Approximate pattern matching using search schemes and in-text verification

Luca Renders[0000−0002−2244−1427], Lore Depuydt[0000−0001−8517−0479], and Jan Fostier[0000−0002−9994−8269]

Ghent University - imec, Technologiepark 126, 9052 Ghent, Belgium
{luca.renders,lore.depuydt,jan.fostier}@ugent.be

Abstract. Search schemes enable the efficient identification of all approximate occurrences of a search pattern in a text. Using a bidirectional FM-index, search schemes describe how to explore the search space in such a way that runtime is minimized. Even though in-index matching has an optimal time complexity, relatively expensive random memory access is required for elementary operations on the FM-index. We analyze to what extent in-index matching can be complemented with in-text verification where a candidate occurrence is directly validated in the text using a bit-parallel, pairwise alignment procedure. We find that hybrid in-index/in-text matching can reduce the running time by more than a factor of two, compared to pure in-index matching. We present Columba 1.1, an open-source (AGPL-3.0 license) software tool written in C++ that efficiently implements these ideas. Using a single CPU core, Columba 1.1 can identify, within a maximum edit distance of four, all occurrences of 100 000 Illumina reads (150 bp) in the human reference genome in roughly half a minute. This significantly outperforms existing, state-of-the-art tools.

Keywords: Lossless sequence alignment · FM-Index · bit-parallel alignment · in-text validation

1 Introduction

Approximate pattern matching is a well-studied problem in computer science and central to many bioinformatics applications. It involves identifying occurrences of a search pattern P in a (much) larger text T. For example, in a typical setting, P could be a short DNA fragment (a read) and T a (collection of) reference genome(s). Due to sequencing errors and genetic diversity among individuals, one is often interested in finding *approximate* occurrences of P in T.

Historically, *lossy* approximate pattern matching algorithms gained a lot of popularity. Such algorithms rely on heuristics to quickly identify *some* (but not necessarily *all*) approximate matches of P in T. By sacrificing some sensitivity, significant performance gains can be obtained. As such, lossy algorithms are used in many state-of-the-art alignment tools such as BLAT [8], BLAST [2], BWA [12], etc. In contrast, in this paper, we focus on *lossless* algorithms which

are guaranteed to retrieve *all* approximate matches of P in T under a certain error distance metric. Specifically, the k-mismatch problem involves identifying all occurrences of P in T with up to k errors. Under the Hamming distance metric, only substitutions are allowed whereas the Levenshtein/edit distance metric allows substitutions, insertions, and deletions. In this work, we focus on the edit distance metric.

Full-text indexes such as suffix trees [5], enhanced suffix arrays [1] and FM-indexes [4] are used within numerous bioinformatics tools [18]. They allow for unidirectional, exact pattern matching, one character at a time, with a runtime proportional to the length of the search pattern and independent of the size of T. A naive approach to lossless approximate pattern matching would be to explore all possible branches in the index (called *backtracking*) within the maximum allowed Hamming/Levenshtein distance of search pattern P. This approach has two problems: a) the number of branches to explore increases rapidly with k and b) the vast majority of branches that are explored eventually turn out not to be matches.

A bidirectional index (such as the affix tree [13], the affix array [24] and the bidirectional FM-index [11]) augments the functionality of its unidirectional counterpart by allowing patterns to be matched in both directions: left-to-right and right-to-left. Using, e.g., a bidirectional FM-index, a query pattern can be searched by starting at any arbitrary position of that pattern and extending the match either to the left or to the right in arbitrary order. More formally, a (partial) match P can be extended by a single character c to either cP or Pc. In 2009, Lam et al. were the first to note that this added functionality opens up new possibilities for faster lossless approximate matching [11]. Leveraging the classical pigeonhole principle, they partitioned P into $k + 1$ parts, from which immediately follows that any approximate occurrence with at most k errors, must have an exact match with at least one of these parts. By first performing an exact search for one part of P (which maps to a single branch of the index) and then extending this partial match with an approximate search (backtracking), significant computational gains are obtained. This idea was generalized by Kucherov et al. who introduced the concept of *search schemes* [10]. Informally, search schemes define how a pattern P is matched using a bidirectional index, such that unsuccessful branches are discarded as quickly as possible and, hence, the runtime is minimized. Kucherov et al. also proposed a number of efficient search schemes with $k + 1$ and $k + 2$ parts for up to $k = 4$ errors. Kianfar et al. [9] further extended this work and used integer linear programming (ILP) to generate additional search schemes for the Hamming distance metric. Additionally, they show that related work on lossless approximate pattern matching by Vroland et al. on 01*0 seeds [26] can also be expressed as search schemes. Therefore, search schemes represent a flexible framework for lossless approximate pattern matching in which a multitude of algorithmic ideas can be expressed.

Recently, we proposed Columba [21], an efficient software tool for lossless approximate pattern matching using arbitrary search schemes. We proposed an algorithm for the dynamic partitioning of search patterns to further reduce the

search space and used an efficient memory layout for the data structures that underlie the FM-index. In this paper, we further build upon this work and we make the following contributions:

1. We adapted the search schemes by Kucherov et al. with $k + 1$ parts by imposing more stringent lower bounds on the cumulative number of errors in the different parts of the search pattern while maintaining the guarantee that all possible error distributions are covered. These adapted search schemes reduce the runtime by nearly 15%.
2. We adopt the bit-parallel, pairwise alignment algorithm by Hyyrö [7]. This algorithm is used to accelerate edit distance computations during in-index matching. Additionally, it is applied to in-text verification where a candidate occurrence of the search pattern is assessed directly in the text T. We show that using hybrid in-index matching/in-text verification can reduce the runtime by half compared to using only in-index matching.
3. We developed Columba 1.1, an open-source implementation in standard C++11 in which the above techniques were implemented. We demonstrate that our implementation is several times faster than other state-of-the-art lossless alignment algorithms such as GEM [14] and Bwolo [26] for the task of identifying all occurrences of 150 bp Illumina reads in the human reference genome within an edit distance of $k = 4$. We show that Columba 1.1 is faster than BWA in mem mode for $k = 1, 2$ and 3 and has a similar runtime for $k = 4$. Columba 1.1 is available at https://github.com/biointec/columba under AGPL-3.0 license.

