Algorithms **2009**, *2*, 1105-1136; doi:10.3390/a2031105

OPEN ACCESS

algorithms

ISSN 1999-4893

www.mdpi.com/journal/algorithms

Article

Approximate String Matching with Compressed Indexes

Luís M. S. Russo [1,3,⋆], Gonzalo Navarro [2], Arlindo L. Oliveira [1,4], and Pedro Morales [2]

[1] INESC-ID, R. Alves Redol 9, 1000 Lisboa, Portugal; E-Mail: aml@inesc-id.pt (A.L.O.)

[2] Department of Computer Science, University of Chile, Avenida Blanco Encalada, 2120, 837-0459 Santiago, Chile Santiago, Chile; E-Mails: gnavarro@dcc.uchile.cl (G.N.); pemorale@dcc.uchile.cl (P.M.)

[3] CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

[4] Instituto Superior Técnico, Universidade Técnica de Lisboa, 1049-001 Lisboa, Portugal

⋆ Author to whom correspondence should be addressed; E-Mail: lsr@di.fct.unl.pt; Tel: +351 21 294 85 36 and Fax: +351 21 294 85 41.

Received: 9 July 2009; in revised form: 8 September 2009 / Accepted: 9 September 2009 / Published: 10 September 2009

Abstract: A compressed full-text self-index for a text T is a data structure requiring reduced space and able to search for patterns P in T. It can also reproduce any substring of T, thus actually replacing T. Despite the recent explosion of interest on compressed indexes, there has not been much progress on functionalities beyond the basic exact search. In this paper we focus on indexed approximate string matching (ASM), which is of great interest, say, in bioinformatics. We study ASM algorithms for Lempel-Ziv compressed indexes and for compressed suffix trees/arrays. Most compressed self-indexes belong to one of these classes. We start by adapting the classical method of partitioning into exact search to self-indexes, and optimize it over a representative of either class of self-index. Then, we show that a Lempel-Ziv index can be seen as an extension of the classical q-samples index. We give new insights on this type of index, which can be of independent interest, and then apply them to a Lempel-Ziv index. Finally, we improve hierarchical verification, a successful technique for sequential searching, so as to extend the matches of pattern pieces to the left or right. Most compressed suffix trees/arrays support the required bidirectionality, thus enabling the implementation of the improved technique. In turn, the improved verification largely reduces the accesses to the text, which are expensive in self-indexes. We show experimentally that our algorithms are competitive and provide useful space-time tradeoffs compared to classical indexes.

Keywords: compressed index; approximate string matching; Lempel-Ziv; compressed suffix tree; compressed suffix array

1. Introduction and Related Work

Approximate string matching (ASM) is an important problem that arises in applications related to text searching, pattern recognition, signal processing, and computational biology, to name a few. The problem consists in locating all the occurrences O of a given pattern string P, of size m, in a larger text string T, of size u, where the distance between P and O is less than a given threshold k. We focus on the edit distance, that is, the minimum number of character insertions, deletions, and substitutions of single characters to convert one string into the other.

The classical sequential search solution runs in $O(um)$ worst-case time (see [1]). An optimal average-case algorithm requires $O(u(k + \log_\sigma m)/m)$ time [2, 3], where σ is the size of the alphabet Σ. Those good average-case algorithms are called *filtration* algorithms: they traverse the text fast while checking for a simple necessary condition, and only when this condition holds do they verify the text area using a classical ASM algorithm. For long texts, however, sequential searching might be impractical as it must scan all the text. To avoid the scanning we use an *index* [4, 5], which is a data structure built on the text.

There exist indexes specifically devoted to ASM [6–9] (see [5] for an overview). Most are oriented to worst-case performance, and experiment a space-time barrier: They require either exponential time (in m or k), or exponential index space (*e.g.,* $O(u \log^k u)$ integers). This renders them impractical in many applications. Another choice is to reuse an index designed for exact searching, all of which are linear-space (*i.e.* $O(u)$ integers), and try to do ASM over it. Indexes such as suffix trees [10], suffix arrays [11], or based on so-called q-grams or q-samples [4], have been used for ASM. There exist several algorithms, based on suffix trees or arrays, which focus on worst-case performance [12–14]. They suffer from the same space-time barrier, achieving a search time independent of u but exponential in m or k. Essentially, they backtrack on the suffix tree or array, simulating the sequential search over all the text suffixes, and take advantage of the factoring of similar substrings achieved by suffix trees or arrays.

Indexes based on q-grams or q-samples are appealing because they require less space than suffix trees or arrays. The algorithms on those indexes do not offer worst-case guarantees, but perform well on average when the *error level* $\alpha = k/m$ is low enough, say $O(1/\log_\sigma u)$. Those indexes basically simulate a sequential filtration algorithm, such that the "necessary condition" checked involves exact matching of pattern substrings, and as such can be verified with any exact-searching index. Such filtration indexes, *e.g.,* [15, 16], cease to be useful for moderate α levels that are still of interest in applications.

The most successful approach, in practice, is in between backtracking and exact-matching filtration, and is called *hybrid* indexing. The index determines the text positions to verify using an approximate-matching condition over pattern pieces. Those positions are determined using backtracking, whose time is exponential in the length of the string or the number of errors. Yet, it is applied over short pattern pieces and allowing few errors, so that the exponential cost is controlled. Indexes of this kind offer average-case guarantees of the form $O(mn^\lambda)$ for some $0 < \lambda < 1$, and work well for higher error levels.

They have been implemented over q-gram indexes [17], suffix arrays [18], and q-sample indexes [19].

Yet, many of those linear-space indexes are very large anyway. For example, suffix arrays require 4 times the text size and suffix trees require at the very least 10 times [20]. In recent years a new and extremely successful class of indexes has emerged. *Compressed full-text indexes* use data compression techniques to produce less space-demanding data structures [21]. It turns out that data compression algorithms exploit the internal structure of a string much in the same way indexes do, and therefore it is possible to build a compressed index that takes space proportional to that of the compressed text, offers indexed searching, and replaces the text as it can reproduce any text substring (in which case they are called *self-indexes*). The size of those indexes is measured in terms of the empirical text entropy, H_k [22], which gives a lower bound on the number of bits per symbol achievable by a k-th order compressor. There are two main families of self-indexes [21]. One is based on Ziv-Lempel compression [23–28], and the other simulates suffix trees and suffix arrays [24, 29–35].

Compressed indexes usually require more operations than their classical counterpart, yet they can operate on a smaller/faster memory in the hierarchy. This is particularly noticeable when they can fit in RAM while the classical indexes must operate on disk. In this case their advantages are most striking. A recent survey covers the practical state of the art in compressed self-indexes [36].

Despite their great success, self-indexes have been mainly used for exact searching. Only very recently some indexes taking $O(u)$ or $O(u\sqrt{\log u})$ *bits* have appeared [8, 37, 38]. Those are specifically designed for ASM, focus on worst case, and unfortunately require time exponential in k [5].

Instead, in this paper we are interested in carrying out ASM over existing compressed full-text self-indexes, and in achieving good practical performance. We start by implementing the exact-match filtration approach over compressed self-indexes. This is the most direct solution because these indexes already provide exact searching. We choose a Ziv-Lempel self-index [25], as this type of index is fast to extract the text areas to be verified, and we optimize the process of extracting those text passages. We also try a suffix-array self-index [24] as these permit efficiently counting the number of occurrences, which allows us to optimize the pattern partitioning.

Then we turn our attention to hybrid indexing, which has been very successful on classical indexes. We show how the ideas of hybrid indexing over q-sample indexes [19] can be carried over Ziv-Lempel indexes, by regarding the text parsing as an irregular variant of the q-sampling. We obtain novel results about q-sample indexes that may find applications beyond compressed indexes.

Finally, we focus on suffix-tree-based self-indexes. Any ASM algorithm over suffix trees can be adapted to work over compressed ones. However, some adjustments are necessary because accessing the text in a self-index is around two orders of magnitude slower than in plain form. To address this problem we study the hierarchical verification technique [17, 39] and show that it can benefit from *bidirectionality*, that is, by extending edit distance computation from both ends of the string. This is particularly relevant on this type of self-indexes, as they permit moving in the text in either direction. This second index is slower than the Lempel-Ziv-based one, but it takes significantly less space. Both hybrid indexing techniques improve upon the results obtained with the exact-matching approach.

The paper has the following structure. In Section 2. we review some basic concepts about ASM, define some notation, and describe our contributions. Section 3. describes our application of exact-matching filters for ASM over self-indexes. Sections 4. and 5. are dedicated to hybrid ASM over Lempel-Ziv com-

pressed indexes. In Section 4. we explain our approach over a general q-samples index, but we assume the size of the samples is fixed. In Section 5. we tackle the problem of an irregular parsing. Section 6. is dedicated to our second hybrid approach, *i.e.* ASM over compressed suffix trees. Section 7. gives experimental results comparing all of our approaches with each other and with other ASM compressed indexes. The paper finishes with Section 8., where we draw some conclusions and point out future research directions.

Preliminary versions of our results were presented in *SPIRE* [40, 41]. In this paper we include previously unpublished ASM algorithms based on exact partitioning (Section 3.), as well as more exhaustive material on the previously published algorithms: the proofs of all the lemmas, illustrative examples of the algorithms, a characterization of homogeneous LZ-phrases (Section 5.3.), and more exaustive experimental results (Section 7.).

2. Basics and Contribution Overview

2.1. Basic Concepts

We denote by T the **text** string, by P the **pattern** string, and by O an occurrence of P in T; by Σ the strings' **alphabet**, of size σ; by $T[i]$ the symbol (or character) at position $(i \bmod u)$ of T; by $S.S'$ the **concatenation** of strings; by $S = S[..i-1].S[i..j].S[j+1..]$ respectively a **prefix**, a **substring**, and a **suffix** of a string S; by $S \sqsubseteq S'$ that S is a substring of S'; and by ε the empty string.

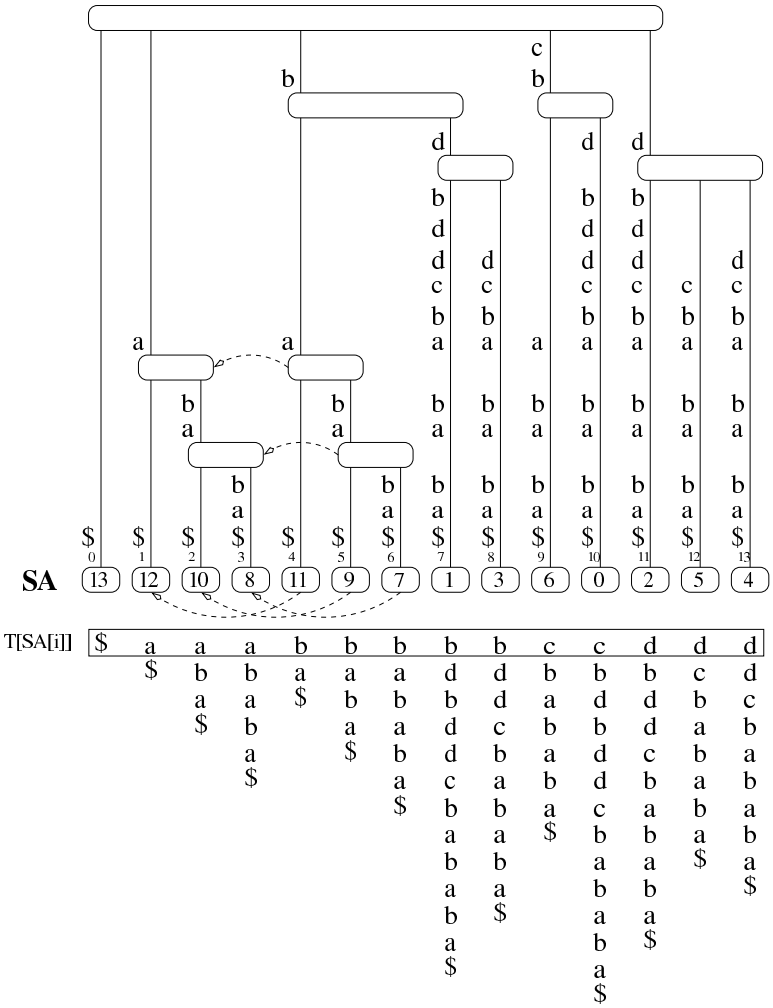
A **trie** is a character-labeled tree where no two children of a node have the same label. The concatenation of the labels from the root to a node v is called the **path-label** of v, and also denoted v. A **compact trie** is obtained by collapsing the unary paths in a trie into single edges, labeled with the concatenation of the collapsed symbols. The **suffix tree** [10, 42] of T is a compact trie such that each suffix of $T\$$ is the path-label of a leaf and vice versa, where $\$ \notin \Sigma$ is a terminator symbol. String labels in a suffix tree are substrings of T, thus they can be represented by a starting position in T plus a length. We will assume that u is the length of $T\$$. The **suffix array** [11] $SA[0, u-1]$ stores the suffix indexes of the leaves in lexicographical order. The suffix tree nodes can be identified with suffix array intervals: each node v corresponds to the range of leaves that descend from v. For a detailed explanation see, *e.g.,* Gusfield's book [43]. Figure 1 shows an example of a suffix tree and a suffix array.

