# Approximate String Matching with Lempel-Ziv Compressed Indexes

Luís M. S. Russo ⋆⋆⋆[1], Gonzalo Navarro†[2] and Arlindo L. Oliveira[1]

[1] INESC-ID, R. Alves Redol 9, 1000 LISBOA, PORTUGAL
lsr@algos.inesc-id.pt, aml@inesc-id.pt
[2] Dept. of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

**Abstract.** A compressed full-text self-index for a text $T$ is a data structure requiring reduced space and able of searching for patterns $P$ in $T$. Furthermore, the structure can reproduce any substring of $T$, thus it actually replaces $T$. Despite the explosion of interest on self-indexes in recent years, there has not been much progress on search functionalities beyond the basic exact search. In this paper we focus on indexed approximate string matching (ASM), which is of great interest, say, in computational biology applications. We present an ASM algorithm that works on top of a Lempel-Ziv self-index. We consider the so-called hybrid indexes, which are the best in practice for this problem. 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 the Lempel-Ziv index. We show experimentally that our algorithm has a competitive performance and provides a useful space-time tradeoff compared to classical indexes.

## 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. It consists in locating all the occurrences of a given pattern string $P[0, m-1]$ in a larger text string $T[0, u-1]$, letting the occurrences be at distance $ed()$ at most $k$ from $P$. In this paper we focus on 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 time $O(u(k + \log_\sigma m)/m)$ [2, 3], where $\sigma$ is the size of the alphabet $\Sigma$. 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 holds they verify the text area using a

⋆⋆⋆ Supported by the Portuguese Science and Technology Foundation by grant SFRH/BD/12101/2003 and Project DBYEAST POSC/EIA/57398/2004.
† Supported in part by Fondecyt Grant 1-050493 (Chile).

classical ASM algorithm. For long texts, however, sequential searching might be impractical because it must scan all the text. To avoid this we use an *index* [4].

There exist indexes specifically devoted to ASM, e.g. [5–8], but these are oriented to worst-case performance. There seems to exist an unbreakable space-time barrier with indexed ASM: Either one obtains exponential times (on $m$ or $k$), or one obtains exponential index space (e.g. $O(u \log^k u)$). Another trend is to reuse an index designed for exact searching, all of which are linear-space, and try to do ASM over it. Indexes such as suffix trees [9], suffix arrays [10], or based on so-called $q$-grams or $q$-samples, have been used. There exist several algorithms, based on suffix trees or arrays, which focus on worst-case performance [11–13]. Given the mentioned time-space barrier, they achieve a search time independent of $u$ but exponential on $m$ or $k$. Essentially, they simulate the sequential search over all the possible text suffixes, taking advantage of the fact that similar substrings are factored out in suffix trees or arrays.

Indexes based on $q$-grams (indexing all text substrings of length $q$) or $q$-samples (indexing non-overlapping text substrings of length $q$) 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 an on-line 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. [14, 15], cease to be useful for moderate $k$ values, which are still of interest in applications.

The most successful approach, in practice, is in between the two techniques described above, and is called "hybrid" indexing. The index determines the text positions to verify using an approximate-matching condition instead of an exact one. This requires a search of the first kind (whose time is exponential on the length of the string or the number of errors). Yet, these searches are done over short strings 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 [16], suffix arrays [17], and $q$-sample indexes [18].

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 [19]. 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 [20–24]. 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, gives 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, $H_k$ [25], which gives a lower bound on the number of bits per symbol achievable by a $k$-th order compressor. In this work we are interested in indexes based on Lempel-Ziv compression [21, 22, 26–28].

Despite the great success of self-indexes, they have been mainly used for exact searching. Only very recently some indexes taking $O(u)$ or $O(u\sqrt{\log u})$ *bits* have appeared [29, 30, 7]. Yet, those are again of the worst-case type, and thus all their times are exponential on $k$. In this paper we present a practical algorithm that runs on a compressed self-index and belongs to the most successful class of hybrid algorithms.

## 2 Our Contribution in Context

One can easily 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. Indeed, this has been already attempted [31] using the FM-index [21] and a Lempel-Ziv index [22]. The Lempel-Ziv index worked better 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 allowed several interesting optimizations (such as factoring out the work of several text extractions) that we will not discuss further here.

Lempel-Ziv indexes split the text into a sequence of so-called *phrases* of varying length. They are efficient to find the (exact) occurrences that lie within phrases, but those that span two or more phrases are costlier.

