DOI: 10.2298/CSIS110606004M

Wavelet trees: a survey

Christos Makris[1]

[1] Department of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece
makri@ceid.upatras.gr

Abstract. The topic of this paper is the exploration of the various characteristics of the wavelet tree data structure, a data structure that was initially proposed for text compression applications but has found a plethora of other uses in text indexing and retrieval. Issues concerning the efficient maintenance of the structure, plus its handling in various applications are explored. Our main aim is to provide to computer science researchers that would like to explore the specific area, an up-to-date comprehensive material covering a wide range of applications. This kind of up-to-date survey is missing from the current bibliography and we hope that it will help young researchers to get familiar with the notions of this research area.

Keywords: information retrieval, text algorithms, data structures.

1. Introduction

The constant increase in the volume of transmitted and stored data makes imperative the design of efficient algorithms and data structures for handling them. The field of data and text compression has been traditionally involved in providing tools that could face effectively the problems emerging in large scale information retrieval applications. Moreover in the time course of the previous decade, a challenging and interesting issue has flourished: self-indexing data structures, that is data structures that do not need the storage of the source text to operate, but embed in them both the text, and the indexer that operates on it. Self indexed data structures move the compression target from the source text to the indexing module and permit further functionalities.

There are a lot of algorithms and compression results that fall under this realm of research activity [39], [40], [43], [44], [45], [94] and from this emergence of results, a data structure called wavelet tree and having its roots in the arena of range searching data structures has emerged. This data structure that begun as a main component of compressed suffix arrays [64] and as a complement to other indices (such as the FM-index [94]), has been fruitfully extended in order to be used in other applications such as image compression, posting lists' handling, spatial searching, XML queries and many more.

The basic theme in the present paper is to examine the various issues involved and the basic characteristics of the specific data structure, dealing with a set of selected applications. There exists various works [38], [51], [68] exploring different aspects of the wavelet tree data structure however there does not exist in the literature an up to date survey covering its plethora of applications and referring to the till now space and time complexities bounds for them, in order to be provided as a roadmap to young researchers entering the area. We aim to cover this gap in the scientific literature by condensing all relevant information in one single reference, appropriately putting all the pieces together, thus helping young researchers and enter them smoothly in the notions of the specific scientific area.

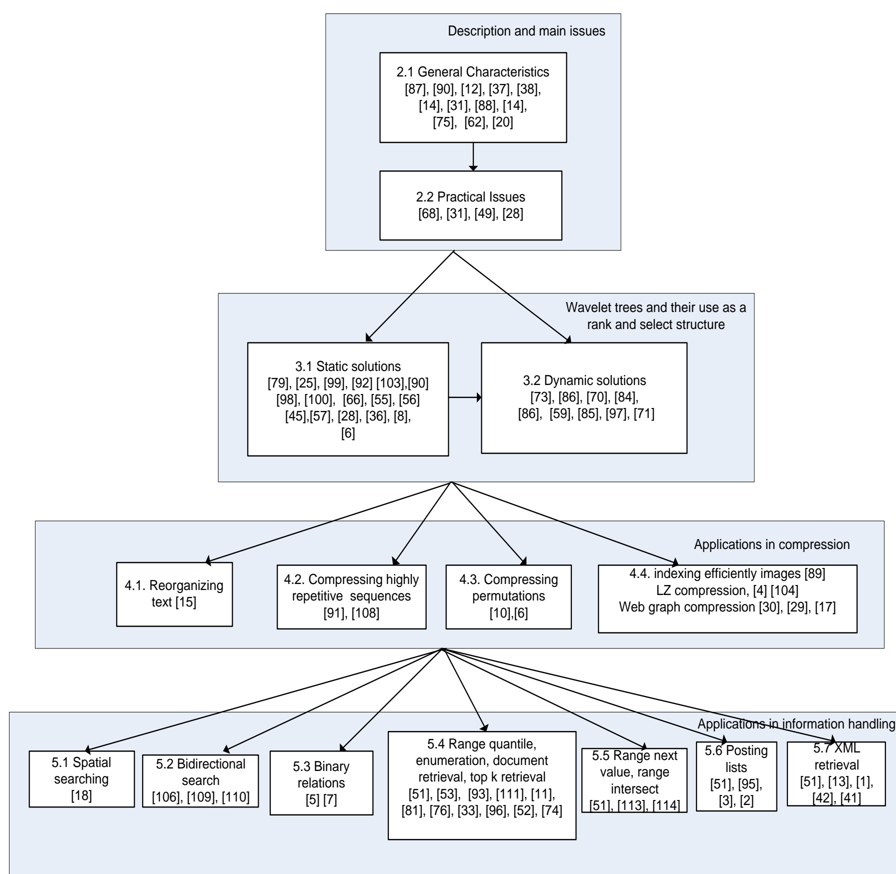

Fig. 1. A tree diagram of the paper's structure

Before proceeding, it would be interesting for the reader to have as motivation, during reading the paper, a practical application. For example, consider that the access log to a set of Web sites is stored as a sequence of Web pages, ordered by their access time, with each page being mapped to a

distinct symbol, and that the length of the sequence is *n*. That is, the produced sequence *S*, is of the form, $S=p_1,p_2,\ldots, p_n$ where $p_i \in \Sigma$, and Σ denotes the alphabet of symbols. Then the wavelet tree can be used to support two elementary operations: (1) locate the number of occurrences of a given page, in a given position of the sequence (starting counting from the beginning of the sequence), and (2) locate the position in the sequence of the i-th occurrence of a given page. Using these two elementary operations, it is possible to answer interesting queries in the available sequence, such as counting how many times a given page was visited in a period of time, find the most visited pages, and queries of similar nature that could be a value to Web designers and/or Web miners. It would be helpful for the reader to have this simple example as a working exercise while reading the paper, and find out how wavelet trees efficiently handle the described operations.

We organize the whole material in the following sections: in section 2 we present the main characteristics of the structure, in section 3 we describe its connection with rank and select structures, in section 4 we describe various of its applications in data compression, and in section 5 we describe its main applications in information handling (information retrieval, data retrieval and spatial objects retrieval). Finally in section 6 we conclude with discussion and open problems. In figure 1 we provide the reader with a tree diagram depicting the structure of the paper, in relation with the various application areas that are covered and the references used in them.

Before proceeding and in order to make the paper self contained we provide some definitions that will be used continuously in the sequel. Consider a sequence *S* of *n* symbols from an alphabet $\Sigma=\{c_1,\ldots,c_\sigma\}$ of cardinality σ, we call as *entropy H* the sum[1]:

$$H = \sum_{i=1}^{\sigma} p_i \log \frac{1}{p_i}$$

where p_i, is the probability of appearance in *S*, of the *i*-th symbol in the alphabet. According to the coding theorem of Shannon [107], this notion of entropy represents a lower bound to the average numbers of bits needed to represent each symbol in *S*, provided that the symbols appear independently. This probabilistic notion of the entropy that takes into account the statistical nature of the source, is usually replaced in the scientific literature by the most practical notion of *empirical entropy*.

In particular the zero-order empirical entropy of *S* is defined as:

$$H_0 = H_0(S) = \sum_{c_i \in \Sigma} \frac{n_i}{n} \log \frac{n}{n_i}$$

where n_i is the number of appearances of character c_i in *S*. If we take also into account the context of appearances this definition can be extended to the so called *k*-th order empirical entropy. For a string $w \in \Sigma^k$ let us denote with w_S the subsequence of characters that follow[2] *w* in *S*, then the *k*-th order empirical entropy of *S*, is defined as follows:

[1] Throughout the paper, the symbol *log*, will denote base 2 logarithms.

[2] In some works they refer to the set of characters that *precede* w, though this seems counterintuitive, it helps in the analysis; either way the difference is small [43].

$$H_k = H_k(S) = \frac{1}{n} \sum_{w \in \Sigma^k} | w_S | H_0(w_S)$$

Both zero-order and the *k*-th order empirical entropy are more practical measures when one needs to lower bound the compressibility of a text (the first when considering symbols as independent from each other, and the second, when taken into account the size *k* context, where the symbols appear), since they do not need to make any assumption concerning the probabilistic nature of the source of the text.

A related notion is the *modified empirical entropy,* a notion that is helpful for highly compressible text [39]. The zero-order modified empirical entropy is defined as:

$$H^*{}_0(S) = \begin{cases} 0 & \text{if } |S| = 0 \\ (1 + \lfloor \log |S| \rfloor) / |S| & \text{if } |S| \neq 0 \text{ and } H_o(S) = 0 \\ H_0(S) & \textit{otherwise} \end{cases}$$

A set S_k of substrings of S of length at most k is termed a *suffix cover* if any string in Σ^k has a unique suffix in S_k; in this case define:

$$H^*{}_{S_k}(S) = \frac{1}{|S|} \sum_{w \in S_k} | w_S | H^*{}_0(w_S)$$

Then, the *k*-th order empirical entropy of S is defined as:

$$H_k{}^*(S) = \min_{S_k} H^*{}_{S_k}(S).$$

2. Description and Main Issues

2.1. General characteristics

The wavelet trees have their roots in a range searching data structure described in a paper by Bernard Chazelle [20] and were introduced as a distinct structure in [64], as a component of a self-indexed compressor based on suffix arrays, and afterwards a set of other papers explored and extended their use (the majority of the papers referred to in the bibliography listing). A wavelet tree acts basically as a mechanism that permits the efficient implementation of specific operations on sets of objects from non-binary alphabets, via the use of the respective bit vector operations on sets of objects from the binary alphabet, with a logarithmic slowdown in the attained performance.

The main advantage of this transformation, that from a worst case point of view and from a query time perspective is inferior to other solutions proposed for these operations on non-binary alphabets, is that this transformation can embed entropy bounds in the attained space complexity. It thus can be used

as a general tool for reducing computationally the compression of a string from an arbitrary alphabet to the compression of binary strings.

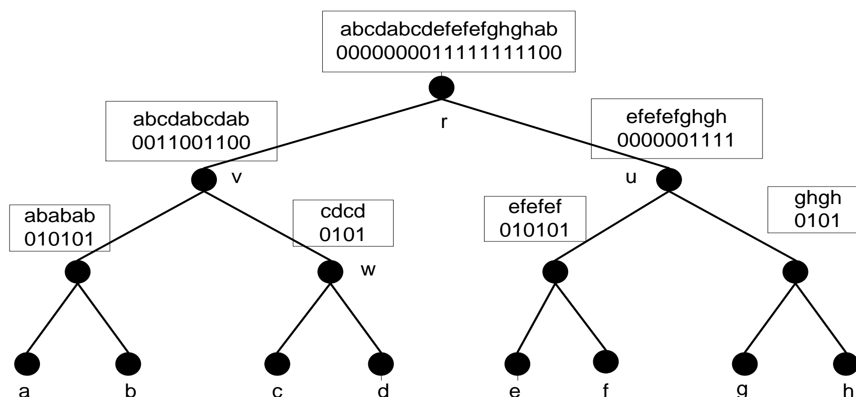

Fig. 2. The wavelet tree for S=abcdabcdefefefghghab, Σ={a,b,c,d,e,f,g,h}

A wavelet tree can act both as a compression mechanism on its own, and as a component of a compression algorithm; that is a wavelet tree can simply store a text by itself [38] and provide both an indexing and a compression mechanism or it can be used as a data structural component storing auxiliary information for a compression algorithm [4], [94].

More analytically consider a sequence S, of length n where the elements belong to an alphabet Σ, of size σ. A wavelet tree T is a static complete binary tree whose leaves store the symbols of the alphabet Σ in order from left to right (that is the number of leaves of the tree is equal to the size of the alphabet) and where each internal node v corresponds to a subsequence S_v of S that is stored through a binary vector B_v. The subsequence S_v is formed by the symbols of the sequence S that are stored in the subtree with root v, while the binary vector B_v has the same length as S_v, and each of its positions corresponds to a distinct symbol in the specific subsequence, such that $B_v[i]=0$ if the i-th symbol of S_v is stored in the left subtree of v, and $B_v[i]=1$ if the i-th symbol of S_v is stored in the right subtree of v.

If we state this recursively each subtree T_v of T is itself a wavelet tree for S_v. As an example see figure 2 for an exposition of a wavelet tree for the sequence S=*abcdabcdefefefghghab,* with alphabet $\Sigma=\{a,b,c,d,e,f,g,h\}$ where at each node we depict the subsequence with the accompanying bit vectors. For example, in the root r since a, d are stored in the left subtree the respective bits are set to 0, while the bits for e and g are set to 1, since they are stored in the right subtree. In node v the symbol a is stored in the left subtree, hence its corresponding bit have value 0, while symbol d being stored in the right subtree has its corresponding bits being set to 1. Finally in node u the symbol e is stored in the left subtree hence its corresponding bits have value 0, while the symbol g being stored in the right subtree has its corresponding bits being set to 1.

In each node of the wavelet tree, the bitvector B_v is stored in a structure suitable for accessing any of its stored elements, and answering efficiently rank and select queries [25], [79], [99], [103]. More analytically, rank and select queries on bit vector are defined as follows:

- $B_v(i)$, for returning the i-th bit of the bit vector B_v,
- $rank_b(B_v,i)$, that returns the number of times bit b appears in the prefix of B_v consisting of its i first symbols and
- $select_b(B_v,i)$, that returns the position of the i-th appearance of bit b in B_v.

Attaching these structures to each node we can use the wavelet tree to answer rank and select queries for any symbol in the sequence S, with a time complexity that is equal to the depth of the tree (that is $log\sigma$) multiplied by the time to perform a rank and select query in a bit vector of size at most n. Note, that the rank and select queries for S are defined similarly to their binary counterparts that is:

- $rank_c(S,i)$, return the number of times symbol c appears in the prefix of S consisting of its i first symbols, and
- $select_c(S,i)$, that returns the position of the i-th appearance of symbol c in S)

In order to answer the rank query $rank_c(S,i)$, we start from the root r of the wavelet tree and if c is stored in the right child we compute as new i the value $rank_1(B_r,i)$ otherwise the new i is the value $rank_0(B_r,i)$, and the procedure continues till reaching a leaf. On the other hand and for the query $select_c(S,i)$ we move bottom up, starting from the leaf corresponding to c, and if it is a right child the corresponding position to its parent v is $select_1(B_v, i)$ otherwise it is $select_0(B_v,i)$; the procedure moves similarly upwards until reaching the root, where the answer is reported.

