Lossless seeds for searching short patterns with high error rates*

Christophe Vroland[1,2,3], Mikaël Salson[1,2], and Hélène Touzet[1,2]

[1] LIFL (UMR CNRS 8022, Université Lille 1),
[2] Inria Lille Nord-Europe, France
[3] Laboratoire Génétique et Evolution des Populations Végétales (UMR CNRS 8198, Université Lille1)
{christophe.vroland, mikael.salson, helene.touzet}@lifl.fr

Abstract. We address the problem of approximate pattern matching using the Levenshtein distance. Given a text T and a pattern P, find all locations in T that differ by at most k errors from P. For that purpose, we propose a filtration algorithm that is based on a novel type of seeds, combining exact parts and parts with a fixed number of errors. Experimental tests show that the method is specifically well-suited for short patterns with a large number of errors.

1 Introduction

We consider the approximate pattern matching problem where a pattern P is searched in a text T with a given number of errors k. An error can be defined in several ways. Here we consider an error as defined by the Levenshtein distance: either a substitution, an insertion, or a deletion. The problem is to find all the locations where the pattern matches the text with at most k errors.

Navarro *et al* distinguish three main approaches [17]. The first one, *neighborhood generation*, consists in generating all the strings within the number of errors of the queried string. Then the generated strings are searched exactly in the text. This generation is exponential in the number of errors. It is often done with dynamic programming, bit-parallelism, or finite automata (for instance [27]).

A second approach, *partitioning approach*, consists in filtering regions of interest. Such regions are found by using pattern substrings, called *seeds*. Once the seeds are found in the text, their occurrences must be extended to check if the pattern occurs within k errors. The pigeonhole principle is used, where the pattern is split in $k + \epsilon$ non-overlapping parts, $\epsilon > 0$. Then ϵ parts are searched exactly, usually $\epsilon = 1$. 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. A recent example of this method, using a modified Burrows-Wheeler transform is shown in [18].

* This work was partly supported by the Mastodons project (CNRS).

Finally, the third approach a hybrid of the two previous established approaches. The pattern is split in non-overlapping parts that can be searched with a given number of errors. For instance, Navarro and Baeza-Yates [16] designed a *hybrid method* which consists of splitting the pattern in $j = (m + k)/\log_\sigma n$ 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 a FM-index [19].

Each of these search strategies can be directly applied to a string and will be linear in time depending on the size of the text. For a survey of online algorithms, refer to [15]. Using a text index, it is also 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 allows to efficiently recover occurrences of a fixed-length word, while the latter allows to search for any pattern of any length. This generally allows one to backtrack so that a word can be searched with some errors. A third family of indexes consists of indexes specifically designed for approximate search [5,13,4,2]. 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.

Despite all these methods, we think there is a need for algorithms dedicated to searching short patterns (< 50 letters) on a small alphabet (*e.g.* DNA alphabet) with a medium to high error-rate (7 %–15 %). This can be used in several applications in computational biology, such as predicting targets of non-protein coding small RNAs [25] and analysing spacers in CRISPR for potential transfers from viruses or plasmids [22,24], to cite a few. More generally, introducing some errors would improve the sensitivity in the presence of sequencing errors or variants. Since we combine a high error rate with a small alphabet, we need to design a method with a good filtration efficiency. This is necessary to limit the number of false positives, thus the number of unnecessary verifications. In this paper, we present a new hybrid method where the pattern will be split in non-overlapping parts, some of them being searched without error, while others are searched with a limited number of errors.

In Section 2, we detail the novel type of seeds we will use and show some properties of those seeds. We also show that in practice those seeds are efficient filters. In Section 3, we explain how we make use of those seeds in our algorithm and how they are searched in a compressed index. In Section 4 we show some experimental results on DNA with random sequences and real data.

2 Approximate seeds for the Levenshtein edit distance

Let A be a finite alphabet. Given two strings u and v of A^*, 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 A. 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. 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 parts do not need to be of the same length. 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 A^* such that $lev(P,U) \leq k$. Then there exists i, j, $1 \leq i < j \leq k+2$, and $U_1, \ldots, U_{j-i-1}$ of A^* 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, $1 \leq \ell \leq j-i-1$, $lev(P_{i+\ell-1}, U_\ell) = 1$.*

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 remaining of the paper. XXXXX

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

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

where $lev^1(u)$ denotes the set of strings whose Levenshtein distance with u is 1.

