A Randomized Numerical Aligner -- rNA

Journal of Computer and System Sciences 78 (2012) 1868–1882

Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

A randomized Numerical Aligner (rNA)

Alberto Policriti [a,b], Alexandru I. Tomescu [a,c], Francesco Vezzi [a,b,*]

[a] Dipartimento di Matematica e Informatica, Università di Udine, Via delle Scienze, 206, 33100 Udine, Italy
[b] Istituto di Genomica Applicata (IGA), Via J. Linussio, 51, 33100 Udine, Italy
[c] Faculty of Mathematics and Computer Science, University of Bucharest, Str. Academiei, 14, 010014 Bucharest, Romania

ARTICLE INFO

Article history:
Received 14 September 2010
Received in revised form 9 March 2011
Accepted 17 November 2011
Available online 23 December 2011

Keywords:
Approximate string matching
Hamming distance
Sequencing
Short string aligner
Next Generation Sequencing

ABSTRACT

With the advent of new sequencing technologies able to produce an enormous quantity of short genomic sequences, new tools able to search for them inside a genomic reference sequence have emerged. Because of chemical reading errors or of the variability between organisms, one is interested in finding not only exact occurrences, but also occurrences with up to k mismatches. The contribution of this paper is twofold. On the one hand, we present a generalization of the classical Rabin–Karp string matching algorithm to solve the k-mismatch problem, with average complexity $\mathcal{O}(n+m)$ (n text and m pattern lengths, respectively). On the other hand, we show how to employ this idea in conjunction with an index over the text, allowing to search a pattern, with up to k mismatches, in time proportional to its length. This novel tool—rNA (randomized Numerical Aligner)—is in general faster and more accurate than other available tools like SOAP2, BWA, and BOWTIE. rNA executables and source code are freely available at http://iga-rna.sourceforge.net/.

1. Introduction

One of the main applications of string matching is computational biology. A DNA sequence can be seen as a string over the alphabet $\Sigma = \{A, C, G, T\}$. Given a genomic reference sequence, we are interested in searching (*aligning*) different sequences (*reads*) of various lengths. Reads are produced by sequencing machines able to read stretches of DNA of a given organism. When aligning such reads against another DNA sequence, we must consider errors due to the sequencer and intrinsic errors due to the variability between organisms. For these reasons, all the programs aligning reads against a reference sequence must deal with mismatches [1,2]. String matching can be divided into two main areas: exact string matching and approximate string matching. When doing approximate string matching, we need to employ a distance metric between strings. The most commonly used metrics are the *edit distance* (or Levenshtein distance) and the *Hamming distance*.

The first algorithms to solve the exact string matching problem are due to Knuth, Morris and Pratt [3], Boyer and Moore [4], running in time $\mathcal{O}(n+m)$ (n text and m pattern lengths, respectively), and Rabin and Karp [5], requiring time $\mathcal{O}(n+m)$ on average.

* Corresponding author at: Dipartimento di Matematica e Informatica, Università di Udine, Via delle Scienze, 206, 33100 Udine, Italy.
E-mail addresses: alberto.policriti@uniud.it (A. Policriti), alexandru.tomescu@uniud.it (A.I. Tomescu), francesco.vezzi@uniud.it (F. Vezzi).

0022-0000/$ – see front matter
doi:10.1016/j.jcss.2011.12.007

When a large number of patterns must be searched, these solutions do not perform well, as the text has to be scanned for each pattern. In such cases, it is convenient to build an index over the text, allowing to search a pattern in time proportional to its length, or over the patterns, in which case the reference is scanned only once. Usually, when the reference is fixed, or the total pattern length is larger than the reference, indexing the text is the preferred solution. For example, this method is employed by popular tools for searching inside DNA strings, such as SOAP2 [6], BWA [7], or BOWTIE [8]. For a detailed discussion on when, in biological applications, an index over the text is preferred to one over the patterns, and vice-versa, refer to [9]. The most popular such indexes are Suffix Trees (see e.g. [10–12]) and Suffix Arrays [13]. For a complete review on indexing algorithms refer to [14].

Approximate string matching at distance k under the edit metric is called the *k-difference problem*, while under the Hamming metric, it is called the *k-mismatch problem*. A simple algorithm for the k-difference problem is based on dynamic programming and it has a running time $\mathcal{O}(nm)$. Several efforts were made to improve this result. Abrahamson [15] shows that string matching with mismatches can be solved in time $\mathcal{O}(n\sqrt{m\log m})$. The fastest solutions for the k-mismatch problem relies heavily on the ability to search the suffix tree of the text and of the pattern. Landau and Vishkin [16,17] introduced a method running in time $\mathcal{O}(nk)$ that uses constant time lowest common ancestor queries on the suffix trees of P and T (which is now known as "kangaroo hopping"). The algorithm of Galil and Giancarlo [18] attains the same complexity $\mathcal{O}(nk)$. A more recent paper [19] proposed a variation of FAAST [20] that has average running time $\mathcal{O}(n(\log m + k)/m)$ that was proved to be optimal for approximate string matching [21]. The asymptotic running time was improved in [22] to $\mathcal{O}(n\sqrt{k\log k})$, by a method based on counting and filtering, the suffix tree with kangaroo hooping, and fast Fourier transforms, which may ultimately lead to a more sophisticated implementation.

The first algorithm that solved the k-mismatch problem with the construction of an index is due to Ukkonen and Jokinen [23]. The first solution with query time depending only on k and m was proposed by Ukkonen [12] using Suffix Trees. More recently [24], the k-difference problem has been solved in time $\mathcal{O}(|\Sigma|^k m^k \max(k, \log n))$ where Σ is the alphabet, using compressed Suffix Arrays [25].

In many practical applications, we are interested in finding the *best* occurrence of the pattern, with at most k mismatches (the *best k-mismatch problem*—to be introduced in Section 1.1). Recently, a flurry of papers presenting new indexing algorithms to solve this problem appeared [7,6,8]. All these algorithms aim to search inside a reference sequence the myriad of reads that are produced by new sequencing technologies (for further details refer to `www.illumina.com`, `www.solid.com`, and `www.appliedbiosystems.com`). For example, the latest available Illumina sequencer, HiSeq2000, can produce 200 billion characters (called *bases*) in a single experiment, grouped in reads of length 100 (their length is expected to grow to 150 bases in the near future). Tools like SOAP2 [6] are able to align this large set of reads in a very short time, thanks to advanced indices and heuristics, that can, however, reduce accuracy.

In this paper we focus on the *best k-mismatch problem*, which we formally introduce in Section 1.1. In Section 2 we show how the on-line algorithm of Rabin and Karp [5] can be generalized to solve the k-mismatch problem, with an average time complexity of $\mathcal{O}(n+m)$. This idea is employed in Section 3, along with an index over the text, for solving in a precise way the *best* k-mismatch problem; this allows the search of a pattern in time proportional to its length. Section 4 sets up a formal framework explaining why this method is computationally efficient, and discusses some envisaged extensions.

Our proposed algorithm for the best k-mismatch problem has been implemented into a usable tool for bioinformatics, as explained in Section 5. This tool, which we call 'randomized Numerical Aligner' (**rNA**), is freely available at `http://iga-rna.sourceforge.net/`. Even though we do not sacrifice accuracy, the experimental results of Section 6 show that our algorithm has better performance than the most used aligners for short reads, such as SOAP2 [6], BWA [7], or BOWTIE [8].

1.1. Problem definition and notations

Let $\Sigma = \{0, 1, \ldots, b-1\}$ be an alphabet of $b \geqslant 2$ characters, and let $c, d \in \Sigma$. Define $\mathrm{neq}(c,d) = 1$ if $c \neq d$, and 0 otherwise. Let $X = X[0]X[1]\ldots X[n-1]$ and $Y = Y[0]Y[1]\ldots Y[n-1]$ be two strings over the alphabet Σ. The *Hamming distance* between X and Y is defined as $d_H(X,Y) =_{\mathrm{def}} \sum_{i=0}^{n-1} \mathrm{neq}(X[i], Y[i])$. Given numbers $0 < m \leqslant n$ and $0 \leqslant s \leqslant n-m$, we denote by $X_{(s)}$ the string $X_{(s)} =_{\mathrm{def}} X[s]X[s+1]\ldots X[s+m-1]$. We denote the numerical radix-b representation of a string X of length n by $x =_{\mathrm{def}} b^{n-1}X[0] + b^{n-2}X[1] + \cdots + bX[n-2] + X[n-1]$. Given a positive integer q, the number $\hat{x}$ stands for $x \bmod q$, and is called the *fingerprint* of the string X. The *k-mismatch problem* is defined as follows:

IN: Text $T = T[0]T[1]\ldots T[n-1]$, pattern $P = P[0]P[1]\ldots P[m-1]$, over the alphabet Σ, and a natural number $k < m$.
OUT: All pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$ and $d_H(P, T_{(s)}) \leqslant k$.

For such a pair $\langle s, d_H(P, T_{(s)})\rangle$, we say that P occurs (with mismatches) with shift s in T. If $d_H(P, T_{(s)}) = 0$, we say that $T_{(s)}$ is an exact occurrence of P.

In many applications, like in the search of a sets of sequences (reads) inside a genomic reference sequence, one is given a text T, of length n, and a collection $\mathcal{P}$ of patterns (usually of the same length m) and is required to find the *best*

occurrences of each $P \in \mathcal{P}$, with at most k mismatches. This problem, referred to in what follows as the *best k-mismatch* (*alignment*) *problem*, can be formulated in the following way:

IN: Text $T = T[0]T[1]\dots T[n-1]$, a collection $\mathcal{P}$ of patterns of length m, all over the alphabet Σ, and a natural number $k < m$.

OUT: For every $P \in \mathcal{P}$, all pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$ and $d_H(P, T_{(s)}) \leqslant k$, such that for all $0 \leqslant s' \leqslant n-m$ we have $d_H(P, T_{(s)}) \leqslant d_H(P, T_{(s')})$.