Our goal in this paper is to provide efficient approximate searching over a small and practical self-index. Based on the described previous experiences, (1) we want an algorithm of the hybrid type, which implies that the self-index should do approximate search for pattern pieces; (2) we want a Lempel-Ziv-based index, so that the extraction of text to verify is fast; (3) we wish to avoid the problems derived from pieces spanning several Lempel-Ziv phrases. We will focus on an index [28] whose suffix-tree-like structure is useful for this approximate searching.

Mimicking $q$-sample indexes is particularly useful for our goals. Consider that the text is partitioned into contiguous $q$-samples. 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. The following lemma, simplified from [4], gives the connection to use approximate searching for pattern substrings with a $q$-samples index [18].

**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 strings $A_i$ and for any $j \geq 1$. Then there is a substring $B'$ of $B$ and an $i$ such that $ed(B', A_i) \leq \lfloor k/j \rfloor$.*

Therefore, if we assume that $B = P$ and $A$ is contained in $O$, we index all the different text $q$-samples into, say, a trie data structure. Then the trie is traversed to find $q$-samples that match within $P$ with at most $\lfloor k/j \rfloor$ errors. All the contexts around all occurrences of the matching $q$-samples are examined for full occurrences of $P$. Note in passing that we could also take $A = P$ and $B$ contained in $O$, in which case we choose how to partition $P$ but we must be able to find any text substring with the index (exactly [15] or approximately [16, 17], depending on $j$). Thus we must use a suffix tree or array [17], or even a $q$-gram index if we never use pieces of $P$ longer than $q$ [15, 16].

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 [18] to an irregular parsing (thus we must stick to the interpretation $B = P$). 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 could 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.

We implement our scheme and compare it with the best technique in practice over classical indexes [17], and with the previous developments over compressed self-indexes [31]. The experiments show that our technique provides a relevant space-time tradeoff for indexed ASM.

## 3 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. Remind that a $q$-samples index stores the locations, in $T$, of all the substrings $T[qi..qi + q - 1]$.

### 3.1 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 (all proofs are in the Appendix for lack of space).

**Lemma 2.** *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 $k_i \in \mathbb{R}$ such that $\sum_{i=1}^{j} k_i > ed(A, B)$. Then there is a substring $B'$ of $B$ and an $i$ such that $ed(A_i, B') < k_i$.*

Lemma 1 is a particular case of Lemma 2: set $k_i = k/j + \epsilon$ for sufficiently small $\epsilon > 0$. Our lemma reminds Lemma 2 of [4], and they can be proved to be equivalent. The current formulation is more advantageous for us because one does not need to know $j$. It can be used to adapt the error levels to the length of the pieces. For example, to try to maintain a constant error level, take $k_i = (1+\epsilon)\ k \cdot |A_i|/|A|$ with $\epsilon > 0$.

### 3.2 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 $k_i = (1+\epsilon)\ k \cdot |A_i|/|A|$ errors for $A_i$, for any $\epsilon > 0$. Conservatively, this is $k_i = (1+\epsilon)\ k \cdot q/(m-k)$ 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 $(1+\epsilon)\ k \cdot q/(m-k)$, 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 $|A_i|_v = (1+\epsilon)\ |A_i|/(|A|-2v)$.

We can now apply Lemma 2 with $k_i = k \cdot |A_i|_v$ provided that $k > 0$. Note that $\sum_{i=1}^{j} k_i = (1+\epsilon)\ k > k$. In this case, if $|A_1| \le v$ 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 occurrence $O = A$, 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.

### 3.3 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 classify 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) (that is, $|O_j|_v = |O_1|_v$). We assume that $q < m - k$ and therefore 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 dynamic programming matrix between every different $q$-sample and any substring of $P$ (see [4]). 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).

## 4 Using a Lempel-Ziv Self-Index

We now adapt our technique to the irregular parsing of phrases produced by a Lempel-Ziv-based index. Among the several alternatives [21, 22, 26–28], we will focus on the ILZI [28], yet the results can be carried over to similar indexes.

The ILZI partitions the text into phrases such that every suffix of a phrase is also a phrase (similarly to LZ78 compressors [32], 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. We do not need more details for this paper.

### 4.1 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.

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 certain structure that makes them easy to find.

**Lemma 3.** *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 $ed(A', B') \leq ed(A, B) - s\lfloor (|A| - |A'|)/q \rfloor$.*

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.

Parameter $s$ permits us balancing 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.

### 4.2 A Hybrid Lempel-Ziv Index

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

**Lemma 4.** *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 $ed(A', B'') < k \cdot |A_i|_v - s\lfloor (|A_i| - |A'|)/q \rfloor$.*

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 4 gives the key to the first phase. We find the relevant phrases via two searches:

**(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 dynamic programming 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 4.

