Approximate String Matching using a Bidirectional Index

Gregory Kucherov* Kamil Salikhov*† Dekel Tsur‡

Abstract

We study strategies of approximate pattern matching that exploit bidirectional text indexes, extending and generalizing ideas of [9]. We introduce a formalism, called search schemes, to specify search strategies of this type, then develop a probabilistic measure for the efficiency of a search scheme, prove several combinatorial results on efficient search schemes, and finally, provide experimental computations supporting the superiority of our strategies.

1 Introduction

Approximate string matching has numerous practical applications and has long been a subject of extensive studies by algorithmic researchers [18]. If errors are allowed in a match between a pattern string and a text string, most of fundamental ideas behind exact string search algorithms become inapplicable.

The problem of approximate string matching comes in different variants. In this paper, we are concerned with the *indexed* variant, when a static text is available for pre-processing and storing in a data structure (index), before any matching query is made. The challenge of indexed approximate matching is to construct a small-size index supporting quick search for approximate pattern occurrences, within a worst-case time weakly dependent on the text length. From the theoretical perspective, even the case of one allowed error turned out to be highly nontrivial and gave rise to a series of works (see [10] and references therein). In the case of k errors, existing solutions generally have time or space complexity that is exponential in k, see [22] for a survey.

The quest for efficient approximate string matching algorithms has been boosted by a new generation of DNA sequencing technologies, capable to produce huge quantities of short DNA sequences, called *reads*. Then, an important task is to *map* those reads to a given reference genomic sequence, which requires very fast and accurate approximate string matching algorithms. This motivation resulted in a very large number of read mapping algorithms and associated software programs, we refer to [13] for a survey.

*CNRS/LIGM, Université Paris-Est Marne-la-Vallée, France

†Mechanics and Mathematics Department, Lomonosov Moscow State University, Russia

‡Department of Computer Science, Ben-Gurion University of the Negev, Israel

Broadly speaking, read mapping algorithms follow one of two main approaches, or sometimes a combination of those. The *filtration* approach proceeds in two steps: it first identifies (with or without using a full-text index) locations of the text where the pattern can *potentially* occur, and then verifies these locations for actual matches. Different filtration schemes have been proposed [5, 7, 8, 17]. Filtration algorithms usually don't offer interesting worst-case time and space bounds but are often efficient on average and are widely used in practice. Another approach, usually called *backtracking*, extends exact matching algorithms to the approximate case by some enumeration of possible errors and by simulating exact search of all possible variants of the pattern. It is this approach that we follow in the present work. Backtracking and filtration techniques can be combined in a *hybrid* approach [15].

Some approximate matching algorithms use standard text indexes, such as suffix tree or suffix arrays. However, for large datasets occurring in modern applications, these indexes are known to take too much memory. Suffix arrays and suffix trees typically require at least 4 or 10 *bytes* per character respectively. The last years saw the development of *succinct* or *compressed full-text indexes* that occupy virtually as much memory as the sequence itself and yet provide very powerful functionalities [16]. For example, the FM-index [6], based on the Burrows-Wheeler Transform [3], may occupy 2–4 *bits* of memory per character for DNA texts. FM-index has now been used in many practical bioinformatics software programs, e.g. [11, 12, 21]. Even if succinct indexes are primarily designed for exact string search, using them for approximate matching naturally became an attractive opportunity. This direction has been taken in several papers, see [19], as well as in practical implementations [21].

Interestingly, succinct indexes can provide even more functionalities than classical ones. In particular, succinct indexes can be made *bidirectional*, i.e. can perform pattern search in both directions [2, 9, 19, 20]. Lam et al. [9] showed how a bidirectional FM-index can be used to efficiently search for strings up to a small number (one or two) errors. The idea is to partition the pattern into $k + 1$ equal parts, where k is the number of errors, and then perform multiple searches on the FM-index, where each search assumes a different distribution of mismatches among the pattern parts. It has been shown experimentally in [9] that this improvement leads to a faster search compared to the best existing read alignment software. Related algorithmic ideas appear also in [19].

In this paper, we extend the search strategy of [9] in two main directions. We consider the case of arbitrary k and propose to partition the pattern into more than $k + 1$ parts that can be of *unequal* size. To demonstrate the benefit of both ideas, we first introduce a general formal framework for this kind of algorithm, called *search scheme*, that allows us to easily specify them and to reason about them (Section 2). Then, in Section 3 we perform a probabilistic analysis that provides us with a quantitative measure of performance of a search scheme, and give an efficient algorithm for obtaining the optimal pattern partition for a given scheme. Furthermore, we prove several combinatorial results on the design of efficient search schemes (Section 4). Finally, Section 5 contains comparative analytical estimations, based on our probabilistic analysis, that demonstrate the superiority of our search strategies for many practical parameter ranges. We further report on large-scale experiments on genomic data supporting this analysis.

2 Bidirectional search

In the framework of text indexing, pattern search is usually done by scanning the pattern online and recomputing *index points* referring to the occurrences of the scanned part of the pattern. With classical text indexes, such as suffix trees or suffix arrays, the pattern is scanned left-to-right (*forward search*). However, some compact indexes such as FM-index provide a search algorithm that scans the pattern right-to-left (*backward search*).

Consider now approximate string matching. For ease of presentation, we present most of our ideas for the case of Hamming distance (recall that the Hamming distance between two strings A and B of equal lengths is the number of indices i for which $A[i] \neq B[i]$), although our algorithms extend to the edit distance as well. Section 3.1.2 below will specifically deal with the edit distance.

Assume that k letter mismatches are allowed between a pattern P and a substring of length $|P|$ of a text T. Both forward and backward search can be extended to approximate search in a straightforward way, by exploring all possible mismatches along the search, as long as their number does not exceed k and the current pattern still occurs in the text. For the forward search, for example, the algorithm enumerates all substrings of T with Hamming distance at most k to a *prefix* of P. Starting with the empty string, the enumeration is done by extending the current string with the corresponding letter of P, and with all other letters provided that the number of accumulated mismatches has not yet reached k. For each extension, its positions in T are computed using the index. Note that the set of enumerated strings is closed under prefixes and therefore can be represented by the nodes of a trie. Similar to forward search, *backward search* enumerates all substrings of T with Hamming distance at most k to a *suffix* of P.

Clearly, backward and forward search are symmetric and, once we have an implementation of one, the other can be implemented similarly by constructing the index for the reversed text. However, combining both forward and backward search within one algorithm results in a more efficient search. To illustrate this, consider the case $k = 1$. Partition P into two equal length parts $P = P_1P_2$. The idea is to perform two complementary searches: forward search for occurrences of P with a mismatch in P_2 and backward search for occurrences with a mismatch in P_1. In both searches, branching is performed only after $|P|/2$ characters are matched. Then, the number of strings enumerated by the two searches is much less than the number of strings enumerated by a single standard forward search, even though two searches are performed instead of one.

A *bidirectional index* of a text allows one to extend the current string A both left and right, that is, compute the positions of either cA or Ac from the positions of A. Note that a bidirectional index allows forward and backward searches to alternate, which will be crucial for our purposes. Lam et al. [9] showed how the FM-index can be made bidirectional. Other succinct bidirectional indexes were given in [2, 19, 20]. Using a bidirectional index, such as FM-index, forward and backward searches can be performed in time linear in the number of enumerated strings. Therefore, our main goal is to organize the search so that the number of enumerated strings is minimized.

Lam et al. [9] gave a new search algorithm, called *bidirectional search*, that utilizes the bidirectional property of the index. Consider the case $k = 2$, studied in [9]. In this case, the pattern is partitioned into three equal length parts, $P = P_1P_2P_3$. There are now 6 cases to consider according to the placement of mismatches within the parts: 011 (i.e. one mismatch in P_2 and one mismatch in P_3), 101, 110, 002, 020, and 200. The algorithm of Lam et al. [9] performs three searches (illustrated in Figure 1):

1. A forward search that allows no mismatches when processing characters of P_1, and 0 to 2 accumulated mismatches when processing characters of P_2 and P_3. This search handles the cases 011, 002, and 020 above.

2. A backward search that allows no mismatches when processing characters of P_3, 0 to 1 accumulated mismatches when processing characters of P_2, and 0 to 2 accumulated mismatches when processing characters of P_1. This search handles the cases 110 and 200 above.

3. The remaining case is 101. This case is handled using a *bidirectional search*. It starts with a forward search on string $P' = P_2P_3$ that allows no mismatches when processing characters of P_2, and 0 to 1 accumulated mismatches when processing the characters of P_3. For each string A of length $|P'|$ enumerated by the forward search whose Hamming distance from P' is exactly 1, a backward search for P_1 is performed by extending A to the left, allowing one additional mismatch. In other words, the search allows 1 to 2 accumulated mismatches when processing the characters of P_1.