2. An on-line algorithm for string matching with *k* mismatches

One of the simplest exact string matching algorithms—that also performs well in practice—is the Rabin–Karp randomized algorithm [5]. For every $s = 0\dots n-m$, the algorithm encodes P and any $T_{(s)}$ by the radix-b numbers p and $t_{(s)}$, respectively, and replaces expensive string comparisons by constant-time suitable numerical comparisons. As usually m is larger than the length of a processor word, instead of storing p and $t_{(s)}$, one keeps the values $\hat{p} = p \bmod q$ and $\hat{t}_{(s)} = t_{(s)} \bmod q$. As an indication that P *may* occur with shift s in T, the algorithm now tests whether $\hat{p} = \hat{t}_{(s)}$ and, if so, it proceeds to a character-by-character comparison of P and $T_{(s)}$. Randomly choosing q to be a prime number in the interval $[2, mn^2]$, the test $\hat{p} = \hat{t}_{(s)}$ produces few false positives [5] (i.e., it gives a positive answer in the case when $P \neq T_{(s)}$). Moreover, as $\hat{t}_{(s+1)}$ can be computed from $\hat{t}_{(s)}$ in constant time, the overall expected time complexity is $\mathcal{O}(n+m)$.

The Rabin–Karp method has already been employed in [26] to solve the k-mismatch problem. That approach is based on generating all the $\sum_{i=0}^{k}\binom{m}{i}(b-1)^i$ strings obtained from P with at most k mismatches. In this paper we will instead make use of some algebraic properties of the Hamming distance under the modulo operation. In this way, we can replace 'generation' by 'verification', and we can reduce the exponential blow-up on m, to an exponential blow-up on the length w of a processor word.

We will retain the advantageous features of the Rabin–Karp algorithm, like encoding strings by a radix-b number, and storing values modulo an appropriate number q. The only point where a change is needed is in the heuristic checking whether the pattern occurs with shift s (i.e., in the test $\hat{p} = \hat{t}_{(s)}$). In what follows, we will seek an answer to these questions:

1. If $d_H(P, T_{(s)}) \leqslant k$, then what *fast* test on the available data (e.g., $\hat{p}$, $\hat{t}_{(s)}$) can we use to detect such a situation?
2. How can we guarantee that this test produces *few* false positives, and what is the probability of such an event?

We note that when $k = 0$, then $\hat{p} = \hat{t}_{(s)}$ is equivalent to $(\hat{p} - \hat{t}_{(s)}) \bmod q = 0$. With this clue in mind, we still compute $(\hat{p} - \hat{t}_{(s)}) \bmod q$, but we will try to characterize the set $\mathcal{Z}(k,q) \subseteq \{0,\dots,q-1\}$, such that whenever $d_H(P, T_{(s)}) \leqslant k$, then $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$ holds. More formally, the set $\mathcal{Z}(k,q)$ is defined as follows.

Definition 1. Given $m > 0$, $0 < k < m$ and $q > 0$, define $\mathcal{Z}(k,q)$ to be the set

$$\mathcal{Z}(k,q) =_{\text{def}} \left\{ (x-y) \bmod q \;\middle|\; X, Y \in \Sigma^m,\ d_H(X,Y) \leqslant k \right\}.$$

We will sometimes refer to the elements of $\mathcal{Z}(k,q)$ as *witnesses*, as they testify that two strings *can* be at Hamming distance at most k. The algebraic difference between the numerical representations of two strings at a given Hamming distance is characterized in Lemma 1.

Lemma 1. *Given two strings X and Y of the same length m, for any $0 < k < m$ we have $d_H(X,Y) = k$* if and only if

$$x - y \in \Big\{ (-1)^{u_1} t_1 b^{i_1} + \cdots + (-1)^{u_k} t_k b^{i_k} \colon\ u_1,\dots,u_k \in \{0,1\},\ t_1,\dots,t_k \in \{1,\dots,b-1\},\ i_1 > \cdots > i_k \in \{0,\dots,m-1\} \Big\}.$$

Plainly, from Lemma 1, $\mathcal{Z}(k,q)$ can be expressed as

$$\mathcal{Z}(k,q) = \{0\} \cup \Big\{ \big((-1)^{u_1} t_1 b^{i_1} + \cdots + (-1)^{u_j} t_j b^{i_j}\big) \bmod q \colon\ 0 < j \leqslant k,\ u_1,\dots,u_j \in \{0,1\},\ t_1,\dots,t_j \in \{1,\dots,b-1\},\ i_1 > \cdots > i_j \in \{0,\dots,m-1\} \Big\}.$$

An upper bound for the cardinality of $\mathcal{Z}(k,q)$ is $\min\{q, \sum_{j=0}^{k}\binom{m}{j}(2(b-1))^j\}$, as for each $0 \leqslant j \leqslant k$, there are $\binom{m}{j}$ ways to choose j pairwise distinct $i_1,\dots,i_j$, and $(2(b-1))^j$ ways to choose $u_1,\dots,u_j$ and $t_1,\dots,t_j$.

In order for the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$ to give few false positives, the size of $\mathcal{Z}(k,q)$ must be *small*, which, working modulo an arbitrary number q, may not be true. The main idea of our approach is to choose $q = b^w - 1$, where $w < m$ is a natural number large enough, according to a few complexity considerations.

Notice that, arithmetic modulo numbers of the form $2^w - 1$ (called Mersenne numbers) is used in various applications, like digital systems based on residue number system, or cryptography, therefore, efficient VLSI circuit architectures for addition and multiplication modulo $2^w - 1$ have been proposed over the years (see, e.g., the discussion in [27], and the references therein). Notice also that, in general, the usage of q of the form $2^w - 1$ is *not* suggested when exact search is performed.

The following lemma shows that the choice $q = b^w - 1$ guarantees that $\mathcal{Z}(k, q)$ has a small cardinality.

Lemma 2. *Given* $1 \leqslant w < m$,

$$\begin{aligned}\mathcal{Z}\big(k, b^w - 1\big) = \{0\} \cup \Big\{ &\big((-1)^{u_1} t_1 b^{i_1} + \cdots + (-1)^{u_j} t_j b^{i_j}\big) \bmod \big(b^w - 1\big): \\ &0 < j \leqslant k,\ u_1, \ldots, u_j \in \{0, 1\},\ t_1, \ldots, t_j \in \{1, \ldots, b-1\}, \\ &i_1 > \cdots > i_j \in \{0, \ldots, w-1\}\Big\}.\end{aligned}$$

Proof. To simplify notation in this proof, we let $\mathcal{Z}^*(k, b^w - 1)$ stand for the set on the right-hand side of the equality claimed above. Hence, we have to show that $\mathcal{Z}(k, b^w - 1) = \mathcal{Z}^*(k, b^w - 1)$.

Since the modulo operation is linear, we have

$$\begin{aligned}b^s \bmod \big(b^w - 1\big) &= b^{w(s \operatorname{div} w) + s \bmod w} \bmod \big(b^w - 1\big) \\ &= \big(\big(b^w\big)^{s \operatorname{div} w} b^{s \bmod w}\big) \bmod \big(b^w - 1\big) \\ &= \big(\big(\big(b^w\big)^{s \operatorname{div} w} \bmod \big(b^w - 1\big)\big)\big(b^{s \bmod w} \bmod \big(b^w - 1\big)\big)\big) \bmod \big(b^w - 1\big) \\ &= \big(\big(\big(b^w \bmod \big(b^w - 1\big)\big)^{s \operatorname{div} w} \bmod \big(b^w - 1\big)\big) b^{s \bmod w}\big) \bmod \big(b^w - 1\big) \\ &= b^{s \bmod w}.\end{aligned}$$

This entails that

$$\begin{aligned}\mathcal{Z}\big(k, b^w - 1\big) = \{0\} \cup \Big\{ &\big((-1)^{u_1} t_1 b^{i_1} + \cdots + (-1)^{u_j} t_j b^{i_j}\big) \bmod \big(b^w - 1\big): \\ &0 < j \leqslant k,\ u_1, \ldots, u_j \in \{0, 1\},\ t_1, \ldots, t_j \in \{1, \ldots, b-1\}, \\ &i_1, \ldots, i_j \in \{0, \ldots, w-1\}\Big\} \\ =_{\text{def}}\ & R(k).\end{aligned}$$

Clearly, $\mathcal{Z}^*(k, b^w - 1) \subseteq R(k)$ (notice that the difference between $\mathcal{Z}^*(k, b^w - 1)$ and $R(k)$ is that the indices $i_1, \ldots, i_j$ are not required to be distinct in $R(k)$). To prove the opposite inclusion, we will proceed by induction on $k < m$. When $k = 1$, the claim is true. Assuming that the claim is true for $k < m - 1$, we will show that it also holds for $k + 1$.

For the sake of clarity, and without loss of generality, we assume onwards that $b = 2$. For any $x \in R(k+1) \setminus R(k)$, where $x = ((-1)^{u_1} 2^{i_1} + \cdots + (-1)^{u_k} 2^{i_k} + (-1)^{u_{k+1}} 2^{i_{k+1}}) \bmod (2^w - 1)$, we have to show that $x \in \mathcal{Z}^*(k+1, 2^w - 1)$. We have that x can be written as

$$\big(\big((-1)^{u_1} 2^{i_1} + \cdots + (-1)^{u_k} 2^{i_k}\big) \bmod \big(2^w - 1\big) + (-1)^{u_{k+1}} 2^{i_{k+1}} \bmod \big(2^w - 1\big)\big) \bmod \big(2^w - 1\big).$$

From the inductive hypothesis, the first of the above two terms belongs to $\mathcal{Z}^*(k, 2^w - 1)$, and hence equal to some $((-1)^{v_1} 2^{h_1} + \cdots + (-1)^{v_j} 2^{h_j}) \bmod (2^w - 1)$, where $0 \leqslant j \leqslant k$, $v_1, \ldots, v_j \in \{0, 1\}$, and $h_1 > \cdots > h_j \in \{0, \ldots, w-1\}$. Moreover, $(-1)^{u_{k+1}} 2^{i_{k+1}} \bmod (2^w - 1) = (-1)^{u_{k+1}} 2^{i_{k+1} \bmod w} \bmod (2^w - 1)$.

