Structuring labeled trees for optimal succinctness, and beyond[*]

Paolo Ferragina
University of Pisa
ferragina@di.unipi.it

Fabrizio Luccio
University of Pisa
luccio@di.unipi.it

Giovanni Manzini
University of Piemonte Orientale
manzini@mfn.unipmn.it

S. Muthukrishnan
Rutgers University
muthu@cs.rutgers.edu

Abstract

Consider an ordered, static tree $\mathcal{T}$ on t nodes where each node has a label from alphabet set Σ. Tree $\mathcal{T}$ may be of arbitrary degree and of arbitrary shape. Say, we wish to support basic navigational *operations such as find the parent of node u, the ith child of u, and any child of u with label α.*

In a seminal work over fifteen years ago, Jacobson [15] observed that pointer-based tree representations are wasteful in space and introduced the notion of succinct data structures. *He studied the special case of* unlabeled *trees and presented a succinct data structure of $2t + o(t)$ bits supporting navigational operations in $O(1)$ time. The space used is asymptotically optimal with the information-theoretic lower bound averaged over all trees. This led to a slew of results on succinct data structures for arrays, trees, strings and multisets. Still, for the fundamental problem of structuring* labeled *trees succinctly, few results, if any, exist even though labeled trees arise frequently in practice, e.g. in the data as in markup text (XML) or in augmented data structures.*

We present a novel approach to the problem of succinct manipulation of labeled trees by designing what we call the xbw *transform of the tree, in the spirit of the well-known Burrows-Wheeler transform for strings.* xbw *transform uses path-sorting and grouping 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 (i) derive the first-known (near-)optimal results for succinct representation of labeled trees with $O(1)$ time for navigation operations, (ii) optimally support the powerful* subpath search *operation for the first time, and (iii) introduce a notion of tree entropy and present linear time algorithms for compressing a given labeled tree up to its entropy* beyond *the information-theoretic lower bound averaged over all tree inputs.*

Our xbw *transform is simple and likely to spur new results in the theory of tree compression and indexing, and may have some practical impact in XML data processing.*

1 Introduction

Consider a rooted, ordered, static tree data structure $\mathcal{T}$ on t nodes where each node u has a label in the 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 of arbitrary shape. We wish to support the basic *navigational* operations such as find the parent of u (denoted $\mathtt{parent}(u)$), the ith child of u (denoted $\mathtt{child}(u, i)$) and any child of u with label α (denoted $\mathtt{child}(u, \alpha)$).[1] The elementary solution is to represent the tree using a mixture of pointers and arrays using a total of $O(t)$ RAM words each of size $O(\log t)$; this trivially supports each of the navigation operations in $O(1)$ time taking a total of $O(t \log t)$ bits.

In a seminal work over fifteen years ago, Jacobson [15] observed that these pointer-based tree representations are wasteful in space and introduced the notion of *succinct data structures*, that is, data structures that use space close to their information-theoretic lower bound and yet support various operations efficiently (thus, succinct data structures are distinct from simply compressing the input to be uncompressed later). [15] initiated this area of research with the special case of *unlabeled* trees, that is, considering the structure of the tree but not the 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. [15] presented a storage scheme in $2t + o(t)$ bits while supporting the navigation 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 [21] extended the results with more efficient as well as a richer set of operations, including subtree size queries. Since then, a slew of results have fur-

[*]Partially supported by NSF DMS 0354600 and by the Italian MIUR projects: Algo-Next, ECD, Grid.it and "Piattaforme abilitanti complesse ad alte prestazioni".

[1]These operations return a NIL value when the output is not defined.

ther generalized these methods to trees with higher degrees [1] and ever richer sets of operations such as level-ancestor queries [10]. Succinct representations have been invented for other data structures including arrays, dictionaries, strings, graphs and multisets.

Despite this flurry of activity, the fundamental problem of structuring *labeled* trees succintly[2] has remained unsolved. Classical applications of trees in Computer Science, be they for representing data or computation, typically generate navigation problems on labeled trees. This includes applications of tries [9], dictionaries [32], parse trees [25], suffix trees [31] and pixel trees [6] as well as trees used in compiler intermediate representations [3, 16, 29], execution traces [14], and mathematical proofs [24, 4]. In modern setting, XML is a tree representation of data where each node has string labels [34]; it has become the *de facto* format for data storage, integration, and is beginning to have native database implementation. At the core, managing such XML databases crucially needs succinct data structures for labeled trees with large Σ. Thus the labeled tree case is significantly better motivated than the unlabeled case.

