A New Approach for Document Indexing Using Wavelet Trees *

Nieves R. Brisaboa, Yolanda Cillero, Antonio Fariña, Susana Ladra, Oscar Pedreira
Database Laboratory, University of A Coruña
Campus de Elviña, 15071 A Coruña, Spain
{*brisaboa, ycillero, fari, sladra, opedreira*}*@udc.es*

Abstract

The development of applications that manage large text collections needs indexing methods which allow efficient retrieval over text. Several indexes have been proposed which try to reach a good trade-off between the space needed to store both the text and the index, and its search efficiency. Self-indexes are becoming more and more popular in the last years. Not only they index the text, but they keep enough information to recover any portion of it without the need of keeping it explicitly. Therefore, they actually replace the text.

In this paper, we focus in a self-index known as wavelet tree. Being originally organized as a binary tree, it was designed to index the characters from a text. We present three variants of this method that aim at reducing its size, while keeping a good trade-off between space and performance, as well as making it well-suited for indexing natural language texts. The first approach we describe joins Huffman compression and wavelet trees. The other two new variants index words instead of characters and use two different word-based compressors.

1. Introduction

The amount of digitally available information has experienced a huge growth during the last years. In many cases this information consists of text, or can be represented as text, as it happens with music, signals, time series, biological sequences or multimedia streams [11]. The need for efficient access to this information has led to the development of efficient indexing and searching techniques that can be applied to any kind of data expressed as text. We focus on the case of text indexing throughout the rest of the paper.

*This work has been partially supported by "Ministerio de Educación y Ciencia" (PGE y FEDER) ref. TIN2006-16071-C03-03, by "Xunta de Galicia" ref. PGIDIT05SIN10502PR and ref. 2006/4, and by "Ministerio de Educación y Ciencia" ref. AP-2006-03214 (FPU Program).

Traditionally, inverted indexes [12] are considered a classical text indexing technique. An inverted index associates to each term in a document a list of pointers to the location of its occurrences. This permits to retrieve all the occurrences of a term or phrase very efficiently. Their main drawback is the large amount of space needed, which can be up to four times the size of the original text, in addition to the space needed to store the text itself, since it is too difficult to reproduce it from the index.

A first approach to reduce space requirements is based on compression. It is well-known [7] that classical Huffman compression reduces natural language texts to around 60% of their original size. The brilliant idea in [9] of using Huffman but encoding words instead of characters led to compression ratios around 25-30% (because words present a more biased distribution of frequencies than characters) and gave the key to join compression and inverted indexes (as words are the atoms in both cases). Joining word-based compression and block addressing inverted indexes [10, 13] reduced not only their size, but also improved notably their efficiency.

Self-indexes present a different approach to reduce the space needed by an index. A compressed index takes advantage of the compressibility of the text and permits to work with the text in a compressed form. Thus, it requires space proportional to that of the compressed text (around two times the size of the compressed text). A self-index is a compressed index which avoids the need of keeping the text along with the index. It contains enough information to reproduce any part of the text from the index, therefore it replaces the text. A wavelet tree [5] is a self-index organized as a binary tree, designed to index the characters of the text. As we show later, we can take advantage of various coding schemes to reduce the size of a wavelet tree and improve the performance of the search operation.

In this paper we present three variants of the original wavelet tree. All of them are based on the idea of building the tree from the codes associated to each character/word from the source text obtained with different coding schemes for text compression. Our techniques aim at reducing the

size of the wavelet tree and improving the performance of the search operation. The paper is organized as follows. Next section describes the coding schemes used as the basis for building our wavelet trees. Section 3 describes indexing and searching with the original wavelet tree. Section 4 presents how these coding techniques can be used to improve the performance of the index and to reduce the space needed. Finally, Section 5 presents our conclusions and future work.

2. Coding schemes for text compression

In this section we briefly describe the coding schemes for text compression used as the base for the wavelet tree improvements presented in next section, particularly Huffman compression [7, 9, 10] and ETDC [3, 1, 2].