We now give a formal definition for the above. Suppose that the pattern P is partitioned into p parts. A *search* is a triplet of strings $S = (\pi, L, U)$ where π is a permutation string of length p over $\{1, \ldots, p\}$, and L, U are strings of length p over $\{0, \ldots, k\}$. The string π indicates the order in which the parts of P are processed, and thus it must satisfy the following *connectivity property*: For every $i > 1$, $\pi(i)$ is either $(\min_{j<i} \pi(j)) - 1$ or $(\max_{j<i} \pi(j)) + 1$. The strings U and L give upper and lower bounds on the number of mismatches: When the j-th part is processed, the number of accumulated mismatches between the active strings and the corresponding substring of P must be between $L[j]$ and $U[j]$. Formally, for a string A over integers, the *weight* of A is $\sum_i A[i]$. A search $S = (\pi, L, U)$ *covers* a string A if $L[i+1] \leq \sum_{j=1}^{i} A[j] \leq U[i]$ for all i (assuming $L[p+1] = 0$). A *k-mismatch search scheme* $\mathcal{S}$ is a collection of searches such that for every string A of weight k, there is a search in $\mathcal{S}$ that covers A. For example, the 2-mismatch scheme of Lam et al. consists of searches $S_f = (123, 000, 022)$, $S_b = (321, 000, 012)$, and $S_{bd} = (231, 001, 012)$. We denote this scheme by $\mathcal{S}_{\text{Lam}}$.

In this work, we introduce two types of improvements over the search scheme of Lam et al.

Uneven partition. In $\mathcal{S}_{\text{Lam}}$, search S_f enumerates more strings than the other two searches, as it allows 2 mismatches on the second processed part of P, while the other two searches allow only one mismatch. If we increase the length of P_1 in the partition of P, the number of strings enumerated by S_f will decrease, while the

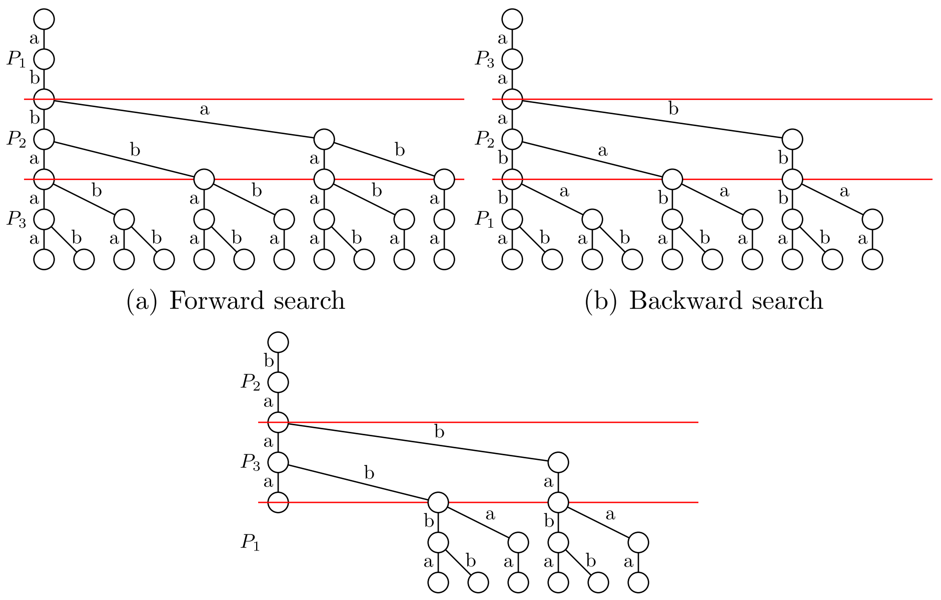
(a) Forward search (b) Backward search (c) Bidirectional search

Figure 1: The tries representing the searches of Lam et al. for binary alphabet $\{a, b\}$, search string $P =$ abbaaa, and number of errors 2. Each trie represents one search and assumes that all the enumerated substrings exist in the text T. In an actual search on a specific T, each trie contains of a subset of the nodes, depending on whether the strings of the nodes in the trie appear in T. A vertical edge represents a match, and a diagonal edge represents a mismatch.

number of strings enumerated by the two other searches will increase. We show that for some typical parameters of the problem, the decrease in the former number is larger than the increase of the latter number, leading to a more efficient search.

More parts. Another improvement can be achieved using partitions with $k+2$ or more parts, rather than $k+1$ parts. We explain in Section 3.2 why such partitions can reduce the number of enumerated strings.

3 Analysis of search schemes

In this section we show how to estimate the performance of a given search scheme $\mathcal{S}$. Using this technique, we first explain why an uneven partition can lead to a better performance, and then present a dynamic programming algorithm for designing an optimal partition of a pattern.

3.1 Estimating the efficiency of a search scheme

To measure the efficiency of a search scheme, we estimate the number of strings enumerated by all the searches of $\mathcal{S}$. We assume that performing single steps of forward, backward, or bidirectional searches takes the same amount of time. It is fairly straightforward to extend the method of this section to the case when these times are not equal. Note that the bidirectional index of Lam et al. [9] reportedly spends slightly more time (order of 10%) on forward search than on backward search.

For the analysis, we assume that characters of T and P are randomly drawn uniformly and independently from the alphabet. We note that it is possible to extend the method of this section to a non-uniform distribution. For more complex distributions, a Monte Carlo simulation can be applied which, however, requires much more time than the method of this section.

3.1.1 Hamming distance

Our approach to the analysis is as follows. Consider a fixed search S, and the trie representing this search (see Figure 1). The search enumerates the largest number of strings when the text contains all strings of length m as substrings. In this case, every string that occurs in the trie is enumerated. For other texts, the set of enumerated strings is a subset of the set of strings that occurs in trie. The expected number of strings enumerated by S on random T and P is equal to the sum over all nodes v of the trie of the probability that the corresponding string appears in T. We will first show that this probability depends only on the depth of v (Lemmas 1 and 2 below). Then, we will show how to count the number of nodes in each level of the trie.

Let $prob_{n,l,\sigma}$ denote the probability that a random string of length l is a substring of a random string of length n, where the characters of both strings are randomly chosen uniformly and independently from an alphabet of size σ. The following lemma gives an approximation for $prob_{n,l,\sigma}$ with a bound on the approximation error.

Lemma 1. $|prob_{n,l,\sigma} - (1 - e^{-n/\sigma^l})| \leq \begin{cases} 4nl/\sigma^{2l} & \text{if } l \geq \log_\sigma n \\ 4l/\sigma^l & \text{otherwise} \end{cases}.$

Proof. Let A and B be random strings of length l and n, respectively. Let E_i be the event that A appears in B at position i. The event E_i is independent of the events $\{E_j : j \in \{1, 2, \ldots, n-l+1\} \setminus F_i\}$, where $F_i = \{i-l+1, i-l+2, \ldots, i+l-1\}$. By the Chen-Stein method [1, 4],

$$\left| prob_{n,l,\sigma} - (1 - e^{-n/\sigma^l}) \right| \leq \frac{1 - e^{-\lambda}}{\lambda} \sum_{i=1}^{n-l+1} \sum_{j \in F_i} (\Pr[E_i] \Pr[E_j] + \Pr[E_i \cap E_j]),$$

where $\lambda = n/\sigma^l$. Clearly, $\Pr[E_i] = \Pr[E_j] = 1/\sigma^l$. It is also easy to verify that $\Pr[E_i \cap E_j] = 1/\sigma^{2l}$. Therefore, $|prob_{n,l,\sigma} - (1 - e^{-n/\sigma^l})| \leq ((1 - e^{-\lambda})/\lambda) \cdot 4nl/\sigma^{2l}$. The lemma follows since $(1 - e^{-\lambda})/\lambda \leq \min(1, 1/\lambda)$ for all λ. ■

The bound in Lemma 1 on the error of the approximation of $prob_{n,l,\sigma}$ is large if l is small, say $l < \frac{1}{2} \log_\sigma n$. In this case, we can get a better bound by observing that $prob_{n,l,\sigma} \geq prob_{n,l_0,\sigma}$, where $l_0 = \frac{3}{4} \log_\sigma n$. Since $prob_{n,l_0,\sigma} \geq 1 - e^{-n/\sigma^{l_0}} - 4l_0/\sigma^{l_0}$, we obtain that $|prob_{n,l,\sigma} - (1 - e^{-n/\sigma^l})| \leq \max(e^{-n/\sigma^l}, e^{-n/\sigma^{l_0}} + 4l_0/\sigma^{l_0})$.

Let $strings(S, X, \sigma, n)$ denote the expected number of strings enumerated when performing a search $S = (\pi, L, U)$ on a random text of length n and random pattern of length m, where X is a partition of the pattern and σ is the alphabet size (note that m is not a parameter for *strings* since the value of m is implied from X). For a search scheme $\mathcal{S}$, $\mathit{strings}(\mathcal{S}, X, \sigma, n) = \sum_{S \in \mathcal{S}} \mathit{strings}(S, X, \sigma, n)$.