Ziv-Lempel compression [23, 44–46] is based on cutting a text T into **phrases** of varying length, so that, essentially, each phrase already appears somewhere earlier in the text. Each phrase is encoded usually with a fixed number of bits, so more compressible texts generate longer phrases. There are many variants of this family, which basically differ in the rules to form the phrases from the previous text.

2.2. Approximate String Matching

The *edit* or *Levenshtein* distance between two strings, $ed(A, B)$, is the smallest number of edit operations that transform A into B. Valid operations are insertions, deletions, and substitutions of characters. Assuming we have a text T, the problem we are interested in solving in this paper is: Given a pattern P and an error limit k, determine all the (starting or ending positions of) substrings O of T for which $ed(P, O) \leq k$. We now review the most basic techniques related to this problem. As our running example consider that $P = abccba$, $k = 2$ and $T = abbbab$. The only occurrence O in T is $abbba$.

Figure 1. On top, suffix tree for *cbdbddcbababa*\$. Some suffix links are shown (dashed arrows). On the bottom, suffix array of T.

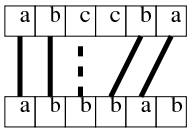


Dynamic programming. There is a well-known dynamic programming (DP) algorithm that computes matrix D, where $D[i,j] = ed(A[..i-1], B[..j-1])$ is the edit distance between the prefixes $A[..i-1]$ and $B[..j-1]$ of A and B. Figure 2 (middle) shows an example of matrix D for $A = abccba$ and $B = abbbab$. By looking at cell $D[6,6] = 3$ we can conclude that $ed(abccba, abbbab) = 3$. Let the size of A and B be m and m' respectively. This matrix can be computed, in $O(mm')$ time, by setting $D[0,0] = 0$ and

$$D[i,j] = \min \begin{cases} D[i-1,j]+1 & \text{if } i > 0 \\ D[i,j-1]+1 & \text{if } j > 0 \quad \textit{where} \\ D[i-1,j-1] + \delta_{A[i-1]=B[j-1]} & \text{if } i,j > 0 \end{cases}$$

$\delta_{x=y}$ is 0 if $x = y$ and 1 otherwise. Ukkonen [47] noted that in order to find cells in D whose value is k there is no need to compute cells with a value larger than k; those can be replaced by $+\infty$. The remaining cells are referred to as *active cells*. With this method, extending the computation of $ed(A,B)$ to $ed(Ac,B)$ or $ed(A,Bc)$ requires only $O(k)$ time: All we need to compute is another row, which can contain at most $2k+1$ entries that are $\leq k$ (as $D[i,j] \geq |i-j|$).

Figure 2. D table computation for strings $abccba$ and $abbbab$. Left: schematic representation of the alignment. Middle: computation of D. The numbers in bold correspond to the alignment shown on the left, and to the shortest path. Right: computation with increasing error bound.



		a	b	c	c	b	a	j:
	0	1	2	3	4	5	6	0
a	1	**0**	1	2	3	4	5	1
b	2	1	**0**	1	2	3	4	2
b	3	2	1	**1**	**1**	2	3	3
b	4	3	2	2	2	**2**	3	4
a	5	4	3	3	3	3	**2**	5
b	6	5	4	4	4	3	**3**	6
i:	0	1	2	3	4	5	6	

		a	b	c	c	b	a	j:
	0							0
a		0						1
b		1	0	1				2
b		2	1	1	1	2		3
b			2	2	2	2		4
a							2	5
b								6
i:	0	1	2	3	4	5	6	

One can regard the edit distance matrix as an **edit graph**: Each matrix position (i, j) is a node, which is the source of three arrows towards $(i, j+1)$ (of weight 1), $(i+1, j)$ (of weight 1), and $(i+1, j+1)$ (of weight $\delta_{A[i]=B[j]}$). Hence $ed(A, B)$ is the weight of the shortest path from $(0, 0)$ to (m, m').

Backtracking. A way of performing ASM over indexed text is as follows. Perform a depth-first search over the suffix tree of T, processing one letter at a time, while simultaneously computing the D table for P and O', where O' is the path-label of the node we are visiting. This table can be used to control the search. When we reach a point O' such that $ed(P, O') \leq k$, *i.e.* $D[|P|, |O'|] \leq k$, O' is reported as an occurrence. Usually we also report all the positions in T at which O' occurs, which means traversing the whole sub-tree of O' and reporting all its leaf positions. Otherwise if $ed(P, O') > k$ but there is at least one active cell in the last row, *i.e.* $D[i, |O'|] \leq k$ for some i, this means that $ed(P[..i-1], O') \leq k$ and therefore O' can potentially be extended into an occurrence, thus the search is allowed to proceed. If, on the other hand, there are no active cells in the last row of D, the search within the subtree of the current node can be abandoned. For example, by looking at Figure 2 we can conclude that the search should not proceed further after $abbbab$ because there are no active cells in the last row of the table. Also, since all the other rows contain active cells, this point is indeed reached by the search. It helps to think of D as a stack of rows that is growing downwards. Note that it is a convenient coincidence that the difference between the D tables of $ed(P, O')$ and $ed(P, O'c)$ is only the last row. This means that we can move between these two tables simply by adding or removing a row. At each step the DFS algorithm either pushes a new element into the stack, *i.e.* moves from $ed(P, O')$ to $ed(P, O'c)$, or it removes a row from the stack, *i.e.* moves from $ed(P, O'c)$ to $ed(P, O')$. The problem with backtracking is that the number of nodes it visits is at least exponential in k, and usually in m. Nevertheless, this process will be a key ingredient in our algorithms.

Filtration and sampling. Filtration algorithms build on the following lemma, simplified from [4].

Lemma 1 *Let A and B be strings such that* $ed(A, B) \leq k$. *Let* $A = A_1 A_2 \ldots A_j$, *for any* $j \geq 1$. *Then there is a substring* B' *of* B *and an* i *such that* $ed(A_i, B') \leq \lfloor k/j \rfloor$.

Consider for example $k = 2$ and $A = P = A_1.A_2.A_3$, $A_1 = ab$, $A_2 = cc$, $A_3 = ba$ and $B = O = abbba$. Since $ed(A, O) = 2$ and $j = 3$, we can conclude that there is a substring B' of B and an i such that $ed(A_i, B') \leq \lfloor 2/3 \rfloor = 0$. In particular $ed(A_1, ab) = 0$. In this case we also have that $ed(A_3, ba) = 0$.

Note from the example that, if $j = k + 1$, then the A_i's can be matched within B with zero errors, that is, using exact matching. If we assume that $A = P$ and B is contained in O, we can partition P into j non-overlapping parts and search T for them. Thus we must be able to find any text substring with the index (exactly [16] or approximately [17, 18], depending on j). Thus we must use a suffix tree or array [18], or even a q-gram index if we never use pieces of P longer than q [16, 17] (a q-gram index collects all text substrings of length q and stores a list of the positions of each different such text substring). The contexts around the occurrences found are then examined (say, with dynamic programming) for full occurrences of P.

Alternatively, if we assume that $B = P$ and A is contained in O, then we can index each s-th text q-gram (for some $s \geq q$), to form a so-called q-samples index. The different text q-samples are stored into, say, a trie data structure. This trie is traversed to find q-samples that match within P with at most $\lfloor k/j \rfloor$ errors. The contexts around those q-samples are then verified. This works as long as q and k are sufficiently small compared to m. More precisely, any occurrence O of P is of length at least $m - k$. Wherever an occurrence lies, it must contain at least $j = \lfloor (m - k - q + 1)/q \rfloor$ complete q-samples, and this defines s. This the basis of an "approximate q-samples" index [19].

2.3. Our Contributions

Reducing to exact matching of pattern pieces. One can use *any* compressed self-index to implement a filtration ASM method that relies on looking for exact occurrences of pattern substrings, as this is what all self-indexes provide. This is our approach in Section 3.. We first use a Lempel-Ziv index [25]. The Lempel-Ziv index works well because it is faster to extract the text to verify (recall that in self-indexes the text is not directly available). The specific structure of the Lempel-Ziv index used allows several interesting optimizations, such as factoring out the work of several text extractions. This is possible only when the occurrences span a single Ziv-Lempel phrase. We also use a compressed suffix array [24], as it permits fast counting of the number of occurrences of any substring. This allows us to quickly find the best partition $A = A_1.A_2 \ldots A_{k+1}$ so that the number of text verifications is minimized.

Hybrid indexing on Ziv-Lempel indexes. In Sections 4. and 5. we turn our attention to hybrid indexing, which is more complex but promises handling higher error levels α more efficiently. We still stick to a Ziv-Lempel index, as it allows fast extracting of text contexts. Considering the complications of Ziv-Lempel indexes when occurrences span more than one phrase, we pay special attention to this problem. A good choice to handle this turns out to be another Ziv-Lempel index [28], whose suffix-tree-like structure is useful for this approximate searching.

Mimicking q-sample indexes is particularly useful for our goals. A Lempel-Ziv parsing can be regarded as an irregular sampling of the text, and therefore our goal, in principle, is to adapt the techniques of the approximate q-samples index [19] to an irregular parsing (thus we must stick to the interpretation $B = P$, recall Section 2.2.). As desired, we would not need to consider occurrences spanning more than one phrase. Moreover, the trie of phrases stored by all Lempel-Ziv self-indexes is the exact analogous of the trie of q-samples. Thus we can search without requiring further structures.

The irregular parsing poses several challenges. There is no way to ensure that there will be a minimum number j of phrases contained in an occurrence. Occurrences could even be fully contained in a phrase!

We develop several tools to face those challenges. (1) We give a new variant of Lemma 1 that distributes the errors in a convenient way when the samples are of varying length. (2) We introduce a new filtration technique where the samples that overlap the occurrence (not only those contained in the occurrence) can be considered. This is of interest even for classical q-sample indexes. (3) We search for q-samples within long phrases to detect occurrences even if they are within a phrase. This technique also includes novel insights.

Hybrid indexing on compressed suffix trees. Finally, in Section 6. we explore the impact of *hierarchical verification* on hybrid searching, using a compressed suffix tree instead of a Lempel-Ziv index. Hierarchical verification means that an area that needs to be verified is not immediately checked with the maximum number of errors; instead the error threshold is raised gradually. This technique was originally proposed for a hybrid index [17] and later extended and used for a sequential algorithm [39]. However, these approaches used hierarchical verification directly over the text T, meaning that none of the repeated computation was factorized. We investigate precisely how to do this computation over the index, thus allowing us to avoid repeated computation. A key ingredient to achieve our goal is *bidirectionality*, that is, the ability of extending the distance computation to the left or to the right of the matched part.

Algorithms for computing edit distance are typically unidirectional, computing from left to right, because they are based on dynamic programming or automata. Interestingly this computation was made bidirectional more than 10 years ago [48]. They showed how to obtain the edit distance for strings A and cB by extending that for for strings A and B, where c is a letter.

Fortunately enough, most compressed suffix trees and arrays support bidirectionality. Typical indexes, classical suffix trees in particular, are unidirectional, meaning that they can search only by adding letters at the end of the search pattern. Due to the RANK/SELECT duality [21], bidirectionality arises naturally in a wide class of compressed indexes (Although not usual, there are some non-compressed indexes, such as affix trees [49], that support bidirectionality).

Combining these bidirectional algorithms we can use hierarchical verification directly over the index, instead of over T. Thus, we fill an important gap in indexed ASM. Moreover, while hybrid methods need careful tuning (where a small error can be disastrous), ours achieves a good performance without the need of tuning (and can be improved by tuning as well).

In addition, our work addresses a very important practical issue. Compressed suffix trees and arrays are usually self-indexes, meaning that they do not store the text T but they are able to obtain it. Yet, experimental results [28] show that this is still two orders of magnitude slower than storing T. This can easily be explained as cache effects, common in modern computer architectures. Efficient algorithms for

ASM over compressed indexes must therefore minimize their accesses to T. This completes the strong symbiotic exchange between hierarchical verification and compressed self-indexing, and provides a very important result for ASM over compressed indexes, both in theory and in practice.

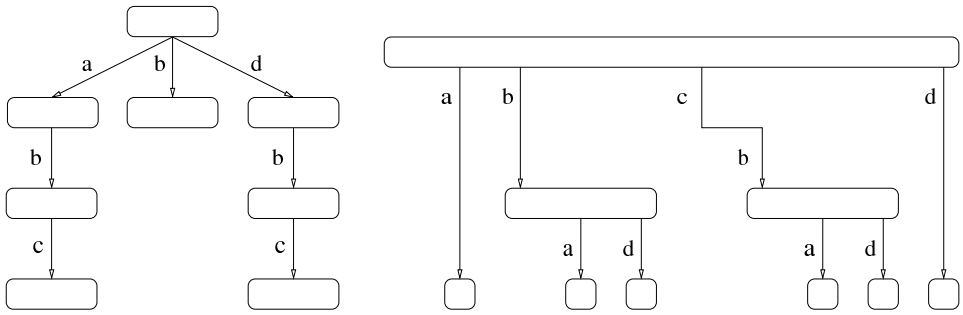
3. A Simple Self-Indexing Method

Any self-index can be used to materialize the search based on partitioning the pattern into $k+1$ pieces and searching for them without errors. Because each such occurrence must be verified with DP, one wishes an index able of extracting fast a context around each piece occurrence. This favors the choice of the LZ-index [25] for this task. Later in this section we will give reasons to try a different kind of index which, despite being slow to extract contexts, allows one to count very fast the number of occurrences of each piece.

