] = \$$, until reaching $BWT[j] = \$$.

Implementing LF requires to encode BWT with a data structure that supports rank_a. A standard solution is to use the wavelet tree of Section 2, which enables to answer LF in $O(\log \sigma)$ time. It is also necessary to have an array $C[1, \sigma]$ storing in $C[c]$ the number of symbols in S that are lexicographically smaller than c. This enables the implementation of the inverse function for LF (denoted as LF^{-1}). That is, given the position $BWT[j]$ that maps $S[i]$, $\mathsf{LF}^{-1}(j) = j'$ returns the index j' such that $BWT[j']$ maps $S[i+1]$.

The BWT also allows to count the number of occurrences of a pattern $P[1, m]$ in S in $\mathcal{O}(m \log \sigma)$ time. The method, called $\mathsf{backwardsearch}$, builds on the observation that if the range $SA[s_j, e_j]$ encoding the suffixes of S prefixed by $P[j, m]$ is known, then it is possible to compute the next range $SA[s_{j-1}, e_{j-1}]$ with the suffixes of S prefixed by $P[j-1, m]$. This computation, or *step*, requires two operations: $s_{j-1} = C[P[j-1]] + \mathsf{rank}_{P[j-1]}(BWT, s_j - 1) + 1$ and $e_{j-1} = C[P[j-1]] + \mathsf{rank}_{P[j-1]}(BWT, e_j)$. Thus, after m steps of $O(\log \sigma)$ time each, $\mathsf{backwardsearch}$ will find the range $SA[s_1, e_1]$ encoding the suffixes of S prefixed by $P[1, m]$ (provided P exists as substring in S).

Bi-directional BWT. The bi-directional BWT [18] of a string $S[1, n]$ is a data structure that maintains the BWT of S and the BWT of the reverse of S (denoted here as $\bar{S}$). Belazzougui et al. [2] demonstrated that it is possible to use this representation to visit the internal nodes in the suffix tree T of S in $O(n \log \sigma)$ time.

The work of Belazzougui et al. exploits the fact that the suffixes of S prefixed by the label of an internal node v in T are stored in a consecutive range $SA[s_v, e_v]$, and that $BWT[s_v, e_v]$ encodes the labels for the Weiner links of v.

Let SA_S and BWT_S be the suffix array and BWT for S (respectively). Equivalently, let $SA_{\bar{S}}$ and $BWT_{\bar{S}}$ be the suffix array and BWT for $\bar{S}$. For any sequence X, Belazzougui et al. maintain two pairs: (s_X, e_X) and $(s_{\bar{X}}, e_{\bar{X}})$, where $SA_S[s_X, e_X]$ stores the suffixes of S prefixed by X and $SA_{\bar{S}}[s_{\bar{X}}, e_{\bar{X}}]$ stores the suffixes of $\bar{S}$ prefixed by $\bar{X}$. They also define a set of primitives for the encoding (s_X, e_X), $(s_{\bar{X}}, e_{\bar{X}})$ of X:

- $\mathsf{isLeftMaximal}(X)$: 1 if $BWT_S[s_X, e_X]$ contains more than one distinct symbol, and 0 otherwise.
- $\mathsf{isRightMaximal}(X)$: 1 if $BWT_{\bar{S}}[s_{\bar{X}}, s_{\bar{X}}]$ contains more than one distinct symbol, and 0 otherwise.
- $\mathsf{enumerateLeft}(X)$: list of distinct symbols in $BWT_S[s_X, e_X]$.
- $\mathsf{enumerateRight}(X)$: list of distinct symbols in $BWT_{\bar{S}}[s_{\bar{X}}, e_{\bar{X}}]$
- $\mathsf{extendLeft}(X, c)$: list $(i, j), (i', j')$ where $SA_S[i, j]$ is the range for cX and $SA_{\bar{S}}[i', j']$ is the range for $\bar{X}c$
- $\mathsf{extendRight}(X, c)$: list $(i, j), (i', j')$ where $SA_S[i, j]$ is the range for Xc and $SA_{\bar{S}}[i', j']$ is the range for $c\bar{X}$

The key aspect of the bi-directional BWT is that, every time it performs a left or a right extension in (s_X, e_X) (respectively, $(s_{\bar{X}}, e_{\bar{X}})$), it also synchronizes $(s_{\bar{X}}, e_{\bar{X}})$ (respectively, (s_X, e_X)). By encoding BWT_S and $BWT_{\bar{S}}$ as wavelet trees (Section 2), it is possible to perform extendLeft and extendRight in $O(\log \sigma)$ time using a backward search step (Section 2), and then synchronizing the other range with rangeCount. The functions enumerateLeft and enumerateRight take $O(u \log \frac{\sigma}{u})$ time as they are equivalent to rangeList. Finally, both isLeftMaximal and isRightMaximal run in $O(\log \sigma)$ time.