Fix S, X, σ, and n. Let $\mathcal{A}_l$ be the set of enumerated strings of length l when performing search S on a random pattern of length m, partitioned by X, and a text $\hat{T}$ containing all strings of length at most m as substrings. Let $A_{l,i}$ be the i-th element of $\mathcal{A}_l$ (an order on $\mathcal{A}_l$ will be defined in the proof of the next lemma). Let $\mathit{nodes}_l = |\mathcal{A}_l|$, namely, the number of nodes at depth l in the trie that represents the search S. Let P^* be the string containing the characters of P according to the order they are read by the search. In other words, $P^*[l]$ is the character such that every node at depth $l-1$ of the trie has an edge to a child with label $P^*[l]$.

Lemma 2. *For every l and i, the string $A_{l,i}$ is a random string with uniform distribution.*

Proof. Assume that the alphabet is $\Sigma = \{0, \ldots, \sigma - 1\}$. Consider the trie that represents the search S. We define an order on the children of each node of the trie as follows: Let v be a node in the trie with depth $l-1$. The label on the edge between v and its leftmost child is $P^*[l]$. If v has more than one child, the labels on the edges to the rest of the children of v, from left to right, are $(P^*[l] + 1) \bmod \sigma, \ldots, (P^*[l] + \sigma - 1) \bmod \sigma$. We now order the set $\mathcal{A}_l$ according to the nodes of depth l in the trie. Namely, let $v_1, \ldots, v_{\mathit{nodes}_l}$ be the nodes of depth l in the trie, from left to right. Then, $A_{l,i}$ is the string that corresponds to v_i. We have that $A_{l,i}[j] = (P^*[j] + c_{i,j} - 1) \bmod \sigma$ for $j = 1, \ldots, l$, where $c_{i,j}$ is the rank of the node of depth j on the path from the root to v_i among its siblings. Now, since each letter of P is randomly chosen uniformly and independently from the alphabet, it follows that each letter of $A_{l,i}$ has uniform distribution and the letters of $A_{l,i}$ are independent. ■

By the linearity of the expectation,

$$\mathit{strings}(S, X, \sigma, n) = \sum_{l \geq 1} \sum_{i=1}^{\mathit{nodes}_l} \Pr_{T \in \Sigma^n}[A_{l,i} \text{ is a substring of } T].$$

By Lemma 2 and Lemma 1,

$$\mathit{strings}(S, X, \sigma, n) = \sum_{l=1}^{m} \mathit{nodes}_l \cdot \mathit{prob}_{n,l,\sigma} \approx \sum_{l=1}^{m} \mathit{nodes}_l (1 - e^{-n/\sigma^l}). \tag{1}$$

We note that the bounds on the approximation errors of $\mathit{prob}_{n,l,\sigma}$ are small, therefore even when these bounds are multiplied by nodes_l and summed over all l, the resulting bound on the error is small.

In order to compute the values of nodes_l, we give some definitions. Let $\mathit{nodes}_{l,d}$ be the number of strings in $\mathcal{A}_l$ of length l with Hamming distance d to the prefix of P^* of length l. For example, consider search $S_{bd} = (231, 001, 012)$ and partition of a pattern of length 6 into 3 parts of length 2, as shown in Figure 1(c). Then, $P^* = $ baaaba, $\mathit{nodes}_{5,0} = 0$, $\mathit{nodes}_{5,1} = 2$ (strings baabb and babab), and $\mathit{nodes}_{5,2} = 2$ (strings baaba and babaa).

Let π_X be a string obtained from π by replacing each character $\pi(i)$ of π by a run of $\pi(i)$ of length $X[\pi(i)]$, where $X[j]$ is the length of the j-th part in the partition X. Similarly, L_X is a string obtained from L by replacing each character $L[i]$ by a run of $L[i]$ of length $X[\pi(i)]$, and U_X is defined analogously. In other words, values $L_X[i], U_X[i]$ give lower and upper bounds on the number of allowed mismatches for an enumerated string of length i. For example, for S_{bd} and the partition X defined above, $\pi_X = 223311$, $L_X = 000011$, and $U_X = 001122$.

Values nodes_l are given by the following recurrence.

$$\mathit{nodes}_l = \sum_{d=L_X[l]}^{U_X[l]} \mathit{nodes}_{l,d} \tag{2}$$

$$\mathit{nodes}_{l,d} = \begin{cases} \mathit{nodes}_{l-1,d} + (\sigma-1)\cdot \mathit{nodes}_{l-1,d-1} & \text{if } l \geq 1 \text{ and } L_X[l] \leq d \leq U_X[l] \\ 1 & \text{if } l = 0 \text{ and } d = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

For a specific search, a closed formula can be given for nodes_l. If a search scheme $\mathcal{S}$ contains two or more searches with the same π-strings, these searches can be merged in order to eliminate the enumeration of the same string twice or more. It is straightforward to modify the computation of $\mathit{strings}(\mathcal{S}, X, \sigma, n)$ to account for this optimization.

Consider equation (1). The value of the term $1 - e^{-n/\sigma^l}$ is very close to 1 for $l \leq \log_\sigma n - O(1)$. When $l \geq \log_\sigma n$, the value of this term decreases exponentially. Note that nodes_l increases exponentially, but the base of the exponent of nodes_l is $\sigma - 1$ whereas the base of $1 - e^{-n/\sigma^l}$ is $1/\sigma$. We can then approximate $\mathit{strings}(S, X, \sigma, n)$ with function $\mathit{strings}'(S, X, \sigma, n)$ defined by

$$\mathit{strings}'(S, X, \sigma, n) = \sum_{l=1}^{\lceil \log_\sigma n \rceil + c_\sigma} \mathit{nodes}_l \cdot (1 - e^{-n/\sigma^l}), \tag{4}$$

where c_σ is a constant chosen so that $((\sigma-1)/\sigma)^{c_\sigma}$ is sufficiently small.

From the above formulas we have that the time complexities for computing $\mathit{strings}(\mathcal{S}, X, \sigma, n)$ and $\mathit{strings}'(\mathcal{S}, X, \sigma, n)$ are $O(|\mathcal{S}|km)$ and $O(|\mathcal{S}|k \log_\sigma n)$, respectively.

3.1.2 Edit distance

We now show how to estimate the efficiency of a search scheme for the edit distance.

We define $\mathit{strings}_{\text{edit}}$ analogously to $\mathit{strings}$ in the previous section, except that edit distance errors are allowed. Fix a search $S = (\pi, L, U)$ and a partition X. We assume without loss of generality that π is the identity permutation. Similarly to the Hamming distance case, define $\mathcal{A}_l$ to be the set of enumerated strings of length l when performing the search S on a random pattern of length m, partitioned by X, and a text $\hat{T}$ containing all the strings of length at most $m + k$ as substrings. Unlike the case of Hamming distance, here the strings of $\mathcal{A}_l$ are not distributed uniformly.

Thus, we do not have the equality $\mathit{strings}_{\mathrm{edit}}(S, X, \sigma, n) = \sum_{l=1}^{m} \mathit{nodes}_l \cdot \mathit{prob}_{n,l,\sigma}$. We will use $\sum_{l=1}^{m} \mathit{nodes}_l \cdot \mathit{prob}_{n,l,\sigma}$ as an approximation for $\mathit{strings}_{\mathrm{edit}}(S, X, \sigma, n)$, but we do not have an estimation on the error of this approximation. Note that in the Hamming distance case, the sizes of the sets $\mathcal{A}_l$ are the same for every choice of the pattern, whereas this is not true for edit distance. We therefore define $\mathit{nodes}_l(P)$ to be the number of enumerated strings of length l when performing the search S on a pattern P of length m, partitioned by X, and a text $\hat{T}$. We also define nodes_l to be the expectation of $\mathit{nodes}_l(P)$, where P is chosen randomly.

We next show how to compute values nodes_l. We begin by giving an algorithm for computing $\mathit{nodes}_l(P)$ for some fixed P. Build a non-deterministic automaton $\mathcal{A}_P$ that recognizes the set of strings that are within edit distance at most k to P, and the locations of the errors satisfy the requirements of the search [7,14] (see Figure 2 for an example). For a state q and a string B, denote by $\hat{\delta}_P(q, B)$ the set of all states q' for which there is a path in $\mathcal{A}_P$ from q to q' such that the concatenation of the labels on the path is equal to B. For a set of states Q and a string B, $\hat{\delta}_P(Q, B) = \cup_{q \in Q} \hat{\delta}_P(q, B)$. Clearly, $\mathit{nodes}_l(P)$ is equal to the number of strings B of length l for which $\hat{\delta}_P(q_0, B) \neq \emptyset$, where q_0 is the initial state. Let $\mathit{nodes}_{l,Q}(P)$ be the number of strings B of length l for which $\hat{\delta}_P(q_0, B) = Q$. The values of $\mathit{nodes}_{l,Q}(P)$ can be computed using dynamic programming and the following recurrence.

$$\mathit{nodes}_{l,Q}(P) = \sum_{c \in \Sigma} \sum_{Q' : \hat{\delta}_P(Q', c) = Q} \mathit{nodes}_{l-1,Q'}(P).$$

The values $\mathit{nodes}_{l,Q}(P)$ gives the values of $\mathit{nodes}_l(P)$, since by definition,

$$\mathit{nodes}_l(P) = \sum_{Q} \mathit{nodes}_{l,Q}(P),$$

where the summation is done over all non-empty sets of states Q.

