Self-Indexing Natural Language *

Nieves R. Brisaboa[1], Antonio Fariña[1], Gonzalo Navarro[2], Angeles S. Places[1], and Eduardo Rodríguez[1]

[1] Database Lab. Univ. da Coruña, Spain.
{brisaboa|fari|asplaces|erodriguezl}@udc.es
[2] Dept. of Computer Science, Univ. of Chile. gnavarro@dcc.uchile.cl

Abstract. Self-indexing is a concept developed for indexing arbitrary strings. It has been enormously successful to reduce the size of the large indexes typically used on strings, namely suffix trees and arrays. Self-indexes represent a string in a space close to its compressed size and provide indexed searching on it. On natural language, a compressed inverted index over the compressed text already provides a reasonable alternative, in space and time, for indexed searching of words and phrases. In this paper we explore the possibility of regarding natural language text as a string of words and applying a self-index to it. There are several challenges involved, such as dealing with a very large alphabet and detaching searchable content from non-searchable presentation aspects in the text. As a result, we show that the self-index requires space very close to that of the best word-based compressors, and that it obtains better search time than inverted indexes (using the same overall space) when searching for phrases.

1 Introduction and Related Work

Text indexing has become the only alternative to provide searching capabilities on the extremely large collections of strings that arise from different fields, such as bioinformatics (DNA and protein sequences), the Web and other natural language collections, software development (source code), multimedia databases and signal processing (music, audio, video and numeric streams), and so on.

For many years, the *inverted index* and its variants [4] have been a simple and effective solution to index natural language text, and the base of the success of Web search engines. We note that "natural language" is used to denote text that is composed of an alternating sequence of "words" and "separators", which can be easily distinguished syntactically; that the set of different words follows some statistical laws such as growing sublinearly with the text size (Heaps' law [14]); and especially that only whole words and sequences thereof (called "phrases") can be searched for. These limitations have been widely accepted despite they exclude many human languages (such as Chinese and Korean).

* Funded in part (for the Spanish group) by MEC grant (TIN2006-15071-C03-03), and (for the third author) by Yahoo! Research grant "Compact Data Structures". We also thank AECI grant A/8065/07.

On so-called natural language text, the basic inverted index consists of a *vocabulary*, that is, the set of different words in the text, and a *posting list* recording the text positions of each vocabulary word in increasing order. This simple data structure immediately answers single-word searches, and can handle phrase searches by essentially intersecting the corresponding posting lists. The way to carry out the intersections is still an active area of research [5, 6, 28, 10].

To save space, compression techniques have been applied to inverted indexes. In general the idea is to differentially encode each posting list (as its numbers are increasing) and encode those *gaps* with an encoding that favors small numbers. Some absolute samples are also inserted to allow fast intersections. The famous book *Managing Gigabytes* [32] describes this technology in detail. The text can be compressed as well, the preferred choice being Huffman coding [15] where source symbols are words and target symbols are bits (hence called "word-oriented bitwise Huffman"). To further save space, the text can be divided into blocks, so that the postings point to the blocks where the word appears. This is called a *block-addressing* inverted index [3, 25]. At search time, the resulting blocks must be sequentially scanned to find the exact occurrences. The block size provides an obvious space/time tradeoff. In this tradeoff, it is advantageous to opt for a text compression method that permits much faster searches than bitwise Huffman [9, 25]. Nowadays, very efficient indexed searching can be obtained by occupying, with the compressed text plus the compressed index, 30% to 40% of the original text size (and removing the original text of course).

Other variants of inverted indexes, out of the scope of this paper, are oriented to document (rather than exact position) retrieval, or to relevance ranking [4].

The situation, up to the last decade, was far less satisfactory with other types of sequences. Without a concept of word, it is necessary to provide searching for *any* text substring. This was accomplished with powerful data structures called suffix trees [31, 1] and suffix arrays [17]. Those were able to locate the *occ* occurrences of a pattern of length m in $O(m + occ)$ time, regardless of the text size. However, they require 10–20 (suffix trees) or at best 4 (suffix arrays) times the text size, plus the text, and this rendered them unsuitable in many cases.