3.1. Using the LZ-index

The LZ-index of a \$-terminated text $T[1,u]$ is based on the LZ78 parsing, which divides $T = Z_0, \ldots, Z_n$ into $n+1$ phrases Z_i so that $Z_0 = \varepsilon$ and each other Z_i is is equal to the longest possible phrase Z_j, $j < i$, plus a single character in Σ. Because the set of phrases is prefix-closed (*i.e.* every prefix of a phrase is also a phrase), the set of phrases $\{Z_i\}$ can be conveniently organized into a trie, so that each trie node is the path-label of some Z_i. This trie, called the *LZ-trie*, is one of the structures of the LZ-index, where it is represented in compressed form. Any phrase can be extracted in reverse order from the LZ-trie, by starting from its corresponding node and moving to the parent in the trie. This requires constant time per character and is efficient in practice [50]. The LZ-index requires $O(uH_k)$ bits. For example the LZ78 parsing of the string $T = ababacddbdbc$ is $a, b, ab, abc, d, db, dbc$. We show the corresponding LZ-trie in Figure 3, moreover we also show a tree composed of the reverse phrases which is also an important component of LZ-indexes.

Figure 3. (left) LZ-trie for strings $\{a, b, ab, abc, d, db, dbc\}$. (right) Reverse tree of the LZ78 trie.



The occurrences of a pattern are classified into three categories in the LZ-index. Those of type 1 fall within a single phrase Z_i. Those of type 2 span two consecutive phrases, that is, fall within some $Z_i.Z_{i+1}$. Those of type 3 span more than two phrases. Because in a LZ78 parsing all the phrases are different, there are at most $O(m^2)$ occurrences of type 3 overall. Occurrences of type 1 are found by first

spotting the type-1 occurrences that lie at the end of their phrase, that is, where P is a suffix of Z_i. Let us call those occurrences *primary*. Primary occurrences are found using other LZ-index structures (in particular, the trie of reverse phrases) in time $O(m)$, plus $O(1)$ per primary occurrence found. Now, it turns out that every type-1 occurrence descends from a primary occurrence in the LZ-trie (alternatively, each occurrence where P is inside Z_i but not a suffix of it, is inherited from an occurrence of P within the phrase copied by Z_i, from which Z_i descends in the LZ-trie). Thus all occurrences of type 1 are obtained by simply traversing all LZ-trie subtrees of the primary occurrences. Occurrences of type 2 and 3 are costlier to obtain.

A simple method for ASM is thus to split P into $k+1$ pieces, as evenly as possible, search for those pieces in the LZ-index, extract a context of $m+2k$ characters around the occurrence, and run DP over it. This approach will be called *LZI* in the experiments.

3.2. Improving the Basic Solution

In the sequel we introduce several improvements to the basic approach, to obtain what will be called *DLZI* in the experiments.

The basic approach carries out redundant work in occurrences of type 1. Assume we are looking for a piece P_r of $P = A_r.P_r.B_r$, $1 \leq r \leq k+1$. Given an occurrence of it, we must extract text substring L_r of length $|A_r|+k$ to the left, and R_r of length $|B_r|+k$ to the right of the occurrence P_r, for verification using DP. Consider a primary occurrence at LZ-trie node $v = Z_i$. To extract the context, we will traverse the upward path from v, so as to obtain the content of Z_i. If necessary, we will locate (using other fast LZ-index structures) the node v_{-1} corresponding to Z_{i-1}, v_{-2} for Z_{i-2}, and so on, until we have extracted $|L_r|$ characters. Now we locate v_{+1} to obtain Z_{i+1}, and so on until extracting $|R_r|$ characters to the right. Note that, towards the left, we can stop exactly when we have extracted $|L_r|$ characters, as the phrases are obtained right-to-left, whereas towards the right we must extract complete phrases until exceeding $|R_r|$, and then discard the unneeded characters from the last phrase.

As explained, we have to repeat this process for each descendant of v. Let $u = Z_j$ be a child of v by character c. The extraction of Z_j from u starts by obtaining c and then repeats exactly the process already carried out for $v = Z_i$ (as $Z_j = Z_i.c$). Our aim is to factor out that redundant work.

Given two strings S and X, let us define the *suffix edit distance* from S to X as

$$sed(S,X) \;=\; \min_j \; ed(S, X[j..]),$$

where we note that the formula is not symmetric. Similarly, we define the *prefix edit distance*

$$ped(S,X) \;=\; \min_j \; ed(S, X[..j]).$$

Note that $sed(S,X)$ is easily computed by traversing X backwards, whereas $ped(S,X)$ is easier if traversing X forwards. Now let P_r be a pattern piece found in the text, so that we have to verify the area $L_rP_rR_r$ with DP. The occurrence O minimizing $ed(P,O)$ may start anywhere in L_r and finish anywhere in R_r. More precisely, there will be an occurrence within $L_rP_rR_r$ iff

$$sed(A_r, L_r) + ped(B_r, R_r) \leq k.$$

We factor out the redundant work in the subtree of a primary occurrence $v = Z_i$ as follows. First, we obtain Z_i and compute $sed(A_r, Z_i)$, remembering the last row L_{row} of the DP matrix (where we have processed A_r and Z_i in reverse order, and assuming Z_i corresponds to the rows). Indeed, if $|Z_i| \geq |L_r^v|$ (where L_r^v is the L_r text substring corresponding to occurrence v), we only process the last $|L_r^v|$ characters of Z_i. Otherwise, we obtain the nodes v_{-1}, v_{-2}, and so on until obtaining L_r^v and computing $k' = sed(A_r, L_r^v)$. If $k' > k$, there is no possible occurrence around v. Otherwise, we obtain and process v_{+1}, v_{+2}, until either we obtain $ped(B_r, Z_{i+1}.Z_{i+2}\ldots) \leq k - k'$ or we process more than $|R_r^v|$ characters.

Now we traverse the subtree of v while avoiding redundant work. We set up a new DP matrix between B_r and an initially empty string, and traverse the subtree while carrying out a process similar to that described in Section 2.2. so that, when we are at node $u = Z_j$, the matrix corresponds to the edit distance computation between B_r and u. Let R_{row} be the last row of this matrix upon reaching u. Now we extract u_{-1}, u_{-2}, and so on, so as to compute $k' = sed(A_r, L_r^u)$ starting from L_{row}, that is, we avoid reprocessing u itself. Then we extract u_{+1}, u_{+2}, and so on until finding $\leq k - k'$ errors or processing $|R_r^u|$ characters.

3.3. Using the FM-index

A problem when partitioning into $k+1$ pieces is that some may be much more frequent than others. Since any partitioning into $k+1$ pieces is good, it is natural to look for the partitioning yielding the least number of verifications to carry out. This can be easily found in time $O(m^2k)$ with (a different) dynamic programming [16]. However, we need to know the frequencies in the text of every pattern substring. That is, we need to be able of efficiently *counting* the number of occurrences of a string.

The LZ-index is not good at this task: It can only count by essentially locating all of the occurrences. We tried some approximate counting methods with no success (more precisely, the final search time decreased, but the time for counting exceeded the gains by far).

Suffix-array based indexes, instead, are very efficient at counting. In particular, we used a variant of the FM-index [24, 36], which is a self-index that simulates a suffix array within $O(uH_k)$ bits of space. The FM-index can easily count the number of occurrences of all the substrings of P in $O(m^2)$ time. Within the same time, the suffix array intervals corresponding to each of the pieces is found. Once the optimal partition is chosen, a context around each occurrence of each of the P_r pieces is obtained, and DP is run on the context to find the approximate matches.

For the experiments we used an FM-index variant that favors speed over space, described in [50, Sec. 12.2]. As optimizing the partition did reduce the overall time, we show only this variant in the experiments.

4. An Improved q-samples Index

In this section we extend classical q-sample indexes by allowing samples to overlap the pattern occurrences. This is of interest by itself, and will be used for an irregular sampling index later. Recall that a q-samples index stores the locations, in T, of all the substrings $T[qi..qi+q-1]$.

4.1. Varying the Error Distribution

We will need to consider parts of samples in the sequel, as well as samples of different lengths. Lemma 1 gives the same number of errors to all the samples, which is disadvantageous when pieces are of different lengths. The next Lemma generalizes Lemma 1 to allow different numbers of errors in each piece.

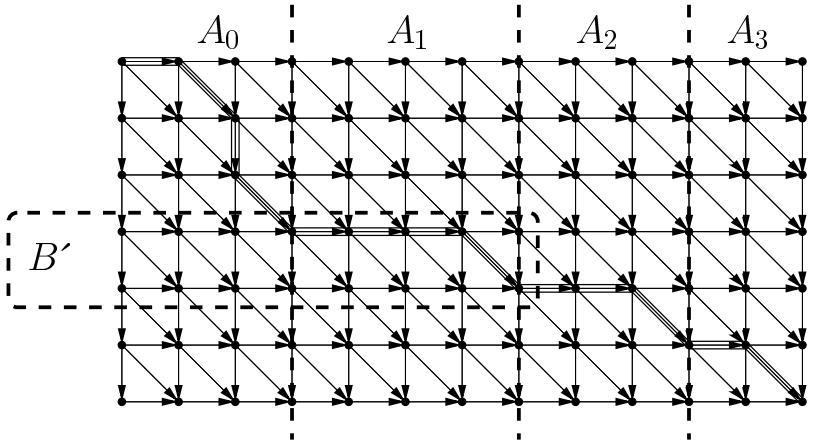
Lemma 2 *Let A and B be strings, let* $A = A_1.A_2 \dots A_j$*, for strings* A_i *and some* $j \geq 1$*. Let* $k_i \in \mathbb{R}$ *such that* $\sum_{i=1}^{j} k_i > ed(A, B)$*. Then there is a substring* B' *of* B *and an* i *such that* $ed(A_i, B') < k_i$*.*

Proof The edit distance between A and B corresponds to the shortest path in the edit graph of A and B. The partition of A induces a partition in this graph and in particular splits the shortest path. Let B_i be the substring of B that shares the shortest path with A_i. This means that $\sum_{i=1}^{j} ed(A_i, B_i) = ed(A, B)$.

Suppose, by absurd, that for every B' substring of B and every i we have that $ed(A_i, B') \geq k_i$. Therefore, this is also true for the B_i's, *i.e.* $ed(A_i, B_i) \geq k_i$. This is absurd since that way $ed(A, B) = \sum_{i=1}^{j} ed(A_i, B_i) \geq \sum_{i=1}^{j} k_i > ed(A, B)$. Therefore, there must exist an i such that $ed(A_i, B_i) < k_i$. Note that B_i is the substring B' we mention in the lemma.

Figure 4 shows an illustration of this Lemma.

Figure 4. A schematic edit distance graph between A and B. The dashed lines show the division of A and B into the A_i's and delimit B'. The shortest path in the graph is enclosed between two parallel lines.



Lemma 1 is a particular case of Lemma 2: set $k_i = k/j + \epsilon$ for sufficiently small $\epsilon > 0$.

Consider the application of Lemma 1 to $k = 9$, $A = A_1.A_2.A_3$, $A_1 = ab$, $A_2 = cdefghijklm$, $A_3 = no$ and $B = axbcxdxexfgxhixjkxlxmnxo$ (see Figure 5). We can conclude that there is a substring B' of B and an i such that $ed(B', A_i) \leq \lfloor 9/3 \rfloor = 3$. In particular, we have that $ed(A_1, axb) = 1$ and $ed(A_3, nxo) = 2$. This number of errors (3) is excessively high for strings a as small as A_1 and A_3. Instead, we can use Lemma 2 with $k_1 = k_3 = 1.2$ and $k_2 = 6.7$; note that $k_1 + k_2 + k_3 = 9.1 > 9$. Using this lemma, we only allow for 1 error for A_1 and A_3. In this case we have that $ed(A_1, axb) = 1 < 1.2$. We also have that $ed(A_3, nxo) = 1 < 1.2$.

Figure 5. Dynamic programming table D for strings, $B = P = axbcxdxexfgxhixjkxlxmnxo$ (vertical) and $A = O = abcdefghijklmno$ (horizontal) with $k = 9$. The shortest path in the edit graph is shown in bold, we do not show inactive cells.

		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	0	1	2	3	4	5	6	7	8	9						
a	1	**0**	1	2	3	4	5	6	7	8	9					
x	2	**1**	1	2	3	4	5	6	7	8	9					
b	3	2	**1**	2	3	4	5	6	7	8	9					
c	4	3	2	**1**	2	3	4	5	6	7	8	9				
x	5	4	3	**2**	2	3	4	5	6	7	8	9				
d	6	5	4	3	**2**	3	4	5	6	7	8	9				
x	7	6	5	4	**3**	3	4	5	6	7	8	9				
e	8	7	6	5	4	**3**	4	5	6	7	8	9				
x	9	8	7	6	5	**4**	4	5	6	7	8	9				
f		9	8	7	6	5	**4**	5	6	7	8	9				
g			9	8	7	6	5	**4**	5	6	7	8	9			
x				9	8	7	6	**5**	5	6	7	8	9			
h					9	8	7	6	**5**	6	7	8	9			
i						9	8	7	6	**5**	6	7	8	9		
x							9	8	7	**6**	6	7	8	9		
j								9	8	7	**6**	7	8	9		
k									9	8	7	**6**	7	8	9	
x										9	8	**7**	7	8	9	
l											9	8	**7**	8	9	
x												9	**8**	8	9	
m													9	**8**	9	
n														9	**8**	9
x															**9**	9
o																**9**

An early version of Lemma 2 was given in the context of hierarchical verification [39]. This result was later improved [4] as follows.