If $(i_{k+1} \bmod w) \notin \{h_1, \ldots, h_j\}$, then the claim is true. Otherwise, suppose that $i_{k+1} \bmod w$ equals some h_J, and that x becomes

$$\begin{aligned}&\big((-1)^{v_1} 2^{h_1} + \cdots + (-1)^{v_{J-1}} 2^{h_{J-1}} + \big((-1)^{v_J} + (-1)^{u_{k+1}}\big) 2^{h_J} \\ &\quad + (-1)^{v_{J+1}} 2^{h_{J+1}} + \cdots + (-1)^{v_j} 2^{h_j}\big) \bmod \big(2^w - 1\big).\end{aligned}$$

If $u_{k+1} = 1 - v_J$, then $x \in \mathcal{Z}^*(k-1, 2^w - 1) \subset \mathcal{Z}^*(k+1, 2^w - 1)$ and the claim is true. Otherwise, assume that $u_{k+1} = v_J = 0$ (the case $u_{k+1} = v_J = 1$ is entirely analogous). Then, x is

$$\big((-1)^{v_1} 2^{h_1} + \cdots + (-1)^{v_{J-1}} 2^{h_{J-1}} + 2^{h_J + 1} + (-1)^{v_{J+1}} 2^{h_{J+1}} + \cdots + (-1)^{v_j} 2^{h_j}\big) \bmod \big(2^w - 1\big),$$

which belongs to $R(k) = \mathcal{Z}^*(k, 2^w - 1) \subset \mathcal{Z}^*(k+1, 2^w - 1)$, completing thus the proof. □

Hence, $|\mathcal{Z}(k, b^w - 1)|$ is at most $\sum_{j=0}^{k} \binom{w}{j} (2(b-1))^j$, as for each $0 \leqslant j \leqslant k$, there are $\binom{w}{j}$ ways to choose j pairwise distinct $i_1, \ldots, i_j$, and $(2(b-1))^j$ ways to choose $u_1, \ldots, u_j$ and $t_1, \ldots, t_j$.

Algorithm 1: String matching with k mismatches

Input: $T = T[0]T[1]\dots T[n-1]$, $P = P[0]P[1]\dots P[m-1]$, both over the alphabet $\Sigma = \{0, 1, \dots, b-1\}$, number of mismatches k ($0 \leqslant k < m$) and word length w.
Output: All pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$ and $d_H(P, T_{(s)}) \leqslant k$.

```
1  q ← b^w − 1;
2  h ← b^{m−1 mod w};
3  Z ← GenerateZ(k, q);
4  Solutions ← ∅;
5  p̂ ← t̂ ← 0;
6  for i ← 0 to m − 1 do
7      p̂ ← (b · p̂ + P[i]) mod q;
8      t̂ ← (b · t̂ + T[i]) mod q;
9  if (p̂ − t̂) mod q ∈ Z then
10     if d_H(P, T_(0)) ⩽ k then
11         Solutions ← Solutions ∪ {⟨0, d_H(P, T_(0))⟩};
12 for s ← 1 to n − m do
13     t̂ ← (b · (t̂ − h · T[s − 1]) + T[s + m − 1]) mod q;
14     if (p̂ − t̂) mod q ∈ Z then
15         if d_H(P, T_(s)) ⩽ k then
16             Solutions ← Solutions ∪ {⟨s, d_H(P, T_(s))⟩};
17 return Solutions;
```

Table 1
The average number of false positives returned by the heuristic test $(\hat{p} - \hat{t}_s) \bmod q \in \mathcal{Z}(k, 4^w - 1)$, when $\Sigma = \{0, 1, 2, 3\}$, n =4G, and $w = 15$ (32-bit architecture) and $w = 31$ (64-bit architecture).

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
f. p. on 32 bits	3.73	339	13,079	279,959	3,662,224	30,549,760
f. p. on 64 bits	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	2.4

Onwards, we suppose to work modulo $q = b^w - 1$, without explicitly mentioning it. Observe also that, as a result of Lemma 2, the set $\mathcal{Z}(k, b^w - 1)$ depends only on b, w and k.

The generalized algorithm (shown as Algorithm 1) works in a similar manner as the Rabin–Karp algorithm [5]. It starts by setting $q = b^w - 1$, $s = 0$, and by computing $\hat{p} = p \bmod q$ and $\hat{t}_{(0)} = t_{(0)} \bmod q$, using Horner's rule and bringing into play the linearity of the modulo operation. Then, for each $0 \leqslant s \leqslant n - m$ it checks whether $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$. If yes, it performs a character-by-character comparison of P and $T_{(s)}$. When incrementing s, the value $\hat{t}_{(s)}$ can be computed in constant time, as follows. For all $0 \leqslant s < n - m$, we have $\hat{t}_{(s+1)} = b \cdot (t_{(s)} - b^{m-1} T[s]) + T[s + m]$. Working modulo q, this equation becomes $\hat{t}_{(s+1)} = (b \cdot (\hat{t}_{(s)} - (b^{m-1} \bmod q) T[s]) + T[s + m]) \bmod q$. If we let $h =_{\text{def}} b^{m-1} \bmod q = b^{(m-1) \bmod w}$, we get $\hat{t}_{(s+1)} = (b \cdot (\hat{t}_{(s)} - h \cdot T[s]) + T[s + m]) \bmod q$.

In Algorithm 1 we assume that procedure GenerateZ(k, q) generates the set $\mathcal{Z}(k, b^w - 1)$, as expressed in Lemma 2.

In order to evaluate the expected complexity of the string matching phase of Algorithm 1, we follow the formalism of [28, Ch. 32.2]. We have to compute the time $c(q)$ the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}$ on lines 9 and 14 takes, and the average number of false positives produced by it. If we denote by $p(q)$ the probability that at a specific shift $0 \leqslant s \leqslant n - m$ this test will produce a false positive, we can estimate the number of false positives as $n \cdot p(q)$. Considering ν to be the number of occurrences of P in T with at most k mismatches, the expected complexity is

$$\mathcal{O}\big(n \cdot c(q) + \big(m \cdot \nu + m \cdot n \cdot p(q)\big)\big).$$

In many applications ν is small (i.e., $\mathcal{O}(1)$) and if we choose q such that $n \cdot p(q) \leqslant 1$, then the expected complexity becomes $\mathcal{O}(n \cdot c(q) + m)$. The only values of $t_{(s)}$ for which $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$, but $d_H(P, T_{(s)}) > k$ are of the form $p + z + j \cdot q$, where $z \in \mathcal{Z}(k, q)$ and $0 \leqslant j \leqslant \lfloor b^m / q \rfloor$. As we have at most $\lfloor b^m / q \rfloor |\mathcal{Z}(k, q)|$ such values, and there are at most b^m possible values for $t_{(s)}$, the probability that at a specific shift s, the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$ produces a false positive is $p(q) \leqslant \frac{|\mathcal{Z}(k,q)|}{q}$, under the assumption that the operation $\bmod(b^w - 1)$ uniformly distributes numbers in the interval $[0 \dots q - 1]$ (for example when $b^w - 1$ is a prime number).

Therefore, to attain the desired time complexity, one has to choose $q = b^w - 1$ such that $b \cdot q$ fits into a processor word and such that $q \geqslant n|\mathcal{Z}(k, q)|$.

Working on a 32-bit processor, with strings over the alphabet $\{0, 1, 2, 3\}$, limits w to 15, therefore, if n or k are large enough, a flurry of false positives are due to appear. If we use a 64-bit architecture, w is limited to 31, and hence the number of false positives drastically decreases. These numbers are computed in Table 1.

We choose to implement the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$ by generating the set $\mathcal{Z}(k, q)$ before-hand, in time $\mathcal{O}(|\mathcal{Z}(k, q)|)$. The data structure storing it can be an ordered array, with search complexity $c(q) = \mathcal{O}(\log |\mathcal{Z}(k, q)|)$. A data

structure more appropriate for unsuccessful queries, as we expect most of them to be, is a trie, with worst case search time $c(q) = \mathcal{O}(w)$. However, due to better memory locality, a hash table with collisions resolved by chaining is preferred. Under the assumption of simple uniform hashing and using $\mathcal{O}(\alpha)$ memory, the average search complexity becomes $c(q) = \mathcal{O}(1 + |\mathcal{Z}(k,q)|/\alpha)$.

If one agrees to use an additional amount $\mathcal{O}(q)$ of memory, then $\mathcal{Z}(k,q)$ can be simply stored as a direct-address table $\mathcal{Z}[0 \dots q-1]$, where $\mathcal{Z}[z] = 1$ iff $z \in \mathcal{Z}(k,q)$, and thus $c(q) = \mathcal{O}(1)$.

Theorem 1. *Algorithm 1 solves the k-mismatch problem; if $q = b^w - 1 \geqslant n|\mathcal{Z}(k,q)|$, and if $c(q)$ denotes the complexity of testing membership in $\mathcal{Z}(k,q)$, its expected search complexity is $\mathcal{O}(n \cdot c(q) + m + |\mathcal{Z}(k,q)|)$.*

3. A randomized numerical string aligner

3.1. An exact string aligner

An exact string aligner is given a text T, of length n, and a collection $\mathcal{P}$ of patterns, and is required to find all exact occurrences of P in T, for every $P \in \mathcal{P}$. In what follows, we will assume that all the patterns are of the same length m. A naive approach is to iteratively apply the Rabin–Karp algorithm for each $P \in \mathcal{P}$, with an overall time complexity of $\mathcal{O}((n+m)|\mathcal{P}|)$. Another solution is to compute before-hand the fingerprints of all the patterns in $\mathcal{P}$ and store them in an appropriate data structure, in which every $\hat{t}_{(s)}$ $(0 \leqslant s \leqslant n-m)$ is searched for. If a matching fingerprint value is found, the corresponding pattern is compared with $T_{(s)}$. This approach takes time $\mathcal{O}(m|\mathcal{P}| + n)$ if a hash table is used to store the fingerprints of the patterns (as done e.g. in [26]), and time $\mathcal{O}(m|\mathcal{P}| + n \log |\mathcal{P}|)$, if they are stored as an ordered array.

