Search Schemes for Approximate Pattern Matching: An Overview

Lore Depuydt
Department of Information Technology, Ghent University - imec, Belgium

Jan Fostier
Department of Information Technology, Ghent University - imec, Belgium

Simon Gottlieb
FB Mathematik & Informatik, Freie Universität Berlin, Germany

Gregory Kucherov
LIGM, CNRS, Université Gustave Eiffel, Marne-la-Vallée, France

Knut Reinert
FB Mathematik & Informatik, Freie Universität Berlin, Germany

Luca Renders
Department of Information Technology, Ghent University - imec, Belgium

Abstract

We provide a brief survey of results on solving the approximate pattern matching problem using search schemes, as introduced by Kucherov et al. (2016). We demonstrate that search schemes constitute a flexible and versatile tool that enable the specification of various search strategies, including several known filtering methods. We present approaches for designing efficient search schemes and for implementing them effectively. Finally, we conclude with experimental results comparing multiple search schemes on DNA sequencing data using the Columba software by Renders et al. (2021).

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases FM-index, bidirectional index, approximate pattern matching, search scheme

Digital Object Identifier 10.4230/OASIcs.Manzini.2025.9

Supplementary Material *Software*: `https://github.com/biointec/columba`
archived at `swh:1:dir:3dc8bae72bcd433f3e2ff1cdf0016666a5b6e816`

Funding *Lore Depuydt*: Ph.D. Fellowship FR (1117322N), Research Foundation - Flanders (FWO).
Luca Renders: Ph.D. Fellowship SB (1SE7822N), Research Foundation - Flanders (FWO).

1 Introduction

Quickly locating a given string (pattern) in a long sequence is a canonical algorithmic problem with numerous practical applications. In bioinformatics, for example, an important task is to locate thousands of *reads* generated by DNA sequencing in a genomic sequence. In this example, as well as in most other practical settings, it is necessary to allow for a certain number of *errors*, i.e., differences between the pattern and its occurrence in the sequence. This leads to the problem of *approximate pattern matching* (APM) that we address in this paper. A common algorithmic formalization, which we follow in this paper, assumes that we are given an error threshold k and that we are looking for *all* occurrences of a given pattern up to k errors in a text. The error model is defined either by the Hamming distance, which considers only character substitutions, or by the edit (Levenshtein) distance, which also allows character insertions and deletions.

Pattern matching is a problem that has been extensively studied. One class of solutions, relevant for most practical applications, is *indexed pattern matching*, where the text is preprocessed into an index data structure that supports efficient pattern search. Among such data structures, the *FM-index*, also known as the BWT-index, is particularly important. It was proposed by Ferragina and Manzini in 2000 [7]. Unlike other popular data structures such as *suffix trees* [43, 28, 40] and *suffix arrays* [26], the FM-index is a succinct data structure whose size (in bits) is asymptotically the same as that of the underlying sequence. Interestingly, the FM-index is built upon the *Burrows-Wheeler transform* [5], which is rooted in word combinatorics [8] and was initially proposed for text compression [27]. However, the idea proved extremely useful for indexing: the FM-index and its variants became integral components of many bioinformatics tools, such as Bowtie [19, 18], BWA and BWA-SW [20, 21], SOAP2 [22], Masai [37], Centrifuge [12], FMAlign [23, 44], Centrifuger [38], or ProPhyle [3].

In contrast to exact pattern matching, which can be solved very efficiently, approximate pattern matching with k errors is a more challenging problem, as known algorithms are, in the worst case, exponential in k with respect to either time or space requirements [39]. Considerable effort has been made to develop practically efficient APM algorithms. One proposed heuristic is *seeding*, based on the observation that approximately matching fragments typically share exact patterns of certain size. The idea is then to use small exactly matching patterns, or their combinations, as *seeds* for potential enclosing approximate matches. Another perspective on this approach is *filtering* where seeds are viewed as indicators of potential match locations in the sequence, filtering out irrelevant regions and narrowing the search. To efficiently detect seeds, all patterns in the text that could serve as seeds are indexed in a data structure, typically a hash table. The most prominent example of such algorithms is the BLAST alignment method, which was a predominant tool in bioinformatics for many years [13]. While seeding is generally not used for exhaustive search, it can also be applied to APM in a non-trivial way through the concept of *spaced seeds* [4, 24, 6]. This approach allows for defining one or several seeds such that *any* pattern occurrence contains at least one of them. However, computing spaced seeds that ensure a lossless search is a challenging problem, even for the Hamming distance [14]. For the edit distance, this approach becomes even more difficult to implement effectively.

In this paper, we focus on a different approach that leverages efficient indexing of the text via an FM-index. Although the standard method of pattern search using an FM-index scans the pattern backward (right-to-left), extensions of the FM-index, known as the *bidirectional FM-index*, support searching in both directions [17, 35, 36, 2, 1, 30]. This feature can be advantageous for approximate pattern matching, as first demonstrated by Lam et al. [17] for restricted cases of one or two errors. The idea of combining forward and backward searches for APM was also explored in [19] and [20], although those works employed a separate index for each direction.

In general, the idea involves partitioning the pattern into $k+1$ or more parts and breaking the search into independent searches, which can potentially run in parallel. Each search begins with one of the parts, allowing either no errors or a small number of errors. This approach drastically reduces the search space compared to the naive *backtracking* method where the pattern is searched in one direction while exploring all possible errors at each step. This idea is formalized through the concept of *search schemes* which we study in this paper. Search schemes constitute a flexible and versatile formalism for specifying search strategies. They can also be used to simulate some existing filtering methods, such as the one based on the pigeonhole principle, the suffix filter [10] or the 01*0 filter [42].

The goal of this paper is to provide an overview of search schemes, relate it to other methods, and present its current state-of-the-art. We will also survey some techniques for designing efficient search schemes. Finally, we provide some experimental results demonstrating and comparing the efficiency of different search schemes.

2 Definitions

A string is denoted as $A = a_1 a_2 \dots a_{|A|}$ where $|A|$ represents its length, and a_i refers to the i-th character of the string. An infix (or substring) of A is denoted as $A_{i,j} = a_i a_{i+1} \dots a_j$. Given an error threshold k, the approximate pattern matching problem for a query pattern $P = p_1 p_2 \dots p_{|P|}$ and a text $T = t_1 t_2 \dots t_n$ is defined as finding the set of all substrings $\{T_{s_1,e_1}, T_{s_2,e_2}, \dots\}$ such that $d(T_{s_i,e_i}, P) \leq k$ for some distance measure $d(\cdot,\cdot)$. In this work, we consider either the Hamming distance d_{ham} or the edit distance d_{edit}. The Hamming distance $d_{\text{ham}}(X,Y)$ between two strings X and Y of equal length is the number of positions at which the corresponding symbols are different. The edit distance $d_{\text{edit}}(X,Y)$ between two strings X and Y is the minimum number of operations required to transform X into Y, where an operation is an insertion, deletion, or substitution of a single character.

