Approximate String Matching Using a Bidirectional Index

Gregory Kucherov, Kamil Salikhov, Dekel Tsur

Approximate String Matching using a Bidirectional Index

Gregory Kucherov[1,2], Kamil Salikhov[1,3], and Dekel Tsur[2]

[1] CNRS/LIGM, Université Paris-Est Marne-la-Vallée, France
[2] Department of Computer Science, Ben-Gurion University of the Negev, Israel
[3] Mechanics and Mathematics Department, Lomonosov Moscow State University, Russia

Abstract. We study strategies of approximate pattern matching that exploit bidirectional text indexes, extending and generalizing ideas of [5]. 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 long been a subject of extensive studies by algorithmic researchers. If errors are allowed in a match between a pattern string and a text string, most of the fundamental ideas behind exact string search algorithms become inapplicable. The Approximate string matching problem comes in different variants. In this paper, we are concerned with the *indexing* variant, when a static text is available for preprocessing and storing in a data structure, before any matching query is made.

The quest for efficient approximate string matching algorithms has been boosted by a new generation of DNA sequencing technologies, able to produce huge quantities of short DNA *reads*. Those reads can then be mapped to a given genomic sequence, which requires very fast and accurate approximate string matching algorithms. Many algorithms and associated software programs have been designed for this task, we refer to [9] for a survey, and many of them rely on full-text indexes.

The classical indexing paradigm consists in building a text index in order to quickly identify pattern occurrences, preferably within a worst-case time weakly dependent on the text length. In the context of approximate matching, even the case of one error turned out to be highly nontrivial and gave rise to a series of works (see [6] and references therein). In the case of k errors, existing solutions generally have time or space complexity that are exponential in k, see [14] for a survey.

Some of the existing 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 emergence of *succinct* or *compressed full-text indexes* that occupy virtually as much memory as the sequence itself and yet provide very powerful functionalities [10]. For example, the FM-index [4], based on the Burrows-Wheeler Transform [2], 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. [7, 8, 13]. 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 [11], as well as in practical implementations [13].

Interestingly, succinct indexes can provide even more functionalities than classical ones. In particular, several succinct indexes can be made *bidirectional*, i.e. can perform pattern search in both directions [1, 5, 11, 12]. Lam et al. [5] 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. Lam et al. implemented the proposed algorithm and reported that it outperforms in speed the best existing read alignment software [5]. Related algorithmic ideas appear also in [11].

In this paper, we extend the search strategy of [5] 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, where k letter mismatches are allowed between a pattern P and a text T. In this paper, we present our ideas for the case of Hamming distance. However, they also apply to the edit distance, see Conclusions. 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. Similarly 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 an 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 S both left and right, that is, compute the positions of both cS or Sc from the positions of S. Note that a bidirectional index allows forward and backward searches to alternate, which will be crucial for our purposes. Lam et al. [5] showed how the FM-index can be made bidirectional. Other succinct bidirectional indexes were given in [1, 11, 12]. 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. [5] gave a new search algorithm, called *bidirectional search*, that utilizes the bidirectional property of the index. Consider the case $k = 2$, studied in [5]. 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. [5] performs three searches:

(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 S 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 S 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 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 $\{0, \ldots, k\}$. String π indicates the order in which the parts of P are processed, and thus it must satisfy the following property: For every $i > 1$, $\pi(i)$ is either $(\min_{j<i} \pi(j)) - 1$ or $(\max_{j<i} \pi(j)) + 1$. 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 M over integers, the *weight* of M is $\sum_i M[i]$. A search $S = (\pi, L, U)$ *covers* a string M if $L[i+1] \leq \sum_{j=1}^{i} M[j] \leq U[i]$ 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 M of weight k, there is a search in $\mathcal{S}$ that covers M. 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 $\mathcal{S}_{\text{LLTWWY}}$.

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

Uneven partition. In $\mathcal{S}_{\text{LLTWWY}}$, 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 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.

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 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. [5] 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 chosen 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.

Let $\#\mathrm{str}(S, X, \sigma, n)$ 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 size of the alphabet (note that m is not a parameter for #str since the value of m is implied from X). For a search scheme $\mathcal{S}$, $\#\mathrm{str}(\mathcal{S}, X, \sigma, n) = \sum_{S \in \mathcal{S}} \#\mathrm{str}(S, X, \sigma, n)$.

