Compressing and indexing labeled trees, with applications*

Paolo Ferragina† Fabrizio Luccio† Giovanni Manzini‡
S. Muthukrishnan ♮

† Dipartimento di Informatica, University of Pisa, Italy

‡ Dipartimento di Informatica, University of Piemonte Orientale, Italy

♮ Department of Computer Science, Rutgers University, USA

March 30, 2007

Abstract

Consider an ordered, static tree $\mathcal{T}$ where each node has a label from alphabet Σ. Tree $\mathcal{T}$ may be of arbitrary degree and shape. Our goal is designing a compressed storage scheme of $\mathcal{T}$ that supports basic *navigational* operations among the immediate neighbors of a node (i.e. parent, ith child, or any child with some label, . . .) as well as more sophisticated *path*-based search operations over its labeled structure.

We present a novel approach to this problem by designing what we call the XBW-transform of the tree in the spirit of the well-known Burrows-Wheeler transform for strings [7]. The XBW-transform uses path-sorting to linearize the labeled tree $\mathcal{T}$ into *two* coordinated arrays, one capturing the structure and the other the labels. Using the properties of the XBW-transform, we go beyond the information-theoretic lower bound. For the first time, our compressed indexes support navigational operations and path search operations within (near)-optimal time bounds and entropy-bounded space.

Our XBW-transform is simple and likely to spur new results in the theory of tree compression and indexing, as well in interesting application contexts. As an example, we use the XBW-transform to design and implement a compressed index for XML documents whose compression ratio is significantly better than the one achievable by state-of-the-art tools, and its query time performance is order of magnitudes faster.

1 Introduction

Consider a rooted, ordered, static tree data structure $\mathcal{T}$ on t nodes where each node u has a label drawn from an alphabet Σ. The children of node u are ranked, that is, have left-to-right order. Tree $\mathcal{T}$ may be of arbitrary degree and shape, and the alphabet Σ may be arbitrarily large. Our goal is to design a compressed storage scheme for $\mathcal{T}$ that supports basic *navigational* operations between adjacent nodes in $\mathcal{T}$, as well as more sophisticated *subpath-search* operations over the labeled structure of $\mathcal{T}$. To be precise, let u be a node in $\mathcal{T}$ and let c be a symbol of Σ:

*CONTACT AUTHOR: Paolo Ferragina, ferragina@di.unipi.it. This paper merges and extends the results published in the *Procs of the 46th IEEE Focs 2005* [16] and in the *Procs of the 15th WWW 2006* [17]. This work has been partially supported by NSF DMS 0354600 and by the Italian MIUR grants MAIN-STREAM and Italy-Israel FIRB "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

Navigational queries ask for the parent of u, the ith child of u, or the degree of u. The last two operations might possibly be restricted to the children of u with label c, given that $\mathcal{T}$ is ordered and no restriction on the labeling of its nodes is imposed.

Visualization queries retrieve the nodes in the subtree rooted at u. Any possible order (i.e. pre, in, post) should be implemented.

Subpath queries ask for the (number occ of) nodes of $\mathcal{T}$ that descend from a labeled subpath Π, which may be anchored anywhere in the tree (i.e. not necessarily in its root). Here "descend" refers to the direct descendants of Π's occurrences, hence its offsprings.

The elementary solution to the tree indexing problem above is to represent the tree using a mixture of pointers and arrays using a total of $\Theta(t \log t)$ bits.[1] This trivially supports each of the navigational operations in constant time, but it would require the whole visit of the tree to implement the subpath queries. A more sophisticated approach is needed to search efficiently for arbitrary subpaths over $\mathcal{T}$: it consists of using a variant of the suffix-tree data structure properly designed to index all $\mathcal{T}$'s paths [32]. Subpath queries can be supported in $O(|\Pi| \log |\Sigma| + occ)$ time, but the required space is still $\Theta(t \log t)$ bits (with large hidden constants due to the use of suffix trees [33]).

If the space issue is a primary concern, we have to renounce to pointer-based tree representations and resort the notion of *succinct data structures* introduced by Jacobson [28] in a seminal work over seventeen years ago. The key issue addressed by these data structures is to use space close to their information-theoretic lower bound and yet support various operations efficiently on the indexed data. Thus, succinct data structures are distinct from simply compressing the input, and then uncompressing it later at query time. This area of research was initiated by [28] in the special case of *unlabeled* trees, that is, considering the structure of the tree but not its labels. The number of binary (unlabeled) trees on t nodes is $C_t = \binom{2t+1}{t}/(2t+1)$; therefore $\log C_t = 2t - \Theta(\log t)$ is an obvious lower bound to the storage complexity of binary trees. [28] presented a storage scheme in $2t + o(t)$ bits while supporting the navigational operations in $O(1)$ time. This is a significant improvement over the standard pointer-based representation of trees, without compromising the performance for navigation operations; it is also asymptotically optimal (up to lower-order terms) in storage space. Nearly ten years later, Munro and Raman [42] extended the results with more efficient and also different operations, including subtree size queries. Since then, a slew of results have further generalized these methods to unlabeled trees with higher degrees [6] and ever richer sets of operations such as level-ancestor queries [23]. Succinct representations have been invented for other data structures too, including ordered sets (e.g. [46]), strings [45], graphs (e.g. [43]), functions [44], permutations [41], and others.

Despite this flurry of activity, the fundamental problem of indexing *labeled* trees succintly has remained mostly unsolved.[2] In fact, the labeled tree case is significantly better motivated than the unlabeled case because many applications of Computer Science generate navigational problems on labeled trees, be they for representing data or computation. A recent application is for `XML`, the *de facto* format for data storage, integration, and exchange over the Internet (see `http://www.w3.org/XML/`). As the relationships between elements in an XML document are defined by nested structures, XML documents are often modeled as trees whose nodes are labeled with strings of arbitrary length drawn from a usually

[1] Throughout this paper we assume that all logarithms are taken to the base 2, whenever not explicitly indicated, and we assume $0 \log 0 = 0$.

[2] We use "indexing succinctly" to mean not only representing or compressing trees, but also supporting navigation and search operations on them.

large alphabet Σ. Therefore, at the core of XML-based applications reside efficient data structures for navigating and searching labeled trees with large Σ.

The information-theoretic lower bound for storing labeled trees is easily seen to be $2t - \Theta(\log t) + t \log |\Sigma|$ bits, where the first two terms follow from the structure of the tree and the last term from the storage of the node-labels. A trivial solution to the compressed indexing of labeled trees would be to replicate $|\Sigma|$ times the known structures for the unlabeled case. This could be somewhat improved by a suitable divide-and-conquer approach [23] to achieve $2t + t \log |\Sigma| + O(t|\Sigma| \frac{\log \log \log t}{\log \log t})$ bits of storage and support navigational operations in constant time. However, this is far from the optimal even for moderately large Σ, since the big-O term dominates the others for $|\Sigma| = \Omega(\log \log t)$. This is discouraging since applications such as XML processing or execution traces routinely generate labeled trees over large alphabets. Equally discouraging is the state of "techniques" we know for indexing trees to succinctness. Jacobson [28] reduced the problem of succinct indexing of unlabeled trees to that of ranking and selection problems on arrays as well as parenthesis matching on a sequence of balanced parentheses. These techniques have since been extended with other algorithmic ideas such as partitioning the tree into subtrees. Still, the techniques we have so far, work on the tree structure and cannot embed the label information in a way that is suitable for efficient navigation or compression.

1.1 Our results

We present a new approach to indexing labeled trees without pointers that supports all the operations stated above in (near-)optimal time and achieves succinctness not only in information-theoretic sense, but also at a deeper level of the *entropy* of the input. This will lead us to compressed indexes for labeled trees. Our results are based upon a new XBW-*transform* of the labeled tree $\mathcal{T}$, denoted by $\mathsf{xbw}[\mathcal{T}]$, which linearizes the tree into *two* coordinated arrays $\langle \mathcal{S}_{\mathsf{last}}, \mathcal{S}_\alpha \rangle$, one capturing the structure and the other the labels of $\mathcal{T}$. These two arrays are compressible and efficiently searchable, and thus lets us transform compression and indexing problems on trees into well-understood problems on strings. $\mathsf{xbw}[\mathcal{T}]$ has the optimal size of $2t + t \log |\Sigma|$ bits (Theorem 1) and can be built and inverted in optimal linear time (Theorems 2 and 3). In designing the XBW-transform we were inspired by the elegant Burrows-Wheeler transform (BWT) for strings [7]. In the past few years, the BWT has been the unifying tool for string compression and indexing, producing many important breakthroughs [45]. In the spirit of BWT, which relies on *suffix* sorting for grouping together similar symbols of the input string, our XBW-transform relies on *path* sorting to linearize and group the labels of $\mathcal{T}$ within the two coordinated $\mathsf{xbw}[\mathcal{T}]$'s arrays.

Using XBW, we present a compression scheme for labeled trees that turns the sophisticated labeled-tree compression problem into an easier string compression problem. The performance of this scheme will be evaluated by means of the kth order entropy of the two strings constituting $\mathsf{xbw}[\mathcal{T}]$ (Theorem 4). The resulting algorithm is off from the obvious lower bound of $2t - \Theta(\log t)$ bits (which does not even take into account the node labels), by a factor $(H_k(\mathcal{S}_\alpha) + 2)/2$, where $H_k(s)$ is the kth order empirical entropy of string s (for a formal definition see Eqn. 1 of Section 3). This bound is never worse than the $\log |\Sigma|$ factor obtained by encoding the tree $\mathcal{T}$ trivially but it can be significantly better depending on the distribution of the node labels in $\mathcal{T}$. The experiments in Section 5.2 will show the effectiveness of such a simple tree-compression approach. In Section 3 we will also comment on more sophisticated instantiations of this scheme that have recently lead to more effective compression results.

Using XBW, we also present a compressed-indexing scheme for labeled trees that turns the tree indexing problem into the design of rank and select primitives over strings drawn from an arbitrary alphabet Σ. We recall that: Given a string $S[1, t]$ over Σ, $\mathsf{rank}_c(S, q)$ is the number of times the symbol c occurs in the prefix $S[1, q]$, and $\mathsf{select}_c(S, q)$ is the position of the q-th occurrence of the symbol c

in S. Literature has many efficient implementations for rank and select (see e.g. [45, 5] and references therein). XBW lets us use such primitives as a *black-box* for implementing navigational and search operations over the labeled tree $\mathcal{T}$, going beyond succinctness to entropy-bounded data structures. Theorems 6, 7, and 8 report our main results which can be summarized as follows. For any alphabet Σ, such that $|\Sigma| = O(\text{polylog}(t))$, there exists a succinct indexing of xbw$[\mathcal{T}]$ that takes at most $tH_0(\mathcal{S}_\alpha) + 2t + o(t)$ bits and supports all navigational and subpath search operations in optimal time. We can turn these results to hold for kth order entropy and larger alphabets, but every operation gets slowed down by a $o(\log \log^3 |\Sigma|)$ time factor. Note that no prior algorithmic result is known for supporting subpath queries on trees represented succinctly. We also note that this approach to the indexing of xbw$[\mathcal{T}]$ can be regarded to as a pointerless representation of $\mathcal{T}$ with additional search functions because it takes at most $(\log |\Sigma| + 2 + o(1))$ bits per node, as $H_0(\mathcal{S}_\alpha) \leq |\Sigma|$ (see section 3). It is interesting to note that a tighter analysis on the compression performance of the proposed indexing scheme shows that it is off of the tree-compression space bound by just an additional $o(t \log |\Sigma|)$ term which is usually negligible in practice, as will be shown in the experiments of Section 5.4. Any time/space improvement in the design of rank and select primitives for strings immediately lead to advancements in the compressed indexing problem for labeled trees.

We mentioned earlier the approach based on the suffix tree of a tree [32]. There have been a number of recent results on succinctly representing the suffix tree of a string [45], but the structural properties of the string have been crucially used in building, inverting and searching that suffix tree. For managing the suffix tree of a tree succinctly, new approaches and algorithms are needed. Our XBW-transform may be thought of as the compressed representation for the suffix tree of the tree.

The results of our paper are theoretical in flavor, nonetheless, these results are of immediate relevance to practical XML processing systems. In Section 5 we discuss some encouraging, preliminary experimental results which were initially published in [17] and are summarized here to highlight the impact of the XBW-transform on real datasets. We show that a proper adaptation of the XBW-transform allows to compress XML data, provide access to its content, navigate up and down the XML tree structure, and search for simple path expressions and substrings, while keeping all data in their compressed form and uncompressing only a tiny fraction of them at each operation. Our experimental analysis will concentrate on two main tools: an XML compressor, called XBZIP, and an XML compressed index, called XBZIPINDEX. The former tool is simple in that it relies on standard string-compression methods to squeeze the two arrays of xbw$[\mathcal{T}]$. Our experiments will show that XBZIP achieves compression ratios comparable to the state-of-the-art XML-conscious compressors, such as [35, 1, 10], which tend to be significantly more sophisticated in employing a number of heuristics to mine structure from the document in order to compress "similar contexts". The latter tool XBZIPINDEX will combine the XBW-transform with effective string-compressed indexes like the FM-index [19], achieving compression performance up to 35% better and time efficiency order of magnitudes faster than state-of-the-art tools like XGRIND [49] and XPRESS [39].

REMARK. Subsequent to the conference versions [16, 17] of the present paper, various results have been published in main conferences on tree compression and indexing problems. Some papers [4, 25, 5] have proposed improved implementations for the rank and select primitives on strings drawn from arbitrary alphabets, and [48, 26, 22] have showed how to turn these *0*th order entropy bounds into kth order entropy bounds. These results can be used as black-boxes within our tree compression and indexing schemes. This is actually what we do in Section 4 where our theorems are improved versions of their corresponding counterparts in [16, 17] because of this ability to plug in.

Some other papers [4, 5, 29] have recently extended the set of navigational operations that may be

supported on labeled and multi-labeled trees to more sophisticated queries— like ancestors, descendants and node depth— based on the DFUDS representation of ordinal trees by [6]. They achieved sub-logarithmic time for those navigational operations and $\Omega(t)$ bits of storage. Therefore their solutions are succinct but cannot be considered entropy-bounded; furthermore, they do not support the powerful subpath search operation over the tree $\mathcal{T}$ as instead we do with the XBW.

