Approximate string matching with Ziv-Lempel compressed indexes

Luís M. S. Russo**,1 Gonzalo Navarro3, and Arlindo L. Oliveira2

1 INESC-ID, R. Alves Redol 9, 1000 LISBOA, PORTUGAL
lsr@algos.inesc-id.pt
2 aml@inesc-id.pt
3 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 used to search patterns \( P \) in \( T \) that requires reduced space, i.e. that depends on the empirical entropy \( (H_k, H_0) \) of \( T \), and is, furthermore, able to reproduce any substring of \( T \). In this paper we present an algorithm for the approximate string matching problem that uses Lempel-Ziv compressed indexes as filtering indexes. To achieve this goal we start by giving new results for \( q \)-samples filtering indexes. We then explain how to adapt this approach for Lempel-Ziv indexes. Moreover our approach can easily be extended to a hybrid index. Hybrid indexes are the state of the art standard for this problem, obtaining sub-linear average time. We show experimentally that our algorithm has a performance similar to the hybrid index of Navarro, providing a trade-off, requiring more time but less space.

1 Introduction

Approximate string matching is an important problem, it is relevant for several branches of computer science. Among others its applications are related to text searching, pattern recognition, signal processing and computational biology.

The problem consists in locating all occurrences of a given pattern string \( P \), of size \( m \), in a larger text string \( T \), of size \( u \), assuming that the pattern can be distorted by errors. The errors considered are only insertions, deletions and substitutions. Similarly to the fundamental string processing problem two approaches are immediately relevant. In the on-line the version the we can pre-process the pattern but not the text. The classical solution to this problem uses dynamic programming and runs \( O(um) \) worst case time [1]. Several algorithms refined this initial approach [2]. An optimal average case algorithm requires \( O(u(k + \log \sigma m)/m) \) [3], where \( \sigma \) is the size of the alphabet \( \Sigma \) and \( k \) is number of errors tolerated.

For long texts however the on-line approach is much less appealing, since the \( \Omega(u) \) time dependency becomes impractical, specially when the number of errors is small. In order to avoid having to scan all the text we can use an auxiliary, index, data structure. Theses indexes are usually related to the indexes of exact matching problem, such as suffix trees, suffix arrays or q-grams. The main problem with these structures is their space requirements, suffix trees in particular are known for their considerable space requirements [4–6]. In recent years a new and extremely successful class of indexes has emerged. Compressed full-text indexes, which use data compression techniques to produce less space demanding data structures have been proposed by several researchers [7–11]. Usually a text stored in compressed format requires less space than its uncompressed version. The idea is that an index based on the compressed format

** Supported by the Portuguese Science and Technology Foundation by grant SFRH/BD/12101/2003 in project POCI 2010 and Project DBYEAST POSC/EIA/57398/2004
may also require less space. In fact, it turns out that data compression algorithms explore the internal structure of a string much in the same way that indexes do. An important tool to describe the space of compressed indexes is the $k$-th order empirical entropy of $T$, defined by Manzini [12], denoted simply by $H_k$. The empirical entropy provides a measure of the complexity of $T$ taken as a finite object. This is opposed to the classical notion of entropy by Shannon. The empirical entropy provides a lower bound to the number of bits needed to compress $T$ using a compressor that encodes each character considering only the context of $k$ characters that follow it in $T$. Navarro and Mäkinen presented a comprehensive survey on compressed full-text indexes [13]. Some examples of compressed indexes are compressed suffix arrays [7, 11], the FM-index [8] and Lempel-Ziv based indexes [8–10, 14, 15], which are the focus of this work.

Algorithms for indexed approximate string matching are usually divided into two categories, the neighborhood generation type (DFS) and the filtering type. The DFS type [16] consists in performing a depth-first search on a suffix tree, the search is guided by the dynamic programming algorithm, i.e. the dynamic programming is used to determine whether a given branch should be further inspected or abandoned. Starting this search from the root of a suffix tree is sufficient to locate all the approximate occurrences of the pattern. The time of this approach is asymptotically independent of $u$ but exponentially dependent on $m$ and $k$, i.e. $O(\min(3^m, (2σm)^k))$ in average and in the worst case. The filtering approach consists in adapting on-line filtering algorithms to the indexed method. Filters are algorithms that discard large parts of the text using a faster algorithm than dynamic programming, i.e. a simpler condition than matching that allows for false positives but not for false negatives. Most filters consist in searching for substrings of the pattern without errors and checking for potential occurrences around those matches. This type of indexes usually resorts to q-grams indexes to quickly locate those substrings. Filtration indexes usually need much less space than suffix trees, however the price they pay is that they perform badly for medium and large error level $\alpha = k/m$. In fact the are only sub-linear for $\alpha = O(1/\log_σ u)$.

