Smaller Self-Indexes for Natural Language*

Nieves R. Brisaboa[1] , Gonzalo Navarro[2], and Alberto Ordóñez[1]

[1] Database Lab., Univ. of A Coruña, Spain. {brisaboa,alberto.ordonez}@udc.es
[2] Dept. of Computer Science, Univ. of Chile, Chile. gnavarro@dcc.uchile.cl

Abstract. Self-indexes for natural-language texts, where these are regarded as token (word or separator) sequences, achieve very attractive space and search time. However, they suffer from a space penalty due to their large vocabulary. In this paper we show that by replacing the Huffman encoding they implicitly use by the slightly weaker Hu-Tucker encoding, which respects the lexical order of the vocabulary, both their space and time are improved.

1 Introduction

Self-indexing [6, 11, 18] is a technique to represent a sequence in compressed form and offer direct access to any portion of the sequence as well as pattern searches on it. They emerged as alternatives to suffix arrays [17], which require several times the text size. Compared to classical solutions like compressed inverted indexes [20], suffix arrays and self-indexes have the important advantage of working on any sequence of symbols, not only on (Western) natural language texts, so they also support indexed searches on genomic and protein sequences, music sequences, Oriental language texts, source code repositories, and so on.

Interestingly, self-indexes also offer improvements on natural language indexing [5]. The key idea is to regard the text collection as a sequence of words (and separators between words), so that pattern searches correspond to word and phrase searches over the text collection. Regarding words as symbols yields much better compression ratios than considering characters, so that the index represents the text within 30%–35% of its original size and in addition offers fast searches on it. In exchange, this index must handle a large alphabet, thus the impact of data structures that are proportional to the alphabet size is not anymore negligible.

In this paper we study a new representation for self-indexes over large alphabets. Inspired by a theoretical result [2] on representing permutations, we replace the Huffman encoding [13] underlying many self-indexes by a Hu-Tucker encoding [12]. This is slightly suboptimal but it does not alter the vocabulary ordering, thus avoiding the need to store the reordering that Huffman encoding carries out. As a result, we show that we reduce both the space and the time of word-based self-indexes.

* Funded by the Spanish MICINN (PGE and FEDER) refs. TIN2009-14560-C03-0, MICINN ref. AP2010-6038 (FPU Program) for Alberto Ordóñez, and Fondecyt Grant 1-110066, Chile for Gonzalo Navarro.

2 Self-Indexes

Let $T[1,n]$ be a sequence of symbols over alphabet Σ of size σ, terminated with a special symbol $T[n] = \$ \in \Sigma$, smaller than all the others in Σ. A *full-text index* is a data structure built on T. Given a *search pattern* $P[1,m]$, the full-text index usually supports the operation *count*, which tells the number of occurrences of P in T, and *locate*, which gives the positions where P occurs in T. A classical full-text index is the suffix array [17], which counts in time $O(m \log n)$ (and some variants in $O(m + \log n)$), and after counting it locates each occurrence in constant time. A disadvantage of the suffix array is that it uses $O(n \log n)$ bits, which is much more than the $n \log \sigma$ bits needed to represent T.

A *self-index* is a data structure that *represents* a text T and in addition supports the above search operations. It represents T via operation *extract*, which retrieves any desired text substring. There exist various self-indexes [18], most of which represent T within the same space a compressor would achieve, yet still support efficient searches. In this paper we focus on the *FM-Index* family [6].

The suffix array $A[1,n]$ of $T[1,n]$ is a permutation $[1,n]$ of all the suffixes $T[i,n]$ so that $T[A[i],n] \prec T[A[i+1],n]$ for all $1 \leq i < n$, being $\prec$ the lexicographic order, where $a \prec b$ means that a precedes b in the lexicographic order. Since all the suffixes starting with a pattern $P[1,m]$ are contiguous in A, we can find the occurrences of the pattern in the text in $O(m \log n)$ time via two binary searches for the first and last suffix starting with P. Once the corresponding interval $A[sp,ep]$ is identified, we know that P occurs $ep - sp + 1$ times in T, and we can list its occurrences $A[i]$, $sp \leq i \leq ep$.

The Burrows-Wheeler Transform (BWT) [3] of T is a reversible transformation $T_{bwt}[1,n]$ such that $T_{bwt}[i] = T[A[i]-1]$, except when $A[i] = 1$, where $T_{bwt}[i] = T[n] = \$$. The BWT consists of a reordering of the characters of T. Given a position j, if we know that $T[j]$ corresponds to $T_{bwt}[i]$, we can know where is $T[j-1]$ via an operation called the *LF-mapping*: $LF(i) = A^{-1}[A[i]-1]$ (except that $LF(i) = A^{-1}[n]$ if $A[i] = 1$). As shown by Ferragina and Manzini [6], $LF(i)$ can be obtained as follows: let $C(c)$ be the number of occurrences of symbols $< c$ in T. Then, it holds $LF(i) = C(c) + rank_c(T_{bwt}, i)$, where $c = T_{bwt}[i]$ and $rank_c(S,i)$ is the number of occurrences of c in $S[1,i]$.