We provide a toy example of how these algorithms work by treating queries $rank_d (S,9)$ and $select_d (S,2)$. The procedure for $rank_d (S,9)$ starts at r. Since d is stored in the left subtree, we compute $rank_0(B_r, 9)=8$, and we go to v. Since d is stored in the right child of v, we compute $rank_1(B_v, 8)=4$, to locate the proper position in the right child and move to w, there d is stored as a leaf in the right child of w, and hence the answer is $rank_1(B_w,4)=2$. In order to answer $select_d (S,2)$, we start from the leaf of the tree corresponding to d and we move upwards. Since the leaf storing d is at the right child of w, the new position is $select_1(B_w,2)=4$. Moving to v, since we come from the right child the new position is $select_1(B_v,4)=8$. Finally, at the root the seeked position is $select_0(B_r,8)=8$.

If the bitvectors are implemented using the structure in [25], then the overall space complexity of the construction is $nlog\sigma(1+o(1))$ bits, while the time to answer a query is $O(log\sigma)$. If we use for the implementation of the rank and select structure the construction described in [103] then the space complexity becomes entropy bounded by $nH_0(S)+O(nloglogn/log_\sigma n)$ bits, while the time complexity remains $O(log\sigma)$.

There is a deep connection between the aforementioned structure and a construction proposed by Chazelle for range searching in [20], this connection will appear more clearly in the following sections, where spatial

searching applications are presented. We explore this connection, that was noticed also by various researchers, see e.g. [87], more deeply.

Therefore, consider a set of points in the *xy*-plane with the *x*- and *y*-coordinates taking values in *{1,…,n}*; assume without loss of generality that no two points share the same *x*- and *y*-coordinates, and that each value in *{1,…,n}* appears as a *x*- and *y*- coordinate. We need a data structure in order to count the points that are in a range $[l_x, r_x] \times [b_y, u_y]$ in time *O(logn),* and permits the retrieval of each of these points in *O(logn)* time. The structure proposed in [20] for this problem needs *nlogn(1+o(1))* bits, and we will describe it as in the version provided in [87], which is simpler, and more clearly described; this structure is essentially equivalent to the wavelet tree. The structure is a perfect binary tree, according to the *x*–coordinates of the search points, with each node of the tree storing its corresponding set of points ordered according to the *y*-coordinate. In this way the tree mimics the distinct phases of a mergesort procedure that sorts the points according to the *y*-coordinate, assuming that the initial order was given by the *x*-coordinate. This construction takes *O(nlogn)* space, that can be reduced to *nlogn(1+o(1))* bits by the *functional approach* introduced in [20] that principally states that each node of a data structure does not need to store the sets of points that correspond to it, but needs only to provide the means (functions) for computing them. Hence each node instead of the real points stores a bitmap B_v with $B_v[i]=0$ iff the *i*-th point in the subsequence mapped to *v* belongs to the left child, otherwise $B_v[i]=1$. This bitmap is preprocessed, for the computation of rank and select queries in constant time.

Using this repertoire of structures we can retrieve the identity of any point, giving a specific node, and a specified bit of its bit vector, by descending a path in the tree to a respective leaf by performing suitable rank queries; the time cost is *O(logn)*. In order to count the points that are in the range $[l_x, r_x] \times [b_y, u_y]$ we just need to find in the tree by two searches the O(log*n*) maximal tree nodes that cover $[l_x, r_x]$, and for each of these nodes perform a subtraction in the result of two rank queries (for more details see [20] and [87]). If we do not want to count but also to retrieve the points we have to identify each of these points, paying an extra logarithmic term for each of them. The aforementioned construction, that works for distinct sequences can also be extended for handling arbitrary sets of points in a continuous space, as shown in [19] .

The connection between wavelet trees and range searching was more clearly depicted in [12] and in [37]. In [12] a solution was presented for storing *n* points with *x*- and *y*- coordinates from the rank space, using *nlogn+o(nlogn)* bits, supporting range counting in *O(logn/loglogn*) time and range reporting in *O(klogn/loglogn)* time, where *k* is the size of the output; a basic ingredient of this solution is a multi-ary wavelet tree. In [37] the entropy of the stored two dimensional set came into play. In particular consider a set of *m* points with *x*- and *y*- coordinates from the space *{1,…n};* the "entropy" *H* of this set is defined as the total number of the different possible grids that is: $\log\binom{n^2}{m}$.

It was proved in [37] that there exists a representation of this set, that takes $H+O(H)$ bits of space; the proposed representation employs a block partitioning of the points combined with wavelet trees for handling these partitions. The solution can answer range counting queries in $O(\log(n^2/m))$ time, range reporting queries in $O(\log^2(n^2/m))$ time per reported element, while point selection queries require $O(\log^2 n)$ time. By taking into account the density of the involved rank space it is possible to have the various times reduced down to $O(1)$ per reported element.

Wavelet trees have been mainly exploited as auxiliary structures for self-indexes based on the Burrows-Wheeler transformation [38], [40], [43], [44], [94]. In these algorithms it has been shown that indexing compressed text is reduced to performing rank and select operations in the sequence that is produced by applying the Burrows-Wheeler transformation on the text. Hence, these works have shown that the Burrows-Wheeler transform is not only useful for compressing a given sequence but it also allows to support search operations, if rank queries are supported over BWT. In these cases the binary vectors of the wavelet tree are compressed using run length encoded or gap encoding [38]. The application of run length encoding[3] compresses the sequence to its k-th order entropy, while the gap encoding does not achieve analogous bounds. However it is possible to apply gap encoding in combination with the compression boosting technique of [39] and have a compression algorithm of size $2.2618|S|H^*_k(S)+\log|S|+\Theta(|\Sigma|^{k+1})$ bits, for any positive k, thus improving the bound achieved in [39] into an almost optimal result. It should be noted that if the wavelet trees store the initial sequence (not its Burrows-Wheeler transformation) then they can be considered as standalone general purpose compressors.

Concerning the performance of wavelet trees as standalone compressors, it was noted in [38] that their use reduces the problem of compressing a string to that of compressing a set of binary strings, with the specification of the set being determined by the topology of the underlying tree structure, and the coding of the alphabet symbols at the tree leaves. The authors pursue their remarks even further by introducing the paradigm of the *generalized wavelet tree* and noting that it is possible by pruning whole subtrees of the initial tree, to have a mixed compression strategy where only some strings are binary and the others use symbols that are compressed with general purpose order zero compressors (for example Huffman or arithmetic coding). The authors apply this pruning strategy, by developing a combinatorial optimization framework based on the notion of leaf covers and by providing a polynomial time algorithm for selecting the optimal tree shape for some special cases.

In [44] it has been shown that run-length compression on the wavelet tree is not mandatory, if the wavelet tree is applied not to the whole Burrows-Wheeler transformed text, but separately to each one of the partitions of the

[3] The casual reader should be careful here to note that run-length compressing or gap-compressing a wavelet tree, means compressing *each* bitmap and *not* the whole sequence, before building the wavelet tree.

transformed text (provided an optimal partitioning has been found). The technique is called *compression boosting* [39] and permits the formation of *k*-order compressors, by using 0-order compressors. If compression boosting is used then the bit vectors in the nodes of the used wavelet trees, can be compressed by simply using the structure in [102]. This technique was improved in [88] (see also [86]) where it was shown that there is no need to apply the wavelet tree approach to every piece produced by the partitioning provided by the Burrows-Wheeler transform, but one can simply use the wavelet tree (using however [103] for representing the bit vectors) for the complete Burrows-Wheeler transformed text. In essence, the technique provides a way to do compression boosting implicitly, with a trivial linear time algorithm, but using a specific zero order compressor [103].

There have been proposed also variants of the above compressed representations of the wavelet tree, that exploit the observation, that the binary skeleton of a wavelet tree, if labeling the left edge of each node with 0 and the right edge of each node with 1, is basically a trie storing the binary string representations corresponding to each distinct symbol of its leaves. Henceforth it could be possible to achieve greater compression, by using instead of the binary representation for each symbol its code according to a prefix free code such as Huffman code (this appeared in [14], [28], [49], [68], [88]). In this case the tree is not balanced, and hence the time complexity of the query operations is no longer $O(\log\sigma)$, but can be as bad as $O(\sigma)$; however the average query time is bounded by $O(\log\sigma)$ since it is bounded by the entropy plus one, and the space needed to store it is reduced, since the average length of the paths to reach each leaf nodes is smaller.

Extending this approach we can use the wavelet tree by codifying words instead of characters and employing multi-ary wavelet trees. An example is provided in [14] where the wavelet tree is built over the codes associated to each word of the text using End-Tagged Dense Code (ETDC). This code produces sequences of bytes, and the constructed wavelet tree is no longer binary but multi-ary since the label of each edge equals a byte (the edge from the root signifies the first symbol) and each node can have up to 256 children, moreover each node will contain now a byte vector instead of a vector of bits. In [14] it is mentioned that, when working with natural languages, the codes generated by ETDC will never have length greater than 4, and hence the produced wavelet tree will have at most four levels.

It should be noted here that wavelet trees can be efficiently externalized and used in a set of applications in external memory, see for example [75] and [62]. In [75] it has been shown how the wavelet tree can be externalized optimally (linear number of blocks in space consumption, and an $\log B$ speedup in answering queries) by replacing the binary tree skeleton and the accompanying bit vectors by a B-ary tree, augmented with vectors of characters each with $\log B$ bits, while in [62] and in the context of presenting a practical self-index for secondary memory based on the FM-index alternative secondary memory implementations, were discussed.

2.2. Practical Issues

Concerning practical implementation and experimental outcomes of the wavelet tree a set of results have been published in [49], [28], [68]. In particular in [49] wavelet trees are implemented using mainly run length encoding and γ encoding for codifying the involved bit vectors. Moreover Huffman shaped wavelet trees, are involved with the provision of applying fractional cascading, in order to access efficiently the various dictionaries. An interesting aspect of the [49] construction is that besides using Huffman shaped wavelet trees (note that in this case of already compressed bit dictionaries variations in the tree shape does not seem to affect the total space) a frequency based construction is applied where the wavelet tree is built as an optimal Huffman prefix tree not on the frequency of appearance of the symbols, but on the a-priory distribution of the queries on them.

Experiments are performed that depict the usefulness of the frequency based heuristics, in search scenarios, though the fractional cascading does not seem to lead to considerable improvements. Moreover the proposed scheme was applied to the Burrows-Wheeler streams of various text files from the Canterbury and Calgary corpora and besides γ coding the following codes (for the run lengths of the bit vectors of the wavelet tree) were used: δ code, Golomb code, Maniscalo code, Bernouli code, or MixBernoulli code. It was depicted that run length encoding in combination with γ coding constitutes a simple solution capable of providing efficient compression.

In [28] a practical implementation of the [103] bit vector dictionary is provided, and its use in wavelet trees is experimentally tested. In particular the [103] dictionary is experimentally tested against the dictionaries described in [58] and it is shown that for uniformly and independently distributed bitmaps the dictionary in [103] is superior for high density data. Moreover by using [103] to implement the bit vectors at the nodes of a balanced wavelet tree of the Burrows Wheeler transform of natural language english text, it is proved that this structure is clearly superior to the [58] implementations; this is natural since the [103] structure exploits local regularities in the bit sequence. Moreover efficient implementations of the Huffman-shaped wavelet trees are provided that use no pointer information (or just $log\sigma$ pointers) while the bitmaps at each level are stored concatenated. Experiments performed in various datasets show that the [103] structure in combination with a Huffman shaped wavelet tree (with or without pointers) provide excellent compression, and helps reaching entropy bounds, even for very large alphabets.

Finally, and extending the findings of [49], [28] a set of alternatives concerning the tree topology and the implementation of the accompanied bit vectors is explored in [68]. The authors initially prove theoretically that for data of low entropy (formally when the 0-th order empirical order is asymptotically less than $log\sigma$), the run length δ coding is superior to the run length γ encoding, achieving $nH_0(S)$ (plus lower order terms) compression with leading constant 1, while run length γ encoding has leading constant 2. Moreover a software package is presented with a parametric implementation

of wavelet trees able to embed a large number of available options both in the tree topologies and in the involved bit dictionaries. The various topologies explored are: a balanced shape, an alphabetic weight balanced wavelet tree [10], and a Huffman shaped wavelet tree; moreover the attached bit vectors are implemented as: the structure in [103], run length γ-encoding [112], run length δ-encoding [112], pure arithmetic coding [112], selective compression at the lower levels of the tree, and compression with *t*-subset encoding.

A set of experiments are performed that depict that run-length encoding gives the more space efficient implementation especially when compressing Burrows Wheeler transformed text and low entropy data; they show that generally run length *γ* encoding is superior but for very low entropy data run length *δ* encoding becomes better. However the query performance of the run length implementation is quite poor, due to the need for handling the run length encoding. In these cases the weight balanced implementation using [103] or with compression at the lower levels are superior. Moreover concerning building times, run length compressed wavelet trees are the slowest, the other implementations are competitive to each other, and all implementations take advantage of Burrows Wheeler transformed text that is amenable to faster compression. Finally, and for applications where a good tradeoff is what is needed, Huffman shaped wavelet trees and weight balanced wavelet trees in conjunction with the structure in [103], are proved to be the best choice, a fact that revalidates the findings of [28].

In [49] and [31] the issue of efficiently building a wavelet tree was handled. In particular in [49] a time efficient algorithm is presented for constructing a wavelet tree for the Burrows-Wheeler output of a sequence *S* of size *n* in $O(n+min(n,H_k(S))log\sigma)$ time; the bit vector dictionaries are run length γ encoded and are concatenated together to heap order. The method is a bottom up procedure, that traverses the tree upwards, processes consecutive runs of equal symbols and extends appropriately the bitvectors of each node. The specific construction depicts that data that are highly compressible can be indexed faster.