Finally, some results have proposed novel uses of our XBW-transform to compress LZ-tries [3] or to *dynamize* the labeled-tree compressed indexing problem [27]. These results again show that the XBW is a very general tool which may spur new results in the theory of tree compression and indexing, as well in other interesting application contexts, not just XML data.

1.2 Structure of the paper

Section 2 introduces the XBW-transform and describes its structural and optimality properties, and the optimal algorithms to convert $\mathcal{T}$ into $\mathsf{xbw}[\mathcal{T}]$ and vice versa. Section 3 describes the tree-compression scheme based on the XBW-transform and comments on some of its instantiations. Section 4 presents the (near-)optimal succinct data structures for indexing $\mathsf{xbw}[\mathcal{T}]$ and thus supporting in (near-)optimal time navigational and subpath search operations over the labeled tree $\mathcal{T}$. Finally, Section 5 presents the compression and indexing tools for XML data, namely XBZIP and XBZIPINDEX, and discuss the experimental results.

2 The XBW-transform for labeled trees

Let $\mathcal{T}$ be an ordered tree of arbitrary fan-out, depth and shape. $\mathcal{T}$ consists of n internal nodes and ℓ leaves, for a total of $t = n + \ell$ nodes. Every node of $\mathcal{T}$ is labeled with a symbol drawn from an alphabet Σ. We assume that Σ is the set of labels effectively used in $\mathcal{T}$'s nodes and that these labels are encoded with the integers in the range $[1, |\Sigma|]$. This assumption needs a preliminary sorting step, but makes the approach general enough to deal uniformly with labels which are *long strings*, as it occurs in various applications (see Section 5).
For each node u, we compute the following information:

- $\mathsf{last}[u]$ is a binary flag set to 1 if u is the rightmost (last) child of its parent;
- $\alpha[u]$ denotes the label of u *plus* one bit that indicates whether node u is internal or a leaf;
- $\pi[u]$ is the string obtained by concatenating the labels on the *upward path* from u's parent to the root of $\mathcal{T}$ (the root as an empty π component).

We point out that the information plugged into $\alpha[u]$ is needed to distinguish between internal nodes and leaves. There are cases in which the additional bit is not needed, for example when Σ consists of two disjoint subsets Σ_N and Σ_L which are used to label internal nodes and leaves, respectively. In this paper, we follow the common practice (see e.g. [23]) that assumes to have one unique alphabet Σ for labeling all of $\mathcal{T}$'s nodes. Nonetheless, for the sake of presentation, we will adopt the notation $\alpha[u] \in \Sigma_N$ or $\alpha[u] \in \Sigma_L$ to indicate whether u is an internal node or a leaf of $\mathcal{T}$, respectively. See Figure 1.a for an illustrative example in which we have used uppercase and lowercase letters to denote Σ_N and Σ_L, respectively.

To define the XBW-transform we build a *sorted multi-set* $\mathcal{S}$ consisting of t *triplets*, one for each node of $\mathcal{T}$. We build $\mathcal{S}$ in two steps: (1) Visit $\mathcal{T}$ in pre-order and, for each visited node u, insert the triplet $\langle \mathsf{last}[u], \alpha[u], \pi[u] \rangle$ in $\mathcal{S}$; (2) Stably sort $\mathcal{S}$ according to the π-component of its triplets. Since

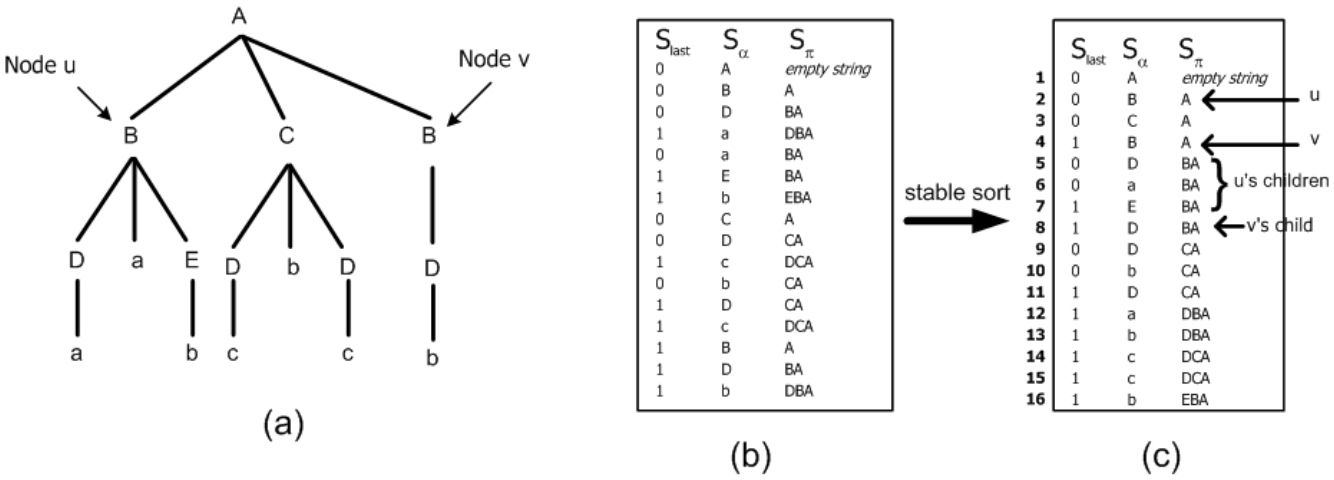

Figure 1: (a) A labeled tree $\mathcal{T}$ where $\Sigma_N = \{A, B, C, D, E\}$ and $\Sigma_L = \{a, b, c\}$. Notice that $\alpha[u] = \alpha[v] = \texttt{B}$ and $\pi[u] = \pi[v] = \texttt{A}$. (b) The multi-set $\mathcal{S}$ obtained after the pre-order visit of $\mathcal{T}$. (c) The final multi-set $\mathcal{S}$ after the stable sort based on the π's component of its triplets.

sibling nodes of $\mathcal{T}$ may be labeled with the same symbol, many nodes may have the same π-string. Consequently, $\mathcal{S}$ is a multi-set, and we need the stable sort to preserve the identity of the triplets. Note that the triplet of the root goes to the first position. Hereafter we will use $\mathcal{S}_{\mathsf{last}}[i]$ (resp. $\mathcal{S}_\alpha[i]$, $\mathcal{S}_\pi[i]$) to refer to the last (resp. α, π) component of the i-th triplet of $\mathcal{S}$.

Theorem 1 *The* XBW*-transform of a labeled tree $\mathcal{T}$ consists of the two arrays $\langle \mathcal{S}_{\mathsf{last}}, \mathcal{S}_\alpha \rangle$ after sorting, and takes $2t + t \lceil \log |\Sigma| \rceil$ bits.*

The space cost is derived by observing that $\mathcal{S}_\alpha$ needs $t(\lceil \log |\Sigma| \rceil + 1)$ bits since we need to encode each symbol of Σ plus the distinguishing (leaf vs. internal node label) bit, and $\mathcal{S}_{\mathsf{last}}$ needs t bits. This space bound is *optimal* in the worst case (up to lower order terms) since $2t - \Theta(\log t)$ bits are needed to represent the t-node ordinal tree structure of $\mathcal{T}$ [28], and $t \lceil \log |\Sigma| \rceil$ bits are needed to represent $\mathcal{T}$'s labels.

We note that any pair $\langle \mathcal{S}_{\mathsf{last}}[i], \mathcal{S}_\alpha[i] \rangle$ corresponds to a node u of the tree $\mathcal{T}$. Consequently, in the rest of the paper we will interchangeably use u or $\mathcal{S}[i]$, depending on the context. We notice that the XBW-transform induces an *implicit numbering* on the tree nodes within the range $[1, t]$. Other numbering schemes do exist in the literature but use a larger numbering range [50], or use a large range which is squeezed on-the-fly by means of constant-time query operations [6], or use the minimal range $[1, t]$ but do not offer the nice compression and indexing properties of the XBW-transform [47].

The following two properties of the ordered multi-set $\mathcal{S}[1, t]$ constitute the basic block upon which we will design our algorithms for computing, inverting, navigating and searching $\mathsf{xbw}[\mathcal{T}]$. They immediately follow from the definition of the transform (Property 1) and from the way $\mathcal{S}$ has been built (Property 2).

Property 1 *"Composition" of $\mathcal{S}$.*

1. *$\mathcal{S}_{\mathsf{last}}$ has n bits set to 1 (one for each internal node), and $\ell = t - n$ bits set to 0.*
2. *$\mathcal{S}_\alpha$ is a permutation of the symbols labeling $\mathcal{T}$'s nodes.*

3. *$\mathcal{S}_\pi$ contains all the upward labeled paths of $\mathcal{T}$ consisting of internal-node labels only. Each path is repeated a number of times equal to the number of its offsprings.*

Property 2 *"Structural relation" between $\mathcal{T}$ and $\mathcal{S}$.*

1. *The first triplet of $\mathcal{S}$ refers to the root of $\mathcal{T}$.*
2. *The triplet of node u precedes the triplet of node v in $\mathcal{S}$ iff either $\pi[u] < \pi[v]$ lexicographically, or $\pi[u] = \pi[v]$ and u precedes v in the pre-order visit of $\mathcal{T}$.*
3. *Let $u_1, \ldots, u_z$ be the children of node u in $\mathcal{T}$. The triplets $s[u_1], \ldots, s[u_z]$ lie contiguously in $\mathcal{S}$ following this order. Moreover, the subarray $\mathcal{S}_{\mathsf{last}}[u_1 \cdots u_z]$ provides the* unary encoding *of u's degree, namely $\mathcal{S}_{\mathsf{last}}[u_z] = 1$ and $\mathcal{S}_{\mathsf{last}}[u_i] = 0$ for all $1 \leq i < z$.*
4. *Let u, v be two nodes of $\mathcal{T}$ having the same label $\alpha[u] = \alpha[v]$. If the triplet of node u precedes the triplet of node v in $\mathcal{S}$, then the (contiguous block of) children of u precede the (contiguous block of) children of v in $\mathcal{S}$.*

As an illustrative example of the above properties, let us look at Figure 1 in which we have two nodes u and v labeled `B`, whose upward path is `A`. The triplet of u occurs at $\mathcal{S}[2]$ whereas the triplet of v occurs at $\mathcal{S}[4]$, thus reflecting the fact that u precedes v in the pre-order visit of $\mathcal{T}$ (Property 2, item 2). Actually, the triplets of the two nodes are not contiguous because of the second child of the root, whose upward path is also `A`. Node u has three children whose triplets occur contiguously at $\mathcal{S}[5, 7]$ with $\mathcal{S}_{\mathsf{last}}[5, 7] = 001$ (Property 2, item 3). Node v follows node u in $\mathcal{S}$, and indeed the single child of v occurs at $\mathcal{S}[8]$ correctly past the triplets of u's children (Property 2, item 4). This holds for any pair of nodes, independently of their upward path: $\mathcal{S}[5]$ and $\mathcal{S}[11]$ are equally labeled but have different upward paths, nonetheless their children $\mathcal{S}[12]$ and $\mathcal{S}[15]$ preserve their relative order.

The following property shows that the tree structure of $\mathcal{T}$ is *implicitly encoded* into $\mathsf{xbw}[\mathcal{T}]$, and can be algorithmically used to simulate a *navigation* of $\mathcal{T}$ given rank/select data structures over the two arrays of $\mathsf{xbw}[\mathcal{T}]$.

Property 3 *Let $c \in \Sigma_N$ be an internal node label, and let $\mathcal{S}[j_1, j_2]$ be all triplets whose π-components are prefixed by symbol c. If u is the ith node labeled c in $\mathcal{S}_\alpha$, its children occur contiguously within $\mathcal{S}[j_1, j_2]$ and delimited by the $(i-1)$st and the ith bit set to `1` in $\mathcal{S}_{\mathsf{last}}[j_1, j_2]$.*

Proof: Given the definition of $\mathcal{S}_\pi$ and the stable sort of $\mathcal{S}$, the range $\mathcal{S}[j_1, j_2]$ identifies all children of nodes labeled c in $\mathcal{T}$. All these children occur in $\mathcal{S}$ subdivided into groups which are delimited by triplets whose last-component is set to 1 (Property 2, item 3). Moreover, these groups preserve the order of their parents (Property 2, item 4). Consequently, if u is the ith node labeled c in $\mathcal{S}_\alpha$, its children will form the ith block of children in $\mathcal{S}[j_1, j_2]$. By Property 2 item 3, u's children are delimited by the $(i-1)$st and the ith `1`s in $\mathcal{S}_{\mathsf{last}}[j_1, j_2]$. ∎

As an illustrative example, let us look in Figure 1 at node v whose label `B` is the second one in $\mathcal{S}_\alpha$. All the children of nodes labeled `B` occur in the range $\mathcal{S}[5, 8]$, they include also the three children of u. According to Property 3, the children of v (just one!) form the *second* group in $\mathcal{S}[5, 8]$ and, hence, are delimited by the first and the second bit set to `1` in $\mathcal{S}_{\mathsf{last}}$ (namely, $\mathsf{last}[7] = \mathsf{last}[8] = 1$). From this we also infer that v has one child and this is stored in $\mathcal{S}[8, 8]$.

Algorithm PathSort($\mathcal{T}$)

1. Create the array IntNodes$[1, t]$ initially empty.
2. Visit the internal nodes of $\mathcal{T}$ in preorder. Let u denote the ith visited node. Write in IntNodes$[i]$ the symbol $\alpha[u]$, the level of u in $\mathcal{T}$, and the position in IntNodes of u's parent.
3. Let $j \in \{0, 1, 2\}$ be such that the number of nodes in IntNodes whose level is $\equiv j \pmod 3$ is at least $t/3$. Sort recursively the upwards subpaths starting at nodes in levels $\not\equiv j \pmod 3$.
4. Sort the upward sub-paths starting at nodes in levels $\equiv j \bmod 3$ using the result of Step 3.
5. Merge the two sets of sorted subpaths by exploiting their lexicographic names.

Figure 2: Optimal algorithm for sorting the π-components of the tree $\mathcal{T}$.