Note that for a string B of length l, set $\hat{\delta}_P(q_0, B)$ is a subset of a set of $(k+1)^2$ states that depends on l. This set, denoted Q_l, includes the $l+1$-th state in the first row of the automaton, states $l, l+1, l+2$ on the second row, states $l-1, l, \ldots, l+3$ on the third row, and so on (see Figure 2). The size of Q_l is $1 + 3 + 5 + \cdots + (2k+1) = (k+1)^2$. Therefore, the number of sets Q for which $\mathit{nodes}_{l,Q}(P) > 0$ is at most $2^{(k+1)^2}$. If $(k+1)^2$ is small enough, a state can be encoded in one machine word, and the computation of $\hat{\delta}_P(Q', c)$ can be done in constant time using precomputed tables. Thus, the time for computing all values of $\mathit{nodes}_{l,Q}(P)$ is $O(2^{k^2} \sigma m)$.

Now consider the problem of computing the values of nodes_l. Observe that for $Q \subseteq \mathrm{Q}_l$, the value of $\hat{\delta}_P(Q, c)$ depends on the characters of $P[l-k+1..l+k+1]$, and does not depend on the rest of the characters of P. Our algorithm is based on this observation. For an integer l, a set $Q \subseteq \mathrm{Q}_l$, and a string P' of length $2k+1$, define

$$\mathit{nodes}_{l,Q,P'} = \sum_{P : P[l-k+1..l+k+1] = P'} \mathit{nodes}_{l,Q}(P).$$

Then,

$$\mathit{nodes}_{l,Q,P'} = \sum_{c' \in \Sigma} \sum_{c \in \Sigma} \sum_{Q' : \hat{\delta}_{P_c}(Q', c) = Q} \mathit{nodes}_{l-1,Q',P'_c},$$

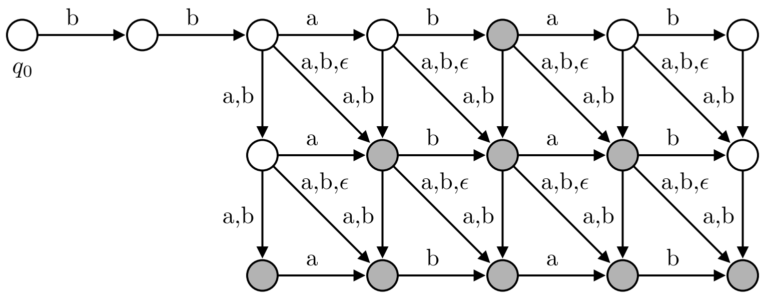

Figure 2: Non-deterministic automaton corresponding to the search $S = (12, 00, 02)$ and pattern $P = \text{bbabab}$ over the alphabet $\Sigma = \{\text{a}, \text{b}\}$. A path from the initial state q_0 to the state in the i-th row and j-column of the automaton correspond to a string with edit distance $i - 1$ to $P[1..j-1]$. The nodes of the set Q_4 are marked by gray.

where $P'_c = c'P'[1..2k]$, and P_c is a string satisfying $P_c[(l-1)-k+1..(l-1)+k+1] = P'_c$ (the rest of the characters of P_c can be chosen arbitrarily).

From the above, the time complexity for computing $\mathit{strings}_{\text{edit}}(S, X, \sigma, n)$ is $O(|\mathcal{S}|2^{k^2}\sigma^{2k+3}m)$. Therefore, our approach is practical only for small values of k.

3.2 Uneven partitions

In Section 2, we provided an informal explanation why partitioning the pattern into unequal parts may be beneficial. We now provide a formal justification for this. To this end, we replace (4) by an even simpler estimator of $\mathit{strings}(S, X, \sigma, n)$:

$$\mathit{strings}''(S, X, \sigma, n) = \sum_{l=1}^{\lceil \log_\sigma n \rceil} \mathit{nodes}_l. \tag{5}$$

As an example, consider scheme $\mathcal{S}_{\text{Lam}}$. Denote by x_1, x_2, x_3 the lengths of the parts in a partition X of P into 3 parts. It is straightforward to give closed formulas for $\mathit{strings}''(S, X, \sigma, n)$ for each search of $\mathcal{S}_{\text{Lam}}$. For example,

$$\mathit{strings}''(S_f, X, \sigma, n) = \begin{cases} N & \text{if } N \le x_1 \\ c_1(N - x_1)^3 + c_2(N - x_1)^2 + c_3(N - x_1) + N & \text{otherwise} \end{cases}$$

where $N = \lceil \log_\sigma n \rceil$, $c_1 = (\sigma-1)^2/6$, $c_2 = (\sigma-1)/2$, and $c_3 = -(\sigma-1)^2/6 + (\sigma-1)/2$. Similar formulas can be given for S_b and S_{bd}. If x_1, x_2, and x_3 are close to $m/3$ and $N < m/3$ then $\mathit{strings}''(\mathcal{S}_{\text{Lam}}, X, \sigma, n, m) = 3N$ and an equal sized partition is optimal in this case. However, if $m/3 < N < 2m/3$, then

$$\begin{aligned} \mathit{strings}''(\mathcal{S}_{\text{Lam}}, X, \sigma, n) &= c_1(N - x_1)^3 + c_2(N - x_1)^2 + c_3(N - x_1) \\ &\quad + c'_1(N - x_3)^2 + c'_2(N - x_3) + c''_1(N - x_2)^2 + c''_2(N - x_2) + 3N. \end{aligned}$$

It is now clear why the equal sized partition is not optimal in this case. The degree of $N - x_1$ in the above polynomial is 3, while the degrees of $N - x_2$ and $N - x_3$ are 2. Thus, if $x_1 = x_2 = x_3 = m/3$, decreasing x_2 and x_3 by, say 1, while increasing x_1 by 2 reduces the value of the polynomial.

3.3 Computing an optimal partition

In this Section, we show how to find an optimal partition for a given search scheme $\mathcal{S}$ and a given number of parts p. An optimal partition can be naively found by enumerating all $\binom{m-1}{p-1}$ possible partitions, and for each partition X, computing $\mathit{strings}'(\mathcal{S}, X, \sigma, n)$. We now describe a more efficient dynamic programming algorithm.

We define an optimal partition to be a partition that maximizes $\mathit{strings}(\mathcal{S}, X, \sigma, n)$. Let $N = \lceil \log_\sigma n \rceil + c_\sigma$. If $m \geq pN$, then any partition in which all parts are of size at least N is an optimal partition. Therefore, assume for the rest of this section that $m < pN$. We say that a partition X is *bounded* if the sizes of the parts of X are at most N. If X is not bounded, we can transform it into a bounded partition by decreasing the sizes of parts which are larger than N and increasing the sizes of parts which are smaller that N. This transformation can only decrease the value of $\mathit{strings}(\mathcal{S}, X, \sigma, n)$. Therefore, there exists an optimal partition which is bounded. Throughout this section we will consider only bounded partitions. For brevity, we will use the term partition instead of bounded partition.

Our algorithm takes advantage of the fact that the value of $\mathit{strings}'(\mathcal{S}, X, \sigma, n)$ does not depend on the entire partition X, but only on the partition of a substring of P of length N induced by X. More precisely, consider a fixed $S = (\pi, L, U) \in \mathcal{S}$. By definition, $\mathit{strings}'(S, X, \sigma, n)$ depends on the values $\mathit{nodes}_1, \ldots, \mathit{nodes}_N$ (the number of nodes in levels $1, \ldots, N$ in the trie that correspond to the search S). From Section 3.1, these values depend on the strings L and U which are fixed, and on the string $\pi_X[1..N]$. The latter string depends on $\pi[1..i_{X,\pi}]$, where $i_{X,\pi}$ is the minimum index such that $\sum_{j=1}^{i_{X,\pi}} X[\pi(j)] \geq N$ and on the values $X[\pi(1)], \ldots, X[\pi(i_{X,\pi})]$.

The algorithm works by going over the prefixes of P in increasing length order. For each prefix P', it computes a set of partitions of P' such that at least one partition in this set can be extended to an optimal partition of P. In order to reduce the time complexity, the algorithm needs to identify partitions of P' that cannot be extended into an optimal partition of P. Consider the following example. Suppose that $m = 13$, $p = 5$, $N = 4$ and $\mathcal{S} = \{S_1, S_2, S_3\}$, where the π-strings of S_1, S_2, S_3 are $\pi^1 = 12345$, $\pi^2 = 32451$, and $\pi^3 = 43215$, respectively. Consider a prefix $P' = P[1..8]$ of P, and let Y_1, Y_2 be two partitions of P', where the parts in Y_1 are of sizes 3,3,2, and the parts in Y_1 are of sizes 4,2,2. Note that Y_1 and Y_2 have the same number of parts, and they induce the same partition on $P[8 - N + 1..8] = P[5..8]$. We claim that one of these two partitions is always at least as good as the other for every extension of both partitions to a partition of P. To see this, let Z denote a partition of $P[9..13]$ into two parts, and consider the three searches of $\mathcal{S}$.

