Approximate search of short patterns with high error rates using the 01 0 lossless seeds

Christophe Vroland, Mikael Salson, Sébastien Bini, Hélène Touzet

Approximate search of short patterns with high error rates using the 01*0 lossless seeds

Christophe Vroland, Mikaël Salson, Sébastien Bini, Hélène Touzet[*]

CRIStAL (UMR CNRS 9189, University of Lille), 59650 Villeneuve d'Ascq, France

INRIA Lille-Nord Europe, France

Abstract

Approximate pattern matching is an important computational problem that has a wide range of applications in computational biology and in information retrieval. However, searching a short pattern in a text with high error rates (10%–20%) under the Levenshtein distance is a task for which few efficient solutions exist. Here we address this problem by introducing a new type of seeds: the 01*0 seeds. These seeds are made of two exact parts separated by parts with exactly one error. We show that those seeds are lossless, and we apply them to two filtration algorithms for two popular applications, one where a compressed index is built on the text and another one where the patterns are indexed. We also demonstrate experimentally the advantages of our approach compared to alternative methods implementing other types of seeds. This work opens the way to the design of more efficient and more sensitive text algorithms.

Keywords: approximate string matching, lossless approximate seeds, Levenshtein distance, computational biology

1. Introduction

We consider the approximate string matching problem, where an error between a pattern P and a text T is measured by the Levenshtein distance: an error is either a substitution, an insertion, or a deletion. The first approach to solve this problem was to perform dynamic programming with an $O(|T| \times |P|)$ algorithm [1]. This kind of approach has been improved by using bit-parallelism or finite automaton (for instance [2]). However it is still not practical to search large databases of sequences or a full genome.

Rather than traversing all the sequences, it is possible to start by filtering candidate regions which show a high-enough similarity between portions of the text and of the pattern. Such portions are called *seeds*. Their occurrences must

[*]Corresponding author: `helene.touzet@univ-lille1.fr`

then be extended to check if the pattern occurs within k errors. This *seed-and-extend* approach is now widely used to solve the approximate pattern matching problem. The seed definition can be very diverse and depending on the type of application (size of the alphabet, rate of errors, ...) the preferred seeds may differ. The most widely known seed is the contiguous seed, which consists in a substring that must be shared between the pattern and the text. For instance it is used in the Blast software for homology search in bioinformatics [3]. Depending on the error rate and on the length of the contiguous seed, those seeds may be *lossy*, meaning that the query algorithm is a heuristic that can lead to false negatives. The classical pigeonhole principle avoids this pitfall by splitting the pattern in exactly $k+1$ parts, which are searched independently as instances of contiguous seeds. This is an example of *lossless* seed, that never discards an actual occurrence. A recent example of this method, using a modified Burrows-Wheeler transform is shown in [4]. However the more errors we allow, the shorter the parts will be and therefore the more potential occurrences we may have. Thus the filtration efficiency becomes lower with higher values of k. This limitation can be partially overcome by the usage of spaced seeds [5] or subset seeds [6] that offer a better selectivity/specificity trade off than contiguous seeds [7, 8]. One main drawback is that those seeds tolerate substitutions only. They do not improve the sensitivity over contiguous seeds with insertions and deletions. Attempts to generalize spaced seeds to the full Levenshtein distance were presented in [9] for example, with a strong restriction on the total number of errors.

Other approaches are *hybrid* approaches that combine seed filtering and exact search. The pattern is split in non-overlapping parts that can be searched with a given number of errors. Each method differs in the number of pieces in the patterns and on the maximal number of errors in each piece. For instance, Navarro and Baeza-Yates [10] designed a hybrid method which consists of splitting the pattern P in $j = (|P| + k)/\log_\sigma |T|$ parts, where σ is the alphabet size, and searching these parts with $\lfloor \frac{k}{j} \rfloor$ errors. This approach has also been used with a LZ-index or an FM-index [11]. In [12], Kärkkäinen and Na introduded a filtering method based on *suffix filters*, where the pattern is cut in $k+1$ factors and the set of seeds is the set of all suffixes with strong matches. In general, hybrid approaches are more flexible than contiguous or spaced seeds as they tolerate any kind of errors, without restriction on the positions. However, their filtration efficiency tends to decrease as the number of errors increases, or they involve factorizations where some parts have to be searched with multiple errors.

Despite this large body of research on seeding techniques for the approximate string matching problem, we believe that there is still room for improvement for small patterns with high error rates (10–20%). In this article we will present a new hybrid method where the pattern will be split in $k+2$ non-overlapping parts, some of them being searched without error, while others are searched with exactly one error. This approach results in the definition of a new kind of lossless seeds, called 01*0 seeds, that can accomodate a large number of insertions and deletions[1]. In Section 2, we will first introduce the theoretical framework of these new seeds. Once the type of seeds has been chosen, there are different ways of using them. The ability to index them or to search them efficiently in a text index is an important criterion. To illustrate this, we will present two real-life applications of those seeds in a bioinformatics context. In Section 3, we will present an algorithm for the approximate pattern matching problem in large genomic sequences. In this application, 01^*0 seeds are used in conjunction with a compressed full-text index, namely the FM-index [14]. In Section 4, we will address the complementary problem of comparing a large collection of patterns against a relatively short DNA sequence such as a gene. For doing so, we will explain how to index the seeds themselves. Finally, in Section 5, we will discuss the application of the 01^*0 seeds to the k-mismatch problem and compare them to spaced seeds.

2. The 01*0 seeds

2.1. Definition

Let Σ be a finite alphabet of size σ. Given two strings U and V of Σ^*, define $lev(U, V)$ to be the Levenshtein distance between U and V. This is the minimum number of operations needed to transform U into V, where the only allowed operations are substitution of a single character and deletion or insertion of a single character. Each such operation is also called an *error*. From now on, we assume that a given natural number k corresponds to a maximum number of errors.

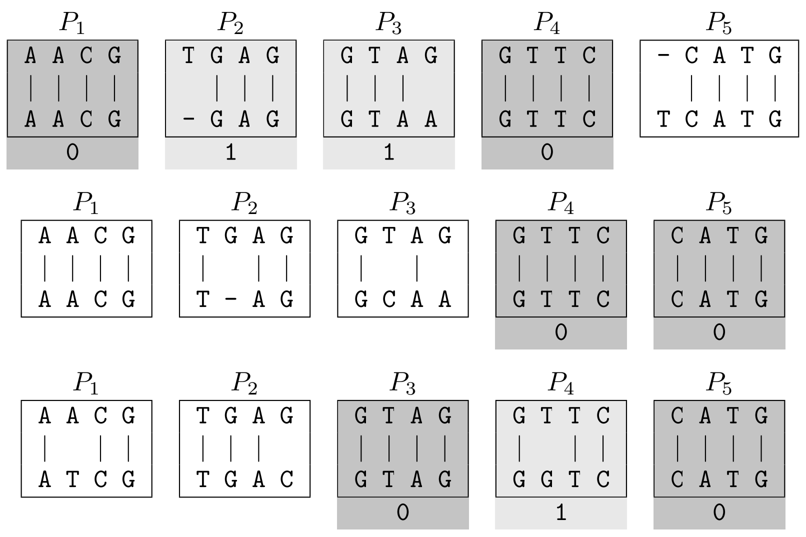
Let P be a pattern of length m over Σ. Using the pigeonhole principle, it is well-known that if P is partitioned into $k + 1$ parts, then every string U, such that $lev(P, U) \leq k$, contains at least one of these parts as a substring. The parts do not need to be of the same length. Similarly, if P is partitioned into $k + 2$ parts, denoted $P_1, \ldots, P_{k+2}$, then U should contain at least two disjoint parts of P. The following lemma allows to push the analysis further. It is indeed possible to request that these two parts be separated by parts with exactly one error.

Lemma 1. *Let U be a string of Σ^* such that $\mathrm{lev}(P, U) \leq k$. Then there exist i, j, $1 \leq i < j \leq k + 2$, and $U_1, \ldots, U_{j-i-1}$ of Σ^* such that*

1. *$P_i U_1 \ldots U_{j-i-1} P_j$ is a substring of U, and*
2. *When $j > i + 1$, for each ℓ, $1 \leq \ell \leq j - i - 1$, $\mathrm{lev}(P_{i+\ell-1}, U_\ell) = 1$.*

Example 1. Assume $k = 3$. Given the pattern $P =$ `AACGTGAGGTAGGTTCCATG` of length 20, we partition it into five parts, of equal length: $P_1 =$ `AACG`, $P_2 =$ `TGAG`, $P_3 =$ `GTAG`, $P_4 =$ `GTTC`, and $P_5 =$ `CATG`. Consider three strings whose Levenshtein distance with P is 3: `AACGGAGGTAAGTTCTCATG`, `AACGTAGGCAAGTTCCATG`

[1]Part of this work has already been published as an extended abstract in the proceedings of the conference Iwoca 2014 [13].