The FM-Index [6] family of self-indexes is based on representing C and T_{bwt}, the latter with $rank_c$ capabilities. The *locate* and *extract* functionality is provided via the LF function together with appropriate samplings of the text, which are not crucial for this paper. To search for $P[1,m]$, the FM-index uses a technique called *backward search*, where the characters of P are considered in reverse order. Let $A[sp_{i+1}, ep_{i+1}]$ be the suffixes starting with $P[i+1,m]$ (initially $[sp_m, ep_m] = [1,n]$). Then it holds $sp_i = C(P[i]) + rank_{P[i]}(T_{bwt}, sp_{i+1} - 1) + 1$ and $ep_i = C(P[i]) + rank_{P[i]}(T_{bwt}, ep_{i+1})$. The final answer is $A[sp,ep] = A[sp_1, ep_1]$. Thus *count* takes the time of $O(m)$ $rank_c$ operations.

The FM-Indexes mainly differ in how T_{bwt} is represented [18]. The modern variants [7] represent $S = T_{bwt}$ using a *wavelet tree* [9]. This is a binary tree with σ leaves, each representing a symbol of Σ. The root represents $S[1,n]$, and divides the alphabet into Σ_1 and Σ_2. A bitmap $B[1,n]$ is stored at the root, so

that $B[i] = 0$ iff $S[i] \in \Sigma_1$. The children of the root represent the complementary subsequences S_1 and S_2 of S formed by the symbols of Σ_1 and Σ_2, respectively, and are built recursively. To access $S[i]$ we examine $B[i]$ at the root. If it is a 0, we continue recursively on the left child with $i{=}rank_0(B, i)$; otherwise we continue on the right with $i{=}rank_1(B, i)$. When we arrive at a leaf representing symbol $c \in \Sigma$, we know $S[i]{=}c$. We can also compute $rank_c(S, i)$. We start at the root and, if $c \in \Sigma_1$, we descend to the left child with $i{=}rank_0(B, i)$; else to the right with $i{=}rank_1(B, i)$. When we arrive at leaf c, the answer is the current i value. We use representations of B that support $rank$ in $O(1)$ time [14, 19].

The space required by the wavelet tree is adequately described with the notion of *empirical entropy*. Measure $nH_0(T)$ is a lower bound to the output size of a statistical semi-static compressor applied on T that encodes each symbol of Σ always in the same way. Measure $nH_k(T)$ is similar but it allows codes to depend on the k characters that follow in T the one to be encoded. It holds $H_k(T) \leq H_{k-1}(T) \leq H_0(T) \leq \log \sigma$ for any k.

We enumerate now the wavelet tree encodings that are competitive for large alphabets. By using a balanced wavelet tree and an uncompressed bitmap representation, the FM-index requires $n \log \sigma + o(n \log \sigma)$ bits of space. If we instead use a particular bitmap representation that compresses them to H_0 space [19], the total space is $nH_k(T) + o(n \log \sigma)$ [16], for any $k \leq \alpha \log_\sigma n$ and constant $\alpha < 1$. In all these cases the operations require $O(\log \sigma)$ time, thus for example counting requires time $O(m \log \sigma)$. By giving the wavelet trees the shape of the Huffman tree for the frequencies in T, the space turns out to be $n(H_0(T)+1)(1+o(1))+O(\sigma \log n)$ bits [4], the last term to represent the model, and the average access and $rank_c$ time drops to $O(1 + H_0(T))$ if positions are probed uniformly at random. If in addition one uses compressed bitmaps [19], the space becomes $nH_k(T) + o(n(1 + H_k(T)) + O(\sigma \log n)$. Finally, a recent so-called "alphabet partitioning" representation achieves $nH_0(T) + o(n(1 + H_0(T))$ bits and $O(\log \log \sigma)$ operation time [1].

Our aim in this paper is to reduce the impact of the $O(\sigma \log n)$ term, which is significant for large alphabets.

3 Huffman versus Hu-Tucker-Shaped Wavelet Trees

We describe our implementation of Huffman-shaped wavelet trees, and then our new variant, Hu-Tucker shaped ones.

3.1 Huffman-Shaped Wavelet Trees

We give the wavelet tree the shape of the Huffman tree of the word frequencies. The total number of bits stored is less than $n(H_0(T)+1)$, where T is the sequence of (word and separator) tokens forming the text collection. We concatenate all the bitmaps and create a unique *rank*-capable structure with the concatenation. The wavelet tree internal nodes store pointers to this concatenation, indicating

where their own bitmap starts. Such pointers use $\log(n(H_0(T)+1))$ bits. The tree is allocated in an array of $2V-1$ nodes, so tree pointers use $\log(2V)$ bits.