**(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 4, where the equations vary according to the roles described in Section 3.3 (that is, 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 $k \cdot |A_i|_v$ formula (both for the matching condition and the backtracking limit) can be bounded by $(1+\epsilon)\ k \cdot \min(|A_i|/(m-k-2v), 1)$, 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 $P[y_1..]$.

**(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 $(1 + \epsilon)\ k \cdot \min((d - v)/(m - k - 2v), 1)$, 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 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 Practical Issues and Testing

We implemented a prototype to test our algorithm on the ILZI compressed index [28]. As a baseline we used efficient sequential bit-parallel algorithms (namely

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

| | ILZI | Hybrid | LZI | DLZI | FMIndex |
|---|---|---|---|---|---|
| English | 55 | 257 | 145 | 178 | 131 |
| DNA | 45 | 252 | 125 | 158 | 127 |
| Proteins | 105 | 366 | 217 | 228 | 165 |

BPM, the bit-parallel dynamic programming matrix of Myers [33], and EXP, the exact pattern partitioning by Navarro and Baeza-Yates [34]).

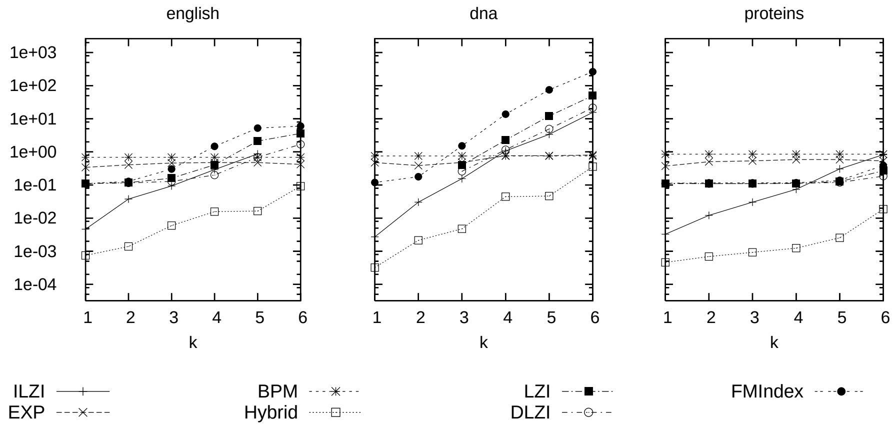
For the real prototype we used a stricter backtracking than as explained in previous sections. For each pattern substring $P[y_1, y_2]$ to be matched, we computed the maximum number of errors that could occur when matching it in the text, also depending on the position $O[x_1, x_2]$ where it would be 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 details for lack of space.

We also included in the comparison an implementation of a filtration index using the simple approach of Lemma 1 with $A = P$ and $B = O$, as briefly described in the beginning of Section 2 [31]. The indexes used in that implementation are the LZ-index [22] (LZI) and Navarro's implementation of the FM-index [21]. We also compare an improved variant over the LZ-index (DLZI [31]). Note that the FM-Index does not divide the text into blocks, however it takes longer to locate occurrences.

The machine was a Pentium 4, 3.2 GHz, 1 MB L2 cache, 1GB RAM, running Fedora Core 3, and compiling with `gcc-3.4 -O9`. 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 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.

The average query time, in seconds, is shown in Fig. 1 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 our approach is always either the fastest or within reasonable distance from the fastest approach. For low error level, $k = 1$ or $k = 2$, our approach is significantly faster, up to an order of magnitude better. This is very important since the compressed approaches seem to saturate at a given performance for low error levels: in English $k = 1$ to 3, in

**Fig. 1.** Average user time for finding the occurrences of patterns of size 30 with $k$ errors. The $y$ axis units are in seconds and common to the three plots.




DNA $k = 1$ to 2, and in proteins $k = 1$ to 5. This is particularly troublesome since indexed approaches are the best alternative only for low error levels. In fact the sequential approaches outperform the compressed indexed approaches for higher error levels. In DNA this occurs at $k = 4$ and in English at $k = 5$.

Our index performed particularly well on proteins, as did the hybrid index. This could owe 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 that are not very compressible. We presented the space that the algorithms need to operate and not just the index size, since the other approaches need intermediate data structures to operate.

## 6 Conclusions and Future Work

In this paper we presented an adaptation of the hybrid index for Lempel-Ziv compressed indexes. 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, however it can improve previous approaches [18]. However for a Lempel-Ziv index it is essential.