Lemma 3 *Let A and B be strings, let* $A = A_1.A_2 \ldots A_j$, *for strings* A_i *and some* $j \geq 1$. *Let* $k_i \in \mathbb{N}_0$ *such that* $j \geq 1$ *and* $\sum_{i=1}^{j} k_i \geq ed(A, B) - j + 1$. *Then there is a substring* B' *of* B *and an* i *such that* $ed(A_i, B') \leq k_i$.

Proof We will prove this lemma using Lemma 2. Let k_i be the numbers referred in this lemma. Consider $k'_i = k_i + 1$. The k'_i numbers satisfy the conditions of Lemma 2, *i.e.* $\sum_{i=1}^{j} k'_i = j + \sum_{i=1}^{j} k_i \geq j - j + 1 + k = k + 1 > k$. Therefore, by Lemma 2, we can conclude that there is a substring B'

of B and an i such that $ed(A_i, B') < k'_i = k_i + 1$. Since $ed(A_i, B')$ and k_i are integers we have that $ed(A_i, B') \leq k_i$.

The proof shows that Lemma 3 is a consequence of Lemma 2. In fact, both are equivalent. To prove that Lemma 3 implies Lemma 2 use the transformation $k'_i = \lceil k_i \rceil - 1$ and reason as above.

In our version the k_i's can be real numbers. However, this is not the case of Lemma 3. For example with $k = 3$, $k_i = 1/3$ and $j = 3$ we conclude that some A_i must occur with at most $1/3$ errors. Since the edit distance between two strings is always an integer, the conclusion would be that $ed(A_i, B') \leq 0$, which is not correct. With Lemma 2, we can expose our results without being forced to work with integers, *i.e.* having to deal with floors and ceilings. Also, we do not need to know j a priori, which is important since in our application we will not know the value of j before locating O in T.

Lemma 2 can be used to adapt the error levels to the length of the pieces. For example, it is appropriate to try to maintain a constant error level, by taking $k_i = (1 + \epsilon) \cdot k \cdot |A_i|/|A|$ for any $\epsilon > 0$. In our recent example, this would yield $k_1 = k_3 = (1 + \epsilon) \cdot 9 \times 2/15 = (1 + \epsilon) \cdot 1.2$ and $k_2 = (1 + \epsilon) \cdot 9 \times 11/15 = (1 + \epsilon) \cdot 6.6$. This gives essentially the same result we presented before.

4.2. Partial q-sample Matching

Contrary to all previous work, let us assume that A in Lemma 2 is not only that part of an approximate occurrence O formed by full q-samples, but instead that $A = O$, so that A_1 is the suffix of a sample and A_j is the prefix of a sample. An advantage of this is that now the number of involved q-samples is at least $j = \lceil (m - k)/q \rceil$, and therefore we can permit fewer errors per piece (*e.g.*, $\lfloor k/j \rfloor$ using Lemma 1). On the other hand, we would like to allow fewer errors for the pieces A_1 and A_j. Yet, notice that any text q-sample can participate as A_1, A_j, or as a fully contained q-sample in different occurrences at different text positions. Lemma 2 tells us that we could allow $k_i = (1 + \epsilon) \cdot k \cdot |A_i|/|A|$ errors for A_i, for any $\epsilon > 0$. Conservatively, this is $k_i = (1 + \epsilon) \cdot k \cdot q/(m - k)$ for $1 < i < j$, and less for the extremes.

In order to adapt the trie searching technique to those partial q-samples, we should not only search all the text q-samples with $(1 + \epsilon) \cdot k \cdot q/(m - k)$, but also all their prefixes and suffixes with fewer errors. This includes, for example, verifying all the q-samples whose first or last character appears in P (cases $|A_1| = 1$ and $|A_j| = 1$). This is unaffordable. Our approach will be to redistribute the errors across A using Lemma 2 in a different way to ensure that only sufficiently long q-sample prefixes and suffixes are considered.

Let v be a non-negative integer parameter. We associate to every letter of A a *weight*: the first and last v letters have weight 0 and the remaining letters have weight $(1 + \epsilon)/(|A| - 2v)$. We define $|A_i|_v$ as the sum of the weights of the letters of A_i. For example if A_i is within the first v letters of A then $|A_i|_v = 0$; if it does not contain any of the first or last v letters then $|A_i|_v = (1 + \epsilon) \cdot |A_i|/(|A| - 2v)$. Notice that for this definition to be sound we need that $|A| - 2v > 0$, *i.e.* $v < |A|/2$.

We can now apply Lemma 2 with $k_i = k \cdot |A_i|_v$ provided that $k > 0$. Note that $\sum_{i=1}^{j} k_i = (1+\epsilon) \cdot k > k$. In this case, if $|A_1| \leq v$ we have that $k_1 = 0$ and therefore A_1 can never be found with *strictly less* than zero errors. The same holds for A_j. This effectively relieves us from searching for any q-sample prefix or suffix of length at most v.

Parameter v is thus doing the job of discarding q-samples that have very little overlap with the occur-

rence $O = A$, and maintaining the rest. It balances between two exponential costs: one due to verifying all the occurrences of too short prefixes/suffixes, and another due to permitting too many errors when searching for the pieces in the trie. In practice tuning this parameter will have a very significant impact on performance.

Recall the application of Lemma 2 with $k = 9$, $A = A_1.A_2.A_3$, $A_1 = ab$, $A_2 = cdefghijklm$, $A_3 = no$, $k_1 = k_3 = 1.2$, $k_2 = 6.7$ and $B = axbcxdxexfgxhixjkxlxmnxo$. In this case we would need to search for all the q-samples that end in a string that is at edit distance 1 from $A_1 = ab$. This, in particular, includes all the q-samples that contain the suffix a, all the q-samples that contain the suffix b and so on. This yields an excessive number of samples to verify. If we consider $v = 1$ in this example we will instead obtain $k_1 = k_3 = (1+\epsilon) \cdot 9 \times 1/13 \approx 0.7$ and $k_2 = (1+\epsilon) \cdot 9 \times 11/13 \approx 7.7$. Note that $k_1 + k_2 + k_3 = 9.1 > 9$. In our example we have that $ed(A_2, cxdxexfgxhixjkxlxm) = 7 < 7.7$. Note that, in this example, we moved the errors to A_2 since A_1 and A_3 are very small.

4.3. A Hybrid q-samples Index

We have explained all the ideas necessary to describe a hybrid q-samples index. The algorithm works in two steps. First we determine all the q-samples O_i for which $ed(O_i, P') < k \cdot |O_i|_v$ for some substring P' of P. In this phase we also determine the q-samples that contain a suffix O_1 for which $ed(O_1, P') < k \cdot |O_1|_v$ for some prefix P' of P (note that we do not need to consider substrings of P, just prefixes). Likewise we also determine the q-samples that contain a prefix O_j for which $ed(O_j, P') < k \cdot |O_j|_v$ for some suffix P' of P (similar observation). The q-samples that qualify are potentially contained inside an approximate occurrence of P, *i.e.* O_i may be a substring of a string O such that $ed(O, P) \leq k$. In order to verify whether this is the case, in the second phase we scan the text context around O_i with a sequential algorithm.

As the reader might have noticed, the problem of verifying conditions such as $ed(O_i, P') < k \cdot |O_i|_v$ is that we cannot know a priori which i does a given text q-sample correspond to. Different occurrences of the q-sample in the text could participate in different positions of an O, and even a single occurrence in T could appear in several different O's. We do not know either the size $|O|$, as it may range from $m - k$ to $m + k$.

A simple solution is as follows. Conservatively assume $|O| = m-k$. Then, search P for each different text q-sample in three roles: (1) as a q-sample contained in O, so that $|O_i| = q$, assuming pessimistically $|O_i|_v = (1+\epsilon)\min(q/(m-k-2v), 1)$; (2) as an O_1, matching a prefix of P for each of the q-sample suffixes of lengths $v < \ell < q$, assuming $|O_1| = \ell$ and thus $|O_1|_v = (1+\epsilon)\ \min((\ell-v)/(m-k-2v), 1)$; (3) as an O_j, matching a suffix of P for each of the q-sample prefixes, similarly to case (2) (*i.e.* $|O_j|_v = |O_1|_v$). We assume that $q < m - k$ and thus the case of O contained inside a q-sample does not occur.

In practice, one does not search for each q-sample in isolation, but rather factors out the work due to common q-gram prefixes by backtracking over the trie and incrementally computing the DP matrix between every different q-sample and any substring of P ([4], recall Section 2.2.). We note that the trie of q-samples is appropriate for role (3), but not particularly efficient for roles (1) and (2) (finding q-samples with some specific suffix). In our application to a Lempel-Ziv index this will not be a problem because we will have also a trie of the reversed phrases (that will replace the q-grams).

5. Using a Lempel-Ziv Self-Index

We now adapt our technique of Section 4. to the irregular parsing of phrases produced by a Lempel-Ziv-based index. Among the several alternatives [24–28], we will focus on the ILZI [28], for the convenient suffix-tree-like structure of phrases generated. The ILZI partitions the text into phrases such that every suffix of a phrase is also a phrase (similarly to LZ78 parsing [23], where every prefix of a phrase is also a phrase). It uses two tries, one storing the phrases and another storing the reverse phrases. In addition, it stores a mapping that permits moving from one trie to the other, and it stores the compressed text as a sequence of phrase identifiers. This index [28] has been shown to require $O(uH_k)$ bits of space, and to be efficient in practice.

5.1. Handling Different Lengths

As explained, the main idea is to use the phrases instead of q-samples. For this sake Lemma 2 solves the problem of distributing the errors homogeneously across phrases. However, other problems arise especially for long phrases. For example, an occurrence could be completely inside a phrase. In general, backtracking over long phrases is too costly.

Even when an occurrence is not completely contained inside a phrase, it is not viable to start searching for an occurrence with too many errors, even if the occurrence is long. This is essentially the argument in favor of filtration and hybrid indexes. Consider our example with $k = 9$, $A = A_1.A_2.A_3$, $A_1 = ab$, $A_2 = cdefghijklm$, $A_3 = no$, $k_1 = k_3 = 0.7$, $k_2 = 7.7$ and $B = axbcxdxexfgxhixjkxlxmnxo$. Recall also that we consider $P = B$ and $O = A$. It is not viable to search for all the samples A' such that $ed(A', cxdxexfgxhixjkxlxm) \leq 7$, by using backtracking. The problem is essentially that, in this way, we allow for all the seven errors to concentrate in the beginning of A', *i.e.* we have to potentially consider for σ^7 strings. Only a very small fraction of these strings will end up extending to a sample A' that verifies $ed(A', cxdxexfgxhixjkxlxm) \leq 7$. To avoid this problem we adopt a hybrid approach.

We resort again to q-samples, this time within phrases. We choose two non-negative integer parameters q and $s < q$. We will look for any q-gram of P that appears with less than s errors within any phrase. All phrases spotted along this process must be verified. Still, some phrases not containing any pattern q-gram with $< s$ errors can participate in an occurrence of P (*e.g.,* if $\lfloor (m - k - q + 1)/q \rfloor \cdot s \leq k$ or if the phrase is shorter than q). Next we show that those remaining phrases have a certain structure that makes them easy to find.

Lemma 4 *Let A and B be strings and q and s be integers such that $0 \leq s < q \leq |A|$ and for any substrings B' of B and A' of A with $|A'| = q$ we have that $ed(A', B') \geq s$. Then for every prefix A' of A there is a substring B' of B such that $ed(A', B') \leq ed(A, B) - s\lfloor (|A| - |A'|)/q \rfloor$.*

Proof This is an application of Lemma 2 and it is important because it gives a good description of the restrictions obtained by the filtration.

Apply Lemma 2 with $A_1 = A'$, $|A_j| < q$ and the remaining A_i's of size q, *i.e.* $|A_i| = q$. Consider $k_1 = ed(A, B) - s\lfloor (|A| - |A'|)/q \rfloor + \epsilon$ with $0 < \epsilon < 1$, $k_j = 0$. The remaining k_i's are equal to s. Note that $j = \lfloor (|A| - |A'|)/q \rfloor + 1$ and therefore $\sum_{i=1}^{j} k_i = ed(A, B) + \epsilon - s\lfloor (|A| - |A'|)/q \rfloor + s(j - 1) = ed(A, B) + \epsilon - s\lfloor (|A| - |A'|)/q \rfloor + s\lfloor (|A| - |A'|)/q \rfloor = ed(A, B) + \epsilon > ed(A, B)$.

Therefore we conclude that there is a substring B' of B and an i such that $ed(A_i, B') < k_i$. We will prove that it must be $i = 1$. If $i = j$, then $ed(A_j, B') < k_j = 0$, which is impossible. If $1 < i < j$, then $ed(A_i, B') < k_i = s$ with $|A_i| = q$, which contradicts the hypotheses of the lemma. Therefore $i = 1$ and $ed(A_1, B') = ed(A', B') < k_1 = ed(A, B) - s\lfloor(|A| - |A'|)/q\rfloor + \epsilon$, which means that $ed(A', B') \leq ed(A, B) - s\lfloor(|A| - |A'|)/q\rfloor$.