A *bidirectional index* of a text is a data structure for pattern matching that supports incremental search of the pattern in both directions. This allows a pattern to be searched starting from any position, extending either to the left or right, and possibly alternating directions. More specifically, given an already matched substring M, the index supports extending that match to either Mc or cM, where c is a character.

Assume a query pattern P is divided into p non-overlapping parts $P = P_1 \dots P_p$. A *search* is a triplet $S = (\pi, L, U)$. Here, $\pi = (\pi[1], \dots, \pi[p])$ is a permutation of $\{1, \dots, p\}$ defining the *search order* in which these parts will be matched. To make it consistent with bidirectional indices, it is required that π fulfills the *connectivity property*: for each $i > 1$, π_i is either $\min_{j<i} \pi_j - 1$, or $\max_{j<i} \pi_j + 1$. $L = L[1] \dots L[p]$ and $U = U[1] \dots U[p]$ are non-negative integer sequences of length p that specify respectively lower and upper bounds on the cumulative number of errors when searching for consecutive parts in the order specified by π. Formally, L and U must satisfy $L[1] \leq \dots \leq L[p] \leq k$, $U[1] \leq \dots \leq U[p] \leq k$, and $L[i] \leq U[i]$ for all $i \leq p$. A *search scheme* $\mathcal{S}$ is a collection of searches $\mathcal{S} = \{S_1, S_2, \dots, S_{|\mathcal{S}|}\}$. For ease of reading, sequences π, L and U are written as strings without separators, for example search $((2,3,1), (0,0,0), (0,1,2))$ is written as $(231, 000, 012)$.

An *error configuration* $e = (e_1, \dots, e_p)$ is a distribution of at most k errors over the p parts, i.e. $\sum_{i=1}^{p} e_i \leq k$. A search $S = (\pi, L, U)$ *covers* e if $L[i] \leq \sum_{j \leq i} e_{\pi[j]} \leq U[i]$ for all i. A search scheme is called *lossless* if each possible error configuration is covered by at least one of its searches. Furthermore, if each error configuration is covered by only one search, it is called *non-redundant*. In the case of Hamming distance, a *lossless* and *non-redundant* search scheme will lead to each match being found exactly once. In contrast, in the case of edit distance, different error configurations can actually correspond to the same match.

An example of a search scheme is given in Fig. 1. The scheme contains three searches for a pattern of length 8 over a binary alphabet, with up to $k = 2$ errors under the Hamming distance. The pattern is partitioned into three parts of length $3, 3, 2$ respectively. Note that scheme S_2 requires bidirectional search: after initially searching part P_2, part P_1 is searched backward and then P_3 is searched forward. The scheme simulates the well-known pigeonhole principle for APM when the pattern is partitioned into $k+1$ parts and an exact occurrence of each part is then extended into a potential match (see Section 3). Observe that the scheme is lossless but redundant.

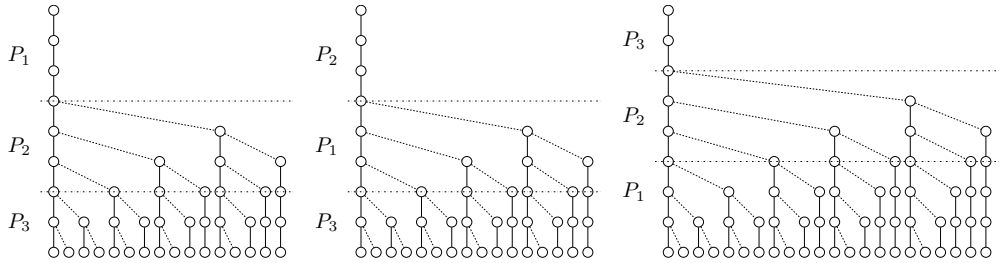

Figure 1 Pigeonhole principle expressed as a search scheme $\mathcal{S}_{\text{ph}} = \{S_1 = (123, 000, 022),\ S_2 = (213, 000, 022),\ S_3 = (321, 000, 022)\}$ with $k = 2$ errors, under the Hamming distance d_{ham}. The pattern $P = P_1P_2P_3$, defined over a binary alphabet, has length $|P| = 8$ and part lengths $|P_1| = 3$, $|P_2| = 3$, and $|P_3| = 2$. A vertical edge (solid line) represents a character match between the query pattern and the text, while a skewed edge (dotted line) denotes a mismatch. The horizontal lines delineate the parts of P.

Fig. 2 depicts another lossless search scheme for the search setting of Fig. 1. This scheme follows the algorithm of Lam et al. [17]. The third search S_{bi} covers error configuration $(1, 0, 1)$, the only one not covered by S_{fwd} and S_{bwd}. One can observe that $\mathcal{S}_{\text{Lam}}$ has a smaller search space than the search scheme from Fig. 1.

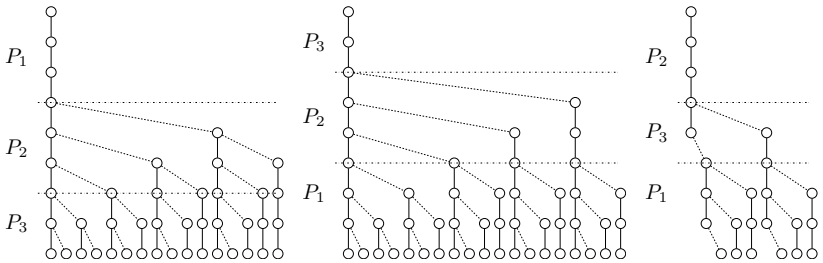

Figure 2 A search scheme described by Kucherov et al. [15] simulating the search strategy proposed by Lam et al. [17]: $\mathcal{S}_{\text{Lam}} = \{S_{\text{fwd}} = (123, 000, 022),\ S_{\text{bwd}} = (321, 000, 012),\ S_{\text{bi}} = (231, 012, 012)\}$. The parameter setting is the same as in Fig. 1.