A third type of algorithms was proposed to strike a balance between the two previous approaches. The idea is to combine the two previous approaches in a hybrid way obtaining even better performance. The algorithm is similar to the filtration approach because in splits the pattern in to pieces. However the pieces do not have to occur exactly, in fact the errors are distributed by the pieces and a DFS is performed for each piece. An algorithm using q-grams was proposed by Myers [17] and one using suffix trees was give by Navarro [18]. In this paper we propose a hybrid algorithm using a Lempel-Ziv compressed index.

This paper is organized as follows: in section 2 we define the basic notation and the concepts related to the edit distance. In section 3 we present our adaptation of the filtration phase for q-samples indexes. In section 4 we give a brief description of Lempel-Ziv compressed indexes, pointing out its main data structures. In section 5 we adapt out the algorithm to work with the LZ-Index. In section 6 we show some experimental results. In section 7 we present some conclusions and future work.

2 Basic Concepts and Notation

For basic concepts related to strings and suffix trees we refer the reader to Gusfield [19]. We use the following conventions: the empty string is denoted by $\varepsilon$; $S[i]$ represents the symbol at position $i$; strings start at index position 0; prefixes, substrings and suffixes are denoted respectively as $S[..i]$, $S[i..j]$, $S[j..]$; The size of a string $S$ is denoted by $|S|$; $m$ is $|P|$ and $u$
is $|T|$; $\Sigma$, of size $\sigma$, is the underlying alphabet. By suffix tree we refer to a generalized suffix tree. The terminator symbols are not considered as part of the edge-labels, instead we allow for unary internal nodes to exist provided that they correspond to a suffix of some indexed string. A point is a node in the suffix trie. We refer indifferently to points in a suffix tree and to their path-labels. By $\text{DFS}(v)$ we refer to the depth-first time-stamp [20] of a node $v$ in a suffix tree.

2.1 Edit Distance

Definition 1. The edit or Levenshtein distance, $\text{ed}(S, S')$, between two strings is the smallest number of edit operations that transform $S$ into $S'$. We consider as operations insertions (I), deletions (D) and substitutions (S).

For example the edit distance of between $abcd$ and $bedf$ is 3, i.e. $\text{ed}(abcd, bedf) = 3$. This value can be computed with a dynamic programming table $D[i, j] = \text{ed}(S[..i], S'[..j])$, see figure 1 for local relations and an example.

A comprehensive survey on the different approaches to compute the edit distance, is available [2] and should be consulted for an in depth description. The survey shows that, in practice, the most effective is algorithms for small values of $m$ are the ones using non-deterministic automata (NFA), implemented using bit-parallelism [21, 22]. For this reason hybrid algorithms use this approach. By error level, $\alpha$, we refer to the quotient of $k$ and $m$, i.e. $\alpha = k/m$.

3 A Filtering q-samples index

The simplest filtration technique consists in searching for approximate occurrences of substrings of the pattern. Recall that the time to locate every approximate match of $P$ in $T$ has significant dependency on $m$, i.e. it is $O(\min(3^m, (2\sigma m)^k))$. Therefore reducing $m$ will improve this time. However this will also mean that the matches found this way are not necessarily occurrences of $P$, i.e. this technique is a filter because it provides false positives but not false negatives and its computational cost is smaller than the precise method. The exact way to apply this idea is given by the following lemma:
Lemma 1. Let $A$ and $B$ be two strings such that $ed(A, B) \leq k$. Let $A = x_0 A_1 x_1 A_2 x_2 \ldots x_{j-1} A_j x_j$, for strings $A_i$ and $x_i$ and for any $j \geq 1$. Then, at least one strings $A_i$ appear in $B$ with at most $\lfloor k/j \rfloor$ errors.