This changed drastically with the rise of *compressed self-indexes*, which were able to represent the text in space proportional to its empirical entropy [18], and within that space, offer indexed searching for any text substring [24]. For example, the smallest compressed self-index [12] offers searching in $O(m\lceil \frac{\log \sigma}{\log \log n}\rceil + occ \cdot \log^{1+\epsilon} n)$ time, where n is the collection size, σ is the alphabet size, and ϵ is any positive constant. Another index, Sadakane's Compressed Suffix Array (CSA) [26], performs equally well in practice (despite not in theory) and has the interest for this paper of smoothly handling very large alphabets.

For example, on natural language texts, these indexes take around 60% to 70% of the original text size (and replace it). This is remarkable compared with the 400% plus text needed by suffix arrays, yet not competitive with the 30% to 40% achieved by compressed inverted indexes over compressed text. However, the comparison is not fair because self-indexes can search for any text substring whereas inverted indexes search only for whole words and phrases.

In this paper we explore the idea of applying a compressed self-index (as developed for general strings) over the sequence of *words* of a natural language text, that is, regarding the words as the basic symbols. This is promising because a self-index achieving high-order entropy should capture the dependence between consecutive words, which is significant in natural language [7, Chapter 4]. Moreover, even the slower CSA is able to locate the occurrences of a phrase of m *words* in $O(m \log n + occ \cdot \log^{1+\epsilon} n)$ time (and know the *number* of occurrences in just $O(m \log n)$ time). This compares favorably with inverted indexes, which need to carry out intersections. For example, for a phrase of 2 words appearing occ_1 and occ_2 times, an inverted index can take time $O(occ_1 + occ_2)$ or $O(occ_1 \log occ_2)$, where both occ_1 and occ_2 are (possibly much) larger than occ.

Applying a self-index to natural language words poses some challenges. A first one is that the alphabet is very large, and this rules out the theoretically best schemes [13, 12], which achieve k-th order entropy at the price of $\Omega(\sigma^k)$ extra space, where σ is the vocabulary size in our case. A text of n words is known to have a vocabulary of size $\sigma = O(n^\beta)$ [14], where $\beta \approx 0.5$ [4]. Thus σ^k may become $\Omega(n)$ already for $k = 2$! However, other self-indexes such as Sadakane's CSA [26] approach high-order entropy space without such a dependence on σ. Our first structure, the *Word CSA (WCSA)*, results from regarding the text as a sequence of word and separator identifiers and representing it with a CSA.

A second challenge is that, in many applications, we wish to have more flexible searching. For example, inverted indexes often permit to find phrases regardless of whether the words are separated by a space, two spaces, a tab, a newline, etc. This complicates the simple WCSA model where the self-index can reproduce the original text and thus the latter can be discarded. We must store some information on the separators in order to be able of exactly recreating the original text. Moreover, it is customary to apply some *filtering* on the text words to be searched [4], that is, users normally want to regard `"preprocess"`, `"pre-process"`, and `"PRE-PROCESS"` as occurrences of `"preprocess"`, and even also `"preprocessing"` and `"preprocessed"` (the latter is achieved by *stemming*, that is, indexing/searching the roots of the words). It is also usual to disregard *stopwords* (articles, prepositions, etc.) in the searches. This shows that there should be a *presentation layer*, where the text is filtered into the *searchable sequence* of (possibly stemmed, lowercase, stopwords removed) bare words, and the *presentation sequence* containing the separators and all extra information on the bare words that permits recreating the original sequence. The searchable sequence is self-indexed, while the presentation sequence is just compressed with a technique that permits fast direct access for displaying purposes. Both sequences are compressed by different means, thus the choice of what is searchable is not a space/time tradeoff but depends on user's needs. We call *Flexible WCSA (FWCSA)* this second data structure.

Our resulting data structures achieve excellent compression results, close to many natural language text compressors (that do not provide any indexing). Texts are usually compressed to around 35-40% of their original size with the FWCSA (values up to 30% can be obtained depending on the parameters used, but the resulting index becomes slow). We compare FWCSA with a block addressing inverted index (II) over compressed text using the same amount of space and offering the same functionality (a full word-addressing inverted index requires much more space, around 60-70%). The results show that, with the same available space requirements, FWCSA overcomes II when we are interested in compression ratios below 40%. When more space is available, the FWCSA is still faster for locating occurrences on either single words or phrases, except on words with many occurrences, where the II becomes superior. Also FWCSA obtains better results in the extraction of snippets for phrases in most cases.