The information-theoretic lower bound for storing labeled trees is easily seen to be $2t + t\log|\Sigma|$, where the first term follows from the structure of the tree and the second from the labels. A trivial solution is to replicate the known structures for the unlabeled case $|\Sigma|$ times. This can be somewhat improved by a suitable divide-and-conquer approach [10] to derive a succinct representation for labeled trees that uses $2t+t\log|\Sigma|+O(t|\Sigma|\frac{\log\log\log t}{\log\log t})$ bits of storage and supports navigational operations in $O(1)$ time. This is far from the optimal for even moderately large Σ since the $O()$ 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 structuring trees to succinctness. Jacobson's [15] reduced the problem of succinct structuring 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 structuring labeled trees without pointers. Our approach not only gives us the first-known (near-)optimal results for succinct representation of labeled trees with $O(1)$ time for navigation operations, but also helps us to optimally support more powerful operations and explore the fundamental notion of succinctness at a deeper level of entropy of the input rather than the information-theoretic notion which takes the worst case over all input instances. More specifically, our contributions are as follows.

1. We introduce a new *transform* of $\mathcal{T}$ denoted $\mathsf{xbw}(\mathcal{T})$. It has optimal size of $2t + t\log|\Sigma|$ bits. We show how to transform $\mathcal{T}$ to $\mathsf{xbw}(\mathcal{T})$ as well as invert $\mathsf{xbw}(\mathcal{T})$ to $\mathcal{T}$ in optimal $O(t)$ time. This transform captures the structural properties of $\mathcal{T}$ and is the basis for all our results.

2. Using xbw, we present a succinct data structure for labeled trees that uses at most $2t\log|\Sigma| + O(t)$ bits and supports all navigational queries such as— $\mathtt{parent}(u)$, $\mathtt{child}(u,i)$, and $\mathtt{child}(u,\alpha)$—in optimal $O(1)$ time. Note that the space used is nearly optimal for any Σ, being at most twice the information-theoretic lower bound of $t\log|\Sigma| + 2t$.

3. We introduce a new, powerful operation: *subpath query*. Given a path p of labels from Σ, the subpath query returns all nodes $u \in \mathcal{T}$ such that there exists a path leading to u labeled by p. Note that u and the origin of p could be internal to $\mathcal{T}$. Subpath query is of central interest to the XPATH query language in XML [35] and is also used as a basic block for supporting more sophisticated path searches [11]. Still, no prior algorithmic result is known for supporting subpath queries on trees represented succinctly.

   We present a succinct data structure for labeled trees based on the xbw transform using optimal (up to lower order terms) $tH_0(\mathcal{S}_\alpha) + 2t + o(t)$ bits and supporting subpath queries in time $O(|p|\log|\Sigma|)$ where $|p|$ is the length of the path. This data structure also supports navigational queries in $O(\log|\Sigma|)$ time. If $|\Sigma| = O(\text{polylog}(t))$, the subpath query takes optimal $O(|p|)$ time, and navigational queries take optimal $O(1)$ time.

Our results above on succinct structuring of labeled trees are the first known (near-)optimal bounds, independent of Σ and the structure of $\mathcal{T}$.

We proceed *beyond* succinctness and initiate the study of data structuring labeled trees using bits proportional in number to the inherent entropy of the tree as well as the labels. For a string of symbols, the notion of entropy is well developed, understood, and exploited in indexing and compression [7, 13]: high-order entropy depends on frequency of occurrences of substrings of length k. For trees, there is information or entropy in the labels as well as in the subtree