The approach we choose to follow does not rely on a data structure on the patterns, but on the text. This can be pre-processed in time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$, by constructing the index

$$\mathcal{T} = \left\{ \langle \hat{t}_{(s)}, s \rangle \colon 0 \leqslant s \leqslant n-m \right\}.$$

The shifts s in T which *may* be exact occurrences of a $P \in \mathcal{P}$ correspond to those pairs $\langle \hat{p}, s \rangle \in \mathcal{T}$. The set $\mathcal{T}$ can be stored is a way similar to a hash by chaining. We use an array indexed by numbers from 0 to $q-1$, having, for all $0 \leqslant r \leqslant q-1$, $\mathcal{T}[r] = \{s_1, \dots, s_l\}$ iff for all $1 \leqslant i \leqslant l$, $\hat{t}_{(s_i)} = r$. Note that when doing exact alignment, q can be chosen to be $\Theta(n)$, according to the complexity analysis of Section 2. This exact aligner has average time complexity $\mathcal{O}(n + m|\mathcal{P}|)$.

3.2. A k-mismatch string aligner

In order to construct a string aligner that solves the best k-mismatch problem, Algorithm 1 can be adapted to use the index $\mathcal{T}$ over the text, by reverting from 'verification' back to 'generation'. For every $P \in \mathcal{P}$, we are interested in finding all the shifts s in T which *may* be occurrences of P with at most k mismatches. They correspond to those pairs $\langle \hat{t}_{(s)}, s \rangle \in \mathcal{T}$ such that $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$. Using linearity of the modulo operation, we thus iteratively search in $\mathcal{T}$ all numbers $(\hat{p} - z) \bmod q$, for every $z \in \mathcal{Z}(k,q)$. For all shifts s such that $\langle (\hat{p} - z) \bmod q, s \rangle \in \mathcal{T}$, we check that indeed $d_H(P, T_{(s)}) \leqslant k$. The average complexity of a search for a pattern is thus $\mathcal{O}(m + |\mathcal{Z}(k,q)|)$, amounting to a total complexity of $\mathcal{O}(n + (m + |\mathcal{Z}(k,q)|)|\mathcal{P}|)$.

However, the larger w is, the lower the probability of a false positive is, but the larger $|\mathcal{Z}(k,q)|$ gets, and vice-versa. We can remediate to this problem by a rather standard use (in this field) of the pigeonhole principle.

Definition 2. Given a string $P = P[0]P[1] \dots P[m-1]$ and a positive integer $1 \leqslant t \leqslant m$, for every $0 \leqslant i < t$, we denote by $P_{\lfloor m/t \rfloor}(i)$ its substring $P[i\lfloor m/t \rfloor] \dots P[(i+1)\lfloor m/t \rfloor - 1]$ and call it the ith *block* of P.

Note that the t blocks of a string P do not overlap, a crucial property for the following lemma to hold.

Lemma 3. *Let T be a text, $P = P[0]P[1] \dots P[m-1]$ be a pattern, and t a positive integer, $1 \leqslant t \leqslant m$. If P occurs in T with at most k mismatches, then there is at least one block $P_{\lfloor m/t \rfloor}(i)$ of P that occurs in T with at most $\lfloor k/t \rfloor$ mismatches.*

Accordingly, instead of searching for an entire pattern P with at most k mismatches, we can perform t searches for all of the blocks of P, each with at most $\lfloor k/t \rfloor$ mismatches. Each occurrence of a block $P_{\lfloor m/t \rfloor}(i)$ $(0 \leqslant i < t)$ of P in T, with shift s, is an indication that P may occur in T with shift $s - i\lfloor m/t \rfloor$. As we are interested in finding the best occurrences of P in T, we will keep the smallest number of mismatches at which an occurrence of P has been found so far in a variable *best_k*. In this way, each block of the pattern is searched with at most $\lfloor best\_k/t \rfloor$ mismatches. The pseudo-code of the resulting procedure is given as Algorithm 2.

Procedure PreProcessText builds the index over the text discussed in Section 3.1 by storing all fingerprints of length l of the text. We assume that procedure GenerateZ(k,q) returns an array containing the elements of the set $\mathcal{Z}(k,q)$, ordered in the following way: for all $0 \leqslant i < k$ the elements of $\mathcal{Z}(i,q)$ are placed before the elements of $\mathcal{Z}(i+1,q) \setminus \mathcal{Z}(i,q)$.

Algorithm 2: The randomized Numerical Aligner (**rNA**)

Input: Text $T = T[0]T[1]\dots T[n-1]$, a collection $\mathcal{P}$ of patterns of length m, all over the alphabet $\Sigma = \{0, 1, \dots, b-1\}$, number k of mismatches ($0 \leqslant k < m$), the number t of blocks in which the patterns get divided ($1 \leqslant t \leqslant k+1$), and word length w.

Output: For all $P \in \mathcal{P}$, all pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$, $d_H(P, T_{(s)}) \leqslant k$ and for all $0 \leqslant s' \leqslant n-m$, it holds that $d_H(P, T_{(s)}) \leqslant d_H(P, T_{(s')})$.

```
1  procedure SearchPattern(P)
2      for i ← 0 to t − 1 do
3          p̂_l(i) ← 0;
4          for j ← i · l to (i + 1) · l − 1 do
5              p̂_l(i) ← (b · p̂_l(i) + P[j]) mod q;
6      Solution ← ∅; best_k ← k;
7      exact_occurrence ← false; j ← 0;
8      while j < |Z(⌊best_k/t⌋, q)| do                          //for every witness Z[j]
9          i ← 0;
10         while i ⩽ t − 1 and (¬exact_occurrence) do              //for every block i
11             foreach s ∈ indexT[(p̂_l(i) − Z[j]) mod q] do          //for all shifts
12                 if s − i · l ⩾ 0 and d_H(P, T_(s−i·l)) ⩽ best_k then
13                     if d_H(P, T_(s−i·l)) < best_k then
14                         best_k ← d_H(P, T_(s−i·l));
15                         Solution ← ∅;
16                     Solution ← Solution ∪ {⟨s − i · l, best_k⟩};
17             if best_k = 0 then exact_occurrence ← true;
18         j ← j + 1;
19     print Solution;
20 end
21 q ← b^w − 1; l ← ⌊m/t⌋;                                   //compute q and the block length t
22 indexT ←PreProcessText(T, b, l, q);
23 Z ←GenerateZ(k, q);
24 foreach P ∈ P do
25     SearchPattern(P);
```

The procedure SearchPattern(P) starts by dividing the pattern in t blocks, each of length $l = \lfloor m/t \rfloor$. For each block $P_l(i)$ ($0 \leqslant i < t$), its fingerprint $\hat{p}_l(i)$ is computed employing Horner's rule and the linearity of the modulo operation (lines 2–5). The variable *best_k* stores the smallest distance at which an occurrence of P has been found so far, while *exact_occurrence* indicates whether an *exact* occurrence has been found in the text.

For each index j ($0 \leqslant j < |\mathcal{Z}(\lfloor best\_k/t \rfloor, q)|$), we iteratively search in the text every *block* $P_l(i)$ ($0 \leqslant i < t$), with at most $\lfloor best\_k/t \rfloor$ mismatches (line 10). Every such shift s where the block i may occur is an indication that the *pattern* may occur at shift $s - i \cdot l$ with at most *best_k* mismatches (if, of course, $s - i \cdot l \geqslant 0$).

If this is indeed the case (line 12), we have to check whether the current occurrence is at distance strictly smaller than *best_k* (line 13). If so, the variable *best_k* is updated with the current distance, and all the shifts s stored so far in the set Solution are discarded. Anyhow, the current shift s together with *best_k* are added to Solution. In other words, at every step of the computation, the set Solution stores occurrences only at distance *best_k*.

Lastly, in line 17 we implement the following optimization: if the pattern occurs in an exact manner in the text, then the first block does as well. Since this block will indicate all exact occurrences, searching the remaining blocks of P brings no additional information. Therefore, we set *exact_occurrence* to true, stopping the search (this is true because *best_k* was changed to 0, hence the loop in line 8 is no longer executed).

4. Extensions and connections

This section attempts to set up a formal framework capturing the properties which make the approach exposed in Sections 2 and 3 computationally efficient. We will argue that fingerprinting using the mod operation is no singular example, and that the xor operator is an alternative. Both these fingerprinting methods have been implemented in our tool, as it will be explained in Section 6. We end this section by a discussion on an envisaged connection between coding theory and our solution to the best k-mismatch problem.

For the clarity of the presentation, we again assume that the input alphabet Σ is $\{0, 1\}$. Formally, we have defined a hash function $h_{\mathsf{mod}} : \{0,1\}^m \to \{0, \dots, 2^w - 2\}$, where $h_{\mathsf{mod}}(X) = \hat{x} = x \bmod (2^w - 1)$. From Lemmas 1 and 2, it holds that for all $X, Y \in \{0,1\}^m$,

$$d_H(X, Y) \leqslant k \quad \Rightarrow \quad h_{\mathsf{mod}}\big(h_{\mathsf{mod}}(X) - h_{\mathsf{mod}}(Y)\big) \in \mathcal{Z}\big(k, 2^w - 1\big),$$

and that the size of $\mathcal{Z}(k, 2^w - 1)$ is bounded by $\sum_{j=0}^{k} 2^j \binom{w}{j}$. First, observe that w must be chosen so that $w > k$. If this were not the case, $\mathcal{Z}(k, 2^w - 1)$ would equal $\{0, \dots, 2^w - 2\}$ and hence the test $h_{\mathsf{mod}}(x) \in \mathcal{Z}(k, 2^w - 1)$ would always hold.