The WCSA requires even less space, around 1-2 percentage points less than FWCSA in compression ratio. We compare the WCSA with recent related works that offer similar functionality: (1) Compressing the text with a word-oriented bytewise Huffman-like compressor prior to applying a basic (character-oriented) self-index to the result [11]; (2) reordering the bytes of the output of a word-oriented dense-code compressor in a wavelet-tree-like [13] shape, to give search capabilities to the compressed text [8]; and finally (3) a block addressing inverted index with the same functionality. Again WCSA is the best choice when little memory is available. By increasing the size of the indexes until around 45% in compression ratio, the WCSA is still the best choice for dealing with searches on phrases composed of several words. However, the wavelet-tree-like index performs better when single-word patterns are searched for.

We note that the (F)WCSA operates in main memory, and therefore requires that the compressed text does not exceed the available RAM. Because of its access pattern, the (F)WCSA is not promising on secondary memory, whereas inverted indexes perform well. Recently, however, there has been much interest in inverted indexes that operate in RAM [30, 28], motivated by the large main memories available at reasonable prices (up to 4GB is standard) and the common distributed architectures where the text collection resides in the RAM of several computers (then the problem is how to integrate the results of several indexes across the slow network). Therefore, main memory data structures are of interest nowadays, unlike what was assumed 10 years ago.

2 Sadakane's Compressed Suffix Array (CSA)

Let $T[1,n]$ be a sequence over an alphabet Σ of size σ. The *suffix array* [17] $A[1,n]$ of T is a permutation of $[1,n]$ of all the suffixes $T[i,n]$ so that $T[A[i],n] \prec T[A[i+1],n]$ for all $1 \le i < n$, being $\prec$ the lexicographic ordering. Since every substring of T is the prefix of a suffix, and all suffixes prefixed by a search pattern $P[1,m]$ are contiguous in A, we can binary search A for the interval $A[sp,ep]$ of the pointers to all the occurrences of (i.e., suffixes starting with) P in T, in time $O(m \log n)$. Each step of the binary search needs to access $T[A[i], A[i]+m-1]$ for some i, in order to compare that string with $P[1,m]$.

Let us now define another permutation $\Psi[1,n]$ such that $\Psi(i) = A^{-1}[A[i]+1]$ (or $A^{-1}[1]$ if $A[i] = n$). Hence $\Psi(i)$ tells where in A is there the pointer following $T[A[i]]$. Assume one has computed $C[1,\sigma]$, so that $C[c]$ is the number of occurrences of symbols $\prec c$ in T. We show how can one obtain the successive letters of $T[A[i]...]$ (so as to carry out the binary search) with Ψ and C and without A and T. To extract the first letter, note that all the suffixes starting with c are in the area $A[C[c]+1, C[c+1]]$, and therefore a binary search on C for the c such that $C[c] < i \leq C[c+1]$ gives the desired first letter, $T[A[i]] = c$. To extract the next letter, we use the identity $T[A[i]+1] = T[A[\Psi(i)]]$, thus we simply have to move to $i' \leftarrow \Psi(i)$ and carry out the same process again to obtain $T[A[i']]$, and so on. This is sufficent to replace A and T.

The binary search on C can be implemented in constant time as follows. Set up a string $S[1, \sigma']$, $\sigma' \leq \sigma$, containing all the different symbols that actually occur in T, in increasing lexicographical ordering. Also, set up a bitmap $D[1, n]$ with all zeros except $D[C[c]+1] = 1$ for all $c \in \Sigma$. Now, the c corresponding to an i value is $c = S[rank(D, i)]$, where $rank(D, i)$ is the number of 1s in $D[1, i]$. This is (easily) computed in constant time using $o(n)$ bits on top of D [16, 22].

The description above is the essential idea of Sadakane's CSA [26], where we have removed several possible optimizations that are not promising for our particular application (backward searching, compressed bitmaps, etc.). One important remaining point is how to compress Ψ, as in principle it is as large as the suffix array A it replaces. Sadakane shows that Ψ is formed by σ increasing subsequences, and thus it can be compressed to around the zero-order entropy of T, more precisely $nH_0(T) + O(n \log \log \sigma)$, by gap encoding its differential values. Furthermore, as shown later [24], Ψ contains at most $nH_k + \sigma^k$ (for any k) *runs* of values, so that consecutive differences equal 1 within each run. Thus, by enriching the gap encoding with run-length compression of those runs one achieves higher-order compression. Absolute Ψ values at regular intervals d are retained to permit fast random access to Ψ (yielding constant time in theory).