1. For search S_1 we have that $\pi^1_{Y_1 \cup Z}[1..N] = 1112$ for every Z, and $\pi^1_{Y_2 \cup Z}[1..N] = 1111$ for every Z. It follows that the value of $\mathit{strings}'(S_1, Y_1 \cup Z, \sigma, n)$ is the same for every Z, and the value of $\mathit{strings}'(S_1, Y_2 \cup Z, \sigma, n)$ is the same for every Z. These two values can be equal or different.

2. For the search S_2 we have that $\pi^2_{Y_1 \cup Z}[1..N] = \pi^2_{Y_2 \cup Z}[1..N] = 3322$. It follows that $\mathit{strings}'(S_2, Y_1 \cup Z, \sigma, n) = \mathit{strings}'(S_2, Y_2 \cup Z, \sigma, n)$ for all Z and this common value does not depend on Z.

3. For the search S_3 we have that $\pi^3_{Y_1 \cup Z}[1..N] = \pi^3_{Y_2 \cup Z}[1..N]$ for every Z. For example, if Z is a partition of $P[9..13]$ into parts of sizes 2,2 then $\pi^3_{Y_1 \cup Z}[1..N] = \pi^3_{Y_2 \cup Z}[1..N] = 4433$. It follows that $\mathit{strings}'(S_3, Y_1 \cup Z, \sigma, n) = \mathit{strings}'(S_3, Y_2 \cup Z, \sigma, n)$ for every Z. This common value depends on Z.

We conclude that either $\mathit{strings}'(\mathcal{S}, Y_1 \cup Z, \sigma, n) < \mathit{strings}'(\mathcal{S}, Y_2 \cup Z, \sigma, n)$ for every Z, or $\mathit{strings}'(\mathcal{S}, Y_1 \cup Z, \sigma, n) \geq \mathit{strings}'(\mathcal{S}, Y_2 \cup Z, \sigma, n)$ for every Z.

We now give a formal description of the algorithm. We start with some definitions. For a partition Y of a substring $P' = P[m''..m']$ of pattern P, we define the following quantities: m_Y is the length of P', l_Y is the length of the last part of Y, p_Y is the number of parts in Y, and r_Y is the left-to-right rank of the part of Y containing $P'[m' - N + 1]$. Let $\mathit{prefix}(Y)$ be the partition of $P[m''..m' - l_Y]$ of P' that is composed from the first $p_Y - 1$ parts of Y. For the example above, $m_{Y_1} = 8$, $l_{Y_1} = 2$, $p_{Y_1} = 3$, $r_{Y_1} = 2$, and $\mathit{prefix}(Y_1)$ is a partition of $P[1..6]$ with parts sizes $3, 3$.

For a partition Y of a prefix P' of P, $\mathcal{S}(Y)$ is a set containing every search $S \in \mathcal{S}$ such that r_Y appears before $p_Y + 1$ in the π-string of S. If the length of P' is less than N we define $\mathcal{S}(Y) = \emptyset$, and if $P' = P$ we define $\mathcal{S}(Y) = \mathcal{S}$. For the example above, $\mathcal{S}(Y_1) = \{S_1, S_2\}$.

Let Y_1 be a partition of a substring $P_1 = P[i_1..j_1]$ of P, and Y_2 be a partition of a substring $P_2 = P[i_2..j_2]$. We say that Y_1 and Y_2 are *compatible* if these partitions induce the same partition on the common substring $P' = P[\max(i_1, i_2).. \min(j_1, j_2)]$. For example, the partition of $P[4..6]$ into parts of sizes $1, 2$ is compatible with the partition of $P[1..6]$ into parts of sizes $2, 2, 2$.

Lemma 3. *Let Y be a partition of a prefix of P of length at least N. Let $S \in \mathcal{S}(Y)$ be a search. The value $\mathit{strings}'(S, X, \sigma, n)$ is the same for every partition X of P whose first p_Y parts match Y.*

Proof. Let i' be the index such that $\pi(i') = p_Y + 1$. Since r_Y appears before $p_Y + 1$ in string π, from the connectivity property of π we have that (1) Every value in π that appears before $p_Y + 1$ is at most p_Y. In other words, $\pi(i) \leq p_Y$ for every $i < i'$. (2) $r_Y, \ldots, p_Y$ appear before $p_Y + 1$ in π. By the definition of r_Y, $\sum_{j=r_Y}^{p_Y} X[j] \geq N$. Therefore, $i_{X,\pi} < i'$ and $\pi(1), \ldots, \pi(i_{X,\pi}) \leq p_Y$. Thus, string $\pi[1..i_{X,\pi}]$ and values $X[\pi(1)], \ldots, X[\pi(i_{X,\pi})]$ are the same for every partition X that satisfies the requirement of the lemma. ■

For a partition Y of a prefix of P of length at least N, define $v(Y)$ to be $\sum_{S \in \mathcal{S}(Y)} \mathit{strings}'(S, X, \sigma, n)$, where X is an arbitrary partition of P whose first p_Y parts match Y (the choice of X does not matter due to Lemma 3). For a partition Y of a prefix of P of length less than N, $v(Y) = 0$. Define

$$\Delta(Y) = v(Y) - v(\mathit{prefix}(Y)) = \sum_{S \in \mathcal{S}(Y) \setminus \mathcal{S}(\mathit{prefix}(Y))} \mathit{strings}'(S, X, \sigma, n).$$

Lemma 4. *Let Z be a partition of a substring $P[m''..m']$ such that $p_Z \geq 2$ and $m_{\mathit{prefix}(Z)} = \min(N, m' - l_Y)$. Let $p' \geq p_Z$ be an integer. The value of $\Delta(Y)$ is the same for every partition Y of $P[1..m']$ with p' parts that is compatible with Z.*

Proof. We assume $N < m' - l_Y$ (the case $N \geq m' - l_Y$ is similar). Since $m_{\mathit{prefix}(Z)} = \min(N, m' - l_Y)$, the set $\mathcal{S}(Y) \setminus \mathcal{S}(\mathit{prefix}(Y))$ is the same for every partition Y of $P[1..m']$ with p' parts that is compatible with Z. For a search $S = (\pi, L, U)$ in this set, r_Y appears before $p_Y + 1$ in π, and p_Y appears before $r_{\mathit{prefix}(Y)}$. Let $i = i_{X,\pi}$, where X is an arbitrary partition of P whose first p_Y parts are the parts of Y. We obtain that $r_{\mathit{prefix}(Y)} \leq \pi(1), \ldots, \pi(i) \leq p_Y$, and the lemma follows. ■

For Z, p' that satisfy the requirements of Lemma 4, let $\Delta(Z, p')$ denote the value of $\Delta(Y)$, where Y is an arbitrary partition of $P[1..m']$ with p' parts that is compatible with Z.

For $m' \leq m$, $p' \leq p$, and a partition Z of $P[\max(m' - N + 1, 1)..m']$ with at most p' parts, let $v(m', p', Z)$ be the minimum value of $v(Y)$, where Y is a partition of $P[1..m']$ into p' parts that is compatible with Z.

Lemma 5. *For $m' \leq m$, $2 \leq p' \leq p$, and a partition Z of $P[\max(m' - N + 1, 1)..m']$ with at most p' parts,*

$$v(m', p', Z) = \min_{Z'} \left(v(m' - l_{Z'}, p' - 1, \mathit{prefix}(Z')) + \Delta(Z', p') \right)$$

where the minimum is taken over all partitions Z' of a substring $P[m''..m']$ of P that satisfy the following: (1) Z' is compatible with Z, (2) $2 \leq p_{Z'} \leq p'$, (3) $m_{\mathit{prefix}(Z')} = \min(N, m' - l_{Z'})$, (4) $p_Z = p'$ if $m'' = 1$.

An algorithm for computing the optimal partition follows from Lemma 5. The time complexity of the algorithm is $O\big((|\mathcal{S}|kN + m) \sum_{j=1}^{\min(p-1,N)} (p-j)\binom{N-1}{j-1}\big)$, where $|\mathcal{S}|kN \sum_{j=1}^{\min(p-1,N)} (p-j)\binom{N-1}{j-1}$ is time for computing Δ values, and $O\big(m \sum_{j=1}^{\min(p-1,N)} (p-j)\binom{N-1}{j-1}\big)$ is time for computing v values.

4 Properties of optimal search schemes

Designing an efficient search scheme for a given set of parameters consists of (1) choosing a number of parts, (2) choosing searches, (3) choosing a partition of the pattern. While it is possible to enumerate all possible choices, and evaluate the efficiency of the resulting scheme using Section 3.1, this is generally infeasible due to a large number of possibilities. It is therefore desirable to have a combinatorial characterization of optimal search schemes.