This implies that $|\mathcal{Z}(k, 2^w - 1)| > 2^k$. Moreover, recall that w was chosen with the additional property that $2^w - 1 \geqslant n|\mathcal{Z}(k, 2^w - 1)| > n2^k$. For texts of practically significant lengths, we may assume that $n \geqslant 2^k$, and hence that $w \geqslant 2k$. This implies that $\binom{w}{j} \leqslant \binom{w}{k}$, for all $0 \leqslant j \leqslant k$. Therefore, $|\mathcal{Z}(k, 2^w - 1)| \leqslant 2^{k+1}\binom{w}{k} = \mathcal{O}(2^k w^k)$.

By the following definition, we would like to capture the main features of this hash function.

Definition 3. We say that a hash function $h : \{0,1\}^m \to \{0,1\}^w$ is *Hamming-aware* if there exist

- a function $\mathsf{compare}_h : \{0,1\}^w \times \{0,1\}^w \to \{0,1\}^w$, computable in time $\mathcal{O}(w)$,
- a constant c and a set $N_h \subseteq \{0,1\}^w$ of size $\mathcal{O}(c^k w^k)$,

with the properties

(i) $d_H(X,Y) \leqslant k \Rightarrow \mathsf{compare}_h(h(X), h(Y)) \in N_h$;
(ii) for all $X \in \{0,1\}^m$, $|\{\xi \in \{0,1\}^w \mid \mathsf{compare}_h(h(X), \xi) \in N_h\}| = \mathcal{O}(c^k w^k)$;
(iii) for all $X \in \{0,1\}^m$, the set $\{\xi \in \{0,1\}^w \mid \mathsf{compare}_h(h(X), \xi) \in N_h\}$ can be listed in time $\mathcal{O}(c^k w^k)$.

Imposing condition (i) means that, given X, we want to be able to characterize (i.e., recognize) those hash values corresponding to strings Y at Hamming distance at most k from X. Notice that the set N_h does not depend on the choice of X or Y, but only on the hash function. Testing membership in N_h can be done, as argued in Section 2, in time $\mathcal{O}(1)$ if N_h is stored as a direct address table of size 2^w, or in time $\mathcal{O}(k \log cw)$ if it is stored as an ordered array, by means of a binary search. Condition (ii) requires that the aforementioned characterization be as precise as possible, while condition (iii) is useful when doing string alignment using a text index: we also want to be able to *enumerate* all those hash values corresponding to strings at Hamming distance at most k from X.

It is clear that by taking $\mathsf{compare}_{h_{\mathsf{mod}}}(h_X, h_Y) = h_{\mathsf{mod}}(h_X - h_Y)$ and $N_{h_{\mathsf{mod}}} = \mathcal{Z}(k, 2^w - 1)$ conditions (i)–(iii) are fulfilled, and hence h_{mod} is a Hamming-aware hash function.

We will now examine other Hamming-aware functions. For this reason, let us come back to h_{mod} and see an alternative interpretation of the operation $\bmod(2^w - 1)$. Suppose, for simplicity, that $m = rw$, $r \geqslant 2$. Given $X \in \{0,1\}^m$, $h_{\mathsf{mod}}(X)$ can be computed as the sum of the numerical values of the r blocks (recall Definition 2) of length w of X, modulo $2^w - 1$. Indeed, since $2^{sw} \bmod (2^w - 1) = 1$, for all $s > 0$, we have

$$h_{\mathsf{mod}}(X) = \big(x_w(0) + 2^w x_w(1) + \cdots + 2^{(r-1)w} x_w(r-1)\big) \bmod \big(2^w - 1\big),$$
$$h_{\mathsf{mod}}(X) = \big(x_w(0) + x_w(1) + \cdots + x_w(r-1)\big) \bmod \big(2^w - 1\big).$$

Example 1. Still working under the assumption that $m = rw$, and denoting by $\otimes$ the xor operator between two binary strings, let us define $h_{\mathsf{xor}} : \{0,1\}^m \to \{0,1\}^w$ as

$$h_{\mathsf{xor}}(X) = X_w(0) \otimes X_w(1) \otimes \cdots \otimes X_w(r-1).$$

Clearly, it holds that for all strings $X, Y \in \{0,1\}^m$,

$$d_H(X,Y) \leqslant k \quad \Rightarrow \quad d_H\big(h_{\mathsf{xor}}(X), h_{\mathsf{xor}}(Y)\big) \leqslant k.$$

To see that h_{xor} is a Hamming-aware function, take $\mathsf{compare}_{h_{\mathsf{xor}}}(h_X, h_Y) = h_X \otimes h_Y$, and $N_{h_{\mathsf{xor}}} = \{\delta \in \{0,1\}^w \mid$ the number of 1's in δ is at most $k\}$. To see that conditions (i)–(iii) are satisfied, observe that $|N_{h_{\mathsf{xor}}}| = \sum_{j=0}^{k} \binom{w}{j} = \mathcal{O}(kw^k)$ and that the set $\{\xi \in \{0,1\}^w \mid h(X) \otimes \xi \in N_{h_{\mathsf{xor}}}\}$ is actually $\{h(X) \otimes \delta \mid \delta \in N_{h_{\mathsf{xor}}}\}$, hence of size $\mathcal{O}(kw^k)$.

At this point, it is not hard to devise more elaborate hash functions.

Example 2. Given m and w, we can randomly choose w numbers from the set $\{0, \ldots, m-1\}$. Then, we can randomly choose another w numbers of the remaining ones, and so on, r times. Given a string X, let us denote by $\tilde{X}_w(0)$ the substring of X formed by taking the bits on the first such randomly chosen positions, by $\tilde{X}_w(1)$ the substring of X formed by the bits on the second randomly chosen positions, and so on. Then, we can define the Hamming-aware function $\tilde{h}_{\mathsf{xor}} : \{0,1\}^m \to \{0,1\}^w$ analogously, as

$$\tilde{h}_{\mathsf{xor}} = \tilde{X}_w(0) \otimes \tilde{X}_w(1) \otimes \cdots \otimes \tilde{X}_w(r-1).$$

Example 3. Given a binary string X, let

$$\mathsf{majority}(X) = \begin{cases} 0 & \text{if at least } \lfloor |X|/2 \rfloor + 1 \text{ characters of } X \text{ are } 0, \\ 1 & \text{otherwise.} \end{cases}$$

Define now $h_{\mathsf{maj}}(X)$ to be the string $\xi \in \{0,1\}^w$ such that for all $0 \leqslant i < w$, the ith character of ξ is

$$\xi[i] = \mathsf{majority}\big(X[i]X[i+w]\ldots X\big[i+(r-1)w\big]\big).$$

As before, it holds that for all strings $X, Y \in \{0,1\}^m$, $d_H(X,Y) \leqslant k \Rightarrow d_H(h_{\mathsf{maj}}(X), h_{\mathsf{maj}}(Y)) \leqslant k$ and we analogously obtain that h_{maj} is a Hamming-aware hash function.

Let us now look at the approach taken in building the string aligner, from a geometric perspective. Our hash function partitions the space of all substrings $T_{(s)}$ of length m of T in such a way that it is easy to identify the classes of this partition which may contain elements of the Hamming ball of center P and radius k. Indeed, every substring $T_{(s)}$ of length m of T is mapped to the class of this partition indexed by $h_{\mathsf{mod}}(T_{(s)})$. Given P, the classes of this partition on T that may intersect the Hamming ball of center P and radius k are indexed by $h_{\mathsf{mod}}(h_{\mathsf{mod}}(P) - z)$, for all $z \in \mathcal{Z}(k, 2^w - 1)$. Consequently, the remaining classes can be safely disregarded.

In order for this approach to be computationally effective, we need to ensure that all the classes have similar size and to filter-out as many classes as possible. The former requirement is implicitly fulfilled by our Hamming-aware hash functions as we assume a quasi-uniform distribution over the text. The latter requirement is satisfied by the property of being Hamming-aware, which indicates only $O(c^k w^k)$ classes that need to be checked.

However, one may also require, at least from a formal point of view, that the classes of the partition induced by the hash function be Hamming balls. However, in our case, no property of the kind 'if two elements are in the same class, i.e., have the same hash value, then they are at Hamming distance at most εk' holds, where ε is an appropriate constant.

We think that the problem of constructing a Hamming-aware hash function such that the partition it induces consists of Hamming balls can be the bridge between our approach and tools from channel coding theory. There, given m, the goal is to choose a set of codewords of length m such that the Hamming balls of radius k, centered around these codewords do not overlap and cover the space of all words of length m as much as possible. Hence, one may want to take the set of these Hamming balls as the partition of the space of the text. However, we consider crucial having the property that the Hamming ball of radius k and centered around P intersects at most $O(c^k w^k)$ classes of the partition, and having an efficient way of enumerating all shifts of the text inside them.

5. Implementation details

As pointed out in the introduction, bioinformatics is the main field where string alignment tools are used. A new generation of machines (called next generation *sequencers*) can process molecules of DNA and produce as output an enormous quantity of sequences (called *reads*). For example, the latest available Illumina sequencer, HiSeq2000, can produce 200 billion characters (called *bases*), grouped in reads of length 100 (their length is expected to grow to 150 bases in the near future). Our implementation of the algorithm exposed in Section 3.2 solves this practical problem, focusing mainly on Illumina reads.

The biological setting is the following. The DNA alphabet is composed of the four characters A, C, G, T. In practice both the text (called the *reference sequence*) and the reads can contain a certain number of ambiguous characters (caused by gaps in the assembly or by sequencing errors). As customary in this field, we will identify them with the character N. Given a string X, let us define its *reverse complement* to be the string $\overline{X}$, such that for all $i \in \{0,\ldots,|X|-1\}$, $\overline{X}[i] = \mathsf{cmpl}(X[|X|-i-1])$, where $\mathsf{cmpl} : \{A,C,G,T,N\} \to \{A,C,G,T,N\}$, and $\mathsf{cmpl}(A) = T$, $\mathsf{cmpl}(C) = G$, $\mathsf{cmpl}(G) = C$, $\mathsf{cmpl}(T) = A$ and $\mathsf{cmpl}(N) = N$. The DNA is a double-stranded molecule: each strand, comprised of a sugar-phosphate backbone and attached bases, is connected to a complementary strand. Since the sequencers, in general, cannot indicate the strand from which each read has been taken, given a read P, we must align both P and $\overline{P}$. Therefore, in this particular instance of the best k-mismatch problem we are interested in finding the occurrences at minimum Hamming distance of either P or $\overline{P}$. In a real setting, the reference sequence is divided into *chromosomes* or into *scaffolds*, hence the input text consists of a database of *genomic* sequences $\mathcal{D} = \{T^1, T^2, \ldots, T^u\}$.