Huffman compression is perhaps the most famous compression technique since its appearance in [7]. This method replaces each character by a binary code. The more frequent the character, the smaller the code assigned to it. The codes assigned to each character are obtained building the Huffman tree. The different characters from the text (vocabulary), which constitute the leafs, are firstly ordered by frequency. Then the two nodes with smaller frequency are chosen to form a parent node whose frequency is set as the sum of the frequency of the two child nodes. This process is repeated until reaching the root of the tree. Finally, left branches are labeled with '0', and right branches with '1'. The code assigned to each character is the obtained in the traversal from the root to its leaf node. Figure 1 shows an example of a Huffman tree built from a given text.

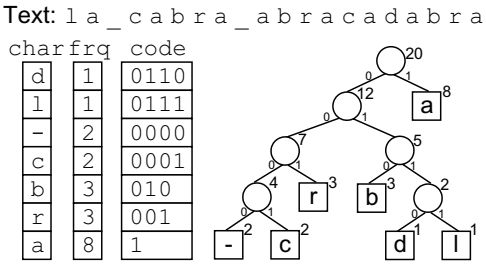

Figure 1. Huffman tree construction

When using characters as the elements of the vocabulary, the compression ratios obtained are around the 60% of the size of the original text. When using words as the elements of the vocabulary, the compression ratios are around 25% of the size of the text. This improvement is due to the more biased distribution of the words of the text [9].

End-Tagged Dense Code is a compression method that assigns byte codes (this is, sequences of bytes instead of bits as Huffman does) to words. The last byte of each code is used as an "end-tag" by marking its first bit to 1. The other bytes of the code have a 0 for this bit. As it happens in Huffman compression, the distinct words are ordered by frequency. Then the first 128 words are associated one-byte codes, from **1**0000000 to **1**1111111. Words from position 2^7 to $2^72^7 + 2^7 - 1$ are given two-byte codes, the first byte taking values between **0**0000000 to **0**1111111 and the second from **1**0000000 to **1**1111111. The process is repeated with the rest of the words of the vocabulary, using as many bytes as necessary. The main advantages of this method are: (i) good compression ratio, (ii) simple coding, and (iii) the possibility of decompressing the text from any position and to do searches directly in the compressed text.

3. Text indexing using wavelet trees

A wavelet tree, as originally proposed in [5], is a self-index organized as a binary tree where each symbol from an alphabet $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$ is associated to a leaf node. As a self-index it implements three main operations: *count* (which returns the number of occurrences of a symbol), *locate* (which returns the position of a given occurrence of a symbol) and *display* (which returns the symbol at any position of the text). In this section we describe how to build the index and how to use it for searching.

3.1. Indexing

Given a text $T = T_1 \ldots T_n$ composed of symbols from an alphabet $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$, a wavelet tree is built as follows. The root of the tree is given a bitmap $B = b_1 \ldots b_n$ of the same length of the text (n), such that $b_i = 0$ if $t_i \in \{s_1, \ldots, s_{\sigma/2}\}$, and $b_i = 1$ if $t_i \in \{s_{\sigma/2+1}, \ldots, s_n\}$. The sequence of the characters given a 1 in this vector are processed in the right child of the node, and those marked 0 are processed in the left child of the node. This process is repeated recursively in each node until reaching the leaf nodes when the sequence of indexed symbols has only one different symbol. In this way, each node indexes half the symbols (from Σ) indexed by its parent node.

Each node stores only a bitmap, and the portion of the alphabet that it covers can be obtained by following the path from the root of the tree to that node. With this information it is not necessary to store the text separately, since any part of the text can be recovered with these bitmaps. Figure 2 shows an example of a wavelet tree built from the text '`la_cabra_abracadabra`', and the alphabet $\Sigma = \{\_, a, b, c, d, l, r\}$.

3.2. Searching and decompressing

We can use this index to obtain the number of occurrences of a symbol (*count*), to search a given occurrence of

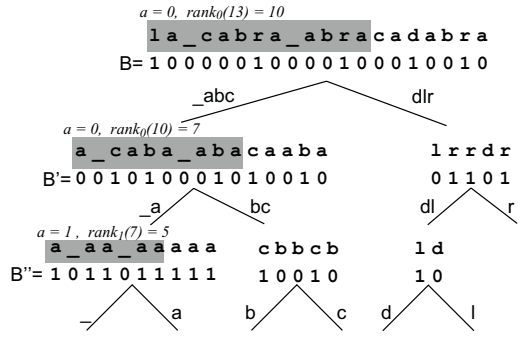

Figure 2. Text indexing with wavelet trees

a symbol (*locate*) or to obtain the symbol in any position of the text (*display*). To implement these three operations we need two basic bitmap operations: *rank* and *select*. Their efficient implementation is a topic of interest nowadays [8, 11]. Given a sequence of symbols B (for example, a sequence of bits), $rank_b(B, i) = y$ if the symbol b appears y times in the sequence $B_{1,i}$, and $select_b(B, j) = x$ if the j^{th} occurrence of the symbol b in the sequence B appears at position x. For example, given a bitmap $B = 1000110$, $rank_1(B, 5) = 2$, and $select_0(B, 4) = 7$.

