Indexed Hierarchical Approximate String Matching

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

[1] INESC-ID, R. Alves Redol 9, 1000 Lisboa, Portugal. aml@algos.inesc-id.pt
[2] Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl
[3] CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, Portugal. lsr@di.fct.unl.pt
[4] Instituto Superior Técnico, Universidade Técnica de Lisboa, Portugal.

Abstract. We present a new search procedure for approximate string matching over suffix trees. We show that hierarchical verification, which is a well-established technique for on-line searching, can also be used with an indexed approach. For this, we need that the index supports bidirectionality, meaning that the search for a pattern can be updated by adding a letter at the right or at the left. This turns out to be easily supported by most compressed text self-indexes, which represent the index and the text essentially in the same space of the compressed text alone. To complete the symbiotic exchange, our hierarchical verification largely reduces the need to access the text, which is expensive in compressed text self-indexes. The resulting algorithm can, in particular, run over an existing fully compressed suffix tree, which makes it very appealing for applications in computational biology. We compare our algorithm with related approaches, showing that our method offers an interesting space/time tradeoff, and in particular does not need of any parameterization, which is necessary in the most successful competing approaches.

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 n, 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.

* Partially funded by the Portuguese Science and Technology Foundation by project ARN, PTDC/EIA/67722/2006.

** Partially funded by Millennium Institute for Cell Dynamics and Biotechnology, Grant ICM P05-001-F, Mideplan, Chile.

The most successful indexed approach to this problem, in practice, is so-called "hybrid" indexing. It starts with a *filtration* phase that determines the positions of potential occurrences. Those positions are then sequentially verified in the text. The pattern pieces searched for in the filtration phase are short enough to control the exponential cost of this search, and long enough so that the number of occurrences to verify in the text is also controlled. By carefully optimizing this partitioning, hybrid indexes achieve $O(mn^{\lambda})$ average time, for some $0 < \lambda < 1$, and work well for reasonably high error levels. Hybrid methods have been implemented over q-gram indexes [1], suffix arrays [2], and q-sample indexes [3]. 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 least 10 times [4]. Compressed indexes, based on succinct and compressed data structures, provide less space-demanding indexes [5]. Their space requirements are measured in terms of the empirical text entropy, H_k, which gives a lower bound for the number of bits per symbol achievable over that text by a k-th order compressor.

There have been several approaches to ASM over compressed indexes. The most successful one in practice is that of Russo *et al.* [6], which builds over a Ziv-Lempel-based compressed index, and approaches hybrid performance in practice. This is faster than our new index, still ours is significantly smaller, in theory and in practice. In addition, our algorithm can run over most compressed text indexes, in particular over *fully-compressed suffix trees* [7] (FCSTs), which offer complete suffix-tree functionality. Hence, our algorithm can be used as a subroutine in other suffix-tree-based algorithms.

2 Our Contribution

In this work we explore the impact of *hierarchical verification* on hybrid search. 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. Curiously enough, this technique was originally proposed by Myers [1] in his hybrid index and later extended and used by Navarro *et al.* [8] for an on-line algorithm. 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.

Simultaneously, our result achieves compressed space, because we use FCSTs, which are functional representations of suffix trees and in particular are bidirectional. Typical indexes, classical suffix trees in particular, are unidirectional, meaning that they can search only by using the letters at the end of the pattern. Due to the RANK/SELECT duality [5], bidirectionalily arises naturally in a class of compressed indexes, which we will refer to as *bidirectional compressed indexes*

Bidirectional indexes are one important ingredient of our approach. Another crucial piece is computing the edit distance. Algorithms for this purpose are typically unidirectional, computed from left to right, because they are based on dynamic programing or automata. Interestingly this computation was made bidirectional, more than 10 years ago, by Landau *et al.* [9]. 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.

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 achieve close performance without need of tuning (and can be improved by tuning as well).

In addition, our work addresses a very important practical issue. Compressed indexes are usually self-indexes, meaning that they do not store the text T but even so they are able to consult it. Even when in theory reading ℓ consecutive letters takes $O(\ell)$ time, experimental results show [10] that this is still two orders of magnitude slower than storing T. This can easily be explained as the penalty of missing cache in modern computer architectures. Efficient algorithms for ASM over compressed indexes must therefore minimize their accesses to T. Hence hierarchical verification directly over the index is a very important technique in this context, both in theory and in practice.