3 Search schemes

To gain intuition about what constitutes an efficient search scheme, we first consider naive backtracking as a baseline algorithm. Naive backtracking can be expressed as a search scheme with a single search $\mathcal{S}_{\text{bt}} = \{(1, 0, k)\}$, where pattern P is not partitioned ($p = 1$) and is matched in its entirety, with a lower bound of 0 and an upper bound of k errors. Using an index such as the FM-index, candidate occurrences O of the pattern P in the text T are incrementally enumerated character by character in a depth-first manner. For simplicity, we assume that O is spelled from left to right. The number of errors between a (partial) candidate occurrence O and (a prefix of) P is tracked using a dynamic programming matrix D, where each matrix element $D(i, j)$ represents $d(O_{1,i}, P_{1,j})$, i.e., the distance between the first i characters of O and the first j characters of P. As soon as all values in row $|O|$ of D exceed the threshold k, it is impossible to further extend O to match P within k errors. In this case, the algorithm backtracks to the most recent branching point with unexplored characters and continues the search from there. If P can be completely matched (i.e., $d(O, P) \leq k$), then O constitutes an approximate occurrence of P in T and its position(s) in T can be

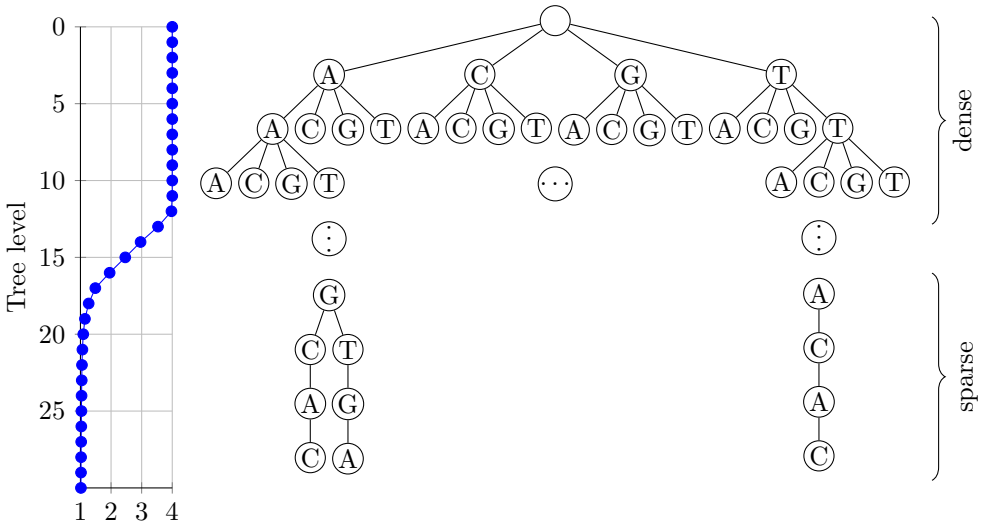

Figure 3 Left panel: average number of children per node at different levels in the substring trie of the human reference genome (GRCh38). Short sequences (length ≤ 12) can be extended by all four nucleotides. In contrast, longer sequences (length > 20) have a unique extension. Right panel: illustration of the corresponding substring trie which is dense near its root and becomes increasingly sparse at deeper levels.

reported. Note that in the case of the edit distance, maintaining a banded matrix D with $2k+1$ entries per row suffices. In the case of the Hamming distance, it is sufficient to track only of the diagonal elements of D.

The naive backtracking algorithm induces a search tree where each node corresponds to a single character extension and, therefore, to a unique substring O of the text that is enumerated. Since the FM-index enables character extensions in constant time, the runtime of naive backtracking is proportional to the number of nodes in this search tree. The size of the search tree, i.e., the total number of strings O enumerated, is governed by two factors: (i) the string O should exist in T, and (ii) either $d(O,P) \leq k$ (and then O should be reported as an occurrence), or it must still be possible to further extend O to some O', such that $d(O',P) \leq k$. Criterion (i) is controlled by the FM-index which spells out only candidate occurrences that exist in T. For (ii), the (banded) dynamic programming (DP) matrix D is used to track the distance between a (partial) candidate occurrence O and pattern P. Using naive backtracking, the search tree grows rapidly with the maximum number of allowed errors k and becomes impractically large even for modest values of k. This is because the region near the root of the search tree is densely branched. This is illustrated in Fig. 3 showing the trie of all substrings of the human reference genome. Each search enumerates a part of this trie, and a large number of short strings O are enumerated only to find out that the vast majority of them cannot be extended to an actual (approximate) occurrence of P in T. The reason that many short strings are enumerated is because short strings O are likely to be present in T and not (yet) exceed the threshold of k errors.

One way to avoid exploring the densely branched region near the root of the substring trie is to use the pigeonhole scheme $\mathcal{S}_{\text{ph}}$ illustrated in Fig. 1. If k errors are allowed, the pattern P is partitioned into $p = k+1$ parts $P = P_1 \dots P_p$. Regardless of how the k errors are distributed across the parts, at least one part must be error-free. Consequently, each

approximate occurrence O of P with at most k errors must contain at least one such error-free part of P. The key idea is to first match this error-free part of P, and then to extend this match with the remaining parts of P using backtracking, allowing up to k errors in total. To identify all occurrences, this procedure must be performed p times, each time assuming a different part of P to be error-free. In each such search, the initial exact matching bypasses the densely branched upper levels of the substring trie, and the backtracking procedure is performed only in deeper levels which are more sparsely branched (nodes have fewer possible character extensions, see Fig. 3). This pigeonhole-based approach can be formally expressed as a search scheme with $p = k + 1$ searches: $\mathcal{S}_{\text{ph}} = \{S_1, \ldots, S_{k+1}\}$, where

$$S_i = (i \ldots p\,(i-1) \ldots 1, 0 \ldots 0, 0k \ldots k) \quad \text{for } i = 1 \ldots k+1. \tag{1}$$

Formula (1) expresses that during search S_i, part P_i is matched first with no errors allowed. Next, this seed is extended with parts $P_{i+1}, \ldots, P_p$ to the right, followed with parts $P_{i-1}, \ldots P_1$ to the left, allowing up to k errors in total. Collectively, searches S_i of $\mathcal{S}_{\text{ph}}$ will identify all approximate occurrences O of P. However, the same occurrence may be reported by multiple searches. Indeed, any occurrence with more than one error-free part will be reported by multiple searches. The pigeonhole-based search has been applied in many earlier works (see e.g. [29]). Note that searches S_i with $i = 2 \ldots p-1$ require *bidirectional* search that supports extending a partial occurrence O with a single character c either to the left (cO) or to the right (Oc).

Although conceptually simple, the pigeonhole search scheme can already outperform naive backtracking by a significant margin in practical applications. Yet, each search in the pigeonhole search scheme already admits the maximum number of k errors starting from the second part that is searched. Intuitively, performance could benefit from only gradually allowing more errors when additional parts are matched. Indeed, deeper levels in the substring trie are associated with fewer branches (see Fig. 3), so admitting more errors is best postponed as much as possible.

Kärkkäinen and Na [10] proposed a *suffix filter* that enhances the pigeonhole method. The pattern is still partitioned into $p = k + 1$ parts $P = P_1 \ldots P_p$, but the search is now performed for *strongly matching* suffixes $P_i \ldots P_p$. A suffix $P_i \ldots P_p$ is said to strongly match a string $O = O_i \ldots O_p$ (suffix of a potential approximate match $O = O_1 \ldots O_p$) if for any j with $i \le j \le p$, the condition $d(P_i \ldots P_j, O_i \ldots O_j) \le j - i$ holds. It was proven in [10] that this filter is lossless, meaning that for any error configuration, there always exists a strongly matching suffix. Note that the distance function $dist$ can be either d_{ham} or d_{edit}.