Note that, since we do not have A anymore, determining the interval $A[sp, ep]$ is not sufficient to locate the occurrences, that is, to output the values $A[i]$ in the interval. For this sake, the text is sampled at regular intervals l, and the suffix array positions pointing to sampled text positions are recorded, in suffix array order, into an array $A_S[1, n/l]$. Those sampled positions in A are marked in a bitmap $B_A[1, n]$, thus if $B_A[i] = 1$ we know that $A[i] = A_S[rank(B_A, i)]$. Otherwise, we try $i \leftarrow \Psi(i)$ successively, as we are virtually moving forward in T by one position at each iteration. Hence, if we determine $A[i] = j$ after k applications of Ψ, then our original value was $j - k$. Due to the regular sampling in T we carry out at most l iterations until finding a sampled position in A.

Finally, in order to discard T, we need to be able to extract any substring $T[a, b]$. For the same sampled text positions $j \cdot l$ sampled above, we store $A^{-1}[j \cdot l]$ in text position order into an array $A_S^{-1}[1, n/l]$. Thus, we find the latest sampled position $j \cdot l$ preceding a, $j = \lfloor a/l \rfloor$, and know that $j \cdot l$ is pointed from $i = A_S^{-1}[j]$. From that i we use the mechanism we have described to extract a string using C and Ψ, to find out the substring $T[j \cdot l, b]$ which covers the one of interest to us. (This is not the way Sadakane's theoretical description handles this [26], but the way he implemented it in practice.)

3 A Word-Based CSA

In this section we present the simple word-based self-index (WCSA). It can be seen as the adaptation of Sadakane's CSA [26] to a large word-based alphabet.

To create the WCSA we first map each different word or separator[1] (let us call both "words") from the source text to an integer *id*. Then, an integer sequence *Sid* is formed with the identifiers of the consecutive text words and a vocabulary array V is created to store the word corresponding to each *id*. Finally, *Sid* is self-indexed by building an integer-based CSA (iCSA) on it. The algorithm to create iCSA first builds the suffix array A of Sid, as well as D, and can discard Sid. Then, arrays A^{-1} and ψ are created, as well as B_A, A_S and A_S^{-1}. Then A and A^{-1} can be discarded. Assuming that there are σ different words, the vocabulary used by the iCSA is $\{1, 2, \ldots, \sigma\}$, so it remains implicit and there is no need to store it (nor $S[1, \sigma']$). Finally, ψ is compressed by storing some absolute samples and Huffman-encoding the consecutive gaps, including a special encoding for the runs.[2] To sum up, WCSA consists of the vector of words V (sorted alphabetically) and a bottom layer composed of an iCSA built on Sid.

As any typical self-index, iCSA provides the following basic functions using the CSA algorithms described: *countiCSA(P')* counts the number of occurrences of pattern P' in Sid; *locateiCSA(P')* locates P's positions in Sid; and *extractiCSA(l,r)* retrieves the integers $Sid[l] \ldots Sid[r]$.

Searches for a pattern $P = \{w_1, w_2, \ldots w_m\}$ on the WCSA start by binary searching V for each word w_i of P to obtain its corresponding id_i (its position in V), hence obtaining a new pattern $P' = \{id_1, id_2, \ldots id_m\}$ to be searched in the iCSA. Operation $countWordsWCSA(P)$ is directly translated into $countiCSA(P')$, and $locateWordsWCSA(P)$ to $locateiCSA(P')$ (note this gives word offsets, not byte offsets, of the occurrences). Finally, $extractWordsWCSA(s, e)$ recovers the original text from the s^{th} to the e^{th} word: We obtain the word ids with $extractiCSA(s, e)$ and then retrieve the original words stored at those positions (ids) in array V. Notice that snippets composed of k words around the occurrences of P' can be obtained by applying $occs = locateWordsWCSA(P')$ followed by $extractWordsWCSA(occs[i - k], occs[i + k])\ for\ each\ i \in [1..|occs|]$.

4 Flexible Word-Based CSA

We show how a more flexible index can be obtained based on WCSA. Our Flexible WCSA (FWCSA) can deal with many typical requirements of natural language searching, such as case-insensitive search, stemming, disregarding stopwords and/or separators, etc. The FWCSA does not index the original

[1] We parse the text using the spaceless model: If a word is followed by a single blank, that separator is not encoded but implicitly regenerated at snippet extraction time. This saves 70% of the separators [21].