This paper is organized as follows. In Section 2, we briefly describe the (bidirectional) FM-index and search scheme functionality. Section 3 introduces the adapted search schemes that are used throughout this work. In Sections 4 and 5, we provide the key algorithms for bit-parallel edit distance computations and their application to in-text verification, respectively. Finally, Section 6 provides performance benchmarks of Columba as well as existing state-of-the-art tools.

2 Preliminaries

2.1 Bidirectional FM-index

In this paper, we use zero-based array indexing, half-open intervals [.,.) and standard notation on strings. A text $T[0, n)$ of size n, which ends with a unique sentinel character \$ (defined as the lexicographically smallest character), has a Burrows-Wheeler transform $\mathrm{BWT}[0, n)$, which is defined as $\mathrm{BWT}[i] = T[\mathrm{SA}[i] - 1]$ if $\mathrm{SA}[i] > 0$ and $\mathrm{BWT}[i] =$ \$ otherwise [3]. Here, SA denotes the suffix array of text T, defined as a permutation over $\{0, 1, \ldots, n - 1\}$, such that $\mathrm{SA}[i]$ is the starting position of the lexicographically i-th suffix of T. To perform exact and approximate matching, we need support for $\mathtt{occ}(c, i)$ queries on the BWT, that return the number of occurrences of a character c in the prefix $\mathrm{BWT}[0, i)$. This is realized through $|\Sigma|$ (where Σ denotes the alphabet) bit vectors with

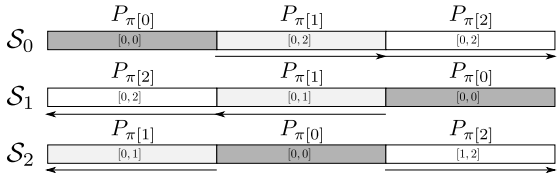

Fig. 1. Search scheme for $k = 2$ errors and 3 parts proposed by Kucherov et al. The parts are processed from darkest to lightest shade of gray. In each part, the lower and upper bound to the cumulative number of errors up to and including that part, are indicated. The arrows indicate the search direction (left-to-right or right-to-left).

constant-time rank support. Exact matching can then be performed by matching character by character from right to left. Consider an interval $[b, e)$ over the suffix array for which the corresponding suffixes are prefixed by P. In order to do exact matching backwards, we want to find interval $[b', e')$ whose corresponding suffixes are prefixed by cP. This can be computed as follows: $b' = C(c) + \texttt{occ}(c, b)$ and $e' = C(c) + \texttt{occ}(c, e)$, where $C(c)$ denotes the number of characters in $\text{BWT}[0, n)$ that are smaller than c. These are pre-computed and stored in a small array of size $|\Sigma|$. Since `occ` queries rely only on constant-time rank operations, exact matching of a pattern P takes $O(|P|)$ time. The number of occurrences of P in T is equal to the size of the interval $[b, e)$, i.e., $e - b$. The positions of these occurrences in T are then found using the suffix array. One can opt to use a sparse version of the suffix array, where $\text{SA}[i]$ is stored only when $\text{SA}[i]$ is a multiple of a pre-defined sparseness factor s. A length-n bit vector B is stored alongside the sparse suffix array to indicate for each index i if $\text{SA}[i]$ is stored. The value $\text{SA}[i]$ for arbitrary i can be inferred in $O(s)$ time. For details of this procedure, we refer to e.g. [20]. The FM-index is a full-text index that comprises a BWT representation and auxiliary tables and that may occupy as little as 2-4 bits of memory per character for DNA sequences [4].

In 2009, the *bidirectional* FM-index was introduced [11]. By also storing BWT_r, the Burrows-Wheeler transform of the reverse of T, and keeping track of both the range $[b, e)$ over the BWT as well as the range $[b', e')$ over BWT_r in a synchronized manner, P can be extended backwards (to cP) or forwards (to Pc). By replacing the 'occ' data structure with a so-called 'Prefix-Occ' structure, both can be done in $O(1)$ time [19].

2.2 Search schemes

To perform lossless approximate pattern matching with up to k errors one needs to explore all the branches of the FM-index that could potentially be matches. Using a naive backtracking approach, an excessive number of unsuccessful branches near the dense root of the search tree will be explored, rendering backtracking computationally unfeasible even for modest values of k. To alleviate this, Kucherov et al. proposed *search schemes* [10]. We adopt their notation. A pattern P is partitioned into p parts P_i $(i = 0 \dots p - 1)$. A *search* $\mathcal{S}$ is a

triplet of arrays (π, L, U) of size p. Here, π is a permutation over $\{0, ..., p-1\}$ that defines the order in which the parts P_i are processed. In order to constitute a valid search scheme, π must satisfy the connectivity property, i.e., a partial match can only be extended in a contiguous manner, either to the left or to the right. The arrays L and U respectively define the lower and upper bound to the cumulative number of errors after each part is processed. The core idea of search schemes is that the number of allowed errors is only gradually increased. This significantly reduces the search space near the dense root of the search tree. To cover all possible error distributions over the length of a pattern, multiple *searches* are required that collectively form a search scheme. We denote an error distribution for p parts and at most k errors as $e_0 e_1 \dots e_{p-1}$, with $\sum_{i=0}^{p-1} e_i \leq k$, where e_i is the number of errors in part P_i. In order for a search scheme for p parts and at most k errors to be valid, all possible error distributions need to be covered by at least one search.

For example, for $k = 2$ errors, Kucherov et al. proposed a search scheme with three searches: $\mathcal{S}_0 = (012, 000, 022)$; $\mathcal{S}_1 = (210, 000, 012)$; $\mathcal{S}_2 = (102, 001, 012)$ (see Fig. 1). In the $\mathcal{S}_0$ search, exact matching is first performed for the leftmost part P_0. Next, this exact match is extended to the right, thus processing parts P_1 and P_2, using a backtracking procedure that allows up to two errors. In the $\mathcal{S}_1$ search, exact matching is first performed for the rightmost part P_2, and extended to the left by first allowing up to a single error in P_1, and then two errors in P_0. Indeed, occurrences of P with two errors in the middle part were already covered by search $\mathcal{S}_0$. Finally, search $\mathcal{S}_2$ first involves an exact matching of P_1, which is then extended to the left allowing a single error, and finally to the right with at least one, and at most two errors. This search also explains the need for bidirectional matching functionality. Kucherov et al. [10] and Kianfar et al. [9] proposed search schemes for up to $k = 4$ errors.