Belazzougui et al. use the primitives above to traverse the suffix link tree and thus visiting the internal nodes of T in $O(n \log \sigma)$ time.

3 Our contribution

3.1 Definitions

We consider the set $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ to be the *DNA alphabet*. For practical reasons, we compact it to the set $\Sigma = [2, 5]$, and regard $\Sigma[1] = \$$ as a *sentinel* that is lexicographically smaller than any other symbol. Given a string R in Σ^*, we define an *homopolymer* as an equal-symbol run $R[i, j] = (c, \ell)$ of maximal length storing $\ell = j - i + 1 > 1$ consecutive copies of a symbol c. Maximal length means that $i = 1$ or $R[i-1] \neq c$, and $j = |R|$ or $R[j+1] \neq c$.

We regard the DNA *complement* as a permutation $\pi[1, \sigma]$ that reorders the symbols in Σ, exchanging 2 (A) with 5 (T) and 3 (C) with 4 (G). The permutation does not modify 1 (\$) as it does not represent a nucleotide (i.e., $\pi(1) = 1$). The *reverse complement* of R, denoted $\hat{R}$, is the string formed by reversing R and replacing every symbol $R[i]$ by its complement $\pi(R[i])$. Given a DNA symbol $a \in \Sigma$, let us define the operator $\underline{a} = \pi(a)$ to denote the DNA complement of a.

The input for our algorithm is a collection $\mathcal{X} = \{R_1, R_2, \ldots, R_k\}$ of k HiFi reads over the alphabet Σ. However, we operate over the expanded collection $\mathcal{R} = \{R_1\$, \hat{R}_1\$, \ldots, R_k\$, \hat{R}_k\$\}$ storing the reads of $\mathcal{X}$ along with their reverse complements, where all the strings have a sentinel appended at the end. $\mathcal{R}$ has $2k$ strings, with a total of $n = \Sigma_{i=1}^{k} 2(|R_i| + 1)$ symbols. We refer to every possible sequence over the DNA alphabet that label a MEM in $\mathcal{R}$ as a *MEM sequence*.

3.2 Description of the problem

Before developing our ideas, we formalize our problem as follows.

Definition 1. *Let $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ be a string collection of k strings and n total symbols. The problem of all-vs-all MEMs consists in reporting every possible pair $(S_x[a, b], S_y[a', b'])$ such that $S_x, S_y \in \mathcal{S}$, $S_x \neq S_y$, and $S_x[a, b] = S_y[a', b']$ is a MEM of length $b - a + 1 = b' - a' + 1 \geq \tau$, where τ is a parameter.*

HiFi data is usually strand unspecific, meaning that, for any two reads $R_a, R_b \in \mathcal{X}$, there are four possible combinations in which we can have MEMs: (R_a, R_b), $(\hat{R}_a, R_b)$, $(R_a, \hat{R}_b)$, $(\hat{R}_a, \hat{R}_b)$. We can access all such combinations in $\mathcal{R}$, but not in $\mathcal{X}$. Hence, our algorithmic framework solves the problem of Definition 1 using $\mathcal{R}$ as input.

4 Bi-directional BWT and DNA reverse complements

In this section, we explain how to exploit the properties of the DNA reverse complements to implement an *implicit* bi-directional BWT for $\mathcal{R}$ that does not require the BWT of the reverse sequences of $\mathcal{R}$ (see Section 2). We assume the BWT of $\mathcal{R}$ is the BCR BWT [1], a variation for string collections. This decision is for technical convenience, and does not affect the output of our framework. We begin by describing the key property in our implicit bi-directional representation:

Lemma 1. *Let X be a string over alphabet Σ that appears as a substring in $\mathcal{R}$. Additionally, let the pairs (s_X, e_X) and $(s_{\hat{X}}, e_{\hat{X}})$ be the ranges in SA of $\mathcal{R}$ storing all suffixes prefixed by X and $\hat{X}$, respectively. It holds that the sorted sequence of DNA complement symbols in $BWT[s_X, e_X]$ matches the right-context symbols of the occurrences of $\hat{X}$ when sorted in lexicographical order. This relationship applies symmetrically to $BWT[s_{\hat{X}}, e_{\hat{X}}]$ and the sorted occurrences of X.*

Proof. Assume the symbol $a \in \Sigma$ appears to the left of p occurrences of Xb in $\mathcal{R}$. We know that for each occurrence of aXb in $\mathcal{R}$ there will be also an occurrence of $\underline{b}\hat{X}\underline{a}$ as we enforce that property by including the DNA reverse complements of the original reads (collection $\mathcal{X}$ of Section 3.1). As a result, $BWT[s_{Xb}, e_{Xb}]$ will contain p copies of a and $BWT[s_{\hat{X}\underline{a}}, e_{\hat{X}\underline{a}}]$ will contain p copies of $\underline{b}$.