5.1. Data structures

Given $\mathcal{D} = \{T^1, T^2, \ldots, T^u\}$, we build the string $T = T^1\$T^2\$\cdots\$T^u$, where \$ is a new character used as delimiter. Once a match is found inside T, its global coordinate is converted into a local coordinate inside a chromosome/scaffold, by doing a binary search on a lookup table.

The main data structure behind **rNA** is the hash table *indexT*, implemented with two arrays, H and V. The former has length $q+1$ and contains pointers to V, while the latter has length equal to $|T|$ and contains pointers to the text. In position $H[r]$ we memorize the *rank* of the fingerprint r, i.e., the number of fingerprints less than r present in T. From position $V[H[i]]$ to position $V[H[i+1]-1]$ we store the shifts of T having fingerprint r. After having computed the fingerprint $\hat{p}$ of a read P, we perform the test in line 13 of Algorithm 2 for all these shifts.

Note that, by scanning the text two times, arrays H and V can be computed in-place, without any supplementary memory. Moreover, both them and T need to be in RAM during search phase. Hence, we need $4 \cdot q + 4 \cdot |T| + |T|$ bytes, if 4 bytes are used for each pointer, and each character of T is stored as one byte. In the case of the grapevine genome, whose length is approximately 480 M, we need approximately 6.3 GB, while with the human genome, whose length is

approximatively 3.2 G we need 20 GB if q is set to $4^{15} - 1$. In order to tackle genomes of longer lengths, two solutions arise, depending on the amount of memory one is disposed to use. The first one is to use a larger q, which would require 8 bytes for each pointer (since we need to pass to a 64-bit architecture). The second solution, which is currently under development, is a distributed implementation which allows to spread the computation over several nodes of a cluster.

Most of the other available tools for the alignment of short reads are based on sophisticated data structures closely related to Suffix Trees [10] and Suffix Arrays [13]. Based on Burrows–Wheeler transformation, FM-indexes [29] are the key point of tools like BWA and SOAP2. The main advantage of such data structure is the possibility to work with a compressed index without loosing performance. BWA [7] claims to use between 2.3 GB and 3 GB during query time when aligning reads on the Human Genome. We are currently exploring the possibility to use some of the concepts illustrated in [29] in order to, possibly, compress the text and the hash table.

5.2. Ambiguous bases

All the N characters inside the reads are treated as mismatches. During the fingerprint computation we simply generate a random character for each of them. In a similar way we treat ambiguous characters in the text: in the construction phase we randomly choose a non-ambiguous base, while in the alignment phase we treat them as mismatches.

BWA uses the same approach for the reads, but it substitutes every ambiguous character of the text with a randomly generated base. SOAP2 extends this approach even to the reads, with the risk of returning false positives.

5.3. Read checking and quality trimming

Reads are usually given in the FASTQ format. A FASTQ file uses 4 lines per read. The first line is the header, which begins with the character @ and contains the read name. The second line is the read itself. The third line is a comment line while the last one is a string of the same length of the read which stores the quality of the read.

Low quality bases are likely to be reading mistakes and are usually concentrated at the beginning and at the end of the read (as a consequence of the chemical reactions used in order to read DNA). We developed a routine similar to the one implemented by the CLCbio Workbench [30] in order to check the read quality. We first trim the low quality bases at the beginning and at the end of the read. If after this process the remaining read has length and average quality higher than two predefined thresholds, the read is aligned, otherwise it is discarded.

5.4. Paired-end mapping

Most of the sequencers are able to produce reads in pairs, by reading two sequences at a fixed distance and with a known orientation. Among the many advantages of this additional information, let us only mention that it can be of help in identifying structural variations [31].

When aligning such a pair, **rNA** first returns the best occurrences of each read of the pair, and then sorts them according to their positions. At this point, a linear scan is performed in order to find a possible alignment of the two reads that satisfies both the distance and the orientation constraints.

5.5. Output

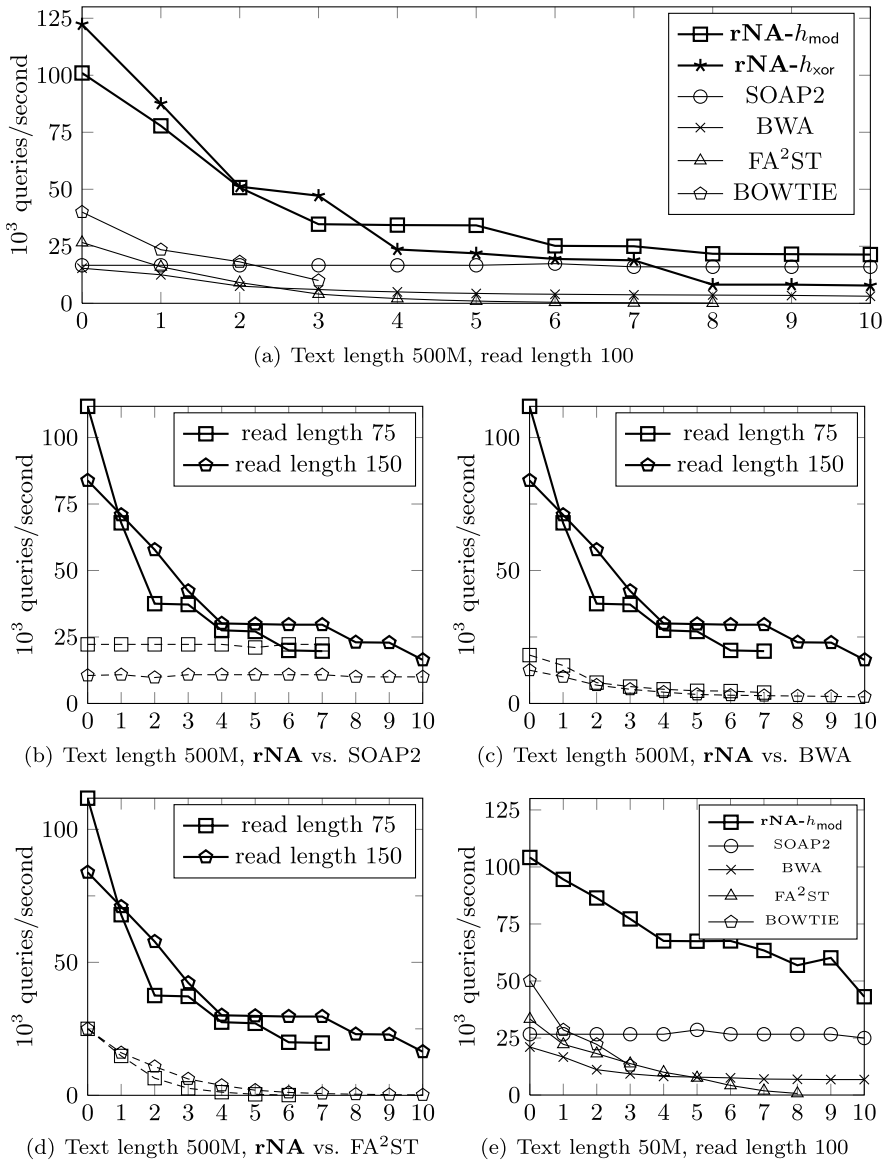
Output is provided in the widely used SAM format [32], making **rNA** compatible with a large number of tools for post-processing alignments.

5.6. Multi-thread

Alignment is a highly parallelizable routine. Presently, **rNA** can be used on a multi-core machine: every CPU reads a chunk of 262,144 reads and aligns them against the reference. Every time a CPU finishes the alignment phase, it writes the result in the output file and reads the next chunk of reads. A distributed version of **rNA** making use of multiple machines is under development, with the aim at searching inside larger genomes, and at further speed improvements.

6. Experimental results

Our tool was compared against SOAP2 [6], BWA [7], BOWTIE [8], and FA^{2}ST [33]. The last tool implements Suffix Arrays and relies on the idea behind Lemma 3, where t, the number of blocks in which the pattern gets divided, is always chosen to be $k+1$. Like **rNA**, it is the only aligner, to the best of our knowledge, that solves the *best k-mismatch problem* in an accurate manner. We tested **rNA** using as fingerprint functions both h_{mod} an h_{xor}. Since, experimentally, h_{mod} performs better on



(a) Text length 500M, read length 100

(b) Text length 500M, **rNA** vs. SOAP2

(c) Text length 500M, **rNA** vs. BWA

(d) Text length 500M, **rNA** vs. FA^{2}ST

(e) Text length 50M, read length 100

Fig. 1. The number of errors is represented on the X-axis, while the Y-axis indicates the number of reads processed per second. Fig. 1(a) compares the read throughput of **rNA-**h_{mod}, **rNA-**h_{xor}, SOAP2, BWA, FA^{2}ST, and BOWTIE when aligning reads of length 100 against a reference sequence of length 500 M. Figs. 1(b)–1(d) compare the performance of **rNA** (continuous lines) with SOAP2, BWA, and FA^{2}ST (dashed lines) when varying the read length (75, 100, 150) and the number of errors. Fig. 1(e) compares the read throughput of the algorithms when aligning reads of length 100 against a reference sequence of length 50 M.

texts of length 500 M, we choose it as the default implementation. In the ongoing, we identify the two implementation with **rNA-**h_{mod}, and **rNA-**h_{xor}, respectively, while with **rNA** we identify the default version, **rNA-**h_{mod}.