P_1 P_2 P_3 P_4 P_5
A A C G | T G A G | G T A G | G T T C | - C A T G
A A C G | - G A G | G T A A | G T T C | T C A T G
0 1 1 0
A A C G | T G A G | G T A G | G T T C | C A T G
A A C G | T - A G | G C A A | G T T C | C A T G
0 0
A A C G | T G A G | G T A G | G T T C | C A T G
A T C G | T G A C | G T A G | G G T C | C A T G
0 1 0

Figure 1: Application of Lemma 1 for sequences of Example 1. The pattern `AACGTGAGGTAGGTTCCATG` is on the top of each alignment. The two parts with no error (written P_i and P_j in the Lemma) are highlighted in dark grey, and the parts with 1 error surrounded by P_i and P_j are highlighted in light grey.

and `ATCGTGACGTAGGGTCCATG`. For each string, we show in Figure 1 the parts that fulfil the conditions of Lemma 1.

We rephrase Lemma 1 as a pure counting problem and establish its proof.

Lemma 2. *Let k be a natural number. Assume you have $k+2$ containers numbered from 1 to $k+2$, and y tokens with $0 \leq y \leq k$. Then there exists two containers i and j, $1 \leq i < j \leq k+2$, such that*

1. *containers i and j are empty, and*
2. *for each ℓ, $i < \ell < j$, the container ℓ contains exactly one token.*

Proof. Let μ be the number of containers with at least two tokens, called the multiple containers. In total there are at least $\mu + 2$ empty containers. The 2 comes from the fact that we have k tokens and $k+2$ containers. Therefore there are at least $\mu + 1$ pairs of empty containers i and j such that each container between i and j is non empty. Each of the μ multiple containers can lie between one distinct pair of empty containers, breaking the property for μ pairs of empty containers. Stated otherwise, we have at most μ pairs of empty containers with one intervening multiple container. This leaves us with $\mu + 1 - \mu = 1$ pair satisfying the property that the intervening containers (if any) have exactly one token. $\square$

As a consequence of Lemma 1, we can design a seeding framework for lossless filtering for the approximate pattern matching problem with k errors. To this end, we introduce some terminology that will be used in the remainder of the paper.

Definition 1. Let $P = P_1 \dots P_{k+2}$ be a pattern divided into $k+2$ parts. Then the 01*0 *seed* for P and k is the regular expression

$$\cup_{i=1}^{k+1} \cup_{j=i+1}^{k+2} \ P_i \ lev^1(P_{i+1}) \dots lev^1(P_{j-1}) \ P_j$$

where $lev^1(u)$ denotes the set of strings whose Levenshtein distance with u is 1. A *subseed* is the regular expression associated to a given pair (i,j):

$$P_i \ lev^1(P_{i+1}) \dots lev^1(P_{j-1}) \ P_j.$$

An *instance* of a seed, or of a subseed, is a string u of Σ^* which is recognized by the seed, or the subseed. Given a text T on Σ^*, an *occurrence* of the seed for P is a substring of T which is an instance of the seed. Therefore, an occurrence is characterized by its start position and its end position in T.

2.2. Filtration efficiency

The filtration efficiency is the primary criterion used to evaluate the performance of a seed. The thorough analysis of the average case for the 01*0 seeds, such as what is done in [15] for example, is beyond the scope of this paper. Having a formula for the expected numbers of occurences of subseed instances for a given pattern is also a non trivial task, since this formula would be highly dependent of the structure of the pattern and of its internal repeats. We leave it as an open question for future work. Here we present a simple lemma, that gives an upper bound for the probability of an occurrence of a subseed.

Lemma 3. *If the text T is a sequence of independent and identically distributed characters of Σ, then the probability to have an occurrence of the subseed $P_i \dots P_j$ at a given position of the text is smaller than*

$$\frac{1}{\sigma^{p_i}} \ \times \ L(p_{i+1}) \ \times \ \dots \ \times \ L(p_{j-1}) \ \times \ \frac{1}{\sigma^{p_j}}$$

where $p_i, \dots, p_j$ are the lengths of the parts $P_i, \dots, P_j$ and
$L(p) = \frac{p(\sigma-1)}{\sigma^p} + \frac{p}{\sigma^{p-1}} + \frac{\sigma(p+1)}{\sigma^{p+1}}$.

Proof. By Definition 1, $P_i \ lev^1(P_{i+1}) \dots lev^1(P_{j-1}) \ P_j$ gives the set of all possible instances of the subseed. The probability to observe P_i and P_j is $\frac{1}{\sigma^{p_i}}$ and $\frac{1}{\sigma^{p_j}}$ respectively. For each part P of length p in $\{P_{i+1}, \dots, P_{j-1}\}$, $\frac{p(\sigma-1)}{\sigma^p} + \frac{p}{\sigma^{p-1}} + \frac{\sigma(p+1)}{\sigma^{p+1}}$ bounds the probability to observe $lev^1(P)$. Indeed, $\frac{p(1-\sigma)}{\sigma^p}$ is an upper bound for the probability of the set of words with one mismatch from P, $\frac{p}{\sigma^{p-1}}$ is an upper bound for the probability of the set of words with one deletion from P, and $\frac{\sigma(p+1)}{\sigma^{p+1}}$ is an upper bound for the probability of the set of words with one insertion from P. □

This lemma can be used to obtain an upper bound on the average number of occurrences found in a random text, by summing up all the probabilities of all subseeds. For example, for $k = 3$, a pattern of length $m = 20$ divided in five equal parts of length 4, a text of size 10^8 over the DNA aphabet $\{A, C, G, T\}$ and the result is 6,747 occurrences.

Complementary to this lemma, we generated an independent and identically distributed random sequence of length 10^8 over the DNA alphabet as well as 100 patterns of length 20. We then searched for the 01*0 seeds for $k = 3$. For each pattern, we counted the total number of occurrences of the seeds in the text, including overlapping occurrences. The distribution is plotted in Table 1. The average number of observed occurrences per pattern is 6,665 (98.7% of our estimated upper bound).

For the purpose of comparison, we also estimated the filtration efficiency of several other types of seeds, for the same text and the same collection of patterns. First, we considered the classical pigeonhole principle, which guarantees lossless filtration. The pattern is split in $k + 1 = 4$ non-overlapping parts of length 5. In Table 1, we call it the $k + 1$ *pigeonhole* seed. The average number of occurrences is 385,651. We then analysed a $k + 2$ version of the pigeonhole principle: the pattern is divided in $k + 2 = 5$ non-overlapping parts of length 4, and two parts must match exactly. In a first case, we require that any two exact parts are separated by a distance compatible with at most k errors: the number of positions between two exact parts P_i and P_j $(i < j)$ should range from $p_{i+1} + \cdots + p_{j-1} - k$ to $p_{i+1} + \cdots + p_{j-1} + k$. In a second case, we consider a weakened form of Lemma 1: all parts in-between the two exact parts should contain exactly one error. As a consequence, the number of positions between P_i and P_j ranges from $p_{i+1}+\cdots+p_{j-1}-(j-i-1)$ to $p_{i+1}+\cdots+p_{j-1}+(j-i-1)$. We refer to this seed as the *bounded* $k + 2$ *pigeonhole* seed. It can be seen as an intermediate seed between the $k+2$ pigeonhole seed and the 01*0 seeds and allows us to better understand the behavior of 01*0 seeds. In those two cases, the average number of occurrences is respectively of 86,145 and 44,746 (see Table 1, $k+2$ pigeonhole and bounded $k+2$ pigeonhole). These plots show that moving from $k+1$ parts to $k+2$ parts allows to improve the filtration efficiency by a factor greater than 4. The additional constraint on distances beween parts further decreases the number of occurrences by almost half. Finally, the 01*0 seed offers a new gain of a factor 6.7. Overall, this is more than 57 times better than the filtering rate achieved with the $k + 1$ pigeonhole principle. As a last case, we chose to divide the pattern in only three parts, of lengths 6, 7, and 7. Contrary to the four preceding seeds, this seed is lossy since it allows for some false negatives. The average number of occurrences is 36,207 (see Table 1, lossy seed). Interestingly, this value is close to the bounded $k + 2$ pigeonhole seed, but the former maintains full sensitivity. It is far from being as selective as 01*0 seed.