It is easily seen that the suffix filter can be simulated by a search scheme that first searches forward for a strongly matching suffix and then extends it backward to the beginning of P. The idea of combining suffix filter with the search on the FM-index was exploited in [25, 41, 16] in application to DNA read alignment problems. Formally, the search scheme for suffix filter is defined by $\mathcal{S}_{\text{suf}} = \{S_1, \ldots, S_{k+1}\}$, where

$$S_i = (i \ldots p\,(i-1) \ldots 1, 0 \ldots 0, 01 \ldots (p-i)\,k \ldots k) \quad \text{for } i = 1 \ldots k+1. \tag{2}$$

Another improvement of the pigeonhole-based approach was proposed by Vroland et al. [42]. Here, pattern P is partitioned into $p = k + 2$ parts instead of $k + 1$. This guarantees that any approximate occurrence O of P contains at least two error-free parts. Among these pairs, there exists one separated by a sequence of zero or more parts each containing exactly one error [42, Lemma 2]. This (variable-length) sequence of parts is referred to as a 01*0 seed. Compared to the suffix filter, the seeding of the 01*0 filter requires searching for parts with no more than one error, which contributes to its efficiency. Unfortunately, a direct

simulation of the 01*0 filter with a search scheme requires $(k+2)(k+1)/2$ searches. However, as observed by Pockrandt [31], one can simulate a weaker version by searching for an exactly matching part P_i followed by part P_{i+1} matching with at most one error. This requires only $k+1$ searches, where the last $(k+1)$-th search can be restricted to be followed by an exactly matching part P_{k+2}. In summary, this search scheme is defined by $\mathcal{S}_{01} = \{S_1, \ldots, S_{k+1}\}$, where

$$S_i = \begin{cases} (i \ldots p\,(i-1) \ldots 1, 0 \ldots 0, 01k \ldots k) & \text{for } i = 1 \ldots k \\ (i \ldots p\,(i-1) \ldots 1, 0 \ldots 0, 00k \ldots k) & \text{for } i = k+1 \end{cases} \tag{3}$$

Compared to the pigeonhole scheme (1) which uses $k+1$ parts, $\mathcal{S}_{01}$ uses $k+2$ parts which are thus slightly shorter, making the starting exact match of P_i a slightly weaker filter. However, the requirement that the following part is matched with at most one error outweighs this weakness in many practical scenarios, as will be shown below in Section 4.

In their seminal paper, Kucherov et al. [15] provided search schemes with $k+1$ and $k+2$ parts for up to $k = 4$ errors. They differ from $\mathcal{S}_{\text{ph}}$, $\mathcal{S}_{\text{suf}}$ and $\mathcal{S}_{01}$ in several aspects. First, they often contain a larger number of searches. For example, for $k = 4$ errors and $p = k+1 = 5$ parts, $\mathcal{S}_{\text{Kuch}} = \{S_1, \ldots, S_8\}$ contains eight searches[1]:

$$\begin{array}{ll} S_1 = (12345, 00000, 02244); & S_2 = (54321, 00000, 01344); \\ S_3 = (21345, 01333, 01334); & S_4 = (12345, 01333, 01334); \\ S_5 = (43521, 00111, 01244); & S_6 = (32145, 00113, 01244); \\ S_7 = (21345, 01224, 01244); & S_8 = (12345, 00334, 00444); \end{array} \tag{4}$$

Increasing the number of searches reduces the search space per search and enables a more gradual increase in the allowed number of errors as additional parts are added. Indeed, in search scheme (4), most searches allow the maximum number of $k = 4$ errors only from the fourth part that is matched. Kucherov et al. [15] define the *critical string* of a search scheme $\mathcal{S}$ as the lexicographically maximal U-string of a search in $\mathcal{S}$. The critical U-string in $\mathcal{S}_{\text{Kuch}}$ (for $k = 4$ errors and $p = 5$ parts) is 02244 from search S_1. This is more favorable than 04444, the critical U-string for the pigeonhole-based search scheme $\mathcal{S}_{\text{ph}}$. Second, the search schemes $\mathcal{S}_{\text{Kuch}}$ were the first to exploit the lower bound L. By carefully introducing a minimum number of errors on certain parts of each search, overlap between searches is minimized, reducing the search space and reducing the number of redundantly reported occurrences. Nevertheless, the searches in $\mathcal{S}_{\text{Kuch}}$ collectively still cover all error configurations and therefore maintain their lossless character. Third, the authors of [15] observed that the search pattern P does not necessarily need to be divided into equal-length parts. In fact, the part lengths can be chosen arbitrarily, provided that the same part lengths are used across all searches. Since search S_1 is associated with the critical U-string, it will likely correspond to the largest search space (and thus the longest runtime) if P is partitioned into equally sized parts. Therefore, it may be beneficial to slightly increase the size of part P_1 while slightly decreasing the sizes of the other parts. This adjustment reduces the search space for searches that match P_1 as their first, error-free part (i.e., searches S_1, S_4, and S_8), while increasing the search space for the remaining searches. Although some searches will become slower while others speed up, the overall effect is a net performance gain. A more formal justification of the benefit of uneven partitioning is given in [15], Section 3.2. Overall, the search schemes proposed in [15] proved to be very efficient for practical bioinformatics applications [34].

[1] Note that [15] has a different definition of lower bound, therefore those were adjusted to the definition we use in this paper. Furthermore, lower bounds in S_6 and S_7 were slightly corrected compared to [15].

3.1 Design of search schemes

Designing efficient (or even optimal) search schemes is a fundamental and difficult combinatorial optimization problem, due to a large number of degrees of freedom in designing a search scheme: the number of searches $|\mathcal{S}|$, the number of parts p, the size of each part $|P_i|$ $(i = 1 \dots p)$, and the strings π_s, U_s and L_s for each search S_s $(s = 1 \dots |\mathcal{S}|)$.