3 Adapted search schemes

In earlier work [21], we concluded that the search schemes by Kucherov et al. with $p = k + 1$ parts showed the best performance for the task of identifying occurrences of Illumina reads in the human reference genome under an edit distance constraint. However, it appears that for some searches $\mathcal{S} = (\pi, L, U)$, the lower bound array L can be made more stringent, while maintaining the guarantee that collectively, all searches within the search scheme cover all possible error distributions over a pattern. Recall that when part P_i has been processed, the cumulative number of errors must be between $L[i]$ and $U[i]$. The benefit of the adapted search schemes is twofold: 1) if fewer error distributions of a search pattern are covered by multiple searches, the number of redundant occurrences decreases, reducing the time to filter them and 2) by making the lower bounds more stringent, the search space that needs to be explored decreases. The original and adapted search schemes are presented in Table 1.

Table 1. The original search schemes by Kucherov et al. for $p = k + 1$ parts and our adapted search schemes for $k = \{1, 2, 3, 4\}$ errors. Changes are highlighted in bold.

k	Original	Adapted
1	(01, 00, 01); (10, 01, 01)	(01, 00, 01); (10, 01, 01)
2	(012, 000, 022); (210, 000, 012); (102, 001, 012)	(012, 0**12**, 022); (210, 000, 012); (102, 001, 012)
3	(0123, 0000, 0133); (1023, 0011, 0133) (2310, 0000, 0133); (3210, 0011, 0133)	(0123, 000**2**, 0133); (1023, 0**11**3, 0133) (2310, 0000, 0133); (3210, 0**1**11, 0133)
4	(01234, 00000, 02244); (43210, 00000, 01344); (10234, 00133, 01334); (01234, 00133, 01334); (32410, 00011, 01244); (21034, 00013, 01244); (10234, 00124, 01244); (01234, 00034, 00444);	(01234, 0000**2**, 02244); (43210, 00000, 01344); (10234, 0**13**34, 01334); (01234, 00**3**34, 01334); (32410, 00**11**1, 01244); (21034, 00**1**13, 01244); (10234, 0**12**24, 01244); (01234, 00**34**4, 00444)

4 Bit-parallel edit distance computation

To enable approximate pattern matching, we rely on edit distance computations. The edit distance between two sequences S_1 and S_2 of lengths m and n, respectively, can be computed in $O(mn)$ time using a dynamic programming algorithm. This entails computing an $(m+1) \times (n+1)$ matrix D such that each element $D(i, j)$ represents the edit distance between prefix $S_1[0 \dots i)$ and prefix $S_2[0 \dots j)$. The values $D(i, j)$ are efficiently computed by following recurrence relation:

$$D(i, 0) = i; D(0, j) = j \qquad \forall i, j \geq 0$$

$$D(i, j) = \min \begin{cases} D(i-1, j-1) + \delta(S_1[i-1], S_2[j-1] \\ D(i-1, j) + 1 \\ D(i, j-1) + 1 \end{cases} \qquad \forall i, j > 0$$

where $\delta(a, b)$ is 0 if $a = b$ and 1 otherwise. The oldest description of this algorithm is by Vintsyuk [25] in 1968; it has been independently rediscovered by others (see e.g. [17] and the references therein). Myers [16] improved the time complexity to $O(mn/w)$, where w denotes the computer word size ($w = 64$ for most CPU architectures). The core idea is to leverage bit-level parallelism to compute multiple values of matrix D simultaneously. Inspired by Myers work, Hyyrö [6] proposed a slightly more efficient bit-parallel algorithm. We first provide a brief description of this algorithm. Next, we describe our specific adaptations.

4.1 Hyyrö's bit-parallel algorithm

Adjacent elements within any row or column of matrix D differ by at most a value of 1, i.e., for all i, j: $D(i, j) - D(i, j-1) \in \{-1, 0, 1\}$ and $D(i, j) - D(i-1, j) \in \{-1, 0, 1\}$ (see [15], lemma 3). Similarly, for adjacent elements on a diagonal, it holds that $D(i, j) - D(i-1, j-1) \in \{0, 1\}$. Rather than computing the values of D directly, each row i is encoded by five delta vectors VP_i, VN_i, HP_i, HN_i,

and $\mathrm{D0}_i$. These delta vectors are stored as bit vectors (i.e., a sequence of 0s and 1s) and are defined as follows:

1. The vertical positive delta vector: $\mathrm{VP}_i[j] = 1 \iff D(i,j) - D(i-1,j) = 1$
2. The vertical negative delta vector: $\mathrm{VN}_i[j] = 1 \iff D(i,j) - D(i-1,j) = -1$
3. The horizontal positive delta vector: $\mathrm{HP}_i[j] = 1 \iff D(i,j) - D(i,j-1) = 1$
4. The horizontal negative delta vector: $\mathrm{HN}_i[j] = 1 \iff D(i,j) - D(i,j-1) = -1$
5. The diagonal zero delta vector: $\mathrm{D0}_i[j] = 1 \iff D(i,j) - D(i-1,j-1) = 0$

The bits $\mathrm{HP}_i[j]$ and $\mathrm{HN}_i[j]$ encode the value $D(i,j) - D(i,j-1)$. The latter equals either 1 (when $\mathrm{HP}_i[j] = 1$), -1 (when $\mathrm{HN}_i[j] = 1$), or 0 (when both $\mathrm{HP}_i[j] = 0$ and $\mathrm{HN}_i[j] = 0$). Similarly, $\mathrm{VP}_i[j]$ and $\mathrm{VN}_i[j]$ encode the value $D(i,j) - D(i-1,j)$. Therefore, because $D(0,0)$ is known (often 0), all other values $D(i,j)$ can be inferred from the delta vectors.