The *critical string* of a search scheme $\mathcal{S}$ is the lexicographically maximal U-string of a search in $\mathcal{S}$. A search of $\mathcal{S}$ is *critical* if its U-string is equal to the critical string of $\mathcal{S}$. For example, the critical string of $\mathcal{S}_{\text{Lam}}$ is 022, and S_f is the critical search. For typical parameters, critical searches of a search scheme constitute the bottleneck. Consider a search scheme $\mathcal{S}$, and assume that the L-strings of all searches contain only zeros. Assume further that the pattern is partitioned into equal-size parts. Let ℓ be the maximum index such that for every search $S \in \mathcal{S}$ and every $i \leq \ell$, $U[i]$ of S is no larger than the number in position i in the critical string of $\mathcal{S}$. From Section 3, the number of strings enumerated by a search $S \in \mathcal{S}$ depends mostly on the prefix of the U-string of S of length $\lceil \lceil \log_\sigma n \rceil / (m/p) \rceil$. Thus, if $\lceil \lceil \log_\sigma n \rceil / (m/p) \rceil \leq \ell$, a critical search enumerates an equal or greater number of strings than a non-critical search.

We now consider the problem of designing a search scheme whose critical string is minimal. Let $\alpha(k,p)$ denote the lexicographically minimal critical string of a k-mismatch search scheme that partitions the pattern into p parts. The next theorems give the values of $\alpha(k,k+2)$ and $\alpha(k,k+1)$. We need the following definition. A string over the alphabet of integers is called *simple* if it contains a substring of the form 01^j0 for $j \geq 0$.

Lemma 6. *(i) Every string A of weight k and length at least $k+2$ is simple.*

(ii) If A is a non-simple string of weight k and length $k{+}1$ then $A[1] \leq 1$, $A[k{+}1] \leq 1$, and $A[i] \leq 2$ for all $2 \leq i \leq k$. Moreover, there are no two consecutive 2's in A.

Proof. *(i)* The proof is by induction on k. It is easy to verify that the lemma holds for $k = 0$. Suppose we proved the lemma for $k' < k$. Let A be a string of weight k and length $p \geq k+2$. If $A[1] \geq 1$ then by the induction hypothesis $A[2..p]$ is simple, and therefore A is simple. Suppose that $A[1] = 0$. Let $i > 1$ be the minimum index such that $A[i] \neq 1$ (i must exist due to the assumption that $p \geq k+2$). If $A[i] = 0$ then we are done. Otherwise, we can use the induction hypothesis on $A[i+1..p]$ and obtain that A is simple.

(ii) Let A be a non-simple string of weight k and length $k+1$. If $A[1] \geq 2$ then $A' = A[2..k+1]$ has weight $k - A[1] \leq k-2$ and length k, and thus by *(i)* we obtain that A' is simple, contradicting the assumption that A is non-simple. Similarly, $A[k+1]$ cannot be greater than 1. For $2 \leq i \leq k$, if $A[i] \geq 3$ then either $A[1..i-1]$ or $A[i+1..k+1]$ satisfies the condition of *(i)*. Similarly, if $A[i] = A[i+1] = 2$ then either $A[1..i-1]$ or $A[i+2..k+1]$ satisfies the condition of *(i)*. ■

We use the following notation. For two integers i and j, $[i,j]$ denotes the string $i(i+1)(i+2)\cdots j$ if $i \leq j$, and the empty string if $i > j$. Moreover, $\overline{[i,j]}$ denotes the string $i(i-1)(i-2)\cdots j$ if $i \geq j$, and the empty string if $i < j$.

Theorem 7. *$\alpha(k,k+1) = 013355\cdots kk$ for every odd k, and $\alpha(k,k+1) = 02244\cdots kk$ for every even k.*

Proof. We first give an upper bound on $\alpha(k,k+1)$ for odd k. We build a search scheme as follows. The scheme contains searches $S_{k,i,j} = ([i,k{+}2]\overline{[i-1,1]}, 0\cdots 0, [0,j]jk\cdots k)$ for all i and j, which cover all simple strings of weight k and length $k+1$. In order to cover the non-simple strings, the scheme contains the following searches.

1. $S^1_{k,i,j} = ([i,k+1]\overline{[i-1,1]}, 0\cdots 0, 013355\cdots jj(j+1)k\cdots k)$ for every odd $3 \leq j \leq k$ (for $j = k$, the U-string is $013355\cdots kk$).

2. $S^2_{k,i,j} = (\overline{[i,1]}[i+1,k+1], 0\cdots 0, 013355\cdots jj(j+1)k\cdots k)$ for every odd $3 \leq j \leq k$ (for $j = k$, the U-string is $013355\cdots kk$).

Let A be a non-simple string of weight k and length $k+1$. By Lemma 6, $A = X0A_1 0A_2 0\cdots 0A_d 0Y$ where each of X and Y is either string 1 or empty string, and each A_i is either 2, 12, 21, or 121. A string A_i is called a *block* of type 1, 2, or 3 if A_i is equal to 12, 21, or 121, respectively. Let $B_1, \ldots, B_{d'}$ be the blocks of type 1 and type 2, from left to right.

We consider several cases. The first case is when X and Y are empty strings, and B_1 is of type 1. Since the weight of A is odd, it follows that d' is odd. If A has no other blocks, A is covered by search $S^1_{k,i,k}$, where $i+1$ is the index in A in which B_1 starts. Otherwise, if B_2 is of type 1, then A is covered by search $S^1_{k,i,j}$, where $i+1$ is the index in A in which B_1 starts, and $i+j+1$ is the index in which the first block to the right of B_1 starts (this block is either B_2, or a block of type 3). Now suppose that B_2 is of type 2. If B_3 is of type 2, then A is covered by search $S^2_{k,i,j}$, where $i-1$ is the index in A in which B_3 ends, and $i-j-1$ is the index in which the first block to the left of B_3 ends. By repeating these arguments, we obtain that A is covered unless the types of $B_1, \ldots, B_{d'}$ alternate between type 1 and type 2. However, since d' is odd, $B_{d'}$ is of type 1, and in this case A is covered by $S^1_{k,i,j}$, where $i+1$ is the index in A in which B_1 starts, and $k-j$ is the index in which the first block to the left of B_1 ends.

Now, if X is empty string and $Y = 1$, define a string $A' = A20$. By the above, A' is covered by some search $S^{j'}_{k+2,i,j}$. Then, A is covered by either $S^{j'}_{k,i,j}$ or $S^{j'}_{k,i,j-2}$. The same argument holds for the case when $X = 1$. The proof for the case when B_1 is of type 2 is analogous and thus omitted.

The lower bound on $\alpha(k, k+1)$ for odd k is obtained by considering the string $A = 012020\cdots20$. The U-string of a search that covers A must be at least $013355\cdots kk$.

We next give an upper bound on $\alpha(k, k+1)$ for even k. We define k-mismatch search schemes $\mathcal{S}_k$ recursively. For $k = 0$, $\mathcal{S}_0$ consists of a single search $S_{0,1} = (1, 0, 0)$. For $k \geq 2$, $\mathcal{S}_k$ consists of the following searches.

1. For every search $S_{k-2,i} = (\pi, 0\cdots0, U)$ in $\mathcal{S}_{k-2}$, $\mathcal{S}_k$ contains a search $S_{k,i} = (\pi \cdot k(k+1), 0\cdots0, U \cdot kk)$.
2. A search $S_{k,k} = (\overline{[k+1, 1]}, 0\cdots0, 01kk\cdots k)$.
3. A search $S_{k,k+1} = (k(k+1)\overline{[k-1, 1]}, 0\cdots0, 01kk\cdots k)$.

Note that the critical string of $\mathcal{S}_k$ is $02244\cdots kk$ corresponding to item 1 above. We now claim that all number strings of length $k+1$ and weight at most k are covered by the searches of $\mathcal{S}_k$. The proof is by induction on k. The base $k = 0$ is trivial. Suppose the claim holds for $k-2$. Let A be a number string of length $k+1$ and weight $k' \leq k$. If $A[k] + A[k+1] \leq 1$, then A is covered by either $S_{k,k}$ or $S_{k,k+1}$. Otherwise, the weight of $A' = A[1..k-1]$ is at most $k'-2$. By induction, A' is covered by some search $S_{k-2,i}$. Then search $S_{k,i}$ covers A.

To prove that $\alpha(k, k+1) \geq 02244\cdots kk$ for even k, consider the string $A = 0202\cdots020$. It is easy to verify that the U-string of a search that covers A must be at least $02244\cdots kk$. ■

Theorem 8. *$\alpha(k, k+2) = 0123\cdots(k-1)kk$ for every $k \geq 1$.*

Proof. We first give an upper bound on $\alpha(k, k+1)$. We build a k-mismatch search scheme $\mathcal{S}$ that contains searches $S_{k,i,j} = ([i, k+2]\overline{[i-1, 1]}, 0\cdots0, [0, j]jk\cdots k)$ for all i and j. Let A be a string of weight k and length $k+2$. By Lemma 6 there are indices i and j such that $A[i..i+j+1] = 01^j0$, and therefore A is covered by $S_{k,i,j}$.