The filtration efficiency is the primary criterion used to evaluate the performance of a seed. To estimate it, we generated an independent and identically distributed random sequence of length 10^8 over the DNA alphabet $\{A, C, G, T\}$ as well as 100 patterns of length 20. We then searched for our 01^*0 seed for $k = 3$. For each pattern, we counted the total number of occurrences of the seed in the text, including overlapping occurrences. The distribution is plotted in Figure 1-(a). The average number of observed occurrences per pattern is 6,665. To compare with exact seeds, we report analogous results obtained with filtration based on q-grams in the same text as well as the same collection of patterns. First, we divide the pattern in $k+1 = 4$ parts, leading to q-grams of length 5, which guarantees lossless filtration (Figure 1-(b)). We also divide the pattern in three parts, of lengths 6, 7, and 7 (Figure 1-(c)). This seed is less sensitive since it allows for some false negatives. In the first case, the average number of occurrences is 390,635, and in the latter case, it is 36,644. Both distributions are shown in Figure 1-(b,c). These empirical measurements show that the 01^*0 seed is significantly more selective than exact seeds, such as q-grams. Of course, this higher selectivity comes at the price of some additional work to locate seeds in the text. However, the fact that errors are not randomly distributed within the seed drastically reduces the combinatorics.

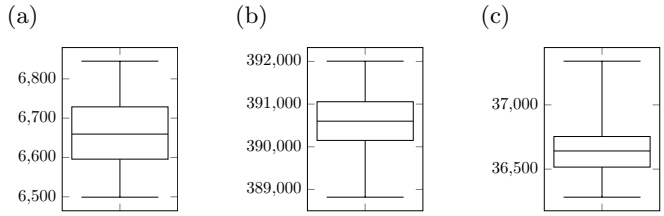


Fig. 1. Distribution of the number of occurrences of three different seeds for 100 patterns of size 20 in a random sequence of length 10^8. This filtration is done (a) with the 01^*0 seed for $k = 3$ and $m = 20$, (b) by dividing the pattern in 4 parts of length 5, (c) by dividing the pattern in 3 parts of lengths 6, 7, and 7, respectively. For each box plot, the bottom and top of the box are the first and third quartiles. The band inside the box is the median, and the ends of the whiskers represent the minimum and maximum of all of the data. There is on average 26.85 occurrences of the whole pattern within 3 errors.

3 Algorithm

Let T be a text over the alphabet A^*. The problem we consider now is that of finding matches of P with at most k errors in T. For this we devise an efficient filtration algorithm based on the seeding framework introduced in Section 2. It is necessary to keep in mind that we want to search small patterns (several dozens of letters) in large texts (millions or billions of letters) with small alphabets (*e.g.* DNA). We first justify our choice of using a FM-index. Then we explain how seeds are searched for and how they are extended when necessary.

3.1 Choice of index

As we have biological applications in mind (*e.g.* searching small DNA sequences on large genomes), we are in the situation where the text is known in advance. Moreover, we may have millions of short sequences to be queried in the text. This situation is particularly suitable for text indexes.

Since patterns do not have fixed sizes, full-text indexes are more appropriate. Furthermore, to limit space consumption, compressed indexes appear to be the indexes of choice. Among compressed indexes, FM-indexes [8] 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 [7].

3.2 Seed filtration

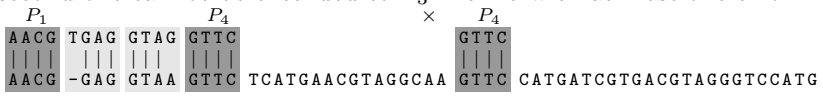
Given a pattern P, we enumerate all possible subseeds for the pattern. Each subseed for P is characterized by two parts P_i and P_j, $1 \leq i < j \leq k + 2$, that occur exactly in the text. According to Lemma **??**, all the intervening parts between P_i and P_j must be searched with exactly one error. 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 [12]). When a part is found exactly, we know that P_i has been reached. Starting with P_ℓ, we can have several parts that fulfill 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).