These empirical measurements show that the 01*0 seed is significantly more selective than all other seeds presented in Table 1. It provides orders of magnitude more efficient filtering. Of course, this higher selectivity comes at the price of some additional work to locate seeds in the text. Each seed is composed

	01*0 seed	$k+1$ pigeonhole	$k+2$ pigeonhole	bounded $k+2$ pigeonhole	lossy seed
average	6,598	385,651	86,145	44,747	36,207
std. dev.	106	4,221	2,078	967	419

Table 1: Distribution of the number of occurrences for five different seeds on a dataset constituted by 100 patterns of size $m = 20$ in a random sequence of length 10^8 on the DNA alphabet. The 01*0 seed and the three pigeonhole seeds are designed for $k = 3$ errors. So for the 01*0 seed and the $k+2$ pigeonhole seeds, the pattern is divided in 5 parts of length 4. For the $k+1$ pigeonhole seed, the pattern is divided in 4 parts of length 5. For the lossy seed, each pattern is split in 3 parts of lengths 6, 7, and 7, respectively. There are on average 26.85 occurrences of the whole pattern within 3 errors.

of $\frac{(k+1)(k+2)}{2}$ subseeds, and some parts of the subseed should be searched with errors. This task, however, is facilitated by the regular structure of the 01*0 seeds. Firstly, each part of the subseed contains exactly 0 or 1 error. Secondly, distinct subseeds may share a common exact prefix or a common exact suffix as well as parts with one error. Those two properties confer a practical advantage over suffix filters introduced in [12], for example.

In the two following sections, we will provide two examples of algorithms based on 01*0 seeds, that show the practicability of these seeds.

3. Application to the pattern matching problem

Let us consider the problem of finding matches of the pattern P with at most k errors in the text T. We assume a small pattern (several dozens of letters) and a large text (millions or billions of letters) over a small alphabet (*e.g.* DNA) that is known in advance. This problem arises naturally in several applications in computational biology, such as identifying regulatory signals along the genome, predicting targets of non-protein coding small RNAs or analysing spacers in CRISPR for potential transfers from viruses or plasmids, to mention a few. More generally, introducing some errors would improve the sensitivity in the presence of sequencing errors or variants.

We devise an efficient filtration algorithm for this problem based on the 01*0 seeds. We will first justify our choice of using an FM-index for T. Then we will explain how seeds are searched in the FM-index, and how the property of the 01*0 seeds is used during the verification step to restrict the number of errors searched for.

3.1. Choice of index

We are in the situation where the text is known in advance, and we may have hundreds of short sequences to be queried in the text. This situation is particularly suitable for text indexes. Using a text index, it is possible to reduce time consumption at the expense of space consumption. Two main families of indexes are used: q-gram indexes and full-text indexes. The former allow to efficiently recover occurrences of a fixed-length word, while the latter allow to search for any pattern of any length. A third family of indexes consists of indexes specifically designed for approximate search [16, 17, 18, 19]. However, these indexes are not compressed indexes (*i.e.* whose space consumption is proportional to the empirical entropy of the text) and, to the best of our knowledge, no implementation of the proposed solutions exists.

Compressed full-text indexes appear to be the indexes of choice, because they limit space consumption and allow for patterns of arbitrary sizes. Among compressed indexes, FM-indexes [14] have an optimal time complexity for counting the occurrences of a pattern, while pattern search is more complex and counting is more time consuming with LZ-indexes [20]. The FM-index accompanied by the Burrows-Wheeler transform has been successfully used in a variety of bioinformatics tools [4, 21, 22]. We will show in the remainder of this section that 01^*0 seeds are appropriate for a search in the FM-index.

3.2. Seed filtration

Given a pattern P, we enumerate all possible subseeds for the pattern. Each subseed for P is characterized by two parts P_i and P_j, $1 \leq i < j \leq k+2$, that occur exactly in the text. According to Lemma 1, all the intervening parts between P_i and P_j must be searched with exactly one error. We recall that in the FM-index, patterns are searched backwards, therefore, we first start by searching any part P_ℓ, with $1 < \ell \leq k+2$, assuming it is P_j. This is an exact search in the index. Then the parts preceding P_ℓ are searched with at most one error (by backtracking as in BWA for instance [21]). When a part is found exactly, we know that P_i has been reached. Starting with P_ℓ, we can have several parts that fulfil our requirements; on reaching different parts $P_{i_1}, \ldots, P_{i_q}$ each of them matching exactly at different locations in the text. All the possible solutions are searched. If P_ℓ cannot be found exactly or if a part cannot be found with at most one error, this P_ℓ is skipped and we move on the next one. Therefore, at most we will have considered the $\frac{(k+1)(k+2)}{2}$ possible pairs (i, j). All those pairs are searched within the FM-index.

Example 2. Let us continue with Example 1, also shown in Figure 1: $k = 3$ and $P =$ `AACG TGAG GTAG GTTC CATG`, which is partitioned into 5 parts of equal length. Assume that the text T is the concatenation of the three strings at distance 3 from P:
$T =$`AACGGAGGTAAGTTCTCATGAACGTAGGCAAGTTCCATGATCGTGACGTAGGGTCCATG`.
– The algorithm first tries $j = 5$. $P_5 =$ `CATG` is found with no error in the FM-index. So, it has some exact occurrences in the text. Therefore, we continue to go through the FM-index to extend P_5 to the left and find all possible values for i. We find $i = 4$ (P_4 occurs exactly), $i = 3$ (P_4 occurs with one error and P_3 exactly) and $i = 1$ (P_4, P_3 and P_2 occur with one error and P_1 exactly). This gives three different seed instances, leading to three seed occurrences.

```
 P1   P2   P3   P4    P5               P4   P5            P3   P4   P5
AACG TGAG GTAC GTTC- CATG             GTTC CATG          GTAG GTTC CATG
||||  ||| |||  ||||  ||||             |||| ||||          |||| | || ||||
AACG -GAG GTAA GTTCT CATG AACGTAGGCAA GTTC CATG ATCGTGAC GTAG GGTC CATG
```

– With $j = 4$, `GTTC` occurs exactly in the FM-index, and corresponds to two occurrences in T. By extending P_4 to the left, we keep just one instance since the second one cannot be extended to $P_3 =$ `GTAG` with at most one error.

```
 P1   P2   P3   P4                    P4
AACG TGAG GTAC GTTC                  GTTC
||||  ||| |||  ||||                  ||||
AACG -GAG GTAA GTTC TCATGAACGTAGGCAA GTTC CATGATCGTGACGTAGGGTCCATG
```

Note that in this particular case, the first occurrence of P in T is covered by two overlapping 01*0 subseeds, characterized by $i = 1$ and $j = 5$, and $i = 1$ and $j = 4$, respectively. This redundancy is solved with the extension and verification step, which is described in the next subsection.

– With $j = 3$, we have two occurrences of `GTAG` in the text. The first one cannot be extended to the left with $P_2 =$ `TGAG`. As for the second occurrence, P_2 is found with one error, but $P_1 =$ `AACG` does not exactly match. So, the occurrence is discarded.

```
                         P3                    P2   P3
                        GTAG                  TGAG GTAG
                        ||||                  |||  ||||
AACGGAGGTAAGTTCTCATGAAC GTAG GCAAGTTCCATGATCG TGAC GTAG GGTCCATG
```

– With $j = 2$, there is no exact occurrence of the part `TGAG` in the text.

At this point, all the seed *instances* occurring in the text are identified. We then proceed to the extension and verification step.

3.3. Elongation and verification

To perform the elongation of an instance of the seed, we first need to have a deeper look at the error distribution along the pattern. We know that the subseed instance has a Levenshtein distance of $j - i - 1$ with $P_i \dots P_j$, which makes $j - i - 1$ errors. The following lemma gives an upper bound for the number of errors in $P_1 \dots P_{i-1}$ and in $P_{j+1} \dots P_{k+2}$.

Lemma 4. *Let U be a string of Σ^* such that $\mathrm{lev}(P, U) = y \leq k$. Then there exists a 01^*0 subseed $P_i \dots P_j$ such that the prefix $P_1 \dots P_{i-1}$ contains exactly $i - 1 - (k - y)$ errors and the suffix $P_{j+1} \dots P_{k+2}$ contains exactly $k + 2 - j$ errors.*

Proof. The hypothesis of the Lemma can be rephrased as follows. Assume you have $k + 2$ containers numbered from 1 to $k + 2$, and y tokens with $0 \leq y \leq k$. Let $\delta = k - y$. The proof is by recurrence on k.