In [31] the focus is on space efficient implementations, and novel algorithms are presented for constructing wavelet trees with virtually no space. The construction works for both binary and multi-ary wavelet trees, however it does work only for uncompressed wavelet trees (the bit vectors are not compressed in contrast to [31]), and extending their techniques to compressed wavelet trees, is left as an open problem. The techniques used are based on in place sorting and exploit properties of permutations, that permit the execution of the necessary *partitioning* (moving top down) and *merging* (moving bottom up) steps of the construction algorithms, without using extra storage. Two classes of algorithms are presented *non-destructive*, that do not alter the initial array of symbols and *destructive* that alter it, and use it for storing the created dictionaries of the tree nodes.

In particular for a bit vector of size *m*, let $C(m)$, $E(m)$, $S(m)$ be the construction time, extra bits required for construction and total space occupation in bits respectively. It should be noted here that these space and time complexities when constructing the bitmap refer to the complexities of

the *extra* data structures for supporting rank and select on the bit dictionaries and *not* on the space of the bitmap itself. The authors initially describe a non-destructive algorithm for a binary wavelet tree that needs $O(n\log\sigma + C(n\log\sigma))$ time and $O(\log n\log\sigma)+E(n\log\sigma)$ bits beyond the space for the array of symbols and the tree nodes. Then they describe destructive algorithms and show that it is possible to store a sequence of n symbols in a binary wavelet tree in (i) $O(n\log n\log^2\sigma)+C(n\log\sigma)$ time using $O(\log n\log\sigma)+E(n\log\sigma)$ extra bits beyond the space required for the tree, (ii) $O(n\log\sigma+C(n\log\sigma))$ time and $n+O(\log n\log\sigma)+E(n\log\sigma)$ extra bits beyond the space required for the tree. Moreover if incrementally constructed the $O(\log n\log\sigma)$ bits term in the space bound can be replaced with $O(\log n)$ bits.

3. Wavelet Trees and their Use as a Rank and Select Structure

3.1. Static Solutions

Wavelet trees, though having many uses, can be considered mainly as rank and select data structures for large alphabets and during their lifetime have been used as component substructures in various solutions to the rank and select problem. We remind the reader that in that problem we are given a sequence S of n symbols from an alphabet Σ of size σ, and we want to access the i-th element of the sequence, and answer, for every symbol c in Σ, the following queries: $rank_c(S,i)$ (that returns the number of times symbol c appears in the prefix of S consisting of its i first symbols), and $select_c(S,i)$ (that returns the position of the i-th appearance of symbol c in S).

In [79] the problem was handled for the case of sequences from a binary alphabet and a data structure was presented that needed $n+o(n)$ bits and supported both rank and select in $O(\log n)$ time. This solution was extended and improved in [25] and [92] where using the same space, both rank and select operations were handled in O(1) time. We will discuss these solutions following the presentation given in [25]. The general idea behind the construction is to store in tables, precomputed answers for the query arguments, and then at query time, just retrieve in O(1) time, with a simple lookup, the appropriate answer. If this idea is applied naively then the space complexity will be $O(n\log n)$ bits, but this can be overcome by precomputing answers for properly chosen samples in the query ranges and by employing multilevel schemes.

This multilevel scheme could lead to a non-constant query time, but when the size of the treated bit vector falls bellow $\log n/c$, (c designates a constant greater than 1) then, all possible subsets, can be treated with a global lookup table that needs $o(n)$ space and answers queries in constant time. These constructions as simpler for rank than select, and one should build the needed tables separately for rank and select queries. As in [94] we will focus

on the *$rank_1$* and *$select_1$* queries, *$rank_0$* follows from *$rank_1$*, while *$select_0$* can be computed as *$select_1$*.

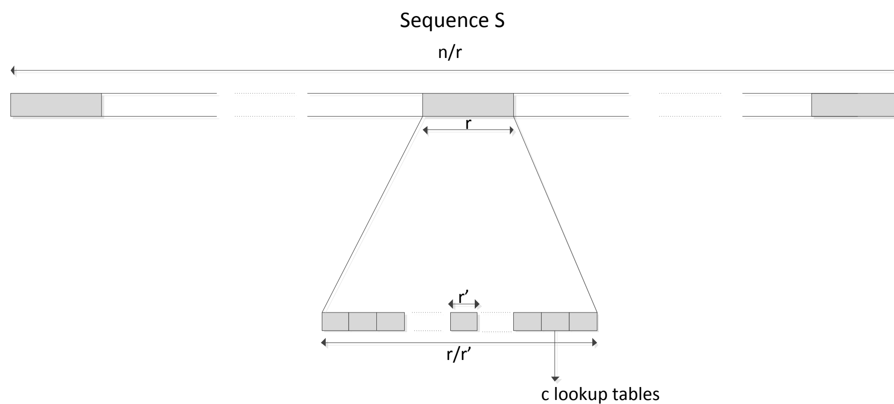

Fig. 3. *The rank multilevel structure*

In the rank case a multi-level directory scheme (see figure 3) is employed where the first level divides the sequence *S*, into consecutive chunks of size $r = \lceil logn \rceil^2$, and stores precomputed answers for arguments that are multiples of *r.* The second level deals separately with each chunk, dividing it in consecutive subparts of size $r' = \lceil logn \rceil$, and storing precomputed answers for arguments that are multiples of *r′*. Finally each subpart having size *r′* is divided into a constant number *c* of subsegments of size *r′ /c*. These segments are handled by a global lookup table that for each possible segment (there are $2^{r'/c}=n^{1/c}$ of these) and for each possible argument stores the answer (this technique can be considered [94] as a different application of the classical *four-Russians* technique).

In order to answer rank queries we first locate, with two table lookups, the correct entries in the first and the second level, and the sum of the appropriate values gives an *r′*-sized range. There we locate in constant time the appropriate subsegment, and using the global lookup table produce the rank in the segment (adding at most *c* values). Overall the query time is constant while the space consumption is: *O(n/logn)* bits for the first level, *O(nloglogn/logn)* bits for the second level, and $O(n^{1/c} lognloglogn)$ bits for the global lookup table, thus in total *o(n)* bits.

Considering now the *$select_1$* operation, it is a folklore solution to solve it by *O(logn)* calls to *$rank_1$* with each call taking *O(1)* time. We can solve the problem by employing again a multilevel dictionary structure that is a bit more complicated than the construction for the *$rank_1$* query. The first level records the position of every $l_1 = \lceil logn \rceil \lceil loglogn \rceil$ 1 bits. If the size of a range *r* in the initial vector between two of these 1-bits is greater than $(l_1)^2$ then the answers are explicitly stored, otherwise the range is partitioned by storing the relative position of each $l_2 = \lceil logr \rceil \lceil loglogn \rceil$ 1-bits. If the size *r'* of a subrange is

greater than $\lceil logr' \rceil \lceil logr \rceil \lceil loglogn \rceil^2$, then we store them explicitly, otherwise we note that the size of the range is less than $(loglogn)^4$. For these small ranges we can again employ a trick similar to the four-Russians technique and a lookup table suffices for performing select in constant time. Hence by a constant number of table lookups we can compute the answer to the select operation.

This construction was improved in [99] and [103] with a data structure of entropy bounded size $nH_0(S)+o(n)$ bits that could answer queries in constant time. In particular the idea is to replace the original bit sequence representation of $O(n)$ bits, with an entropy bounded representation, upon which the extra information and techniques needed by the [25] and [92] of O(1) query time tank and select, is employed. In particular the original sequence is divided into chunks of $b=(logn)/2$ size. Each of these chunks can belong to one of $(logn)/2$ classes according to its number of 1-bits, and a class with identifier j (that is having j bits set), can contain at most $\binom{b}{j}$ elements (that is, different chunks). A chunk therefore can be coded with two numbers: its class and its position inside its class. It is proved in [25] and [92] that the total storage of this representation is $nH_0(S)+o(n)$ bits, while the constant time query complexities for rank and select remain unaffected, by employing the same techniques but on the new representation.

In [90] it is shown how to implement efficiently the rank and select operations on top of sparse bit sequences using the gap encoding technique (see also [69]); in particular the gap between consecutive 1's was encoded using arbitrary random access self-delimiting integer codes. The provided construction needs constant time for both queries and takes $l\alpha log(n/l)+O(l)+o(n)$ bits of space, where α is a constant depending on the coding used and l is the number of 1 bits. In the same paper, a new problem is also introduced, the so called *position restricted substring searching*, and based on this, the rank and select operations are extended by taking as input not a single symbol but a whole string, while the presented solution entails the heavy use of wavelet trees. In particular if the text is of length n, from an alphabet of size σ, and the searched substring is p, then the provided solution needs $O(ntlog\sigma)$ bits of space and supports the various rank and select operations in $O(|p|log\sigma/loglogn)$ time; here t is a given construction parameter such that $|p|<t$.

Finally, and still remaining in binary alphabets, in [98] several practical alternatives were presented, while in [100], [55], [56], [66] a suite of solutions with various tradeoffs concerning the space and time complexities were presented. In particular, the sparse problem where the number of ones is small attracted attention; note that if the sequence contains k 1's then the optimal space consumption is: $B = \log\binom{n}{k}$.

The best solution was provided in [100] depicting that $O(t)$ query time can be attained using[4] $B+n/(logn/t)^t+\tilde{O}(n^{3/4})$ bits of space, while in [66] a parametric construction was provided that for $0\leq\delta\leq1/2$, $0\leq\varepsilon\leq1$, (δ, ε are any real constants), and positive integer s, has $O(s\delta^{-1}+\varepsilon^{-1})$ query time and uses $B+O(k^{1+\delta}+k(n/k^s)^\varepsilon)$ bits.

Moving now to the case of large alphabets, wavelet trees provide a solution that needs $nH_0(S)+o(nlog\sigma)$ (or $nH_0(S)+o(n)log\sigma$) bits of space and $O(log\sigma)$ time for rank and select. This construction was improved in [45] by designing efficient structures for alphabets of small size and then moving to alphabets of larger size by employing properly defined multi-ary wavelet trees. In that paper initially a solution using $nH_0(S)+O((\sigma nloglogn)/log_\sigma n)$ bits was provided answering queries in constant time, when the alphabet size is of the order $o(logn/loglogn)$.

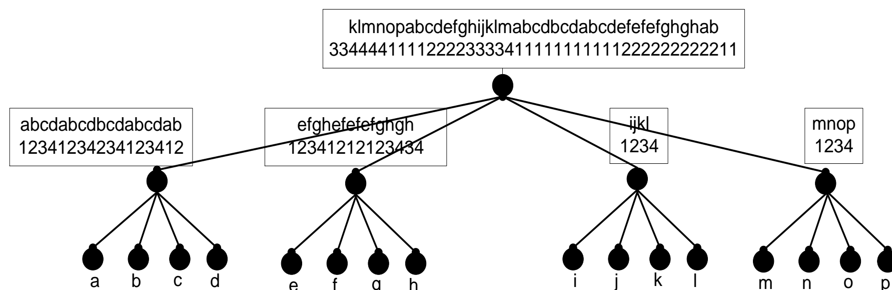

Fig. 4. *The 4-ary wavelet tree*

The construction is inspired by the ideas of [99] and [103], concerning the representation used for getting entropy bounds. The original ideas are delicately modified in order to handle non-binary alphabets, and are carefully tuned in order to attain efficient compression, this is the reason for the upper bound on the alphabet size. Let $\Sigma=\{c_1, .., c_\sigma\}$ be the alphabet. The basic idea, as in [99] and [103], is to separate the sequence S into chunks of size $\lceil(log_\sigma n)/2\rceil$, and codify the classes of the chunks according to the participation of the symbols in the chunk; a *symbol decomposition* is defined as an σ-tuple $(n_1, n_2, \ldots, n_\sigma)$, where n_i is the frequency of appearance of symbol c_i. Hence a chunck is codified by its symbol composition, and its unique relative position among all the chunks having the same symbol composition. Moreover precomputed tables store the answers for all possible combinations of rank and select queries, while the rank and select procedures follow the same logic as previously described.

The encoding of the sequence using the block decomposition and the relative position of the block in its symbol decomposition gives rise to the $nH_0(S)$ space complexity, with the various other tables employed needing a total of extra $O((\sigma nloglogn)/log_\sigma n)$ bits; these extra bits are the reason for the

[4] With the Õ notation poly-logarithmic factors are ignored, that is $\tilde{O}(f(n))=O(f(n)log^{O(1)}n)$

constraint on the size of the alphabet, since larger values could make the space complexity larger than $nlog\sigma$, which is the uncompressed space complexity.

The result is then generalized to alphabets of arbitrary size by employing a multi-ary generalization of the binary wavelet tree. In particular an h-ary tree is used with height $1+log_h\sigma$, with each node of the tree containing instead of binary vectors, vectors of integers in the range $[1,h]$. Let v be a node with children $v_1 \dots v_h$, then the subsequence S_v is stored in an array A_v such that $A_v[j]=i$, iff the j-th symbol in the sequence is stored in the i-th subtree of v. All the arrays at a specific level are physically stored concatenated, and they are handled by using the rank and select structures for the small alphabet case. In particular the rank query is implemented as in the binary tree case by starting from the root moving to the proper child in O(1) time, and calculating as new argument of the rank operation the result of the rank at the root, the same procedure is followed until reaching the appropriate leaf. Similarly and for the select operation we start from the leaf of the given symbol, and move upwards by setting as new argument the result of the select operation at the current node; the answer is the value returned at the root of the tree. Finally the value of h is set to be equal to $O(log^{\delta}n)$ with $\delta < 1$, and since descending each level takes constant time, it follows that the whole construction uses $o(nlog\sigma)$ extra space over the $nH_0(S)$ term and $O(log\ \sigma / log\ log\ n)$ query time. In figure 4, a 4-ary wavelet tree is depicted where for reasons of simplicity we depict the arrays at each level separately at each node.