The key advantage of using the delta vectors is that they can be computed in a bit-parallel manner as shown in Algorithm 1:

Algorithm 1: Bit-parallel computation of the delta vectors at row i from those at row $i-1$

$\mathrm{D0}_i \leftarrow (((\mathrm{M}_{S_1[i-1]} \,\&\, \mathrm{HP}_{i-1}) + \mathrm{HP}_{i-1}) \,\hat{}\, \mathrm{HP}_{i-1}) \mid \mathrm{M}_{S_1[i-1]} \mid \mathrm{HN}_{i-1}$
$\mathrm{VP}_i \leftarrow \mathrm{HN}_{i-1} \mid \sim(\mathrm{D0}_i \mid \mathrm{HP}_{i-1})$
$\mathrm{VN}_i \leftarrow \mathrm{D0}_i \,\&\, \mathrm{HP}_{i-1}$
$\mathrm{HP}_i \leftarrow (\mathrm{VN}_i << 1) \mid \sim(\mathrm{D0}_i \mid (\mathrm{VP}_i << 1))$
$\mathrm{HN}_i \leftarrow (\mathrm{D0}_i \,\&\, (\mathrm{VP}_i << 1))$

Here, the symbols &, |, $\hat{}$, $\sim$ and $<<$ respectively denote the bitwise AND, OR, XOR, NOT and left shift operators. $\mathrm{M}_{S_1[i-1]}$ is a match vector (again a bit vector) that indicates which positions in S_2 match character $S_1[i-1]$. The four match vectors M_c (with $c \in \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$) are pre-computed. For the exact details of Algorithm 1, we refer to [6].

4.2 Bit-parallel banded alignment

In the context of this work, we want to identify approximate occurrences within a distance of at most k edit operations of search pattern P. Therefore, computations can be restricted to those elements $D(i,j)$ for which $|i-j| \leq k$, i.e., within a band along the diagonal. Each row (or column) of matrix D thus contains at most $2k+1$ values to compute. For this problem of banded alignment, Hyyrö proposed a bit-parallel algorithm [6]. Our implementation is heavily influenced by these ideas but uses a different layout of bit vectors. It is described below.

The global layout of the banded dynamic programming matrix D is depicted in Fig. 2. Search pattern P is the 'horizontal' sequence while candidate occurrence O is the 'vertical' sequence. The FM-index spells out candidate occurrences character by character, therefore, we leverage bit-parallel computations at the

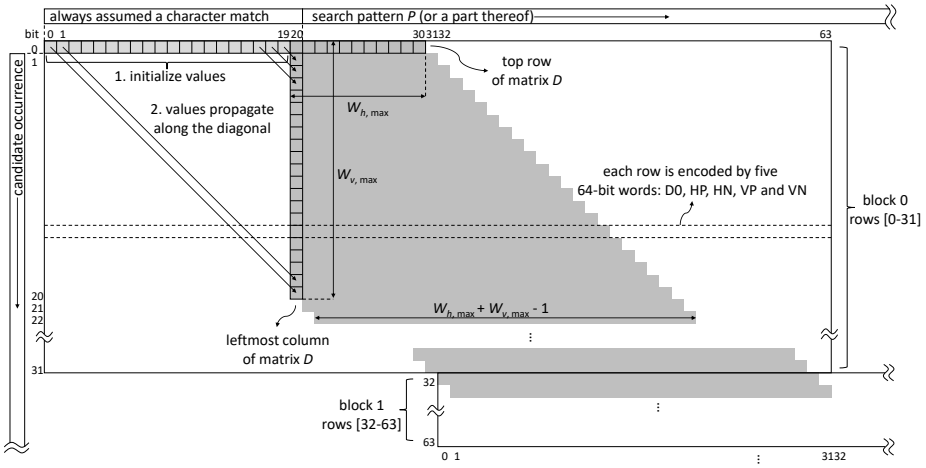

Fig. 2. Layout of the banded dynamic programming matrix D as 64-bit words.

level of *rows* of D. During in-index searching, candidate occurrences are generated by a depth-first exploration of the search tree. To support backtracking, the delta vectors of each row are kept in a stack data structure.

Our implementation can compute edit distance values up to $k = 10$ for sequences of arbitrary length. Because k is sufficiently small, a single 64-bits word can be used to represent a delta vector and all computations per row are done in $O(1)$ time. Support for larger values of k could easily be achieved by representing a delta vector by multiple words, at the cost of some loss of performance. Rows are grouped into *blocks* of 32 rows each. At each next block, the delta vectors are shifted by 32 bit positions such that they overlap all relevant values of the banded dynamic programming matrix (gray-shaded cells in Fig. 2). For each block, four match vectors M_c (with $c = \{\text{A, C, G, T}\}$) are pre-computed to indicate character matches between c and the overlapping positions of P. At each row i, we also keep track of the value $D(i, i)$. Using the D0_i delta vector, $D(i, i)$ can easily be computed from $D(i-1, i-1)$. The knowledge of $D(i, i)$ and the HP_i and HN_i delta vectors allows for the computation of any value $D(i, j)$. By using population count ('popcount') instructions, this can be achieved in $O(1)$ time. Finally, we adopted Hyyrö's algorithm to evaluate in a bit-parallel manner whether all values on a row exceed the maximum edit distance threshold k. This is important to signal the backtracking algorithm that the current candidate occurrence O should no longer be extended and that the search procedure should backtrack and explore a different branch of the search tree. For details on this algorithm, we refer to [7].

4.3 Matrix initialization

Traditionally, the first row and column of matrix D are initialized with gap penalties (i.e., $D(i, 0) = i$ and $D(0, j) = j$) in the case of global alignment, or

with zero values (i.e., $D(i,0) = 0$ and/or $D(0,j) = 0$) in case of semi-global alignment. For our use case of search schemes, we need to be able to initialize the leftmost column of D with $2k+1$ *arbitrary* values. Indeed, using search schemes, search pattern P is matched part by part. Therefore, assuming left-to-right matching, when matching part P_i, the first column of D should be initialized with the values from the last column of the matrix of part P_{i-1} in order to continue the alignment.

In the bit-parallel implementation, the initialization of the first row of D is straightforward: we set the appropriate value for $D(0,0)$ (e.g., $D(0,0) = 0$) and encode the other values $D(0,j)$ using the HP_0 and HN_0 delta vectors. For example, to encode $D(0,j) = j$, we set $\mathrm{HP}_0[j] = 1$ and $\mathrm{HN}_0[j] = 0$ for $j = 1 \ldots k$.