In essence what lemma 1 means is that if we split the pattern into $j$ pieces, the $A_i$, and distribute the errors evenly by the pieces then at least one of the pieces will occur in the text with at most $\lfloor k/j \rfloor$ errors, i.e. the approximate match of the pattern implies the approximate match of some of its pieces. Therefore when searching the text, either on-line or with an index, if we find an approximate match of a piece then it is worth to scan the surrounding location for an approximate match of the pattern.

Observe that the lemma lets us decide which string to split, $P$ or the occurrence of $P$ in $T$, this is done by choosing the string we wish to control the partition to be the string $A$, on the other hand the substrings of $B$ cannot be chosen, instead they are induced by the partition of $A$. In a q-grams index it is possible to use $A$ as $P$ because no matter how the pattern induces a partition on the text it is always possible to use a q-gram that is aligned with that partition, i.e. we can choose how to partition $P$. On q-sample indexes on the other this approach is not so simple. The original work on q-samples filtering indexes was done by Sutinen and Tarhio [23] and was later extended into a hybrid index by Navarro et. al. [24]. In their work the lemma is applied by considering that $B$ is the pattern and that $A$ is its occurrence in the text. This is a very natural approach since in a q-samples index only some substrings, the samples, of $T$ are indexed, not all of them. In q-sample indexes there is a criteria for choosing which samples to consider, that consist in choosing $q$, the sample size, and $h$, the sampling step. The sampling step determines from which positions of $T$ to take samples, i.e. if $i \equiv 0 \mod h$ then $T[i..i+q]$ is taken as a sample, $h \geq q$ most be true to avoid overlapping the samples. In this discussion we considered that $h = q$.

In this work however it is not crucial to decide with string does $P$ correspond to, because we will need an iterated version of the lemma either way, still in order to simply the formulas we present we assume that $A$ represents the pattern and that string $B$ represents its occurrence in the text.

It is important to notice that $j \cdot \lfloor k/j \rfloor = k$ does not always hold and therefore the errors are not only being distributed across the pieces but some of them are also being removed. In this work we adopt a simple partition strategy, that is not optimal. We split $P$ as $P_0 \ldots P_j$ where $|P_i| = \lfloor (m - |P_0 \ldots P_{j-1}|)/(j-i) \rfloor$. Observe also that we ignored the $x_i$ strings in the lemma, these strings are useful when we wish to discard part of the pattern. By discarding we mean is that those parts are not searched for, this in turn implies that the errors are not distributed by the $x_i$ pieces. It may seem therefore that the $x_i$ pieces are not relevant since they end up increasing the error level, i.e. making the problem harder. However they are conceptually important and will be important for our algorithm. For now they will not be used, i.e. they are assumed to be $\epsilon$.

A q-samples index is a q-grams index, i.e. an index the indexes only substrings of $T$ of size $q$, for which the indexed strings do not overlap. In this work we assume that the q-samples are consecutive, i.e. there is no spacing between samples. For example for $T = cbdbdcbababa$ a 2-samples index would split $T$ a cb.db.dd.cb.ab.ab.a. For each sample the index stores a list of the indexes in $T$ where the sample occurs. For example sample ab would store \{8, 10\}, in order to save space the previous values can be stored divided by $q$.

The initial work on approximate matching with Lempel-Ziv compressed indexes was done by Castillo and Navarro [25] and was a filtration approach. They applied lemma 1 with $A = P$
and \( j = k + 1 \) and then used the LZIndex [9]. They also presented some interesting refinements that we will not discuss here. The fundamental problem of applying lemma 1 directly over a q-samples index is the alignment between the points in which \( P \) is split and the points where the samples finish or begin in \( T \). Essentially what happens is that there is no relation between this points, i.e. the points in the pattern and in the text are not aligned. Suppose we are searching for pattern \( ddcbabbb \) with one error, partitioned as \( ddeb.abb \). Observe that this does occur in \( T \), \( cbdbdd.abbb.a \), note that we are using a 6-samples index. Applying lemma 1 we have that either \( ddeb \) or \( abbb \) appear exactly in \( T \), in fact it is \( ddeb \). However \( ddeb \) occurs misaligned in the 6-samples index, i.e. it does not appear as a q-sample or inside a q-sample but split across q-samples. Therefore in order to be able to find this occurrence it is necessary to further split the pieces of the pattern. It is necessary to split the pieces at every position and to search every occurrence of the respective piece, i.e. in this example we would search for strings \( ddeb, ddc, dd \) or \( cb \) and \( dcb \) because of \( ddeb \) and strings \( abbb, abb, ab \) and \( bbb \) because of \( abbb \). We assume that checking for an occurrence of a piece is fast i.e. \( O(1) \), however this method does require a considerable amount of verifications.