[2] Further details were omitted for lack of space.

text as such, but rather a *normalized* version of it. Normalization is a user-defined function from (original) words and separators to (normalized) words or a null word. It can be used to express the requirements above[3]. We map the set of different normalized words to integer ids, then replace each word from the original text by the *id* of its normalized version (or ignore it if the normalization gives the null word), and finally build an iCSA on the resulting sequence of ids.

As we want FWCSA to be able to recover any part of the original text, some additional information has to be stored in what we call the *presentation layer*.

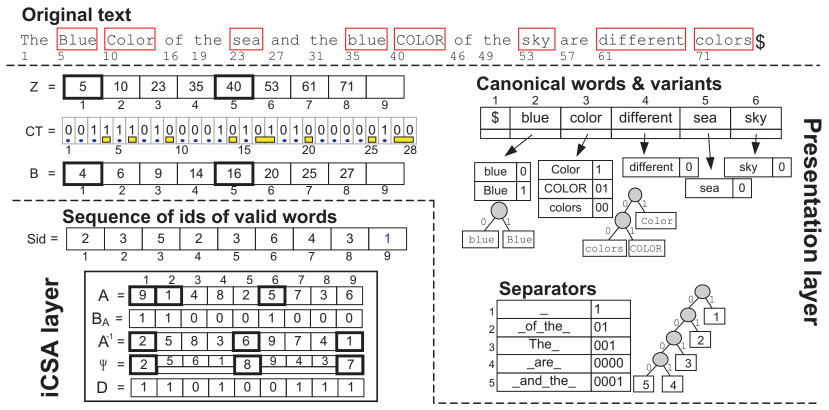

Fig. 1. General Structure of FWCSA.

Fig. 1 shows the general structure of WCSA. A first pass over the original text is needed to gather some statistics from the source text. We split the original text into "valid words" and "separators". A "valid word" is a text word or separator[4] that normalization does not map to the null word. A "separator" in this context is all the text between valid words, that is, a maximal sequence of text words and separators mapped to the null word by normalization. Hence valid words and separators strictly alternate in the text[5]. A vocabulary of *canonical* (i.e., normalized) words is built, and kept sorted alphabetically. For each canonical word, a list with all the variants that the normalization process maps to it is stored (sorted by frequency). Similarly, a vocabulary containing all the "separators" in the source text is created and sorted by frequency.

A second pass over the original text permits to fill the structures from the presentation layer shown in Fig. 1, as well as array *Sid*. Notice that $Sid[1] = 2$ because the first valid word from the text, "`Blue`" is mapped via normalization

[3] For example, if one wishes a case-insensitive search ignoring stopwords and separators, a proper normalization could map all words to their lowercase version, and stopwords and separators to the null word.

[4] What is a text word can also be user-defined, being the typical definition a maximal sequence of letters and digits.

[5] If normalization wishes to keep separators as valid words, we insert dummy "separators" between valid words.

to the second canonical word, "`blue`". Once the presentation layer is built, the iCSA structure is constructed over the sequence *Sid*.

In the presentation layer, bitmap CT keeps a compressed representation of the presentation aspect of the text. Based on the alternation between words and separators, CT will have a codeword belonging to a word, followed by the codeword of a separator, and so on. As an example, in Fig. 1, we can observe that $CT[1..3] =$'001' is the codeword associated to the separator "`The `" and $CT[4] =$'1' is the codeword of the variant "`Blue`" of the canonical word "`blue`". Those codewords are obtained as follows. On the one hand, separators are assigned a codeword using a word-based Huffman's algorithm [19, 15] over the whole vocabulary of separators (storing the shape of that tree requires little overhead using canonical Huffman [20]). On the other hand, the variants of each canonical word (that are also kept sorted by frequency) are also encoded with the same method. Therefore, along with the variants of each canonical word, the shape of the Huffman tree used to encode them has also to be known for decoding. In practice, when a canonical word has a unique variant it is actually not encoded in CT (however, in the example in Fig. 1 we used 1 bit for clarity). Together with the information on canonical words provided by *Sid* (which is not explicitly stored but obtained via iCSA), we can recreate the original text, as *Sid* indicates which Huffman tree to access when decoding words from CT.