To initialize the first column of D with arbitrary values, we append dummy columns with a 'negative' column index to D (illustrated in a lighter shade of gray in Fig. 2). Again, we use the HP_0 and HN_0 delta vectors to encode the part of the first row of D with negative column indexes such that $D(0,-i)$ equals the desired value for $D(i,0)$. By always assuming a character match at negative column indexes, each value $D(0,-i)$ will effectively propagate along a diagonal and ultimately set $D(i,0)$ to its correct value. This is easily achieved by setting 1-bits in the corresponding part of M_c for all $c = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. Even in the presence of backtracking, the elements $D(i,0)$ will always be computed correctly. Because the computations for the negative column indexes are handled within the same 64-bit word as the regular column indexes, this procedure imposes no computational overhead.

Because we support a maximum allowed edit distance of 10, we require at most $W_{h,\max} = 11$ elements at the top row of D (e.g., to encode the values $\{0, 1, 2, \ldots, 10\}$ and at most $W_{v,\max} = 21$ elements at the leftmost column of D (e.g., to encode the values $\{10, \ldots, 1, 0, 1, \ldots, 10\}$). Thus, the parts of the delta vectors that *could* contain relevant values are indicated in a darker shade of gray in Fig. 2. Depending on the use-case (the actual allowed edit distance $k \leq 10$, and how precisely matrix D is initialized) only a subset of these cells will effectively contain relevant data.

5 In-text verification

In principle, search schemes rely purely on in-index matching: using the bidirectional FM-index, candidate occurrences O of a search pattern P are spelled character by character. Extending a candidate occurrence by a single character ultimately translates into rank operations on bit vectors. Collectively, these rank operations lead to a random memory access pattern. The expression *random memory access* refers to the fact that the memory access pattern is unpredictable, and hence, will suffer from a large number of cache misses. Therefore, extending a candidate occurrence by a character is a relatively expensive operation: Pockrandt et al. estimated at least 100 CPU clock cycles per character [20].

At all times during the spelling of a candidate occurrence O, a range $[b, e)$ over the suffix array is maintained that refers to the starting positions of each

instance of O in T. Thus, at any point, the size of the range $e-b$ corresponds to the number of times O occurs in T. This number of instances decreases monotonically when more characters are added to O. When the value $e-b$ becomes small, it can be beneficial to abandon the in-index matching procedure and to verify each of the instances of O directly in T using the previously described pairwise alignment procedure. As detailed in Section 4, pairwise alignment can be performed efficiently using bit-parallel techniques in a cache-friendly manner. In contrast, when the value $e-b$ is large, in-index matching is more computationally advantageous, because all instances of O of in T are handled simultaneously by the FM-index.

In our implementation, part $P_{\pi[0]}$ is always matched using the FM-index. In practice, matching $P_{\pi[0]}$ always entails an exact pattern matching procedure (see search schemes in Table 1). From that point onwards, whenever the value $e-b$ becomes smaller than or equal to a pre-defined threshold t (referred to as the 'tipping point'), candidate occurrence O is no longer extended using the index and the search procedure switches to in-text verification. When O has been fully evaluated, the search procedure will backtrack and explore other candidate occurrences, again using the FM-index.

This idea of hybrid in-index matching/in-text verification within the context of search schemes has been explored previously by Pockrandt et al. for the Hamming distance metric. The authors report speed-ups between 1.6× and 2.1× and an optimal tipping point of 25 [20]. Performing in-text verification for the edit distance metric is more complex because 1) pairwise alignment is computationally more expensive and thus needs to be highly optimized to have overall performance gains; 2) the precise start and end positions of each approximate occurrence of P in T are not known in advance. To this end, the bit-parallel alignment algorithm from section 4 is easily modified to support semi-global alignment.

6 Results and Discussion

All benchmarks were performed using a dataset of 100 000 Illumina NovaSeq 6000 reads (150 bp), randomly sampled from a larger whole genome sequencing dataset (accession no. SRR9091899). We identified all approximate read occurrences up to an edit distance of $k = \{1, 2, 3, 4\}$ on both strands of the human reference genome (GRCh38) [22]. We recall that we consider only lossless algorithms that are guaranteed to report all occurrences. We replaced non-ACGT characters in the reference genome (e.g., Ns) by a randomly chosen nucleotide. The different chromosomes were concatenated into a single string. As such, a read can be mapped across the borders of adjacent chromosomes. Such spurious matches can easily be filtered during post-processing.

All results were obtained using a single core of a 32-core Intel® Xeon® E5-2698 v3 CPU running at a base clock frequency of 2.30 GHz. To quantify variability in runtime, each benchmark run was repeated 20 times. We report both

Table 2. Comparison of the original search schemes by Kucherov et al. and our adapted search schemes, for different values of the maximum allowed edit distance k. In both cases, 100 000 Illumina reads of length 150 bp are mapped to both strands of the human reference genome.

Search scheme	Wall clock time ± SD	No. of nodes visited (search space)	No. of redundant occurrences
	$k = 2$, unique occurrences = 676 528, reads mapped 90.5%		
Original	15.91 ± 1.58 s	62 035 887	267 541
Adapted	14.73 ± 1.44 s (−7.4%)	57 263 477 (−7.7%)	264 671 (−1.1%)
	$k = 3$, unique occurrences = 1 416 632, reads mapped 93.1%		
Original	30.89 ± 1.80 s	128 708 469	719 576
Adapted	26.82 ± 0.60 s (−13.2%)	116 965 983 (−9.1%)	648 817 (−9.8%)
	$k = 4$, unique occurrences = 2 579 745, reads mapped 94.8%		
Original	72.07 ± 2.54 s	364 385 491	1 492 806
Adapted	61.35 ± 0.59 s (−14.9%)	305 476 323 (−16.2%)	1 420 668 (−4.8%)

the average wall clock time as well as the standard deviation. Redundant occurrences (as defined in [21]) were filtered.

6.1 Original versus adapted search schemes

In Table 2, the original and adapted search schemes (as defined in Table 1) are compared for edit distance values of $k = \{2, 3, 4\}$ as for $k = 1$, both search schemes are identical. We report the average runtime and standard deviation on a single CPU core and the number of nodes visited in the search tree. The latter equals the number of times a partial match is extended by a single character c (in either direction). In practice, this involves expensive random memory access that largely determines the runtime. It is therefore a clear indication of intrinsic performance, regardless of the quality of implementation. It is clear that both in the size of search space (number of nodes visited) and runtime the adapted search schemes are superior. This is no surprise, as the adapted search schemes have tighter bounds and thus reduce the search space.