The lower bound is obtained from the string $A = 011 \cdots 110$. It is easy to verify that the U-string of a search that covers A must be at least $0123 \cdots (k-1)kk$. ■

An important consequence of Theorems 7 and 8 is that for some typical cases, partitioning the pattern into $k+2$ parts brings an advantage over $k+1$ parts. For $k=2$, for example, we have $\alpha(2,3) = 022$ while $\alpha(2,4) = 0122$. Since the second element of 0122 is smaller than that of 022, a 4-part search scheme potentially enumerates less strings than a 3-part scheme. On the other hand, the average length of a part is smaller when using 4 parts, and therefore the branching occurs earlier in the searches of a 4-part scheme. The next section shows that for some parameters, $(k+2)$-part schemes outperform $(k+1)$-part schemes, while for other parameters the inverse occurs.

5 Case studies

In this Section, we provide results of several computational experiments we have performed to analyse practical applicability of our techniques.

We designed search schemes for 2, 3 and 4 errors (given in Appendix) using a greedy algorithm. The algorithm iteratively adds searches to a search scheme. At each step, the algorithm considers the uncovered string A of weight k such that the lexicographically minimal U-string that covers A is maximal. Among the searches that cover A with minimal U-string, a search that covers the maximum number of uncovered strings of weight k is chosen. The L-string of the search is chosen to be lexicographically maximal among all possible L-string that do not decrease the number of uncovered strings. For each search scheme and each choice of parameters, we computed an optimal partition.

5.1 Numerical comparison of search schemes

We first performed a comparative estimation of the efficiency of search schemes using the method of Section 3.1.1 (case of Hamming distance). More precisely, for a given search scheme $\mathcal{S}$, we estimated the number of strings $\mathit{strings}(\mathcal{S}, X, \sigma, n)$ enumerated during the search.

Results for 2 mismatches are given in Table 1 and Table 2 for 4-letter and 30-letter alphabets respectively. Table 3 contains estimations for nonuniform letter distribution. Table 4 contains estimations for 3 mismatches for 4-letter alphabet.

We first observe that our method provides an advantage only on a limited range of pattern lengths. This conforms to our analysis (see Section 3.2) that implies that our schemes can bring an improvement when $m/(k+1)$ is smaller than $\log_\sigma n$ approximately. When $m/(k+1)$ is small, Tables 1–4 suggest that using more parts of unequal size can bring a significant improvement. For big alphabets (Table 2), we observe a larger gain in efficiency, due to the fact that values nodes_l (see equation (2)) grow faster when the alphabet is large, and thus a change in the size of parts can have a bigger influence on these values. Moreover, if the probability distribution of letters in both the text and the pattern is nonuniform, then we obtain an even

Table 1: Values of $strings(\mathcal{S}, X, 4, 4^{16})$ for 2-mismatch search schemes, for different pattern lengths m. Second column corresponds to search scheme $\mathcal{S}_{\text{Lam}}$ with three equal-size parts, the other columns show results for unequal partitions and/or more parts. The partition used is shown in the second sub-column.

m	3 equal	3 unequal		4 unequal		5 unequal	
24	1197	1077	9,7,8	959	7,4,4,9	939	7,1,6,1,9
36	241	165	15,10,11	140	12,5,7,12	165	11,1,9,1,14
48	53	53	16,16,16	51	16,7,9,16	53	16,1,15,1,15

Table 2: Values of $strings(\mathcal{S}, X, 30, 30^7)$ for 2-mismatch search schemes.

m	3 equal	3 unequal		4 unequal		5 unequal	
15	846	286	6,4,5	231	5,2,3,5	286	5,1,3,1,5
18	112	111	7,6,5	81	6,2,4,6	111	6,1,4,1,6
21	24	24	7,7,7	23	7,3,4,7	24	7,1,6,1,6

Table 3: Values of $strings(\mathcal{S}, X, 4, 4^{16})$ for 2-mismatch search schemes, using a non-uniform letter distribution (one letter with probability 0.01 and the rest with probability 0.33 each).

m	3 equal	3 unequal		4 unequal		5 unequal	
24	3997	3541	10,8,6	3592	6,7,1,10	3541	6,1,7,1,9
36	946	481	16,10,10	450	11,6,6,13	481	10,1,9,1,15
48	203	157	18,15,15	137	16,7,9,16	157	15,1,14,1,17

larger gain (Table 3), since in this case, the strings enumerated during the search have a larger probability to appear in the text than for the uniform distribution.

For 3 mismatches and 4 letters (Table 4), we observe a smaller gain, and even a loss for pattern lengths 36 and 48 when shifting from 4 to 5 parts. This is explained by Theorem 7 showing the difference of critical strings between odd and even numbers of errors. Thus, for 3 mismatches and 4 parts, the critical string is 0133 while for 5 parts it is 01233. When patterns are not too small, the latter does not lead to an improvement strong enough to compensate for the decrease of part length. Note that the situation is different for even number of errors, where incrementing the number of parts from $k + 1$ to $k + 2$ leads to transforming the critical strings from $0224 \cdots$ to $0123 \cdots$.

Another interesting observation is that with 4 parts, obtained optimal partitions have equal-size parts, as the U-strings of all searches of the 4-part scheme are all the same (see Appendix).

These estimations suggest that our techniques can bring a significant gain in efficiency for some parameter ranges, however the design of a search scheme should be done carefully for each specific set of parameters.

Table 4: Values of $\mathit{strings}(\mathcal{S}, X, 4, 4^{16})$ for 3-mismatch search schemes. Best partitions obtained for 4 parts are equal.

m	4 equal/unequal		5 unequal	
24	11222	6,6,6,6	8039	4,6,5,1,8
36	416	9,9,9,9	549	6,11,5,1,13
48	185	12,12,12,12	213	11,11,11,1,14

5.2 Experiments on genomic data

To perform large-scale experiments on genomic sequences, we implemented our method using the 2BWT library provided by [9] (`http://i.cs.hku.hk/2bwt-tools/`). We then experimentally compared different search schemes, both in terms of running time and average number of enumerated substrings. Below we only report running time, as in all cases, the number of enumerated substrings produced very similar results.

The experiments were done on the sequence of human chromosome 14 (*hr14*). The sequence is $88 \cdot 10^6$ long, with nucleotide distribution 29%, 21%, 21%, 29%. Searched patterns were generated as i.i.d. sequences. For every search scheme and pattern length, we ran 10^5 pattern searches for Hamming distance and 10^4 searches for the edit distance.

5.2.1 Hamming distance

For the case of 2 mismatches, we implemented the 3-part and 4-part schemes (see Appendix), as well as their equal-size-part versions for comparison. For each pattern length, we computed an optimal partition, taking into account a non-uniform distribution of nucleotides. Results are presented in Table 5.

Using unequal parts for 3-part schemes yields a notable time decrease for patterns of length 24 and 33 (respectively, by 24% and 16%). Furthermore, we observe that using unequal part lengths for 4-part schemes is beneficial as well. For pattern lengths 24 and 33, we obtain a speed-up by 27% and 28% respectively. Overall, the experimental results are consistent with numerical estimations of Section 5.1.

For the case of 3 mismatches, we implemented 4-part and 5-part schemes from Appendix, as well as their equal part versions for comparison. Results (running time) are presented in Table 6. In accordance with estimations of Section 5.1, here we observe a clear improvement only for pattern length 15 and not for longer patterns.

5.2.2 Edit distance

In the case of edit distance, along with the search schemes for 2 and 3 errors from the previous section, we also implemented search schemes for 4 errors (see Appendix). Results are shown in Table 7 (2 errors), Table 8 (3 errors) and Table 9 (4 errors).