Display: let us suppose we want to retrieve the character at the position i in the text. The bit B_i in the bitmap of the root tells us if this character is indexed by either the left ($B_i = 0$) or right ($B_i = 1$) child of this node. In addition, $rank_{B_i}(B, i)$ gives us the position corresponding to this character in the bitmap of the child node. This process is repeated until reaching a leaf node which gives us the desired character. As an example, let us suppose we want to know the character at position 13 from the text indexed in Figure 2[1]. Starting at the root node, $B_{13} = 0$ indicates that this character is indexed in the left branch of the root, and $rank_0(B, 13) = 10$ means that the 10^{th} position in the bitmap B' of the child node corresponds to this character. In the next level, $B'_{10} = 0$ (move to left child) and $rank_0(B', 10) = 7$ (the position in the bitmap B"). One level down, we obtain $B''_7 = 1$ (go to right child) and $rank_1(B'', 7) = 5$. Since the right child is a leaf node (corresponding to character 'a') we know that the 13^{th} position from the source text contained the 5^{th} 'a'. This process can be used to recover the complete text (one character at a time) from the information contained in the index structure.

Count and Locate: The use of the wavelet tree for searches starts at the leaf node corresponding to the character we are searching for (which is easy to locate from its position in Σ) and traverses the tree from that leaf until the root. Let us assume that a leaf node is represented by the path from the root $\overline{b_0 b_1 \dots b_k}$, and that $B, B', \dots B^k$ are the bitmaps stored in the nodes from that path. *Count* operation is obtained by just computing $rank_{b_k}(B^k, |B^k|)$. If we want to retrieve the i^{th} occurrence of a character c, whose corresponding leaf is $\overline{b_0 \dots b_k}$ (*locate*), the search process is performed as follows. By computing $i_k = select_{b_k}(B^k, i)$ we obtain that i_k is the position of the i^{th} occurrence of c in B^k. We repeat this process (one level up) obtaining $i_{k-1} = select_{b_{k-1}}(B^{k-1}, i_k)$ to move to the next level in the tree and so on until reaching the root. The last $i_0 = select_{b_0}(B, i_1)$ give us the position of the i^{th} occurrence of c in the text.

For example, if we want to search the 4^{th} occurrence of 'a' in the example of Figure 2, the search will start at the leaf $\overline{001}$, which corresponds to this symbol. In this leaf node we compute $i_2 = select_1(B'', 4) = 6$ and we move to the parent node. In the node $\overline{00}$ we obtain $i_1 = select_0(B', 6) = 8$ and continue to the next level. Finally, in the node $\overline{0}$ we compute $i_0 = select_0(B, 8) = 10$, which tells us that the character we are searching for appears in the 10^{th} position of the text.

4. Improving wavelet trees

In this section we present three variants of wavelet trees. The first one reduces its size by using Huffman coding, and the others permit to search for words instead of characters, which is needed for indexing natural language texts.

4.1. Joining wavelet trees and char-based Huffman codes

As shown, the original wavelet tree indexes the characters of the text, resulting in a balanced binary tree. However, it is possible to encode the characters of the text with Huffman codes and build the tree over those codes. The idea is simple, using the Huffman codes associated to each character to obtain the position of that character in the tree. With this strategy, the tree is not usually balanced, but the space needed to store it is reduced, since the average length of the paths to reach each leaf nodes will become smaller now.

The building process starts from the Huffman codes associated to each character. The root node of the tree contains a bitmap B with one bit for each character in the text. For each position i, either $B_i = 0$ or $B_i = 1$ if the first bit of the code associated to the character at position i is either 0 or 1 respectively. Those characters with $B_i = 0$ are indexed in the left branch of the tree and those with $B_i = 1$ in the right branch. In the bitmaps contained in the nodes at the next level, the same process is repeated with the second bit of the code associated to each character, and so on until reaching the leafs of the tree.