Table 2 also reports the total number of unique and redundant (filtered out) occurrences for the different values of k. Because search schemes are lossless, the number of unique occurrences does not differ between the original and adapted search scheme. Clearly, the tighter lower bounds also reduce the number of redundant occurrences (i.e., occurrences reported by multiple searches in the search scheme).

Finally, Table 2 reports the fraction of reads that have at least one occurrence in the reference genome ('reads mapped'), for the different values of k.

6.2 In-index versus in-text verification

We compared the runtime for matching 100 000 Illumina patterns to both strands of the human reference genome with up to $k = 4$ edit operations for different

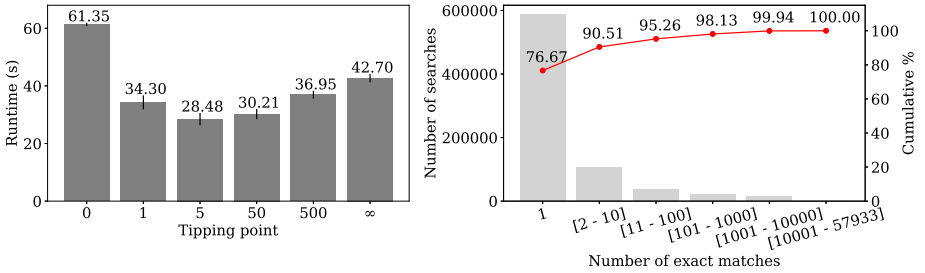

Fig. 3. Left: the runtime for mapping 100 000 Illumina reads of length 150 bp to both strands of the human reference genome ($k = 4$) as a function of the tipping point t. Right: histogram of the number of matches for part $P_{\pi[0]}$ across all searches.

values of the tipping point $t = 0$, 1, 5, 50, 500 and ∞. A value of $t = 0$ means that all patterns are entirely matched using the FM-index and that no in-text verification is performed whereas $t = \infty$ denotes that after the initial matching of the first part $P_{\pi[0]}$, all candidate occurrences are verified directly in T and that no further in-index extension takes place. For the intermediate tipping point values, the search procedure switches to in-text verification when $e - b \leq t$.

Figure 3 (left) shows the runtime as a function of tipping point t. Clearly, using purely in-index matching shows the worst performance for this particular dataset. This is because in-index matching involves expensive random memory access in the FM-index for each character that is added to a candidate occurrence. Switching to in-text verification when there is only a single candidate occurrence in T ($t = 1$) reduces runtime by almost half. This is because bit-parallel, pairwise alignment between the appropriate substring of T and P can be performed very efficiently. This effect increases with larger tipping point values and for $t \approx 5$, runtime is minimized. For larger tipping point values ($t \geq 50$), the increasing overhead of suffix array lookup operations and pairwise alignments associated with in-text verification (that often turn out to be unsuccessful) dominates the gains. Remarkably, for this dataset, never performing in-index extension beyond the exact matching of the first part $P_{\pi[0]}$ ($t = \infty$) is still significantly faster than pure in-index matching ($t = 0$). For $t = \infty$, the matching process degenerates to a very simple procedure: exact pattern matching of part $P_{\pi[0]}$ followed by in-text verification of each of the candidate occurrences. For our dataset, the largest suffix array range size encountered was 57 933. This range was encountered for a single read for which $P_{\pi[0]}$ consists of 29 consecutive characters A.

Collectively over all reads, a tipping point t between 2 and 10 yields the best performance. Within this range and for our dataset, the runtime is largely insensitive to the precise choice of t (data not shown). Only for larger values of the tipping point ($t \geq 10$), we again observe an increase in runtime. For other values of k, a similar conclusion is reached: hybrid in-index matching/in-text

verification reduces runtime by 38.43% for $k = 1$, 45.24% for $k = 2$ and 51.30% for $k = 3$.

Breakdown of reads The search scheme for $k = 4$ errors consists of eight searches (see Table 1). Therefore, for the task of identifying all approximate occurrences of 100 000 reads on both strands of the reference genome, 1 600 000 searches are executed in total. For more than half of these searches (834 198), the first part $P_{\pi[0]}$ has no exact match in T and, hence, the search will immediately be terminated. This is no surprise, as most reads have approximate occurrences on only one strand of the reference genome. For the remaining 765 802 searches, Fig. 3 (right) shows a breakdown as a function of the number of (exact) occurrences of part $P_{\pi[0]}$. Remarkably, 76.67% (587 103) of those searches yield only a single occurrence in T for $P_{\pi[0]}$. In other words, for most reads, matching only a single part of P already suffices to point to a unique position in T. For such cases, in-text verification of that sole candidate occurrence outperforms a further in-index character-by-character extension. This explains the large performance difference between tipping point values $t = 0$ and $t = 1$. Additionally, 13.84% (106 022) of the searches yield between 2 and 10 occurrences in T for part $P_{\pi[0]}$. Also for these cases, in-text verification at each of these candidate positions in T is superior to in-index matching.

In contrast, only a relatively small fraction of 9.49% (72 677) of the searches deal with patterns for which $P_{\pi[0]}$ has more than 10 occurrences in T. In certain cases, this number of instances is vast. For example, 14 329 searches yield more than 1 000 instances of $P_{\pi[0]}$ in T, seven of which amount to more than 50 000 instances. The latter all correspond to low-complexity poly-A/T or poly-CA/GT patterns which are highly repeated in the human genome. Here, in-index matching has a clear advantage as all repeated candidate occurrences are handled simultaneously by the FM-index.

We conclude that in-text verification is beneficial for those searches for which the number of occurrences of $P_{\pi[0]}$ in T (and hence, the number of candidate occurrences of P itself), is limited (≤ 10). For our dataset, this holds for roughly 90% of the searches. In contrast, the remaining searches (10%) deal with search patterns with many potential occurrences in T, a task which is best performed using in-index matching and the search scheme. We find that these 'difficult' searches, although limited in number, account for roughly two-thirds of the total runtime. In total, these complex searches account for 96.0% of unique matches over the entire dataset.

SA space-time tradeoff In-text verification requires a lookup operation in the suffix array (SA) to retrieve, for each candidate occurrence, its position in T. The number of candidate occurrences for which in-text verification is performed, and hence, the number of required lookup operations in the SA, increases with higher values of the tipping point t.