In order to achieve high performance, many of the currently available aligners sometimes sacrifice correctness over speed, by skipping a small number of occurrences of a read. For example, BWA [7] and SOAP2 [6] search only the first $l \leqslant m$ characters of the read (usually called *seed*) in the reference text with at most $d < k$ mismatches, and for each such occurrence, they check that the entire pattern occurs with at most k mismatches (generally, the default values of l and d are 30 and 2, respectively, but in most cases they can also be set by the user). This heuristic, usually called *seed&extend*, is based on the biological assumption that reading mistakes are less frequent in the first bases. Despite this, our goal is to solve the *best k-mismatch problem*, hence we search each block of the pattern, as described in Algorithm 2. Our choice obviously lowers the performance of **rNA**, but, as we will soon show, it increases its accuracy. Note that, for the very same reason, the performance of **rNA** is highly dependent on whether a read occurs or not in the text. **rNA** and BOWTIE allow only mismatches while BWA and SOAP2 can, optionally, allow small insertions/deletions (indels) between the pattern and the reference. In order to achieve a fair comparison we run these two last tools without allowing indels.

Table 2
Comparison between **rNA**, SOAP2, BOWTIE, and BWA on two real datasets. We used the grapevine reference genome PN40024 of length 480 M to align 33,675,544 reads of length 100 belonging to the Sangiovese grapevine variety and the human genome reference hg18 of length 3.2 G to align 24,177,454 reads of length 100 belonging to a Korean adult male. All reads are aligned with at most 7 mismatches. Tools SOAP2 and BOWTIE do not offer the trimming option. All the tools have been used allowing 8 threads.

Program	Grapevine		Human	
	Time	Aligned (%)	Time	Aligned (%)
rNA noTRIM	21 m	78.80	23 h 15 m	53.45
rNA TRIM	1 h 20 m	83.88	19 h 23 m	58.93
SOAP2 noTRIM	1 h 24 m	57.36	1 h 54 m	38.18
BOWTIE noTRIM	30 m	57.36	41 m	42.61
BWA noTRIM	1 h 05 m	70.94	4 h 38 m	51.27
BWA TRIM	55 m	76.87	1 h 10 m	55.04

The tests were performed over a machine running Linux 2.6.24, on two quad-core Intel Xeon 3 GHz processors with 32 GB of RAM.

We performed our benchmarks on two datasets: a simulated dataset and a real dataset. The former dataset has been used to show the maximum achievable performance with the tested tools. For this purpose all the experiments in the simulated dataset were run using a single CPU. The latter dataset has been used to highlight the performance and the accuracy in real-case scenarios. In order to show the time required in real situations, we ran all the experiments always allowing 8 threads.

The simulated dataset was constructed by extracting from the grapevine genome 5 sequences of sizes 50 K, 500 K, 5 M, 50 M, and 500 M. In order to avoid the extra time needed to convert from global to local coordinates, these sequences consist of a single scaffold. From each such reference, we extracted 400,000 reads of length m ($m \in \{75, 100, 150\}$), with an average error rate of 2%. These assumptions are similar to the technical specifications of the Illumina sequencer. During the alignment of such sequences, we disabled the quality check and the trimming heuristics for the tools with such options. For all the possible combinations of tool, text length and read length, several experiments where done varying the input parameters and only the best result was considered. Fig. 1(a) compares the 5 tools on reference length 500 M and query length 100. When allowing less than 4 mismatches, **rNA** greatly outperforms all other tools with both the implemented fingerprints. If the number of allowed mismatches increases, the only tool that achieves comparable results is SOAP2 (whose performance tends to be constant). It is important to stress again the fact that, while SOAP2 uses the *seed*&*extend* heuristic, **rNA** solves the *best k-mismatch problem*. From Figs. 1(a) and 1(b) we can have a complete comparison between **rNA** and SOAP2 on a reference of length 500 M bases. When the ratio between the number of mismatches and the length of the read is low, **rNA** is significantly faster than SOAP2. In particular, for read length 150 and at most 10 mismatches, **rNA** is always better than SOAP2. Fig. 1(e) shows the performance on a 50M text. Other results for references of length 50 K, 500 K, 5 M, and 50 M are presented in Appendix A.

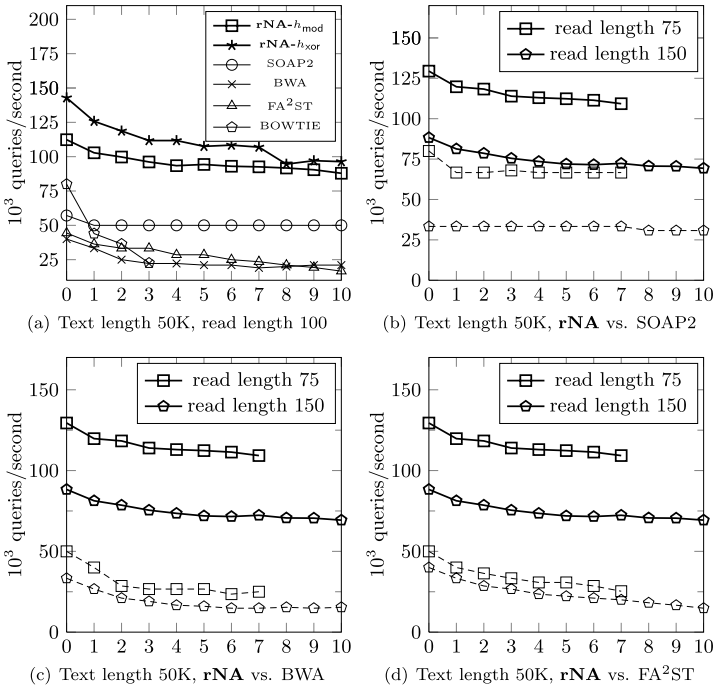
The real dataset was formed by two reference sequences: the grapevine genome of length 480 M composed of 33 sequences (20 Chromosomes and 13 unordered scaffolds) and the human genome of length 3.2 G composed of 23 Chromosomes. We aligned against the first reference a real Illumina lane composed of 33,675,544 reads of length 100 belonging to the grapevine variety Sangiovese (experiment performed at IGA, Institute of Applied Genomics), while against the human reference hg18 we aligned another real lane composed of 24,177,454 reads of length 100 belonging to a Korean adult male (downloaded from NCBI–SRA experiment SRX011536). The results of these experiments are summarized in Table 2. In both cases we aligned each read with a maximum of 7 mismatches, allowing in this way 7% of difference in both cases. In the case of **rNA** and BWA, we used the tools with and without the trimming option. When evaluating the output produced by SOAP2, we noticed that, despite having set the maximum number of total allowed mismatches to 7, there were a certain number of alignments with a larger number of errors. For the sake of our experimental comparison, we decided to discard them.

In the case of the grapevine genome, we can see that **rNA** without the trimming option is the fastest tool, and also the one able to align the highest number of reads. When the trimming option is turned on, the performance of **rNA** decreases, but in return it aligns 83.88% of the reads. In the human genome case, **rNA** is between 5 and 20 times slower than the other tools. Nevertheless, **rNA** still aligns the highest number of reads.

Acknowledgments

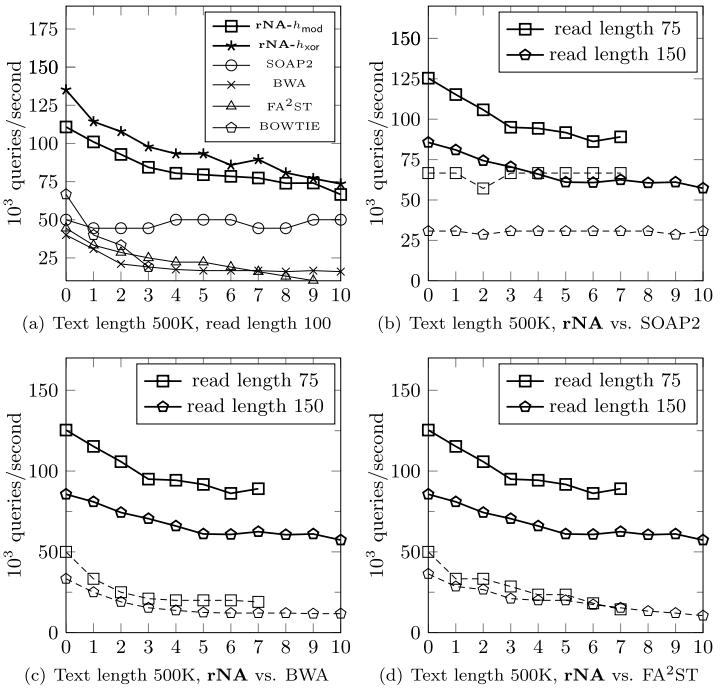
We thank Cristian Del Fabbro and Simone Scalabrin for the help provided during **rNA**'s implementation. Cristian developed important modules like Input/Output, paired read handling, and SAM format support. Simone provided essential feedback which guided the implementation towards a practical tool for the bioinformatics community.

Appendix A. More experimental results



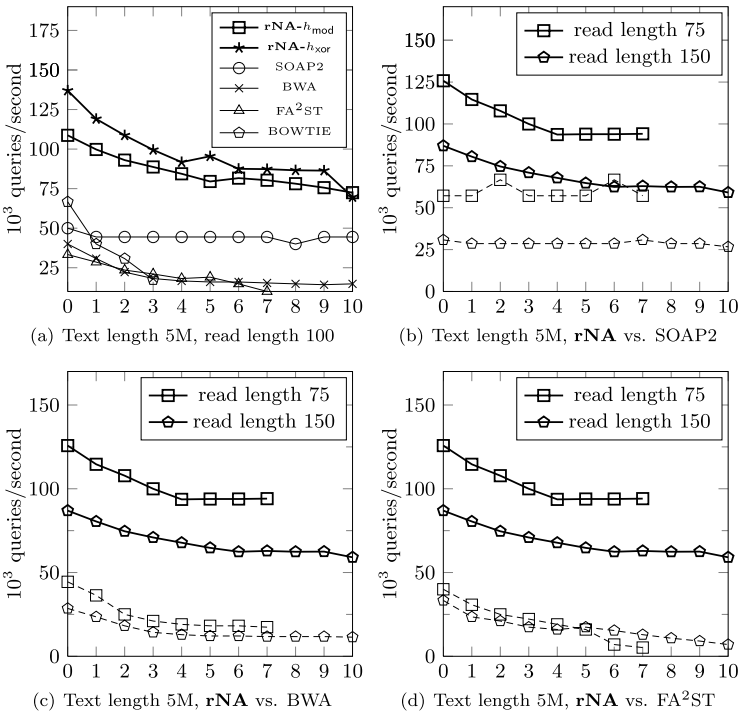
(a) Text length 50K, read length 100
(b) Text length 50K, **rNA** vs. SOAP2
(c) Text length 50K, **rNA** vs. BWA
(d) Text length 50K, **rNA** vs. FA^{2}ST

Fig. 2. The number of errors is represented on the X-axis, while the Y-axis indicates the number of reads processed per second. Fig. 2(a) compares the read throughput of **rNA-**h_{mod}, **rNA-**h_{xor}, SOAP2, BWA, FA^{2}ST and, BOWTIE when aligning reads of length 100 against a reference sequence of length 50 K. Figs. 2(b)–2(d) compare the performance of **rNA** (continuous lines) with SOAP2, BWA, and FA^{2}ST (dashed lines) when varying the read length (75, 150) and the number of errors.