2.1 From $\mathcal{T}$ to xbw$[\mathcal{T}]$: constructing the transform

The explicit construction of $\mathcal{S}$ through the concretization of π-strings requires too much space and time. As a limit case, if $\mathcal{T}$ consists of a single path of t nodes, the overall size of $\mathcal{S}_\pi$ is $\Theta(t^2)$. Therefore the construction of $\mathcal{S}$ must be *implicit*. Such a construction will be done in linear time. However, for the sake of presentation, we start by proposing an algorithm which runs in $O(t \log t)$ time and uses $O(t \log t)$ bits of space. We will then refine this idea to achieve the optimal $O(t)$ time complexity, based on a simple generalization of the *skew algorithm* for suffix array construction [30], here extended from strings to labeled trees.

An $O(t \log t)$-time algorithm for constructing xbw$[\mathcal{T}]$. The algorithm is a reminiscence of the so called *tree contraction technique* used in the PRAM model to solve several tree-based problems in logarithmic parallel-time. It operates in $O(\log t)$ phases, and its ultimate goal is to assign to each upward path in $\mathcal{T}$ an integer (hereafter called *name*) that denotes its *lexicographic rank* among all upward paths of $\mathcal{T}$. Given these path-names, the sorting of all $\mathcal{S}$'s triplets would boil down to a linear-time radix sort. In phase i, the algorithm operates on a labeled tree $\mathcal{T}_i$ which is a *contracted version* of $\mathcal{T}$: the parent $p_i(u)$ of a node u in $\mathcal{T}_i$ is the 2^i-th ancestor of u in the original tree $\mathcal{T}$. At the beginning we set $\mathcal{T}_0 = \mathcal{T}$; in the $(i+1)$th phase, we derive the structure of $\mathcal{T}_{i+1}$ from the one of $\mathcal{T}_i$ by just *contracting two* parent pointers from any node. This means that, at any recursive step, the height of $\mathcal{T}_i$ is shrinking by a factor two and the fan-out of $\mathcal{T}_i$ is possibly increasing. The latter issue will not be a problem because we will always access the tree via parent pointers in constant time.

In order to assign names to upward paths of $\mathcal{T}$, the algorithm proceeds via better and better approximations during its recursive steps. At the beginning we have $\mathcal{T}_0 = \mathcal{T}$, and the algorithm assigns to every node u a label name$_0[u] = \alpha[u]$. At a generic phase $i > 0$, the label name$_i[u]$ becomes the *lexicographic rank* of the labeled (upward) subpath in $\mathcal{T}$ that starts from u and leads to its 2^ith ancestor in $\mathcal{T}$. Let us denote this subpath with $\pi_{2^i}[u]$. Notice that $\pi_{2^i}[u]$ is actually the prefix of length 2^i of $\pi[u]$, and this prefix is actually the one denoted by the (contracted) edge that connects u to its parent $p_i(u)$ in $\mathcal{T}_i$. It is easy to note that $\Theta(\log t)$ phases are sufficient to assign lexicographic names to all (full) upward paths of $\mathcal{T}$. For deriving name$_{i+1}[u]$ from name$_i[u]$, radix-sort all pairs $\langle$name$_i[u]$, name$_i[p_i(u)]\rangle$. Given that the names are integers in the range $[1, t]$, every recursive step takes $O(t)$ time, and thus the algorithm takes overall $O(t \log t)$ time.

An optimal $O(t)$-time algorithm for constructing xbw$[\mathcal{T}]$. In order to achieve time optimality, we need to avoid some duplicate work that naturally arises in the algorithm above. The tree-contraction technique leads to considers all $\Theta(t)$ upward paths at all recursive phases, independently of the existence

of shared subpaths. Our optimal algorithm, summarized in Figure 2, operates recursively over a tree which *shrinks* by height *and* by number of nodes (and hence by processed upward paths). The algorithm is based on a simple generalization of the *skew algorithm* for suffix array construction of strings [30], here extended to deal with tree-based data. The only non-trivial step of this generalization is the recursion (Step 3) in which we restrict the radix-sorting to only the (upward) subpaths that start at nodes in levels $\not\equiv j \pmod 3$ (above we sorted all possible subpaths). The parameter j is chosen in such a way that the number of nodes being at level $\equiv j \pmod 3$ is at least $t/3$; hence a constant fraction of upward paths are ensured to be dropped from the subsequent recursive steps.[3] Note that: (1) the height of the new (contracted) tree shrinks by a factor three (instead of two), hence the node naming requires the radix sort over triples of names rather than on pairs of names; (2) given the choice of j, the number of nodes of the new (contracted) tree will be at most $2t/3$, thus ensuring that the running time of the algorithm satisfies the recurrence $T(t) = T(2t/3) + \Theta(t) = \Theta(t)$; (3) following an argument similar to [30], the names of the dropped subpaths can be computed in $O(t)$ time from the names of the non-dropped subpaths, by radix sorting. In fact, it suffices to note that any subpath starting at level $\equiv j \pmod 3$, can be expressed as the concatenation of a node and a subpath starting at level $\not\equiv j \pmod 3$ (whose name is recursively known). The merging of the two sets of subpaths, to achieve a unique naming assignment, can be done as in [30]. Noting that each element of the array IntNodes takes $O(\log t)$ bits we have:

Theorem 2 *Let $\mathcal{T}$ be a labeled tree with t nodes and labels drawn from an alphabet Σ. The transform $\mathsf{xbw}[\mathcal{T}]$ can be computed in $O(t)$ time and $O(t \log t)$ bits of working space.*

Recall that we are assuming all symbols in Σ be used to label $\mathcal{T}$'s nodes, and that they are packed into the range $[1, t]$. Otherwise, we should add the sorting cost of naming the n internal node labels of $\mathcal{T}$ by consecutive integers.

2.2 From $\mathsf{xbw}[\mathcal{T}]$ to $\mathcal{T}$: inverting the transform

Property 3 ensures that the two arrays $\langle \mathcal{S}_{\mathsf{last}}, \mathcal{S}_\alpha \rangle$ forming $\mathsf{xbw}[\mathcal{T}]$ contain enough information for deriving parent-child relations between $\mathcal{T}$'s nodes. The key issue is therefore to show that the reconstruction of $\mathcal{T}$ may be done in $O(t)$ optimal linear time, and thus that each parent-child relationship can be inferred in constant-amortized time. The pseudocode of RebuildTree is given in Figure 3. It deploys two subroutines BuildF and BuildJ which are detailed in Figures 4 and 5.

The algorithm RebuildTree works in three phases, takes $O(t)$ optimal time, and reconstructs the tree $\mathcal{T}$ in depth-first order. In the first phase (Step 1) it builds the array F that approximates $\mathcal{S}_\pi$ at its first symbol: $\mathsf{F}[x]$ stores the position in $\mathcal{S}$ of the first triplet whose π-component is prefixed by x. For the example in Figure 1 we have $\mathsf{F}[\mathtt{B}] = 5$, since $\mathcal{S}[5]$ is the first triplet having its π-component prefixed by the internal-node label B. In the second phase (Step 2), the algorithm exploits F to efficiently build the array J which encodes the first-child pointer of each node in $\mathcal{T}$: $\mathsf{J}[i] = j$ if $\mathcal{S}[j]$ is the first child of $\mathcal{S}[i]$, and $\mathsf{J}[i] = -1$ if $\mathcal{S}[i]$ is a leaf. For the example in Figure 1 we have $\mathsf{J}[2] = 5$, since the node u at $\mathcal{S}[2]$ has its first child stored at $\mathcal{S}[5]$. In the third final phase, the algorithm deploys the array J to simulate a depth-first visit of $\mathcal{T}$, creates its labeled nodes, and properly connects them to their parents. In what follows we concentrate on proving the correctness of algorithm RebuildTree, since its linear time complexity can be easily derived from its computational structure.

[3] In the original Skew Algorithm [30], the recursion consists of sorting all suffixes starting at positions $\not\equiv 1 \pmod 3$. The algorithm works equally well if instead of 1 we use either 0 or 2 because any choice ensures a constant shrinking of the contracted string which is passed to the recursive call. Of course, this may be not the case for a tree $\mathcal{T}$ of arbitrary fan-out, depth and shape, as the one we are dealing with.

Algorithm RebuildTree(xbw[$\mathcal{T}$])

1. $\mathsf{F} = \mathsf{BuildF}(\mathsf{xbw}[\mathcal{T}])$; $\{\mathsf{F}[x] =$ *first entry in* $\mathcal{S}$ *whose* π*-component is prefixed by symbol* $x\}$
2. $\mathsf{J} = \mathsf{BuildJ}(\mathsf{xbw}[\mathcal{T}], \mathsf{F})$; $\{\mathsf{J}[i] =$ *position in* $\mathcal{S}$ *of the first-child of* $\mathcal{S}[i]$*;* $\mathsf{J}[i] = -1$ *if leaf.*$\}$
3. Create node r and set $\mathcal{Q} = \{\langle 1, r\rangle\}$;
4. **while** $\mathcal{Q} \neq \emptyset$ **do** $\{$*We still have nodes to create in* $\mathcal{T}\}$
5. $\langle i, u\rangle = \mathtt{pop}(\mathcal{Q})$;
6. $j = \mathsf{J}[i]$; $\{$*Take the block of* u*'s children in* $\mathcal{S}\}$
7. **if** $(j = -1)$ **then continue**; $\{u$ *is a leaf of* $\mathcal{T}\}$
8. Find first $j' \geq j$ such that $\mathcal{S}_{\mathsf{last}}[j'] = 1$; $\{\mathcal{S}[j, j']$ *are the children of* u *in* $\mathcal{T}\}$
9. **for** $h = j'$ **downto** j **do** $\{$ *Recall that* $\mathcal{Q}$ *is a stack*$\}$
10. Create the node v labeled $\mathcal{S}_\alpha[h]$;
11. Attach v as first child of u;
12. **push**$(\langle h, v\rangle, \mathcal{Q})$;
13. **return** node r.

Figure 3: Reconstruct $\mathcal{T}$ from $\mathsf{xbw}[\mathcal{T}]$ in depth-first order.

Algorithm BuildF(xbw[$\mathcal{T}$])

1. **for** $i = 1, \ldots, t$ **do** $\mathsf{C}[\mathcal{S}_\alpha[i]]$ ++; $\{$ *Count occurrences of node labels* $\}$
2. $\mathsf{F}[1] = 2$; $\{$ $\mathcal{S}_\pi[1]$ *is the empty string* $\}$
3. **for** $i = 1, \ldots, |\Sigma_N| - 1$ **do** $\{$ *consider just the internal-node labels* $\}$
4. $s = 0$; $j = \mathsf{F}[i]$;
5. **while** $(s \neq \mathsf{C}[i])$ **do** $\{$ *not all blocks of children have been passed* $\}$
6. **if** $(\mathcal{S}_{\mathsf{last}}[j++] = 1)$ **then** $s++$; $\{$ *one further block of children has passed*$\}$
7. $\mathsf{F}[i+1] = j$;
8. **return** F.

Figure 4: Compute array F such that $\mathsf{F}[i] = j$ iff $\mathcal{S}_\pi[j]$ is the first entry of S prefixed by i.

Algorithm BuildJ(xbw[$\mathcal{T}$], F)

1. **for** $i = 1, \ldots, t$ **do**
2. **if** ($\mathcal{S}_\alpha[i] \in \Sigma_L$) **then** J$[i] = -1$; { $\mathcal{S}_\alpha[i]$ *is a leaf label*}
3. **else**
4. $z =$ J$[i] =$ F$[\mathcal{S}_\alpha[i]]$; { $\mathcal{S}_\alpha[i]$ *is an internal-node label* }
5. **while** ($\mathcal{S}_{\mathsf{last}}[z] \neq 1$) **do** $z++$; { *reach the last child of* $\mathcal{S}_\alpha[i]$ }
6. F$(\mathcal{S}_\alpha[i]) = z + 1$;
7. **return** J

Figure 5: Compute array J such that J$[i] = j$ if $\mathcal{S}[j]$ is the first child of $\mathcal{S}[i]$, and J$[i] = -1$ if $\mathcal{S}[i]$ is a leaf.

We start noting that the computation of array F may be limited to symbols in Σ_N since π-components are built upon internal-node labels only (Property 1, item 3). Algorithm BuildF in Figure 4 first stores in C$[y]$ the number of occurrences of each symbol y in $\mathcal{T}$, and then computes F inductively. The base case is obvious: Step 2 sets F$[1] = 2$ since $\mathcal{S}_\pi[1]$ is the empty string (Property 2, item 1), and we have assumed that all symbols of Σ are present in $\mathcal{T}$. For the ith inductive step, we know that all π-components prefixed by i correspond to children of nodes labeled i, and these children occur contiguously in $\mathcal{S}$ starting at position F$[i]$ by Property 3. Moreover, since the nodes labeled i are $C[i]$ in number, we have C$[i]$ contiguous blocks of children which can be identified by looking at entries set to 1 in $\mathcal{S}_{\mathsf{last}}$. The loop in Steps 5–6 serves for this purpose, and therefore the value F$[i + 1]$ is correctly set at Step 7.

Algorithm BuildJ in Figure 5 computes the array J in $O(t)$ optimal time via a single scan of $\mathcal{S}_\alpha$, by reusing the space allocated for array F with the following invariant: for every symbol c, F$[c]$ points to the block of children of the next occurrence in $\mathcal{S}_\alpha$ of a node labeled c (step 6). At the beginning F is correctly set, due to its definition. For the inductive step we exploit Property 3, that is, if $c = \mathcal{S}_\alpha[i]$ is the kth occurrence of symbol c in $\mathcal{S}_\alpha$, then the children of $\mathcal{S}[i]$ correspond to the k-th block of children counting from position F$[c]$. Given that we are scanning $\mathcal{S}_\alpha$, F$[c]$ advances on these blocks of children as occurrences of c are met over $\mathcal{S}_\alpha$ (Steps 5 and 6). Similarly we set the entries of the array J (Step 4). Noting that each element of this array takes $O(\log t)$ bits we have:

Theorem 3 *A t-node labeled tree $\mathcal{T}$ can be reconstructed from its transform* xbw$[\mathcal{T}]$ *in optimal $O(t)$ time and $O(t \log t)$ bits of working space.*

3 Compressing labeled trees

The locality principle exploited in string compressors states that each element of a string depends strongly on its nearest neighbors, namely, predecessor and/or successor symbols. The *context* of a symbol c is defined on strings as the *substring that precedes* c. A $k-context$ is a context of length k. The larger is k, the better should be the prediction of c given its k-context. Given this, the compressibility of a string s is usually measured in terms of its kth order *empirical entropy*, where $k \geq 0$ [37]. The *0*th order empirical entropy of a string s is defined as: $H_0(s) = -(1/|s|) \sum_{c \in \Sigma} (s_c \log(s_c/|s|))$, where s_c is the number of occurrences of symbol c in s. Given H_0, the k-th order empirical entropy of string s is defined as:

$$H_k(s) = (1/|s|) \sum_{\rho \in \Sigma^k} |s_\rho| \, H_0(s_\rho), \tag{1}$$

Algorithm TreeCompress($\mathcal{T}, \mathcal{C}_{\mathsf{last}}, \mathcal{C}_\alpha$)

1. Compute the XBW-transform of $\mathcal{T}$, namely $\mathsf{xbw}[\mathcal{T}] = \langle \mathcal{S}_{\mathsf{last}}, \mathcal{S}_\alpha \rangle$.
2. Use compressor $\mathcal{C}_{\mathsf{last}}$ to squeeze $\mathcal{S}_{\mathsf{last}}$.
3. Use compressor $\mathcal{C}_\alpha$ to squeeze $\mathcal{S}_\alpha$.