Fix S, X, σ, and n. Let $\mathcal{A}_l$ be the set of enumerated strings of length l when performing 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 as substrings. Select a random order on the elements of $\mathcal{A}_l$, and let $A_{l,i}$ be the i-th element of $\mathcal{A}_l$. By the linearity of the expectation,

$$\#\mathrm{str}(S, X, \sigma, n) = \sum_{l \geq 1} \sum_{i=1}^{n_l} \Pr_{T \in \Sigma^n}[A_{l,i} \text{ is a substring of } T],$$

where $n_l = |\mathcal{A}_l|$. For any l and i, the string $A_{l,i}$ is a random string with uniform distribution over Σ^l. Therefore, the probability that $A_{l,i}$ is a substring of T can be approximated by $1 - e^{-n/\sigma^l}$ using the Chen-Stein method [3]. Therefore,

$$\#\mathrm{str}(S, X, \sigma, n) \approx \sum_{l=1}^{m} n_l (1 - e^{-n/\sigma^l}). \tag{1}$$

In order to compute the values of n_l, we give some definitions. Let $n_{l,d}$ be the number of strings in $\mathcal{A}_l$ of length l with Hamming distance d to the prefix of P of length l. Let U' be a string obtained from U by replacing each character $U[i]$ of U by a run of $U[i]$ of length $x_{\pi(i)}$, where x_j is the length of the j-th part in the partition X. The string L' is defined analogously. In other words, the values $L'[i], U'[i]$ give a lower and upper bounds on the number of allowed mismatches for an enumerated string of length i. The values of n_l are given by the following

recurrence.

$$n_l = \sum_{d=L'[l]}^{U'[l]} n_{l,d}, \quad n_{l,d} = \begin{cases} n_{l-1,d} + (\sigma-1)n_{l-1,d-1} & \text{if } l \geq 1 \text{ and } L'[l] \leq d \leq U'[l] \\ 1 & \text{if } l = 0 \text{ and } d = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

For a specific search, a closed form formula can be given for n_l.

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 n_l increases exponentially, but the base of the exponent of n_l is $\sigma - 1$ whereas the base of $1 - e^{-n/\sigma^l}$ is $1/\sigma$. We can then approximate $\#\text{str}(S, X, \sigma, n)$ with function $\#\text{str}'(S, X, \sigma, n)$ defined by

$$\#\text{str}'(S, X, \sigma, n) = \sum_{l=1}^{\lceil \log_\sigma n \rceil + c_\sigma} n_l(1 - e^{-n/\sigma^l}), \tag{3}$$

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

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 $\#\text{str}(\mathcal{S}, X, \sigma, n)$ to account for this optimization.

3.2 Computing an optimal partition

Let p be the number of parts. An optimal partition can be naively found by enumerating all $\binom{m-1}{p-1}$ possible partitions, and for each partition X, computing $\#\text{str}'(\mathcal{S}, X, \sigma, n)$. We now describe a more efficient dynamic programming algorithm that computes an optimal partition for a given search scheme $\mathcal{S}$.

The algorithm takes advantage of the fact that the value of $\#\text{str}'(S, X, \sigma, n)$ does not depend on the entire partition X, but only on the partition of a substring of P of length $N = \lceil \log_\sigma n \rceil + c_\sigma$ induced by X. We first give some definitions. Define $\mathcal{S}(i, j)$ to be the set containing every search $S \in \mathcal{S}$ such that i appears before j in the π-string of S (we assume $\mathcal{S}(i, p+1) = \mathcal{S}$). A partition of P is a partition of the set $\{1, 2, \ldots, m\}$ into disjoint sets of the form $\{i, i+1, \ldots, j\}$ which will be called *intervals*. A partition of a substring $P[i..j]$ is a partition of $\{i, i+1, \ldots, j\}$ into disjoint intervals.