We start by noting that for $k = 1$, the solutions are $(1, 2)$ (token in container 3), $(2, 3)$ (token in container 1) or $(1, 3)$ (token in container 2), when $y = 1$. When $y = 0$ the unique solution is $(2, 3)$.

Assume now that $k > 1$. We choose i and j, such that the pair (i, j) fulfils the conditions of Lemma 2, and i is the container furthest to the left satisfying this condition. Let η be such that $i - 1 - \delta + \eta$ is the number of tokens present in containers 1 to $i - 1$. There are three possibilities for η.

If $\eta = 0$: the tokens from containers 1 to $i - 1$ sum up to $i - 1 - \delta$ and the tokens from $j + 1$ to $k + 2$ sum up to $k + 2 - j$. So the pair of containers (i, j) fulfils the criteria of our lemma.

If $\eta > 0$: there are $(i - 1 - \delta + \eta) + (j - i - 1) = j - \delta - 2 + \eta$ tokens in containers 1 to j. This leaves $y - (j - \delta - 2 + \eta)$ tokens in containers $j + 1$ to $k + 2$. In other words we have $k - j + 2 - \eta$ tokens in the $k - j + 3$ containers from j to $k + 2$ (as container j is empty). By recurrence, we know that there is a pair (j', j''), with $j \leq j' < j'' \leq k + 2$, such that containers j' and j'' are empty and there are $j' - j - (\eta - 1)$ tokens in containers j to j' and there are $k + 2 - j''$ tokens in containers $j'' + 1$ to $k + 2$. Finally from container 1 to j' we have $(j - \delta - 2 + \eta) + (j' - j - (\eta - 1)) = j' - 1 - \delta$ tokens. Therefore we found a pair (j', j'') that fulfils our conditions.

If $\eta < 0$: since container i is empty there are also $i - 1 - \delta + \eta$ tokens until container i. By recurrence, we know that we can find a pair (i_1, i_2) which fulfils Lemma 2. That contradicts our initial hypothesis, as $i_1 < i_2 \leq i$. □

As a consequence of this Lemma, seed instances are first extended to the left, to find $P_1 \dots P_{i-1}$ with at most $i - 1$ errors. To gain more efficiency, this extension is directly carried out in the FM-index. Indeed, the retrieval of the positions of occurrences is the most time consuming part in an FM-index (in $O(\log^{1+\epsilon} n)$ per occurrence [23]). Once this extension has been performed, all occurrences of the prefix $P_1 \dots P_j$ are retrieved. Then the extension to the right is performed in the text using a banded dynamic programming algorithm. The starting point of the extension is the ending position of the occurrence of $P_1 \dots P_j$ in the text. Following Lemma 4, $P_{j+1} \dots P_{k+2}$ must be searched with at most $k-j+2$ errors in the text. Therefore, the bandwidth is $2 \times (k-j+2)+1$ in the dynamic programming algorithm. Note that the extension to the right could also have been performed in the index using a bidirectional Burrows-Wheeler transform [24, 25]. That would, however, increase the memory footprint and provide only a moderate speed up, since many false positive seed instances have been removed at this step.

3.4. Implementation

This algorithm was implemented in a software called Bwolo. Bwolo is written in C++, with the help of SeqAn library and the FM-Index implemented within [26]. It can be downloaded from `http://bioinfo.lifl.fr/olo`. In this implementation, patterns are divided into parts whose length differs by at most one character.

3.5. Experimental results

In this section, we present some experimental results in order to measure the performance of our algorithm. We compare Bwolo to a selection of tools that were chosen for their complementarity. Widely utilized in bioinformatics, Exonerate is a generic tool for pairwise sequence alignment, which uses exact sparse dynamic programming to perform the search [27]. We use it as a standard for an on-line exact algorithm for our problem. RazerS3 is a read mapping program based on counting q-grams [28]. It performs the verification via an implementation of the improved Myers bit-vector algorithm proposed by Hyyr [29]. RazerS3 works without a precomputed index for the text. Bowtie2 [22], like our tool, indexes the text with an FM-index and searches for exact contiguous seeds. It then uses backtracking for handling errors and dynamic programming to build the full alignment. Readaligner [30] relies on suffix filters [12] searched within an FM-index too. Lastly, we used an in-house implementation for approximate search in an FM-index written with the SeqAn library. It is based on a breadth-first search method with no prior filtration. Unfortunately, we were not able to include hybrid methods described in [11] in our benchmark, since the implementation is not available.

All these tools were configured to be fully sensitive and output all occurrences of the pattern. Exonerate was launched using the options `--exhaustive -m affine:bestfit --bestn 1 --score -3 --gapextend -1 --gapopen -1 --showalignment yes --verbose 0`, RazerS3 with `--filter pigeonhole --percent-identity` $[Id]$ `--recognition-rate 100 -f` such that $[Id] = 100 \times (1 - \frac{k}{m})$, Bowtie2 with `-a -L` $[Seeds]$ `-i C,`$[Seeds]$`,0` such that $[Seeds] = \frac{m}{k+1}$ and Readaligner with options `--all -i 3`. Moreover, for each tool the score system is based on the unit score, which computes the Levenshtein distance.

The tests were run on a single thread of a server equipped with two Intel(R) Xeon(R) CPU E5-2420 and 205GB of RAM. The CPU time and the memory consumption were measured using the GNU `time` command.

3.5.1. Randomly generated sequences

This first test uses independent and identically distributed sequences on the DNA alphabet. The size n of the sequences ranges from 10^4 to 10^9. We also generated 100 patterns of length 20 at random and measured the computation time of each tool for $k = 2$ and $k = 3$. Results are shown in Figure 2.

Bwolo is the fastest tool for long sequences, from 10^6 nucleotides, apart for $k = 2$ where Readaligner is the fastest when the sequence is longer than 2×10^8. As expected, the added-value of Bwolo is even more obvious when $k = 3$. The two tools with no filtration, Exonerate and the exact search in the FM-index, are slow. Bowtie2 operates slowly compared to all the other tools, especially with larger values of n. This confirms that Bowtie2's heuristics, which has been designed for long patterns (at least 50 nt) and few errors, is not well adapted to shorter patterns with higher error rate. Unfortunately, there is not yet a specialized tool for this type of problem. In our benchmark, Bowtie2 is obliged to use a seed with low filtering power that lets too many occurrences happen. This dramatically increases the verification effort due to the cost of retrieving text positions from the FM-index. Interestingly enough, RazerS3, which uses the same seed, functions well on this data. This is consistent with the fact that a linear method can, in certain conditions for large k and n, be more efficient than a method based on a text index [10]. However, Bwolo is still five times faster than RazerS3 for sequences of length 10^9. Indeed, the number of seed occurrences is an order of magnitude less with Bwolo, which offsets the additional time needed to query the FM-index in the verification step. With the configuration we used, Readaligner relies on suffix filters to align the sequences. Those filters have very good performances when the number of errors is not too high ($k = 2$ in our case). With suffix filters the most time-consuming case occurs when all parts contain exactly one error but the last one. With $k = 3$, the last part is 5nt long. Searching such a small part leads to many false positives.

For $k = 2$, we can observe that there are smaller differences in the CPU time between FM-index and Bwolo on larger texts. The former takes 18.4 s on the 1GB sequence while the latter takes 13.8 s. This small difference is actually misleading. Unserializing the index (which is the same in both cases) takes 12 s on that same sequence. Ignoring that step, which is constant whatever the number of sequences to search is, leads to a three-fold speedup using the 01*0 seeds compared to the breadth-first approach. With a higher error rate ($k = 3$) we have a seventy-five-fold speedup on the 1GB sequence. For the sake of comprehensiveness, we should mention that Readaligner takes 1s to unserialize its index and RazerS3 takes 8s to load the 1GB sequence. Taking that into account shows why Readaligner becomes quicker with long sequences and a small error rate: this is due to the loading of the index which is much more efficient than for RazerS3 and Bwolo.

All tools have a reasonable memory consumption, independent of the value of k, which grows linearly with the size of the text. For example, it is 13 MB for Readaligner, 27 MB for Bwolo, 99 MB for Bowtie2, 25 MB for RazerS3, and 31 MB for Exonerate for $n = 10^7$. The memory consumption of Bwolo and Bowtie2 is dominated by the size of the FM-index. It is larger for Bowtie2 because it also deals with the inverted text and uses a different implementation. It is quite surprising that RazerS3 and Exonerate have a memory peak in the same order. It may be possible that they load both all the text and keep all results in memory.