Figure 6: Algorithm to compress the labeled tree $\mathcal{T}$.

where s_ρ is the string formed by all symbols x such that x immediately follows an occurrence of substring ρ in s. As an example, if $s = \texttt{ababcabdaa}$ and $\rho = \texttt{ab}$, then $s_\rho = \texttt{acd}$.

Our starting point for compressing $\mathsf{xbw}[\mathcal{T}]$ is a generalization of the notion of k-context to labeled-tree data. The theory of Markov random fields [53] extends the k-context notion for strings to more general mathematical structures, including trees, by defining the symbol's nearest neighbors as its ancestors, its children, or any set of *nearest* nodes. In this paper, we naturally define the *k-context* of a node u in $\mathcal{T}$ as the k-long prefix of $\pi[u]$, and denote it by $\pi_k[u]$. Therefore $\pi_k[u]$ is the k-long subpath leading to u in $\mathcal{T}$, or equivalently u descends from a subpath labeled as $\pi_k[u]$ (note that the nodes in $\pi_k[u]$ are met upwards). As for string data, we postulate that *similarly labeled nodes descend from similar k-contexts*, and that the longer is k the better should be the prediction provided by $\pi_k[u]$ for the symbol labeling node u (i.e. $\alpha[u]$). Section 5 supports this statement with a practical example drawn from XML data.

The $\mathsf{xbw}[\mathcal{T}]$ shows a *local homogeneity* property on the string $\mathcal{S}_\alpha$ that can be proved via the notion of k-contexts on trees. This property mimics, on labeled trees, the same strong property obtained for strings by the Burrows-Wheeler Transform [7]. Specifically, *node labels get distributed over $\mathcal{S}_\alpha$ according to a pattern that clusters closely the labels which descend from "similar" upward paths sharing long prefixes.* To see this, let us pick any two nodes u and v, and consider their contexts $\pi[u]$ and $\pi[v]$. Given the sorting of $\mathcal{S}$, the longer is the shared prefix between $\pi[u]$ and $\pi[v]$, the closer are the labels $\alpha[u]$ and $\alpha[v]$ in $\mathcal{S}_\alpha$. These close labels are thus expected to be *few distinct ones*, and thus $\mathcal{S}_\alpha$ is expected to be locally homogeneous. Hence we may exploit all the algorithmic machinery recently developed for BW-based compressors to achieve high compression (see e.g. [15]). As far as the compressibility of $\mathcal{S}_{\mathsf{last}}$ is concerned, we note that it depends on the sorting of $\mathcal{S}$ and thus, we might exploit some proper compressors for it too.

Our compression scheme for trees, as indicated in Figure 6, deploys the XBW-transform for turning the sophisticated labeled-tree compression problem into an easier string compression problem. To this aim, it uses two string compressors $\mathcal{C}_\alpha$ and $\mathcal{C}_{\mathsf{last}}$ to squeeze the two strings that compose $\mathsf{xbw}[\mathcal{T}]$, by exploiting their fine specialties. Of course, many choices are possible for $\mathcal{C}_{\mathsf{last}}$ and $\mathcal{C}_\alpha$, each having implications on the algorithm time and compression bounds. The approach is left purposedly general. In the following Theorem 4 we merely suggest a possible implementation, which will be investigated experimentally in Section 5.2, and then comment on some more sophisticated tree-compressors like the ones proposed in [16] and [29]. We have:

Theorem 4 *Let $\mathcal{C}_\alpha$ be a kth order string compressor that compresses any string w into $|w|H_k(w) + |w| + o(|w|)$ bits, in $O(|w|)$ time (e.g., see [15]); and let $\mathcal{C}_{\mathsf{last}}$ be an algorithm that stores $\mathcal{S}_{\mathsf{last}}$ without compression. With this instantiation, algorithm* TreeCompress *compresses the labeled tree $\mathcal{T}$ within $tH_k(\mathcal{S}_\alpha) + 2t + o(t)$ bits and takes $O(t)$ optimal time.*

This simple solution is off from the lower bound of $2t - \Theta(\log t)$ bits (which does not even take into account the node labels), by a factor $(H_k(\mathcal{S}_\alpha) + 2)/2$. Since $H_k(\mathcal{S}_\alpha) \leq \log |\Sigma|$, the above bound is never worse than the $\log |\Sigma|$ factor obtained by encoding trivially the tree $\mathcal{T}$. However it can be significantly

better depending on the distribution of the labels among the tree nodes. Indeed the experiments in Section 5.2 will support this statement.

We can obviously aim for more by using two distinct compressors specialized on the features of strings $\mathcal{S}_{\mathsf{last}}$ and $\mathcal{S}_\alpha$. As an example, note that $\mathcal{S}_\alpha$ may be partitioned into substrings $s_1, s_2, \ldots, s_r$ such that, s_i is formed by all symbols that descend from a k-long subpath labeled ρ_i in $\mathcal{T}$. The local homogeneity property of $\mathcal{S}_\alpha$ (see above) implies that each s_i is highly compressible, and the sorting of $\mathcal{S}$ implies that $\mathcal{S}_\alpha$'s partitioning can be efficiently identified via a simple modification of algorithm PathSort. The compression of $\mathcal{S}_\alpha$ therefore may proceed in two steps [16]:

1. Partition $\mathcal{S}_\alpha$ into substrings $s_1, s_2, \ldots, s_r$ according to the r distinct subpaths of length k occurring in $\mathcal{T}$.
2. Compress individually each of these substrings s_i via any string-based compressor.

The compression performance of this algorithm has been evaluated in [16] by generalizing the notion of kth order empirical entropy from strings to trees, and by adopting the boosting technique in [15] for computing a partition of $\mathcal{S}_\alpha$ which is optimal under some specific space-saving criteria. As far as the compression of $\mathcal{S}_{\mathsf{last}}$ is concerned, we may deploy the fact that this string is actually the concatenation of unary encodings of node degrees, and thus we can use a 0th order compressor for them [29]. This could achieve high space saving in the case of very regular trees (see [29] for details). We prefer not to detail this approach which would require introducing and discussing new entropy notions and many other technicalities. Also, we think that these novel entropy notions for trees, and the corresponding sophisticated tree-compression algorithms, need further experimental evaluations to prove that they yield significant improvements over the ones we present in the following sections. Recent experiments on the boosting technique for strings [13], which underlies the tree-compressor of [16], have indeed shown that such a technique is slow, and comparable results can be achieved by simpler approaches. We therefore refer the interested reader to the seminal papers [16, 29] for further details and research issues.

Clearly TreeCompress may benefit from any advancement in string compression or (unlabeled) tree compression, as it actually occurred with the results recently published in [14, 13, 4, 5, 29]. Moreover, we point out that $\mathcal{S}_{\mathsf{last}}$ is negligible in size with respect to $\mathcal{S}_\alpha$ for most practical applications (see Section 5), so that any advancement in $\mathcal{T}$'s compression may mainly come from $\mathcal{S}_\alpha$'s squeezing. This is what we will mainly address in Section 5.2, where we will instantiate the pseudocode of Figure 6 with PPM, a very effective kth order compressors.

4 Indexing compressed labeled trees

Property 3 ensures that the two arrays $\langle \mathcal{S}_{\mathsf{last}}, \mathcal{S}_\alpha \rangle$ of $\mathsf{xbw}[\mathcal{T}]$ provide an equivalent and pointerless representation of the labeled tree $\mathcal{T}$ which implicitly encodes its parent-child information. In this section we make one step further, showing that it is possible to design a compressed representation of $\mathsf{xbw}[\mathcal{T}]$ that efficiently (in fact, optimally) supports both navigational and sophisticated path-search operations over the labeled tree $\mathcal{T}$.

Let u be the node represented by the triplet $\mathcal{S}[i]$; k be a positive integer at most equal to the maximum node degree in $\mathcal{T}$; and c be a symbol of Σ. The following list summarizes the operations supported by the compressed representation.

GetRankedChild(i, k) returns the position in $\mathcal{S}$ of the kth child of u; the output is -1 if this child does not exist. As an example, $\mathsf{GetChildren}(2, 2) = 6$ in Figure 1.

GetCharRankedChild(i, c, k) returns the position in $\mathcal{S}$ of the triplet representing the kth child of u among the ones whose label is c. The output is -1 if this child does not exist. As an example, $\mathsf{GetChildren}(1, \mathtt{B}, 2) = 4$ in Figure 1.

GetDegree(i) returns the number of children of u.

GetCharDegree(i,c) returns the number of children of u labeled c.

GetParent(i) returns the position in $\mathcal{S}$ of the triplet representing the parent of u. The output is -1 if $i = 1$ (the root). As an example, $\mathsf{GetParent}(8) = 4$ in Figure 1.

GetSubtree(i) returns the node labels of the subtree rooted at u. Any possible order (i.e. pre, in, post) may be implemented.

SubPathSearch(Π) Let Π be the labeled path $c_1 c_2 \cdots c_k$. This operation determines the range $\mathcal{S}[\mathsf{First}, \mathsf{Last}]$ of nodes which are *immediate* descendants of each occurrence of Π in $\mathcal{T}$. Note that all strings in $\mathcal{S}_\pi[\mathsf{First}, \mathsf{Last}]$ are prefixed by Π^R. As an example, $\mathsf{SubPathSearch}(BD) = [12, 13]$ and $\mathsf{SubPathSearch}(AB) = [5, 8]$ in Figure 1.

In the rest of this paper we will call the first six operations *navigational*, and the last one *search* operation. We remark that $\mathsf{SubPathSearch}(\Pi)$ is the one that actually distinguishes the $\mathsf{xbw}[\mathcal{T}]$ from all the other tree encodings proposed in the literature, like BP [43] and DFUDS or its variations [6, 23, 5, 29]. In section 5 we propose another search operation over $\mathsf{xbw}[\mathcal{T}]$ which is specialized to work on XML data. This further strengthens the generality of our transform.

An important ingredient of our compressed-indexing solution is the use of rank and select primitives over strings drawn from an arbitrary alphabet Σ. Given a string $S[1, t]$ over alphabet Σ:

- $\mathsf{rank}_c(S, q)$ is the number of times the symbol $c \in \Sigma$ occurs in the prefix $S[1, q]$.
- $\mathsf{select}_c(S, q)$ is the position of the q-th occurrence of the symbol c in S.

The algorithmic literature provides many efficient implementations for rank and select (e.g., see [45, 5] and references therein). We shall use the best known results in this context as a *black-box* for implementing our navigational and search operations. This actually shows that the $\mathsf{xbw}[\mathcal{T}]$ *reduces* the sophisticated compressed indexing of labeled trees to the basic problem of compressed rank and select queries over strings. As a result, any improvement to rank and select implementations would naturally lead to an improvement of our solutions. The following Lemma 5 states the best known bounds in this context.

Lemma 5 *Let S be a string over the alphabet Σ. We have:*

1. *For $|\Sigma| = O(\text{polylog}(|S|))$, the generalized wavelet tree of [20] supports* rank *and* select *queries in $O(1)$ time using $|S|H_0(S) + o(|S|)$ bits of space, and supports the retrieval of any character of S in the same time bound.*

2. *For general Σ, the data structure in [5] supports* rank *and* select *queries in $o(\log \log^3 |\Sigma|)$ time, using $|S|H_k(S) + o(|S| \log |\Sigma|)$ bits of space, and supports the retrieval of any character of S in the same time bound.*[4] ∎

[4]To be precise, rank takes $O(\log \log^2 |\Sigma| \log \log \log |\Sigma|)$ time, and select takes $O(\log \log |\Sigma| \log \log \log |\Sigma|)$ time.

Algorithm GetChildren(i)

1. **if** ($\mathcal{S}_\alpha[i] \in \Sigma_L$) **then return** -1; { $\mathcal{S}[i]$ *is a leaf* }
2. $c = \mathcal{S}_\alpha[i]$; { $\mathcal{S}[i]$ *is labeled* c }
3. $r = \mathsf{rank}_c(\mathcal{S}_\alpha, i)$;
4. $y = \mathsf{select}_1(\mathcal{A}, c)$; { $y = \mathsf{F}[c]$ }
5. $z = \mathsf{rank}_1(\mathcal{S}_{\mathsf{last}}, y-1)$;
6. $\mathsf{First} = \mathsf{select}_1(\mathcal{S}_{\mathsf{last}}, z+r-1)+1$;
7. $\mathsf{Last} = \mathsf{select}_1(\mathcal{S}_{\mathsf{last}}, z+r)$;
8. **return** (First, Last).

Figure 7: Algorithm for computing the block $\mathcal{S}[\mathsf{First}, \mathsf{Last}]$ of children of $\mathcal{S}[i]$, if any.

The compressed indexing of $\mathsf{xbw}[\mathcal{T}]$ will be based on three compressed data structures that support rank and select queries over the two strings $\mathcal{S}_\alpha$ and $\mathcal{S}_{\mathsf{last}}$, and over an auxiliary binary array $\mathcal{A}[1,t]$ defined as: $A[1]=1$, $\mathcal{A}[j]=1$ iff the first symbol of $\mathcal{S}_\pi[j]$ differs from the first symbol of $\mathcal{S}_\pi[j-1]$. It easy to see that, by means of rank and select operations over $\mathcal{A}$, we can succinctly implement the array F deployed in the algorithms of figures 4 and 5.