[1]Note that the text shown in the nodes is not actually stored there; it only appears in the figure for clarity reasons.

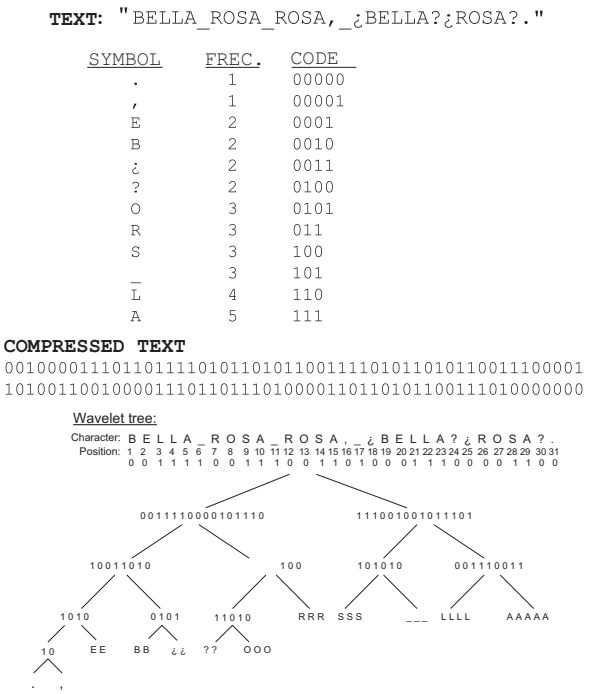

Figure 3. Wavelet tree built from Huffman codes for characters

Figure 3 shows an example that includes both the Huffman codes obtained for each character from the text and the wavelet tree built from them. In this example we can see how the Huffman code associated to each character determines its position in the tree. Searching and decompressing processes are analogous to those described in the previous section. However, instead of using the position of each character in the alphabet, the traversals of the tree are now based on the codewords associated to each character.

4.2. Joining wavelet trees and word-based Huffman codes

As shown in [9], the use of words instead of characters as the elements of the vocabulary improves significantly the compression ratio of statistic compression methods. In this case we take advantage of this fact, indexing the text with a wavelet tree built over the Huffman codes associated to each word in the text.

The first step for the index building is to process the text associating a Huffman code to each word. The wavelet tree is now built following the same process as in the previous case. The main difference is that in this case the bitmaps stored in each node of the tree make reference to codes associated to words instead of characters.

Figure 4 shows the wavelet tree built over word-based codes for the same text used in previous sections, and the

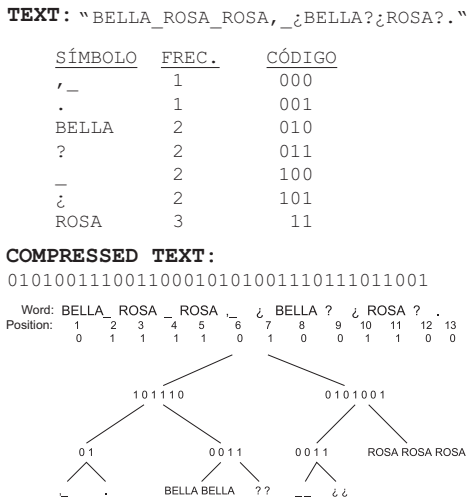

Figure 4. Wavelet tree built from Huffman codes for words

codes associated to each word of the text. The advantage of this approach is that the use of words as the elements of the alphabet instead of characters reduces the size of the bitmaps stored in each node of the tree (there are less words than characters). Thus, taking into account that the word Huffman compression is optimal [9], this wavelet tree is expected to need half the space required by the previous proposal. Another advantage of this structure is that when searching a word in the text, we only need a traversal of the tree.

The main disadvantage of this approach is that the tree becomes higher since the alphabet has now much more different symbols. The average length of the codes associated to the words is around 8-10 bits. However, for low-frequency words, the associated codes can be longer, around 20-25 bits. As a consequence, the performance of this version could be reduced due to the number of $rank$ and $select$ operations needed during each traversal of the tree. This problem led us to the idea of using byte-oriented codes for each word, which is presented next.