To enable decoding from any random word position in the text we provide synchronism the codewords of CT, by using a vector B. Given a position i in *Sid*, $B[i] = p$ tells the offset in CT from which the corresponding variant of the canonical word $j = Sid[i]$ can be decoded (using the Huffman tree associated to the j^{th} canonical word). After decoding one symbol from that point p in CT, we will find the beginning of the codeword of a separator, and after it the codeword of the variant of the canonical word in $Sid[i + 1]$, and so on. In our example, we can see that $B[5] = 16$ is the beginning in CT of the codeword '01' that corresponds to the third ($Sid[5] = 3$) canonical word ('01' $\rightarrow$ "`COLOR`"). Then, $CT[18, 19] =$'01' is the codeword of the separator "` of the `".

A second array, Z, is needed for *locate* and *display* operations. It maps any position i from vector *Sid* to its actual byte offset in the original text T: $Z[i] = j$ means that $T[j]$ is the first character of the word represented by $Sid[i]$.

To save space, both B and Z are sampled at regular positions $i \cdot k_b$ and $i \cdot k_z$, respectively, and only those positions are stored. A non-sampled value p from B ($ik_b < p < i(k_b + 1)$) is obtained by moving to position $B[i \cdot k_b]$ in CT and then decoding alternatively $p - (ik_b)$ words and separators. The number of decoded bits from CT added to the value $B[i \cdot k_b]$ tells us the value of $B[p]$. A non-sampled value p from Z is obtained similarly by adding to the previous sampled value $Z[ik_z]$ the number of characters decoded after processing $p - ik_z$ words and $p - ik_z$ separators. In this case decoding starts at position $B[ik_z]$ of CT.

For lack of space we omit the detailed structures of the presentation layer and the details of the search operations on FWCSA: *countWords*, *locateWords*, and *extractSnippet*.

5 Experimental results

We used a large text collection with 1023MiB, obtained by aggregating several corpora from TREC-2: AP Newswire 1988 (AP) and Ziff Data 1989-1990 (ZIFF), as well as from TREC-4: Congressional Record 1993 (CR) and Financial Times 1991 to 1994, and finally Calgary corpus[6]. An isolated Intel®Pentium®-IV 3.00 GHz, with 4 GB RAM was used. It ran Debian (kernel 2.4.27), using gcc version 3.3.5 with `-O9` optimizations. Time results measure CPU user time.

We compared our self-indexes WCSA and FWCSA against two in-memory block-addressing inverted indexes (II and FII) with similar features[7]. II is the same index from authors in [8] and FII is its *Flexible* counterpart. Therefore, the text is compressed with ETDC [9]; whereas postings are encoded differentially with ETDC and absolute samples are kept every k values to speed-up intersections. This approach differs only slightly from that in [10], and obtains similar results in practice. The normalization process of FWCSA and FII consisted of: (1) choosing as valid words maximal alphanumeric sequences, (2) skipping separators and stopwords, (3) folding to lowercase.

We measured *locateWords* time and also the time needed to extract a snippet containing 20-words around all the occurrences of a given pattern. We used 100 test patterns from 4 different groups of single-word patterns (with different frequency ranges) and also 4 groups of phrase-patterns composed of 2, 4, 6, and 8 words. Results for both *locateWords* and for *snippet* extraction refer to average time per occurrence (in msec/occurrence).

WCSA vs II. We consider two configurations varying the memory usage of the indexes. We used two setups of WCSA depending on the parameters of its iCSA layer; that is, on the sampling periods for its structures: $\{t_\psi, t_A, t_{A^{-1}}\}$. One, named WCSA$_1$, used $\{t_\psi, t_A, t_{A^{-1}}\} = \{16, 16, 64\}$; the other, WCSA$_2$, was set to $\{32, 32, 64\}$. For II, two parameters are needed, $\{k, b\}$, that refer to the sampling period to index its compressed postings lists, and the block size (in Kbytes). We call II$_1$ the setup $\{k, b\} = \{8, 16\}$, and II$_2$ to $\{k, b\} = \{32, 256\}$.

Table 1 shows that WCSA$_2$ overcomes II$_2$ in all aspects. In practice, when little memory is available the WCSA is clearly the best choice. Only when we use more memory, II$_1$ can compete with WCSA$_1$ in the extraction of snippets for either single-word patterns or short phrases. However, WCSA$_1$ is still faster than II$_1$ for locating. When we search for phrase patterns, the performance gaps between WCSA and II increase with the number of words in the phrase.