In the next sections we detail the implementation of navigational and search operations over $\mathcal{T}$, building them on rank and select data structures for arbitrary strings. This actually shows that the $\mathsf{xbw}[\mathcal{T}]$ *reduces* the sophisticated compressed indexing of labeled trees to the basic problem of compressed rank and select queries over strings.

4.1 Tree Navigation

We start this section by introducing a subroutine, called GetChildren(i), that returns the contiguous range of positions in $\mathcal{S}$ representing the children of $\mathcal{S}[i]$. The pseudocode is given in Figure 7. Given the label c of node $\mathcal{S}[i]$ (Step 2), GetChildren determines the number r of occurrences of c in $\mathcal{S}_\alpha[1,i]$ (Step 3), and then the position $\mathsf{F}[c]$ through a select operation on $\mathcal{A}$ (Step 4). This operation exploits the presence of all the symbols of Σ in $\mathcal{S}_\alpha$, where c is coded with an integer. By Property 3, the children of $\mathcal{S}[i]$ are located at the rth block of children following position $\mathsf{F}[c]$. Steps 5–7 compute this block. Note that $(\mathsf{Last} - \mathsf{First} + 1)$ provides the output of GetDegree(i).

The pseudocodes of GetRankedChild in Figure 8, and GetCharRankedChild in Figure 9, easily follow from algorithm GetChildren. We just point out that the value $(y2 - y1)$ computed in Step 4 of GetCharRankedChild provides the number of children of $\mathcal{S}[i]$ labeled with symbol c (hence the result of GetCharDegree (i,c)). We also notice that algorithm GetSubtree(i) can be implemented by invoking GetChildren over all nodes descending from $\mathcal{S}[i]$. By varying the order in which we process the children of every visited node, we can implement various kinds of tree visits– pre, in, post.

Example 1 Pick node u at entry $\mathcal{S}[2]$ of Figure 1. GetChildren(2) determines the label $\mathcal{S}_\alpha[2] = \mathtt{B}$ of u, the starting position $y = \mathsf{select}_1(\mathcal{A}, 2) = F[\mathtt{B}] = 5$ of the π-strings prefixed by B (whose integer code is 2), the rank $k = \mathsf{rank}_{\mathtt{B}}(\mathcal{S}_\alpha, 2) = 1$ of u's label in $\mathcal{S}_\alpha$, and $z = \mathsf{rank}_1(\mathcal{S}_\alpha, 4) = 1$. By Property 3, the children of u are located at block $k = 1$ counting from $\mathcal{S}[5]$, or equivalently at block $z + k = 2$ counting from the beginning. The algorithm reports $\mathsf{First} = \mathsf{select}_1(\mathcal{S}_{\mathsf{last}}, 1) + 1 = 5$ and $\mathsf{Last} = \mathsf{select}_1(\mathcal{S}_{\mathsf{last}}, 2) = 7$. Thus, u has $\mathsf{Last} - \mathsf{First} + 1 = 3$ children, located in the range $\mathcal{S}[5,7]$. ∎

Algorithm GetRankedChild(i, k)

1. $(\mathsf{First}, \mathsf{Last}) = \mathsf{GetChildren}(i)$;
2. **if** $(k > \mathsf{Last} - \mathsf{First} + 1)$ **return** -1;
3. **else return** $\mathsf{First} + k - 1$.

Figure 8: Algorithm for computing the kth child of node $\mathcal{S}[i]$. The output is -1 if the node is a leaf or k is larger than its fan-out.

Algorithm GetCharRankedChild(i, c, k)

1. $(\mathsf{First}, \mathsf{Last}) = \mathsf{GetChildren}(i)$;
2. $y1 = \mathsf{rank}_c(\mathcal{S}_\alpha, \mathsf{First} - 1)$;
3. $y2 = \mathsf{rank}_c(\mathcal{S}_\alpha, \mathsf{Last})$;
4. **if** $(k > y2 - y1)$ **return** -1;
5. **else return** $\mathsf{select}_c(\mathcal{S}_\alpha, y1 + k)$.

Figure 9: Algorithm for computing the kth c-labeled child of $\mathcal{S}[i]$, if any.

Algorithm GetParent is actually the inverse of GetChildren. It computes the symbol c that prefixes the upward path leading to $\mathcal{S}[i]$. This is actually the label of the parent of this node. As in GetChildren step 2 implements this by taking advantage of array $\mathcal{A}$ and the presence of all Σ's symbols in $\mathcal{S}_\alpha$. Then the parent of $\mathcal{S}[i]$ is searched among the nodes labeled c in $\mathcal{S}_\alpha$. To do this we exploit Property 3 in a reverse manner. Namely, we compute the number k of children-blocks in the range $\mathcal{S}[y, i]$ (Step 4), these are children of nodes labeled c and preceding i in the stable sort of $\mathcal{S}$. Then we select the kth occurrence of c in $\mathcal{S}_\alpha$ (Step 5), which is properly the parent of $\mathcal{S}[i]$.

Example 2 Consider Figure 1 and pick the child $\mathcal{S}[8]$ of node v in $\mathcal{T}$. GetParent(8) should therefore return 4, since $v = \mathcal{S}[4]$. The algorithm GetParent computes v's label $c = 2$ (i.e., B), $y = 5$ (i.e. $\mathcal{S}_\pi[5, 8]$ are strings prefixed by B) and $k = 1$ so $\mathcal{S}[i]$ belongs to the $(k + 1) = 2$nd group of children of nodes labeled B. Finally p is correctly set to $\mathsf{select}_{\mathtt{B}}(\mathcal{S}_\alpha, 2) = 4$. ∎

By using proper Rank/Select data structures over the arrays in $\mathsf{xbw}[\mathcal{T}]$ and $\mathcal{A}$, we can prove the following.

Algorithm GetParent(i)

1. **if** $(i == 1)$ **then return** -1;
2. $c = \mathsf{rank}_1(\mathcal{A}, i)$;
3. $y = \mathsf{select}_1(\mathcal{A}, c)$;
4. $k = \mathsf{rank}_1(\mathcal{S}_{\mathsf{last}}, i-1) - \mathsf{rank}_1(\mathcal{S}_{\mathsf{last}}, y-1)$;
5. $p = \mathsf{select}_c(\mathcal{S}_\alpha, k + 1)$;
6. **return** p.

Figure 10: Algorithm for computing the parent of $\mathcal{S}[i]$. The output is -1 if $\mathcal{S}[i]$ is the root of $\mathcal{T}$.

Theorem 6 *For any alphabet Σ, such that $|\Sigma| = O(\text{polylog}(t))$, there exists a succinct indexing of $\mathsf{xbw}[\mathcal{T}]$ that takes at most $tH_0(\mathcal{S}_\alpha) + 2t + o(t)$ bits and supports all navigational operations listed at the beginning of Section 4 in $O(1)$ time. The original tree $\mathcal{T}$, and any of its subtrees, can be recovered in optimal linear time.*

Proof: Use Lemma 5 (point 1) to implement rank and select data structures over $\mathcal{S}_\alpha$, $\mathcal{S}_{\mathsf{last}}$, and $\mathcal{A}$. Since arrays $\mathcal{S}_{\mathsf{last}}, \mathcal{A}$ are binary, we have $H_0(\mathcal{S}_{\mathsf{last}}) \leq 1$, $H_0(\mathcal{A}) \leq 1$, and the upper bound $2t + o(t)$ easily follows. ∎

We note that $H_0(\mathcal{S}_\alpha) \leq \log |\Sigma|$, hence we are indexing $\mathsf{xbw}[\mathcal{T}]$ in the same space of its plain representation, up to lower order terms (cfr. Theorem 1). This also means that $(1 + o(1)) \log |\Sigma|$ bits per node are enough to support navigational operations over $\mathcal{T}$. In other words, this succinct index over $\mathsf{xbw}[\mathcal{T}]$ is a *pointerless representation* of $\mathcal{T}$ with additional functionalities.

Theorem 7 *For any alphabet Σ, there exists a compressed representation for $\mathsf{xbw}[\mathcal{T}]$ that takes at most $t(H_k(\mathcal{S}_\alpha) + H_k(\mathcal{S}_{\mathsf{last}}) + H_k(\mathcal{A})) + o(t \log |\Sigma|)$ bits and supports all navigational operations listed at the beginning of Section 4 in $o(\log \log^3 |\Sigma|)$ time. The original tree $\mathcal{T}$ can be recovered in optimal linear time. A s-sized subtree of $\mathcal{T}$ can be recovered in $o(s \log \log^3 |\Sigma|)$ time.*

Proof: Use Lemma 5 (point 2) to implement rank and select over the three strings $\mathcal{S}_\alpha$, $\mathcal{S}_{\mathsf{last}}$ and $\mathcal{A}$. ∎

As we will show in Section 5 the string $\mathcal{S}_\alpha$ is the most demanding to be compressed; so we can crudely state the bound $H_k(\mathcal{S}_{\mathsf{last}}) + H_k(\mathcal{A}) \leq 2$. Then, from the above Theorem 7, we are able to index a labeled tree $\mathcal{T}$ using at most $2 + H_k(\mathcal{S}_\alpha)$ bits per node, where $H_k(\mathcal{S}_\alpha) \leq H_0(\mathcal{S}_\alpha)$ and it is possibly much smaller. If we compare this bound with the plain storage of $\mathsf{xbw}[\mathcal{T}]$ (Theorem 1), we note that it is never worse, but can be significantly better depending on the distribution of the node labels in $\mathcal{T}$. Furthermore, it offers navigational operations over tree $\mathcal{T}$. Compared to the compressed storage of $\mathcal{T}$ in Theorem 4, we note that our compressed-indexing bound is off of that one by just an additional $tH_k(\mathcal{A})$ term. This latter may be $o(t)$ in theory, and is usually negligible in practice as shown in Section 5.4.

4.2 Tree search

Given a labeled path $\Pi = c_1 c_2 \cdots c_l$, the algorithm SubPathSearch given in figure 11 aims at finding the nodes u which are *immediate* descendants of a subpath Π, anchored at any (internal) node of $\mathcal{T}$. Because of the sorting of $\mathcal{S}$, the triplets corresponding to these nodes are contiguous in $\mathcal{S}$, and their π-components are prefixed by Π^R (since they denote upward paths in $\mathcal{T}$). Hereafter we will use the notation $[\mathsf{First}, \mathsf{Last}]$ to indicate this range of $\mathcal{S}$'s entries. Given the range, we can easily count in constant time the number of nodes descending from Π in $\mathcal{T}$. A bit more tricky is the computation of the number of times Π occurs in $\mathcal{T}$ (see below).

Algorithm SubPathSearch computes the range $[\mathsf{First}, \mathsf{Last}]$ in $|\Pi| = l$ phases, each one preserving the following invariant:

Invariant of Phase i: At the end of the i-th phase, $\mathcal{S}_\pi[\mathsf{First}]$ is the first entry prefixed by $\Pi[1, i]^R$, and $\mathcal{S}_\pi[\mathsf{Last}]$ is the last entry prefixed by $\Pi[1, i]^R$, where s^R is the reversal of string s.

At the beginning (i.e. $i = 1$), First and Last are easily determined via the entries $\mathsf{F}[c_1]$ and $\mathsf{F}[c_1 + 1] - 1$, which point to the first and last entry of $\mathcal{S}_\pi$ prefixed by c_1 (by definition of array F). Since we do not have

Algorithm $\mathsf{SubPathSearch}(\Pi)$

1. $\mathsf{First} = \mathsf{F}(c_1)$; $\mathsf{Last} = \mathsf{F}(c_1 + 1) - 1$; $\{$*Use* $\mathcal{A}$ *to compute* $\mathsf{F}\}$
2. **if** ($\mathsf{First} > \mathsf{Last}$) **then return** "$\Pi$ is not a subpath of $\mathcal{T}$";
3. **for** $i = 2, \ldots, k$ **do**
4. $\quad k_1 = \mathsf{rank}_{c_i}(\mathcal{S}_\alpha, \mathsf{First} - 1)$; $z_1 = \mathsf{select}_{c_i}(\mathcal{S}_\alpha, k_1 + 1)$; $\{$ *first entry in* $\mathcal{S}_\alpha[\mathsf{First}, t]$ *labeled* c_i $\}$
5. $\quad k_2 = \mathsf{rank}_{c_i}(\mathcal{S}_\alpha, \mathsf{Last})$; $z_2 = \mathsf{select}_{c_i}(\mathcal{S}_\alpha, k_2)$; $\{$ *last entry in* $\mathcal{S}_\alpha[1, \mathsf{Last}]$ *labeled* c_i $\}$
6. $\quad$ **if** ($z1 > z_2$) **then return** "Π is not a subpath of $\mathcal{T}$";
7. $\quad \mathsf{First} = \mathsf{GetRankedChild}(z_1, 1)$; $\{$ *get the first child of* $\mathcal{S}[z_1]$ $\}$
8. $\quad \mathsf{Last} = \mathsf{GetRankedChild}(z_2, \mathsf{GetDegree}(z_2))$; $\{$ *get the last child of* $\mathcal{S}[z_2]$ $\}$
9. **return** $(\mathsf{First}, \mathsf{Last})$.

Figure 11: Compute the range of $\mathcal{S}$'s entries whose upward path is prefixed by $\Pi^R = c_l c_{l-1} \cdots c_1$.

array F, we implement this operations via rank and select queries over array $\mathcal{A}$. Let us assume that the invariant holds for Phase $i-1$, and prove that the ith iteration of the `for`-loop in algorithm $\mathsf{SubPathSearch}$ preserves the invariant. More precisely, let $\mathcal{S}_\pi[\mathsf{First}, \mathsf{Last}]$ be all entries prefixed by $\Pi[1, i-1]^R$. So $\mathcal{S}[\mathsf{First}, \mathsf{Last}]$ contains all nodes descending from $\Pi[1, i-1]$. $\mathsf{SubPathSearch}$ determines $\mathcal{S}[z_1]$ (resp. $\mathcal{S}[z_2]$) as the first (resp. last) node in $\mathcal{S}[\mathsf{First}, \mathsf{Last}]$ that descends from $\Pi[1, i-1]$ and is labeled c_i, if any (Steps 4–6). Then it jumps to the first child of $\mathcal{S}[z_1]$ and the last child of $\mathcal{S}[z_2]$. From Property 2 item 2, and the correctness of algorithms $\mathsf{GetChildren}$ and $\mathsf{GetDegree}$, we infer that the positions of these two children are exactly the first (resp. last) entry in $\mathcal{S}$ whose π-component is prefixed by $\Pi[1, i]^R$.