Example 1. Let us continue with Example **??**, also shown in Figure **??**: $k = 3$ and
$P =$ `AACG TGAG GTAG GTTC CATG`, which is partitioned into 5 parts of equal length. Assume that this text 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.

– With $j = 4$, `GTTC` occurs exactly in the FM-index, and correspond 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.



Note that in this particular case, the first occurrence of P in T is covered by two overlapping 01^*0 seeds, 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.

– 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 elongation 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. Via Lemma **??**, we know that there are at least $k - j + 2$ errors in $P_{j+1} \dots P_{k+2}$. So, since the total number of errors should not exceed k, there should be at most $i - 1$ errors in $P_1 \dots P_{i-1}$. As a consequence, each seed instance is 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 to filter out candidates. 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 [9]). Once this extension is performed, the occurrences of $P_1 \dots P_j$ 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. Let us assume that an instance of a given prefix $P_1 \dots P_j$ has been found with e errors in the FM-index. Thus, $P_{j+1} \dots P_{k+2}$ must be searched with at most $k - e$ errors in the text. Therefore, the bandwidth is $2 \times (k - e) + 1$ 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 [21,3]. 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

Our algorithm was implemented in a software called Bwolo. Bwolo is written in C++, with the help of SeqAn library and the the FM-Index implemented within [6]. It is open source and can be downloaded from `http://bioinfo.lifl.fr/bwolo`. In this implementation, patterns are divided into parts whose length differ by at most one character.

4 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 pair-wise sequence alignment, which uses exact sparse dynamic programming to perform the search. [23]. 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 [26]. It performs the verification via an implementation of the improved Myers bit-vector algorithm proposed by Hyyrö [10]. RazerS3 works

without a precomputed index for the text. So, we also selected Bowtie2 [11], that, like our tools, indexes the text with an FM-index. It then uses backtracking for handling errors and dynamic programming to build the full alignment. Lastly, we used an in-house implementation for approximate search in an FM-index written with the SeqAn library. It is based on a simple breadth-first search method with no prior filtration. Unfortunately, we were not able to include hybrid methods described in [19] in our benchmark, since the implementation is not available.

All these tools were configured to be full sensitive and output all occurrences of the pattern: option `--exhaustive` for Exonerate, `--filter pigeonhole --percent-identity` $[Id]$ `--recognition-rate 100` such that $[Id] = 100 \times (1 - \frac{k}{m})$ for RazerS3 and `-a -L` $[Seeds]$ `-i C,`$[Seeds]$`,0` such that $[Seeds] = \frac{m}{k+1}$ for Bowtie2. 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.

4.1 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 20 nt at random and measured the computation time of each tool for $k = 2$ and $k = 3$. Results are shown in Figure 2.

In both cases, Bwolo is the fastest tool for long sequences, from 10^6 nt. As expected, the added-value of Bwolo is even more obvious when $k = 3$. 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 have 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 [16]. 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.

For $k = 2$, we can observe that there are fewer 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. Loading the index from disk (which is the same in both cases) and unserializing the data structures takes 12 s on that same sequence. Ignoring the loading of the index 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 RazerS3 takes 8s to load the 1GB sequence from disk.