Kianfar et al. [11] were the first to tackle this problem in a systematic way for the Hamming distance. They proposed a mixed integer linear program (MILP) that minimizes the expected size of the search tree, i.e., the expected number of strings O enumerated. Assuming a fixed number of $|\mathcal{S}|$ searches, a fixed number of parts p parts and fixed part lengths $|P_i|$ $(i = 1 \dots p)$, and a maximum number of allowed errors k, this objective translates into

$$\text{Minimize} \quad \sum_{s=1}^{|\mathcal{S}|} \sum_{i=1}^{p} \sum_{j=1}^{|P_{\pi_s[i]}|} \sum_{d=L_s[i-1]}^{U_s[i]} n_{l,d}, \qquad \text{where } l = j + \sum_{i'=1}^{i-1} |P_{\pi_s[i']}|. \tag{5}$$

Here, $n_{l,d}$ denotes the expected number of enumerated strings O of length l that have exactly d mismatches with respect to the corresponding part of search pattern P. The summations express that, during each search S_s, the number of errors d is constrained by $L_s[i-1] \leq d \leq U_s[i]$, and that parts of P are matched in the sequence specified by the permutation array π_s. Note that *during* the matching of part $P_{\pi_s[i]}$, the lower bound is $L_s[i-1]$, rather than $L_s[i]$, because the lower bound $L_s[i]$ applies only *after* part $P_{\pi_s[i]}$ has been fully matched. $L_s[0]$ is always 0. If the characters of the search text T and the search pattern P are assumed to be independently and uniformly drawn from the alphabet, $n_{l,d}$ can be efficiently computed for the Hamming distance (see [15, Section 3] for details). The MILP formulation imposes several constraints to ensure that the resulting search scheme $\mathcal{S}$ is valid. For example, the search scheme $\mathcal{S}$ should be lossless and the permutation array π should satisfy the connectivity property. Using the CPLEX solver on the MILP formulation, optimal triplets (π_s, L_s, U_s) are provided for each search S_s $(s = 1 \dots |\mathcal{S}|)$ as output. Note that the value of $|\mathcal{S}|$, provided as input to the MILP solver, denotes an *upper bound* on the number of searches. If a resulting, MILP-optimal search scheme has $|\mathcal{S}^*| < |\mathcal{S}|$ searches, then $|\mathcal{S}| - |\mathcal{S}^*|$ *empty* searches are generated. The solves achieves this by setting $L_s[i] > U_s[i]$ for some i.

Using their MILP formulation, Kianfar et al. [11] were able to generate search schemes for up to $k = 4$ errors for the Hamming distance. For example, for $k = 4$ and $p = k + 1 = 5$ parts, the resulting search scheme $\mathcal{S}_{\text{Kian}}$ consists of three searches:

$$\begin{aligned} S_1 &= (12345, 00004, 03344); \\ S_2 &= (23451, 00000, 22334); \\ S_3 &= (54321, 00033, 00444); \end{aligned} \tag{6}$$

Note that search S_2 already allows two errors in the first part of P that is matched. This makes $\mathcal{S}_{\text{Kian}}$ less suitable for matching under the edit distance, as the dense region near the root of the search tree becomes very large.

More recently, Renders et al. [32] applied a MILP approach to generate search schemes, albeit with a different optimization function. The authors focus on the edit distance, where computing the expected number of enumerated strings O during a search is computationally more challenging. Additionally, they argue that the expected number of enumerated strings – computed under the assumption of uniformly random T and P – is a poor approximation of the actual number observed in real-world applications, such as read mapping against the

human reference genome. This discrepancy arises because the human reference genome has a complex repeat structure, in which certain patterns occur significantly more frequently than would be expected under a uniform randomness assumption.

Therefore, rather than trying to minimize the expected number of generated strings, the following objective function is used in their MIP formulation:

$$\text{Minimize} \quad \sum_{s=1}^{|\mathcal{S}|} \sum_{i=1}^{p} (k+1)^{(p-i)} \cdot U_s[i] - \sum_{s=1}^{|\mathcal{S}|} \sum_{i=1}^{p} (p-i+1) \cdot L_s[i] + \sum_{q=1}^{Q} \sum_{s=1}^{|\mathcal{S}|} \lambda_{q,s}. \tag{7}$$

The first term of objective function (7) seeks to provide lexicographically small U-arrays. The weighting factors, $(k+1)^{(p-i)}$, are larger for smaller values of i, meaning that the number of allowed errors in U_s should increase only gradually as more parts of P are matched. The intuition is that allowing more errors should be deferred until the search tree becomes sparsely branched. The second term aims to maximize the L-arrays. In this case, the weighting factors are smaller, because the impact on runtime of the L-arrays is less pronounced compared to the U-arrays. Finally, the third component promotes minimizing the redundant reporting of occurrences by multiple searches. Here, $\lambda_{q,s}$ is a boolean variable indicating whether an error distribution q is covered by search s. Note that there are $Q = \binom{p+k}{k}$ possible ways to distribute at most k errors over p parts and that each error distribution should be covered by at least one search, i.e., $\sum_{s=1}^{|\mathcal{S}|} \lambda_{q,s} \geq 1$.

Assuming $p = k+1$ parts and a search scheme $\mathcal{S}$ with exactly $|\mathcal{S}| = p$ searches, Renders et al. [33] provided MILP-optimal search schemes for up to $k = 7$ errors. These search schemes are referred to as *MinU* search schemes due to the primary focus on providing lexicographically small U-arrays. For reference, we provide three resulting optimal search schemes $\mathcal{S}_{\text{MinU,A}}$, $\mathcal{S}_{\text{MinU,B}}$, and $\mathcal{S}_{\text{MinU,C}}$ for $k = 4$ errors and $p = k+1 = 5$ parts:

	$\mathcal{S}_{\text{MinU,A}}$	$\mathcal{S}_{\text{MinU,B}}$	$\mathcal{S}_{\text{MinU,C}}$
S_1	(12345, 00222, **02244**)	(12345, 01114, 01444)	(12345, 01114, 01444)
S_2	(23145, 00000, 01244)	(21345, 00003, 01444)	(21345, 00003, 01444)
S_3	(32145, 01111, 01244)	(34521, 01111, **02244**)	(34521, 01111, 01244)
S_4	(45321, 00003, 01444)	(43521, 00000, 01244)	(43521, 00000, 01244)
S_5	(54321, 01114, 01444)	(54321, 00222, 01244)	(54321, 00222, **02244**)

(8)

Next to these MILP-optimal search schemes for up to $k = 7$ errors, Renders et al. [32] provide search schemes for up to $k = 13$ errors that were generated using a greedy heuristic.

Note that despite this progress, the design of efficient search schemes remains an open research problem. First, due to the large number of variables involved, generating MILP-optimal search schemes becomes computationally challenging for larger values of k; even for $k = 7$, the solving the optimization problem (7) requires many hours of computing time. Second, it is difficult to formulate the goal of minimizing the search space as an optimization criterion that is both accurate and easy to evaluate.

3.2 Dynamic selection of search schemes

For a fixed long text, such as a sequence of human genome, further optimizations of the search process are possible. The expected size of the search tree, and consequently the expected workload, associated with the optimal search schemes $\mathcal{S}_{\text{MinU,A}}$, $\mathcal{S}_{\text{MinU,B}}$, and $\mathcal{S}_{\text{MinU,C}}$ is identical. In other words, across a large number of (random) patterns P, and assuming equally sized parts P_i, the three search schemes should require the same amount of work. This can be verified by comparing their L- and U-arrays.