We must also spend $2V \log V$ bits to encode the permutation π of words induced by Huffman coding, and its inverse π^{-1}. To access $T^{bwt}[i]$, we traverse the wavelet tree until reaching a leaf. At this point, we can know the sum of all the leaf sizes to the left of the current leaf: it is a matter of accumulating the 0s to the left of the current position each time we go right. To convert this position into a leaf rank, that is, to know how many leaves are there to the left of the one we arrived at, we store an array $D[1,V]$. This is identical to C, but considers the cumulative word frequencies in the order given by π. A binary search on D tells the leaf number r corresponding to the position arrived at. Then, $\pi^{-1}(r)$ gives the actual word identifier. Array D requires $V \log n$ bits.

To compute $rank_c(T_{bwt}, i)$, we use $d = \pi(c)$ to convert it into a leaf number, and then traverse the wavelet tree towards that leaf. Array D can be used to guide the search: if $D[d] < rank_0(B, n)$, where B is the bitmap root, then d is to the left, else to the right. The criterion inside the descendant nodes is similar.

The total space is at most $n(H_0(T)+1)(1+o(1))$ for the bits. Related to the vocabulary, we spend $V(\log(n(H_0(T)+1))+2\log(2V)+2\log V+2\log n)$ bits for the pointers to bitmaps, tree pointers, permutations, C and D, respectively. This is $n(H_0(T)+1)(1+o(1))+V(3\log n+4\log V+O(\log\log V))$, since $H_0(T) \leq \log V$.

3.2 Hu-Tucker-Shaped Wavelet Trees

Based on the idea of Barbay and Navarro [2], we use a Hu-Tucker encoding [12] (see also Knuth [15, p. 446]) instead of Huffman. The Hu-Tucker algorithm produces an optimal prefix-free code from a sequence of frequencies $X = \langle x_1, \ldots, x_V \rangle$ such that: (1) the i-th lexicographically smallest code is for the i-th symbol and; (2) if l_i is the length associated to the i-th run, then $\sum l_i n_i$ is minimal, and upper bounded by $n(H_0(X)+2)$.

Since the leaves of the Hu-Tucker-shaped wavelet tree are in alphabetic order, it is not necessary to store π nor π^{-1}. Furthermore, we do not need to store D, as it is identical to C. Thus the space becomes at most $n(H_0(T)+2)(1+o(1))+V(2\log n+2\log V+O(\log\log V))$. That is, we have replaced $V\log n+2V\log V$ bits by n further bits in the encoding. However, in practice the difference between Huffman and Hu-Tucker codes is much less than one bit per symbol. We note that Hu-Tucker shaped wavelet trees have been studied in other scenarios [10].

4 Experimental Evaluation

We compare several wavelet tree encodings that are competitive for large alphabets, on the task of implementing an FM-index on words. The wavelet trees use either plain or compressed bitmaps. For plain bitmaps we used a simple 1-level *rank* implementation [8] of Jacobson's solution [14], and for compressed bitmaps we used a simple 1-level *rank* implementation [4] of Raman et al.'s solution [19]. We consider a balanced wavelet tree with compressed

Name	Size (MB)	Words (n)	Voc. (V)	H_0	gzip fast	gzip best	bzip2 fast	bzip2 best
ZIFF1	158.89	39,395,522	212,195	9.74	39.69%	33.02%	29.68%	25.14%
AP	254.20	61,281,811	250,592	9.96	43.27%	37.39%	33.39%	27.41%
FR	259.72	66,753,342	227,241	9.31	32.32%	25.68%	23.66%	20.06%
DOE	183.81	41,912,456	241,124	9.68	40.19%	33.44%	29.93%	25.44%

Table 1. Collection statistics and compressibility.

bitmaps (Balanced-WT-RRR, achieving $nH_k(T) + o(n \log V)$ bits [16] as no pointers are used), a Huffman-shaped wavelet tree with plain bitmaps (HWT-PLAIN, achieving $n(H_0(T)+1)(1+o(1))+O(V \log n)$ bits) and with compressed bitmaps (HWT-RRR, achieving $nH_k(T) + o(n(H_0(T)) + 1) + O(V \log n)$ bits), a Hu-Tucker-shaped wavelet tree with plain bitmaps (HTWT-PLAIN, achieving $n(H_0(T)+2)(1+o(1))+O(V \log n)$ bits) and with compressed bitmaps (HTWT-RRR, achieving $nH_k(T)+o(n(H_0(T))+1)+O(V \log n)$ bits), and an "alphabet partitioned" representation [1] (A-partition, achieving $nH_0(T)+o(n(H_0(T)+1))$ bits). As a control value, we introduce in the comparison an existing FM-index for words: the WSSA [5], using zero space for samplings.