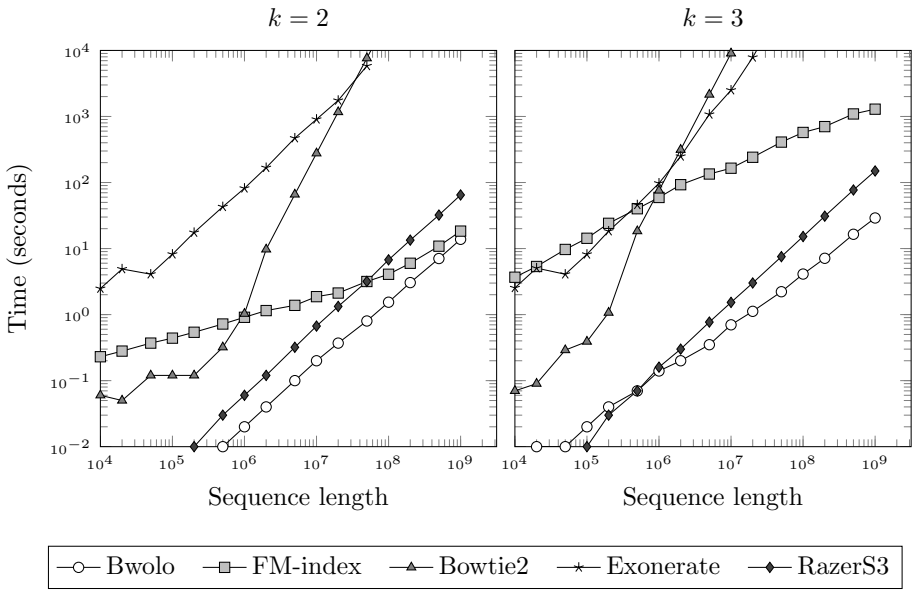


Fig. 2. Running time for 100 randomly generated sequences. Both axes are in logarithmic scale. Bwolo is our algorithm. FM-index refers to the breadth-first search implementation in a FM-index.

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 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.

4.2 Reads from the Human genome

In order to test our algorithm with an external dataset made of short sequences we relied on the work done by Schbath *et al* [20]. Their $\mathcal{H}_3$ dataset contains 10 millions of reads of length 40nt 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 Bowtie2 on the full dataset within a reasonable amount of time, we also used a random selection of 10,000 reads. Table 1 shows the results. As in the previous test, Bwolo achieves the best performances. However, the difference between Bwolo and RazerS3 was 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 in the previous test.

	index construction		$10,000$ reads		10^7 reads	
	time	memory	time	memory	time	memory
Bwolo	$7,594$	$9,584$	97	$6,522$	$55,493$	$9,054$
RazerS3	0	0	502	$6,469$	$467,413$	$152,045$
Bowtie2	$10,584$	$5,379$	$156,164$	$8,260$	NA	NA

Table 1. Running time on the Human genome benchmark. All times are in seconds, and the memory in Mo. NA: non available.

5 Conclusion

We have introduced a new seed framework, which we named 01^*0 seeds. These seeds achieve a good balance between the filtration step and the verification effort. Moreover, we have shown that they can be efficiently searched in a compressed full-text index, such as the FM-index. We believe that this method is especially well-suited to deal with patterns containing a high rate of errors and constitutes a promising alternative to existing algorithms. In this paper, we chose to show how to apply these seeds to searching a preprocessed text stored in an index. Our results offer some other perspectives. For instance, when dealing with a large collection of patterns, preprocessing them would allow us to take advantage of the parts that are shared among several patterns in order to speed up the algorithm. The filtration algorithm could also be applied online. Identifying the 01^*0 seeds requires us to identify an exact part first, which we then extend to the 1^* parts. This can be performed efficiently using bit-wise operations. Once the seeds are identified, we can compute the left and right extensions using a bit-parallel algorithm [14].

The generalisation of 01^*0 seeds to $(01^*)^\varepsilon 0$ could also be promising in further studies. This would not be as straightforward as one would think, since splitting the pattern in $k+1+\varepsilon$ parts is not sufficient.

Finally, albeit having been beyond the scope of this paper, an important aspect to thoroughly analyze would be the average case of our algorithm, as Baeza-Yates and Perleberg did in [1].

