Improved Filters for the Approximate Suffix-Prefix Overlap Problem

Gregory Kucherov, Dekel Tsur

Improved filters for the approximate suffix-prefix overlap problem

Gregory Kucherov[1,2] and Dekel Tsur[2]

[1] CNRS/LIGM, Université Paris-Est Marne-la-Vallée, France
[2] Department of Computer Science, Ben-Gurion University of the Negev, Israel

Abstract. Computing suffix-prefix overlaps for a large collection of strings is a fundamental building block for the analysis of genomic next-generation sequencing data. The approximate suffix-prefix overlap problem is to find all pairs of strings from a given set such that a prefix of one string is similar to a suffix of the other. Välimäki et al. (Information and Computation, 2012) gave a solution to this problem based on suffix filters. In this work, we propose two improvements to the method of Välimäki et al. that reduce the running time of the computation.

1 Introduction

Genomic sequences are deciphered by reading short overlapping fragments. Modern *next-generation* sequencing technologies, can produce, in a single run, tens or hundreds of millions of such fragments, called *reads*, each of the order of a hundred of letters. Dealing with these gigabytes of sequence data raises a number of algorithmic challenges.

A basic operation on a collection of genomic reads is the computation of overlaps: we need to be able to quickly retrieve reads which have a significant overlap with a given read. This operation is a prerequisite for many algorithms dealing with reads, and most prominently for *genome assembly* algorithms which follow the so-called *overlap-layout-consensus* paradigm [6]. These algorithms are based on *overlap graphs* (also called *string graphs* or *assembly graphs*) that represent all significant overlaps between reads. A recent example is provided by SGA assembler [12]. Earlier, this approach was taken by several "first-generation" methods of genome assembly, such as Celera assembler [8] used to assemble one of the first versions of the human genome.

The goal of this work is to propose an efficient way of computing significant *approximate suffix-prefix overlaps* of a set of strings. Previously, several solutions have been proposed to compute all *exact* suffix-prefix overlaps [2, 11, 12]. However, in practice, we are interested in the *approximate* case when strings can overlap within a certain number of errors.

Most practical methods for computing approximate string similarities are based on the filtering approach, when the search is done in two steps: at the first step, *candidate* regions are identified that *potentially* correspond to sought matches, and at the second step, those candidates are checked to *actually* verify

the desired matching condition. Filtering algorithms usually do not yield interesting theoretical time bounds but are often very efficient in practice. As an example, *spaced seeds* [1, 7] constitute one of the filtering techniques that has been successfully used for DNA sequence comparison, e.g. [7, 10].

To compute approximate suffix-prefix overlap, the above-mentioned SGA assembler [12] uses a basic *substring filtering*. Välimäki et al. [13] proposed to apply a modified version of *suffix filters* earlier proposed by Kärkkäinen and Na [3]. Suffix filter provide a more selective filtering criterion and therefore a more efficient algorithm.

In this paper, we show how the method of [13] can be further improved. We propose two improvements that reduce the search space and therefore the running time of the algorithm: a new family of suffix filters and a new partitioning scheme. We report on estimations on random datasets that support the superiority of our schemes.

Throughout the paper, we present our method for the Hamming distance between strings, although it can be generalized to the edit distance, similar to [3, 13]. This, however, would entail additional technical details and a more involved presentation that we wanted to avoid.

2 Preliminaries

2.1 Notation

For a sequence of integers A, let $\mathsf{PrefixSum}(A)$ be a sequence of integers of the same length as A in which $\mathsf{PrefixSum}(A)[i] = \sum_{j=1}^{i} A[j]$. For two sequences of integers A and B of the same length, we define $A \leq B$ iff $A[i] \leq B[i]$ for all i.

The Hamming distance between strings A and B of the same length, denoted $\mathsf{Ham}(A, B)$, is the number of indices i for which $A[i] \neq B[i]$. If $\mathsf{Ham}(A, B) \leq k$, we say that A and B *k-match*.

Let A be a string that has a partition $A = A_1 A_2 \cdots A_k$ into k disjoint parts. Let B be a string with the same length as A. The partition of A induces a partition $B = B_1 B_2 \cdots B_k$ of B in which $|B_i| = |A_i|$ for all i. The *partition distance* between A and B, denoted $\mathsf{pd}(A, B)$, is a sequence of integers of length k in which $\mathsf{pd}(A, B)[i] = \mathsf{Ham}(A_i, B_i)$. The *accumulated partition distance* between A and B, denoted $\mathsf{apd}(A, B)$, is the sequence $\mathsf{PrefixSum}(\mathsf{pd}(A, B))$. That is, $\mathsf{apd}(A, B)[i]$ is the total number of mismatches between the first i parts of A and B.