3.5.2. Reads from the Human genome

The second test uses an external dataset made of short sequences, taken from [31]. In this article, the $\mathcal{H}_3$ dataset contains 10 millions of sequencing reads of length 40 that have been generated from the Human genome (assembly 37.1 from the NCBI, 25 chromosomes for 2.7 Gbp) with exactly three mismatches. Compared to the previous test, it allows us to evaluate the performance of the software with longer patterns, hence longer seeds. The maximum number k of errors is 3 (including indels, not only substitutions). We ran Bwolo, RazerS3, and Bowtie2 on the full set of reads (10^7 reads). Since we were not able to obtain results with both Bowtie2 and Readaligner on the full dataset within a

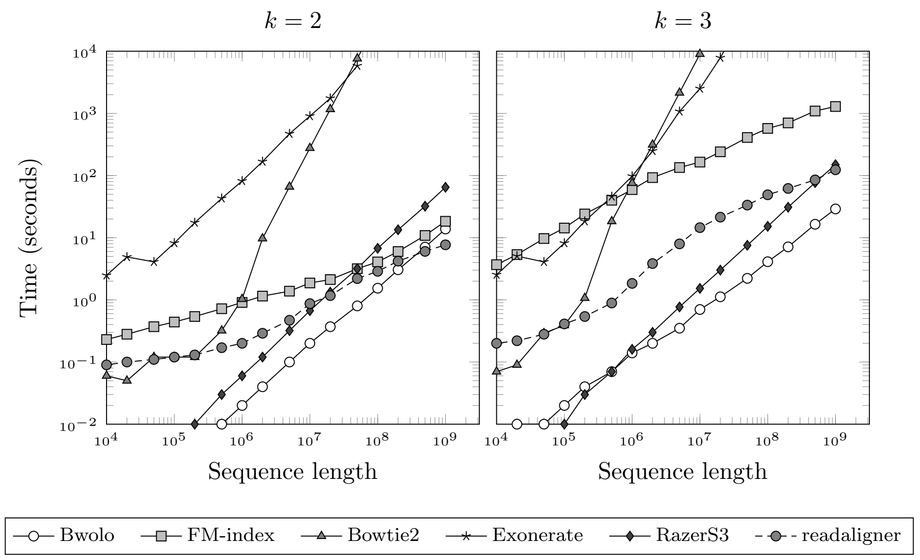


Figure 2: Running time for 100 randomly generated sequences. Both axes are in logarithmic scale. Bwolo is our algorithm. FM-index refers to the breadth-first search implementation in an FM-index.

reasonable amount of time, we also used a selection of 10,000 reads. Table 2 shows the results. As in the previous test, Bwolo achieves the best performances. However, the difference between Bwolo, Readaligner and RazerS3 is even more striking than in the previous test. This is due to the time needed to load the index. It was negligible on this dataset, but it constituted an important part of the search time with a much smaller dataset as in the previous test.

	index construction		10,000 reads		10^7 reads	
	time	mem.	time	mem.	time	mem.
	(min.)	(GB.)	(min.)	(GB.)	(min.)	(GB.)
Bwolo	127	9.6	1.6	6.5	925	9.1
RazerS3	0	0	8.4	6.5	7,790	152
Readaligner	451	5.5	41	3.6	NA	NA
Bowtie2	176	5.4	2,603	8.3	NA	NA

Table 2: Running time and space consumption on the Human genome benchmark. NA: non available.

4. Application to the approximate dictionary matching problem

In this section, we consider the approximate dictionary matching problem: finding all approximate occurrences of a set $\mathcal{P}$ of string patterns in a query text

Q. More specifically, we assume that $\mathcal{P}$ is a database of short patterns (about 20 letters each) that is known in advance and can be preprocessed. We want to query this database with a sequence Q of a few hundred letters. We will show how the 01*0 seeds can be used to solve this problem by indexing $\mathcal{P}$.

4.1. Indexing pairs of exacts parts in 01*0 *seeds*

Let us assume that we have a collection $\mathcal{P}$ of patterns and that all patterns have the same length m. We will discuss later, in Subsection 4.4, the case where the patterns have different lengths. The query is a sequence Q whose length n is greater than m.

In accordance with the principle of the 01*0 seeds, every pattern P of $\mathcal{P}$ is split into $k+2$ non overlapping parts, $P_1, \ldots, P_{k+2}$. For the sake of simplicity we will assume for now that $k+2$ divides m and that all parts have the same length p. The first step of the algorithm is to search for all possible pairs of exact parts of all patterns of $\mathcal{P}$ that are present in the text Q. For this purpose, we work subseed by subseed and for each pair i and j, $1 \leq i < j \leq k+2$ corresponding to a given subseed, we will use an array $A_{i,j}$, indexed by strings of length $2p$, corresponding to pairs of parts. Given a string S of length $2p$, $A_{i,j}[S]$ is the set of all patterns whose part i concatenated to part j equals S:

$$A_{i,j}[S] = \{P \in \mathcal{P}; P_i P_j = S\}$$

There are as many arrays as possible subseeds, which makes $k(k+1)$ arrays, each of size σ^{2p}. This is of course practical only for small values of p and σ, which is the case in our application.

Example 3. We show an example of construction of the $A_{i,j}$ arrays in Figure 3. The set $\mathcal{P}$ consists of three patterns of length 8: P^1 = `ATACCACT`, P^2 = `TAACATCT`, P^3 = `ACCATTAT`. The number of errors is $k = 2$. So each pattern is split in four parts of length 2. We have 6 arrays, $A_{1,2}$, $A_{1,3}$, $A_{1,4}$, $A_{2,3}$, $A_{2,4}$ and $A_{3,4}$. We give the content of each of these 6 arrays. For example, $A_{1,2}$ has three elements. The first one is 3 indexed by `ACCA`, meaning that the pattern P^3 is such that P^3_1 = `AC` and P^3_2 = `CA`. For the array $A_{2,4}$, we have $A_{2,4}$[`ACCT`] = {`1,2`}, because P^1 and P^2 share the same second part, which is `AC`, and the same fourth part, which is `CT`.

4.2. Querying the $A_{i,j}$ arrays

The two exact parts of the 01*0 seeds are identified using the $A_{i,j}$ arrays. We still have to determine what is the range of distances between those two parts on Q to make it possible that P_i and P_j are the exact parts of a subseed of a 01*0 seed. For that, we will use the weakened form of Lemma 1, introduced in Section 2.2 for the *bounded* $k+2$ *pigeonhole* seed. Let assume that for a given pattern P, the parts P_i and P_j match on the query sequence Q. There are $j-i-1$ parts between the exact parts P_i and P_j in the pattern. Each part has length p, and should occur with exactly one error in Q. So the total number of

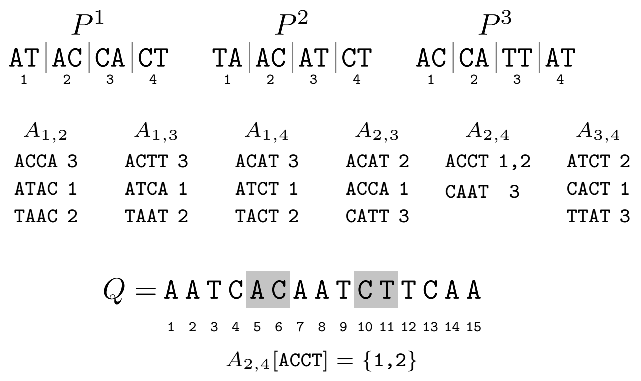

P^1 P^2 P^3
AT|AC|CA|CT TA|AC|AT|CT AC|CA|TT|AT
1 2 3 4 1 2 3 4 1 2 3 4
$A_{1,2}$: ACCA 3, ATAC 1, TAAC 2
$A_{1,3}$: ACTT 3, ATCA 1, TAAT 2
$A_{1,4}$: ACAT 3, ATCT 1, TACT 2
$A_{2,3}$: ACAT 2, ACCA 1, CATT 3
$A_{2,4}$: ACCT 1,2, CAAT 3
$A_{3,4}$: ATCT 2, CACT 1, TTAT 3
$Q =$ A A T C A C A A T C T T C A A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
$A_{2,4}$[ACCT] = {1,2}

Figure 3: Example of indexing and querying strategy on a set $\mathcal{P}$ of three patterns of length 8: P^1 = ATACCACT, P^2 = TAACATCT, P^3 = ACCATTAT. The number of errors is $k = 2$. Each pattern is split in four parts of length 2.

positions between P_i and P_j on Q should be comprised between $(i-j-1)(p-1)$ and $(i-j-1)(p+1)$. Therefore at each position of the text Q, we must search for:

1. two consecutive parts in $A_{1,2}, A_{2,3}, \ldots, A_{k+1,k+2}$,
2. two parts separated by $p-1$, p or $p+1$ positions in $A_{1,3}, \ldots, A_{k,k+2}$,
3. ...
4. two parts separated by $k(p-1), \ldots, k(p+1)$ positions in $A_{1,k+2}$.