Besides wavelet trees, in [57] two data structures were proposed, one that can access an element and handle the rank operation in $O(log\ log\ \sigma)$ time, and the select operation in O(1) time, using $nlog\sigma + no(log\sigma)$ bits of space, and another that supports the operations in the same time, but using space $nH_0(S)+O(n)$ bits. The main idea is to reduce the problem over one sequence to chunks of length σ, handle efficiently each chunk, and represent in a unified way all the chunks together.

As mentioned, in [28] a practical study was presented on the compact representation of sequences supporting rank, select, and access queries. This paper implements the structures described in [103] and in [57] (the first such implementation at the time [28] was written) and offers a well tuned implementation of Huffman-shaped wavelet trees showing that this provides an excellent solution for representing a sequence within zero-order entropy bounded space, even when the alphabet size is very large.

In [36] different implementation strategies are presented (without however providing new complexity bounds), while in [8] (see also [46]) a structure was provided using $nH_k(S)+o(nlog\sigma)$ bits supporting access/rank/select operations in $o((loglog\sigma)^2)$ time, but without using wavelet trees. In particular in [105], [46], [61], general techniques are presented for compressing general sequences to the k-th entropy bound, without affecting the query complexities; these constructions permit the compression of the various rank/select structures to $nH_k(S)+o(n)$ bits.

Finally interesting constructions are provided in [6] and a novel technique of combining previous results on rank and select data structures was

presented in order to handle large alphabets. The specific technique uses the frequencies of appearances of the characters, in order to divide the alphabet into subalphabets, and produces a new string by replacing each character on the original string by identifiers of their subalphabets; this string is stored in a multi-ary wavelet tree. Moreover the projections of the original string to the characters belonging to each sub-alphabet, are stored using the structure in [57] for large alphabets.

The outcome of this construction is a data structure that stores a string of n characters over an alphabet of size σ, in $nH_0(S) + o(n)(H_0(s)+1)$ bits. This structure supports the operators access and rank in time $O(\log\log\sigma)$, and the select operator in constant time; while it can be extended and be easily implemented. The technique can be improved achieving $nH_k(S)+o(n)\log\sigma$ bits space complexity and needing poly double logarithmic time, for the query time. Moreover the construction has applications to the design of full-text self-indices, to the representation of compressed permutations, and to efficient handling the compressed representation of dynamic collections of disjoint sets.

3.2. Dynamic Solutions

In the dynamic version of the problem, the rank and select operations are extended by the update operations of inserting an arbitrary object (bit/symbol) between two specified objects in the sequence, and by deleting an object (bit/symbol) from the sequence. The problem was dealed with in [73] for binary sequences with a solution requiring $n+o(n)$ bits of space, $O(b)$ amortized update time, and $O(\log_b n)$ time for rank, select, where $b=\Omega(\text{polylog}n)$.

In [86] the aforementioned solution was improved producing a construction able to handle binary strings in $nH_0(S)+o(n)$ bits of space, and $O(\log n)$ worst case time complexity for all operations. In that paper the dynamic solution was generalized to symbols in an arbitrary alphabet by using the wavelet tree machinery and preprocessing the bit vectors at each level of the tree structure with the dynamic structure for handling bit sequences. The result is a structure that uses the same $nH_0(S)+o(n\log\sigma)$ bits of space, and supports all the operations in $O(\log n\log\sigma)$ time (that is the use of the wavelet tree, as anticipated, incurred a $\log\sigma$ slowdown in the operations time complexity).

In particular the authors consider firstly the case where the sequence is binary, and they sketch a simple solution with $O(n)$ space and $O(\log n)$ worst case update and query time. They simply divide the sequence into blocks of $O(\log n)$ bits, attach every block to a leaf of a balanced binary tree, store at each internal node the size and number of bits set in the respective subtree (see figure 5 for a snapshot of the structure), and they merge/split leaves when the capacity of blocks shrinks or expands at $(1/2)\log n$ and $2\log n$ respectively. However this simple solution is not (asymptotically) entropy bounded and is not even succinct. In order to make the structure succinct the authors employ a two level blocking scheme at the lower levels of the tree.

Now a leaf is attached to a *superblock* that is equivalent to a group of simple blocks (though it can be considered simply as a block of a larger capacity).

Fig. 5. The tree structure

A superblock is composed of $f(n)$ consecutive blocks ($f(n)$ is a parameter judiciously chosen for guaranteeing acceptable time and space bounds) with the size of them being equal to $(1/2)logn$. The superblocks are all full, with the exception of some partial leafs; the authors impose the invariant that every $f(n)$ full leaves there exists a partial leaf, and implement each leaf as circular arrays except the partial leafs that are implemented as simple arrays. It is shown that the whole scheme needs $O(n/f(n))=o(n)$ extra bits and permits the handling of the various operation in $O(logn)$ time. In particular the space is $n+O(n/f(n))$ bits and the update/query time is at most $O(logn + f(n))$, hence by choosing $f(n)=(logn)$ we get a total construction of $n+O(n/logn)$ bits and $O(logn)$ worst case time for all query and update operations.

Then the authors propose two approaches for making the scheme entropy bounded either by applying gap encoding or by applying block identifier encoding, in order to compress the original sequence. Note that in both cases the dynamic structure follows the same design principles and the time bounds remain $O(logn)$ worst case with the difference that in the second case the $f(n)$ function should be $(logn)^{1/2}$. Assuming that w designates the machine's word size, then it is proved that: (i) using gap encoding, the attained space is $n(1+o(1))H_0(S)+O(l+n^{1/2}polylog(n)+w)$, where l denotes the number of 1's, and (ii) using block identifier encoding, the attained space is $n(1+o(1))H_0(S)+O(nloglogn/logn + w)$.

The construction is then extended to small alphabets of size σ (with the limitation that $\sigma=o((logn)^{1/2})$ by using the same techniques (superblocks contain $2f(n)$ blocks, while blocks are $(1/2)logn$ bits long that is $(1/2)log_\sigma n$ symbols long), and by employing a similar representation scheme as that

used in [45] (that is exploiting the symbol decomposition and encoding a block by its position in its specific class). By carefully handling the various technical details arising from this new representation and by using again (as in the block identifier scheme mentioned before) $f(n)=(logn)^{1/2}$ the attained complexity is $nH_0(S)+o(nlog\sigma)$ $+O(w)$ bits of space, $O(logn)$ worst case time for rank and select, and $O(\sigma logn)$ worst case time for insert/delete. The careful reader would notice here the difference between query and update time, a difference that has focused the attention of several authors for possible improvement. Finally, the construction (as [45]), employs a multilevel wavelet tree in order to extend the solution to larger alphabets. In particular a q-ary wavelet tree is employed with $q=o(logn)$ and the resulted construction needs $nH_0(S)+o(nlog\sigma)+O(w)$ bits of space, handles query operations in $O(lognlog_q\sigma)$ worst-case time, and update operations in $O(qlognlog_q\sigma)$ worst case time; for $q=2$ the worst case query and update time are asymptotically the same: $O(lognlog\sigma)$.

The above structures achieve entropy bounds in the space complexity, but are not so fast; this was tried to be handled in subsequent efforts that removed the entropy from the space complexity, in order to become faster in the time complexity. In particular in [70], a construction was provided without using wavelet trees that used $nlog\sigma+o(nlog\sigma)$ bits in the space complexity, providing $O((1/\delta)loglogn)$ time for rank and select queries, and $O((1/\delta)n^{\delta})$ time for updates.

In another attempt, in [84] an improvement to the [86] data structure was presented in the context of a general dynamic rank and select framework that was later explored in other works too. In particular the authors presented initially a solution for a small alphabet with size σ less than $logn$, and then they extended this solution for an alphabet of arbitrary size. Their small alphabet solution follows the [86] solution with the difference that it separated their construction into two schemes a counting and a storing scheme, thus having the opportunity to optimize both of them simultaneously; the proposed construction supports rank and select queries in $O(logn)$ worst case time, and update operations in $O(logn)$ amortized time, moreover the space required is $nlog\sigma+o(nlog\sigma)$ bits. In order to handle alphabets of arbitrary size the authors employ once more multi-ary wavelet trees, for a carefully chosen branching factor. In particular for an alphabet of size σ they regard each symbol of $log\sigma$ bits as $log\sigma/loglogn$ digits from a $logn$-size alphabet. Hence the built h-ary wavelet tree is an $logn$-ary tree. As usual the sequence vectors at each node are represented by the dynamic rank and select structures employed previously for the small dictionaries, and it is proved that the provided structure can handle sequences of alphabets of arbitrary size using $nlog\sigma+o(nlog\sigma)$ bits, supporting updates in $O(logn(1+log\sigma/loglogn))$ amortized time, and needing $O(logn(1+log\sigma/loglogn))$ worst case rank and select query time.

As it can be noted the above two solutions [84] and [86] are complementary to each other, that is while [84] is fast and uncompressed, the solution in [86] is slower but compressed. In [59] an effort was made to provide a solution that could combine the best features of these two

constructions. Besides combining ideas, the authors in [59] imported also new ideas such as embedding a substructure for handling dynamic partial sums and novel deamortization approaches. The proposed construction initially provides a solution for alphabets of small size ($\sigma=o(logn/loglogn)$) that needs $nH_0(S)+O(nlog\sigma/ (logn)^{1/2})$ bits of space, and supports all operations in $O(logn)$ worst case time. This solution is then extended for alphabets of larger size by using a generalized h-ary wavelet tree where $h=\Theta(logn)$, and thus obtaining a solution with $nH_0(S)+O(nlog\sigma/(logn)^{1/2})$ bits of space and supporting all operations in $O((1+log\sigma/loglogn) logn)$ worst case time.

In [85] an alternative to the [59] structure is presented that is a compressed version of their construction in [84]. In particular while [59] uses a block identifier encoding to compress and a theoretical counting argument to guarantee worst case time, the authors in [85] employ gap encoding in order to achieve compression plus a simple counting structure to have amortized updated time. The proposed solution needs $nH_0(S)+o(nlog\sigma)+O(n)$ bits, the rank/select queries take worst-case time $O(logn)$ and the update operations are handled in amortized $O(logn)$ time for a $logn$-sized alphabet. This can be extended to arbitrary alphabets by using $logn$-ary wavelet trees that need the same space, rank/select queries are handled in $O((1+log\sigma/loglogn)logn)$ time, while update operations are handled in $O((1+log\sigma/loglogn)logn)$ amortized time.

Finally in [97], [71] an improvement to the above time complexities was attained.

In particular in [97] a data structure for manipulating efficiently bit vector operations the *range min max* tree was presented, in the context of dealing effectively with various operations on static and dynamic succinct trees. By using this structure and exploiting techniques presented in [21], [45], [86], the authors were able to show that: (i) a sequence S of n bits can be maintained in $nH_0(S)+O(nloglogn/logn)$ bits such that query and update operations can be handled in $O(logn/loglogn)$ time, (ii) a sequence S of n symbols of an alphabet $\sigma=O(log^{1-\varepsilon}n/loglogn)$ can be handled in $nH_0(S)+O(n\sigma loglogn/logn)$ bits of space, so that all the operations can be handled in $O(logn/loglogn)$ time. The result for alphabets of small size, can be extended by using a multi-ary wavelet tree with branching factor $O(1+log\sigma/((1-\varepsilon)loglogn)$. The resultant structure needs space $nH_0(S)+O(nlog\sigma/log^{\varepsilon}n + \sigma log^{\varepsilon}n)$ bits and supports the query and update operations in $O(logn/loglogn(1+log\sigma/loglogn))$ time.

On the other hand in [71], a construction was proposed that follows (and improves) the construction of [59], and employs generalized wavelet trees in order to support rank, select, insert and delete in $O(logn/loglogn(log\sigma/loglogn+1))$ time using $nH_0(S)+o(n)log\sigma+O(w)$ bits. It should be noted that the construction of [71] for small alphabets (of size σ=O(polylog(n)) support all the operations in O(logn/loglogn) time.

4. Applications in Compression

4.1 In [15] a novel application of wavelet trees, in order to effectively compress natural language text is proposed. In particular it is shown that wavelet trees can be used as a means of reorganizing natural text that has been word-compressed in order to guarantee self-synchronization, even for compression algorithms that are not self-synchronized.

Table 1. codewords

Symbol	Codeword
you	b_4
can	b_3
compute	b_5b_0
the	b_5b_1
complexity	b_5b_2
in	$b_5b_3b_0$
a	$b_5b_3b_1$
data	$b_5b_3b_2$
structure	$b_5b_3b_3$

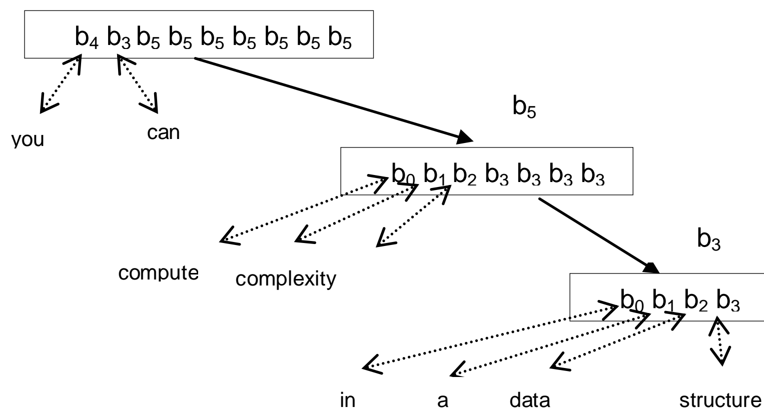

Fig. 6. The wavelet tree for reorganizing text P

Note here that self-synchronization for compressed text is a property that permits fast search and random access, while its lack means that locating a word in the compressed text needs to sequentially traverse it from the beginning. For example Huffman and Restricted Prefix Byte Codes are not self-synchronized while Tagged Huffman, End-Tagged Dense Code and (s,c)-Dense Code are; however their self-synchronization comes with a loss in efficiency.