2.2 Suffix filters

For the problem of approximate pattern matching with q errors, the most basic filtering method, called PEX in [9], consists in splitting the pattern into $k = q+1$ parts and searching for those parts independently. Once one of the parts is found, it provides a candidate location for the whole pattern. Kärkkäinen and Na [3] proposed an interesting extension of this principle called *suffix filters*. Suffix

filters have been shown to generate many fewer candidates than substring filters, and therefore to be much more efficient. Since the present work builds on this idea, we briefly explain it here.

We still split pattern P into $k = q + 1$ parts $P = P_1 P_2 \cdots P_k$, but instead of searching for substrings P_i, we will be searching for *suffixes* $P_i P_{i+1} \cdots P_k$ for $i = 1, \ldots, k$ allowing errors distributed according to a specific pattern

$$\mathsf{filter}_{k,i} = 01 \cdots (k - i).$$

This means that for every i, we are searching for strings B such that $\mathsf{apd}(P_i \cdots P_k, B) \leq \mathsf{filter}_{k,i}$. The key observation of [3] is that this scheme detects all possible occurrences of P within q errors.

For example, let $q = 2$, namely, we want to find the substrings of the text that 2-match to $P = P_1 P_2 P_3$. All such substrings are detected using three filters, denoted by sequences 012, 01, and 0. Filter 012 detects substrings B such that $\mathsf{apd}(P_1 P_2 P_3, B) \leq 012$. That is, $B = B_1 B_2 B_3$, such that $|B_i| = |P_i|$ for all i, $B_1 = P_1$, $\mathsf{Ham}(B_2, P_2) \leq 1$, and $\mathsf{Ham}(B_2 B_3, P_2 P_3) \leq 2$. By a slight abuse of language, the set of all such strings B will be said to be *enumerated* by the filter. Similarly, filter 01 detects substrings $B = B_2 B_3$ such that $B_2 = P_2$ and B_3 is within one error from P_3. Each such string is a suffix of a candidate approximate occurrence of P. Finally, filter 0 detects substrings B_3 with $B_3 = P_3$, which provides again a suffix of a candidate occurrence of P.

Observe that there are 9 cases for the partition distance between P and B — 011, 101, 110, 002, 020, 200, 100, 010, and 001 — which are all covered by suffix filters 012, 01 and 0. Indeed, filter 012 covers cases 011, 002, 010, and 001, filter 01 covers cases 101, 200 and 100, and filter 0 covers cases 020 and 110.

The set of strings enumerated by a filter can be naturally represented by a *trie* (see Figure 1), where branching nodes correspond to positions where the filter allows a possible mismatch to occur. The number of nodes in the tries of all the filters is a crucial parameter for the efficiency of a filtering scheme.

2.3 Suffix filters and full-text indexes

One of the advantages of suffix filters (as opposed e.g. to spaced seeds) is that they can be naturally implemented using full-text indexes that support incremental string matching. Those indexes include classical indexes such as suffix trees, but also succinct indexes such as FM-index and its variants [14]. These indexes allow reading a pattern left-to-right[3] and quickly updating the index point after each letter, so that all occurrences of the prefix read so far can be retrieved efficiently.

Implementing suffix filters on a full-text index can be done simply by *enumerating* all "approximate suffixes" of the pattern detected by a filter, and reporting the occurrences of all these strings in the text, thus generating the candidate set.

[3] For some indexes, such as FM-index, matching is performed right-to-left, in which case we can just assume that the indexed sequences are reversed. On the other hand, FM-index can also be modified to perform matching left-to-right, see [5].