(a) Text length 500K, read length 100
(b) Text length 500K, **rNA** vs. SOAP2
(c) Text length 500K, **rNA** vs. BWA
(d) Text length 500K, **rNA** vs. FA^{2}ST

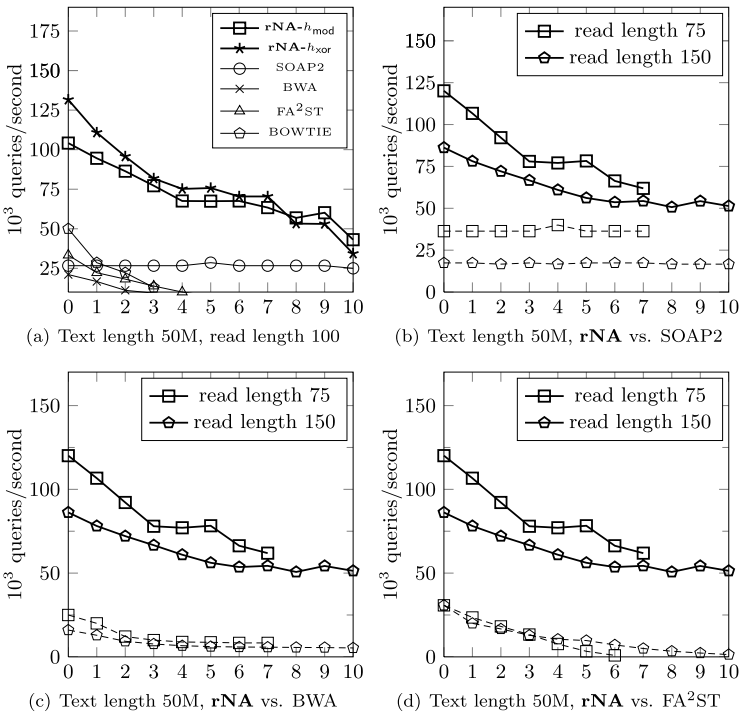
Fig. 3. The number of errors is represented on the X-axis, while the Y-axis indicates the number of reads processed per second. Fig. 3(a) compares the read throughput of **rNA-**h_{mod}, **rNA-**h_{xor}, SOAP2, BWA, FA^{2}ST and, BOWTIE when aligning reads of length 100 against a reference sequence of length 500 K. Figs. 3(b)–3(d) compare the performance of **rNA** (continuous lines) with SOAP2, BWA, and FA^{2}ST (dashed lines) when varying the read length (75, 150) and the number of errors.



(a) Text length 5M, read length 100 (b) Text length 5M, **rNA** vs. SOAP2

(c) Text length 5M, **rNA** vs. BWA (d) Text length 5M, **rNA** vs. FA^{2}ST

Fig. 4. The number of errors is represented on the X-axis, while the Y-axis indicates the number of reads processed per second. Fig. 4(a) compares the read throughput of **rNA-**h_{mod}, **rNA-**h_{xor}, SOAP2, BWA, FA^{2}ST and, BOWTIE when aligning reads of length 100 against a reference sequence of length 5 M. Figs. 4(b)–4(d) compare the performance of **rNA** (continuous lines) with SOAP2, BWA, and FA^{2}ST (dashed lines) when varying the read length (75, 150) and the number of errors.



(a) Text length 50M, read length 100 (b) Text length 50M, **rNA** vs. SOAP2

(c) Text length 50M, **rNA** vs. BWA (d) Text length 50M, **rNA** vs. FA^{2}ST

Fig. 5. The number of errors is represented on the X-axis, while the Y-axis indicates the number of reads processed per second. Fig. 5(a) compares the read throughput of **rNA-**h_{mod}, **rNA-**h_{xor}, SOAP2, BWA, FA^{2}ST and, BOWTIE when aligning reads of length 100 against a reference sequence of length 50 M. Figs. 5(b)–5(d) compare the performance of **rNA** (continuous lines) with SOAP2, BWA, and FA^{2}ST (dashed lines) when varying the read length (75, 150) and the number of errors.

References

[1] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, D.J. Lipman, Gapped BLAST and PSI–BLAST: a new generation of protein database search programs, Nucleic Acids Res. 25 (17) (1997) 3389–3402.
[2] W.J. Kent, BLAT—The BLAST-like alignment tool, Genome Res. 12 (4) (2002) 656–664.
[3] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[4] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. ACM 20 (10) (1977) 762–772.
[5] R. Karp, M. Rabin, Efficient randomized pattern-matching algorithms, IBM J. Res. Develop. 31 (2) (1987) 249–260.
[6] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, J. Wang, SOAP2: an improved ultrafast tool for short read alignment, Bioinformatics 25 (15) (2009) 1966–1967.
[7] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows–Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.
[8] B. Langmead, C. Trapnell, M. Pop, S. Salzberg, Ultrafast and memory-efficient alignment of short DNA sequences to the human genome, Genome Biology 10 (3) (2009) R25.
[9] D.S. Horner, G. Pavesi, T. Castrignano, P.D. De Meo, S. Liuni, M. Sammeth, E. Picardi, G. Pesole, Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing, Briefings Bioinformatics (2009), bbp046+.
[10] P. Weiner, Linear pattern matching algorithms, in: Proc. 14th Ann. Symp. on Switching and Automata Theory (SWAT 1973), 1973, pp. 1–11.
[11] A. Apostolico, The myriad virtues of sub-word trees, Combinatorics on Words 112 (1985) 85–96.
[12] E. Ukkonen, Approximate string matching over suffix trees, in: Proc. 4th Ann. Symp. on Combinatorial Pattern Matching, 1993, pp. 228–242.
[13] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, in: SODA'90: Proc. 1st Ann. ACM–SIAM Symp. on Discrete Algorithms, 1990, pp. 319–327.
[14] P. Ferragina, String algorithms and data structures, CoRR abs/0801.2378.
[15] K. Abrahamson, Generalized string matching, SIAM J. Comput. 16 (6) (1987) 1039–1051.
[16] G.M. Landau, U. Vishkin, Efficient string matching in the presence of errors, in: Proc. 26th IEEE Symp. on Foundations of Computer Science, 1985, pp. 126–136.
[17] G.M. Landau, U. Vishkin, Efficient string matching with k mismatches, Theoret. Comput. Sci. 43 (1986) 239–249.
[18] Z. Galil, R. Giancarlo, Improved string matching with k mismatches, SIGACT News 17 (4) (1986) 52–54.
[19] L. Salmela, J. Tarhio, P. Kalsi, Approximate Boyer–Moore string matching for small alphabets, Algorithmica 58 (3) (2010) 591–609.
[20] Z. Liu, X. Chen, J. Borneman, T. Jiang, A fast algorithm for approximate string matching on gene sequences, in: Proc. 16th Ann. Symp. on Combinatorial Pattern Matching, in: Lecture Notes in Comput. Sci., vol. 3537, 2005, pp. 79–90.
[21] W.I. Chang, T.G. Marr, Approximate string matching and local similarity, in: Proc. 5th Ann. Symp. on Combinatorial Pattern Matching, 1994, pp. 259–273.
[22] A. Amir, M. Lewenstein, E. Porat, Faster algorithms for string matching with k mismatches, J. Algorithms 50 (2004) 257–275.
[23] P. Jokinen, E. Ukkonen, Two algorithms for approximate string matching in static texts, in: Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science, in: Lecture Notes in Comput. Sci., vol. 520, 1991, pp. 240–248.
[24] T.N.D. Huynh, W.-K. Hon, T.-W. Lam, W.-K. Sung, Approximate string matching using compressed suffix arrays, Theoret. Comput. Sci. 352 (1) (2006) 240–249.
[25] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM J. Comput. 35 (2) (2005) 378–407.
[26] R. Muth, U. Manber, Approximate multiple string search, in: Proc. 7th Ann. Symp. on Combinatorial Pattern Matching, 1996, pp. 75–86.
[27] R. Zimmermann, Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication, in: IEEE Symposium on Computer Arithmetic, IEEE Computer Society, 1999, pp. 158–167.
[28] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, MIT Press/McGraw–Hill Book Company, 2001.
[29] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. 41st Ann. Symp. on Foundations of Computer Science, 2000, pp. 390–398.
[30] http://www.clcbio.com/files/usermanuals/CLC_Genomics_Workbench_User_Manual.pdf, pp. 405–406.
[31] S. Lee, F. Hormozdiari, C. Alkan, M. Brudno, MoDIL: detecting small indels from clone-end sequencing with mixtures of distributions, Nat. Methods 6 (7) (2009) 473–474.
[32] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, 1000 genome project data processing subgroup, the sequence alignment/map format and SAMtools, Bioinformatics 25 (16) (2009) 2078–2079.
[33] C. Del Fabbro, Repeated sequences in bioinformatics: assembly, annotation and alignments, PhD thesis, University of Udine, 2010.