To achieve different space/time trade-offs, we use samplings $\{32, 64, 128, 180\}$ for bitmaps. We test the different indexes using collections *ZIFF1*, *AP*, *FR*, and *DOE* taken from TREC (`http://trec.nist.gov/data.html`). Table 1 gives some statistics and information on compressibility of each collection, in terms of space achieved by well-known compressors like *gzip* and *bzip2*.

We used an isolated Intel®Xeon®-E5335@2.00GHz with 16 GB RAM, running Ubuntu 9.10 (kernel 2.6.32-39-server). We used gcc version 4.4.3 with `-O9` options. Time results refer to CPU user time.

Figure 1 shows the different space/time trade-offs achieved, for the process of counting the occurrences of a phrase of 4 words. It can be seen that our variant HTWT-RRR dominates most of the space/time map, and it also clearly surpasses the best compressors. The only competitive alternative, using much more space, is again our HTWT-PLAIN, and sometimes, using even more space, A-partition. In particular, each HTWT variant is smaller (and slightly faster) than its corresponding HWT version. The WSSA is not competitive.

References

1. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. In: ISAAC. pp. 315–326 (part II) (2010)
2. Barbay, J., Navarro, G.: Compressed representations of permutations, and applications. In: STACS. pp. 111–122 (2009)
3. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation (1994)
4. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: In: SPIRE. pp. 176–187 (2008)
5. Fariña, A., Brisaboa, N., Navarro, G., Claude, F., Places, A., Rodríguez, E.: Word-based self-indexes for natural language text. ACM Trans. Inf. Sys. 30(1), 1–34 (2012)

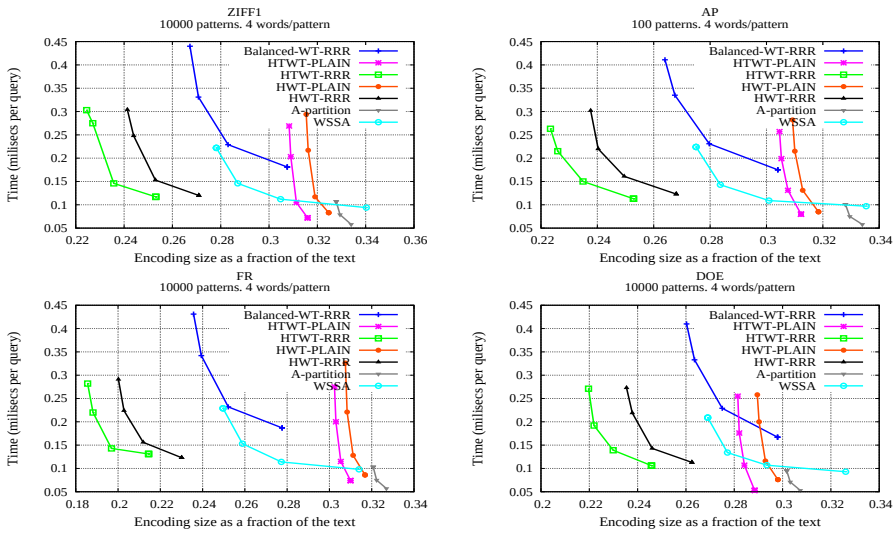

Fig. 1. Space/time trade-off for *count* queries.

6. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: FOCS. pp. 390–398 (2000)
7. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), art. 20 (2007)
8. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: WEA (posters). pp. 27–38 (2005)
9. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: SODA. pp. 841–850 (2003)
10. Grossi, R., Vitter, J., Xu, B.: Wavelet trees: From theory to practice. In: CCP. pp. 210–221 (2011)
11. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: STOC. pp. 397–406 (2000)
12. Hu, T.C., Tucker, A.C.: Optimal computer search trees and variable-length alphabetical codes. SIAM J. Appl. Math. 21(4), 514–532 (1971)
13. Huffman, D.A.: A method for the construction of minimum-redundancy codes. In: Proc. I.R.E. vol. 40, pp. 1098–1101 (1952)
14. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS. pp. 549–554 (1989)
15. Knuth, D.E.: The Art of Computer Programming. Vol. 3: Sorting and Searching. Addison-Wesley, 2nd edn. (1998)
16. Mäkinen, V., Navarro, G.: Implicit compression boosting with applications to self-indexing. In: SPIRE. pp. 214–226 (2007)
17. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Comp. 22(5), 935–948 (1993)
18. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), art. 2 (2007)
19. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA. pp. 233–242 (2002)
20. Witten, I.H., Moffat, A., Bell, T.C.: Managing gigabytes: compressing and indexing documents and images. Morgan Kaufmann, 2nd edn. (1999)