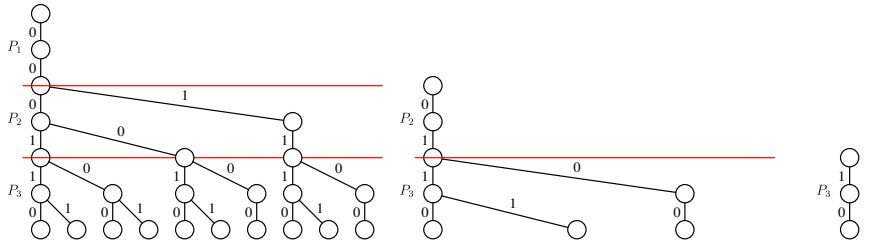

Fig. 1. The tries of the filters 012, 01, and 0 applied on the pattern $P = 000110$, for binary alphabet and $q = 2$ errors. The pattern P is partitioned into 3 parts $P = P_1P_2P_3$ where $P_1 = 00$, $P_2 = 01$, and $P_3 = 10$. The filter 012 enumerates the strings corresponding to the leaves of the tries in Figure (a), namely 000110, 000111, ..., 001100. The filter 01 enumerates the strings 0110, 0111, and 0100, and the filter 0 enumerates the string 10.

More precisely, for each of the tries that correspond to the filters, the algorithm traverses the trie in depth-first search order, and updates the index point after each descend in the trie. When reaching a node in the trie whose corresponding string does not appear in the index, the search does not continue to the descendants of this node in the trie. When reaching a leaf of the trie, all occurrences of the string corresponding to the leaf are retrieved from the index, and are added to the set of candidates.

2.4 Suffix filters applied to suffix-prefix overlap problem

Given a set $\mathcal{S}$ of strings, the *approximate suffix-prefix overlap* problem is to compute all significant approximate overlaps between pairs of strings of $\mathcal{S}$. "Significance" is defined by a lower threshold $l_{\min}$ on the overlap size. Since the overlap size is variable, imposing a fixed number of errors is not reasonable, and a relative error rate is specified instead. Formally, given an integer $l_{\min}$, and $\epsilon > 0$, we have to find all pairs of strings $S, S' \in \mathcal{S}$ such that there is an integer $l \geq l_{\min}$ for which the prefix of S of length l and the suffix of S' of length l $\lceil \epsilon l \rceil$-match.

In this Section, we explain how suffix filters can be used to solve the approximate suffix-prefix overlap problem. We proceed by enumerating all strings $S \in \mathcal{S}$. For each $S \in \mathcal{S}$ and for each $l \geq l_{\min}$, we want to identify all strings $S' \in \mathcal{S}$ whose suffix of length l $\lceil \epsilon l \rceil$-match the prefix $S[1..l]$. We want to do it by applying suffix filters designed for patterns of length l and $\lceil \epsilon l \rceil$ mismatches. If such a filter applies, then a candidate overlap is generated, which is a triplet (S, S', l). At the verification stage, the actual Hamming distance between the prefix of S of length l and the suffix of S' of length l is computed, and the overlap is reported if this distance is no more than $\lceil \epsilon l \rceil$.

Let us now explain how the filtering algorithm works with the filters of Section 2.2. Fix S and a partition $S = S_1S_2 \cdots$ of S into disjoint parts. Fix $l \geq l_{\min}$

and let $P = S[1..l]$. The partition of S induces a partition $P = P_1 \cdots P_k$ of P, where $P_i = S_i$ for $i < k$ and P_k is a prefix of S_k. Suppose that the partition of S was chosen a way to ensure that $k \geq \lceil \epsilon l \rceil + 1$. This allows us to apply suffix filters of Section 2.2.

Consider the above filters $\mathsf{filter}_{k,1}, \ldots, \mathsf{filter}_{k,k}$, where $\mathsf{filter}_{k,i} = 01 \cdots (k-i)$. Each filter $\mathsf{filter}_{k,i}$ enumerates all strings B such that $\mathsf{apd}(P_i \cdots P_k, B) \leq F$, and B appears as a substring of some string in $\mathcal{S}$. For each such string B, the algorithm further selects all the strings $S' \in \mathcal{S}$ that *end with* B and adds the triplets (S, S', l) to the list of candidates.

The main difficulty in applying suffix filters to the approximate suffix-prefix overlap problem is that the length l and, consequently, the number of errors are not fixed. Once the partition of S is defined, our goal is to deal with all values of l in one left-to-right traversal of S. For each $l \geq l_{\min}$, we should be able to efficiently identify appropriate filters that apply to the corresponding partition of $S[1..l]$. We now describe how it is done for the filters of Section 2.2.