The lemma implies that, if a phrase is close to a substring of P, but none of its q-grams are sufficiently close to any substring of P, then the errors must be distributed uniformly along the phrase. Therefore we can check the phrase progressively (for increasing prefixes), so that the number of errors permitted grows slowly. This severely limits the necessary backtracking to find those phrases that escape from the q-gram-based search.

Let us consider $q = 4$ in our running example. Note that for a substring A' of A of size 4 we have that $k \cdot |A'|_v = (1 + \epsilon) \cdot 9 \times 4/13 \approx 2.8$. Therefore we choose $s = 2$. Note that we cannot use Lemma 4 with $A = A_2 = cdefghijklm$ and $B = cxdxexfgxhixjkxlxm$, since $ed(fghi, fgxhi) = 1 < 2 = s$ and $|fghi| = 4$. This means that there is an exceptionally well preserved area of B in A where we only deleted 1 letter out of 5 consecutive ones. Consider instead that $A = cdefghijklm$ and that $B = cxdxexfgxhxijxkxlm$. In this case we can use Lemma 4 and obtain bound $ed(cdefghijklm[..i], B') \leq 7 - 2\lfloor(11 - i - 1)/4\rfloor$ for some substrings B' of B. Note in particular that this is tight for $i = 6$, *i.e.* $ed(cdefghijklm[..6], B') = ed(cdefghi, cxdxexfgxhxi) = 5 = 7 - 2 \times \lfloor 4/4\rfloor$.

Parameter s enables us to balance between two search costs. If we set it low, then the q-gram-based search will be stricter and faster, but the search for the escaping phrases will be costlier. If we set it high, most of the cost will be absorbed by the q-gram search.

5.2. A Hybrid Lempel-Ziv Index

The following lemma describes how we combine previous results to search using a Lempel-Ziv index.

Lemma 5 *Let A and B be strings such that $0 < ed(A, B) \leq k$. Let $A = A_1.A_2 \ldots A_j$, for strings A_i and some $j \geq 1$. Let q, s and v be integers such that $0 \leq s < q \leq |A|$ and $0 \leq v < |A|/2$. Then there is a substring B' of B and an i such that either:*

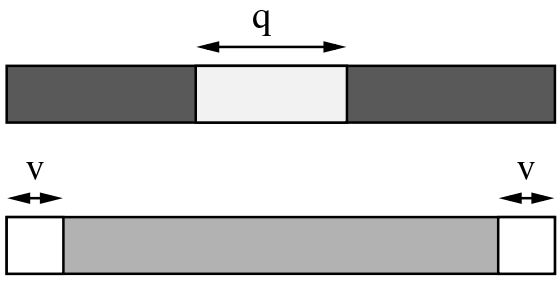
1. *there is a substring A' of A_i with $|A'| = q$ and $ed(A', B') < s$, or*
2. *$ed(A_i, B') < k \cdot |A_i|_v$ in which case for any prefix A' of A_i there exists a substring B'' of B' such that $ed(A', B'') < k \cdot |A_i|_v - s\lfloor(|A_i| - |A'|)/q\rfloor$.*

Proof As we have explained, our approach first consists in applying Lemma 2 considering $k_i = k \cdot |A_i|_v$ for $0 \leq i \leq j$. Observe that, by the definition of $|A_i|_v$, we have that $\sum_{i=1}^{j} k \cdot |A_i|_v = k(\epsilon + (|A| - 2v)/(|A| - 2v)) = k(\epsilon + 1) > k$. Now, we classify the resulting A_i into one of the two classes we defined before. The first class justifies the first condition of this Lemma. If the A_i belongs to the second class, we apply Lemma 4 with the resulting k_i.

Figure 6 shows a schematic representation of Lemma 5.

As before the search runs in two phases. In the first phase we find the phrases whose text context must be verified. In the second phase we verify those text contexts for an approximate occurrence of P. Lemma 5 gives the key to the first phase. We find the relevant phrases via two searches:

Figure 6. A schematic representation of the two possible cases mentioned in Lemma 5. The boxes represent the A string and the density of the filling represents the density of errors with B.



(1) We look for any q-gram contained in a phrase which matches within P with less than s errors. We backtrack in the trie of phrases for every $P[y_1..]$, descending in the trie and advancing y_2 in $P[y_1, y_2]$ while computing the DP matrix between the current trie node and $P[y_1, y_2]$. We look for all trie nodes at depth q that match some $P[y_1, y_2]$ with less than s errors. Since every suffix of a phrase is a phrase in the ILZI, every q-gram within any phrase can be found starting from the root of the trie of phrases. All the phrases Z that descend from each q-gram trie node found must be verified (those are the phrases that start with that q-gram). We must also spot the phrases suffixed by each such Z. Hence we map each phrase Z to the trie of reverse phrases and also verify all the descent of the reverse trie nodes. This covers case 1 of Lemma 5.

This is precisely the case in our example, since we have that $q = 4$ and $ed(fghi, P[9..13]) = ed(fghi, fgxhi) = 1 < 2 = s$. We then determine that the phrase $cdefghijklm$ contains the string $fghi$ and locate the occurrence by searching the context around that phrase.

(2) We look for any phrase A_i matching a portion of P with less than $k \cdot |A_i|_v$ errors. This is done over the trie of phrases. Yet, as we go down in the trie (thus considering longer phrases), we can enforce that the number of errors found up to depth d must be less than $k \cdot |A_i|_v - s\lfloor(|A_i| - d)/q\rfloor$. This covers case 2 in Lemma 5, where the equations vary according to the roles described in Section 4.3. (***i.e.*** depending on i):

(2.1) $1 < i < j$, in which case we are considering a phrase contained inside O that is not a prefix nor a suffix. The formula $k \cdot |A_i|_v$ (both for the matching condition and the backtracking limit) can be bounded by $(1+\epsilon) \cdot k \cdot \min(|A_i|/(m-k-2v), 1)$, which depends on $|A_i|$. Since A_i may correspond to any trie node that descends from the current one, we determine *a priori* which $|A_i| \leq m - k$ maximizes the backtracking limit. We apply the backtracking for each $P[y_1..]$.

This is what occurs in the variation of our example with $P = axbcxdxexfgxhxijxkxlmnxo$, that obtains the limit $ed(cdefghijklm[..d+1], B') \leq 7.7 - 2\lfloor(11-d)/4\rfloor$ for some substring B' of $P[3..13] = cxdxexfgxhxijxkxlm$. Note that, in particular, this limit is valid for string $cdefghijklm$. Any string for which this is not the case is abandoned by the search.

(2.2) $i = j$, in which case we are considering a phrase that starts by a suffix of O. Now $k \cdot |A_i|_v$ can be bounded by $(1+\epsilon) \cdot k \cdot \min((d-v)/(m-k-2v), 1)$, yet still the limit depends on $|A_i|$ and must be maximized a priori. This time we are only interested in suffixes of P, that is, we can perform m searches with $y_2 = m$ and different y_1. If a node verifies the condition we must consider also those that descend

from it, to get all the phrases that start with the same suffix.

(2.3) $i = 1$, in which case we are considering a phrase that ends in a prefix of O. This search is as the case $i = j$, with similar formulas. We are only interested in prefixes of P, that is $y_1 = 0$. As the phrases are suffix-closed, we can conduct a single search for $P[0..]$ from the trie root, finding all phrase suffixes that match each prefix of P. Each such suffix node must be mapped to the reverse trie and the descent there must be included. The case $i = j = 1$ is different, as it includes the case where O is contained inside a phrase. In this case we do not require the matching trie nodes to be suffixes, but also prefixes of suffixes. That is, we include the descent of the trie nodes and map each node in that descent to the reverse trie, just as in case 1.

5.3. Homogeneous Lempel-Ziv Phrases

In this section we give further insight into the structure of the homogeneous Lempel-Ziv phrases, *i.e.* those that verify the conditions of Lemma 4. The objective of this section is to analyze the complexity of the search for the escaping phrases, *i.e.* those that are not found in the q-grams based search. This search for these homogeneous phrases is described in point **(2)** of Section 5.2..

We need a couple of lemmas that give further insight into the structure of these homogeneous phases. Lemma 4 explained that by restricting the minimal number of errors that a substring of A, of size q, may have we are also restricting the maximal number of errors that it can have. In fact this condition also restricts the spacing between consecutive errors. The next lemmas explain this property.

Lemma 6 *Let A and B be strings and q and s be integers such that $0 \leq s \leq q \leq |A|$ and for any substrings B' of B and A' of A with $|A'| = q$ we have that $ed(A', B') \geq s$. Then for any prefix of A' of A such that for any substring B' of B, $ed(A', B') \geq 1$ we can conclude that $|A'| > |A| - q\lceil(ed(A, B) - 1 + \epsilon)/s\rceil$, for any $\epsilon > 0$.*

Proof Suppose by absurd that there is a prefix A' of A such that for any substring B' of B we have $ed(A', B') \geq 1$ and $|A'| \leq |A| - q\lceil(ed(A, B) - 1 + \epsilon)/s\rceil$ for some $\epsilon > 0$.

Now apply Lemma 2 with $A_1 = A'$, $|A_j| < q$ and the remaining A_i's of size q, *i.e.* $|A_i| = q$ and $j = 2 + \lceil(ed(A, B) - 1 + \epsilon)/s\rceil$. Consider $k_1 = 1$, $k_j = 0$ and the remaining k_i's equal to s. Therefore $\sum_{i=1}^{j} k_i = 1+s(j-2) = 1+s\lceil(ed(A, B)-1+\epsilon)/s\rceil \geq 1+ed(A, B)-1+\epsilon = ed(A, B)+\epsilon > ed(A, B)$.

Therefore we conclude that there is a substring B' of B and an i such that $ed(A_i, B') < k_i$. This, however, is absurd because no value of i may verify this condition. Suppose that $i = j$. Then $ed(A_j, B') < k_j = 0$, which is impossible. Suppose that $1 < i < j$. Then $ed(A_i, B') < k_i = s$ with $|A_i| = q$, which contradicts the hypotheses of the lemma. Finally, it cannot be that $i = 1$ either. That would mean that $ed(A_1, B') = ed(A', B') < k_1 = 1$ for some B' and this contradicts our initial hypotheses that $ed(A', B') \geq 1$ for any B'.

Lemma 7 *Let A and B be strings and q and s be integers such that $0 \leq s \leq q \leq |A|$ and for any substrings B' of B and A' of A with $|A'| = q$ we have that $ed(A', B') \geq s$. Then for any prefix of A' of A for which there is a substring B' of B, $ed(A', B') \leq 1$ we can conclude that $|A'| < q\lceil(1 + \epsilon)/s\rceil$, for any $\epsilon > 0$.*

Proof Suppose by absurd that there is a prefix A' of A for which there is a substring B' of B such that $ed(A', B') \leq 1$ and $|A'| \geq q\lceil(1+\epsilon)/s\rceil$, for some $\epsilon > 0$.

Consider the suffix A'' that results from A by removing A'. Consider also the suffix B'' that results when removing B' from B. Note that B' is not necessarily a prefix of B, but we are only interested in the remaining suffix.

We must have $ed(A, B) \leq ed(A', B') + ed(A'', B'')$, because of the shortest path interpretation. Therefore $ed(A'', B'') \geq ed(A, B) - ed(A', B') \geq ed(A, B) - 1$. The suffix A'' however is too small to contain this number of errors. To see this, apply Lemma 6, replacing 1 by $ed(A, B) - 1$ and using A'' and B''. We conclude that we must have $|A''| > |A| - q\lceil(1+\epsilon)/s\rceil$. On the other hand, by our initial hypotheses, we conclude that $|A''| = |A| - |A'| \leq |A| - q\lceil(1+\epsilon)/s\rceil$. Therefore we have reached an absurd condition and the lemma is proved.

Ukkonen studied the structure of the set of O strings such that $ed(P, O) = k$, denoted as $U_k(P)$ [51], showing that $|U_k(P)| \leq (12/5)(m+1)^k(\sigma+1)^k$. We will now count the number of strings for which the errors are spread homogeneously. We define as $U_{k,q,s}(P)$ the set of O strings such that $ed(P, O) = k$, and for any substring O' of O, with $|O'| = q$, and substring P' of P we have that $ed(O', P') \geq s$.

Lemma 8 $|U_{k,q,s}| \leq (2\sigma+1)^k(2k+1)[q(2+k/s)+k-m-1]^k$.

Proof From the previous lemmas we conclude that, for the strings in $U_{k,q,s}$ the spacing from an error to the next varies from $|O| - q\lceil(k-1+\epsilon)/s\rceil$ to $q\lceil(1+\epsilon)/s\rceil$. Therefore, counting the number of strings in $U_{k,q,s}$ is a matter of choosing these spacings and the type of error to use. The total number of error types is $2\sigma+1$. One σ counts insertions, the other counts substitutions, and the 1 counts the deletions. The number of spacings available is given by the following expression:

$$q\lceil(1+\epsilon)/s\rceil - (|O| - q\lceil(k-1+\epsilon)/s\rceil) - 1 \leq q(2+k/s) - |O| - 1.$$

Therefore, the number of strings in $U_{k,q,s}$ can be bounded as follows:

$$\begin{aligned} |U_{k,q,s}| &\leq (2\sigma+1)^k \sum_{|O|=m-k}^{m+k} [q(2+k/s) - |O| - 1]^k \\ &= (2\sigma+1)^k [q(2+k/s)+k-m-1]^k (2k+1). \end{aligned}$$

This bound is somewhat loose. In fact, the $(2\sigma+1)$ and $[q(2+k/s)+k-m-1]$ never occur together. For example when $[q(2+k/s)+k-m-1]$ occurs, the other factor is 1.