The technique proposed by the authors of the paper can be applied to any word-based, byte-oriented semistatic statistical prefix-free compression

algorithm, and it is shown that with this reorganization the compressed text is synchronized even for codes that by nature are not self synchronized and hence all the required advantages (good compression, fast search and random access ability) can be gained simultaneously.

In particular the main idea is to firstly compress the text and then reorganize the different bytes of each codeword laying them out as they appear in the different nodes of a tree structure resembling closely that of a wavelet tree. In this reorganization the first byte of each codeword appears in the root of the wavelet tree, in the same order as it appears in the text, while the root has so many children as are the different first bytes that a codeword can have; inductively at the i-th level, at node v, will be stored the i-th bytes of all codewords that start with the byte sequence from the root to v. It is clear that the produced tree will have depth equal to the maximum length that a codeword can have.

For example let us assume that the text is P="you can compute the complexity in a data structure" and assume that the codewords are as that provided in table 1. In figure 6 it is depicted how the respective wavelet tree is constructed and the bytes of the words of the text reorganized.

In [15] it is proved that by using little extra space, the compressed text can be indexed so that the resultant structure can compete block-addressing inverted indices (the natural choice for indexing natural language texts) and appear as a viable alternative to them; experiments are performed by the authors of the paper to validate such claims.

4.2. In [91], [108] the problem of compressing and indexing highly repetitive sequence collections is considered. Such collections appear in various applications of practice, such as version control systems, and in computational molecular biology for the storage of biological data. In such highly repetitive collections the classical self indexes do not work effectively since the notion of k-th order entropy around which these indices are built, fails to capture the notion of repetitiveness. In order to face this fact the authors present a variant of the wavelet tree data structure that embeds in it run length encoding; the compression method of choice when someone compresses repetitive collections. The result is the so called run length encoded wavelet tree and is described as follows: let R be the number of runs (that is sequences of identical symbols) in the sequence of n characters to be stored (either the initial text or its Burrows-Wheeler transformation). Let B^{all} be the concatenation of all the bit vectors, in a level of the wavelet tree. The bit vector B^{all} is encoded into two separate vectors B^{1} and B^{rl}, with B^{1} storing the starting positions of 1-bit runs, and B^{rl} encoding the run lengths of these runs in unary coding. The value $rank_1(B^{all}, i)$ can be computed, in this modified wavelet tree, by noting that the required answer is the number of 1-bits in $[1, j-1]$ and $[j, i]$, where j is the first place in the 1-bit run that precedes i; these quantities can be computed by appropriate rank and select queries to B^{1} and B^{rl} (see [91], [108]).

By representing succinctly, using the structures of [69] these bit vectors, then the wavelet tree takes a total of $R\log\sigma\log(2n/R)(1+o(1))+O(R\log\sigma\log\log(2n/R))+O(\sigma\log n)$ bits of space, while the various queries take

time $O(t_{BSD}\log\sigma)$, where t_{BSD} is the query time in [69] (u is the length of the handled bit vector and b is its number of 1's) :

$$t_{BSD}(b,u) = O(\min(\sqrt{\frac{\log b}{\log\log b}}\frac{\log\log u \log\log b}{\log\log\log u}, \log\log b + \frac{\log b}{\log\log u}))$$

Moreover the structure, similarly to the various structures mentioned previously for dynamic rank and select, can be made dynamic, by using various available dynamic bit vector representations for B^1 and B^{all}.

4.3 In [10] various techniques are examined to compress a permutation π over n integers, taking advantage of ordered subsequences in π, while supporting the application of $\pi(i)$ and the application of its inverse $\pi^{-1}(i)$ efficiently. The problem is interesting since the techniques presented can be used for various applications and its byproducts improve previous results; it should be remembered that the main idea behind the wavelet tree construction, the range search structure in [20] emerged as an application of mergesort, that can be defined as a series of element permutations. Before proceeding we should note that a *run* in a permutation is defined as the maximal range of consecutive positions that do not contain two consecutive elements in the wrong order. It is shown in [10] that there exists an encoding scheme with space complexity $n(2+H(Runs))(1+o(1))+O(\rho\log n))$ bits that can encode a permutation over the n first integer numbers, that is covered by ρ runs of length $Runs=<n_1, n_2,..,n_\rho>$; here $H(Runs) = \sum \frac{n_i}{n}\log\frac{n}{n_i}$ is the entropy of the runs. The construction supports $\pi(i)$ and $\pi^{-1}(i)$ in $O(1+\log\rho)$ time for any value i in $\{1,\dots,n\}$; if i is chosen uniformly at random then the average time is $O(1+H(Runs))$.

The specific construction uses a wavelet tree that is built as follows: proceed from the root, and handle recursively each child; when at leaves do nothing and when coming from two siblings of a node, merge their permutations in order to produce the permutation at the father and append a 1 bit, when an element is taken from the right child and a 0 bit when an element is taken from the left child. When the construction is finished the permutation has been sorted in $O(n + \rho\log\rho)$ time, plus the total number of bits appended to all bitmaps, and if the wavelet tree is Hu-Tucker shaped then the total number of bits is at most $n(2+H(Runs))$. Using the wavelet tree and the attached bit maps, $\pi()$ and $\pi^{-1}()$ can be computed in time $O(1+\log\rho)$. The construction was improved in [6] where they used their rank and select data structure, and solved the problem in $2nH(Runs)+o(n)(H(Runs)+1)$ bits supporting the queries $\pi()$ and $\pi^{-1}()$ in $O(1+\log\log\rho)$ time.

4.4. Concerning other applications of the wavelet tree in data compression, in [89] it is shown how to use the wavelet tree in order to index efficiently images. Two such techniques are presented one that is based on transforming the problem to that of efficiently handling one dimensional suffix arrays and the other that is based on 2-dimensional suffix arrays and an application of well-known techniques to index images based on so called L-

suffixes. Experimental results in that paper validate the usefulness of the proposed approaches. In particular it is verified experimentally that wavelet trees can be seen as an improvement over the classical bit-plane encoding, and be used to obtain both lossless and lossy compression.

Use of the wavelet tree structure also appears in [4], [104] where it is applied as a range structure, for their compressed self-index on texts compressed with the LZ78 data compression, and in [82] as a component of various substructures, in their implementation of a self-index handling texts compressed with the LZ77 data compression algorithm. The last application is very interesting since LZ77 can benefit from the existence of frequent repetitions in a text, and as discussed previously these repetitions, appear in various applications. It is mentioned that the proposed self-index is smaller (up to one half) than the best current self-index for repetitive collections, and in many cases it is also faster.

Finally, in [30] it is depicted how the use of wavelet trees can support efficient representation and navigational operations in a compressed representation of the Web graph (for other approaches see [29], [17]). In particular the paper represents the adjacency list of each node of the graph as a sequence that is compressed using the re-pair [83] compression technique, and then it is handled by two alternative representations for rank and select, the structure in [57] and a wavelet tree, in order to support the operations of retrieving direct and reverse neighbors. Experiments with these two approaches and with various alternatives, depict that the wavelet tree representation though slower (sometimes by an order of magnitude) from its alternatives achieves the smaller space reported in the scientific literature (in comparison with similar techniques to represent the web graph), and can be considered to be the practically better choice.

5. Applications in Information Handling

5.1 In [18] the relationship between wavelet trees and range searching data structures, that in plain words makes the wavelet tree equivalent to an orthogonal range searching data structure suitable for handling points with discrete coordinates, was pushed a bit further by placing them as a viable alternative to spatial data structures that use and store minimum bounding rectangles in order to handle various spatial queries as they appear in geographical information system. In that paper experiments were performed comparing the proposed structure with variants of the R-tree family (the R*-tree and the STR R-tree) both in space and in time performance in synthetic and real datasets. Concerning the space complexity the structure needs less space than the R*-tree in both datasets, while it needs less space than the STR R-tree in real datasets, and a comparable space in synthetic. Concerning the time complexity if the dataset is uniform then the wavelet tree outperforms both variants of the R-tree while in the case that it is not it outperforms them for very selective queries.

In particular consider a set of *N* objects that are represented by their Minimum Bounding Rectangles (MBR) as defined by their lower left and their upper right corners (each corner is a pair of coordinates in x- and y- axis).

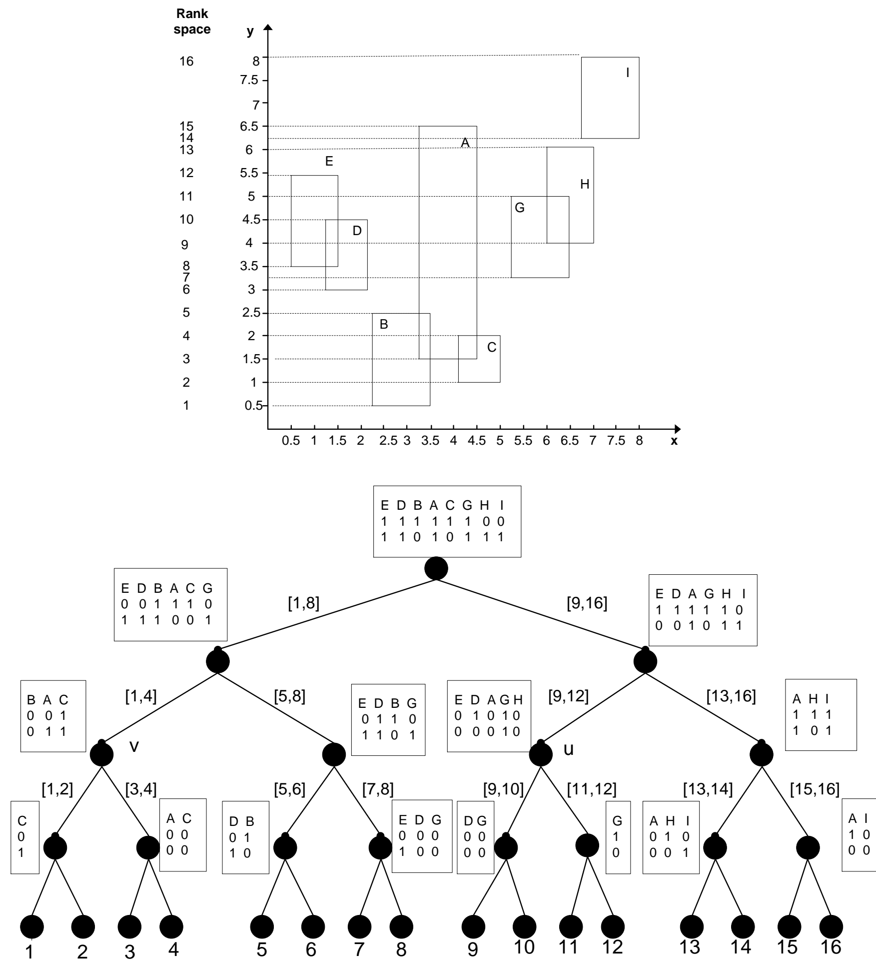

Fig. 7. The wavelet tree for representing a set of maximal Minimum Bounding Rectangles {A, B, C, D, E, F, G}

The proposed structure handles intersection queries by initially splitting the set of MBRs into *k maximal* sets and then building a wavelet tree separately for each maximal set. A set of MBRs is termed *maximal* if it is not possible for a projection over the x-axis of a rectangle in this set to contain the projection of another rectangle in the same set. Given a set of *N* rectangles the optimal partitioning to maximal sets can be performed in *O(NlogN)* time, by a procedure that is related to the longest increasing subsequence problem, and to the problem of decomposing permutations.

So let us suppose that the set at hand is maximal, this set can be represented in a square grid of *2N* rows and columns, with each corner point in each row and column. This grid can be represented simply in rank space, by storing the real coordinates of the MBRs in suitable structures, in order to translate a given query rectangle to the discrete coordinates in the rank space. Because the handled set is maximal, the order in the x-coordinate of the left and right sides is the same, hence two sorted arrays for the *x*-coordinates and one sorted array for the *y*-coordinates, plus arrays storing the identifiers of the MBRs in the same order suffice for the translation from the geographic space to rank space.

The constructed wavelet tree is built according to the *y*-coordinates, and each node of the tree stores the respective MBRs according to the *x*-order, with the novelty that instead of one, two bit vectors are used. The first bit vector projects onto the left child and has the value 1, if the respective MBR is processed in the left child, and the second bit vector projects onto the right child and has the value 1, if the respective MBR is processed in the right child.

An MBR is processed in a node if it intersects the *y*-range corresponding to the node, and hence an MBR can be processed in both the left and the right child of a node. In order to avoid the quadratic space explosion due to the fact, that an MBR could be stored in a linear number of nodes at a level, a modification in the structure is presented that resembles that of a segment tree, by demanding that when an MBR covers the whole range of a node, then it is not stored in any of its descendants.

Figure 7 depicts the construction for a set of maximal MBRs. In the upper part of the figure the set of rectangles is depicted plus the corresponding rank space for the y-coordinates, while in the constructed wavelet tree the bit vector for the left child of a node is depicted above the bit vector for the right child, In node *u* for example rectangle *D* intersects only the left and not the right child's range, hence it stores 1 in the first bit vector and 0 in the second, while concerning the space saving heuristic, in node *u* rectangles A, E and in node *v* rectangle *B*, contain the whole range of the respective nodes, hence the bits for both bit vectors are equal to 0.

The above structure and assuming that it stores a maximal set of MBRs can answer range queries by appropriately descending paths of the tree according to the provided query range, thus giving a time complexity of *O(logN)* for each path traversed. In particular the authors provide a recursive algorithm that projects the initial query range at a node to suitable query ranges in the first and right child by applying rank operations in the two bitmaps.