The algorithm builds *partial partitions* of prefixes of P, incrementally from left to right. That is, the partitions of $P[1..m'']$ are built from partitions of $P[1..m']$ for $m' < m''$ by extending them with interval $\{m'+1, \ldots, m''\}$. Consider a partition X' of $P[1..m']$, $m' \geq N$, into p' parts, and let p'_2 be the left-to-right rank of the interval in X' containing the position $m' - N + 1$. For a search $S \in \mathcal{S}(p'_2, p'+1)$, the value of $\#\text{str}'(S, X, \sigma, n)$ is the same for every extension of X' to a partition X of P, since $\#\text{str}'(S, X, \sigma, n)$ is determined by the partition of a substring of P of length N contained in $P[1..m']$. Thus, for every partial partition X', the algorithm computes the value of $\sum_{S \in \mathcal{S}(p'_2, p'+1)}(S, X, \sigma, n)$ for some

extension of X' to a partition X. This value will be denoted $v(X')$. Note that for a partition X of P, $v(X) = \#\mathrm{str}'(\mathcal{S}, X, \sigma, n)$. When the algorithm extends the partial partition X' to a partial partition $X'' = X' \cup \{m'+1, \ldots, m''\}$, it has to compute $v(X'')$. This is done by adding $v(X'') - v(X')$ to the already computed $v(X')$. By definition, $v(X'') - v(X') = \sum_{S \in \mathcal{S}(p_2'', p''+1) \setminus \mathcal{S}(p_2', p'+1)} \#\mathrm{str}'(S, X, \sigma, n)$, where $p'' = p' + 1$ and p_2'' is the left-to-right rank of the interval in X' containing the position $m'' - N + 1$. If X_1' and X_2' are partitions of $P[1..m']$ with the same number of parts and these partitions induce the same partition on $P[m' - N + 1..m']$, then for every partition X'' of $P[m'+1..m'']$ we have $v(X_1' \cup X'') - v(X_1') = v(X_2' \cup X'') - v(X_2')$. Thus, for every $p' \leq p$ and every partition X' of $P' = P[m' - N + 1..m']$, we need to store only one partition X^* of $P[1..m']$ into p' parts that induces the partition X' on P'. The partition X^* is chosen such that $v(X^*)$ is minimum among all partitions of $P[1..m']$ into p' parts that induce the partition X' on P'. We obtain an algorithm whose time complexity is $O(m^2 + (|\mathcal{S}|Nk + mp) \sum_{p'=1}^{p} \binom{N-1}{p'-1})$. Further details are omitted due to space limitations.

4 Properties of optimal search schemes

Designing an efficient search scheme for a given set of parameters consists of (1) choosing the number of parts, (2) choosing the searches, (3) choosing the 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 $\mathcal{S}_{\mathrm{LLTWWY}}$ 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 theorem gives the values of $\alpha(k, k+2)$ and $\alpha(k, k+1)$. We omit the proof due to space limitations.

Theorem 1. *$\alpha(k, k+1) = 013355\cdots kk$ for every odd k, and $\alpha(k, k+1) = 02244\cdots kk$ for every even k. $\alpha(k, k+2) = 0123\cdots(k-1)kk$ for every $k \geq 1$.*

Table 1. Values of $\#\text{str}(\mathcal{S}, X, 4, 4^{16})$ for 2-mismatch search schemes (recall that 4 is the size of the alphabet, and 4^{16} is the length of the text). The second column gives #str values for the 3-part search scheme with equal-size parts. The other columns give #str values for different search schemes using an optimal partition of the pattern. For each search scheme, the optimal value of #str is shown in the first sub-column, and the optimal partition 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

5 Case studies

In this section, we examine the efficiency of several 2-mismatch and 3-mismatch search schemes. The search schemes were generated by a greedy algorithm. At each step, the algorithm considers the uncovered string M of weight k such that the lexicographically minimal U-strings of searches that covers M is maximal. Among the searches that cover M 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 obtained an optimal partition and computed the efficiency of the scheme according to Section 3.

Results for 2 mismatches are given in Table 1 and Table 2, for 4-letter and 30-letter alphabets respectively, and results for 3 mismatches are given in Table 3.

Our theoretical analysis indicates that an increased number of parts combined with uneven partitioning improves over a scheme with $k+1$ equal sized parts when $m/(k+1)$ is smaller than $\log_\sigma n$ (details omitted due to lack of space). This is confirmed by the numerical results in Tables 1, 2, and 3, which show significant improvement when $m/(k+1)$ is small.