We will use Lemma 1 to implement the functions enumerateRight, extendRight and isRightMaximal (Section 2) on top of the BWT of the text. Unlike the technique of Belazzougui et al., we synchronize the pairs $(s_X, e_X), (s_{\hat{X}}, e_{\hat{X}})$. Another difference is that both pairs $(s_X, e_X), (s_{\hat{X}}, e_{\hat{X}})$ map to the suffix array of the text. In the original version, the second pair maps to the suffix array of the reverse text. To implement the functions above, we need to update both pairs every time we perform extendLeft and extendRight.

Belazzougui et al. implement extendLeft(X, c) by performing a backward search step over $BWT[s_X, e_X]$ using the symbol c. The result of this operation is the suffix array range for cX. To modify $(s_{\hat{X}}, e_{\hat{X}})$ so it maps to the suffix array range for $\hat{X}\underline{c}$, we sum the frequencies of the distinct symbols within $BWT[s_X, e_X]$ whose DNA reverse complements are lexicographically smaller than $\underline{c}$. This operation comes directly from Lemma 1. Assume the sum is y and that the frequency of c in $BWT[s_X, e_X]$ is z, then we compute $s_{\hat{X}\underline{c}} = s_{\hat{X}} + y$ and $e_{\hat{X}\underline{c}} = s_{\hat{X}} + y + z$. We use a special form of rangeCount to get the value for y. If $c < \underline{c}$, then we will use $y =$ rangeCount$(BWT, s_X, e_X, c+1, \sigma)$. In the other case, $c > \underline{c}$, we use rangeCount$(BWT, s_X, e_X, 1, c-1)$. The rationale for computing rangeCount comes from the relationship between complementary nucleotides in the permutation π of Section 3.1. The operation extendRight(X, c) is analogous; we perform the backwardsearch step over $BWT[s_{\hat{X}}, e_{\hat{X}}]$ using $\underline{c}$ as input, and then we count the number of symbols that are lexicographically smaller than c.

The functions enumerateRight(X) and isRightMaximal(X) are implemented with minor changes. The only caveat is that, when we use enumerateRight, we need to spell the DNA reverse complements of the symbols returned by rangeList.

Corollary 1. *Given a collection $\mathcal{X}$ of DNA sequences and its expanded version $\mathcal{R}$ that contains the strings of $\mathcal{X}$ along their reverse complement sequences, we can construct an implicit bi-directional BWT index that does not require the BWT of the reverse of $\mathcal{R}$ and that answers the queries* enumerateRight, extendRight *and* isRightMaximal *in $O(u \log \frac{\sigma}{u})$ and $O(\log \sigma)$ time, respectively, where u is the number of distinct symbols within the input range for* extendRight.

Observe the BWT for $\mathcal{R}$ is implicitly bi-directional as the DNA reverse complements are just the reverse strings with their symbols permuted according to π (see Definitions). However, in the case of $\mathcal{R}$, both BWTs are merged in a single representation. Producing a standard bi-directional BWT would increase the size of $\mathcal{X}$ by a factor of 4. In real applications where the data is a multiset of DNA sequencing reads, we have to transform $\mathcal{X}$ into $\mathcal{R}$ regardless if we construct a bi-directional BWT as the reads are strand-unspecific (see Section 3.2).

Contraction operations in the implicit bi-directional BWT. Given a range $SA[i,j]$ of suffixes prefixed by a string X, and a parameter $w \leq |X|$, a *contraction* operation returns the range $i' \geq i, j \leq j'$ in SA storing the suffixes of the text prefixed by $X[1,w]$. It is possible to solve this query efficiently with either the wavelet tree of the LCP or with a compact data structure that encodes the suffix tree's topology. The problem with those solutions is that we have to deal with the overhead of constructing and storing those representations. We describe how to use our implicit bi-directional BWT to visit the ancestors of a node v in the suffix tree in $O(|\mathsf{label}(v)| \log \sigma)$ time to solve contraction operations. This idea is slower than using the LCP or the suffix tree's topology, but it does not require extra space, and it is faster than the quadratic cost of using a regular BWT. Our technique is a byproduct of our framework, and it is of independent interest. The inputs for the ancestors' traversal are the range $SA[s_v, e_v]$ for v, and its string depth $d = |\mathsf{label}(v)|$. The procedure is as follows: starting from $BWT[s_v]$, we perform d LF^{-1} operations to spell $\mathsf{label}(v)$. Simultaneously as we spell the sequence, we also perform backward search steps using the DNA complement of the symbols we obtain with LF^{-1}. We use Lemma 1 to keep the ranges of the backward search steps synchronised with the ranges for the distinct prefixes of $\mathsf{label}(v)$. Recall that backwardsearch consumes the input from right to left. In our case, this input is a sequence W that matches the DNA reverse complement of $\mathsf{label}(v)$. Thus, by Lemma 1, we know that visiting the SA ranges for the suffixes of W is equivalent to visit the SA ranges for the prefixes of $\mathsf{label}(v)$. Finally, each time we obtain a new range $SA[i', j']$ with the backward search step, we use isLeftmaximal to check if $BWT[i', j']$ is unary. If that is the case, then we report the synchronized range of $SA[i', j']$ as an ancestor of v. The rationale is that if W is left-maximal, then $\hat{W} = \mathsf{label}(v)[1, |W|]$ is right-maximal too, and hence, its sequence is the label of an ancestor of v in the suffix tree.