To reduce the memory footprint of the FM-index, a sparse version of the SA is often used. In our implementation, every s-th suffix of the SA is stored, where s

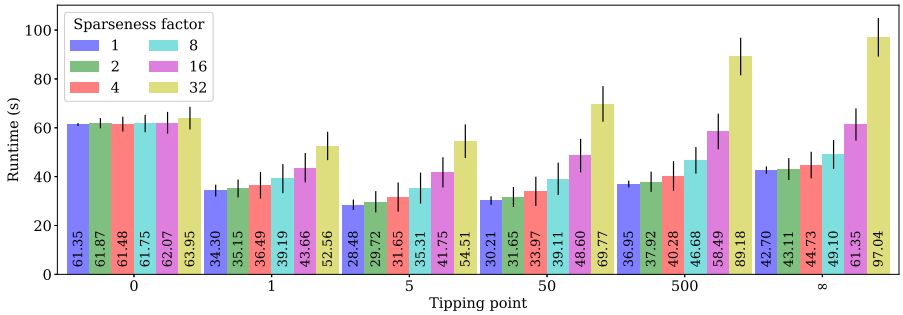

Fig. 4. Runtime for mapping 100 000 Illumina reads (150 bp) to both strands of the human reference genome as a function of the tipping point t and sparseness factor s.

denotes the sparseness factor, i.e., SA[i] is stored if and only if SA[i] $\mod s = 0$. It is well-known that a suffix at an arbitrary index i can then be inferred in $O(s)$ time [20]. Thus, the sparseness factor s controls the space-time tradeoff. As each in-text verification requires a lookup operation in the SA, a larger sparseness factor s will diminish the gains in the runtime of in-text verification.

Figure 4 shows the runtime for different sparseness factors s and tipping points t. The results for $s = 1$ (dense SA) are identical to those of Fig. 3 (left). For all values of t, the runtime increases with the sparseness factor s, as lookup operations in the SA become more expensive. For $t = 0$, the increase in runtime from $s = 1$ to $s = 32$ is limited to only 4.2% whereas for $t = \infty$, the runtime more than doubles.

Therefore, especially for larger values of the sparseness factor s, the tipping point t should not be set to (too) high values for good performance. In our experience, up to $s = 16$, a choice of $t \approx 5$ appears appropriate. For sparseness factors of $s = 32$ and larger, a tipping point of $t = 1$ or $t = 2$ showed the best performance.

6.3 Comparison to state-of-the-art tools

In earlier work [21], we presented Columba 1.0, a fast software implementation for lossless approximate pattern matching using search schemes. Columba 1.0 implements the ideas outlined in [21] such as a cache-friendly BWT representation and dynamic partitioning of search schemes.

The techniques described in this paper (bit-parallel edit distance computations, in-text verification, and the adapted search schemes) are implemented in Columba 1.1. In this section, we benchmark Columba 1.1 against state-of-the-art lossless pattern matching tools, including Columba 1.0. We use the adapted search schemes proposed in Table 1, a tipping point $t = 5$ and a SA sparseness factor $s = 1$ (dense SA).

Table 3. Runtime comparison of state-of-the-art lossless alignment tools, with the exception of BWA in 'mem' node, which is a lossy alignment algorithm.

Tool	Language	Reference	$k = 1$	$k = 2$	$k = 3$	$k = 4$
Columba 1.1[1]	C++	This paper	5.15 ± 0.44 s	8.66 ± 1.00 s	13.06 ± 1.31 s	28.48 ± 2.13 s
Columba 1.0[2]	C++	[21]	7.05 ± 0.16 s	13.10 ± 0.26 s	25.62 ± 0.33 s	67.75± 0.51 s
BWA[3]	C	[12]	14.73± 0.23 s	133.11 ± 2.39 s	1454.40 ± 24.64 s	DNC (> 3h)
Bwolo	C++	[26]	12.53± 0.55 s	25.24 ± 0.86 s	63.67 ± 1.32 s	189.78 ± 2.25 s
GEMv3[4]	C	[14]	9.0 ± 1.5 s	18.6 ± 2.4 s	38.5 ± 4.6 s	84.6 ± 4.9 s
Yara v0.9.11[5]	C++	[23]	4.49 ± 0.13 s	21.00 ± 0.34 s	81.90 ± 0.84 s	537.26 ± 7.65 s
BWA mem (lossy)	[12]		32.42 ± 0.67 s (independent of k)			

In Table 3, we compare the performance of Columba 1.1 to Columba 1.0, Bwolo [26], GEM [14], Yara [23] and BWA [12] in all-mapping mode. Note that Columba 1.0 and Bwolo do not report the CIGAR string of the alignments in their output whereas the other tools do (including Columba 1.1). For the GEM aligner, not all occurrences could be reported as the tool failed when using the `all` parameter. Therefore, GEM was configured to report at most 1000 occurrences per read.

Columba 1.1 outperforms Columba 1.0 for all values of k, even though Columba 1.0 does not compute the CIGAR string. Gains are achieved through the tighter lower bounds as specified in the adapted search schemes and bit-parallel, in-text verification. Clearly, these gains outweigh the extra computations required to generate the CIGAR string.

Both Columba 1.1 and 1.0 outperform all other lossless alignment tools for $k \geq 2$. For $k = 1$, both are slightly slower than Yara. This is likely due to the overhead imposed by the use of the bidirectional FM-index, whereas Yara relies on a unidirectional index. For $k \geq 2$, Columba 1.1 is at least twice as fast as other tools. For $k = 4$, Columba 1.1 appears roughly 3× faster than GEM, 6× faster than Bwolo, and even 18× times faster than Yara. Clearly, BWA was not designed to run in lossless mode for higher values of k.

We also compare Columba 1.1 with BWA in (lossy) mem mapping mode. In mem mode, BWA does not require a maximum number of errors k to be specified and it will typically report only a single candidate alignment position for each read. Note that the time to read the index structure from disk is included in BWA's runtime, which is not the case for Columba 1.1. Also note that BWA outputs SAM format and is able to handle paired-end reads, which is not the case for Columba 1.1. Columba 1.1 appears faster than BWA for $k = 1, 2$ and 3. For $k = 4$, the runtime of Columba 1.1 is similar to that of BWA. This indicates that the performance gap between lossless and lossy alignment tools is closing for practical bioinformatics applications such as read mapping.