In the general case and since we have at our disposal *k* maximal sets, we need to query *k* wavelet trees, and hence the total time complexity of the tree paths traversed is bounded by *O(klogN).*

5.2. In [106] the wavelet tree is proposed as an index in order to implement bidirectional search in a string. The motivation for this application comes from the area of Bioinformatics and in particular genome encoding where after finding suitable matches, both ends of the string need to be extended by

exploiting the complementarily in the bases. This kind of search can be implemented by using bidirectional search. Indices performing this kind of search have been proposed in the bibliography in the form of affix trees and affix arrays [109], [110]. The affix tree of a string is comprised by its suffix tree and the suffix tree of its reverse, while the affix array combines the suffix arrays of the string and the suffix array of its reverse. Generally, when answering queries, it is difficult to implement the interplay of these structures while the space complexity is usually large. In order to face these shortcomings the authors in [106] present the bidirectional wavelet index of a string which consists of two wavelet trees, the wavelet tree of the Burrows-Wheeler transformation of the string, and the wavelet tree of the Burrows-Wheeler transformation of its inverse. The difficult part in using these structures is how to synchronize the search on both indices, but the authors in [106] depict, that by a carefully designed search procedure it is possible to perform bidirectional search. Performed experiments depict that the proposed index decreases the space requirement by a large constant factor of 21 (in comparison to affix arrays) making it possible to apply their algorithm in very large strings.

5.3. In [5] the wavelet tree is extended in order to support binary relations [7]. Binary relations are an important abstraction that is inherent in many combinatorial objects such as trees, graphs, inverted lists, and web graphs. In particular a binary relation B between a set of objects with identifiers in $[1,n]$ and a set of labels with identifiers in $[1,\sigma]$ can be considered as a set of t pairs from a total universe of $n\sigma$ possible pairs; it can also be represented as a matrix with σ rows (the labels) and n columns (the objects). The notion of entropy can be extended to handle these combinatorial objects by defining it as:

$$H(B) = \log\binom{n\sigma}{t} = t\log\frac{n\sigma}{t} + O(t).$$

In the specific paper besides listing operations of potential interest and giving reductions between operators, two data structures are presented. The first data structure uses the reduction of binary relation operators to string operators by representing a binary relation with a bitmap $B[1,n+t]$ concatenating the cardinalities of the columns of the relation in unary, and with a string $S[1,t]$ over the alphabet $[1,\sigma]$ containing the labels of the pairs in column major order. Using this representation it is possible to get an efficient implementation of a binary relation that supports the various operations in $O(\log\sigma)$ time, and needs $t\log\sigma+o(t)\log\sigma+O(\min(n,t)\log((n+t)/\min(n,t))$ bits of space.

The second structure is the so called *Binary Relation Wavelet Tree*. This tree is a wavelet tree like construction with the leaves corresponding to individual rows of the relation, and with each node containing two bitmaps per level. The first bitmap corresponds to objects in the left child, and has its i-th bit equal to 1, if there exists an object in the left child with label having identifier i, while the second bitmap corresponds to objects in the right child and has its i-th bit equal to 1, if there exists an object in the right child with

label having identifier *i*. Note the similarity of the construction, with that in the extension of the wavelet tree in order to handle spatial objects; here also it is possible for an object *x* to propagate both left and right and this means that the sizes of the bitmaps at a level may add up to more than *n* bits. The authors prove that the specific construction use $log(1+\sqrt{2})tH(B)+O(tH(B))+O(t+n+\sigma)$ bits of space, and can support the various operations in $O(log\sigma)$ time.

5.4 In [53] (see also [51]) it is described how to use wavelet trees in order to support efficient range quantile queries in a list *S* of *n* numbers. A range quantile query takes a rank and the endpoints of a sublist and returns the number with that rank in that sublist. More analytically the wavelet tree is built for the list of numbers together with the appropriate rank and select data structures at the bit vectors of its nodes. Suppose that we are given the endpoints *l* and *r* of a sublist, and a rank *k*, and we want to report the element with rank *k* in the sublist between *l* and *r*.

Starting from the root, and accessing its binary vector *B*, we apply two rank queries with arguments *l-1* and *r*, in order to find the number of 0s and 1s in *B[1,l-1]* and *B[l,r]*. If there are more than *k* zeroes in *B[l,r]* then we have to move to the left subtree, and we set *l* to be one more than the number of 0s in *B[1,l-1], r* to be the number of 0's in *B[1,r]* and we recurse on the left subtree, otherwise we have to move to the right subtree of *T*, hence the new *k* is the old *k* minus the number of 0s in *B[l,r],* the new *l* is one more than the number of 1's in *B[1,l-1],* we set *r* to be equal to the number of 1s in *B[l,r]* and we recurse to the right subtree; when a leaf is reached its label is returned. If *σ* is the total number of distinct elements (note that this number is smaller than both *n*, and the size of the universe from which the elements take value) then the tree has height $O(log\sigma)$, and since accessing each node's binary vector costs O(1), it follows that the time complexity is $O(log\sigma)$. Concerning space, the cost is equal to the space complexity of the wavelet tree, hence depending on the constant time dictionaries that are used for the bit vectors, it can have different values from $O(nlog\sigma)$ to $nH_0(S)+O(nloglogn/log_\sigma n)$ bits.

As a working example consider figure 8, depicting a wavelet tree for the sequence *S=15,14,1,5,6,4,11,12,13,8,9,7,16,2,3,10* and assume that we would like to locate the 4-th element in *S[3..11]*. We start from the root *r*, and compute $rank_0(B_r, 2)=0$, και $rank_0(B_r, 11)=5$, hence the number of 0s and 1s in *S[3,11]* is 5 and 4 respectively; since the number of 0s is larger than 4, we go to *v*, looking for the 4th element in *[1,5].* There we have $rank_0(B_v,0)=0$ and $rank_0(B_v,5)=2$, hence the number of 0s and 1s in [1,5] is 2 and 3 respectively; since the number of 0s is less than 2, then we have to move to *w*, looking for the $4\text{-}2=2^{th}$ element in [1,3]. In *w* we have $rank_0(B_w,0)=0$ and $rank_0(B_w,3)=2$, hence the number of 0s and 1s in [1,3] is 2 and 1 respectively; since the number of 0's is equal to 2 we move to the left child *u* of *w* looking for the 2th element in [1,2]. We have $rank_0(B_u,0)=0$ and $rank_0(B_u,2)=1$ hence the number of 0,s and 1s in [1,2] is 1 and 1 respectively; since the number of 0's is less than 2 we move to the right and the answer is affirmatively the number 6.

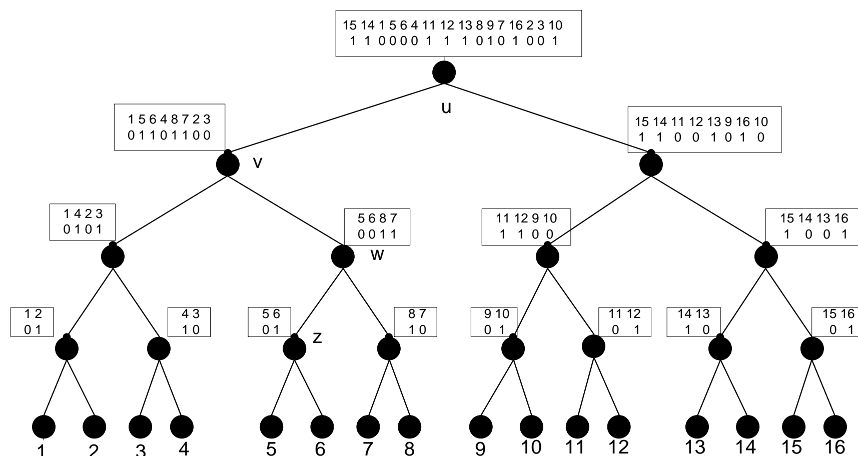

Fig. 8. A wavelet tree for the sequence S=15,14,1,5,6,4,11,12,13,8,9,7,16,2,3,10

The aforementioned structure can be generalized to any constant number of dimensions, by using the same base tree structure on the distinct elements, and by employing a multidimensional range counting structure in the internal nodes. If the structure is the one in [80] then it is proved in [53] (in the full version of the paper appearing as technical report) that for any positive constants d and t there exists a structure of size $O(n^{1+t}log\sigma)$ bits, that can answer d-dimensional range quantile queries in $O(log\sigma)$ time.

As another application of the solution to the 1-dimensional range quantile query, the above algorithm can be extended in order to enumerate the f distinct items in a given sublist in $O(flog\sigma)$ time. The reporting algorithm, can find the first element in a given range, in $O(log\sigma)$ time with a range quantile query, and then exploiting the fact that in $O(log\sigma)$ time a wavelet tree can compute the number of occurrences of a symbol in the prefix of a sequence, it can transform the search of successive elements by a suitable range quantile query costing $O(log\sigma)$ time. This operation is important since it can be used in the document listing problem [93].

In the document listing problem we are given a set of n documents of total size N characters from an alphabet Σ of size σ (without word separation) and we want to locate the documents that contain a pattern. In [93] a solution was provided using a suffix tree and a document array, being accompanied by a compressed array, using space $O(Nlogn)$ bits and $O(m+t)$ query time, where m is the size of the seeked pattern and t denotes the output size. The space was reduced in [111] by employing a compressed suffix array and a wavelet tree to represent the document array plus a range minimum query structure and the query time was worsened to $O(mlog\sigma+tlogn)$, where t denotes the size of the output. In particular the space complexity was improved to $|CSA|+2N+Nlogn(1+o(1))$ bits, where $|CSA|$ is the size of a suitable compressed array and is less than $Nlog\sigma(1+o(1))$ bits. The aforementioned extention of the range quantile query permits the use of only the wavelet tree

structure without the accompanying machinery (the range minimum data structure) thus reducing even further the space of the solution in [111] (by $N+o(N)$ bits), while keeping the same time complexity. It should also be noted that in [111] a remark concerning autocompletion search [11], was also made. In particular it was noted that the structure presented in [11] can also be used for the document listing problem with the same space but worse time complexity. It is worth to be mentioned that the autocompletion search structure provided in [11] is similar to the wavelet tree construction, though different in its functionality and use of accompanying bit vectors.

A problem closely related to the document listing problem is the top-k query problem [33], [76], [81], in which we want to locate the top k documents where a pattern appears more often (top-k retrieval). Wavelet trees fit nicely in this extension of the problem since they can be used to compute pattern frequencies and report rankings. In particular in [33] the range quantile query approach in [53] was extended with two heuristics, that permitted the effective calculation of frequencies and top-k results reporting; the greedy traversal heuristic and the quantile probing heuristic. Both were tested experimentally and the greedy heuristic stood satisfactory as a competitor to inverted files, for natural language query processing. The above two heuristics do not come with bounded time estimations; this does not hold for the solution in [76], where it is shown that $O(m+k\log^{3+\varepsilon} N)$ query time can be attained using $|CSA|+o(N)+n\log(N/n)$ bits; as noted in [96] wavelet trees can be used in this solution as a substructure to count frequencies.

Finally, in [96] a further development was achieved by reducing the space complexity of the wavelet tree involved in the above constructions (see also for a similar attempt in [52]). This space compaction was achieved by applying grammar compression (re-pair compression [83]) to the bitmap dictionaries and then employing these compressed representation for storing the subsequences at each node of the wavelet tree. The proposed structure is shown experimentally that it achieves great space savings at the cost of a somehow inferior query time; moreover combinations of this structure with the approaches in [33], [76] are tested.

Concluding in [81] wavelet trees were engaged providing an asymptotically optimal solution for the top-k color problem. That is, consider an array with N elements each with a color and a priority, then there exists an $O(N\log c)$ bits data structure that reports k colors with the highest priority in a query interval in $O(k)$ time; here c denote the number of different colors. This solution can be exploited for the *top-k* document retrieval problem leading to an algorithm with $O(N\log N)$ bits and $O(m+k)$ query time.

Before closing, it should be noted that a further extension to the above problems was presented in [74] where it was shown how wavelet trees can be effectively applied in conjunction with the multiway search paradigm introduced in [32] in order to solve the document listing problem in the case when the query consists of $m>1$ patterns $\{P_1, P_2, .., P_m\}$. The proposed solution needs linear space and can answer these queries in time $O(p)+\tilde{O}(t^{1/m} n^{1-1/m})$, where p is the total length of the patterns and t is the size of the output. Moreover in the case of two patterns the achieved bound is

$O(|P_1|+|P_2|+\sqrt{nt}\log^2 n)$. It is also shown how other problems dealed with in [93] can be efficiently handled. A basic ingredient of the aforementioned construction is a variant of the wavelet tree, the so called *weight balanced wavelet tree*, that is interesting ın its own right, since it is different from other wavelet tree implementations. In this variation it is explicitly settled during construction that the number of 0's and 1's at the bit vector at each node is almost equal and hence the *total* number of symbols (and not the number of *distinct* symbols as usual) stored at each child of a node is almost equal. In particular the construction is performed top down, and at each node the symbols are processed in decreasing frequency order, and a symbol is going left or right depending on where the total number of symbols is bigger (right or left respectively). The weight balanced wavelet tree has the property that for every node at depth *d*, the corresponding bit vector has size at most $4n/2^d$, moreover its space consumption is bounded by the zero entropy of the stored sequence, without even needed to compress the bit vectors.

5.5 Using similar techniques on the wavelet tree [51] as in the range quantile query, it is possible to solve the *range next value* problem and the *range intersect* problem.

Let us first consider the range next value query on a wavelet tree *T*. In this problem we are given two endpoints *l, r* of a query interval plus a value *x*, and we want to compute the smallest value greater or equal to *x*. Initially we start from the root, if *x* is at a leaf at the right subtree of the root, we move to the right child by setting as new left value $rank_1(B_{root}, l-1)+1$, and as new right value $rank_1(B_{root},r)$; otherwise the search continues with the left child by setting as new left value $rank_0(B_{root},l-1)+1$, and as new right value $rank_0(B_{root},r)$. If at some point the interval becomes empty the recursion stops returning no value. If the recursion returns from the right child of the root, the answer is that no value exists, otherwise it could be possible that a number exists in the T_r, hence a final attempt is made to T_r, but in this time with a query that demands the minimum value in the sublist between *l* and *r*. This query is a special case of the range quantile query and hence can be handled as described previously. The whole approach overall needs to follow a path in the wavelet tree, performing constant time queries in each visited node, and thus the search time is $O(\log\sigma)$, with σ being the number of distinct values in the list. The space complexity is $n\log\sigma+O(n)$ bits.