References

1. Baeza-Yates, R.A., Perleberg, C.H.: Fast and practical approximate string matching. Information Processing Letters 59(1), 21–27 (1996)
2. Belazzougui, D.: Improved space-time tradeoffs for approximate full-text indexing with one edit error. Algorithmica pp. 1–27 (2014)
3. Belazzougui, D., Cunial, F., Kärkkäinen, J., Mäkinen, V.: Versatile succinct representations of the bidirectional Burrows-Wheeler transform. In: Bodlaender, H.L., Italiano, G.F. (eds.) Algorithms – ESA 2013, pp. 133–144. No. 8125 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (2013)
4. Chan, H.L., Lam, T.W., Sung, W.K., Tam, S.L., Wong, S.S.: A linear size index for approximate pattern matching. J. of Discrete Algorithms 9(4), 358–364 (2011)
5. Chávez, E., Navarro, G.: A Metric Index for Approximate String Matching. In: Rajsbaum, S. (ed.) LATIN 2002: Theoretical Informatics, pp. 181–195. No. 2286 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (2002)
6. Döring, A., Weese, D., Rausch, T., Reinert, K.: SeqAn an efficient, generic C++ library for sequence analysis. BMC bioinformatics 9(1), 11–19 (2008)
7. Ferragina, P., González, R., Navarro, G., Venturini, R.: Compressed text indexes: From theory to practice. J. of Experimental Algorithmics (JEA) 13, 12 (2009)
8. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM (JACM) 52(4), 552–581 (2005)
9. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. on Alg. (TALG) 3(2) (2007)
10. Hyyrö, H.: A Bit-vector Algorithm for Computing Levenshtein and Damerau Edit Distances. Nordic J. of Computing 10(1), 29–39 (2003)
11. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with Bowtie 2. Nature methods 9(4), 357–359 (2012)
12. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics (Oxford, England) 25(14), 1754–1760 (2009)
13. Maaß, M.G., Nowak, J.: Text indexing with errors. J. of Discrete Algorithms 5(4), 662–681 (2007)
14. Myers, G.: A Fast Bit-vector Algorithm for Approximate String Matching Based on Dynamic Programming. J. ACM 46(3), 395–415 (1999)
15. Navarro, G.: A guided tour to approximate string matching. ACM computing surveys (CSUR) 33(1), 31–88 (2001)
16. Navarro, G., Baeza-Yates, R.: A Hybrid Indexing Method for Approximate String Matching. J. of Discrete Algorithms 1, 19–27 (2001)
17. Navarro, G., Sutinen, E., Tanninen, J., Tarhio, J.: Indexing text with approximate q-grams. In: Combinatorial Pattern Matching. pp. 350–363. Springer (2000)
18. Petri, M., Culpepper, J.S.: Efficient indexing algorithms for approximate pattern matching in text. In: Proceedings of the Seventeenth Australasian Document Computing Symposium. p. 9–16. ADCS '12, ACM, New York, NY, USA (2012)
19. Russo, L., Navarro, G., Oliveira, A.L., Morales, P.: Approximate string matching with compressed indexes. Algorithms 2(3), 1105–1136 (2009)
20. Schbath, S., Martin, V., Zytnicki, M., Fayolle, J., Loux, V., Gibrat, J.F.: Mapping Reads on a Genomic Sequence: An Algorithmic Overview and a Practical Comparative Analysis. Journal of Computational Biology 19(6), 796–813 (2012)
21. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional search in a string with wavelet trees. In: CPM, pp. 40–50. No. 6129 in LNCS, Springer (2010)

22. Shah, S.A., Hansen, N.R., Garrett, R.A.: Distribution of CRISPR spacer matches in viruses and plasmids of crenarchaeal acidothermophiles and implications for their inhibitory mechanism. Biochemical Society Transactions 37(1), 23 (2009)
23. Slater, G.S.C., Birney, E.: Automated generation of heuristics for biological sequence comparison. BMC Bioinformatics 6, 1–11 (2005)
24. Stern, A., Keren, L., Wurtzel, O., Amitai, G., Sorek, R.: Self-targeting by CRISPR: gene regulation or autoimmunity? Trends in genetics 26(8), 335–340 (2010)
25. Storz, G., Altuvia, S., Wassarman, K.M.: An abundance of rna regulators. Annu. Rev. Biochem. 74, 199–217 (2005)
26. Weese, D., Holtgrewe, M., Reinert, K.: RazerS 3: Faster, fully sensitive read mapping. Bioinformatics 28(20), 2592–2599 (2012)
27. Wu, S., Manber, U.: Fast text searching: allowing errors. Communications of the ACM 35(10), 83–91 (1992)