3 Basic Concepts

We denote by T a **string**; by Σ the **alphabet** of size σ; by $T[i]$ the symbol at position $(i \bmod n)$; by $T.T'$ **concatenation**; by $T = T[..i-1].T[i..j].T[j+1..]$ respectively a **prefix**, a **substring** and a **suffix**; by $S \sqsubseteq S'$ that S is a substring of S'. We refer indifferently to nodes and to their path-labels, also denoted by v. The **suffix tree** of T is the deterministic compact labeled tree for which the path-labels of the leaves are the suffixes of $T\$$, where \$ is a terminator symbol not belonging to Σ. We will assume n is the length of $T\$$. For a detailed explanation see Gusfield's book [11].

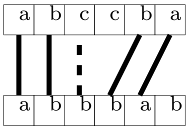

Fig. 1. Schematic representation of the edit distance between *abccba* and *abbbab*.

		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	

Fig. 2. D table computation for strings *abccba* and *abbbab*. (left) The numbers in bold refer to the alignment shown in Fig. 1. (right) Computation with increasing error bound.

The **suffix array** $A[0, n-1]$ stores the suffix indexes of the leaves in lexicographical order.

3.1 Bidirectional Compressed Indexes

Our 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, most compressed indexes are. For example, FM-indexes [12] offer a so-called LF mapping operation, which moves from the suffix array position k such that $A[k] = i$, to position k' such that $A[k'] = i - 1$. Compressed suffix arrays [13], instead, offer function ψ, moving to a k' such that $A[k'] = i+1$, thus the inverse of ψ serves as an LF mapping as well.

FCSTs [7] build complete suffix tree functionality on top of a compressed bidirectional index, in particular an FM-index fits best. The LF mapping allows FCSTs implement Weiner links [14]: $\textsc{WeinerLink}(v, a)$, 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 birectionality. 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 ($\mathrm{LCA}(v, v')$).

3.2 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. We consider as operations insertions, deletions, and substitutions.There is a well-known dynamic programming (DP) algorithm that computes the D matrix, where $D[i,j]$ is the edit distance, $ed(A[..i-1], B[..j-1])$, between the prefixes $A[..i-1]$ and $B[..j-1]$ of A and B. Fig. 2(left) shows an example of the D matrix for $A = abccba$ and $B = abbbab$. Therefore 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

$$D[i,j] = \min \left\{ \begin{array}{ll} D[i-1,j]+1 & \text{if } i > 0 \\ D[i,j-1]+1 & \text{if } j > 0 \\ D[i-1,j-1]+\delta_{A[i-1]=B[j-1]} & \text{if } i,j > 0 \end{array} \right\},$$

where $\delta_{x=y}$ is 0 if $x = y$ and 1 otherwise. Ukkonen [15] noted that in order to find cells in D whose value is k there is no need to compute cells with 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.

Assuming we have a text T, previously pre-processed into a FCST, the problem we are interested in solving in this paper is: given a pattern P and error limit k, determine all the substrings O of T for which $ed(P,O) \leq k$. As our running example consider that $P = abccba$, $k = 2$ and $T = abbbab$.The only substring O of T is $abbba$. A way to find this string, not always the most efficient one, is to perform a depth-first search over the suffix tree of T, moving one letter at a time, 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' and $ed(P,O') \leq k$, which can be checked as $D[|P|,|O'|] \leq k$, this string is reported as an occurrence. Usually we also report all the positions in T at which O' occurs, which means traversing the whole subtree 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 and the search is allowed to proceed. If, on the other hand, there are no active cells in the last row of D, the search can be abandoned, not proceeding to deeper points. For example by looking at Fig. 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 $ed(P, O')$. This process is known as neighborhood generation and it will be a key ingredient in our algorithm. The problem with this process is that it might have a very low success rate, *i.e.* only a small percentage of the nodes visited by the process turn out to be occurrences of P.

4 Bidirectional Traversal

Our algorithm 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.* [9] 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. The extension is not restricted to B, *i.e.* we can also extend $ed(A, B)$ to $ed(cA, B)$. 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. Finally, the data structure they use in their algorithm 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[5]. For our algorithm this idea suffices since, as in the previous paragraph, the states we need to visit are always organized in a stack. Therefore 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 smaller pieces. We will use the following filtration lemma.