A key point here is that the enumeration processes for different values of l are connected. Let k_l denote the number of parts in the partition of $S[1..l]$ that is induced by the partition of S. Let $\mathcal{B}_{S,i,l}$ be the set of strings enumerated by the filter $\mathsf{filter}_{k_l,i}$ when it is applied to $S[1..l]$. That is, $\mathcal{B}_{S,i,l}$ are the strings enumerated by $\mathsf{filter}_{k_l,i}$ when considering the filtering scheme for the fixed value of l. Let $\mathsf{trie}_{S,i}$ be the trie representing the set $\mathcal{B}_{S,i,|S|}$. We have the following property: For every l, where $l_{\min} \leq l < |S|$, and every $i \leq k_l$, the prefix of $\mathsf{filter}_{k_{|S|},i}$ of length $k_l + 1 - i$ is equal to $\mathsf{filter}_{k_l,i}$. It follows that the set $\mathcal{B}_{S,i,l}$ is equal to the set of strings that correspond to the nodes of $\mathsf{trie}_{S,i}$ at depth $l - \sum_{j<i} |S_j|$. Therefore, generation of candidates can be done for all values of l by the following algorithm. For $i = 1, 2, \ldots, k_S$, traverse $\mathsf{trie}_{S,i}$. When the traversal is at a node that corresponds to a string B, if $B \in \mathcal{B}_{S,i,l}$ for some $l \geq l_{\min}$, find all the strings $S' \in \mathcal{S}$ that ends with B, and for each such S' add the triplet $(S, S', |B| + \sum_{j<i} |S_j|)$ to the list of candidates.

Checking whether $B \in \mathcal{B}_{S,i,l}$ is done with the following *candidate generation condition*.

Condition 1 *$B \in \mathcal{B}_{S,i,l}$ if and only if*

$$|B| + \sum_{j<i} |S_j| \geq l_{\min} \tag{C1}$$

2.5 The filtering scheme of Välimäki et al. [13]

Välimäki et al. [13] observed that the filtering procedure of Section 2.4 is very inefficient. The inefficiency is caused by the filters $\mathsf{filter}_{k,k} = 0$ that has to be applied, during the search, to short strings including those consisting of a single letter. Formally, when traversing the trie $\mathsf{trie}_{S,i}$, a node of the trie at depth 1, whose corresponding single-letter string is $B = S[1 + \sum_{j<i} |S_j|]$, can generate candidates if Condition (C1) is satisfied. Since B has length 1, there can be many strings $S' \in \mathcal{S}$ that ends with B, generating many spurious candidates. More

generally, assuming the strings of $\mathcal{S}$ are sampled randomly from an i.i.d. source, the expected number of candidates generated by a string B corresponding to some node in a trie is $m/\sigma^{|B|}$, where m is the number of strings in $\mathcal{S}$, and σ is the size of the alphabet. Therefore, the total number of candidates is dominated by the number of candidates generated by nodes of small depth.

The solution of Välimäki et al. to this problem is to drop the filters $\mathsf{filter}_{k,k}$ from the filtering scheme. In order to handle the cases of partition distances that were covered by this filter, filters $\mathsf{filter}_{k,1} = 01\cdots k$ are modified to $\mathsf{filter}_{k,1} = 12\cdots(k+1)$. It is shown [13] that with this modification, all combinations of partition distance are still covered.

Due to the dropping of filters $\mathsf{filter}_{k,k}$, Condition 1 should be modified for the new filtering scheme. The modified candidate generating condition will now be as follows.

Condition 2 *$B \in \mathcal{B}_{S,i,l}$ if and only if Condition (C1) is satisfied and*

$$|B| > |S_i| \tag{C2}$$

Clearly, the additional condition (C2) reduces the number of generated candidates.

3 New filtering scheme

In the filtering scheme of Välimäki et al., the filters $\mathsf{filter}_{k,1}$ start with 1 whereas the other filters start with 0. This has the consequence that the number of nodes in the trie $\mathsf{trie}_{S,1}$ is much larger than the number of nodes in the tries $\mathsf{trie}_{S,i}$ for $i > 1$. We now present a new filtering scheme without this property. Our filtering scheme consists of filters $\mathsf{filter}_{k,1}, \ldots, \mathsf{filter}_{k,k-1}$, where $\mathsf{filter}_{k,i}$ is a sequence of length $k-i+1$ whose first $k-i$ elements are $0, 1, \ldots, (k-i-1)$, and the last element is $k-i-1$. For example, for $k=4$ the filters are $\mathsf{filter}_{4,1} = 0122$, $\mathsf{filter}_{4,2} = 011$, and $\mathsf{filter}_{4,3} = 00$. Our filtering scheme requires a difference of at least 2 between the number of parts in the partition of $S[1..l]$ and $\lceil \epsilon l \rceil$ (recall that in the scheme of Välimäki et al., the required difference is at least 1). We show the correctness of this scheme in the following lemma.