So, for each d in $[0..k]$ and for each $i \in [1..k-d+1]$, the array $A_{i,i+d+1}$ needs $2d+1$ queries. Since there are $k+1-d$ such arrays for a given value of d, this gives a total of $\sum_{d=0}^{k}(2d+1)(k+1-d) = O(k^3)$ queries. Finally identifying candidates in $\mathcal{P}$ which have compatible pairs with Q can be done in time $O(|Q|k^3 + occ)$, where *occ* is the number of occurrences. Indeed, queries in $A_{i,j}$ arrays can be done in constant time (plus the time needed to retrieve the occurrences). This step is illustrated in the example below.

Example 3 (continued). We now have a text to query: Q = AATCACAATCTTCAA. We show how the text is scanned in the bottom part of Figure 3. At a given point, the parts queried will be the ones in a frame. While maintaining the first part identical (AC), the second part queried in $A_{2,4}$ will be AT, TC and CT as we must account for potential deletion or insertion in the inner part. The case shown in the figure illustrates the case where an insertion is assumed.

4.3. Aligning candidates against the query

Once the candidates have been retrieved, they must be aligned against Q to check if there are at most k errors between the pattern and Q. Let assume that P is a pattern of $\mathcal{P}$ that has been selected by the preceding filtering step of the algorithm. By construction, there exist i and j, $1 \leq i < j \leq k+2$, q,

$0 \leq q \leq n-1$ and δ, $(j-i-1)(p-1) \leq \delta \leq (j-i-1)(p+1)$, such that $P_i = Q[q \ldots q+p-1]$ and $P_j = Q[q+\delta \ldots q+\delta+p-1]$. First we need to check whether the two parts P_i and P_j could be the exact parts of a 01*0 subseed. For that, the alignment of $P_{i+1} \ldots P_{j-1}$ against $Q[q+p \ldots q+\delta-1]$ must verify that each part has exactly one error to the corresponding substring in Q. Then to extend this subseed, we must align the two remaining substrings $P_1 \ldots P_{i-1}$ and $P_{j+1} \ldots P_{k+2}$ to Q by dynamic programming. Similarly to what is done in Bwolo, we can bound the number of errors for each subtring and speed up the computation accordingly. When aligning $P_1 \ldots P_{i-1}$ against Q, we know by Lemma 4 that we can assume that there are at most $i-1$ errors.

Example 3 (continued). We continue with the example of Figure 3. Recall that at this step of the algorithm, the framed subwords correspond to parts P^1_2 and P^1_4 in P^1, and to parts P^2_2 and P^2_4 in P^2. The next step is to align `AAT` (the inner sequence between the two considered parts in Q) against $P^1_3 = \texttt{CA}$ and $P^2_3 = \texttt{AT}$ with exactly one error. As $lev(\texttt{CA}, \texttt{AAT}) > 1$, P^1 is not a candidate anymore. Regarding P^2, since $lev(\texttt{AT}, \texttt{AAT}) = 1$, we have found an instance of a subseed. It means that P^2 is still a valid candidate, that must be fully aligned against Q. As we are considering parts 2 and 4, we must align the prefix P^2_1 on Q, before position 5, with exactly one error (if part 1 had no error we would already have a hit).

4.4. Dealing with variable-length database sequences

For the sake of simplicity, we started by assuming that patterns in $\mathcal{P}$ all have the same length m and that $k+2$ divides this length. We will now explain how to relax those assumptions.

First we will consider the case where a pattern P of $\mathcal{P}$ is strictly longer than m. The first $k+1$ parts of P will all have the same length p. The length of the last part will be $p' = |P| - (k+1)p$. Therefore the first $k+1$ parts of P are treated in the same way as the parts of the other sequences. This cannot be true for the last part whose size can vary. This part of P is still indexed on its first p letters. However the $A_{i,k+2}$ arrays (with $1 \leq i \leq k+2$) have to store slightly more information compared to the other arrays. In addition to the indices, the trailing $p'-p$ letters of the last parts are also stored:

$$A_{i,k+2}[S] = \{(P,w); P_i P_{k+2}[1 \ldots p] = S \text{ and } w = P_{k+2}[p+1 \ldots p']\}$$

Therefore when querying any array $A_{i,k+2}$ with some subtring $S = Q[q \ldots q+p-1]Q[q' \ldots q'+p-1]$, all the entries $(P',w) \in A_{i,k+2}[S]$ are retrieved, such that $w = Q[q'+p \ldots q'+p+|w|-1]$. Rather than storing entries of $A_{j,k+2}$ as plain lists, they are stored in a trie. In this trie, a path labelled w from the root to a leaf, gives all the patterns stored in $A_{j,k+2}$ that have w as trailing letters.

Second we consider the case where a pattern P in $\mathcal{P}$ is shorter than m. We will assume that only the last part is shorter than the other ones. Let $p' = |P_{k+2}|$, for each part $i, 1 \leq i \leq k+1$, we will have $P \in A_{i,k+2}[P_i P_{k+2} w]$, for all $w \in \Sigma^{p-p'}$. If other parts also have to be smaller, we process them similarly.

4.5. Implementation

This algorithm for the multi-pattern matching problem has been implemented in C++ in a software called Piccolo, which is available at `http://bioinfo.lifl.fr/olo/`. Piccolo uses the same partition as Bwolo. For the $A_{i,j}$ arrays, the index data structure is simply a lookup table, like in Blastn.

4.6. Experimental results

We measure the performance of Piccolo on a case study from computational biology: the classification of plant miRNAs (microRNAs) in families. Plant miRNAs are short non-coding RNAs that are 21–24 nucleotides long, and that are produced from larger precursor sequences that can be hundreds of nucleotides long. Many miRNA genes are conserved across plant species, and form gene families. So an important step for the miRNA annotation is to compare a precursor sequence to a known miRNAs, in order to identify homologs. For that, the protocol is to align the sequence of the precursor to a database of known miRNAs, such as miRBase [32]. What is found commonly in the literature is to require a full-length alignment of a miRNA with up to 3 errors [33].

To our knowledge, there exist no dedicated tools for such a task, either with a precomputed index or not. One possibility is to perform dynamic programming on each miRNA sequence of the database against the precursor sequence. This is slow in practice, and one has to resort to heuristic methods, in particular Blastn. In this case Blastn provides quick results at the cost of some false positive and false negative results.

We will therefore test our program against Exonerate (as in Section 3) and Blastn. The data was retrieved from miRBase 21. It consists of 124 miRNA precursors from *Amborella trichopoda* (minimum length: 80, maximum length: 2354, average length: 284, median length: 228) and 4,713 plant miRNAs (excluding those from *Amborella trichopoda*, minimum length: 17, maximum length: 26, average length: 22, median length: 21). The programs were launched on one thread of an Intel i7-4600 dual core processor. Exonerate was run with the same options as in Section 3. For fair comparison we used a slightly modified version that searches matches on the forward strand only. Blastn was launched in its standalone version but with the smallest seed available in the online version (as most users usually do), which spans seven characters. Blastn does not allow arbitrary costs for the gaps; we chose the costs that were closest to those for Exonerate. The options were `-strand plus -outfmt 6 -task blastn-short -min_raw_gapped_score 53 -penalty -5 -reward 4 -gapopen 5 -gapextend 5`.

The computation times, including the indexation time for Blastn and Piccolo, are presented in Table 3. Unsurprisingly the dynamic programming approach of Exonerate is much slower. It takes more than 400 times longer than Blastn and Piccolo. The very well optimized Blastn is not the fastest, as Piccolo is slightly faster. Moreover, Blastn has higher memory footprint. This shows that even a naive implementation of the $A_{i,j}$ arrays based on a lookup table yields good performances on the space consumption side. We deliberately

	time	memory	alignments	false negatives	false positives
	(s.)	(MB.)			
Exonerate	168	4	1428	–	–
Blastn	0.4	130	1282	146	225
Piccolo	0.3	72	1428	–	–

Table 3: Computation time and memory usage for Exonerate, Blastn and Piccolo for a dataset of 4,713 plant miRNAs searched among each of 124 miRNA precursors. The column *alignments* gives the total number of alignments found by the tool and satisfying the specification of the problem: up to 3 errors. For Blastn, we also report the number of false negatives (alignments that have not been found) and false positives (alignments that do not satisfy the specification).

opted for this index data structure to enhance the running time. In presence of a larger database of patterns, the algorithm could also be implemented with hashtables or a compressed full-text index.