Let us assume the \( T \) is random and determine the expected number of positions to verify. First observe that strings of size \( m/j \) are expected to occur in \( u\sigma^{-m/j} \) positions in a random string of size \( u \), with underlying alphabet of size \( \sigma \). We will now determine the overhead of using this approach, i.e. the factor of \( u\sigma^{-m/j} \). We assume that divisions always return integers in order to simplify the analysis. It is complicated do deal with the value of \( q \) in a coherent way since Lempel-Ziv indexes are q-sample indexes with samples of different sizes, in this particular analysis we assume that \( q > m/j \). Moreover it is not trivial to determine the size of the resulting pieces if we allow for errors to occur in them, therefore in the following analysis we assume that \( j = k + 1 \). However the expect that an analysis allowing errors would produce similar results. Using this approach the overall number of inspected positions, adding up over all pieces, is affected by the following factor:

\[
\frac{m}{j} 2^{\left(1+\sigma+\ldots+\sigma^{m/(2j)}\right)} = \frac{2m}{j} \left(\frac{\sigma^{1+m/(2j)} - 1}{\sigma - 1}\right) = O(m\sigma^{m/(2j)}/j)
\]

This rough analysis means that by further splitting the pieces we incur in a significant penalization. In fact the analysis also shows that the worst case happens precisely when the subtrings must be split in half, it is in this case that we get the \( \sigma^{m/(2j)} \) factor that dominates the overall time. By analyzing this case more closely we can explain the main problem of this approach. The problem is that this successive splitting of the pattern ends up forcing us to search for very small strings in the index. In our example we started of with a string of size 8 and ended up searching for substrings of size 2 to avoid dealing with one error. We will now show that it is possible to adapt another strategy in which we never have to search for patterns that small. In fact in this example we can always search form strings of size 3. The fundamental problem in this example is that we didn’t searched for \( cba \), instead we searched only for \( cb \) which made the problem harder than it needed to be. What condition can we use to justify searching for \( cba \) instead of \( cb \)? can we search for \( cbab \) instead ? if not why not ?

The answer to these questions lies again in lemma 1. Let us assume the like previously we are going to use this lemma with \( j = 2 \), however this time we are going to try a different approach. In the previous approach we first divided the pattern with lemma 1 and then worried about the alignment of the pieces with the q-samples. We will try a sort of integrated approach. We apply lemma 1 to \( ddebabbb \) tree times, first we divide it as \( ddeb.bab.XX \) then as
Proof. The proof consists in applying lemma 1 several times with some characters in the edges. We first apply the lemma using the following partition $A_i = \{q_i..q_{i+1}\}$ for $1 \leq i \leq j$, $x_i = \epsilon$ for $0 \leq i < j$, $x_j = P[q(i+1)$..]. As a second partition we set $A_i = P[1 + q_i..1 + q(i+1) - 1]$ for $1 \leq i \leq j$, $x_i = \epsilon$, for $0 < i < j$, $x_0 = P[0]$ and $x_j = P[1 + q(i+1)$..]. Therefore for some general integer $h$, such that $0 \leq h \leq m - qj$ we set $A_i = P[h + q_i..h + q(i+1) - 1]$ for $1 \leq i \leq j$, $x_i = \epsilon, 0 < i < j$, $x_0 = P[..h - 1]$ and $x_j = P[h + q(i+1)$..] as a partition. An occurrence of the pattern in the text will be aligned with some $h$ and therefore the respective application of lemma 1 finds it. □