We have therefore shown that, by choosing q and s properly, $U_{k,q,s}$ is much smaller than U_k. Consider for example the optimal case where $ms = qk$. Then $U_{k,q,s} = O((2\sigma+1)^k(k-1+2(m.s/k))^k(2k+1))$. This shows that the dependence on m can be reduced to ms/k, which is considerably smaller, especially when we take $s = 1$. To obtain this optimal result we choose the q and s parameters in the same way as in the hybrid index, by using $q = \lfloor m/h \rfloor$ and $s = \lfloor k/h \rfloor + 1$ for an appropriate parameter h. This parameter plays the role of the j parameter in the classical hybrid index [52].

6. Hierarchical Approximate String Matching

6.1. Bidirectional Compressed Indexes

Our final algorithm can be implemented over any bidirectional index. This means that, from the index point corresponding to a text substring $T[i..j]$ we can efficiently move to that of $T[i..j+1]$ but also to that of $T[i-1..j]$.

Although classical text indexes are not usually bidirectional, the compressed indexes that simulate suffix arrays are. For example, FM-indexes [24, 31] offer a so-called LF mapping operation, which moves from the suffix array position k such that $A[k] = i$, to the position k' such that $A[k'] = i-1$. Compressed suffix arrays [29, 30, 32], instead, offer function ψ, which moves to a k' such that $A[k'] = i+1$, thus ψ is the inverse of LF. Both kinds of compressed indexes can implement the inverse of their basic functionality, and hence are bidirectional (see [53] for the case of LF).

The existing compressed suffix trees [33–35] build complete suffix tree functionality on top of a compressed suffix array, and hence are bidirectional as well. To fix ideas, let us focus in the so-called *fully compressed suffix trees* (FCSTs) [35], which build on top of an FM-index [24, 31] and require $uH_k + o(u \log \sigma)$ bits, although any other combination would essentially fit. The LF mapping of FM-indexes allow FCSTs implement Weiner links [10]: WEINERLINK(v, a), for node v and letter a gives the suffix tree node v' with path-label $a.v[0..]$, and it is the key to move from a v representing $T[i..j]$ to a v' representing $T[i-1..j]$, that is, to bidirectionality. The other direction, that is, from $T[i..j]$ to $T[i..j+1]$, is supported just by moving to a child of v. FCSTs support all of the usual suffix tree navigation operations, including suffix links (via ψ) and lowest common ancestors (LCA(v, v')).

Our algorithm will carry out a hybrid search by backtracking on the compressed suffix tree in order to find the approximate occurrences of pattern pieces, yet it will proceed in a slightly more sophisticated fashion. Instead of extending O' only in one direction, to the right, we will use a bidirectional search. Landau *et al.* [48] obtained the surprising result that it is possible to compute $ed(A, cB)$ from $ed(A, B)$, also in time $O(k)$. The resulting algorithm is very sophisticated and the reader should consult the original paper. For our purposes all we need are the following observations. (1) The extension is not restricted to B, *i.e.* we can also extend $ed(A, B)$ to $ed(cA, B)$. (2) The number of errors does not have to be fixed, *i.e.* we can extend a computation with k errors to a computation on $k+1$ errors in $O(k+1)$ time. (3) Finally, the data structure they use are two doubly linked lists organized in a grid. This means that if we compute $ed(A, cB)$ from $ed(A, B)$ we can revert back to the $ed(A, B)$ state by simply keeping a rollback log of which pointers to revert, which requires $O(k)$ computer words (It should be possible to extend their algorithm to support this directly, but if that is not the case we can still use the rollback log idea). For our algorithm this idea suffices since the states we need to visit are always organized in a stack. Thus we never need to compute a sequence such as $ed(A, B)$ to $ed(A, cB)$ to $ed(dA, cB)$ to $ed(dA, B)$.

To improve the success rate of the process described above we should start our search from an area of P that is well preserved. To limit the number of errors we divide the pattern into small pieces. We will use Lemma 2, taking $A = P$ and $B = O$, and dividing the errors in a homogeneous fashion, *i.e.* choosing $k_i = \alpha|A_i| + \epsilon$, where $\alpha = k/m$ and $\epsilon > 0$ is a number that can be as small as we want and is only used to guarantee that $ed(A, B) < \sum_{i=0}^{j} k_i$. Recall our running example with $O = abbba$ and $P = abc.cba$, assuming this is the partition of A. Therefore we should have $k_0 = k_1 = (2/6) \times 3 + \epsilon$.

Hence the lemma says that in any O there is at least one substring O' such that $ed(O', abc) < 1 + \epsilon$ or $ed(O', cba) < 1 + \epsilon$. In our example there are in fact two substrings O' that satisfy this property, $ed(abb, abc) \leq 1$ and $ed(bba, cba) \leq 1$. This redundancy will make the same string to be found in more than one way. To avoid it, notice that we do not need to add ϵ to both k_i's, *i.e.* we can choose k_0 as before and $k_1 = 1$. This means that the conclusion of the lemma now states that there should be an O' such that $ed(O', abc) \leq 1$ or $ed(O', cba) < 1 \Rightarrow ed(O', cba) \leq 0$, and hence the redundancy is eliminated.

Note that the condition on O' is no guarantee that there exists an occurrence O of P, since it is a one-way implication. Hence the area around O' must be verified to determine whether there is an occurrence or not. Note that in previous work the usual verification procedure is computed on T, not taking advantage of the index. Therefore, verifying those occurrences can cost $O(k(m + k))$ operations. The problem with dividing P too much, such as when $j = k + 1$, is that the number of positions to verify can become excessively large and we get a low success rate, *i.e.* only a small percentage of the O's verified by the process turn out to be occurrences of P.

The hybrid approach tries to maximize the overall success rate by finding an optimal balance between filtration and backtracking. It was shown [52] that the optimal point occurs for $j = \Theta(m/\log_\sigma u)$, with a complicated constant. Our approach can have a slightly different optimal point, but if we use their j the resulting algorithm is never worse than theirs. Moreover we also attempt to automatically determine the optimal point and hence eliminate the need for parameterization.

6.2. *Indexed Hierarchical Verification*

We modify the verification phase, after filtration, in two ways: (1) We will perform it over the FCST instead of over T, to factor out possibly repeated computations. (2) We use hierarchical instead of direct verification, which also provides a strategy to approximate the optimal point.

The idea of hierarchical verification is to gradually extend the error level instead of jumping directly to k. This is obtained by iterating Lemma 2.

This technique was shown to be extremely efficient for the sequential approach [52]. We use the following lemma.

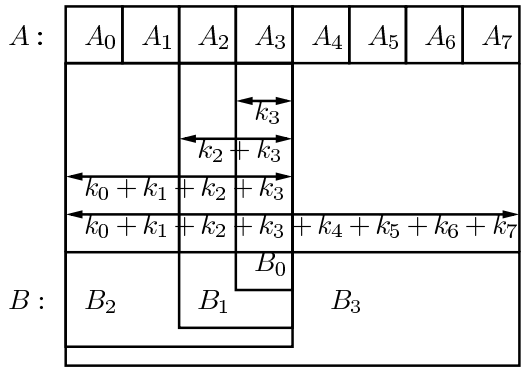
Lemma 9 *Let A and B be strings, $A = A_0.A_1 \ldots A_j$, for strings A_i and some $j+1 = 2^h \geq 1$. Let $k_i \in \mathbb{R}$ such that $ed(A, B) < \sum_{i=0}^{j} k_i$. For some fixed $0 \leq i \leq j$, define $A'_{i'} = A_{2^{i'}\lfloor i/2^{i'} \rfloor} \ldots A_{2^{i'}(1+\lfloor i/2^{i'} \rfloor)-1}$, for any $0 \leq i' \leq h$, as the hierarchical upward path from A_i to A, and define accordingly $k'_{i'} = \sum_{i''=2^{i'}\lfloor i/2^{i'} \rfloor}^{2^{i'}(1+\lfloor i/2^{i'} \rfloor)-1} k_{i''}$ as the error level corresponding to each $A'_{i'}$. Then there are strings $B_0 \sqsubseteq \ldots \sqsubseteq B_h = B$ and an i such that for any $0 \leq i' \leq h$ we have $ed(A'_{i'}, B_{i'}) < k'_{i'}$. Moreover, for each i', if $A'_{i'}$ is a prefix(suffix) of $A'_{i'+1}$ then $B_{i'}$ is a prefix(suffix) of $B_{i'+1}$.*

Proof For $i' = h$ the lemma is trivial because $ed(A'_h, B_h) < k'_h$ reduces to $ed(A, B) < \sum_{i=1}^{j} k_i$. To obtain the result for $i' = h - 1$, consider the partition of A into two strings, $A^{left} = A_0 \ldots A_{2^{i'}-1}$ and $A^{right} = A_{2^{i'}} \ldots A_{2^{i'+1}-1}$ (depending on i, either $A'_{h-1} = A^{left}$ or $A'_{h-1} = A^{right}$), and its edit distance matrix to B achieving edit distance $ed(A, B) < \sum_{i=1}^{j} k_i = k^{left} + k^{right}$, for $k^{left} = \sum_{i''=0}^{2^{i'}-1} k_{i''}$ and $k^{right} = \sum_{i''=2^{i'}}^{2^{i'+1}-1} k_{i''}$. Then there must be a partition $B = B^{left}.B^{right}$ such that $ed(A^{left}, B^{left}) < k^{left}$ or $ed(A^{right}, B^{right}) < k^{right}$. Thus $B_{h-1} = B^{left}$ if $A'_{h-1} = A^{left}$ (note $k_{h-1} = k^{left}$ in this case), and

otherwise $B_{h-1} = B^{right}$. To obtain the result for $i' = h - 2$ we repeat the process from A'_{h-1}, B_{h-1}, and k_{h-1}.

Figure 7 gives a schematic representation of Lemma 9.

Figure 7. A schematic representation of Lemma 9.



Consider our running example with $k = 2$ and $P = abccba$. Instead of applying Lemma 9 we will instead iterate Lemma 2, which is actually the way we compute the partition in practice. We divide $P = A = abc.cba$ into pieces of size 3 and therefore we have $k'_0 = k'_1 = 3\times(2/6)+\epsilon = 1+\epsilon$, which in practice means 1 error per piece. Now we divide these pieces as $ab.c.cb.a$ and we have $k_0 = k_2 = 2 \times (2/6) + \epsilon$ and $k_1 = k_3 = 1 \times (2/6) + \epsilon$, which means 0 errors for all the pieces. Notice that we can refine our method by adding ϵ to only one k_i, as we did in Section 2.2.. Hence we can choose $k_0 = k_2 = 2/3$ and $k_1 = 1/3 + \epsilon$ and $k_3 = 1/3$ (and thus $k'_0 = 1 + \epsilon$ and $k'_1 = 1$). Notice that in our example the occurrence $abbba$ verifies this Lemma because $ed(ab, ab) < 2/3$ and $ed(abb, abc) < (2/3) + (1/3) + \epsilon$, where ab and abc are substrings of P.

This lemma is used to reduce the cost of verifying an occurrence. Instead of directly verifying the space around a B_0 when $ed(A_i, B_0) < k_i$ for a string B such that $ed(A, B) < k$, we extend the error level gradually. Assuming i is even, this means checking for $ed(A_i.A_{i+1}, B_1) < k_i + k_{i+1}$ first, for some B_1. Figure 2 (right) shows an example of this process, computed with table D. Whenever a row reaches a certain level in the hierarchy and contains active cells, the computation on that row is extended to activate the cells that are $< k_i + k_{i+1}$. For example since $D[2, 2] = 0$ the cells in row 2 that can be $< 1 + \epsilon$ are activated, *i.e.* cells $D[1, 2]$ and $D[3, 2]$, that correspond to $ed(a, ab)$ and $ed(abc, ab)$. A similar process happens at row 3. In theory we can compute all the cells that have a value equal or smaller than k all the time. Still, we can also start to compute them at a given row, especially since it is not necessary to fill upwards the missing cells in the table. That is, we can compute the missing cells, up to $k_i + k_{i+1}$, from the ones already in the table. There is no problem if the value of the new cells is larger than their value on the complete D table. In fact it is desirable. This will only make the algorithm skip occurrences that, because of Lemma 9, will be found in another case.

To determine that $ed(A_i, B_0) < k_i$ we must compute the D table for these two strings. Extending this computation to $ed(A_i.A_{i+1}, B_1) < k_i + k_{i+1}$ is simple because table D only needs to be updated in its natural directions (to the right and downwards). From the suffix tree point of view this situation is also natural because it involves descending in the tree.