4.1 Homopolymer errors and MEM sequences

A MEM algorithm that runs on top of the suffix tree of $\mathcal{R}$ is unlikely to report all the real[2] matches if the input collection is HiFi data. The difficulty is that some of the MEMs are "masked" in the suffix tree. More specifically, suppose we have two nodes v and u, with $\mathsf{label}(v) \neq \mathsf{label}(u)$. It might happen that, by removing or adding copies of symbols in the equal-symbol runs of $\mathsf{label}(u)$, we can produce $\mathsf{label}(v)$. If those edits are small enough for the PacBio machine to produce them during the sequencing process, then it is plausible to assume that $\mathsf{label}(u)$ is an homopolymer error of $\mathsf{label}(v)$. This situation becomes even more likely if $\mathsf{label}(u)$ is long and its frequency is low in the collection.

Looking for all the possible suffix tree nodes that only have small differences in the length of homopolymer runs similar to v and u could be expensive. A simple workaround is to run-length compress $\mathcal{R}$ and execute the suffix-tree-based MEM algorithm with that as input. Now the problem is that we can report false positive MEMs between different sequences that have the same run-length representation but that are not homopolymer errors. Fortunately, filtering those false positive is not so difficult. Before explaining our idea, we formally define the notion of equivalence between sequences.

Definition 2. *Let A be a string whose run-length encoding is the sequence of pairs $A = (a_1, \ell_1), (a_2, \ell_2), \ldots, (a_p, \ell_p)$, where a_i is the symbol of the ith equal-symbol run, and $\ell_i \geq 1$ is its length. Additionally, let the operator $\mathsf{rlc}(A) = a_1, a_2, \ldots, a_p$ denote the sequence of run heads for A. We say that two strings A and B are equivalent iff $\mathsf{rlc}(A) = \mathsf{rlc}(B)$.*

We use equivalent sequences (Definition 2) to define a filtering parameter to discard false positive MEMs. We call this parameter the *run-length excess*:

Definition 3. *Let A and B be two distinct strings with $\mathsf{rlc}(A) = \mathsf{rlc}(B)$. Additionally, let the pair sequences $A = (x_1, \ell^a_1), (x_2, \ell^a_2), \ldots, (x_p, \ell^a_p)$ and $B = (x_1, \ell^b_1), (x_2, \ell^b_2), \ldots, (x_p, \ell^b_p)$ be the run-length encoding for A and B, respectively. Each $x_i \in \Sigma$ is the ith run head, and $\ell^a_i, \ell^b_i \geq 1$ are the lengths for x_i in A and B, respectively. Now consider the string $E = |\ell^a_1 - \ell^b_1|, \ldots, |\ell^a_n - \ell^b_n|$ storing the absolute differences between the run lengths of A and B. We define the run-length excess as $\mathsf{rlexcess}(A, B) = \max(E[1], E[2], \ldots, E[n])$.*

Intuitively, equivalent sequences that have a high run-length excess are unlikely to have a masked MEM. The reason is because, although the PacBio sequencing process makes mistakes estimating the lengths of the equal-symbol runs, the error in the estimation is unlikely to be high.

Now that we have a framework to detect MEMs in run-length-compressed space, we construct a new collection $\mathcal{R}^h$ of $n_h \leq n$ symbols encoding the same strings of $\mathcal{R}$ but with their homopolymers compacted. Namely, every equal-symbol run $R_u[i, j] = (c, \ell)$ of maximal length $\ell > 1$ in $\mathcal{R}$ is represented with

[2] Those we would obtain in a collection with no homopolymer errors.

a special metasymbol $c^* \notin \Sigma$ in $\mathcal{R}^h$. We store the ℓ values in another list H, sorted as their respective homopolymers occur in $\mathcal{R}$. Each element of Σ has its own metasymbol, including the sentinel. We reorder the alphabet $\Sigma \cup \Sigma^h$ of $\mathcal{R}^h$ to the set $\{\texttt{\$}, \texttt{A}, \texttt{A}^*, \texttt{C}, \texttt{C}^*, \texttt{G}^*, \texttt{G}, \texttt{T}^*, \texttt{T}, \texttt{\$}^*\}$, which we map to its compact version $\Sigma^{hp} = [1, 10]$. This reordering will facilitate the synchronization of ranges when we perform extendLeft or extendRight in our implicit bi-directional BWT.