Let us just focus on the performance of using this approach. Using this approach instead of the previous has a significant impact on performance, since the number of positions to verify is much smaller than in the previous approach. For random text the factor associated with $u\alpha^{-m/j}$ is given by the following expression:

$$\left( m - \frac{m+1}{j+1} \right)^{\frac{m}{j}} \frac{m(j+1)}{j+1} = O(m\alpha^{(m-j)/j(j+1)})$$

Note that this is a significant improvement since $j$ is always at least 1 and therefore $j+1 \geq 2$. Moreover as $j$ becomes larger the more significant the improvement is. Even so these two factors are not immediately comparable. In the previous approach every position indicated by the index was checked in $O(1)$ to verify if it corresponded to a piece of the pattern, only after that was dynamic programming employed to verify an occurrence of the whole pattern. In our approach every position indicated by the index is immediately verified for an approximate match of the pattern, this requires $O(m^2)$ time and therefore the factors we should be comparing are $O(m\alpha^{m^2/(2j)j})$ and $O(m^3\alpha^{(m-j)/j(j+1)})$. Even so our approach will given a significant improvement as we show in the results section. Moreover instead of using dynamic programming right away we can apply the previous filter, this will reduce our factor back to $O(m^2\alpha^{(m-j)/j(j+1)})$ if we need to extract the letters from the samples or to $O(m\alpha^{(m-j)/j(j+1)})$ if we do not.

From this comparison it seems that the larger $j$ is the better the improvement is. This however is not exactly the case since filtering indexes are only sub-linear, while $\alpha = k/m = O(1/\log \sigma u)$. Recall that we are considering that $j = k+1$, therefore as $k$ becomes larger both searches became linear the less significant the improvement is.

4 Lempel-Ziv Compressed Indexes

In this section we will provide a brief description of Lempel-Ziv compressed indexes. An easy way to understand Lempel-Ziv indexes is to realize that they are very similar to q-sample
indexes, i.e. they consist in picking some substrings of $T$, samples, and keeping track of where they occur in $T$. Several Lempel-Ziv compressed indexes have been proposed \[10, 8, 9, 14, 15\] in the literature. In this work will give a simplified general version.

Q-sample indexes are very compact, however the criteria for choosing samples is not particularly mindful of the internal structure of $T$, therefore they require optimal size only if $T$ is a random text. In order to guarantee that we have an index of optimal size, $O(uH_k)$ bits we must do a more criterial selection of the samples. For this purpose we will use data compression techniques. The Lempel-Ziv is a data compression algorithm that works by parsing the text string into a sequence of blocks.

**Definition 2.** The **LZ78 parsing** of a string $T^R$ is the sequence $Z_1, \ldots, Z_n$ of strings such that $T^R = Z_1 \ldots Z_n$ and for every $i$, $Z_i = Z_jc$ where $Z_j$ is the largest prefix of $Z_i \ldots Z_n$ among the $Z_1, \ldots, Z_{i-1}$.

For technical reasons we will parse $T$ reversed, in our example $T^R$ is parsed into $a.b.ab.abc.d.db.dbc$. The $Z_i$ strings are referred to as the LZ-blocks. The $Z_i^R$ strings are the samples we are going to use in our index. Using the Lempel-Ziv algorithm gives a degenerated type of samples index, since each sample, i.e. LZ-block, occurs in only one position in $T$. In the work of Russo and Arlindo \[15\] this is not necessarily true, but the improvement they obtain in their algorithm is not applicable to approximate string matching.

In order to be able to search for strings inside the $Z_i$ strings we organize them in labeled trees. The tree formed by the $Z_i$’s is known as the Lempel-Ziv trie and allows us to extend the $Z_i$’s, which is similar to searching $T$ from left to right, recall that we parsed $T^R$. The tree formed by the $Z_i^R$’s, denoted by $T_{78}$, allows us to extend the $Z_i^R$’s, i.e. similar to searching $T$ from right to left. An interesting characteristic of the $T_{78}$ is that, since the $Z_i^R$’s form a suffix closed set, it is a suffix tree. This fact is explored by Russo and Arlindo in their search algorithm and in the representation of $T_{78}$. This structure is denominated as Lempel-Ziv suffix tree or by RevTrie \[9\] when it is represented as a trie.