Exonerate and Piccolo, which are exact methods, both find 1,428 alignments. Since Blastn is a heuristic, it has false positive and false negative results: Blastn misses 146 alignments (almost 10%), and outputs 225 local alignments that contain more than three errors when extended to the full length miRNA. Such false results can bias the type of miRNAs found and therefore could hinder the understanding in their evolution. In conclusion, Piccolo is as fast as Blastn, but with the advantage of being fully sensitive.

5. Application to the k-mismatch problem

In this last section, we make a short digression and study the application of the 01*0 seeds to the k-mismatch problem: given a pattern P, a threshold k, find all strings U whose Hamming distance between P and U is less than k. 01*0 seeds are not specifically designed to solve this problem. The reason why we perform this analysis is because it allows us to compare the filtering power of 01*0 seeds to spaced seeds. A spaced seed consists of an ℓ-mer in which some predefined positions can match any character symbol in the alphabet. Exact match positions are usually denoted `#`, and positions which are not required to match `-`. Spaced seeds are known to offer a good sensitivity-selectivity trade off when dealing with mismatches [7, 8]. In particular, spaced seeds are better than contiguous seeds. Note also that the k-mismatch problem has attracted attention very recently in [34], and that our 01*0 seeds when restrained to substitutions could be viewed as a special case of the framework introduced in their paper.

Given a pattern P of length m partitioned in $k+2$ parts, $P_1, \dots, P_{k+2}$, we define $S(P_1, \dots, P_{k+2})$ to be the set of words of Σ^m that are recognized by the 01*0 seed based on this partition. Our goal is to compute the cardinality of $S(P_1, \dots, P_{k+2})$. Since we have only mismatches, this value depends only on the distribution of $p_1, \dots, p_{k+2}$, the lengths of $P_1, \dots, P_{k+2}$, and on the size σ of the alphabet.

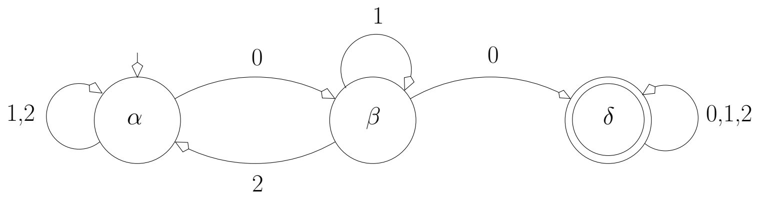


Figure 4: Deterministic Finite Automaton for the regular expression $\{0,1,2\}^*01^*0\{0,1,2\}^*$. α is the initial state and δ is the unique final state.

Lemma 5. *The cardinality of $S(P_1,\ldots,P_{k+2})$ is δ_{k+2} where δ_{k+2} is defined by the linearly recursive sequence.*

$$\begin{pmatrix}\alpha_0\\ \beta_0\\ \delta_0\end{pmatrix}=\begin{pmatrix}1\\0\\0\end{pmatrix}\quad\begin{pmatrix}\alpha_i\\ \beta_i\\ \delta_i\end{pmatrix}=\begin{pmatrix}\sigma^{p_i}-1 & \sigma^{p_i}-(\sigma-1)p_i-1 & 0\\ 1 & (\sigma-1)p_i & 0\\ 0 & 1 & \sigma^{p_i}\end{pmatrix}\begin{pmatrix}\alpha_{i-1}\\ \beta_{i-1}\\ \delta_{i-1}\end{pmatrix}$$

Proof. We convert the problem in a new alphabet, $\{0,1,2\}$. For each i, $1\le i\le k+2$, define $H_i:\Sigma^{p_i}\to\{0,1,2\}$ by

$$\begin{aligned}H_i(U)&=0, && \text{if } ham(P_i,U)=0\\ H_i(U)&=1, && \text{if } ham(P_i,U)=1\\ H_i(U)&=2, && \text{otherwise } (ham(P_i,U)\ge 2)\end{aligned}$$

where *ham* denotes the Hamming distance. In this context, the set $S(P_1,\ldots,P_{k+2})$ is exactly the set of words U of Σ^m such that $H_1(U_1)\ldots H_{k+2}(U_{k+2})$ is accepted by the regular expression $\{0,1,2\}^*01^*0\{0,1,2\}^*$, where $U_1,\ldots,U_{k+2}$ is the unique partition of U in parts of length $p_1,\ldots,p_{k+2}$ respectively. The DFA (Deterministic Finite Automaton) for this regular expression is given in Figure 4. It has three states: α, β and δ. Let α_i (respectively β_i, δ_i) denote the number of strings U of $\Sigma^{p_1+\cdots+p_i}$, such that the DFA starting at the initial state ends in the state α (respectively β, δ) after reading the string $H_1(U_1)\cdots H_i(U_i)$. From the DFA, we infer the following relations.

$$\begin{pmatrix}\alpha_0\\ \beta_0\\ \delta_0\end{pmatrix}=\begin{pmatrix}1\\0\\0\end{pmatrix}$$

$$\begin{pmatrix}\alpha_i\\ \beta_i\\ \delta_i\end{pmatrix}=\begin{pmatrix}\overline{H}_i(1)+\overline{H}_i(2) & \overline{H}_i(2) & 0\\ \overline{H}_i(0) & \overline{H}_i(1) & 0\\ 0 & \overline{H}_i(0) & \overline{H}_i(0)+\overline{H}_i(1)+\overline{H}_i(2)\end{pmatrix}\begin{pmatrix}\alpha_{i-1}\\ \beta_{i-1}\\ \delta_{i-1}\end{pmatrix}$$

$m = 20$	$k+1$ pigeonhole	$k+2$ pigeonhole	spaced-seed	01*0 seed
$k = 1$	10,10	7,7,6	`###-###-###-###-#`	7,7,6
$k = 2$	7,7,6	5,5,5,5	`##-#--##-#---##`	5,5,5,5
$k = 3$	5,5,5,5	4,4,4,4,4	`#-#---#-#---#-#`	4,4,4,4,4
$k = 4$	4,4,4,4,4	4,4,3,3,3,3	`#----#----#----#`	4,4,3,3,3,3

$m = 20$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
$k+1$ pigeonhole	2,097,151	402,616,321	4,288,679,935	21,307,718,401
$k+2$ pigeonhole	36,862	6,283,267	166,465,276	2,104,103,941
spaced-seed	65,464	100,506,112	1,593,257,920	21,307,718,401
01*0 seed	20,500	3,174,595	69,334,045	973,241,233

Table 4: The top table displays the form of seed used for each value of k. For the $k+1$ pigeonhole, $k+2$ pigeonhole and 01*0 seeds, each line gives the length of each non-overlapping part. For the space seed, # are for fixed positions and - are for variable positions. In the bottom table, we report the number of words of length $m = 20$ recognized by the corresponding seed.

where for each i, $1 \leq i \leq k+2$, for each x in $\{0, 1, 2\}$, $\overline{H}_i(x)$ denotes the number of strings U of Σ^{p_i} that are mapped to x by H_i. In other words, this is the cardinality of $H_i^{-1}(x)$. From the definition of H_i, it is straightfoward to see that

$$\begin{array}{rcl}\overline{H}_i(0) & = & 1 \\ \overline{H}_i(1) & = & (\sigma-1)p_i \\ \overline{H}_i(2) & = & \sigma^{p_i} - (\sigma-1)p_i - 1\end{array}$$

This gives the expected result. □

Lemma 5 allows us to characterize exactly the number of words caught by the 01*0 seed. In Table 4, we report a comprehensive comparison with spaced seeds and $k+1$, $k+2$ pigeonhole seeds for the special case of a pattern of length $m = 20$ on the DNA alphabet, and values of k ranging from 1 to 4. Spaced seeds were designed with Iedera [6, 35]: in each case, we selected the lossless spaced seed with maximum weight (the number of #), and then maximum length (the number of # plus the number of -). The two $k+2$ pigeonhole seeds considered so far are identical here, since only mismatches are allowed: Two exact parts P_i and P_j should be separated by exactly $p_{i+1} + \cdots + p_{j-1}$ positions. In the bottom table, we report the number of words of length $m = 20$ recognized by the corresponding seed. For the $k+1$ pigeonhole seed, we require that one part occurs exactly in the string, at the same position. For the $k+2$ pigeonhole seed, we require that two parts occur exactly in the string, at the same positions. For the spaced seed, we require that there exists at least one seed shared by the pattern and the string. Finally, for the 01*0 seed, we use Lemma 5. This table shows that the 01*0 seed is always the best seed, and that it improves the filtering power of spaced seeds by a factor of at least 20 as soon as there are at least 2 errors in the pattern.