Lemma 1. *For every $k \geq 2$ and every sequence of integers M of length k whose sum is at most $k-2$, there is an integer i such that $\mathsf{PrefixSum}(M[i..k]) \leq \mathsf{filter}_{k,i}$.*

Proof. The proof is by induction of k. The base of the induction $k = 2$ is trivial since in this case $M = 00$ and therefore $\mathsf{PrefixSum}(M) \leq \mathsf{filter}_{k,k-1}$. Now consider $k > 2$. Let M be a sequence of length k whose sum is at most $k-2$. If $\mathsf{PrefixSum}(M) \leq \mathsf{filter}_{k,1}$ we are done. Otherwise, there is an index i such that $\mathsf{PrefixSum}(M)[i] > \mathsf{filter}_{k,1}[i]$. Since $\mathsf{PrefixSum}(M)[i] \leq k-2$ and $\mathsf{filter}_{k,1}[j] = k-2$ for $j > k-2$, it follows that $i \leq k-2$. Therefore $\mathsf{filter}_{k,1}[i] = i-1$ and $\mathsf{PrefixSum}(M)[i] \geq i$. Let $M' = M[i+1..k]$. The length of M' is $k-i$ and the sum of M' is at most $k-2-\mathsf{PrefixSum}(M)[i] \leq k-2-i$. By the induction

hypothesis, there is an index i' such that $\mathsf{PrefixSum}(M'[i'..k-i]) \leq \mathsf{filter}_{k-i,i'}$. The lemma follows since $M'[i'..k-i] = M[i+i'..k]$ and $\mathsf{filter}_{k-i,i'} = \mathsf{filter}_{k,i+i'}$. □

To illustrate Lemma 1, observe that the three above-mentioned filters $\mathsf{filter}_{4,i}$, $i = 1, 2, 3$ cover all possible partition distances for the case of 4 parts and 2 errors. Indeed, filter 0122 cover cases 0020, 0002, 0101 and 0110, filter 011 cover cases 0011, 1001 and 1010, and finally filter 00 covers cases 2000, 0200 and 1100.

In addition to reducing the number of nodes in the tries, our filtering scheme also reduces the number of generated candidates. In the filtering scheme of Välimäki et al., $\mathsf{filter}_{k,i}$ is a prefix of $\mathsf{filter}_{k',i}$ for all $k < k'$ and i. Our filtering scheme does not have this property. Therefore, we need a new condition for checking whether $B \in \mathcal{B}_{S,i,l}$.

Condition 3 *$B \in \mathcal{B}_{S,i,l}$ if and only if conditions (C1) and (C2) are satisfied and*

$$\mathsf{apd}(S', B) \leq \mathsf{filter}_{k,i} \tag{C3}$$

where S' is the prefix of $S_i S_{i+1} \cdots$ of length $|B|$ and k is the number of parts in the partition of S' induced by the partition of S.

Observe that in the filtering scheme of Välimäki et al., every node of $\mathsf{trie}_{S,i}$ with depth $|S_i| + 1$ generated candidates. However, in our new scheme, only the node whose corresponding string B is equal to S' generates candidates among the nodes of depth $|S_i| + 1$ (for every other node, $\mathsf{apd}(S', B) = 01$ and therefore Condition (C3) is not satisfied). The same is true for depths $|S_i| + 2, \ldots, |S_i| + |S_{i+1}|$.

Our scheme can be generalized by introducing a parameter $s \geq 2$. The filters for a given value of s are $\mathsf{filter}_{k,i} = 01 \cdots (k - i - s)(k - i - s + 1)^s$ for $i = 1, \ldots, k - s + 1$. This scheme requires a difference of at least s between the number of parts in the partition of $S[1..l]$ and $\lceil \epsilon l \rceil$.

4 Partition schemes

The efficiency of the algorithm of the previous section depends on the sizes of the parts in the partition of S: having larger parts reduces the number of trie nodes and the number of candidates. For correctness of the algorithm, the partitioning of a string S must satisfy the following property.

(P1) For every $l \geq l_{\min}$, the number of parts in the partition of $S[1..l]$ induced by the partition of S is at least $\lceil \epsilon l \rceil + s$, where $s = 1$ for the filtering scheme of Välimäki et al. and $s = 2$ for our new scheme (or some fixed value of s for our extended scheme).