Lemma 1 ([10]). *Let A and B be strings, let $A = A_0 A_1 \ldots A_j$, for strings A_i and some $j \geq 1$. Let $k_i \in \mathbb{R}$ such that $ed(A, B) < \sum_{i=0}^{j} k_i$. Then there is a substring B' of B and an i such that $ed(A_i, B') < k_i$.*

In our algorithm we will use $A = P$ and $B = O$ and divide the errors in a homogeneous fashion, *i.e.* choose $k_i = \alpha|A_i| + \epsilon$, where $\alpha = k/m$ and

[5] It seems to us that it is possible to extend their algorithm to support this directly, but if that is not the case we can still use the rollback log idea.

$\epsilon > 0$ is a number that can be as small as we want and it 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 $ed(bba, cba) \leq 1$. On one hand this is good because it validates the lemma. On the other hand it is excessive because the same string will be found in more than one way. To solve this redundancy 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 in 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 again 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 neighborhood generation. It was shown [2] that the optimal point occurs for $j = \Theta(m/\log_\sigma n)$, 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 hybrid point and hence eliminate the need for parameterization.

5 Indexed Hierarchical Verification

We modify the verification phase, after filtration, in two ways. (1) We will perform it over the FCSTs instead of over T, to factor our 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 1.This technique was shown to be extremely efficient for the online approach [2]. We use the following lemma (proof omitted).

Lemma 2. *Let A and B be strings, let $A = A_0 A_1 \ldots A_j$, for strings A_i and some $j + 1 = 2^h \geq 1$. Let $k_i \in \mathbb{R}$ such that $ed(A, B) < \sum_{i=1}^{j} k_i$. 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 $B_{i'+1}$.*

Consider our running example with $k = 2$ and $P = abccba$. Instead of applying Lemma 2 we will instead iterate Lemma 1, 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 $k_1 = k_3 = 1 \times (2/6) + \epsilon$, this 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 3.2. Hence we can choose $k_0 = k_2 = 2/3$ and $k_1 = 1/3 + \epsilon$ and $k_3 = 1/3$. 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. Fig. 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 are $\leq k$ 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 2, will be found in another case.

To determine that $ed(A_i, B_0) < k_i$ we must compute the DP 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 3.1. Moreover we need to extend the DP in unnatural directions (to the left and upwards). For this we use the result [9] mentioned in Section 3.2. 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 neighborhood generation, 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.

6 Practical Issues and Testing

We implemented a prototype, BiFMI, to test our algorithm. Lacking a FCST implementation, we simulated it with a bidirectional FM-Index over one wavelet tree [5]. We reverse the search so that the most common search (forwards) is done using LF (where the FM-index is faster) instead of ψ. We use efficient sequential algorithms as a baseline (namely *BPM*, the bit-parallel DP matrix of Myers [16], and *EXP*, the exact pattern partitioning by Navarro and Baeza-Yates [17]). We also included in the comparison authors' implementation of several competing indexes: *Hybrid* is the classical hybrid technique over plain suffix arrays [2]; *LZI* and *DLZI*

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

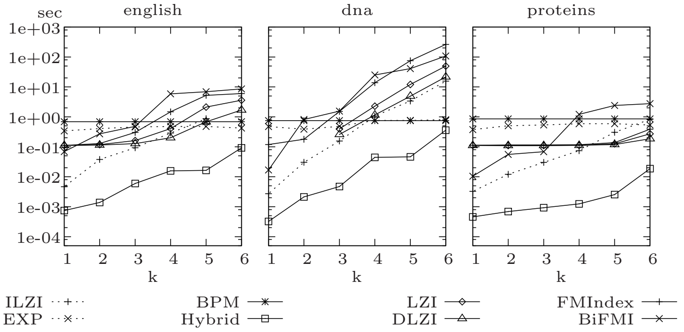

Fig. 3. Average user time for finding the occurrences of patterns of size 30 with k errors. The y axis units are in seconds and common to the three plots.

are basic and improved algorithms based on the LZ-index [18], which partition into $j = k + 1$ exact searches for pattern pieces and decompress the candidate text areas for (non-hierarchical) verification [19]; *FMIndex* is the same strategy applied over Navarro's fast and large FM-index implementation (which is much faster than our own FM-index); and finally *ILZI* is a recent ASM algorithm [6] over the ILZI compressed index [10].