For big alphabets (Table 2), we observe a larger gain in efficiency. The is due to the fact that the values of n_l (see equation (2)) increase more rapidly when the alphabet is large, and thus a change in the size of parts can have a bigger influence on these values.

For 3 mismatches (Table 3), we observe smaller gain. This is partly explained by Theorem 1, as with 3 mismatches and 4 parts, the critical string starts with 01 (compared to 02 for 2 mismatches and 3 parts), therefore 4 parts provide already a competitive search. Another interesting observation is that with 4 parts, the optimal partition is an even one, as the U-strings in all searches in the 4-part scheme are the same.

We implemented our method using the *2BWT* library, provided by [5] (available at http://i.cs.hku.hk/2bwt-tools/) and experimentally compared different search schemes. The experiments were done on the sequence of human chromosome 14. The sequence is 88M long, with nucleotide distribution 29%, 21%, 21%, 29%. Searched patterns were obtained from Next-Generation Sequencing

Table 2. Values of $\#\mathrm{str}(\mathcal{S}, X, 30, 30^7)$ 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 $\#\mathrm{str}(\mathcal{S}, X, 4, 4^{16})$ for 3-mismatch search schemes.

m	4 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

Table 4. Total time (in seconds) of searching for one million patterns in human chromosome 14, up to 2 mismatches.

m	3 equal	3 unequal		4 equal	4 unequal	
18	230	230 (100%)	6,6,6	209 (91%)	203 (88%)	5,5,1,7
21	175	161 (92%)	8,6,7	147 (84%)	150 (86%)	6,4,3,8
24	142	120 (85%)	10,7,7	117 (82%)	107 (75%)	7,4,4,9
27	119	99 (83%)	12,7,8	96 (81%)	82 (69%)	9,4,5,9
30	101	84 (83%)	12,9,9	66 (65%)	68 (67%)	10,4,6,10
33	83	70 (84%)	13,10,10	53 (64%)	58 (70%)	11,5,6,11
36	68	66 (97%)	13,11,12	49 (72%)	50 (74%)	12,5,7,12
39	56	56 (100%)	13,13,13	48 (86%)	45 (80%)	13,6,7,13
42	45	45 (100%)	14,14,14	44 (98%)	38 (84%)	14,6,8,14

reads by cutting out strings of required length. Both the sequence and the reads were downloaded from http://gage.cbcb.umd.edu/data/. For every pattern length and every search scheme, 10^6 patterns were searched and the average number of enumerated strings was computed.

For the case of 2 mismatches, we implemented the 3-part and 4-part schemes from Section 5, as well as their equal part versions for comparison. For each pattern length, we computed an optimal partition, taking into account a non-identical distribution of nucleotides. Results are presented in Table 4. Using unequal parts for 3-part schemes yields a notable time decrease (8–17%) for patterns of length between 21 and 33. Furthermore, using 4 parts leads to an even more important improvement, showing a significantly better results for all pattern length compared to the 3-equal-parts scheme of [5]. For pattern lengths from 21 to 42 we observe 16–36% improvement in running time and 12–35% improvement in the number of enumerated strings. For pattern lengths 24, 27, 39, 42, we observe that using unequal part lengths for 4-part schemes is beneficial. Overall, the experimental results are consistent with numerical estimations of Section 5. However, for pattern lengths 30-36, the 4-equal-parts scheme performs better than the 4-unequal-parts one, which illustrates that the optimal partition found for random texts may not be the best one for genomic sequences.

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. The results of this paper can be extended to approximate string matching under edit distance. The estimation of n_l (Section 3) becomes more complicated though.

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. 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.
2. M. Burrow and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation, California, 1994.
3. L H. Y. Chen. Poisson approximation for dependent trials. *The Annals of Probability*, pages 534–545, 1975.
4. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Symposium on Foundation of Computer Science (FOCS)*, pages 390–398, 2000.
5. 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.
6. 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.
7. 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.
8. H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.
9. H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.
10. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.
11. L.M.S. Russo, G. Navarro, A.L. Oliveira, and P. Morales. Approximate string matching with compressed indexes. *Algorithms*, 2(3):1105–1136, 2009.
12. 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.
13. J.T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.
14. W.-K. Sung. Indexed approximate string matching. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms*, pages 1–99. Springer US, 2008.