Recall from Section 4 that, when we call the operation $\mathsf{extendLeft}(X, c)$ (respectively, $\mathsf{extendRight}(X, c)$), we need to perform $\mathsf{rangeCount}(BWT, s_X, e_X)$ to get the number of symbols within $BWT[s_X, e_X]$ whose DNA complements are smaller than c. For this counting operation to serve to synchronize $BWT[s_{\hat{X}}, e_{\hat{X}}]$ in constant time, we need the BWT alphabet to be symmetric. Concretely, the permutation π for the DNA complements has to exchange $\Sigma^{hp}[1]$ with $\Sigma^{hp}[\sigma]$, $\Sigma^{hp}[2]$ with $\Sigma^{hp}[\sigma - 1]$, and so on. This is the reason why the sentinel has a metasymbol too, even though there are no sentinel homopolymers in $\mathcal{R}$. Additionally, we define a function $g : \Sigma^{hp} \to \Sigma$ to map metasymbols back to their nucleotides in Σ. When the input for g is not a metasymbol, g returns the nucleotide itself.

The next step is to run the suffix-tree-based algorithm to solve the all-vs-all MEM problem of Definition 1 (see Section 2) using $\mathcal{R}^h$ as input. However, we add one extra step. For each candidate MEM $(R_a[i, j], R_b[i', j'])$, with $R_a, R_b \in \mathcal{R}^h$, reported by the algorithm, we check if the run-length excess between $R_a[i, j]$ and $R_b[i, j]$ is below some minimum threshold e. If that is not the case, then we discard that pair as a MEM. We can easily compute the run-length excess value using the suffix array of $\mathcal{R}^h$ and the vector H. If the MEM algorithm detects that an internal node v of the suffix tree encodes a list of MEMs, then we use the suffix array of $\mathcal{R}^h$ to access the text positions $\mathsf{label}(v)$. Subsequently, we map those positions to H to get the lengths of the distinct variations of $\mathsf{label}(v)$ on the text, and thus compute excess among them.

4.2 Computing MEMs in compressed space

We now have all the elements to solve Problem 1 in run-length-compressed space using our implicit bi-directional BWT. Our input is the BWT of $\mathcal{R}^h$ (encoded as a wavelet tree BWT), the array H storing the lengths of the homopolymers in the HiFi reads, and the parameters τ and e for, respectively, the minimum MEM length and the maximum run-length excess (see Section 4.1).

We resort to the algorithm of Belazzougui et al. [2] to visit the internal nodes in the suffix tree T of $\mathcal{R}^h$ in $O(n_h \log |\Sigma^{hp}|)$ time, with $n_h = \Sigma_1^{|\mathcal{R}_h|} |R_i|$ (see Section 2). The advantage of their method is that we can use backward search operations over BWT to navigate T without visiting its edge labels (i.e., unary paths in the suffix trie of $\mathcal{R}^h$). Algorithm 1 describes the procedure.

Each internal node v of T with more than one Weiner link (i.e., $BWT[s_v, e_v]$ is not unary) encodes a group of MEMs. This property holds because $\mathsf{label}(v)$ has more than one left-context symbol and more than one right-context symbol in the text. Thus, any possible combination of strings $a{\cdot}\mathsf{label}(v){\cdot}b$ and $y{\cdot}\mathsf{label}(v){\cdot}z$ we can decode from v, with $a, b, y, z \in \Sigma^{hp}$, $a \neq y$, and $b \neq z$, corresponds to a MEM sequence (see Definitions). The sequences we obtain from v can have multiple occurrences in $\mathcal{R}^h$, and we need to report all of them. However, some of them might be false positives. For instance, the pair of text positions conforming a MEM are in the same string, or in strings that are DNA reverse complements of each other. We filter those cases as they are artefacts in our model.

When we visit a node v with more than one Weiner link during the traversal of T, we access its MEM sequences as follows: we use $\mathsf{enumerateRight}$ and $\mathsf{extendRight}$ to compute every range $SA[s_u, e_u]$, with $s_v \leq s_u \leq e_u \leq e_v$, encoding a child u of v. Then, over each $SA[s_u, e_u]$, we perform $\mathsf{enumerateLeft}$ and $\mathsf{extendLeft}$ to compute every range $SA[s^c_u, e^c_u]$ encoding a Weiner link c of u. This procedure yields a set $\mathcal{M} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_p\}$, where p is the number of children of v, and $\mathcal{I}_q$, with $q \in [1, p]$, is the set of ranges in SA for the Weiner links of the qth child of v (from left to right).