However, the workload for *individual* patterns P may vary significantly between the optimal search schemes. Recall that the critical U-string of a search scheme is the lexicographically largest U-string. Kucherov et al. [15] showed that in a search scheme with k errors and $p = k + 1$ parts, the minimal critical U-string is given by $02244 \dots kk$ when k is even, and $013355 \dots kk$ when k is odd [15, Theorem 1]. In $\mathcal{S}_{\text{MinU,A}}$, $\mathcal{S}_{\text{MinU,B}}$, and $\mathcal{S}_{\text{MinU,C}}$, searches S_1, S_3, and S_5, respectively, have the critical U-string 02244 (indicated in boldface in (8)). As these searches already allow for two errors in the second part that is matched ($P_{\pi[2]}$), they are expected to require more work than the other searches in their respective search schemes. In other words, to minimize the workload associated with a search scheme, one should primarily focus on the search that contains the critical U-string.

These insights lead to the following heuristic for reducing workload. First, perform exact matching for parts P_i (for $i = 1, 3, 5$) individually. Next, compare their number of exact matches. If P_1 has the fewest exact matches, execute search scheme $\mathcal{S}_{\text{MinU,A}}$. Similarly, if P_3 has the fewest matches, execute search scheme $\mathcal{S}_{\text{MinU,B}}$. Otherwise, execute search scheme $\mathcal{S}_{\text{MinU,C}}$. The number of exact matches in P_i (for $i = 1, 3, 5$) is thus used as a proxy for the size of the search tree that must be explored during the search that contains the critical U-string. While this heuristic offers no strong guarantees regarding the workload of individual patterns, it performs very well *on average* across a large number of patterns. Additionally, the results from the exact matching of P_i (for $i = 1, 3, 5$) in the first step of the heuristic can be reused during the execution of the individual searches. Consequently, the heuristic incurs minimal overhead.

3.3 Dynamic partitioning of search patterns

Recall that search patterns P can be partitioned arbitrarily, as long as the same partitioning is consistently applied across all searches within a given search scheme. This flexibility allows for partitioning P in a way that balances the workload among searches.

In their original work, Kucherov et al. [15] proposed a dynamic programming algorithm to compute an optimal pattern partition for a given search scheme, under the assumption of a random pattern searched in a random text. They also provided some examples of uneven partitions outperforming even ones. They point out, however, that an optimal partition is very sensible to the choice of parameters such as the search scheme, number of parts but also the text length.

In practice, we are often interested to design an efficient search for a fixed text, such as a human genome. If all searches within a search scheme have the same expected workload –such as in the pigeonhole-based search scheme $\mathcal{S}_{\text{ph}}$– it is advantageous to partition P so that each part P_i has approximately the same number of exact occurrences in the text. However, for a specific pattern P, an equal-sized partitioning does not necessarily achieve this balance, since certain parts P_i may occur more frequently within the search text T than others. In other words, an effective partitioning strategy should be tailored to the content of P itself.

To address this, Renders et al. [34] proposed a greedy heuristic that determines the part sizes dynamically to ensure a suitably balanced distribution of exact occurrences across the parts P_i. What constitutes a "suitably balanced" distribution depends on the search scheme. For symmetric search schemes like $\mathcal{S}_{\text{ph}}$, this means ensuring a uniform distribution of the number of occurrences of P_i. In general, balance is achieved when the number of occurrences of each P_i is inversely proportional to the expected workload of the searches that begin at P_i. The heuristic by Renders et al. operates by determining part sizes while performing exact matching for each P_i, introducing minimal overhead. However, it does not provide guarantees regarding the workload of individual patterns in the context of dynamic

Table 1 The average size of the search tree (lower = better) for different search schemes when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba, allowing for at most k errors in Hamming distance. Each value represents the average size for a single read and its reverse complement. A dash ('-') indicates no search scheme is available for that value of k. dnc = did not complete.

Search scheme	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
Naive backtracking	3 946	48 630	430 831	2 950 884	dnc	dnc	dnc
Pigeonhole [17]	439	1 151	3 605	10 851	29 956	74 175	165 197
Suffix filter [10]	439	954	2 275	5 153	10 902	21 589	40 707
01 [31]	516	1 115	2 749	6 510	14 363	30 098	61 651
Kucherov $k+1$ [15]	439	902	1 715	4 260	-	-	-
Kucherov $k+2$ [15]	516	1 156	2 131	6 020	-	-	-
Kianfar [11]	439	899	6 355	52 492	-	-	-
Man_{best} [31]	-	-	-	5 549	-	-	-
MinU [33]	439	902	1 715	3 506	6 595	10 394	16 371

search scheme selection. Nevertheless, it has been shown to yield performance improvements of approximately 30% on average. Notably, dynamic partitioning of search patterns and dynamic selection of search schemes are independent techniques and can be combined for further optimization.

4 Results

4.1 Dataset

In all benchmarks, all occurrences of search patterns were identified in both strands of the human reference genome (GRCh38) with up to k errors under the Hamming or edit distance metrics. Non-ACGT characters (e.g., 'N's) in the reference sequence were replaced with randomly chosen nucleotides. We sampled 1,000,000 Illumina NovaSeq 6000 reads (151 bp) from a whole-genome sequencing dataset (accession no. SRR13586123). Given our focus on approximate pattern matching, all reads were treated as single-end, ignoring paired-end read information. All results were obtained using Columba version 2.0, an efficient tool for approximate pattern matching using search schemes. The C++ source code of Columba is available at `https://github.com/biointec/columba` under AGPL-3.0 license.

4.2 Empirical evaluation of search schemes

We exemplify the practical benefit of using different search schemes using the mentioned data set. More extensive results can also be found in [9]. Tables 1 and 2 show the average size of the search tree to identify all approximate occurrences of an Illumina read in the human reference genome for different search schemes and different values of allowed errors k under the Hamming[2] and edit distance[3], respectively. The search tree corresponds to the total number of (partial) candidate occurrence strings O that were generated during the execution of the search scheme. It evaluates the efficiency of a search scheme regardless of implementation characteristics of the software. Note that Tables 1 and 2 report the number of generated candidate occurrences O that actually exist in T.

[2] Command for Hamming distance:
`./columba -a all -e [e] -K 0 -p uniform -i 0 -nD -m hamming -v -c [path/to/search/scheme]`

[3] Command for edit distance:
`./columba -a all -e [e] -K 0 -p uniform -i 0 -nD -v -c [path/to/search/scheme]`

Table 2 The average size of the search tree (lower = better) for different search schemes when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba, allowing for at most k errors in edit distance. Each value represents the average size for a single read and its reverse complement. A dash ('-') indicates no search scheme is available for that value of k. dnc = did not complete.