If we compute $\mathsf{Last} - \mathsf{First} + 1$ we get the number of offsprings from Π. If, instead, we are interested in the number of occurrences of path Π in $\mathcal{T}$, then we have to compute: $\mathsf{rank}_1(\mathcal{S}_{\mathsf{last}}, \mathsf{Last}) - \mathsf{rank}_1(\mathcal{S}_{\mathsf{last}}, \mathsf{First} - 1) + 1$. This operation actually counts the number of blocks of children (i.e. offsprings) descending from Π (Property 2, item 3).

Example 3 Refer to Figure 1, and let $\Pi =$ `BD`. $\mathsf{SubPathSearch}(\Pi)$ returns the range $[12, 13]$ as follows. At the beginning $\mathsf{First} = \mathsf{F}[\mathtt{B}] = 5$ and $\mathsf{Last} = \mathsf{F}[\mathtt{C}] - 1 = 8$. In fact, $\mathcal{S}[5, 8]$ are all nodes descending from the subpath `B`. The next phase takes $c_2 =$ `D`, and computes $k_1 = 0$ and $k_2 = 2$. As a result $z_1 = 5$ and $z_2 = 8$, the first child of $\mathcal{S}[5]$ is at $\mathcal{S}[12]$ and the last child of $\mathcal{S}[8]$ is at $\mathcal{S}[13]$ (it is actually the only child of $\mathcal{S}[8]$). Then, $\mathsf{SubPathSearch}$ correctly returns the range $[12, 13]$. The number of offsprings of subpath Π is 2, and the number of occurrences of subpath Π is also 2, as indeed we have two occurrences of 1 in the range $\mathcal{S}_{\mathsf{last}}[12, 13]$. ∎

We are ready to state the main result of this section, which easily follows from the pseudocode of Figure 11 and the use of the data structures described in Theorems 6 and 7.

Theorem 8 *For any alphabet Σ, there exists a compressed representation of $\mathsf{xbw}[\mathcal{T}]$ that supports subpath searches of a string Π in:*

- $O(|\Pi|)$ *time and at most* $tH_0(\mathcal{S}_\alpha) + 2t + o(t)$ *bits, if* $|\Sigma| = O(\text{polylog}(t))$*;*
- $o(|\Pi| \log\log^3 |\Sigma|)$ *time and at most* $t(H_k(\mathcal{S}_\alpha) + H_k(\mathcal{S}_{\mathsf{last}}) + H_k(\mathcal{A})) + o(t \log |\Sigma|)$ *bits, if* $\Sigma = \Omega(\text{polylog}(t))$*;*

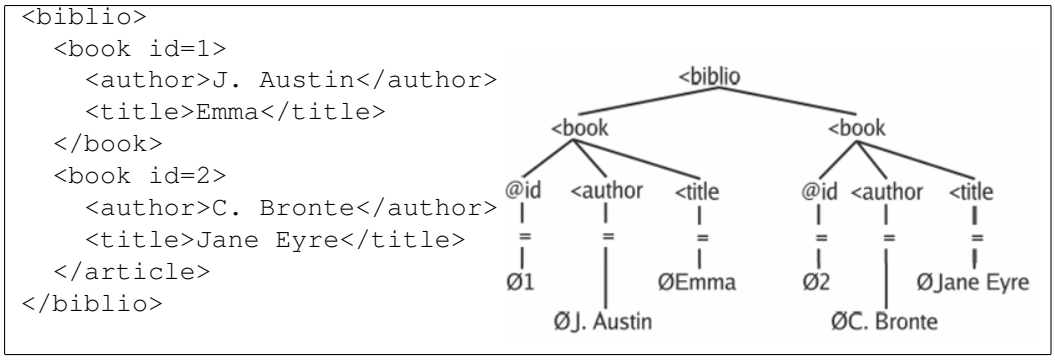

Figure 12: An XML document d (left) and its corresponding ordered labeled tree $\mathcal{T}$ (right).

Recently it has been shown in [29] how to design a compressed index for the binary array $\mathcal{S}_{\mathsf{last}}$ that achieves a better space bound than $H_k(\mathcal{S}_{\mathsf{last}})$. This data structure could be plugged into our algorithmic scheme to obtain the same time bounds of Theorems 6–7–8, but with improved space occupancy. This result further highlights the generality and elegance of our labeled-tree indexing approach based on $\mathsf{xbw}[\mathcal{T}]$ and rank-select primitives on strings.

5 A relevant application to XML data

In 1996 the W3C[5] started to work on XML as a way to enable data interoperability over the Internet; today, XML is the standard for information representation, exchange and publishing over the Web, and is getting embedded into many applications. XML is popular because it encodes a considerable amount of metadata in its plain-text format (see Figure 12); as a result, applications can be more savvy about the semantics of the items in the data source. This comes at a twofold cost. First, the XML representation is *verbose* because the entire schema description of each data item is repeated at each of its occurrences. Second, XML documents have a natural *tree structure* with mixed elements, with both text and attributes, so that XML queries are richer than commonly used SQL queries in that they include path and content searches on labeled trees.

In this section we address the basic problems of compression, navigation and searching of XML documents by designing a compressor called XBZIP, and a compressed index called XBZIPINDEX, whose algorithmic cores are based on the XBW-transform. We will also experimentally compare XBZIP and XBZIPINDEX with other known indexing and compression tools for XML data. The net result will be that XBZIP achieves comparable compression ratio to state-of-the-art XML compressors by simpler means, while XBZIPINDEX achieves significantly improved compression ratio and query performance with respect to known XML-compressed indexes.

5.1 The XBW-transform for XML data

As the relationships between elements in an XML document are defined by nested structures, XML documents are often modeled as *(DOM) trees* whose nodes are labeled with *strings of arbitrary length* drawn from a usually large alphabet. These strings are called *tag* or *attribute* names for the internal nodes, and *content data* (shortly PCDATA) for the leaves. Given an XML document d, we build an ordered labeled DOM tree $\mathcal{T}$ consisting of four types of nodes (see Figure 12):

[5] http://www.w3.org/XML/

$\mathcal{S}_{\text{last}}$	$\mathcal{S}_\alpha$	$\mathcal{S}_\pi$
1	<biblio	*empty string*
0	<book	<biblio
0	@id	<book<biblio
1	=	@id<book<biblio
1	**0**1	=@id<book<biblio
0	<author	<book<biblio
1	=	<author<book<biblio
1	**0**J. Austin	=<author<book<biblio
1	<title	<book<biblio
1	=	<title<book<biblio
1	**0**Emma	=<title<book<biblio
1	<book	<biblio
0	@id	<book<biblio
1	=	@id<book<biblio
1	**0**2	=@id<book<biblio
0	<author	<book<biblio
1	=	<author<book<biblio
1	**0**C. Bronte	=<author<book<biblio
1	<title	<book<biblio
1	=	<title<book<biblio
1	**0**Jane Eyre	=<title<book<biblio

Stable sort →

Rk	$\mathcal{S}_{\text{last}}$	$\mathcal{S}_\alpha$	$\mathcal{S}_\pi$
1	1	<biblio	*empty string*
2	1	=	<author<book<biblio
3	1	=	<author<book<biblio
4	0	<book	<biblio
5	1	<book	<biblio
6	0	@id	<book<biblio
7	0	<author	<book<biblio
8	1	<title	<book<biblio
9	0	@id	<book<biblio
10	0	<author	<book<biblio
11	1	<title	<book<biblio
12	1	=	<title<book<biblio
13	1	=	<title<book<biblio
14	1	=	@id<book<biblio
15	1	=	@id<book<biblio
16	1	**0**J. Austin	=<author<book<biblio
17	1	**0**C. Bronte	=<author<book<biblio
18	1	**0**Emma	=<title<book<biblio
19	1	**0**Jane Eyre	=<title<book<biblio
20	1	**0**1	=@id<book<biblio
21	1	**0**2	=@id<book<biblio

$$\widehat{\mathcal{S}}_{\text{last}} = 111010010011111$$
$$\widehat{\mathcal{S}}_{\alpha} = \text{<biblio==<book<book@id<author<title@id<author<title====}$$
$$\widehat{\mathcal{S}}_{\text{pcdata}} = \mathbf{0}\text{J. Austin}\mathbf{0}\text{C. Bronte}\mathbf{0}\text{Emma}\mathbf{0}\text{Jane Eyre}\mathbf{0}1\mathbf{0}2$$

Figure 13: The set $\mathcal{S}$ computed from document d of Figure12 after the pre-order visit of $\mathcal{T}$ (left), and after the stable sort (right). The arrays $\widehat{\mathcal{S}}_{\text{last}}, \widehat{\mathcal{S}}_\alpha, \widehat{\mathcal{S}}_{\text{pcdata}}$, output of XBW$[d]$, are show at the bottom.

1. each occurrence of an opening tag <t> originates a **tag node** labeled with the string <t;
2. each occurrence of an attribute name a originates an **attribute node** labeled with the string @a;
3. each occurrence of an attribute value or textual content of a tag, say ρ, originates two nodes: a **text-skip node** labeled with the character =, and a **content node** labeled with the string $\mathbf{0}\rho$, where $\mathbf{0}$ is a special character not occurring elsewhere in d.

This encoding of the node labels allows us to easily distinguish between internal-node labels vs leaf labels because the former are prefixed by $\{<, @, =\}$ and the latter are prefixed by the special symbol $\mathbf{0}$ (cfr. Σ_N and Σ_L in Section 2).

The *structure* of the tree $\mathcal{T}$ is derived from the XML document d as follows. An XML *well-formed* substring of d, say $\sigma =$ <t a$_1$="ρ_1" ... a$_l$="ρ_1"> τ </t>, generates a subtree of $\mathcal{T}$ rooted at a node labeled <t. This node has l children (subtrees) originating from t's attribute names and values (i.e. @a$_i \rightarrow = \rightarrow \rho_i$), plus other children (subtrees) originating by the recursive parsing of the string τ. Note that attribute nodes and text-skip nodes have only one child. Tag nodes may have an arbitrary number of children. Content nodes have no children and thus form the leaves of $\mathcal{T}$.[6]

Given this labeled tree representation of an XML document, it is natural to use the XBW-transform for compactly representing it and for supporting navigational and search operations over its tree structure. We actually propose a slight variation of the XBW motivated by the features of the XML documents and the XML queries. Let d be an XML document; $\mathcal{T}$ be the corresponding tree; and n be the number of internal nodes of $\mathcal{T}$. We pose:

[6]Document d may contain *empty tags* not including anything (i.e. <t/> or <t></t>). These tags are managed by transforming them to <t>λ</t>, where λ is a special symbol not occurring elsewhere in d.

Definition 1 *The* XBW*-transform of d consists of three arrays:* $\mathsf{xbw}[d] = \langle \widehat{\mathcal{S}}_{\mathsf{last}}, \widehat{\mathcal{S}}_\alpha, \widehat{\mathcal{S}}_{\mathsf{pcdata}} \rangle$*, where* $\widehat{\mathcal{S}}_{\mathsf{last}} = \mathcal{S}_{\mathsf{last}}[1,n]$*,* $\widehat{\mathcal{S}}_\alpha = \mathcal{S}_\alpha[1,n]$*, and* $\widehat{\mathcal{S}}_{\mathsf{pcdata}} = \mathcal{S}_\alpha[n+1,t]$*.*

Note that we are still using the acronym `xbw` for this slight variation of the XBW-transform, but we are adopting the argument d instead of $\mathcal{T}$. An illustrative example for $\mathsf{xbw}[d]$ is given in Figure 13. Notice that $\widehat{\mathcal{S}}_\alpha$ contains the labels of the internal nodes only, whereas $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ contains the labels of the leaves, that is, the PCDATA. This is because if u is a leaf, the first symbol of its upward path $\pi[u]$ is = which we assume be lexicographically larger than the characters < and @ that prefix the upward path of internal nodes. Since leaves have no children, $\mathcal{S}_{\mathsf{last}}[n+1,t]$ consists of just 1s, and thus can be dropped.

Let us comment on some of our implementation choices for $\mathsf{xbw}[d]$. Even if the symbols of Σ may be very long strings, we do not need any additional information for recovering them from the strings $\widehat{\mathcal{S}}_\alpha$ and $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$. In fact, characters $\{<, @, =\}$ cannot appear inside a tag or attribute name because of the XML syntax, and the special character **0** cannot appear inside PCDATA. Additionally, the use of the text-skip nodes (labeled =) is crucial to separate PCDATA from internal-node labels, which otherwise would be intermixed within $\mathcal{S}_\alpha$. These choices have a twofold advantage: (i) the two strings $\widehat{\mathcal{S}}_\alpha$ and $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ are *a fortiori* homogeneous and hence highly compressible (see Sect. 5.2), (ii) search and navigational operations over $\mathcal{T}$ are greatly simplified (see Sect. 5.3). Details will be given in the following subsections.

We conclude this section by observing that our compressor and index do need to construct $\mathsf{xbw}[d]$. We did not implement the optimal algorithm of Section 2.1, but we followed a simpler approach motivated by the fact that in practice the DOM trees are *shallow*. Our algorithm represents $\mathcal{S}_\pi$ as an array of pointers to $\mathcal{T}$ nodes and (indirectly) sorts this array by the standard C-procedure `qsort`. The time required to build $\mathsf{xbw}[d]$ on few hundred MBs of XML data is a few tens of seconds. Of course, future algorithmic engineering research might be devoted to make this algorithm scaling better on larger XML collections!

5.2 Compressing XML data: the XBZIP tool

Most XML-conscious compressors— like XMILL [35], SCMPPM [1], XMLPPM [10]— are designed to "compress together" the data enclosed in the same tag, or in a few immediately enclosing tags, since such data usually have similar statistics. In Section 3 we called those enclosing tags: k-contexts. Those compressors usually look at small k since it is much space (and time) consuming to maintain separate statistics for all k-contexts as k grows. The XBW-transform provides a simple and efficient mechanism to take advantage of the regularities occurring at large k, without incurring in their computational drawbacks.