The next step is to report the text positions of the MEM sequences encoded by $\mathcal{M}$. For this purpose, we consider the list of pairs $\{(\mathcal{I}_e, \mathcal{I}_g) \mid \mathcal{I}_e, \mathcal{I}_g \in \mathcal{M} \text{ and } \mathcal{I}_e \neq \mathcal{I}_g\}$. Every element $(SA[i, j], SA[i', j']) \in \mathcal{I}_e \times \mathcal{I}_g$ is a pair of ranges such that $SA[i, j]$ stores the suffixes of $\mathcal{R}^h$ prefixed by a label $a{\cdot}\mathsf{label}(v){\cdot}b$ and $SA[i', j']$ stores the suffixes of $\mathcal{R}^h$ prefixed by another label $y{\cdot}\mathsf{label}(v){\cdot}z$. We know that b and z are different as they come from different children of v. However, the symbols a and y might be equal, which means $\mathsf{label}(v)$ is not a MEM sequence when we match $a{\cdot}\mathsf{label}(v){\cdot}b$ and $y{\cdot}\mathsf{label}(v){\cdot}z$. We can find out this information easily: if $SA[i, j]$ and $SA[i', j']$ come from different buckets[3], then $a \neq y$. If that is the case, we have to report the MEMs associated to $(SA[i, j], SA[i', j'])$. For doing so, we first get the string depth $d = |\mathsf{label}(v)|$ of v. Then, we regard $X = \{i, \ldots, j\}$ and $O = \{i', \ldots, j'\}$ as two different sequences of consecutive indexes in SA, and iterate over their Cartesian product $X \times O$. When we access a pair ($SA[x]$, $SA[o]$), with $(x, o) \in X \times O$, we compute the run-length excess e' between $\mathcal{R}^h[SA[x] + 1, SA[x] + d]$ and $\mathcal{R}^h[SA[o] + 1, SA[o] + d]$ as described in Section 4.1, and discard the MEM in $(SA[x], SA[o])$ if $e' \geq e$. We also discard it if $SA[x]$ and $SA[o]$ map the same string or map different strings that are reverse complements between each other. This procedure is described in Algorithm 2.

Theorem 1. *Let $\mathcal{R}^h$ be the run-length encoded collection of HiFi reads, with an alphabet of $\sigma_h = |\Sigma^{hp}|$ symbols. Additionally, let v be an internal node in the suffix tree of $\mathcal{R}^h$ that has more than one Weiner link. The string depth of v is $d = |\mathsf{label}(v)|$ and its associated range $SA[i, j]$ has length $x = j - i + 1$. We can compute all the MEMs encoded by v in $O(\sigma_h^2 \log \sigma_h + x^2 d)$ time.*

Proof. We first compute the ranges for the children of v with the operations $\mathsf{enumerateRight}$ and $\mathsf{extendRight}$. These two operations take $O(\sigma_h \log \sigma_h)$ time. Then, for every child, we compute its Weiner links. The node v has up to σ_h children, each child has up to σ_h Weiner links, and to compute each of these takes $\log \sigma_h$ time via $\mathsf{extendLeft}$, making $O(\sigma_h^2 \log \sigma_h)$ time in total. The number of suffixes of $\mathcal{R}^h$ in $\mathcal{M}$ is x, and the total number of suffix pairs we visit during the scans of the Cartesian products between sets of $\mathcal{M}$ is bound by x^2. Each time

[3] The bth bucket of SA is the range containing all suffixes prefixed by symbol $b \in \Sigma$.

we visit a pair of suffixes, computing the run-length excess between them takes us $O(d)$ time. Thus, the time for reporting the MEMs from v is $O(\sigma_h^2 \log \sigma_h + x^2 d)$. □

4.3 Improving the time complexity for reporting MEMs

We can think of the problem of reporting MEMs from v as two-dimensional sorting. We need the occurrences of $\mathsf{label}(v)$ to be sorted by their left and right contexts at the same time (the dimensions) to report the MEMs from v efficiently. We can implement this idea using a grid $\mathcal{G}$ with dimensions $n_h \times n_h$. We (logically) label the rows of $\mathcal{G}$ with the suffixes of $\mathcal{R}^h$ sorted in lexicographical order, and do the same with the columns. We then store the values of *SA* in the grid cells, with the (row,column) coordinate for each $SA[j]$ being $(j, \mathsf{LF}(j))$. We encode $\mathcal{G}$ with the data structure of Chan et al. [5] that increases the space by $O(n_h \log n_h) + o(n_h \log n_h)$ bits and allows reporting of the *occ* points in the area $[x_1, x_2], [y_1, y_2]$ of $\mathcal{G}$ in $O((occ+1)(1+\log n_h / \log\log n_h))$ time. In exchange, we no longer require *SA*.