Search scheme	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
Naive backtracking	7 248	160 736	2 419 153	24 728 576	dnc	dnc	dnc
Pigeonhole [17]	444	1 205	3 936	12 444	36 386	95 614	225 615
Suffix filter [10]	444	990	2 434	5 698	12 515	25 846	50 837
01 [31]	523	1 161	2 946	7 180	16 426	35 978	77 455
Kucherov $k+1$ [15]	444	937	1 821	4 631	-	-	-
Kucherov $k+2$ [15]	523	1 100	2 157	5 394	-	-	-
Kianfar [11]	444	934	10 344	166 809	-	-	-
Man_{best} [31]	-	-	-	6 233	-	-	-
MinU [33]	444	937	1 821	3 821	7 336	11 848	19 021

Table 3 The runtime (in seconds, lower = better) for different search schemes when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba, allowing for at most k errors in Hamming distance. A dash ('-') indicates no search scheme is available for that value of k. The runtimes for Naive Backtracking are extrapolated from a dataset with only 10 000 reads. dnc = did not complete in 3 hours.

Search scheme	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
Naive Backtracking	328	3 613	32 354	235 843	dnc	dnc	dnc
Pigeonhole	102	223	585	1 681	4 611	dnc	dnc
Suffix filter	103	193	380	771	1575	3 051	5 581
01	105	208	443	947	2051	4 340	dnc
Kucherov $k+1$	101	187	315	638	-	-	-
Kucherov $k+2$	108	209	344	723	-	-	-
Kianfar	99	187	763	4 889	-	-	-
Man_{best}	-	-	-	914	-	-	-
MinU	98	190	311	577	1 000	1 573	2 334

Tables 1 and 2 include the search space for naive backtracking as a baseline scenario. Results for backtracking with $k > 4$ are omitted due to computational impracticality. The search scheme based on the pigeonhole principle already outperforms backtracking by a large margin. This emphasizes the necessity to avoid having to explore the densely branched region near the root of the substring trie. Exploiting the 01 principle further reduces this search space. However, custom-designed search schemes with stringent lower and upper bounds generally perform best, with a more significant difference for larger values of k. For example, for $k = 7$, the MinU search schemes have a 10× and 4× smaller search space than the pigeonhole principle-based and 01-based search schemes, respectively. Note that Tables 1 and 2 include Man_{best}, a search scheme designed by Pockrandt [31] for $k = 4$.

Tables 3 and 4 report the runtime to align 1 million Illumina reads to the human reference genome on a single core 64-core Intel® Xeon® E5-2698 v3 CPU, running at a base clock frequency of 2.30 GHz CPU. These values correlate to a large degree with Tables 1 and 2.

4.3 Effect of dynamic selection of search schemes

To ensure an unbiased comparison between search schemes, certain optimizations in Columba, such as dynamic selection of search schemes and dynamic partitioning of search patterns, were disabled in the previous comparison, as they may affect different search schemes in different ways. For example, exploiting dynamic selection of search schemes requires having equivalent alternatives to choose from.

Table 4 The runtime (in seconds, lower = better) for different search schemes when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba, allowing for at most k errors in edit distance. A dash ('-') indicates no search scheme is available for that value of k. The runtimes for Naive Backtracking are extrapolated from a dataset with only 10 000 reads. dnc = did not complete in 3 hours.

Search scheme	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
Naive Backtracking	684	13 125	200 593	dnc	dnc	dnc	dnc
Pigeonhole	105	256	725	2 207	6 657	dnc	dnc
Suffix filter	102	231	479	1 044	2 251	4 741	9 687
01	112	243	550	1 271	2 920	6 648	dnc
Kucherov $k+1$	105	219	405	877	-	-	-
Kucherov $k+2$	110	239	442	975	-	-	-
Kianfar	113	215	1243	-	-	-	-
Man_{best}	-	-	-	1 210	-	-	-
MinU	104	217	392	762	1 398	2 281	3 618

Table 5 The average size of the search space for different dynamic techniques when aligning 1 million Illumina reads (151 bp) and their reverse complements to the human reference genome using Columba with the minU search schemes, allowing for at most k errors in edit distance. A dash ('-') indicates that dynamic selection is not useful for this value of k.

Heuristic	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
None	444	937	1 821	3 821	7 336	11 848	19 021
Dynamic Partitioning	340	742	1 589	3 791	7 174	11 679	18 333
Dynamic Selection	-	845	-	3 721	-	11 379	-
Both	-	708	-	3 458	-	10 779	-

Table 5 presents the average size of the search space for the same alignment task using the MinU search schemes and the edit distance metric under four scenarios: i) no heuristics applied; ii) using only dynamic partitioning of search patterns; iii) using only dynamic selection of search schemes (applicable only to even k); iv) both heuristics enabled. Note that the results in scenario i) are identical to those in Table 2 for the MinU search scheme.

The use of dynamic partitioning of search patterns adaptively determines the size of the search patterns so that their number of exact occurrences is balanced. Across all values of k, and for a large number of search patterns, this technique reduces the search space by 4% to 23% for this dataset. The largest relative reduction occurs for smaller values of k, where fewer but larger parts provide greater flexibility in adapting their sizes.

Enabling dynamic selection of search schemes allows Columba to choose between optimal search schemes based on the number of exact matches in their individual parts. This technique applies only to even values of k, where the critical U-string is $02244\ldots kk$. Again, this technique reduces the search space by a similar margin.

In case both techniques are combined, dynamic partitioning of the search pattern is applied first. Once a partitioning for the particular read has been fixed, the expected best-performing search scheme is selected. Combining both heuristics yields a reduction in search space between 9% ($k=6$) and 24% ($k=2$). Both heuristics are easy to implement and impose almost no overhead. These results demonstrate their effectiveness for practical applications.

References

1 Djamal Belazzougui and Fabio Cunial. Fully-functional bidirectional Burrows-Wheeler indexes and infinite-order de bruijn graphs. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual*

Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy, volume 128 of *LIPIcs*, pages 10:1–10:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CPM.2019.10.

2 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2013. doi:10.1007/978-3-642-40450-4_12.

3 Karel Břinda, Leandro Lima, Simone Pignotti, Natalia Quinones-Olvera, Kamil Salikhov, Rayan Chikhi, Gregory Kucherov, Zamin Iqbal, and Michael Baym. Efficient and robust search of microbial genomes via phylogenetic compression. *bioRxiv:2023.04.15.536996*, 2024. to appear in Nature Methods. doi:10.1101/2023.04.15.536996.

4 Stefan Burkhardt and Juha Kärkkäinen. Better filtering with gapped q-grams. In Amihood Amir and Gad M. Landau, editors, *Combinatorial Pattern Matching, 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1-4, 2001 Proceedings*, volume 2089 of *Lecture Notes in Computer Science*, pages 73–85. Springer, 2001. doi:10.1007/3-540-48194-X_6.

5 M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation, 1994.

6 Martin Farach-Colton, Gad M. Landau, S. Cenk Sahinalp, and Dekel Tsur. Optimal spaced seeds for faster approximate string matching. *Journal of Computer and System Sciences*, 73(7):1035–1044, 2007. Bioinformatics III. doi:10.1016/j.jcss.2007.03.007.