Suppose the XML fragment of Figure 12 is a part of a large bibliographic database for which we have computed the XBW-transform. Consider the string `=<author`. The properties of the XBW-transform ensure that the labels of the nodes whose upward path is prefixed by `=<author` are consecutive in $\mathcal{S}_\alpha$. In other words, there is a substring of $\mathcal{S}_\alpha$ consisting of all the data (immediately) enclosed in an `<author>` tag. Similarly, another substring of $\mathcal{S}_\alpha$ contains the labels of all nodes whose upward path is prefixed by `<book<biblio`, and therefore this substring likely consists of `<author`, `<title` or `@id` strings. This means that $\mathcal{S}_\alpha$, and therefore $\widehat{\mathcal{S}}_\alpha$ and $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$, likely have a strong *local homogeneity property*, hence they are highly compressible.

If we are only interested in a compressed (non-searchable) representation of d, we simply need to compress the arrays $\widehat{\mathcal{S}}_{\mathsf{last}}$, $\widehat{\mathcal{S}}_\alpha$ and $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$. This is done by the XBZIP tool whose pseudocode is given in Figure 14. Experimentally we found that, instead of compressing $\widehat{\mathcal{S}}_{\mathsf{last}}$ and $\widehat{\mathcal{S}}_\alpha$ separately, it is more convenient to merge them in a unique array $\widehat{\mathcal{S}}'_\alpha$ obtained from $\widehat{\mathcal{S}}_\alpha$ adding a label `</` in correspondence of

Algorithm XBZIP

1. Compute $\mathsf{xbw}[d] = \langle \widehat{\mathcal{S}}_{\mathsf{last}}, \widehat{\mathcal{S}}_{\alpha}, \widehat{\mathcal{S}}_{\mathsf{pcdata}} \rangle$;
2. Merge $\widehat{\mathcal{S}}_{\alpha}$ and $\widehat{\mathcal{S}}_{\mathsf{last}}$ into $\widehat{\mathcal{S}}'_{\alpha}$;
3. Compress separately the strings $\widehat{\mathcal{S}}'_{\alpha}$ and $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$.

Figure 14: Pseudocode of XBZIP. Our implementation uses PPMDI in Step 3.

bits equal to 1 in $\widehat{\mathcal{S}}_{\mathsf{last}}$. This is exactly what Step 2 does in XBZIP. For example, merging the arrays $\widehat{\mathcal{S}}_{\mathsf{last}}$ and $\widehat{\mathcal{S}}_{\alpha}$ of Figure 13 yields:

$$\begin{aligned} \widehat{\mathcal{S}}'_{\alpha} \;=\; & \texttt{<biblio</=</=</<book<book</@id<author} \\ & \texttt{<title</@id<author<title</=</=</=</=</} \end{aligned}$$

This simple strategy captures the repetitiveness in the tree structure, so that, somewhat surprisingly, it yields compression ratios close to sophisticated state-of-the-art compressors. Theoretical and algorithmic engineering research is needed to further investigate the efficiency and efficacy of the other sophisticated strategies based on the XBW-transform, discussed in Section 3 and detailed in [16, 29].

5.3 Indexing compressed XML data: the XBZIPINDEX tool

In Section 4 we showed that navigation and search operations over a labeled tree can be implemented with the XBW transform by means of rank and select queries over strings. In this section we provide practical implementations for these operations, and introduce the following *content-based query* motivated by XML applications.

ContentSearch(Π, β)**.** Let Π be a labeled path and β be a string of arbitrary length. This operation retrieves all leaves of $\mathcal{T}$ that (immediately) descend from Π and contain β as a substring of their labels.

We remark that the leaves considered by ContentSearch must have Π as immediately enclosing context, and thus cannot be arbitrary descendant of Π in $\mathcal{T}$. We will comment on such a variation in Section 6.

As an example, let $\Pi = <$ `title` and $\beta =$ `Jane` in the XML document in Figure 12. The query ContentSearch(Π, β) returns the leaf containing the PCDATA: **0**`Jane Eyre`. We notice that the operation SubPathSearch corresponds to an XPATH query having the form `//`Π, whereas the operation ContentSearch corresponds to an XPATH query of the form `//`Π`[contains(.,`β`)]`.

Text book solutions to represent XML documents for navigation use a mixture of pointers and hash arrays. These representations constitute the standard for DOM tree encodings, but unfortunately are space consuming and practical only for small XML documents. Benchmarks show that DOM tree encodings need at least 4 times the original XML file size. This can be understood as follows: the simplest (empty) tag `<a/>` requires 4 bytes in the XML document, but at least 16 bytes as tree node: a name pointer plus three node pointers to the parent, the first child, and the next sibling. Of course, there exist more compact DOM tree encodings, e.g. Galax [12], but they use yet more memory than the original XML document and are very slow on SubPathSearch and ContentSearch queries because they need scanning the whole DOM tree.

Algorithm XBZIPINDEX

1. Compute $\mathsf{xbw}[d] = \langle \widehat{\mathcal{S}}_{\mathsf{last}}, \widehat{\mathcal{S}}_{\alpha}, \widehat{\mathcal{S}}_{\mathsf{pcdata}} \rangle$;
2. Store $\widehat{\mathcal{S}}_{\mathsf{last}}$ using a compressed representation supporting rank/select queries (see text);
3. Store $\widehat{\mathcal{S}}_{\alpha}$ using a compressed representation supporting rank/select queries (see text);
4. Split $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ into buckets, such that two elements are in the same bucket if they have the same upward path;
5. Build a compressed (full-text) index on each bucket (see text).

Figure 15: Pseudocode of XBZIPINDEX.

If SubPathSearch is a key concern, we may use any *summary index* data structure [9] that represents *all* paths of the tree document in an index (two famous examples are Dataguide [24], and 1- or 2-indexes [38]). This significantly increases the space needed by the index, and yet, it does not support ContentSearch queries efficiently. If ContentSearch queries are the prime concern, we need to resort to more sophisticated approaches— like XML-native search engines, e.g. XQUEC [2], F&B-INDEX [51], etc.. All these engines need space several times larger than the size of the indexed XML document. At the other extreme, if space is a primary concern we may use any XML-queryable compressors, like [49, 39, 11, 8], but we would still incur into the scan of the whole *compressed* XML file and need the decompression of large parts of it in the worst case.

We now show that $\mathsf{xbw}[d]$ can be combined with proper compressed indexing data structures for rank, select, and substring search operations [19], to resolve the dichotomy of time-efficient vs space-efficient solutions for XML compressed indexing. To this end we design a tool, called XBZIPINDEX, that supports on a compressed representation of the XML document efficient tree navigation (forward and backward), SubPathSearch and ContentSearch operations. The pseudocode for XBZIPINDEX is given in Figure 15. Note that this tool has additional features and may find other applications besides XML compressed searching. For example it could be used within *native XML search engines* for providing either subpath statistics for XML-query optimizations or as a document-collection storage system. Or it could be used within an *XML visualizer* for compressing an XML document, still supporting efficient subtree decompression and visualization. Details on the implementation of XBZIPINDEX follow.

The array $\widehat{\mathcal{S}}_{\mathsf{last}}$. To implement rank_1 and select_1 operations over $\widehat{\mathcal{S}}_{\mathsf{last}}$ we use a simple *one-level bucketing* storage scheme. We choose a constant L (default is $L = 1000$), and partition $\widehat{\mathcal{S}}_{\mathsf{last}}$ into *variable*-length blocks containing L bits set to 1. For each block we store:

- the number of 1's preceding this block in $\widehat{\mathcal{S}}_{\mathsf{last}}$ (called 1-*blocked rank*);
- a compressed image of the block obtained by GZIP;[7]
- a pointer to the compressed block and its 1-blocked rank.

It is easy to see that rank_1 and select_1 operations over $\widehat{\mathcal{S}}_{\mathsf{last}}$ can be implemented by decompressing and scanning a single block, plus a binary search over the (small) table of *1-blocked ranks*.

The array $\widehat{\mathcal{S}}_{\alpha}$. Recall that $\widehat{\mathcal{S}}_{\alpha}$ contains only the labels of the internal nodes of $\mathcal{T}$. We represent it using again a *one-level bucketing* storage scheme. We partition $\widehat{\mathcal{S}}_{\alpha}$ into *fixed*-length blocks (default is 8Kb), and for each block we store:

[7] We experimented other approaches, e.g. Elias coding on the distances between 1s [52] or better string compressors than GZIP, but they were not competitive in terms of time-space performance. In fact, the size of $\widehat{\mathcal{S}}_{\mathsf{last}}$ is negligible with respect to $\mathcal{S}_{\alpha}$, see Table 1.

- a compressed image of the block (obtained using GZIP). Note that single blocks are usually highly compressible because of the local homogeneity of $\widehat{\mathcal{S}}_\alpha$;[8]
- a table containing for each internal-node label c the number of its occurrences in the preceding prefix of $\widehat{\mathcal{S}}_\alpha$ (called *c-blocked ranks*);
- a pointer to the compressed block and its c-blocked rank.

Since the number of *distinct* internal-node labels is usually small with respect to the document size, c-blocked ranks can be stored without adopting any sophisticated solution. The implementation of $\mathsf{rank}_c(\widehat{\mathcal{S}}_\alpha, i)$ and $\mathsf{select}_c(\widehat{\mathcal{S}}_\alpha, i)$ easily derives from the information we have stored.

The array $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$. This array is usually the largest component of $\mathsf{xbw}[d]$ (see the last column of Table 1). Recall that $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ consists of the PCDATA items of d, ordered according their upward paths. Note that the procedures for navigating and searching $\mathcal{T}$ do not require rank/select operations over $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$. Therefore we use a representation of $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ that efficiently supports SubPathSearch and ContentSearch operations. To this end we use a bucketing scheme where buckets are induced by the upward paths. Formally, let $\mathcal{S}_\pi[i, j]$ be a maximal interval of equal strings in $\mathcal{S}_\pi$. We form one bucket of $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ by concatenating the strings in $\widehat{\mathcal{S}}_{\mathsf{pcdata}}[i, j]$. In other words, two elements of $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ are in the same bucket iff they have the same upward path. Each block will likely be highly compressible, since is formed by *homogeneous strings* having the same full-context.[9] For each bucket we store the following information:

- an *FM-index* [18, 19] of the bucket.[10] The FM-index is a compressed full-text index that supports efficient substring searches which access only a tiny portion of the compressed bucket. This portion is proportional to the length of the searched string, and not to the length of the bucket itself (see [19] for details);
- a counter of the number of PCDATA items preceding the current bucket in $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$;
- a pointer to the FM-indexed block and its counter.

Using this representation of $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$, we can answer the query `//`Π`[contains(.,`β`)]` as follows (pseudocode in Figure 16). The procedure SubPathSearch identifies the nodes whose leading path is Π, being internal nodes or leaves. Since ContentSearch looks for leaves only, we identify the substring $\widehat{\mathcal{S}}_{\mathsf{pcdata}}[\mathsf{F}, \mathsf{L}]$ that contains the labels of the leaves whose leading path is Π=. Note that $\widehat{\mathcal{S}}_{\mathsf{pcdata}}[\mathsf{F}, \mathsf{L}]$ consists of an integral number of buckets, say b, because of our partitioning strategy of $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$. To answer the query, we then search for β in these b buckets using their FM-indexes. The time cost of ContentSearch is efficient for *selective* queries.

Lemma 9 *The compressed index* XBZIPINDEX *identifies the buckets of $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ containing β's occurrences in time proportional to $|\Pi| + b|\beta|$, where b is the number of distinct upward paths prefixed by Π in $\mathcal{T}$. Solving the operation* ContentSearch(Π, β) *takes additional $occ \cdot \mathtt{polylog}(N)$ time, where occ is the number of occurrences of β in those b buckets, and N is the total length of* PCDATA *in $\mathcal{T}$.*

Proof: Recall that algorithm SubPathSearch takes time proportional to $|\Pi|$ (Theorem 8) to identify the range of nodes whose leading path is Π. Furthermore, the FM-index [19] takes $O(|\beta|)$ time to count the number occ of occurrences of β in each indexed bucket. Retrieving those occurrences and thus determining the leaves output of ContentSearch takes additional $occ \cdot \mathtt{polylog}(N)$ time [19]. ∎

[8] Other trade-offs could be possible by using compressors offering different time vs. compression ratios trade-offs, like PPMDI or BZIP2. We preferred GZIP because of its time efficiency and reasonable compression performance on $\widehat{\mathcal{S}}_\alpha$.

[9] Notice that XCQ [34] uses a similar partitioning of the PCDATA, however, subsequent queries are supported by *fully* scanning the tree structure.

[10] We used the following parameter settings for the FM-index (cfr [18]): $b = 2$Kb, $B = 32$Kb and $f = 0.05$. These parameters can be tuned for trading space usage for query time.

Algorithm $\mathsf{ContentSearch}(\Pi, \beta)$

1. $(\mathsf{First}, \mathsf{Last}) \leftarrow \mathsf{SubPathSearch}(\Pi)$;
2. $\mathsf{F} \leftarrow \mathsf{rank}_{=}(\widehat{\mathcal{S}}_{\alpha}, \mathsf{First} - 1) + 1$;
3. $\mathsf{L} \leftarrow \mathsf{rank}_{=}(\widehat{\mathcal{S}}_{\alpha}, \mathsf{Last})$;
4. Let $\mathcal{B}[i, j]$ be the range of buckets covering $\widehat{\mathcal{S}}_{\mathsf{pcdata}}[\mathsf{F}, \mathsf{L}]$;
5. Search for β in the FM-INDEX of the bucket $\mathcal{B}[h]$, for $h = i, i + 1, \ldots, j$;
6. **Return** the indexes of the buckets that contain at least one occurrence of β.

Figure 16: Returning the leaves of $\mathcal{T}$ whose leading (sub)path is Π and whose label contains β.

DATASET	SIZE (BYTES)	TREE SIZE	#LEAVES	TREE DEPTH MAX/AVG	#TAG AND ATTR (DISTINCT)	$\lvert\widehat{\mathcal{S}}_{\alpha}\rvert$ in bytes	$\lvert\widehat{\mathcal{S}}_{\mathsf{pcdata}}\rvert$ in bytes
PATHWAYS	79,054,143	9,338,092	5,044,682	10 / 3.6	4,293,410 (49)	24,249,238	36,415,927
DBLP	133,856,133	10,804,342	7,067,935	7 / 3.4	3,736,407 (40)	24,576,759	75,258,733
SWISSPROT	114,820,211	13,310,810	8,143,919	6 / 3.9	5,166,891 (99)	30,172,233	51,511,521
NEWS	244,404,983	8,446,199	4,471,517	3 / 2.8	3,974,682 (9)	28,319,613	176,220,422

Table 1: XML documents used in our experiments.