The procedure to report MEMs is now as follows. When we reach v during the suffix tree traversal, we perform $\mathsf{extendLeft}$ with each of v's Weiner links. This produces a list $\mathcal{L}$ of up to σ_h non-overlapping ranges in *SA*. We then create another list $\mathcal{Q}$ with the ranges obtained by following v's children. Notice that the ranges of $\mathcal{Q}$ are a partition of the range $[i, j]$ in *SA* for $\mathsf{label}(v)$. For every $[l_1, l_2] \in \mathcal{L}$, we extract the points in $\mathcal{G}$ in the area $[l_1, l_2], [i, j]$, and partition the result into subsets according to $\mathcal{Q}$. The partition is simple as the points can be reported in increasing order of the y coordinates (range $[i, j]$). The idea is to generate a list $\mathcal{I} = \{I_1, I_2, \ldots, I_x\}$ of at most σ_h^2 elements, where each element is a point set for an area $[l_1, l_2], [q_1, q_2] \in \mathcal{L} \times \mathcal{Q}$. Finally, we scan all possible distinct pairs of $\mathcal{I}$ that yield MEMs, processing suffixes as in lines 18-23 of Algorithm 2. Let $I_i, I_j \in \mathcal{I}$ be two point sets, extracted from the areas $[l_1, l_2], [q_1, q_2]$ and $[l'_1, l'_2], [q'_1, q'_2]$ of $\mathcal{G}$, respectively. The points of I_i will have MEMs with the points of I_j if $[l_1, l_2] \neq [l'_1, l'_2]$ and $[q_1, q_2] \neq [q'_1, q'_2]$. See Figure 1.

Corollary 2. *By replacing SA with the grid of Chan et al. [5], reporting the MEMs associated with internal node v of the suffix tree of $\mathcal{R}^h$ takes $O((x + \sigma)(1 + \log n_h / \log\log n_h) + l^2 d)$ time, where x is the number of occurrences of $\mathsf{label}(v)$ in $\mathcal{R}^h$, $l \leq x$ is the number of those occurrences that have MEMs, and $d = \mathsf{label}(v)$.*

5 Concluding remarks

We presented a framework to compute all-vs-all MEMs in a collection of run-length encoded HiFi reads. Our techniques can be adapted to other types of collections with properties similar to that of HiFi data (e.g., Nanopore sequencing data, proteins, Phred scores, among others). The larger alphabet of proteins and Phred scores make our MEM reporting algorithm that uses the geometric data structure more relevant (as it avoids the σ^2 complexity of our first method). We are also applying these techniques to *de novo* assembly of HiFi reads.

References

1. Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the BWT of string collections. *Theoretical Computer Science*, 483:134–148, 2013.
2. Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Proc. 21st European Symposium on Algorithms (ESA)*, pages 133–144, 2013.
3. Christina Boucher, Travis Gagie, Tomohiro I, Dominik Köppl, Ben Langmead, Giovanni Manzini, Gonzalo Navarro, Alejandro Pacheco, and Massimiliano Rossi. PHONI: Streamed matching statistics with multi-genome references. In *Proc. 21st Data Compression Conference (DCC)*, pages 193–202, 2021.
4. Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
5. Timothy Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th Annual Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.
6. William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4):327–344, 1994.
7. David Clark. *Compact PAT Trees.* PhD thesis, University of Waterloo, Canada, 1996.
8. Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.
9. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *Journal of the ACM (JACM)*, 67(1):1–54, 2020.
10. Travis Gagie, Gonzalo Navarro, and Simon J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426:25–41, 2012.
11. Szymon Grabowski and Wojciech Bieniecki. copMEM: finding maximal exact matches via sampling both genomes. *Bioinformatics*, 35(4):677–678, 2019.
12. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High–Order Entropy–Compressed Text Indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
13. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
14. Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.
15. Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2001.
16. Zia Khan, Joshua S. Bloom, Leonid Kruglyak, and Mona Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, 2009.
17. Nilesh Khiste and Lucian Ilie. E-MEM: efficient computation of maximal exact matches for very large genomes. *Bioinformatics*, 31(4):509–514, 2015.
18. Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional bwt. In *Proc. 3rd International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 31–36, 2009.

19. Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.
20. Yuansheng Liu, Leo Yu Zhang, and Jinyan Li. Fast detection of maximal exact matches via fixed sampling of query k-mers and bloom filtering of index k-mers. *Bioinformatics*, 35(22):4560–4567, 2019.
21. Glennis A. Logsdon, Mitchell R. Vollger, and Evan E. Eichler. Long-read human genome sequencing and its applications. *Nature Reviews Genetics*, 21(10):597–614, 2020.
22. Udi Manber and Gene Myers. Suffix arrays: a new method for on–line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
23. Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 322–333, 2010.
24. Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 347–358, 2010.
25. Massimiliano Rossi, Marco Oliva, Paola Bonizzoni, Ben Langmead, Travis Gagie, and Christina Boucher. Finding maximal exact matches using the r-index. *Journal of Computational Biology*, 29(2):188–194, 2022.
26. Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *Journal of Computational Biology*, 29(2):169–187, 2022.
27. Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.
28. Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.
29. Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11, 1973.
30. Aaron M. Wenger, Paul Peluso, William J. Rowell, Pi-Chuan Chang, Richard J. Hall, Gregory T. Concepcion, Jana Ebler, Arkarachai Fungtammasan, Alexey Kolesnikov, Nathan D. Olson, et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nature Biotechnology*, 37(10):1155–1162, 2019.