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[6], 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 into an insertion, deletion, or substitution. All the patterns had length $m = 30$. Every configuration was tested during at least 60 seconds using at least 5 repetitions. Hence the number of repetitions varied between 5 and 130,000. To parameterize the hybrid index we tested all the j values from 1 to $k + 1$ and reported the best time. We did a similar process on the ILZI index. We tested our algorithm, BiFMI, in automatic mode, *i.e.* not using any parameterization.

The average query time, in seconds, is shown in Fig. 3 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. Our BiFMI index, on the other hand,

[6] `http://pizzachili.dcc.uchile.cl`

achieves the smallest space (and it can still be reduced). 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 was faster than the basic one, even if run on a much slower index (our versus Navarro's FM-Index).

Aside from the hybrid index, the fastest approach in reduced space is the ILZI-based one. The performance of our prototype closely follows that of ILZI, except for the DNA file. This indicates that we were able to approach hybrid performance. We were also, mostly, able to reduce the gap caused by cache misses. Notice that the ILZI index is consistently at most one order of magnitude slower than Hybrid, for $k \leq 3$. Our algorithm was not so effective in the DNA file but was still able to avoid two orders of magnitude slowdown for proteins and English. Notice that this is important, since aside from the ILZI, the other compressed approaches seem to saturate at a given performance for low error levels: in English $k = 1$ to 3, in 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 $k = 5$.

We did not implement the algorithm of Landau *et al.* [9]; instead we used the bit-parallel NFA of Wu *et al.* [20] 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.

7 Conclusions and Future Work

In this paper we 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.* $nH_k + O(n \log \sigma)$ bits. Compared to 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

$5nH_k + o(n \log \sigma)$ bits, but, in practice that is closer to $3nH_k$, including the sublinear term. On the other hand, a FCST requires $nH_k + o(n \log \sigma)$ 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.

References

1. Myers, E.W.: A sublinear algorithm for approximate keyword searching. Algorithmica **12**(4/5) (1994) 345–374
2. Navarro, G., Baeza-Yates, R.: A hybrid indexing method for approximate string matching. Journal of Discrete Algorithms **1**(1) (2000) 205–239
3. Navarro, G., Sutinen, E., Tarhio, J.: Indexing text with approximate q-grams. J. Discrete Algorithms **3**(2-4) (2005) 157–175
4. Kurtz, S.: Reducing the space requirement of suffix trees. Softw., Pract. Exper. **29**(13) (1999) 1149–1171
5. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. **39**(1) (2007) article 2
6. Russo, L.M.S., Navarro, G., Oliveira, A.L.: Approximate string matching with Lempel-Ziv compressed indexes. In: 14th SPIRE. LNCS 4726 (2007) 264–275
7. Russo, L., Navarro, G., Oliveira, A.: Fully-Compressed Suffix Trees. In: 8th LATIN. LNCS 4957 (2008) 362–373
8. Navarro, G., Baeza-Yates, R.: Improving an algorithm for approximate pattern matching. Algorithmica **30**(4) (2001) 473–502
9. Landau, G.M., Myers, E.W., Schmidt, J.P.: Incremental string comparison. SIAM J. Comput. **27**(2) (1998) 557–582
10. Russo, L.M.S., Oliveira, A.L.: A compressed self-index using a Ziv-Lempel dictionary. In: 13th SPIRE. LNCS 4029 (2006) 163–180
11. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press (1997)
12. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM **52**(4) (2005) 552–581
13. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. of Algorithms **48**(2) (2003) 294–313
14. Weiner, P.: Linear pattern matching algorithms. In: IEEE Symp. on Switching and Automata Theory. (1973) 1–11
15. Ukkonen, E.: Finding approximate patterns in strings. Journal of Algorithms (1985) 132–137
16. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM **46**(3) (1999) 395–415
17. Navarro, G., Baeza-Yates, R.: Very fast and simple approximate string matching. Information Processing Letters **72** (1999) 65–70
18. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. Discrete Algorithms **2**(1) (2004) 87–114
19. Morales, P.: Solving complex queries over a compressed text index. Undergraduate thesis, Dept. Comp. Sci., Univ. Chile (2005) In Spanish. G. Navarro, advisor.
20. Wu, S., Manber, U.: Fast text searching allowing errors. Commun. ACM **35**(10) (1992) 83–91