A similar problem, termed *range successor* problem was handled in [113], [114]. In this problem we are given a set of *n* points in a grid of distinct coordinates, and we would like to preprocess them so that for any given orthogonal range, we can report the point with the smallest *y*-coordinate. One of the solutions presented in [113], [114] takes $O(\log n/\log\log n)$ time to answer the query, needs $O(n)$ words space and makes heavy use of wavelet trees, improving a previous $O(\log n)$ query time construction presented in [90]. The construction is basically a multiway wavelet tree with branching factor $(\log n)^{1/2}$. In each node a substructure is attached that is also proposed by the authors that can handle $m<=n$ integers with range $r=O(\log n/\log\log n)$, in $O(m\log r)$ bits, so that together with an $o(n)$ bit table can handle range successor queries in $O(1)$ time. By effectively embedding the structure in the

multiway wavelet tree, they show how to achieve the aforementioned time and space bounds.

In the *range intersect* problem, given k ranges in a list, we want to find the distinct common values in these ranges. In order to handle this problem we will first consider the case that we have two ranges; the case of multiple ranges can be handled easily by reducing it to multiple applications of the problem of intersecting two lists or it can be handled by an immediate extension of the basic procedure. The problem is extremely interesting since it has various applications and resembles the problem of intersecting the posting lists in an inverted file. In this kind of problems the inherent complexity is captured through a measure termed alternation α, which can be defined as the number of switches from one list to the other in the sorted union of the two ranges.

Hence assume that the two ranges are $[l_1,r_1]$ and $[l_2, r_2]$. The procedure starts from the root of the tree and tests if either of the ranges is empty, if they are the procedure stops, otherwise, by using the bit vectors at the root the algorithm descends to the left and to the right child, and continues recursively to the nodes where both ranges are non empty. When a leaf is reached then the corresponding element is in the intersection and we report its respective number of occurrences in the first and in the second list. The same procedure can be applied for multiple lists, and descending a path is stopped when one of the ranges becomes empty.

It is proved in [51] that the specific algorithm can compute the intersection of k ranges in time $O(\alpha k \log(\sigma/\alpha))$ where k is the number of ranges, α is the alternation complexity of the problem and σ denotes the number of distinct values in the sequence.

Another algorithm given in [51] for the same problem, uses two variables x_1 and x_2, that scan the two lists simultaneously, and move through calls of the range next value procedure, thus simulating the well known merge procedure of two sorted lists, with the difference that the movement is achieved via calls of the range next value procedure; if at any time x_1=x_2, then an intersection is reported and the procedure continues by setting as x_1 the range next value in the first list of x_2+1. The specific procedure makes a switch each time an alternation is met and therefore the time complexity is $O(\alpha\log\sigma)$. This can be improved to $O(\alpha\log(\sigma/\alpha))$ by implementing the range next value procedure, so that it remembers the path that was traversed the last time, in a form of fingered search and thus getting a time complexity of $O(\alpha\log(\sigma/\alpha))$.

5.6. In [51] and [95] it is shown how to use the wavelet tree in order to implement the search procedure that is needed when processing the posting lists of an inverted file. Inverted files are the method of choice when creating indices for both ranked and boolean retrieval.

In order to have both modes of inverted files operations operated in minimum space, it is needed to efficiently retrieve the documents both by decreasing weights (permits effective handling of queries) and by increasing document number (permits effective compression). In [95] and [51] it is depicted how to use the wavelet trees in order to implement efficiently, in the

space required for a single compressed inverted index, the operations of retrieving efficiently the documents either in decreasing weight, or in increasing identifier order.

The construction basically concatenates and sorts all the posting lists of the terms (these should contain the document identifiers where a term appears in descending order of the term frequency of the specific term), thus creating a sequence list of document identifiers, and this sequence is stored in a wavelet tree. In parallel for each term in the collection, it is stored the starting position of its list in the sequence of identifiers, and the term frequencies are stored in differential and run-length compressed form in a separate sequence. Finally the starting positions of the list of each term in the concatenated sequence are stored in separate bitmap preprocessed for rank and select queries.

Let D be the total number of documents in the collection, let L be the length of the list with all document identifiers and let N be the total length in words of the text collection. Then, it is shown that the proposed construction takes space $NH_0(L)+o(NlogD)$ bits plus the cost for storing the various term frequency values in differential and run length compressed form. It is proved that with this construction the various operation of interest in both boolean retrieval and ranked retrieval can be efficiently supported.

Another construction was also employed in [3] for handling the posting lists retrieval problem, that needs $NH_0(L)+o(N)(H_0(L)+1)$ bits of space, such that a conjunctive query of k terms can be performed in $O(k\alpha loglog\sigma)$ time. This construction is mainly focused on the conjunctive query problem and represents the set of documents as a concatenated sequence, but this time with alphabet the distinct terms and employs a rank and select structure for arbitrary alphabets in order to handle it. The aforementioned bounds are valid if the fast rank and select structure proposed in [6] is used, however a valid alternative (especially in a practical implementation) is that of Huffman shaped wavelet tree. In general, using the wavelet tree structure one is able to retrieve the frequency information of a term, its positional information, the respective posting list, and answer conjunctive queries. Moreover the authors show how to retrieve effectively snippets enclosing the appearances of a given term. In comparison to the inverted files approach the wavelet tree though needing more time, need less space, and can providing more functionalities.

In [2] the approach of [3] is extended in order to handle the situation where the documents are stored in an P x D array of search node processors (P denotes the number of text partitions, and D denotes the replication level inside each partition). It is depicted both theoretically and experimentally that wavelet trees can efficiently work in this setting, while they can be used to dynamically maintain a cache with the most recently used posting lists. In particular it is shown that all processing phases (index decompression, ranking and snippet extraction) can be performed by a simple unifying structure thus permitting the use of less processors for the same performance or putting out it differently permits with the same number of processors better performance. The above is achieved by unifying into a

single unit the pair of search nodes and document servers. Moreover besides employing the wavelet tree's ability to generate efficiently snippets the authors exploit its capability to produce posting lists and compute intersections in combination with caching strategies. In this case it is assumed that a term partitioning approach is applied in the replicas, and the employed wavelet trees are used to support various functionalities on a dynamically maintainable cash.

5.7 The wavelet tree has also found various applications in the retrieval of XML documents; see for example [1], [13], [41], [42], [51]. From these papers the most interesting are the constructions provided in [13] and [51].

In [13] the hypothesis is made that the XML tree can be modeled as a tree with document identifiers (that is passages of text) appearing only to the leaves of the tree, and with internal nodes being mapped to structural separators. The structure used for indexing is a parenthesis representation of the tree modeled as a sequence that is obtained through a preorder traversal. To each parenthesis a tag name is mapped and the sequence of tag names is represented using a wavelet tree; moreover for each distinct tag name a parenthesis representation of the nodes of the XML tree that are tagged by it, is stored. The total space for all these structures is $2n\log\tau+O(n)$ bits where n is the number of leaves of the tree (the basic text structures) and τ, is the number of distinct tags in the collection.

In [13] it is proved, that this representation can answer effectively various operations on the XML structure such as: locating the lowest ancestor of a text passage (leaf) where an occurrence appears, and return the range of text passages corresponding to this ancestor, listing text passages restricted to structural unit tagged with a specific name, counting number of occurrences of a query in a retrievable node, and computing intersections on retrievable units.

Finally, an interesting combination of the XML machinery and the wavelet tree literature appears in [51] where a novel structure the XML wavelet tree, is proposed as a self index for XML documents; this structure can handle XPath queries more efficiently than using the uncompressed counterparts, and also appears to be more competitive than inverted indices of the same space consumption. The idea is based in a combination of the (s,c)-DC compressor [16] with the wavelet tree. In particular the (s,c)-DC compressor is initially used in order to compress the XML document, and then the idea of [15] that reorganizes the compressed codewords in a structure resembling a wavelet tree is applied, in order to facilitate immediate access and retrieval of the appropriate codeword. Experiments performed in [51] validate the practical benefits of the proposed approach.

6. Conclusions and open problems

Wavelet trees represent a clear example where a data structure designed for a theoretical construction (Chazelle's work [20]) benefits the practitioners

(designers of data compression algorithms) providing them with tools for efficiently handling several practical problems. Besides improving the bounds in the aforementioned uses of the wavelet tree (the majority of them constitutes in these days hot research topics and are the target of active research in the area of algorithms and information retrieval), it would be interesting to find out other applications of the specific technique and explore issues of extending the structure itself. Possible areas of future extensions could employ: self adjusting heuristics, embedding machinery of persistent data structures, compressing in various ways the structure itself (see examples for such approaches in [96], [81]) and cache oblivious extensions.

In particular weight balanced wavelet trees and frequency based wavelet tree have been proposed in [74], [49] for various applications, it would be interesting to see how these approaches combine together, and explore the suitability of splay heuristics if being applied to these structures (the frequency based heuristics of [49] provide such a clue, while it should be noted that splay trees are a natural data structural choice in various compression settings). Moreover compression algorithms of the self-indexed family generally fail to deal with repetitions in the text [91], [108] a phenomenon that could probably be dealt effectively by applying techniques from the persistent machinery [35]. Therefore it would be worthwhile to study the way that persistent techniques, in all of its various manifestations (partial, full, confluent) could be applied in wavelet trees. Moreover in [81] it has been shown (and it is trick that is used main times) how to jump levels of the structure in order to speed up it the descend of its levels; it would be nice to have this as a general technique that could be applied as a general framework in the various of the structure’s applications. Finally since the structure itself is easy to be implemented in secondary memory [62], [75], it would be interesting to have a study of its various applications in the more advanced cache oblivious model.

Concluding, it would be also worthwhile to find out if achievements and research progress in the area of geometric search algorithms could possibly embed new tactics and methodologies, in the efficient maintenance of wavelet trees, since the roots of the structure are from this computer science research field. Especially in the last years, in the computer science literature, there are a lot of algorithms and variants of techniques dealing with the range searching problem, and it should be explored how these achievements reflect on the wavelet tree.

Acknowledgements The author would like to thank the two anonymous referees for their valuable suggestions that helped to improve the quality of the paper. Moreover, this research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

1. Arroyuelo, D., Claude, F., Maneth, S., Makinnen, V., Navarro, G., Nguyen, K., Siren, J., Valimaki, N.: Fast in-memory XPath search over compressed text and tree indexes. ICDE, 417-428, (2010).
2. Arroyuelo, D., Gil-Costa, V., Gonzalez, S., Marin, M., Oyarzun, M.: Distributed search based on self-indexed compressed text. Information Processing and Management (2011).
3. Arroyuelo, D., Gonzalez, S., and Oyarzun.: Compressed self-indices supporting conjunctive queries on document collections. In SPIRE, LNCS 6393, pp. 43-54 (2010).
4. Arroyuelo, D., Navarro, G.: Space-efficient construction of Lempel-Ziv compressed text indexes. Information and Computation 209, 1070-1102 (2011).
5. Barbay, J., Claude, F., Navarro, G.: Compact Rich Functional Binary Relation Representations. LATIN, pp. 170-183 (2010).
6. Barbay, J., Gagie, T., Navarro, G., Nekrich, Y.: Alphabet partitioning for compressed rank/select and applications. ISAAC (2), 315-326 (2010), see also the version in *CoRR, abs/0911.4981, 2009.*
7. Barbay, J., Golynski, A., Munro, J., Rao, S.: Adaptive searching in succinctly encoded binary relations and tree structured documents. Theoretical Computer Science, 387 (3): 284-297, (2007).
8. Barbay, J., He, M., Munro, J., Rao, S.: Succinct indexes for strings binary relations and multi-labeled trees. ACM/SIAM SODA, 680-689 (2007).
9. Barbay, J., and Kenyon, C., Adaptive intersection and t-threshold problems. In SODA, pages 390-399, 2002
10. Barbay, J., Navarro, G.: Compressed representations of permutations and applications. Symposium on Theoretical Aspects of Computer Science. 111-122 (2009).
11. Bast, H., Mortensen, C.W., Weber, I.: Output sensitive autocompletion search. Information Retrieval. 11:269-286 (2008).
12. Bose, P., He, M., Maheswari, A., Morin, P.: Succinct orthogonal range search structures on a grid with application to text indexing. WADS, 98-109 (2009).
13. Brisaboa, N., Cerdeira-Pena, A., Navarro, G.: A compressed self-indexed representations of XML documents. ECDL, 273-284 (2009).
14. Brisaboa, N., Cillero, Y., Farina, A., Ladra, S., Pedreira, O.: A New Approach for Document Indexing using Wavelet Trees. DEXA Workshops, 69-73 (2007).
15. Brisadoa, N., Farina, A., Ladra, S., Navarro, G.: Reorganizing Compressed Text. ACM SIGIR, 139-146 (2008).
16. Brisaboa, N., Farina, A., Navarro, G., Esteller, M.: (s,c)-Dense Coding: an optimized compression code for natural language text databases. SPIRE, pp. 122-136, (2003).
17. Brisaboa, N., Ladra, S., Navarro, G.: k^2-trees for Compact Web Graph representations. SPIRE, pp. 18-30, (2009).
18. Brisaboa, N., Luaces, M., Navarro, G., Seco, D.: A fun application of compact data structures to indexing geographic data. FUN, 77-88 (2010).
19. Brisaboa, N., Luaces, M., Navarro, G., Siego, D.: A new point access method based on wavelet trees. ER Workshops, 297-306 (2009).
20. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM J. Computing, 17(3), 427-462 (1988).
21. Chan H.L., Hon W.K., Lam, T.W., and Sadakane K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms, 3(2):article 21, 2007.