4.3. Joining wavelet trees and ETDC

The approach described in this section is again based on a clear parallelism between indexing and compression methods. As shown in [10, 3, 2], the use of byte-oriented instead of bit-oriented codes improves greatly the decompression and search processes, paying for it an increment of around a 5% in the compression ratio. The index structure presented here is based on this idea. In this case, the wavelet tree is built over the codes associated to each word of the text using End-Tagged Dense Code (ETDC).

Since ETDC generates codes as sequences of bytes, the wavelet tree built in this way presents several differences with the approach in previous section. Firstly, the tree is not binary. In this case we move from a node to any of its children by using the first byte of the code associated to the word, so each node can have up to 256 children. Among them, the last 128 lead to leaf nodes, and the 128 first ones lead to intermediate nodes. Thus, the first level of the wavelet tree has 128 leaf nodes, the second level has 128^2, the third 128^3, and so on. In general, when working with natural language, ETDC will never generate codes of length greater than 4 [2], so the wavelet tree will have at most four levels.

The building process is similar to that of the previous approach. However, each node contains now a byte-map instead of a map of bits. This is, the root node contains a vector B with the bytes corresponding to the first byte of the code associated to each word. In the second level, the leaf nodes correspond to words, and the non-leaf nodes contain vectors B'_i with the second byte of the code of these words whose first byte corresponds to that node; and so on until reaching the leaves of the last level.

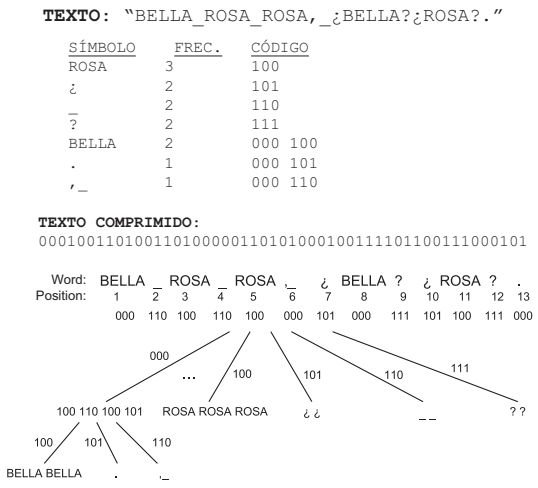

Figure 5. Wavelet tree built from ETDC codes

Figure 5 shows a wavelet tree built over ETDC codes, for the same example of the previous sections. Searches and decompression processes are also analogous, but will now use non-binary *rank* and *select* operations, which are harder to compute than the simpler binary counterparts[2].

5. Conclusions and future work

We present three variants of wavelet trees that give new interesting trade-offs between space and search performance. Moreover, the last two techniques adapts well for text indexing by using words instead of characters. This increases the vocabulary size, but obtains some added advantages: (i) the ability to search for a word by just performing one traversal through the index, and (ii) a more biased distribution of the words in the vocabulary which makes it more compressible with techniques such as word-based Huffman or ETDC. As future work, we pretend to perform exhaustive experiments over different texts, and work in efficient implementations of byte-oriented rank and select operations.

[2] As shown, since the height of this wavelet tree is much smaller, performance is not worsened with respect to the previous approaches.

References

[1] N. Brisaboa, A. Fariña, G. Navarro, and M. Esteller. (s,c)-dense coding: An optimized compression code for naturallanguage text databases. In *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, pages 122–136. Springer, 2003.

[2] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.

[3] N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In *Proceedings of the 25th European Conference on Information Retrieval Research (ECIR'03)*, LNCS 2633, pages 468–481, 2003.

[4] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proceedings of SPIRE 2005: 12th International Conference String Processing and Information Retrieval*, LNCS 3772, pages 1–12. Springer, 2005.

[5] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 03)*, pages 841–850, 2003.

[6] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, New York, 1978.

[7] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Inst. Radio Eng.*, pages 1098–1101, Sept. 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.

[8] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 2006. Special issue on "The Burrows-Wheeler Transform and its Applications". To appear.

[9] A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.

[10] 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.

[11] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):1–66, 2006.

[12] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, USA, 1999.

[13] N. Ziviani, E. Silva de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval system. *IEEE Computer*, 33(11):37–44, 2000.