FWCSA vs FII. We used three setups of FWCSA using fixed values $B = 32$ and $Z = 512$ (presentation layer) and depending on the three sampling

[6] `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus`

[7] Some freely available inverted indexes were checked, but they either: *i)* used a different retrieval model than that of (F)WCSA, for example *Zettair* (retrieves passages, http://www.seg.rmit.edu.au/zettair/), *Wumpus* (needs the text separately, http://www.wumpus-search.org), and *Lemur* (ranked document retrieval, http://www.lemurproject.org); or *ii)* were not ready, or not public, or we could not install them, such as *Galago* (http://www.galagosearch.org/ [30]), those in [10] and [29], and *Terrier* (http://ir.dcs.gla.ac.uk/terrier/).

Table 1. Results comparing WCSA against II and FWCSA against FII.

		WCSA$_i$		II$_i$		FWCSA$_i$			FII$_i$		
		i=1	i=2	i=1	i=2	i=1	i=2	i=3	i=1	i=2	i=3
	Ratio (%)	45.03	38.08	45.54	39.07	41.42	38.84	37.54	41.32	38.93	37.50
		Locate				*Locate*					
Words	1-100	0.009	0.018	0.018	0.246	0.030	0.058	0.070	0.042	0.161	0.503
(freq	101-1000	0.007	0.019	0.019	0.237	0.030	0.059	0.070	0.019	0.074	0.200
range)	1001-10000	0.006	0.019	0.023	0.163	0.030	0.058	0.069	0.021	0.089	0.171
	10000+	0.006	0.019	0.014	0.029	0.028	0.057	0.067	0.011	0.020	0.022
phrases	2	0.005	0.014	0.028	0.113	0.027	0.054	0.063	0.044	0.118	0.159
	4	0.005	0.009	1.128	3.737	0.030	0.058	0.069	0.026	0.064	0.089
#words	6	0.069	0.069	14.028	76.319	0.032	0.062	0.074	0.077	0.304	0.485
	8	0.059	0.059	7.396	50.118	0.044	0.059	0.074	3.086	15.551	27.795
		Snippet				*Snippet*					
Words	1-100	0.055	0.091	0.027	0.255	0.086	0.148	0.160	0.041	0.161	0.512
(freq	101-1000	0.053	0.083	0.021	0.238	0.087	0.151	0.161	0.022	0.078	0.204
range)	1001-10000	0.054	0.084	0.024	0.164	0.085	0.149	0.159	0.024	0.093	0.174
	10000+	0.054	0.084	0.015	0.030	0.083	0.145	0.155	0.014	0.023	0.025
phrases	2	0.046	0.070	0.028	0.114	0.078	0.139	0.148	0.047	0.121	0.163
	4	0.029	0.043	1.130	3.737	0.085	0.148	0.158	0.029	0.067	0.092
#words	6	0.069	0.139	14.028	76.389	0.092	0.159	0.170	0.080	0.307	0.486
	8	0.118	0.118	7.396	50.059	0.084	0.153	0.162	3.110	15.463	27.717

parameters of its iCSA. The first, named FWCSA$_1$, used the values $\{16, 16, 32\}$; FWCSA$_2$ was obtained by setting $\{32, 16, 64\}$; and FWCSA$_3$ used the values $\{32, 32, 64\}$. For FII, $\{k, b\}$ were set to {64,16} to obtain FII$_1$; FII$_2$ was created with the values $\{64, 128\}$, and finally, II$_3$ used values $\{64, 1024\}$.

The results show that, when compression ratio is around 40% there is not a clear winner. FWCSA is better than FII for dealing with long phrases, but FII obtains the best results on high frequency words. However, as the amount of memory decreases, the results of FII worsen much faster than those of FWCSA.

Moreover, it is noticeable that II and FII versions are lower bounded in size by around 35%. However, with that amount of available memory (F)WCSA performs much better. Furthermore, we can always set (F)WCSA space to around 30%, yet with a clear loss in performance.

Non-II Alternatives. Other competitors to the inverted index, in a spirit similar to our WCSA, have recently appeared in the literature. We briefly compare with these in this section.

We first compare WCSA$_1$ and WCSA$_2$ with the wavelet-tree index on words (WT) [8]. We used two configurations of WT with different memory usage. WT$_1$ occupies 44.37% of the original text, whereas WT$_2$ uses 38.61%. Results in Table 2 show that, to search for single-word patterns, WT$_1$ is faster than WCSA$_1$. However, WCSA$_1$ overcomes WT$_1$ when locating phrases. Similar results are obtained for WT$_2$ versus WCSA$_2$.