Therefore LZ-blocks do not only provided a sampling of $T$ that can be represented in $uH_k + o(u \log \sigma)$ bits, this is essentially what Lempel-Ziv compression algorithms such as gzip do, they also contain enough structure to built an organized index. These two trees can be linked by a canonical mapping $R$ that maps the nodes of $T_{78}$ to the nodes of the Lempel-Ziv trie, by reverting the path-labels (see figure 2). Note that this correspondence isn’t defined for every node of $T_{78}$ since $cb$ has no corresponding node $bc$ in the Lempel-Ziv trie.

For our purposes a Lempel-Ziv compressed index will consist of the LZ-Trie, $T_{78}$, and $T$ stored in compressed form, for example as the sequence of DFS values in $T_{78}$. Several ways of representing this structure have been proposed that differ in the functionality they provide for $T_{78}$. Since in this case we won’t be needing suffix links we can use the representation of Arroyuelo et. al that requires $(2 + \epsilon)uH_k + o(u \log \sigma)$ bits of space for any constant $\epsilon > 0$.

## 5 A Hybrid Lempel-Ziv index

In this section we explain how adapt to our q-samples algorithm for approximate string matching in order for it to work with LZ-indexes. As we pointed out, regardless of its internal structure, LZ-indexes are essentially q-samples indexes but where the samples are not all of the same size. We might naively think that this was not a very relevant difference and that the previous approach would still work. However this is not the case, consider the pattern $ddccabab$ that we wish to search for with one error in $T$ supposing that it is sampled as
**Fig. 2.** (left) Lempel-Ziv trie. Nodes show their DFS values. The $R$ mapping is shown and $R(3)$ is indicated by a bold arrow. (right) Suffix tree for strings \{a, b, ba, cba, cbd, d\}. Suffix link from cb to b shown by a dashed arrow. Nodes show their DFS values in $T_{78}$.

cbd.bd.decbab.aba. If we were to use lemma 2 we would search for all substring of $P$ of size 3, i.e. ddc, dec, cca, cdb, aba, bab. However none of this strings will be useful to find the match, the first one and the last two are not helpful since for this particular match they are misaligned. The remaining 3 substrings should solve the problem, however they do not, note that the only error between $P$ and $T$ is the c that is underlined in all these samples, therefore none of them will find this occurrence. Note that by searching for a string like aba we do not mean exactly the sample aba but all the samples that contain aba, these are obtained by combining a DFS in $T_{78}$ with a DFS in the LZ-trie by using the $R$ mapping.

A simple way to solve this problem is by reinforcing lemma 2, in essence what that lemma shows is how to pay computationally to be able to guarantee that an occurrence of $P$ is somehow aligned with the beginning of a sample. Naturally we can apply the same technique now to guarantee that the occurrence of $P$ is aligned with the end of a sample.

**Lemma 3.** Let $T$ be a text of size $u$ indexed by an LZ-index, $P$ a pattern of size $m$ then this index can be used as a filtering index for $k$ errors by consulting the samples that contain substrings of $P$ of size $\lfloor (m+1)/(j+2) \rfloor = \lfloor (m+1)/(k+3) \rfloor$.

**Proof.** Like lemma 2 the proof consists in applying lemma 1 several times with $j = k + 1$. Let $q = \lfloor (m+1)/(j+2) \rfloor$ we consider the partitions with $A_i = P[h + i..h + q(i + 1) - 1]$ for $1 \leq i \leq j$, $x_i = \epsilon, 0 < i < j$, $x_0 = P[.h - 1]$ and $x_j = P[h + q(i + 1).]$ for any $h$ such that $0 \leq h \leq m - q(j + 1)$. What this approach is doing is to focus in the middle string $P[q..qj + 1]$, putting all the errors in this string. For every point in this string we have at least one configuration that considers the point as beginning of an $A_i$ and one configuration that considers the point as the end of an $A_i$. The end result is that we can allow for the samples to have any configuration, except of course when they are all too small, for example if every sample is just one letter. To solve this problem we search the samples that are substrings of the $A_i$, it is very important to notice that these substrings are not extended to the left or right, i.e. they are not considered as substrings of other samples beyond themselves. □

Observe that what the previous lemma is doing is to reserve some pattern space to be able to solve the problem we pointed out. In the previous example if there was some pattern left after ab that could have been used to find the occurrence.
Also note that searching for the LZ-blocks that are substrings of $P$ smaller than $q$ will add at most $O(qm)$ verifications. For a random text the factor associated with $u\sigma^{-m/j}$ is:

$$\left( m - \frac{m + 1}{j + 2} \right) \frac{m + 1}{j + 2} \sigma^{-m/j} = \frac{mj + m - 1}{j + 2} \frac{2m - j}{j(j+2)} = O(m\sigma^{(2m-j)/(j(j+2))})$$

Although this factor is not as good as the second one it still is better than the first filtering approach as soon as $j = 2$, which requires only one error.