When i is odd the situation is a bit trickier. This time we must check for $ed(A_{i-1}.A_i, B_1) < k_{i-1} + k_i$. This is much more difficult because we need to move in the FCST by prepending letters to the current point. This is possible with the WEINERLINK operation (recall Section 6.1.). Moreover we need to extend the DP in unnatural directions (to the left and upwards). For this we use the result [48] mentioned also in Section 6.1.. Hence computing each new row requires only $O(k)$ operations. Note that the underlying operation on which their algorithm relies is the longest common prefix of any two suffixes of A and B. To solve this we build a FCST for P, in $O(m)$ time, in uncompressed format so that the LCA operation takes $O(1)$ time. Note that this FCST is built only once at the beginning of the algorithm and adds $O(m \log m)$ bits to the space requirements of the algorithm. We determine the positions of $O'[i..]$ in that suffix tree, in $O(m')$ time, with the PARENT and WEINERLINK operations. Together with the LCA operation we can compute the size of the necessary longest common prefixes. Note that whenever O' is extended to/contracted from cO', this information must be updated, by recomputing in $O(m')$ time.

Our algorithm consists in backtracking, where the error bound is gradually increased. Depending on the position of current P's substring in the hierarchical verification, the string O' is extended either to the left or to the right. Hence, as mentioned before, the $ed(P, O')$ states are stored in a stack, whereas the O' string being generated is stored in a double stack structure that can be pushed/popped at both ends.

7. Practical Issues and Testing

In this section we present the experimental performance of the algorithms we described in the previous sections. As a baseline we used efficient sequential bit-parallel algorithms (namely BPM, the bit-parallel DP matrix of Myers [54], and EXP, the exact pattern partitioning by Navarro and Baeza-Yates [55]). We also used a classical uncompressed hybrid index [52].

The algorithms of Section 3. lead to implementations LZI (the basic one, described in Section 3.1.) and DLZI (the advanced one, described in Section 3.2.), as well as FMIndex, described in Section 3.3.. These structures work over the LZ-Index and FM-Index implemented by Navarro [50], where the second structure is a larger and faster variant of the usual FM-index implementations. Recall that these are pure filtration indexes, using Lemma 1 with $A = P$ and $B = O$, as described in the beginning of Section 2.3..

The algorithm explained in Sections 4. and 5. is the prototype called ILZI and it is based on the same ILZI compressed index [28]. For this prototype we used a stricter backtracking than what was explained in previous sections. For each pattern substring $P[y_1, y_2]$ to be matched, we computed the maximum number of errors that could occur when matching it in the text, also depending on the position $O[x_1, x_2]$ where it would be matched, and maximizing over the possible areas of O where the search would be necessary. For example, the extremes of P can be matched with fewer errors than the middle. This process involves precomputing tables that depend on m and k. We omit the technical details. Note that we could use the ILZI to implement the exact partitioning algorithms run over the LZI and DLZI, yet the ILZI allows for a much stronger search, and we found it more interesting to stress this fact in the experiments.

We also implemented a prototype, BiFMI, to test the algorithm described in Section 6.. We replaced the FCST with a bidirectional FM-Index over one wavelet tree [21]. The wavelet tree is a data structure that allows one to compute LF in $O(\log \sigma)$ time. It contains σ bitmaps which add up to $n \log \sigma$ bits (that is, the same space of the original text), on which $rank$ and $select$ operations are carried out to

compute LF and its inverse. We compress these bitmaps using a technique [56] that solves $rank/select$ in constant time and, when applied in the context of an FM-index, achieves nH_k bits of space [57]. We reverse the search so that the most common search (forwards) is done using LF (where the FM-index is faster) instead of ψ. Replacing a suffix tree (FCST) by a compressed suffix array (BiFMI) was possible because we did not use the SLINK operation of the FCST.

Note that, since the BiFMI prototype is an FM-Index, it could also have been used to implement the algorithm described in Section 3.3., instead of the FMIndex structure (The converse is not true, as that FMIndex is intrinsically non-bidirectional. The formula for the inverse of LF [53] requires an operation called *select* that can be carried out efficiently on wavelet-tree-based FM-indexes, but not on this implementation). Yet, the result is in all cases inferior to the hybrid indexing on the BiFMI. On the other hand, the BiFMI cannot reach the speed of the FMIndex for unidirectional search even if using the same amount of space. Thus we found it more interesting to implement the exact partitioning method over the larger and faster FMIndex, to show a different tradeoff.

We tested our algorithm, BiFMI, in *automatic* mode, *i.e.* Lemma 9 is used until every $k_i < 1$. We did not implement the algorithm of Landau *et al.* [48]. Instead we used the bit-parallel NFA of Wu *et al.* [58] and recomputed the D table whenever it was necessary to change the computing direction. Note this requires $O(m)$ time when we switch from right to left or vice versa, but after the change it will require only $O(k)$ time for each new row. Although in theory this process could slow down our algorithm by a factor of $O(\log k)$, in practice this factor was negligible.

We used the texts from the Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl`), with 50 MB of English and DNA and 64 MB of proteins. The machine was a Pentium 4, 3.2 GHz, 1 MB L2 cache, 1 GB RAM, running Fedora Core 3, and compiling with `gcc-3.4 -O9`. The pattern strings were sampled randomly from the text and each character was distorted with 10% of probability. All the patterns had length $m = 30$. Every configuration was tested during at least 60 seconds using at least 5 repetitions. Hence the numbers of repetitions varied between 5 and 130,000. To parametrize the hybrid index we tested all the j values from 1 to $k + 1$ and reported the best time. To parametrize we choose $q = \lfloor m/h \rfloor$ and $s = \lfloor k/h \rfloor + 1$ for some convenient h, since we can prove that this is the best approach and it was corroborated by our experiments. To determine the value of h and v we also tested the viable configurations and reported the best results. In our examples choosing v and h such that $2v$ is slightly smaller than q yielded the best configuration. Figure 9 shows the sensitivity of the ILZI to this parameter. The LZI and DLZI are not parametrized.

The average query time, in seconds, is shown in Figure 8, and the respective memory heap peaks for indexed approaches are shown in Table 1. The hybrid index provides the fastest approach to the problem. However it also requires the most space. Aside from the hybrid index the ILZI is always either the fastest or within reasonable distance from the fastest approach. For low error level, $k = 1$ or $k = 2$, the ILZI is significantly faster, up to an order of magnitude better. This is very important since this is the area where indexed ASM has chances to make a difference. The sequential approaches outperform all compressed indexing approaches for sufficiently high error levels. In DNA this occurs at $k = 4$ and in English at $k = 5$.

Figure 8. Average user time for finding the occurrences of patterns of size 30 with k errors in DNA, English, and Proteins. The y axis units are in seconds.

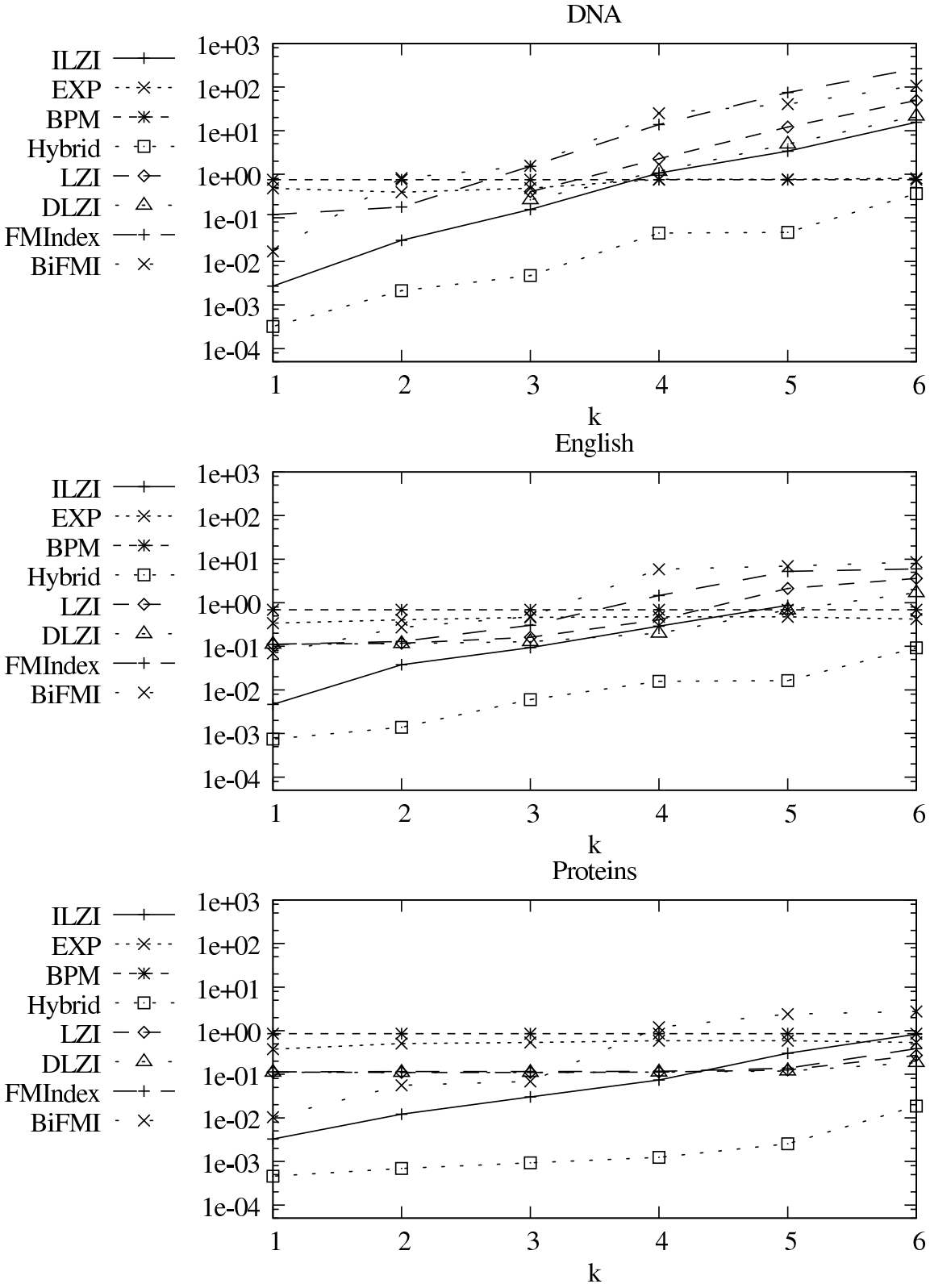


Figure 9. Average user time, in seconds, that the ILZI takes to find occurrences of patterns of size 30 with k errors, using different v's.

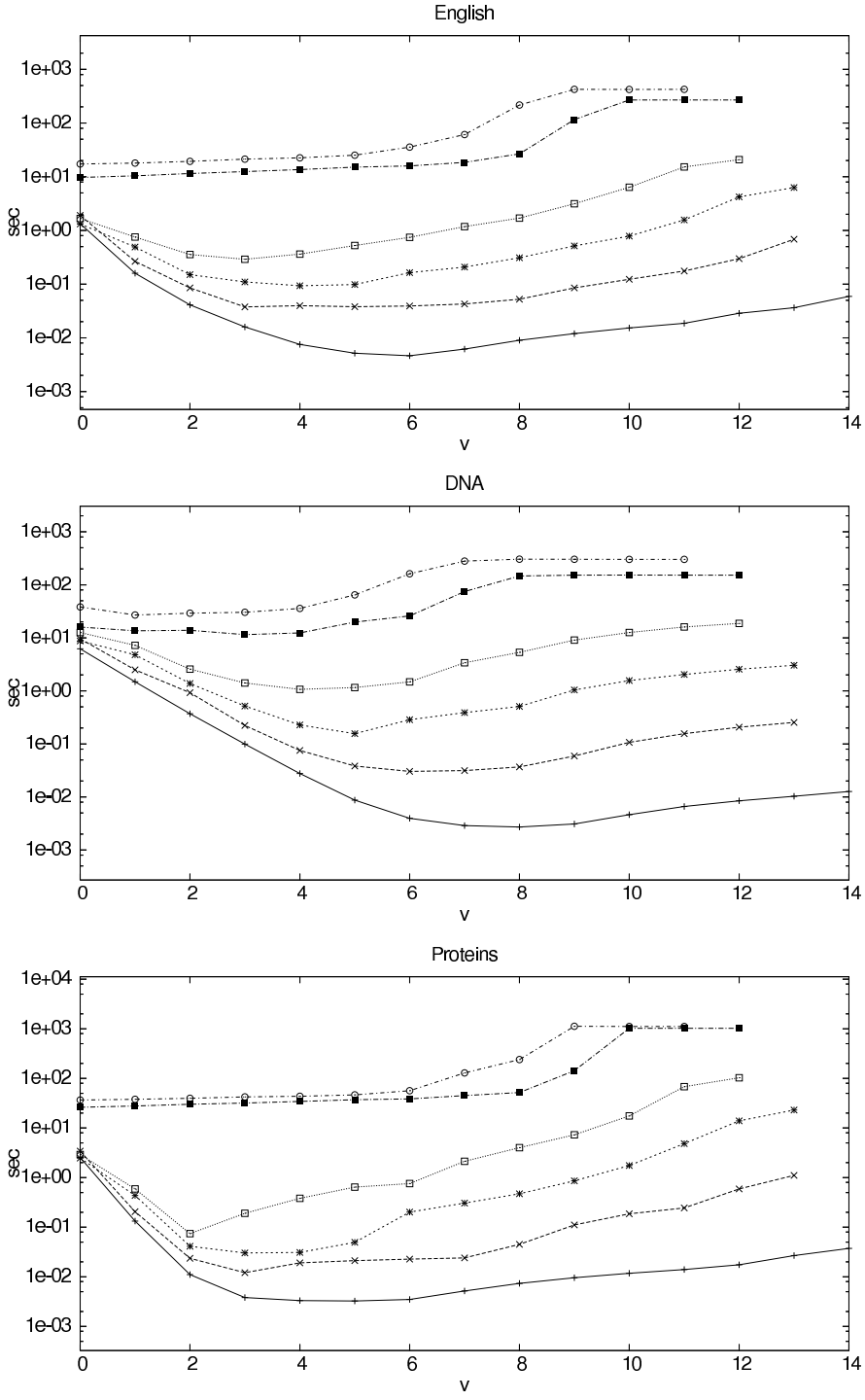


Table 1. Memory peaks, in Megabytes, for the different approaches when $k = 6$.