For 2 errors, we observe up to two-fold speed-up for pattern lengths 15, 24 and 33. For the case of 3 errors, the improvement is achieved for pattern lengths 15

Table 5: Total time (in sec) of search for 10^5 patterns in *hr14*, up to 2 mismatches. 2nd column contains time obtained on partition into three equal-size parts. The 3rd (respectively 4th and 5th) column shows the running time respectively for the 3-unequal-parts, 4-equal-parts and 4-unequal-parts searches, together with their ratio (%) to the corresponding 3-equal-parts value.

m	3 equal	3 unequal		4 equal	4 unequal	
15	24.8	25.4 (102%)	6,6,3	25.3 (102%)	25.3 (102%)	3,5,1,6
24	5.5	4.2 (76%)	10,7,7	5.2 (95%)	4.0 (73%)	7,4,4,9
33	1.73	1.45 (84%)	13,10,10	2.07 (120%)	1.25 (72%)	11,5,6,11
42	0.71	0.71 (100%)	14,14,14	1.24 (175%)	0.82 (115%)	14,6,8,14

Table 6: Total time (in sec) of search for 10^5 patterns in *hr14*, up to 3 mismatches.

m	4 equal	5 equal	5 unequal	
15	241	211 (86%)	206 (85%)	2,3,5,1,4
24	19.7	26.7 (136%)	19.6 (99%)	2,9,3,1,9
33	4.3	6.9 (160%)	4.7 (109%)	6,9,6,1,11
42	1.85	2.52 (136%)	2.05 (111%)	10,10,9,1,12
51	1.07	1.57 (147%)	1.06 (99%)	12,13,12,1,13

Table 7: Total time (in sec) of search for 10^4 patterns in *hr14*, up to 2 errors (edit distance).

m	3 equal	3 unequal		4 equal	4 unequal	
15	11.5	11.4 (99%)	6,6,3	10.9 (95%)	11.1 (97%)	3,5,1,6
24	2.1	1.3 (62%)	11,5,8	1.5 (71%)	1.0 (48%)	7,4,4,9
33	0.34	0.22 (65%)	13,10,10	0.35 (103%)	0.19 (56%)	11,5,6,11
42	0.08	0.08 (100%)	14,14,14	0.18 (225%)	0.08 (100%)	14,6,8,14

Table 8: Total time (in sec) of search for 10^4 patterns in *hr14*, up to 3 errors (edit distance).

m	4 equal	5 equal	5 unequal	
15	233	174 (75%)	168 (72%)	2,2,6,1,4
24	13.5	13.2 (98%)	10.8 (80%)	3,8,3,1,9
33	0.74	1.81 (245%)	1.07 (145%)	5,10,5,1,12
42	0.28	0.45 (161%)	0.37 (132%)	9,10,9,1,13
51	0.13	0.24 (185%)	0.14 (108%)	12,12,12,1,14

and 24 (respectively 28% and 20%). Finally, for 4 errors, we obtain a significant speed-up (18% to 30%) for pattern lengths between 15 and 51.

5.2.3 Experiments on simulated genomic reads

Experiments of Section 5.2 have been made with random patterns. In order to make experiments closer to the practical bioinformatic setting occurring in mapping genomic reads to their reference sequence, we also experimented with patterns

Table 9: Total time (in sec) of search for 10^4 patterns in *hr14*, up to 4 errors (edit distance).

m	5 equal	5 unequal		6 equal	6 unequal	
15	4212	3222 (76%)	3,1,8,1,2	4028 (96%)	3401 (81%)	2,2,1,7,1,2
24	145	133 (92%)	7,3,5,1,8	131 (90%)	113 (78%)	2,7,3,4,5,3
33	6.5	5.8 (89%)	8,7,5,8,5	6.6 (102%)	5.1 (78%)	4,8,6,3,5,7
42	1.66	1.16 (70%)	12,8,7,8,7	1.51 (91%)	1.17 (70%)	7,8,8,5,2,12
51	0.60	0.49 (82%)	13,11,9,9,9	0.74 (123%)	0.54 (90%)	9,10,9,9,1,13
60	0.28	0.24 (86%)	14,13,11,11,11	0.44 (157%)	0.28 (117%)	11,12,11,11,1,14

Table 10: Total time (in sec) of search for 10^5 reads in *hr14*, up to 4 errors. First row corresponds to read set with constant error rate 0.03. Second row corresponds to read set with error rate increasing from 0.0 to 0.03.

m	5 equal	6 equal	6 unequal	
100	247	250 (101%)	283 (115%)	20,20,20,19,1,20
100	415	367 (88%)	350 (84%)	20,20,20,19,1,20

simulating reads issued from genomic sequencers. For that, we generated realistic single-end reads of length 100 (typical length of ILLUMINA reads) from *hr14* using DWGSIM read simulator (`https://github.com/nh13/DWGSIM`). Two sets of reads were generated using two different error rate values (parameter `-e` of DWGSIM): `0.03` for the first dataset and `0.0-0.03` for the second one. This means that in the first set, error probability is uniform over the read length, while in the second set, this probability gradually increases from 0 to 0.03 towards the right end of the read. The latter simulates the real-life situation occurring with current sequencing technologies including ILLUMINA.

The results are shown in Table 10. As expected, due to a large pattern length, our schemes did not produce a speed-up for the case of constant error rate. Interestingly however, for the case of non-uniform distribution of errors, our schemes showed a clear advantage. This illustrates another possible benefit of our techniques: they are better adapted to a search for patterns with non-uniform distribution of errors, which often occurs in practical situations such as mapping genomic reads.

6 Conclusions

This paper can be seen as the first step towards an automated design of efficient search schemes for approximate string matching, based on bidirectional indexes. More research has to be done in order to allow an automated design of optimal search schemes. It would be very interesting to study an approach when a search scheme is designed simultaneously with the partition, rather than independently as it was done in our work.

We expect that search schemes similar to those studied in this paper can be applied to hybrid approaches to approximate matching (see Introduction), as well as possibly to other search strategies.

Acknowledgements. GK has been supported by the ABS2NGS grant of the French government (program *Investissement d'Avenir*) as well as by a EU Marie-Curie Intra-European Fellowship for Carrier Development. KS has been supported by the *co-tutelle* PhD fellowship of the French government. DT has been supported by ISF grant 981/11.

References

[1] A. D. Barbour, L. Holst, and S. Janson. *Poisson approximation.* Clarendon Press Oxford, 1992.

[2] D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *Proc. 21st European Symposium on Algorithms (ESA)*, pages 133–144, 2013.

[3] M. Burrow and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation, California, 1994.

[4] L. H. Y. Chen. Poisson approximation for dependent trials. *The Annals of Probability*, 3(3):534–545, 1975.

[5] M. Farach-Colton, G. M. Landau, S. C. Sahinalp, and D. Tsur. Optimal spaced seeds for faster approximate string matching. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 3580, pages 1251–1262, 2005.

[6] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Symposium on Foundation of Computer Science (FOCS)*, pages 390–398, 2000.

[7] J. Kärkkäinen and J. C. Na. Faster filters for approximate string matching. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–90, 2007.

[8] G. Kucherov, L. Noé, and M. Roytberg. Multi-seed lossless filtration. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(1):51–61, January-March 2005.

[9] T. W. Lam, R. Li, A. Tam, S. C. K. Wong, E. Wu, and S.-M. Yiu. High throughput short read alignment via bi-directional BWT. In *Proc. IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 31–36, 2009.

[10] T. W. Lam, W. K. Sung, and S. S. Wong. Improved approximate string matching using compressed suffix data structures. In *Proc. 16th International Symposium on Algorithms and Computation (ISAAC)*, pages 339–348, 2005.

[11] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

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

[13] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

[14] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistic*, 30(4):451–477, 2004.

[15] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms*, 1(1):205–239, 2000.

[16] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[17] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[18] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001.

[19] L.M.S. Russo, G. Navarro, A.L. Oliveira, and P. Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.

[20] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012.

[21] J.T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

[22] W.-K. Sung. Indexed approximate string matching. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 1–99. Springer US, 2008.

Appendix

The following search schemes were used in experiments described in Section 5.
For 2 mismatches or errors:

1. Slightly modified scheme $\mathcal{S}_{\text{Lam}}$. The searches are: $S_f = (123, 000, 022)$, $S_b = (321, 000, 012)$, and $S'_{bd} = (213, 001, 012)$. Note that the π-string of S'_{bd} is 213 and not 231 as in S_{bd}. While S_{bd} and S'_{bd} have the same efficiency for equal-size partitions, this in not the case for unequally sized parts.

2. 4-part scheme with searches $(1234, 0000, 0112)$, $(4321, 0000, 0122)$, $(2341, 0001, 0012)$, and $(1234, 0002, 0022)$.

For 3 mismatches or errors:

1. 4-part scheme with searches $(1234, 0000, 0133)$, $(2134, 0011, 0133)$, $(3421, 0000, 0133)$, and $(4321, 0011, 0133)$.

2. 5-part scheme with searches $(12345, 00000, 01233)$, $(23451, 00000, 01223)$, $(34521, 00001, 01133)$, and $(45321, 00012, 00333)$.

For 4 mismatches or errors:

1. 5-part scheme with searches $(12345, 00000, 02244)$, $(54321, 00000, 01344)$, $(21345, 00133, 01334)$, $(12345, 00133, 01334)$, $(43521, 00011, 01244)$, $(32145, 00013, 01244)$, $(21345, 00124, 01244)$ and $(12345, 00034, 00444)$.

2. 6-part scheme with searches $(123456, 00000, 012344)$, $(234561, 00000, 012344)$, $(654321, 000001, 012244)$, $(456321, 000012, 011344)$, $(345621, 000023, 011244)$, $(564321, 000133, 003344)$, $(123456, 000333, 003344)$, $(123456, 000044, 002444)$, $(342156, 000124, 002244)$ and $(564321, 000044, 001444)$.