Appendix

Algorithm 1 Computing MEMs in one traversal of the suffix tree T of $\mathcal{R}^h$. Arrays BWT, SA, and H are implicit in the pseudo-code. Each node $v \in T$ is encoded by the pair (v, d), where $v = (i, j), (i', j')$ are the ranges in SA for $\mathsf{label}(v)$ and its DNA reverse complement $\mathsf{label}(\hat{v})$, and d is the string depth.

Input: Suffix tree T for $\mathcal{R}^h$ encoded by the implicit bi-directional BWT.
Output: MEMs as described in Definition 1.

```
$S \leftarrow \emptyset$                                             ▷ Empty stack
$r \leftarrow (1, n+1), (1, n+1)$                                    ▷ The root of $T$
push$(S, (r, 0))$
while $S \neq \emptyset$ do
    $(v, d) \leftarrow$ top$(S)$          ▷ Extract suffix tree node $v$ from the top of the stack
    pop$(S)$
    if $d \geq \tau$ and isLeftMaximal$(v)$ and isRightMaximal$(v)$ then
        repMEM$(v, e, d)$
    end if
    for $c \in$ enumerateLeft$(v)$ do             ▷ Continue visiting other suffix tree nodes
        $u \leftarrow$ extendLeft$(v, c)$
        if isLeftMaximal$(u)$ then
            insert$(S, (u, d+1))$
        end if
    end for
end while
```

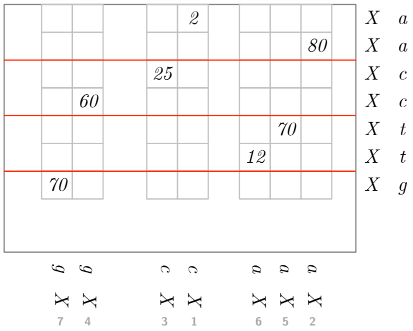

Fig. 1. Reporting MEMs from an internal node v labeled $\mathsf{label}(v) = X$ using the grid $\mathcal{G}$. The rows are labeled with the suffixes prefixed by X, while the column are labeled with the suffixes prefixed with the labels of v's Weiner links. The horizontal red lines represents the partition of the SA range for X induced by the children of v. The grey numbers below the column labels are the LF^{-1} values. For each column j', its associated SA value is in the row $\mathsf{LF}^{-1}(j') = j$.

Algorithm 2 Report all-vs-all MEMs from a suffix tree node v. Arrays BWT and H for $\mathcal{R}^h$ are implicit in the pseudo-code. Node v is encoded as described in Algorithm 1

Input: An internal node $v \in T$ with more than one Weiner link, run-length excess threshold e, and $d = |\mathsf{label}(v)|$.

Output: List of MEMs among strings of $\mathcal{R}^h$ that can be computed from v.

```
procedure repMEM($v$, $d$, $e$)
    $\mathcal{C} \leftarrow \emptyset$
    for $c \in$ enumerateRight($v$) do        ▷ Partition $SA[v.i, v.j]$ according $v$'s children
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{$extendRight($v, c$)$\}$
    end for
    $\mathcal{M} \leftarrow \emptyset$
    for $x \leftarrow 1$ to $|\mathcal{C}|$ do        ▷ Get Weiner links for every child of $v$
        $\mathcal{I}_x \leftarrow \emptyset$
        for $d \leftarrow$ enumerateLeft($\mathcal{C}[x]$) do
            $\mathcal{I}_x \leftarrow \mathcal{I}_x \cup \{$extendLeft($\mathcal{C}[x], d$)$\}$
        end for
        $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{I}_x$
    end for
    for $\mathcal{I}_a, \mathcal{I}_b \in \mathcal{M}$ with $\mathcal{I}_a \neq \mathcal{I}_b$ do ▷ $\mathcal{I}_a$ and $\mathcal{I}_b$ come from different children of $v$
        for $(X, Y) \in \mathcal{I}_a \times \mathcal{I}_b$ do
            if $X$ and $Y$ belong to distinct $SA$ bucket then
                for $(q, e) \in X \times Y$ do
                    $R_q \leftarrow$ string in $\mathcal{R}^h$ for $SA[q] + 1$
                    $R_e \leftarrow$ string in $\mathcal{R}^h$ for $SA[e] + 1$
                    $e' \leftarrow$ rlExcess($SA[q] + 1, SA[e] + 1, d$)
                    if $e' \leq e$ then
                        Report MEM in $(q, e)$
                    end if
                end for
            end if
        end for
    end for
end procedure
```