Välimäki et al. used a partition of S into equal sized parts of size p, except for the last part whose size is at most p. The value of p is chosen to be the maximum integer for which Property (P1) is satisfied.

We propose a partitioning scheme in which most parts are larger than those of the equal sized parts partitioning. Since the efficiency of the filtering approach depends on the sizes of the parts, the new partitioning scheme gives better performance.

Let S be a string to be partitioned. Let $l_0 < l_1 < \cdots < l_q$ be all the indices in the range $l_{\min}, \ldots, |S|$ for which $\lceil \epsilon(l-1) \rceil < \lceil \epsilon l \rceil$, and let $l_{q+1} = |S| + 1$. Let $k = \lceil \epsilon l_0 \rceil + s - 1$. We partition S as follows. The sizes of the first k parts are chosen in order to satisfy the following properties.

1. The total length of the first k parts is $l_0 - 1$.
2. The length of the k-th part is at least $l_0 - l_{\min}$.

We can set for example the length of the k-th part to be $L = \max(\lceil (l_0 - 1)/k \rceil, l_0 - l_{\min})$, and set the lengths of the first $k-1$ parts to be p or $p+1$, where $p = \lfloor (l_0 - 1 - L)/(k-1) \rfloor$. The lengths of the remaining parts in the partition are $l_1 - l_0, l_2 - l_1, \ldots, l_{q+1} - l_q$. We now show that this partition satisfies Property (P1). Moreover, the partitioning is optimal in the sense that the inequalities of Property (P1) are satisfied with equality.

Lemma 2. *For every $l \geq l_{\min}$, the number of parts in the partition of $S[1..l]$ induced by the partition of S is $\lceil \epsilon l \rceil + s$.*

Proof. For $l_{\min} \leq l < l_0$ (assuming $l_0 > l_{\min}$) we have by construction that the number of parts in the partition of $S[1..l]$ is $k = \lceil \epsilon l_0 \rceil + s - 1$. By the definition of l_0, $\lceil \epsilon l \rceil = \lceil \epsilon l_0 \rceil - 1$, so the equality of the lemma is satisfied. Similarly, if $l_i \leq l < l_{i+1}$ then the number of parts in the partition of $S[1..l]$ is $k + 1 + i$. Moreover, $\lceil \epsilon l \rceil = \lceil \epsilon l_0 \rceil + i$. Therefore, the lemma follows. □

As an example, consider partitioning a string of length 200 with the parameters $l_{\min} = 40$ and $\epsilon = 0.1$. If $s = 1$, the equal sized partition uses parts of size 8. In our new partitioning scheme, the first 5 parts have size 8, and the remaining parts have size 10.

5 Experimental results

In this Section, we compare the performance of 4 filtering schemes:

(1) the filtering scheme of Välimäki et al. [13] using partitioning into equal parts,
(2) the filtering scheme of Välimäki et al. combined with our new partitioning scheme (Section 4),
(3) our new filtering scheme (Sections 3-4), and
(4) our extended filtering scheme (see end of Section 3), with $s = 3$.

For our filtering scheme (3), string partitioning is done with our new partitioning scheme. The comparisons have been done using the technique described in Kucherov et al. [4]. In this analysis, we assume that characters of the strings of $\mathcal{S}$ are randomly chosen uniformly and independently from the alphabet. Under this assumption, we analytically estimate the expected number of nodes in the tries

Table 1. Expected performance of the filtering scheme of Välimäki et al. [13] and our filtering schemes. It is assumed that $\mathcal{S}$ contains m random strings of length 300 over an alphabet of size 4. For each scheme, the first column shows the expected number of nodes in the tries $\mathsf{trie}_{S,i}$ for all i (for a single $S \in \mathcal{S}$), and the second column is the expected number of candidates generated for S.

m	$l_{\min}$	ϵ	method of [13]		[13] with unequal parts		our scheme		our scheme extended for $s = 3$	
10^6	20	0.1	31257	18950	11782	144	8582	341	20026	95
10^7	20	0.1	64201	189506	22161	1449	13219	3412	46662	952
10^6	40	0.1	10416	839	8260	65	6912	28	8868	0.5
10^7	40	0.1	14391	8391	11138	651	8921	280	12916	4
10^7	40	0.15	207504	857318	71271	82559	40671	18842	82164	116

$\mathsf{trie}_{S,i}$ and the expected number of generated candidates, following the method developed in [4]. The results are summarized in Table 1. Columns 2 to 5 of the Table correspond to schemes (1) to (4) above, respectively.