Finally, our 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 as assigning smaller weights to less frequent letters of $O$. This should decrease the number of positions to verify and improve the overall performance.

*Acknowledgments.* We are thankful to Pedro Morales for lending us the LZI, DLZI and FMIndex prototypes.

## References

1. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1) (2001) 31–88
2. Chang, W.I., Marr, T.G.: Approximate string matching and local similarity. In Crochemore, M., Gusfield, D., eds.: CPM. Volume 807 of LNCS. (1994) 259–273
3. Fredriksson, K., Navarro, G.: Average-optimal single and multiple approximate string matching. ACM Journal of Experimental Algorithmics **9**(1.4) (2004)
4. Navarro, G., Baeza-Yates, R., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Engineering Bulletin **24**(4) (2001) 19–27
5. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC. (2004) 91–100
6. Maaß, M., Nowak, J.: Text indexing with errors. In: CPM. (2005) 21–32
7. Chan, H.L., Lam, T.W., Sung, W.K., Tam, S.L., Wong, S.S.: A linear size index for approximate pattern matching. In: CPM. (2006) 49–59
8. Coelho, L., Oliveira, A.: Dotted suffix trees: a structure for approximate text indexing. In: SPIRE. (2006) 329–336
9. Weiner, P.: Linear pattern matching algorithms. In: IEEE 14th Annual Symposium on Switching and Automata Theory. (1973) 1–11
10. Manber, U., Myers, E.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing (1993) 935–948
11. Gonnet, G.: A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland (1992)
12. Ukkonen, E.: Approximate string matching over suffix trees. In: CPM. (1993) 228–242
13. Cobbs, A.: Fast approximate matching using suffix trees. In: CPM. (1995) 41–54
14. Sutinen, E., Tarhio, J.: Filtration with q-samples in approximate string matching. In: CPM. Volume 1075 of LNCS. (1996) 50–63
15. Navarro, G., Baeza-Yates, R.: A practical $q$-gram index for text retrieval allowing errors. CLEI Electronic Journal **1**(2) (1998)

16. Myers, E.W.: A sublinear algorithm for approximate keyword searching. Algorithmica **12**(4/5) (1994) 345–374
17. Navarro, G., Baeza-Yates, R.: A hybrid indexing method for approximate string matching. Journal of Discrete Algorithms **1**(1) (2000) 205–239
18. Navarro, G., Sutinen, E., Tarhio, J.: Indexing text with approximate $q$-grams. J. Discrete Algorithms **3**(2-4) (2005) 157–175
19. Kurtz, S.: Reducing the space requirement of suffix trees. Softw., Pract. Exper. **29**(13) (1999) 1149–1171
20. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Algorithms **48**(2) (2003) 294–313
21. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM **52**(4) (2005) 552–581
22. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. Discrete Algorithms **2**(1) (2004) 87–114
23. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2) (2005) 378–407
24. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys **39**(1) (2007) article 2
25. Manzini, G.: An analysis of the Burrows-Wheeler transform. Journal of the ACM **48**(3) (2001) 407–430
26. Kärkkäinen, J., Ukkonen, E.: Lempel-Ziv parsing and sublinear-size index structures for string matching. In: South American Workshop on String Processing, Carleton University Press (1996) 141–155
27. Arroyuelo, D., Navarro, G., Sadakane, K.: Reducing the space requirement of LZ-Index. In: CPM. Volume 4009 of LNCS. (2006) 318–329
28. Russo, L.M.S., Oliveira, A.L.: A compressed self-index using a Ziv-Lempel dictionary. In: SPIRE. Volume 4209 of LNCS. (2006) 163–180
29. Huynh, T., Hon, W., Lam, T., Sung, W.: Approximate string matching using compressed suffix arrays. In: CPM. (2004) 434–444
30. Lam, T., Sung, W., Wong, S.: Improved approximate string matching using compressed suffix data structures. In: Algorithms and Computation (ISAAC). (2005) 339–348
31. Morales, P.: Solución de consultas complejas sobre un indice de texto comprimido (solving complex queries over a compressed text index). Undergraduate thesis, Dept. of Computer Science, University of Chile (2005) G. Navarro, advisor.
32. Ziv, J., Lempel, A.: Compression of individual sequences via variable length coding. IEEE Transactions on Information Theory **24**(5) (1978) 530–536
33. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM **46**(3) (1999) 395–415
34. Navarro, G., Baeza-Yates, R.: Very fast and simple approximate string matching. Information Processing Letters **72** (1999) 65–70