	ILZI	Hybrid	LZI	DLZI	FMIndex	BiFMI
English	55	257	145	178	131	54
DNA	45	252	125	158	127	40
Proteins	105	366	217	228	165	63

The ILZI performed particularly well on proteins, as did the hybrid index. This could be due to the fact that proteins behave closer to random text, and this means that the parametrization of ours and the hybrid index indeed balances between exponential worst cases.

In terms of space the ILZI is also very competitive, as it occupies almost the same space as the plain text, except for proteins. We presented the space that the algorithms need to operate and not just the index size, since the other approaches build intermediate data structures at search time.

Our BiFMI index, on the other hand, achieves the smallest space. We maintain a sparse sampling for our prototype, to show that even within little space we can achieve competitive performance. The FMIndex, on the other hand, needs a much denser sampling to be competitive. Thus our hierarchical and bidirectional verification method is faster than the basic one, even if run on a much slower index.

The performance of the BiFMI prototype closely follows that of ILZI, except for the DNA file. This indicates that it was able to approach hybrid performance in minimal space. It is also, mostly, capable of reducing the gap caused by cache misses.

Among the exact partitioning indexes, the DLZI obtained the best performance. Although it is in some cases the fastest when we exclude Hybrid, in general the compressed hybrid approaches achieved the same or better performance using much less space.

8. Conclusions and Future Work

In this paper we presented two algorithms for ASM in compressed space: an adaptation of the hybrid index for Lempel-Ziv compressed indexes and an hierarchical verification over FCST's.

We started by addressing the problem of approximate matching with q-samples indexes, where we described a new approach to this problem. We then adapted our algorithm to the irregular parsing produced by Lempel-Ziv indexes. Our approach was flexible enough to be used as a hybrid index instead of an exact-searching-based filtration index. We implemented our algorithm and compared it with the simple filtration approach built over different compressed indexes, with sequential algorithms, and with a good uncompressed index.

Our results show that our index provides a good space-time tradeoff, using a small amount of space (at best 0.9 times the text size, which is 5.6 times less than a classical index) in exchange for searching from 6.2 to 33 times slower than a classical index, for $k = 1$ to 3. This is better than the other compressed approaches for low error levels. This is significant since indexed approaches are most valuable, if compared to sequential approaches, when the error level is low. Therefore our work significantly improves the usability of compressed indexes for approximate matching. A crucial part of our work was our approach for the prefixes/suffixes of O. This approach is in fact not essential for q-samples indexes, but it can improve previous approaches [19]. For a Lempel-Ziv index it is indeed essential.

Our ILZI implementation can be further improved since we do no secondary filtering, that is, we do not apply any sequential filter over the text context before fully verifying them. We also plan to further explore the idea of associating weights to the letters of O. We will investigate the impact of assigning smaller weights to less frequent letters of O. This should decrease the number of positions to verify and improve the overall performance.

We also studied the impact of hierarchical verification in ASM. We obtained an automatic hybrid index that uses fully-compressed suffix trees. This a very important result because it is the first algorithm that approximates the performance of the hybrid index automatically and effectively in practice. Our result is also very important because FCSTs require only compressed space, *i.e.* $uH_k + O(n \log \sigma)$ bits. Compared with other compressed indexes, our approach was more efficient for low error levels. Although it was less efficient than the ILZI-based algorithm, it requires less space in theory and in practice. In theory, the ILZI requires $5uH_k + o(n \log \sigma)$ bits, but in practice that is closer to $3uH_k$, including the sublinear term. On the other hand, a FCST requires $uH_k + o(n \log \sigma)$ bits in theory, but this becomes a bit higher in practice if we consider the sublinear term. Moreover our algorithm can be used as a subroutine in a suffix tree algorithm whereas the ILZI-based algorithm cannot.

Acknowledgements

First and third authors partially funded by the Portuguese Science and Technology Foundation by project ARN: Algorithms for the Identification of Genetic Regulatory Networks, PTDC/EIA/67722/2006. Second author partially supported by Fondecyt Grant 1-080019.

References and Notes

1. Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv.* **2001**, *33*, 31-88.
2. Chang, W.; Marr, T. Approximate string matching and local similarity. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Asilomar, CA, USA, June 5-8, 1994; pp. 259-273.
3. Fredriksson, K.; Navarro, G. Average-optimal single and multiple approximate string matching. *ACM J. Exp. Algorithmics* **2004**, *9*, No. 1.4.
4. Navarro, G.; Baeza-Yates, R.; Sutinen, E.; Tarhio, J. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.* **2001**, *24*, 19-27.
5. Sung, W.K. *Indexed approximate string matching*; Springer: Berlin, Germany, 2008; pp. 408-411.
6. Cole, R.; Gottlieb, L.A.; Lewenstein, M. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC)*, Chicago, IL, USA, June 13-16, 2004; pp. 91-100.
7. Maaß, M.; Nowak, J. Text indexing with errors. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Jeju Island, Korea, June 19-22, 2005; pp. 21-32.
8. Chan, H.L.; Lam, T.W.; Sung, W.K.; Tam, S.L.; Wong, S.S. A linear size index for approximate pattern matching. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Barcelona, Spain, July 5-7, 2006; pp. 49-59.
9. Coelho, L.; Oliveira, A. Dotted suffix trees: a structure for approximate text indexing. In *Proceed-*

ings of the 13th International Symposium on String Processing and Information Retrieval (SPIRE), Glasgow, UK, October 11-13, 2006; pp. 329-336.

10. Weiner, P. Linear pattern matching algorithms. In *14th IEEE Annual Symposium on Switching and Automata Theory*, Iowa City, USA, October 15-17, 1973; pp. 1-11.
11. Manber, U.; Myers, E. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* **1993**, *22*, 935-948.
12. Gonnet, G. *A tutorial introduction to Computational Biochemistry using Darwin*; Technical report; Informatik E.T.H.: Zuerich, Switzerland, 1992.
13. Ukkonen, E. Approximate string matching over suffix trees. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Asilomar, CA, USA, June 5-8, 1994; pp. 228-242.
14. Cobbs, A. Fast approximate matching using suffix trees. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Espoo, Finland, July 5-7, 1995; pp. 41-54.
15. Sutinen, E.; Tarhio, J. Filtration with q-samples in approximate string matching. In *Proceeding of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Laguna Beach, CA, USA, June 10-12, 1996; pp. 50-63.
16. Navarro, G.; Baeza-Yates, R. A practical *q*-gram index for text retrieval allowing errors. *CLEI Electron. J.* **1998**, *1*, No. 2.
17. Myers, E.W. A sublinear algorithm for approximate keyword searching. *Algorithmica* **1994**, *12*, 345-374.
18. Navarro, G.; Baeza-Yates, R. A hybrid indexing method for approximate string matching. *J. Discrete Algorithms* **2000**, *1*, 205-239.
19. Navarro, G.; Sutinen, E.; Tarhio, J. Indexing text with approximate *q*-grams. *J. Discrete Algorithms* **2005**, *3*, 157-175.
20. Kurtz, S. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.* **1999**, *29*, 1149-1171.
21. Navarro, G.; Mäkinen, V. Compressed full-text indexes. *ACM Comput. Surv.* **2007**, *39*, No. 2.
22. Manzini, G. An analysis of the Burrows-Wheeler transform. *JACM* **2001**, *48*, 407-430.
23. Ziv, J.; Lempel, A. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530-536.
24. Ferragina, P.; Manzini, G. Indexing compressed text. *JACM* **2005**, *52*, 552-581.
25. Navarro, G. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms* **2004**, *2*, 87-114.
26. Kärkkäinen, J.; Ukkonen, E. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proceedings of the 3rd South American Workshop on String Processing (WSP)*, Recife, Brazil, August 08-09, 1996; pp. 141-155.
27. Arroyuelo, D.; Navarro, G.; Sadakane, K. Reducing the space requirement of LZ-Index. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Barcelona, Spain, July 5-7, 2006; pp. 318-329.
28. Russo, L.M.S.; Oliveira, A.L. A compressed self-index using a Ziv-Lempel dictionary. *Inf. Retr.* **2008**, *11*, 359-388.
29. Sadakane, K. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms* **2003**, *48*, 294-313.

30. Grossi, R.; Gupta, A.; Vitter, J. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Baltimore, MD, USA, January 12-14 2003; pp. 841-850.
31. Ferragina, P.; Manzini, G.; Mäkinen, V.; Navarro, G. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms* **2007**, *3*, No. 20.
32. Grossi, R.; Vitter, J. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.* **2005**, *35*, 378-407.
33. Sadakane, K. Compressed suffix trees with full functionality. *Theory Comput. Syst.* **2007**, *41*, 589-607.
34. Fischer, J.; Mäkinen, V.; Navarro, G. An(other) entropy-bounded compressed suffix tree. In *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Pisa, Italy, June 18-20, 2008; pp. 152-165.
35. Russo, L.; Navarro, G.; Oliveira, A. Fully-compressed suffix trees. In *Proceedings of the 8th Latin American Symposium on Theoretical Informatics (LATIN)*, Búzios, Brazil, April 7-11, 2008; pp. 362-373.
36. Ferragina, P.; González, R.; Navarro, G.; Venturini, R. Compressed text indexes: From theory to practice. *ACM J. Exp. Algorithmics (JEA)* **2009**, *13*, No. 12.
37. Huynh, T.; Hon, W.K.; Lam, T.W.; Sung, W.K. Approximate string matching using compressed suffix arrays. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Jeju Island, Korea, June 19-22, 2005; pp. 434-444.
38. Lam, T.W.; Sung, W.K.; Wong, S.S. Improved approximate string matching using compressed suffix data structures. In *Proceedings of the 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, Hainan, China, December 19-21, 2005; pp. 339-348.
39. Navarro, G.; Baeza-Yates, R. Improving an algorithm for approximate pattern matching. *Algorithmica* **2001**, *30*, 473-502.
40. Russo, L.M.S.; Navarro, G.; Oliveira, A.L. Approximate string matching with Lempel-Ziv compressed indexes. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, Santiago, Chile, October 29-31, 2007; pp. 264-275.
41. Russo, L.M.S.; Navarro, G.; Oliveira, A.L. Indexed hierarchical approximate string matching. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, Melbourne, Australia, November 10-12, 2008; pp. 144-154.
42. Apostolico, A. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*; Springer-Verlag: New York, NY, USA, 1985; pp. 85-96.
43. Gusfield, D. *Algorithms on Strings, Trees and Sequences*; Cambridge University Press, Cambridge, UK, 1997.
44. Lempel, A.; Ziv, J. On the complexity of finite sequences. *IEEE Trans. Inf. Theory* **1976**, *22*, 75-81.
45. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337-343.
46. Welch, T. A technique for high performance data compression. *IEEE Comput. Mag.* **1984**, *17*, 8-19.
47. Ukkonen, E. Finding approximate patterns in strings. *J. Algorithms* **1985**, *6*, 132-137.

48. Landau, G.M.; Myers, E.W.; Schmidt, J.P. Incremental string comparison. *SIAM J. Comput.* **1998**, *27*, 557-582.
49. Maaß, M. Linear bidirectional on-line construction of affix trees. *Algorithmica* **2003**, *37*, 43-74.
50. Navarro, G. Implementing the lz-index: Theory versus practice. *ACM J. Exp. Algorithmics* **2009**, *13*, No. 2.
51. Ukkonen, E. Finding approximate patterns in strings. *J. Algorithms* **1985**, *6*, 132-137.
52. Navarro, G.; Baeza-Yates, R. A hybrid indexing method for approximate string matching. *J. Discrete Algorithms* **2000**, *1*, 205-239.
53. Lee, S.; Park, K. Dynamic rank-select structures with applications to run-length encoded texts. In *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Pisa, Italy, June 18-20, 2008; pp. 95-106.
54. Myers, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *JACM* **1999**, *46*, 395-415.
55. Navarro, G.; Baeza-Yates, R. Very fast and simple approximate string matching. *Inf. Proc. Lett.* **1999**, *72*, 65-70.
56. Raman, R.; Raman, V.; Rao, S.S. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms*, San Francisco, CA, USA, January 6-8, 2002; pp. 233-242.
57. Mäkinen, V.; Navarro, G. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms* **2008**, *4*, 32:1-32:38.
58. Wu, S.; Manber, U. Fast text searching allowing errors. *Commun. ACM* **1992**, *35*, 83-91.

© 2009 by the authors; licensee Molecular Diversity Preservation International, Basel, Switzerland. This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/).