Finally notice that even thought, in lemma 3, we presented the index only as a filtering index it can in fact be used as hybrid index by choosing a value of $j$ smaller than $k + 1$. It is important to notice that the factors we presented are not accurate in that case, however we expect the results to be similar. In practice all we have to do is to use dynamic programming over $T_{78}$ the same way as in the hybrid index.

6 Practical Issues and Testing

We implemented a prototype for testing these ideas. Our prototype is based on the ILZ-Index of Russo and Arlindo (ILZI). The main difference between our algorithm and the theory
above is that we divided $P$ into pieces of size $(m + 1)/(j + 1) = q$ instead of pieces of size $(m + 1)/(j + 2)$. The main reasoning for this was that once we know were a piece $A_i$ begins we may already have an idea were it ends, for example we know it is limited by the size of the largest existing block. Exploring the structure of $T_{78}$ suffix tree can lead to more accurate bounds. In practice what we did was a second search were we allowed for all the errors to focus in the middle, i.e. with an error level of $\alpha = k/(m − 2(q − 1))$. However for this search we did not extended the samples that were found, much like what we did with the substrings of the $A_i$’s. In theory this phase could dominate the overall time but in practice it used around half the overall time. As a baseline we also implemented an NFA, based on the bit-parallel algorithm of Wu and Manber [21], (ShiftAnd).

We also used an implementation of a filtration index based on Navarro’s LZIndex using the simple filtering approach, i.e. the first approach, (LZI), and a variation with improved dynamic programming filtering, (DLZI). Finally we used another compressed index as a filtering index, the FMIndex. Note that the FMIndex does not divide the text into blocks, however it does take longer to locate occurrences. As texts we used the files in the Pizza&Chili corpus, with around 50 Megabytes each4.

The pattern strings were sampled randomly from the text and each character was distorted with 10% of probability. The patterns all had size 30, i.e. $m = 30$. The Tests ran for around 60 seconds each. Note that the time axis uses a logarithmic scale.

The results, presented in figure 3, show that the behavior of our algorithm is very similar to the hybrid index, this confirms our claim that we obtained a similar performance were we need more time but less space. It can also be observed that our algorithm is not much better than the simple filtering approach when $\sigma$ is small, as is the case in DNA. Our algorithm is also not much better than the simple filtering approach for English, which is expected since our analysis was for random text, however for a small error level we did obtain significantly better results. In the proteins file our approach was considerably better since the alphabet is large enough to obtain some improvements and the random model is better at describing this text than it is at English.

7 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 gave a brief description of LZ-Indexes as q-sample indexes and adapted our algorithm for this kind of indexes. Moreover our approach was flexible enough to be used as a hybrid index instead of just a filtering index. We implemented our algorithm and compared it against the simple filtration approach, both with the LZIndex and the FMIndex, a bit-parallel NFA algorithm and the hybrid index. Our results show that, for a small amount of space, our approach is competitive, it is also the fastest for a low error level and for large alphabets when the string is not too structured. Also bear in mind that our implementation is not overly sophisticated and in particularly we do no secondary filtering, i.e. we can apply the smaller filter before using dynamic programming. Applying secondary filtering, possibly with hierarchical verification, should provide a method that is always better than our approach and the simple filtering technique, giving the fastest algorithm for this problem with reduced space requirements.

4 Tested on Pentium 4, 3.2 GHz, 1 MB of L2, 1Gb of RAM, with Fedora Core 3, compiled with gcc-3.4 -O9.
Acknowledgments

We are thankful to Pedro Morales Castillo for providing lending us the LZI, DLZI and FMIIndex prototypes.

References