Note that for different parameters, the bottleneck of the computation can be either the size of traversed tries, or the number of generated candidates. In all cases, we observe a significant decrease of both these measures compared to the original method of Välimäki et al. [13]. When the threshold $l_{\min}$ is small (in our experiments, 20 for the sequence length 300), the filters of [13] combined with our partitioning scheme presents a trade-off with our filtering scheme: our scheme yields a smaller number of traversed trie nodes but a larger number of generated candidates. However, when $l_{\min}$ is large enough (in our experiments, 40 for the sequence length 300), our scheme outperforms the one of Välimäki et al. in both the number of nodes in the tries and the number of generated candidates. The extended scheme with $s = 3$ yields a smaller number of candidates, but at the cost of increased number of traversed trie nodes.

6 Conclusions

In this paper, we proposed an improved filtering scheme for the approximate suffix-prefix overlap problem directly raised by bioinformatics applications. Two improvements are proposed: we provide a more efficient filtering scheme as well as new way of partitioning the query string. We show, through analytical estimations, the superiority of our scheme in terms of the size of the search space (size of traversed tries) as well as the selectivity (number of generated candidates).

Several directions for future work can be envisaged. We did not compare the actual performance of the different filtering schemes on real data. However, previous work [4] provides strong grounds to assume that the better performance will be supported by real data too. This, however, remains to be verified experimentally. Another direction, already mentioned in Introduction, concerns the generalization of our results to the case of edit distance. While we don't expect significant obstacles in this generalization, it does bring an additional technical difficulty.

Acknowledgements. GK has been supported by the ABS2NGS grant of the French government (program *Investissement d'Avenir*) as well as by a EU Marie-Curie Intra-European Fellowship for Carrier Development. DT has been supported by ISF grant 981/11.

References

1. S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informaticae*, 56(1,2):51–70, 2003.
2. D. Gusfield, G. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, March 1992.
3. J. Kärkkäinen and J. C. Na. Faster filters for approximate string matching. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–90, 2007.
4. G. Kucherov, K. Salikhov, and D. Tsur. Approximate string matching using a bidirectional index. In *Proceedings of the 25th Annual Symposium on Combinatorial Pattern Matching (CPM), June 16-18, 2014, Moscow (Russia)*, volume 8486 of *Lecture Notes in Computer Science*, pages 222–231. Springer, 2014. Full version at http://arxiv.org/abs/1310.1440.
5. T. W. Lam, R. Li, A. Tam, S. C. K. Wong, E. Wu, and S.-M. Yiu. High throughput short read alignment via bi-directional BWT. In *Proc. IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 31–36, 2009.
6. Z. Li, Y. Chen, D. Mu, J. Yuan, Y. Shi, H. Zhang, J. Gan, N. Li, X. Hu, B. Liu, B. Yang, and W. Fan. Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-Bruijn-graph. *Brief Funct Genomics*, 11(1):25–37, Jan 2012.
7. B. Ma, J. Tromp, and M. Li. PatternHunter: Faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.
8. E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. Reinert, K. A. Remington, E. L. Anson, R. A. Bolanos, H. H. Chou, C. M. Jordan, A. L. Halpern, S. Lonardi, E. M. Beasley, R. C. Brandon, L. Chen, P. J. Dunn, Z. Lai, Y. Liang, D. R. Nusskern, M. Zhan, Q. Zhang, X. Zheng, G. M. Rubin, M. D. Adams, and J. C. Venter. A whole-genome assembly of Drosophila. *Science*, 287(5461):2196–2204, Mar 2000.
9. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.
10. L. Noé and G. Kucherov. YASS: enhancing the sensitivity of DNA similarity search. *Nucleic Acid Research*, 33:W540–W543, 2005.
11. E. Ohlebusch and S. Gog. Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Information Processing Letters*, 110(3):123 – 128, 2010.
12. J. T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Res.*, 22(3):549–556, Mar 2012.
13. N. Välimäki, S. Ladra, and V. Mäkinen. Approximate all-pairs suffix/prefix overlaps. *Information and Computation*, 213:49–58, 2012.
14. M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. Prospects and limitations of full-text index structures in genome analysis. *Nucleic Acids Res.*, 40(15):6993–7015, Aug 2012.