5.4 Some experimental results

The results of our paper are mainly theoretical. Nevertheless, we comment in this section on some of the experimental results published in [17], for highlighting the impact of the XBW-transform on XML compression and indexing. The reader interested in a much deeper experimental analysis may refer to the cited paper.

The tools XBZIP and XBZIPINDEX are packaged in a library, called XBZIPLIB, consisting of about 4000 lines of C-code and running under Linux and Windows. This library can be either included in another software or it can be directly used at the command-line with a full set of *options* for compressing, indexing and searching XML documents. We have tested it on a PC running Linux with two P4 CPUs at 2.6Ghz, 512Kb cache, and 1.5Gb internal memory. Table 1 reports the characteristics of the four XML files used in our experiments. They cover a range of XML data formats, both data centric or text centric, and deep or shallow tree structures.[11]

We have evaluated the real advantages of XBZIP with respect to state-of-the-art XML-conscious compressors like XMILL [35] and SCMPPM [1], as well as against general-purpose string compressors like GZIP, BZIP2, and PPMDI. The experiments, summarized in Table 2, show two opposite facts. As expected, XML-conscious compressors are better than commodity compressors; but the absolute difference in their compression ratio is within a 5% (cfr. PPMDI), which is surprisingly low. Actually, XBZIP and SCMPPM are the best compressors on the experimented data and achieve about the same compression ratio ranging from 1.84% of PATHWAYS to 10.61% of SWISSPROT. The time efficiency of XML-conscious compressors must be significantly improved. Profiling shows that 90% of XBZIP running time is spent for the computation of the XBW-transform (see Section 5.1 for comments on this issue). The decompression time of XBZIP is comparable to others.

Our experimental results show that XML-conscious compressors are still far from being a *clearly* advantageous alternative to general-purpose compressors. Nevertheless, the best performance achieved

[11] Source data: PATHWAYS is at `www.genome.jp/kegg/xml/`; DBLP and SWUISSPROT are at `www.cs.washington.edu/research/xmldatasets/`; and NEWS is at `www.di.unipi.it/~gulli/`.

DATASET	GZIP	BZIP2	PPMDI	XMILL	SCMPPM	XBZIP
PATHWAYS	7,78	4,70	3,93	10,07	3,13	1,84
DBLP	17,90	11,94	10,53	10,79	8,67	9,69
SWISSPROT	11,97	7,60	6,73	5,68	5,21	4,66
NEWS	22,94	15,06	12,62	11,54	10,71	10,71

Table 2: Comparison of XML compressors: XMILL, SCMPPM and XBZIP use PPMDI as their base compressor.

DATASET	HUFFWORD	XPRESS	XQZIP	XBZIPINDEX	XBZIP
PATHWAYS	33.68	–	–	3.62	1.84
DBLP	44.00	48	30	14.13	9.69
SWISSPROT	43.10	42	38	7.87	4.66
NEWS	45.15	–	–	13.52	10.61

Table 3: Compression ratio achieved by XML-queryable compressors over the files in our dataset. For XPRESS and XQZIP we report results taken from [39, 11] (the symbol – indicates a result not available in these papers). Note that we can trade space usage for query time by tuning the parameters of the FM-index [18].

by XBZIP lead us to think favorably about the XBW-compression paradigm in that XBZIP has not yet fully deployed it: we are simply applying PPMDI on $\mathsf{xbw}[d]$'s arrays without fully taking advantage of their local homogeneity properties (see Section 5.2 and comments therein).

A much more positive scenario arises when dealing with the practical performance of our compressed index XBZIPINDEX. In Table 3 we compare our XBZIPINDEX against the best known XML-queryable compressors. Some figures are missing because some software is either no longer available, or is unable to run on our XML files. However, whenever possible we report on the performance of these tools as stated in their reference papers. We point out that HUFFWORD [40] is not an XML-queryable compressor, but a typical storage scheme of (Web) search engines and Information Retrieval tools. Therefore we use its compression performance as a *lower bound* to the storage complexity of these approaches (see e.g. [31]).

Looking at Table 3 we observe that XBZIPINDEX significantly improves the compression ratio of the known XML-queryable compressors from 20% to 35% of the original document size. Table 4 details the space required by the various indexing data structures present in XBZIPINDEX, and its last two columns highlight the additional cost of storing the indexing information upon XBZIP. As expected, the indexing of $\widehat{\mathcal{S}}_{\mathsf{last}}$ and $\widehat{\mathcal{S}}_{\alpha}$ requires negligible space, thus proving again that these two strings are highly compressible and even a simple compressed-indexing approach, as the one we adopted in XBZIPINDEX, pays off. Conversely, $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$ takes most of the space and a fine tuning of the FM-index parameters might further improve the performance of XBZIPINDEX (see Section 5.3). This also shows that the reduction from labeled-tree indexing to text indexing, induced by the XBW-transform, strengthens the interest toward the design of this string-based indexing tools [21, 45].

As far as query and navigation operations are concerned, the large set of experiments in [17] showed that navigational and subpath searches are pretty much insensitive to the document size, as theoretically predicted, and indeed require few milliseconds. Conversely, all the others XML-queryable compressors require tens of seconds per query because they need scanning the whole set of compressed data.

DATASET	% INDEX $\widehat{\mathcal{S}}_{\mathsf{last}}$	% INDEX $\widehat{\mathcal{S}}_{\alpha}$	INDEX $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$	AUXILIARY	BYTES PER NODE
PATHWAYS	1.7	0.8	6.0	9.7	0.31
DBLP	4.9	2.3	32.3	8.1	1.75
SWISSPROT	2.2	2.5	14.0	8.0	0.68
NEWS	1.0	0.5	18.5	0.6	3.91

Table 4: Space occupancy. Percentage of each index part with respect to the corresponding indexed string. *Auxiliary* info includes all the prefix-counters mentioned in Section 5.3, and it is expressed as a percentage of the total index size. The last column gives an estimate of the average number of bytes spent for each tree node.

6 Conclusions

We have introduced the XBW-transform as a new approach to compress and index tree-shaped data. This transform allows to *reduce* the compressed indexing of labeled trees to the basic problem of compressed rank and select queries over strings. Consequently, any improvement to rank and select implementations would naturally lead to an improvement of our solutions for the labeled-tree compression and indexing problem.

Compared to other tree encodings proposed in the literature, like BP [43] and DFUDS or its variations [6, 23, 5, 29], the XBW-transform is the only one supporting subpath search operations, which have applications to XML data, as we largely commented in Section 5. In this paper we have also proposed an implementation of the XBW-transform and tested its practical performance on some real datasets. The experimental results are promising and show that there is still much room for improvement at the software level.

For the theory, we list three data structural and compression problems whose solution, combined with the XBW-transform, would extend our results even more.

Problem 1. In our approach, each label l of a node u has been treated as an element of given alphabet Σ. However in some applications, of which XML is just one example, labels are in fact strings of arbitrary length drawn from a different alphabet Γ. In this case two notions of context seem to be relevant, together with the corresponding entropy measurement. A "vertical" context for l as considered in Section 3, given by the labeled sequence $\pi[u] \in \Sigma^*$ in the path from u's parent to the tree root; and a "horizontal" context for each character c of l, given by the sequence $\sigma(c) \in \Gamma^*$ of the characters preceding c in l. We leave open the problem of defining a new notion of entropy that takes into account both contexts, and designing an efficient compression algorithm that achieves optimality under this new notion.

Problem 2. Motivated by XML applications, we would like to extend operation $\mathsf{ContentSearch}(\Pi, \beta)$ to searching for all leaves that *descend from* a subpath Π and whose label contains β as a substring. ContentSearch is now limited to leaves whose *leading* path is Π. Even in this simpler setting, our solution of Section 5.3 is sub-optimal in space and time because of the use of the bucketing of $\widehat{\mathcal{S}}_{\mathsf{pcdata}}$. From the implementation of ContentSearch, it seems that improving this solution requires the avoidance of the bucketing scheme, and thus the design of a compressed full-text index that supports a sort of *position-restricted* substring search operation. Known compressed indexes[45] cannot restrict the search to a substring of the indexed text. Some known full-text indexes [36] do support this restriction but they are not compressed.

Problem 3. Is it possible to design a *unique* transform that combines the navigational and search operations of the XBW, with the sophisticated ancestor, descending and subtree-size queries of DFUDS-encoding ?

References

[1] J. Adiego, P. de la Fuente, and G. Navarro. Merging prediction by partial matching with structural contexts model. In *Procs of IEEE Data Compression Conference (DCC)*, page 522, 2004.

[2] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. XQueC: pushing queries to compressed XML data. In *Proc. 29th International Conference on Very Large Data Bases (VLDB)*, pages 1065–1068, 2003.

[3] D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proc. 17th Combinatorial Pattern Matching conference (CPM)*, pages 318–329, 2006.

[4] J. Barbay, A. Golynski, I. Munro, and S.S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proc. 17th Combinatorial Pattern Matching conference (CPM)*, 2006.

[5] J. Barbay, M. He, J.I. Munro, and S. Srinivasa Rao. Succinct indexes for string, bynary relations and multi-labeled trees. In *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007.

[6] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, 2005.

[7] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[8] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML documents. In *Proc. 31st International Conference on Very Large Data Bases*, 2005.

[9] B. Catania, A. Maddalena, and A. Vakali. XML document indexes: a classification. *IEEE Internet Computing*, pages 64–71, September-October 2005.

[10] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Procs of IEEE Data Compression Conference (DCC)*, pages 163–172, 2001.

[11] J. Cheng and W. Ng. XQzip: Querying compressed XML using structural indexing. In *International Conference on Extending Database Technology*, pages 219–236, 2004.

[12] M. F. Fernandez, J. Simeon, B. Choi, A. Marian, and G. Sur. Implementing Xquery 1.0: the galax experience. In *Proc. of 29th International Conference on Very Large Data Bases (VLDB)*, pages 1077–1080, 2003.

[13] P. Ferragina, R. Giancarlo, and G. Manzini. The engineering of a compression boosting library: Theory vs practice in bwt compression. In *Proc. 14th European Symposium on Algorithms (ESA '06)*, pages 756–767. Springer Verlag LNCS n. 4168, 2006.

[14] P. Ferragina, R. Giancarlo, and G. Manzini. The myriad virtues of wavelet trees. In *Proc. 33th International Colloquium on Automata and Languages (ICALP '06)*, pages 561–572. Springer Verlag LNCS n. 4051, 2006.

[15] P. Ferragina, R. Giancarlo, G. Manzini, and M. Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52:688–713, 2005.

[16] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, 2005.

[17] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching xml data via two zips. In *Proc. 15th International World Wide Web Conference (WWW)*, pages 751–760, 2006.

[18] P. Ferragina and G. Manzini. An experimental study of a compressed index. *Information Sciences: special issue on "Dictionary Based Compression"*, 135:13–28, 2001.

[19] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[20] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc. 11th Symposium on String Processing and Information Retrieval (SPIRE '04)*, pages 150–160. Springer-Verlag LNCS n. 3246, 2004. To appear in "ACM Transactions on Algorithms", see also Tech. Report of Univ. Chile #TR/DCC-2004-5, 2004.

[21] P. Ferragina and G. Navarro. Pizza&Chili corpus home page. `http://pizzachili.dcc.uchile.cl/` or `http://pizzachili.di.unipi.it/`.

[22] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007.

[23] R.F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 1–10, 2004.

[24] R. Goldman and J. Widom. Dataguides: enabling query formulation and optimization in semistructured databases. In *Proc. of 23rd International Conference on Very Large Data Bases (VLDB)*, pages 436–445, 1997.

[25] A. Golynski, I. Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[26] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proc. 17th Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 295–306, 2006.

[27] A. Gupta, W.K. Hon, R. Shah, and J.S. Vitter. Dynamic rank/select dictionaries with applications to XML indexing. Technical Report Purdue University, 2006.

[28] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[29] J. Jansson, K. Sadakane, and W.K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007.

[30] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, pages 943–955. Springer-Verlag LNCS n. 2719, 2003.

[31] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proc. 20th International Conference on Data Engineering (ICDE)*, page 829, 2004.

[32] S. R. Kosaraju. Efficient tree pattern matching. In *Proc. 20th IEEE Foundations of Computer Science (FOCS)*, pages 178–183, 1989.

[33] S. Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999.

[34] W.Y. Lam, W. Ng, P.T. Wood, and M. Levene. XCQ: XML compression and querying system. In *Proc. 12th International World Wide Web Conference (WWW)*, 2003.

[35] H. Liefke and D. Suciu. XMill: an efficient compressor for xml data. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 153–164, 2000.

[36] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 3887, pages 703–714, 2006.

[37] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[38] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. 3rd International Conference on Database Theory*, pages 277–295, 1999.

[39] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. Xpress: A queriable compression for XML data. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 122–133, 2003.

[40] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[41] I. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP '03)*, pages 345–356. Springer-Verlag LNCS n. 2719, 2003.

[42] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. of the 38th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[43] I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Computing*, 31:762–776, 2001.

[44] I. Munro and S.S. Rao. Succinct representations of functions. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 3142, pages 1006–1015, 2004.

[45] G. Navarro and V. Mäkinen. Compressed full text indexes. *ACM Computing Surveys*. To Appear.

[46] R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[47] P.R. Raw and B. Moon. PRIX: Indexing and querying XML using Prüfer sequences. In *Proc. 20th International Conference on Data Engineering (ICDE)*, pages 288–300, 2004.

[48] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithm (SODA)*, pages 1230–1239, 2006.

[49] P. M. Tolani and J. R. Haritsa. XGRIND: A query-friendly XML compressor. In *Proc. 18th International Conference on Data Engineering (ICDE)*, pages 225–234, 2002.

[50] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: a dynamic index methd for querying XML data by tree structures. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 110–121, 2003.

[51] W. Wang, H. Wang, H. Lu, H. Jang, X. Lin, and J. Li. Efficient processing of XML path queries using the disk-based F&B index. In *Proc. 31st International Conference on Very Large Data Bases (VLDB)*, pages 145–156, 2005.

[52] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[53] Z. Ye and T. Berger. *Information measures for discrete random fields*. Science Press, 1998.