We also briefly compared our WCSA against the approach called TH+AFFM in recent work [11] (word-based compression followed by character-wise self-indexing). We consider locating of phrases composed of 4 words (other choices give similar results). We adjust WCSA to work with the same memory of TH+AFFM, for different parameter combinations of both methods. It turns out that WCSA searches around 5 times faster in all cases.

Table 2. Results comparing WCSA against WT.

		WCSA_i		WT_i		WCSA_i		WT_i	
		$i = 1$	$i = 2$	$i = 1$	$i = 2$	$i = 1$	$i = 2$	$i = 1$	$i = 2$
	Ratio (%)	45.03	38.08	44.37	38.61	45.03	38.08	44.37	38.61
		Locate				*Snippet*			
Words	1-100	0.009	0.018	0.005	0.007	0.055	0.091	0.026	0.058
(freq	101-1000	0.007	0.019	0.004	0.044	0.053	0.083	0.025	0.093
range)	1001-10000	0.006	0.019	0.002	0.007	0.054	0.084	0.021	0.051
	10000+	0.006	0.019	0.002	0.003	0.054	0.084	0.019	0.045
phrases	2	0.005	0.014	0.010	0.021	0.046	0.070	0.025	0.058
	4	0.005	0.009	0.458	0.926	0.029	0.043	0.476	0.958
#words	6	0.069	0.069	9.028	21.181	0.069	0.139	9.028	21.250
	8	0.059	0.059	5.562	15.562	0.118	0.118	5.621	15.562

6 Conclusions and Future Work

We have shown that a self-index applied to natural language text, seen as a sequence of words rather than symbols, offers a very relevant alternative to the traditional inverted indexes. With sizes around 40% the inverted indexes can still compete with our (F)WCSA in some operations (extraction of snippets), but when we aim at using less space, our proposal performs much better.

In this work we have focused on one self-index, Sadakane's CSA. We plan to try out others that have mild dependence on the alphabet size. In particular, adapting the LZ-index [23, 2] should offer fast locating of occurrences.

Inverted indexes are also used for other purposes, as explained [4]. For example they are used to implement the tf-idf model by recording the number of occurrences of each word in each document, in decreasing order of frequency. Only a short prefix of the posting list is fetched to solve queries. Can we provide similar functionalities with a self-index? Some initial advances have been made by Sadakane [27], but we are far from a definitive answer.

References

1. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
2. D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proc. 17th CPM*, LNCS 4009, pages 319–330, 2006.
3. R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *J. of the American Society for Information Science*, 51(1):69–82, 2000.
4. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.
5. R. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proc. 12th SPIRE*, pages 13–24, 2005.
6. J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In *Proc. 5th WEA*, pages 146–157, 2006.
7. T. Bell, J. Cleary, and I. Witten. *Text compression.* Prentice Hall, 1990.
8. N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st ACM SIGIR.* ACM Press, 2008. To appear.

9. N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.
10. J. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. 14th SPIRE*, pages 137–148, 2007.
11. A. Fariña, G. Navarro, and J. Paramá. Word-based statistical compressors as natural language compression boosters. In *Proc. 18th DCC*, pages 162–171, 2008.
12. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.
13. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th ACM-SIAM SODA*, pages 841–850, 2003.
14. H. Heaps. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press, NY, 1978.
15. D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E.*, 40(9):1090–1101, 1952.
16. G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th FOCS*, pages 549–554, 1989.
17. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
18. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
19. A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.
20. A. Moffat and J. Katajainen. In-place calculation of minimum-redundancy codes. In *Proc. 4th WADS*, LNCS 955, pages 393–402, 1995.
21. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.
22. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS v. 1180, pages 37–42, 1996.
23. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.
24. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
25. G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.
26. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
27. K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms (JDA)*, 5(1):12–22, 2007.
28. P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. 9th ALENEX*, 2007.
29. P. Sanders and F. Transier. Compressed inverted indexes for in-memory search engines. In *Proc. 10th ALENEX*, 2008.
30. T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. 30th ACM SIGIR*, pages 175–182. ACM Press, 2007.
31. P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.
32. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers, second edition, 1999.