22. Chien, Y.F., Hon, W.K.: Geometric Burrows-Wheeler transform: linking range searching and text indexing. In Proceedings of the 2008 IEEE Data Compression Conference (DCC '08), Snowbird, UT, March (2008).
23. Chien, Y.F., Hon, W.K., Shah, R, Vitter, J.S.: Compressed Text Indexing and Range searching. TR 2006-21, Purdue University, (2006).
24. Chiu, S., Hon, W., Shah, R., Vitter, J.S.: I/O Efficient Compressed Text Indexes: From Theory to Practice. Data Compression Conference, 426-434 (2010).
25. Clark, D.: Compact Pat Trees. Phd thesis, University of Waterloo, (1996).
26. Claude, F., Navarro, G.: Self-indexed text compression using straight-line programs. In: Proc. MFCS, pp. 235-246, LNCS (2009).
27. Claude, F., Farina, A., Martinez-Prieto, M., Navarro, G.: Compressed q-gram indexing for highly repetitive biological sequences. IEEE International Conference on Bioinformatics and Bioengineering, 426-434 (2010).
28. Claude, F., Navarro, G.: Practical Rank/Select Queries over Arbitrary Sequences. SPIRE, 176-187, (2008).
29. Claude, F., Navarro G.: Fast and compact Web graph representations. ACM Transactions on The Web, Vol. 4, No. 4, Article 16, Pub. date: September (2010).
30. Claude, F., Navarro, G.: Extended compact web graph representations. Algorithms and Applications, pp. 77-91 (2010).
31. Claude, F., Nicholson, P., Seco, D.: Space efficient wavelet tree construction. In SPIRE'11 (2011), pp. 185-196.
32. Cohen, H., Porat, E.: Fast set intersection and two-patterns matching. In LATIN, LNCS 6034, pp. 234-242, (2010).
33. Culpepper, S., Navarro, G., Puglisi, S., and Turpin, A.: Top-k ranked document search in general text databases. ESA, pp. 194-205, (2010).
34. Deorowicz, S.: Second step algorithms in the Burrows-Wheeler compression algorithm, Software- Practice and Experience. 32:99-111 (2002).
35. Driscoll, J., Sarnak, N. Sleator, D.D., Tarjan, R.E.: Making Data Structures Persistent. J. Comput. Syst. Sci. 38(1): 86-124 (1989)
36. Farina, A., Ladra, S., Pedreira, O., Places, A.: Rank and select for succinct data structures. Electr. Notes Theor. Comput. Sci. 236: 131-145 (2009).
37. Farzan, A., Gagie, T., Navarro, G.: Entropy bounded representation of point grids. ISAAC: 327-338 (2010).
38. Ferragina, P., Giancarlo, R., Manzinni, G.: The Myriad Virtues of Wavelet Trees. ICALP, pp. 561-572 (2006), journal version in Information and Computation 207 (2009), 849-866.
39. Ferragina, P., Giancarlo, R., Manzini, G.: Sciortino, M.: Boosting textual compression in optimal linear time. Journal of the ACM, 52(4),pp. 688-713, (2005).
40. Ferragina, P., Gonzalez, R., Navarro, G., Venturini, R.: Compressed Text Indexes: From theory to practice. ACM Journal of Experimental Algorithmics 13: (2008).
41. Ferragina, P. Lucio, F., Manzinni, G., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness and beyond. IEEE FOCS, pp. 1-9,(2005).
42. Ferragina, P., Lucio, F., Manzini, G., Muthukrishnan, S.: Compressing and Searching XML data via two zips. WWW, pp. 751-760, (2006).
43. Ferragina, P., Manzini, G.: Indexing Compressed Text. Journal of the ACM, 52(4), pp. 552-581, (2005).
44. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: An Alphabet-Friendly FM-Index. SPIRE: 150-160, (2004).

45. Ferragina, P., Manzini, G., Makinen, V., Navarro, G.: Compressed representation of sequences and full-text indexes. ACM Transactions on Algorithms 3(2): (2007)
46. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. SODA: pp. 690-696, (2007).
47. Ferragina P., Venturini, R.: The compressed permuterm index. ACM SIGIR, 535-542, (2007).
48. Fischer, J.: Data structures for efficient string algorithms. Phd Thesis, (2007).
49. Foschini, L., Grossi, R., Gupta, A., Vitter, J.: When indexing equals compression: experiments with compressing suffix arrays and applications. ACM Transactions on Algorithms, 2(4), pp. 611-639, (2006).
50. Foschini, L., Grossi, R., Gupta, A., Vitter, J.: Fast compression with a static model in high-order entropy. Data Compression Conference: pp. 62-71 (2004).
51. Gagie, T., Navarro, G., Puglisi, S.: New Algorithms on Wavelet Trees and Application to Information Retrieval, SPIRE (2009), 1-6, also as full version in CoRR abs/1011.4532: (2010)
52. Gagie, T., Navarro, G., Puglisi, S.: Colored range queries and document retrieval. SPIRE, pp. 67-81 (2010).
53. Gagie, T., Puglisi, S., Turpin, A.: Range quantile queries: another virtue of wavelet trees, In Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 5721, pp. 1-6, (2009), also as arXiv:0903.4726v6.
54. Golynski, A.: Optimal lower bounds for rank and select indexes. In Proc. ICALP, 370-381, (2006).
55. Golynski, A., Grossi, R., Gupta, A., Raman, R., Rao, A.: On the size of succinct indices. ESA, pp. 371-382, (2007).
56. Golynski, A., Raman, R., Rao, S.: On the redundancy of succinct data structures. SWAT, pp. 148-159, (2008).
57. Golynski, A., Munro, J., Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. SODA, pp. 368-373, (2006).
58. Gonzalez, R., Grabowski, S., Makinnen, V., and Navarro, G.: Practical implementation of rank and select queries. In the 4th Workshop on Efficient and Experimental Algorithms (WEA), 2005, pp. 27-38.
59. Gonzalez, R., Navarro, G.: Improved dynamic rank select entropy-bound structures. LATIN 2008, LNCS 4957, pp. 374–386, (2008) (also as Rank/Select on Dynamic Compressed Sequences and Applications, Theor. Comput. Sci. 410(43): 4414-4422 (2009))
60. Gonzalez, R., Navarro, G.: Compressed Text Indexes with Fast Locate. CPM, pp. 216-227, (2007).
61. Gonzalez, R., Navarro, G.: Statistical encoding of succinct data structures. CPM: pp. 294-305, (2006).
62. Gonzalez, R., Navarro, G.: A compressed text index in secondary memory. IWOCA, 80-91, (2007).
63. Greve, M., Jorgensen, A.G., Larsen, K.D., Truelsen, J.: Cell probe lower bounds ad approximations for range mode. ICALP, pp. 605-616 (2010).
64. Grossi, R., Gupta, A., Vitter, J.: High-Order Entropy-Compressed Text Indexes. ACM-SIAM SODA, 841-850, (2003).
65. Grossi, R., Orlandi, A., Raman, R.: Optimal trade-offs for succinct string indexes. ICALP (1): pp. 678-689, (2010).
66. Grossi, R., Orlandi, A., Raman, R., and Rao, S.:, More haste less waste: lowering the redundancy in fully indexable dictionaries. Symposium on Theoretical Aspects on Computer Science, 517-528 (2009).

67. Grossi, R., Vitter J.S..: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing, vol. 35, no. 32. pp. 378-407 (2005).
68. Grossi, R., Vitter J., Xu, B.. Wavelet trees from theory to practice. In Proc. of International Conference on Data Compression, Communication and Processing (2011)
69. Gupta, A., Hon, W., Shah, R., Vitter, J.: Compressed data structures: Dictionaries and data-aware measures. In Proc. 16th DCC, 213-222, (2006).
70. Gupta, A., Hon, W., Shah, R., Vitter, J.: A framework for dynamizing succinct data structures. In Proceedings of the 34th International Colloquium on Automata, Languages and Programming, pp. 521-532, (2007).
71. He, M., Munro, J.: Succinct Representations of Dynamic Strings. SPIRE 2010: 334-346, see also CoRR abs/1005.4652: (2010)
72. He, M., Munro, J., Rao, S.: A Categorization Theorem on suffix arrays with applications to space efficient text indexes. ACM SIAM SODA, pp. 23-32, (2005).
73. Hong, W.K., Sadakane, K., Sung, W.K.: Succinct data structures for searchable partial sums. In Proc. ISAAC, LNCS 2906, pp. 505-516, (2003).
74. Hon, W.K., Shah, R., Thankachan, S., and Vitter, J.S., String Retrieval for Multi-pattern queries. In SPIRE, LNCS 6393, pp. 55-66 (2010).
75. Hon, W.K., Shah, R., Vitter, J.: Ordered Pattern Matching: towards full-text retrieval. TR 2006-8, Purdue University, (2006).
76. Hon, W.K., Shah, R., Vitter, J.: Space-efficient framework for top-k string retrieval problems. In FOCS, pp. 713-722, (2009).
77. Hon, W.K., Shah, R., Vitter, J.: Efficient index for retrieving top-k most frequent documents. SPIRE, pp.182-193, (2009), journal version in J. Discrete Algorithms 8(4): 402-417 (2010).
78. Hon, W.K., Shah, R., Vitter, J.: Compression, Indexing and retrieval for Massive String Data. CPM 2010: pp. 260-274, (2010).
79. Jacobson, G.: Space efficient static trees and graphs. In Proc. of FOCS, pp. 549-554, (1989).
80. Karpinski, M., Nekrich, Y.: Space efficient multidimensional range reporting. CoRR abs/0806.4361: (2008)
81. Karpinski, M., Nekrich, Y.: Top-K Color queries for document retrieval. SODA: 401-411, (2011).
82. Kreft, S., Navarro, G.: Self-indexing based on LZ77. DCC: pp. 239-248 (2010).
83. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Data Compression Conference: 296-305 (1999).
84. Lee, S., Park, K.: Dynamic rank/select structures with applications to run-length encoded texts. Theoretical Computer Science 410, pp. 4402-4413 (2009).
85. Lee, S., Park, K.: Dynamic Compressed Representation of Texts with Rank/Select. Journal of Computing Science and Engineering, Vol. 3, No. 1, March, pp. 15-26 (2009).
86. Makinen, V., Navarro, G.: Dynamic Entropy Compressed Sequences and Full-Text indexes. ACM Transactions on Algorithms, Vol. 4, No. 3, Article 32, Publication date: June (2008).
87. Makinen, V., Navarro, G.: Position Restricted Substring Searching. LATIN 2006, pp. 703-714, (2006).
88. Makinnen, V., Navarro, G.: Implicit Compression Boosting with Applications to Self Indexing. SPIRE, pp. 229-241 (2007).
89. Makinnen, V., Navarro, G.: On self-indexing images – image compression with added value. DCC, 422-431, (2008).

90. Makinnen, V., Navarro, G.: Rank and select revisited and extended. Theoretical Computer Science 387(3): pp. 332-347 (2007)
91. Makinnen, V., Navarro, G., Siren, J., Valimaki, N.: Storage and retrieval of highly repetitive sequence collections. JCB (2009).
92. Munro, J.: Tables. In Proc. Of FSTTCS, pp. 37-42, (1996).
93. Muthukrishnan, S.: Efficient algorithms for document retrieval problems. In SODA 2002, pp. 657-666 (2002).
94. Navarro, G., Makinen, V.: Compressed full text indexes. ACM Computing Surveys, 39(1): (2007).
95. Navarro, G., Puglisi, S.: Dual sorted inverted lists. In Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 6393, pp. 310-322, (2010).
96. Navarro, G., Puglisi, S.J., Valenzuela, D.: Practical Compressed Document Retrieval. SEA: 193-205, (2011).
97. Navarro, G., Sadakane, K.: Fully functional static and dynamic succinct trees. CoRR abs/0905.0768: (2009), see also the conference version entitled Fully functional succinct trees, in SODA, 2010, pp. 134-149.
98. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank-select dictionaries. In ALENEX (2007).
99. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM J. Computing, 31(2):353-363, (2001).
100. Patrascu, M.: Succincter. IEEE FOCS, pp. 434-443, (2008).
101. Puglisi, S., Smyth W.F., Turpin, A.: Inverted files versus suffix arrays for locating patterns in primary memory. In SPIRE, LNCS 4209, pp. 122-133, (2006).
102. Raman, R., Raman, V., Srinivasa, S.: Succinct dynamic data structures. WADS, 426-437, (2001).
103. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In SODA 232-242, 2002, see also the full version in ACM Transactions on Algorithms 3(4): (2007).
104. Russo, L., Oliveira, A.: A compressed self-index using a Ziv-Lempel Dictionary. Information Retrieval, 5(3):501-513, (2008).
105. Sadakane, K., Grossi, R.: Squeezing succinct data structures into entropy bounds. In Proc. 17th SODA, pp.1230-239, 2006.
106. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional search in a string with wavelet trees. CPM, LNCS 6129, pp. 40-50, (2010).
107. Shannon, C.E., and Weaver W.: A mathematical theory of communication. The Bell System Technical Journal, 27, 379-423, 623-656, (1948).
108. Siren, J., Valimaki, N., Makinnen, V., Navarro, G.: Run-Length compressed indexes are superior for highly repetitive sequence collections. SPIRE, 164-175, (2008).
109. Stoye, J.: Affix trees. Technical report 2000-04, University of Bielefeld (2000).
110. Strothmann, D.: The affix array data structure and its applications to RNA secondary structure analysis. Theoretical Computer Science 389, 278-294, (2007).
111. Valimaki, N., Makinnen, V.: Space efficient algorithms for document retrieval. CPM, pp. 205-215, (2007).
112. Witten, I., Moffat, A., Bell, T.: Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kauffman, 1999.
113. Yu, C.-C., Hon, W.-K., and Wang B.-F.: Efficient data structures for the orthogonal range successor problem. In COCOON, LNCS 5609, pp. 96-105, (2009).

114. Yu, C.C., Hon, W.-K., Wang, B.-F.: Improved data structures for the orthogonal range successor problem. Computational Geometry 44, pp. 148-159, (2011)

Christos Makris is an Assistant Professor in the Department of Computer Engineering and Informatics, School of Engineering, University of Patras, Greece. His research interests include Data Structures, Web Algorithmics, Computational Geometry, Data Bases and Information Retrieval. He has published over 80 papers in various scientific journals and refereed conferences and has over 250 references. Lists of paper topics**:** information retrieval, text algorithms, data structures, compression.

Received: June 06, 2011; Accepted: February 06, 2012.