[1] -e k -i 5 -ss ../search_schemes/kuch_k+1_adapted/
[2] -e k -ss ../search_schemes/kuch_k+1/
[3] aln -N -n k -i 0 -l 150 -k k
[4] -t 1 -e [k] -s [k] –alignment-model edit –mapping-mode complete -M 1000
[5] -e [k] -s [k] -y full -t 1

7 Conclusion

We introduced Columba 1.1, a tool for lossless approximate pattern matching using search schemes under the edit distance metric. Columba 1.1 implements hybrid in-index matching/in-text verification using a bit-parallel, pairwise alignment algorithm. It is demonstrated that this technique reduces runtime by more than a factor of two, compared to pure in-index matching. We provided an analysis of the effect of in-text verification for different types of reads. For reads with a limited number of occurrences, switching to in-text verification greatly reduces the runtime. In contrast, for reads with many potential occurrences, in-index matching appears the better option. We showed that the use of a sparse suffix array somewhat diminishes the performance gains of using in-text verification. Nevertheless, for all practical values of the suffix array sparseness factor, in-text verification proves beneficial. Finally, Columba 1.1 shows superior performance to state-of-the-art lossless aligners.

Acknowledgments. Luca Renders and Lore Depuydth are funded by the Research Foundation – Flanders (FWO), through a PhD Fellowship SB (1SE7822N) and a PhD Fellowship FR (1117322N), respectively.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms **2**(1), 53 – 86 (2004). https://doi.org/https://doi.org/10.1016/S1570-8667(03)00065-0
2. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology **215**(3), 403–10 (1990)
3. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Tech. rep., Digital Systems Research Center (1994)
4. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings 41st Annual Symposium on Foundations of Computer Science. pp. 390–398 (Feb 2000). https://doi.org/10.1109/SFCS.2000.892127
5. Gusfield, D.: Algorithms on strings, trees, and sequences : computer science and computational biology. Cambridge Univ. Press (Jan 2007)
6. Hyyrö, H.: A bit-vector algorithm for computing Levenshtein and Damerau edit distances. Nord. J. Comput. **10**(1), 29–39 (2003)
7. Hyyrö, H., Navarro, G.: Faster bit-parallel approximate string matching. In: Annual Symposium on Combinatorial Pattern Matching. pp. 203–224. Springer (2002)
8. Kent, W.J.: BLAT – The BLAST-like alignment tool. Genome Res **12**(4), 656–64 (2002)
9. Kianfar, K., Pockrandt, C., Torkamandi, B., Luo, H., Reinert, K.: FAMOUS: fast approximate string matching using optimum search schemes. CoRR (2017), http://arxiv.org/abs/1711.02035
10. Kucherov, G., Salikhov, K., Tsur, D.: Approximate string matching using a bidirectional index. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) Combinatorial Pattern Matching. pp. 222–231. Springer International Publishing, Cham (2014)

11. Lam, T., Li, R., Tam, A., Wong, S., Wu, E., Yiu, S.: High throughput short read alignment via bi-directional BWT. In: IEEE International Conference on Bioinformatics and Biomedicine. pp. 31 – 36 (Dec 2009). https://doi.org/10.1109/BIBM.2009.42
12. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows–Wheeler transform. Bioinformatics **25**(14), 1754–1760 (2009). https://doi.org/10.1093/bioinformatics/btp324
13. Maaß, M.G.: Linear bidirectional on-line construction of affix trees. In: Giancarlo, R., Sankoff, D. (eds.) Combinatorial Pattern Matching. pp. 320–334. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
14. Marco-Sola, S., Sammeth, M., R, G., Ribeca, P.: The gem mapper: fast, accurate and versatile alignment by filtration. Nature Methods **9**(12), 1185–1188 (2012). https://doi.org/10.1028/nmeth.2221
15. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. Journal of Computing and System Sciences **20**(1), 18–31 (1980)
16. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. In: Farach-Colton, M. (ed.) Combinatorial Pattern Matching. pp. 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
17. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (mar 2001). https://doi.org/10.1145/375360.375365, https://doi.org/10.1145/375360.375365
18. Navarro, G., Baeza-Yates, R.: A hybrid indexing method for approximate string matching. Journal of Discrete Algorithms **1**(1), 205–239 (2000)
19. Pockrandt, C., Ehrhardt, M., Reinert, K.: EPR-dictionaries: A practical and fast data structure for constant time searches in unidirectional and bidirectional FM-indices (2016)
20. Pockrandt, C.M.: Approximate String Matching: Improving Data Structures and Algorithms. Ph.D. thesis, Freien Universität Berlin (2019), http://dx.doi.org/10.17169/refubium-2185
21. Renders, L., Marchal, K., Fostier, J.: Dynamic partitioning of search patterns for approximate pattern matching using search schemes. iScience **24**(7), 102687 (2021). https://doi.org/10.1016/j.isci.2021.102687
22. Schneider, V., Graves-Lindsay, T., Howe, K., Bouk, N., Chen, H.C., Kitts, P., Murphy, T., Pruitt, K., Thibaud-Nissen, F., Albracht, D., Fulton, R., Kremitzki, M., Magrini, V., Markovic, C., McGrath, S., Steinberg, K., Auger, K., Chow, W., Collins, J., Church, D.: Evaluation of GRCh38 and de novo haploid genome assemblies demonstrates the enduring quality of the reference assembly. Genome Research **27** (2017). https://doi.org/10.1101/gr.213611.116
23. Siragusa, E.: Approximate string matching for high-throughput sequencing. Ph.D. thesis (2015)
24. Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis. Theoretical Computer Science **389**(1), 278 – 294 (2007). https://doi.org/https://doi.org/10.1016/j.tcs.2007.09.029
25. Vintsyuk, T.K.: Speech discrimination by dynamic programming. Cybernetics **4**(1), 52–57 (1968). https://doi.org/doi:10.1007/bf01074755
26. Vroland, C., Salson, M., Bini, S., Touzet, H.: Approximate search of short patterns with high error rates using the 01*0 lossless seeds. Journal of Discrete Algorithms **37**, 3–16 (2016). https://doi.org/10.1016/j.jda.2016.03.002