6. Conclusion

We have introduced a new type of seeds, named 01*0 seeds, that are especially well-suited to deal with patterns containing a high rate of errors. We have shown that these seeds are easy to use with two complementary applications: the pattern matching problem, where the seeds are efficiently searched in a compressed full-text index, and the multi-pattern matching problem where the seeds are indexed in a classical lookup table and the text is processed on-line. Each of these two algorithms were implemented. The programs are open source and are available at `http://bioinfo.lifl.fr/olo`. We also made some theoretical comparisons to contiguous and spaced seeds and we showed that 01*0 seeds have a better filtering power than their counterparts under the k-mismatch problem.

We believe that 01*0 seeds constitute a promising alternative to existing approaches. They should prove useful for a variety of applications where the sensitivity of the search is critical. We have illustrated this through a selection of examples coming from computational biology. Another application in this field concerns the processing of third generation sequencing data, such as produced by PacBio or Oxford Nanopore sequencers [36]. Those emerging technologies are known to deliver longer sequencing reads at the price of a higher error rate. So exploiting the power of these sequencers requires the development of new algorithms for the assembly problem as well as for the mapping problem that are able to accomodate error-prone sequences, which is a key point of 01*0 seeds.

The 01*0 seeds also raise new computational problems. The first one is a formal analysis of the selectivity of those seeds. In this article, we have only provided an empirical study on randomly generated sequences. Another question of interest is the indexing strategy of 01*0 seeds, that could still be improved as we must take into account the potential shifts introduced by insertions or deletions in the 1* parts of the 01*0 seeds. A future research would be to devise a more efficient indexing strategy that could help in lowering the query complexity.

7. References

[1] R. Wagner, M. Fischer, The string-to-string correction problem, Journal of the ACM 21 (1) (1974) 168–173.

[2] S. Wu, U. Manber, Fast text searching: allowing errors, Communications of the ACM 35 (10) (1992) 83–91.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, D. Lipman, Basic local alignment search tool, Journal of Molecular Biology 215 (3) (1990) 403–410.

[4] M. Petri, J. S. Culpepper, Efficient indexing algorithms for approximate pattern matching in text, in: Proceedings of the Seventeenth Australasian Document Computing Symposium, ADCS '12, ACM, New York, NY, USA, 2012, pp. 9–16.

[5] B. Ma, J. Tromp, M. Li, Patternhunter - faster and more sensitive homology search, Bioinformatics 18 (3) (2002) 440–445.

[6] G. Kucherov, L. Noé, M. Roytberg, A unifying framework for seed sensitivity and its application to subset seeds, Journal of Bioinformatics and Computational Biology 4 (02) (2006) 553–569.

[7] D. G. Brown, M. Li, B. Ma, A tutorial of recent developments in the seeding of local alignment, Journal of Bioinformatics and Computational Biology 2 (04) (2004) 819–842.

[8] U. Keich, M. L. Li, B. Ma, J. Tromp, On spaced seeds for similarity search, Discrete Applied Mathematics 138 (3) (2004) 253–263.

[9] S. Burkhardt, J. Kärkkäinen, One-Gapped q-Gram Filters for Levenshtein Distance, in: Combinatorial Pattern Matching, no. 2373 in Lecture Notes in Computer Science, 2002, pp. 225–234.

[10] G. Navarro, R. Baeza-Yates, A Hybrid Indexing Method for Approximate String Matching, Journal of Discrete Algorithms 1 (2001) 19–27.

[11] L. Russo, G. Navarro, A. L. Oliveira, P. Morales, Approximate string matching with compressed indexes, Algorithms 2 (3) (2009) 1105–1136.

[12] J. Kärkkäinen, J. C. Na, Faster filters for approximate string matching, in: ALENEX, SIAM, 2007, pp. 84–90.

[13] C. Vroland, M. Salson, H. Touzet, Lossless seeds for searching short patterns with high error rates, in: IWOCA (International Workshop On Combinatorial Algorithms), no. 8986 in Lecture Notes in Computer Science, 2014, pp. 364–375.

[14] P. Ferragina, G. Manzini, Indexing compressed text, Journal of the ACM 52 (4) (2005) 552–581.

[15] R. Baeza-Yates, C. Perleberg, Fast and practical approximate string matching, Information Processing Letters 59 (1) (1996) 21–27.

[16] E. Chávez, G. Navarro, A Metric Index for Approximate String Matching, in: S. Rajsbaum (Ed.), LATIN : Theoretical Informatics, no. 2286 in Lecture Notes in Computer Science, Springer, 2002, pp. 181–195.

[17] M. G. Maaß, J. Nowak, Text indexing with errors, Journal of Discrete Algorithms 5 (4) (2007) 662–681.

[18] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, S.-S. Wong, A linear size index for approximate pattern matching, Journal of Discrete Algorithms 9 (4) (2011) 358–364.

[19] D. Belazzougui, Improved space-time tradeoffs for approximate full-text indexing with one edit error, Algorithmica (2014) 1–27.

[20] P. Ferragina, R. González, G. Navarro, R. Venturini, Compressed text indexes: From theory to practice, Journal of Experimental Algorithmics 13 (2009) 12.

[21] H. Li, R. Durbin, Fast and accurate short read alignment with Burrows-Wheeler transform, Bioinformatics 25 (14) (2009) 1754–1760.

[22] B. Langmead, S. Salzberg, Fast gapped-read alignment with Bowtie 2, Nature methods 9 (4) (2012) 357–359.

[23] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms 3 (2).

[24] T. Schnattinger, E. Ohlebusch, S. Gog, Bidirectional search in a string with wavelet trees, in: Combinatorial Pattern Matching, no. 6129 in Lecture Notes in Computer Science, Springer, 2010, pp. 40–50.

[25] D. Belazzougui, F. Cunial, J. Kärkkäinen, V. Mäkinen, Versatile succinct representations of the bidirectional Burrows-Wheeler transform, in: Algorithms (ESA), no. 8125 in Lecture Notes in Computer Science, Springer, 2013, pp. 133–144.

[26] A. Döring, D. Weese, T. Rausch, K. Reinert, SeqAn an efficient, generic C++ library for sequence analysis, BMC Bioinformatics 9 (1) (2008) 11–19.

[27] G. Slater, E. Birney, Automated generation of heuristics for biological sequence comparison, BMC Bioinformatics 6 (2005) 1–11.

[28] D. Weese, M. Holtgrewe, K. Reinert, RazerS 3: Faster, fully sensitive read mapping, Bioinformatics 28 (20) (2012) 2592–2599.

[29] H. Hyyrö, A Bit-vector Algorithm for Computing Levenshtein and Damerau Edit Distances, Nordic Journal of Computing 10 (1) (2003) 29–39.

[30] V. Mäkinen, N. Välimäki, A. Laaksonen, R. Katainen, Unified view of backward backtracking in short read mapping, in: Algorithms and Applications, Springer, 2010, pp. 182–195.

[31] S. Schbath, V. Martin, M. Zytnicki, J. Fayolle, V. Loux, J.-F. Gibrat, Mapping Reads on a Genomic Sequence: An Algorithmic Overview and a Practical Comparative Analysis, Journal of Computational Biology 19 (6) (2012) 796–813.

[32] A. Kozomara, S. Griffiths-Jones, miRBase: annotating high confidence microRNAs using deep sequencing data, Nucleics Acids Research 42 (2014) D68–D73.

[33] U. Chorostecki, V. Crosa, A. Lodeyro, N. Bologna, A. Martin, N. Carrillo, C. Schommer, J. Palatnik, Identification of new microRNA-regulated genes by conserved targeting in plant species, Nucleic Acids Research 40 (18) (2012) 8893–8904.

[34] G. Kucherov, K. Salikhov, D. Tsur, Approximate string matching using a bidirectional index, in: Combinatorial Pattern Matching, Lecture Notes in Computer Science, Springer, 2014, pp. 222–231.

[35] G. Kucherov, L. Noé, M. Roytberg, Subset seed automaton, in: International Conference on Implementation and Application of Automata (CIAA), no. 4783 in Lecture Notes in Computer Science, 2007, pp. 180–191.

[36] J. Thompson, P. Milos, The properties and applications of single-molecule DNA sequencing, Genome Biology 12 (2) (2011) 217–226.