7 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Symp. on Foundations of Computer Science (FOCS), Redondo Beach, California, USA*, pages 390–398. IEEE Computer Society, February 2000. doi:10.1109/SFCS.2000.892127.

8 Ira M Gessel and Christophe Reutenauer. Counting permutations with given cycle structure and descent set. *Journal of Combinatorial Theory, Series A*, 64(2):189–215, 1993. doi:10.1016/0097-3165(93)90095-P.

9 Simon Gene Gottlieb and Knut Reinert. Search schemes for approximate string matching. *NAR Genomics and Bioinformatics*, 7(1):lqaf025, March 2025. doi:10.1093/nargab/lqaf025.

10 Juha Kärkkäinen and Joong Chae Na. Faster filters for approximate string matching. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007. doi:10.1137/1.9781611972870.8.

11 Kiavash Kianfar, Christopher Pockrandt, Bahman Torkamandi, Haochen Luo, and Reinert Knut. Optimum search schemes for approximate string matching using bidirectional FM-index. In *RECOMB-Seq*, pages 1–13, March 2018. available at arxiv:1711.02035.

12 Daehwan Kim, Li Song, Florian P Breitwieser, and Steven L Salzberg. Centrifuge: rapid and sensitive classification of metagenomic sequences. *Genome research*, 26(12):1721–1729, 2016.

13 Gregory Kucherov. Evolution of biosequence search algorithms: a brief survey. *Bioinformatics*, 35(19):3547–3552, October 2019. doi:10.1093/bioinformatics/btz272.

14 Gregory Kucherov, Laurent Noé, and Mikhail A. Roytberg. Multiseed lossless filtration. *IEEE ACM Trans. Comput. Biol. Bioinform.*, 2(1):51–61, 2005. doi:10.1109/TCBB.2005.12.

15 Gregory Kucherov, Kamil Salikhov, and Dekel Tsur. Approximate string matching using a bidirectional index. *Theoretical Computer Science*, 638:145–158, 2016. preliminary version in CPM'2014. doi:10.1016/J.TCS.2015.10.043.

16 Gregory Kucherov and Dekel Tsur. Improved filters for the approximate suffix-prefix overlap problem. In Edleno Silva de Moura and Maxime Crochemore, editors, *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science*, pages 139–148. Springer, 2014. doi:10.1007/978-3-319-11918-2_14.

17 Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *2009 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2009, Washington, DC, USA, November 1-4, 2009, Proceedings*, pages 31–36. IEEE Computer Society, 2009. doi:10.1109/BIBM.2009.42.

18 Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.

19 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

20 Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. doi:10.1093/bioinformatics/btp324.

21 Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows–Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. doi:10.1093/BIOINFORMATICS/BTP698.

22 Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, June 2009. doi:10.1093/bioinformatics/btp336.

23 Huan Liu, Quan Zou, and Yun Xu. A novel fast multiple nucleotide sequence alignment method based on FM-index. *Briefings in Bioinformatics*, 23(1):bbab519, 2022. doi:10.1093/BIB/BBAB519.

24 Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, March 2002. doi:10.1093/bioinformatics/18.3.440.

25 Veli Mäkinen, Niko Välimäki, Antti Laaksonen, and Riku Katainen. Unified view of backward backtracking in short read mapping. In Tapio Elomaa, Heikki Mannila, and Pekka Orponen, editors, *Algorithms and Applications: Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday*, volume 6060 of *Lecture Notes in Computer Science*, pages 182–195. Springer, 2010. doi:10.1007/978-3-642-12476-1_13.

26 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

27 Giovanni Manzini. Invited lecture: The Burrows-Wheeler transform: Theory and practice. In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*, volume 1672 of *Lecture Notes in Computer Science*, pages 34–47. Springer, 1999. doi:10.1007/3-540-48340-3_4.

28 Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

29 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001. doi:10.1145/375360.375365.

30 Christopher Pockrandt, Marcel Ehrhardt, and Knut Reinert. EPR-dictionaries: A practical and fast data structure for constant time searches in unidirectional and bidirectional FM indices. In S. Cenk Sahinalp, editor, *Research in Computational Molecular Biology*, pages 190–206, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-56970-3_12.

31 Christopher Maximilian Pockrandt. *Approximate string matching: improving data structures and algorithms*. PhD thesis, Freie Universität Berlin, 2019.

32 Luca Renders, Lore Depuydt, Sven Rahmann, and Jan Fostier. Automated design of efficient search schemes for lossless approximate pattern matching. In Jian Ma, editor, *Research in Computational Molecular Biology - 28th Annual International Conference, RECOMB 2024, Cambridge, MA, USA, April 29 - May 2, 2024, Proceedings*, volume 14758 of *Lecture Notes in Computer Science*, pages 164–184. Springer, 2024. doi:10.1007/978-1-0716-3989-4_11.

33 Luca Renders, Lore Depuydt, Sven Rahmann, and Jan Fostier. Lossless approximate pattern matching: Automated design of efficient search schemes. *Journal of Computational Biology*, 31(10):975–989, 2024. doi:10.1089/cmb.2024.0664.

34 Luca Renders, Kathleen Marchal, and Jan Fostier. Dynamic partitioning of search patterns for approximate pattern matching using search schemes. *iScience*, 24(7):102687, 2021. doi:10.1016/j.isci.2021.102687.

35 Luís M. S. Russo, Gonzalo Navarro, Arlindo L. Oliveira, and Pedro Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009. doi:10.3390/A2031105.

36 Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012. doi:10.1016/J.IC.2011.03.007.

37 Enrico Siragusa, David Weese, and Knut Reinert. Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic acids research*, 41(7):e78–e78, 2013.

38 Li Song and Ben Langmead. Centrifuger: lossless compression of microbial genomes for efficient and accurate metagenomic sequence classification. *Genome Biology*, 25(1):106, 2024.

39 Wing-Kin Sung. Indexed approximate string matching. In *Encyclopedia of Algorithms*, pages 964–968. Springer, 2016. doi:10.1007/978-1-4939-2864-4_188.

40 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

41 Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. *Inf. Comput.*, 213:49–58, 2012. doi:10.1016/J.IC.2012.02.002.

42 Christophe Vroland, Mikael Salson, Sébastien Bini, and Hélène Touzet. Approximate search of short patterns with high error rates using the 01*0 lossless seeds. *Journal of Discrete Algorithms*, 37:3–16, 2016. doi:10.1016/j.jda.2016.03.002.

43 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.

44 Pinglu Zhang, Huan Liu, Yanming Wei, Yixiao Zhai, Qinzhong Tian, and Quan Zou. FM-Align2: a novel fast multiple nucleotide sequence alignment method for ultralong datasets. *Bioinformatics*, 40(1):btae014, 2024. doi:10.1093/BIOINFORMATICS/BTAE014.