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

structure in the neighborhood of each node. The simplest approach to labeled tree compression is to serialize it using say pre or postorder traversal to get a sequence of labels and apply any known compressor such as `gzip`. This is fast and is used in practice, but it does not capture the entropy contained in the tree structure. Another approach is to identify repeated substructures in the tree and collapse them a la Lempel-Ziv methods for strings. Such methods have been used for grammar-based compression, but there is no provable information-theoretic analysis of the compression obtained. A more formal approach is to assume a *parent-child model* of tree generation in which node labels are generated according to the labels of their parents, or more generally, by a descendant or ancestor subtree of height k, for some k [3, 30, 27]. These statistical models are related to Markov random fields over trees [28, 33]. From these approaches heuristic algorithms for tree compression have been derived and validated only experimentally. In XML compression, well-known softwares like XMILL and XMLPPM [19, 5], group data according to the labeling path leading to them and then use specialized compressors ideal for each group. The length k of the predictive paths may be selected manually [19] or automatically [5] by following the classical `PPM` paradigm [32]. Even for these simplified approaches, no formal analysis of achievable compression is known. Despite work on labeled tree compression methods in multiple applied areas, there is no theory of achievable compression with associated lower and upper bounds for trees.

We intend to initiate here a formal analysis of labeled tree compression. Based on the standard *parent-child model* of tree generation we define the *kth order empirical entropy* $H_k(\mathcal{T})$ of a labeled tree $\mathcal{T}$ that mimics on trees, in a natural way, the well-known definition of kth order empirical entropy over strings. Further, based on our xbw transform, we present an algorithm to compress (and uncompress) $\mathcal{T}$ in $O(t)$ time, getting a representation with at most $tH_k(\mathcal{T}) + 2.01t + o(t)$ bits. This is off from the lower bound of representing just the tree without labels (roughly $2t$) by a factor that mainly depends on the entropy of the instance and may be significantly smaller than the $O(\log|\Sigma|)$ term that is the worst case over all inputs. While such results have been previously known for string compression, our work is the first theory of entropy-based compression for trees. Again, these results rely on our xbw transform.

Finally, in addition to their theoretical significance, our results are of immediate relevance to practical XML processing systems. Our xbw transform yields a provable succinct storage of XML data that supports searching and navigation queries in optimal time, something not achievable by existing XML compress/search engine.[3] We have encouraging, preliminary experimental experience compared to known XML compression and searching tools, but this submission will only focus on our theoretical results.

1.2 Overview of our techniques

Jacobson's seminal work [15] reduced the problem of succinctly structuring unlabeled trees to that of arrays and balanced parentheses. Since then, the algorithmic results have become quite technical, involving stratifications, tree partitioning, etc. Extending these techniques to labeled trees and the powerful subpath query would have entailed making the algorithms even more complicated. Further, these techniques do not directly lead beyond succinctness to entropy-bounded data structures.

Instead, we take a different approach. In particular, we were inspired by the elegant and surprising Burrows-Wheeler transform (BWT) for strings [2]. BWT transforms the string into a permutation, nontrivially derived from sorting the suffixes. Similar substrings get grouped together in this transform. In the past few years, the BWT has been the unifying tool for string compression and indexing, producing many important breakthroughs, eg., [20, 8, 7, 13].

The centerpiece of our technical contribution is the new xbw transform we design that, in the spirit of BWT, relies on *path* sorting and grouping to linearize the labeled tree $\mathcal{T}$ into *two* coordinated arrays, one capturing the structure and the other the labels. Using the properties of xbw transform, we reduce succinct data structure problems on labeled trees to certain other succinct data structure problems on arrays. This is the first such transform known for trees. Standard tree traversals (without path sorting) will not have the structural properties of this transform we use to derive all our results. Also, (de)constructing the transform and reductions need new algorithms that we design. As a result, we get both entropy-bounded as well as succinctness results in a unified framework for structuring labeled trees. As an aside, our xbw transform-based approach is extremely simple, is likely to be useful in practice and connects known succinct data structures on arrays to useful problems on XML data processing. We believe that our xbw transform is likely to spur new results in tree compression and indexing.

Finally, a different perspective may be useful. The natural approach to supporting subpath queries is to index all node-to-root paths, ie., build a *suffix tree of the tree* $\mathcal{T}$ defined to be the trie of all paths of $\mathcal{T}$ starting at all nodes and

[3]"XML Bloat" refers to the increase in file size when data is converted to XML. Pointer-based representation of XML documents makes it space-inefficient. This is a serious concern and has led to the development of many XML compressors like ZXML, XMILL, XGRIND, XMLPPM, XPRESS, MILLAU, DBXML, XMLZIP, etc., all of which address the issue of succinct structuring of labeled trees, albeit heuristically, by reducing the problem to using GZIP after dictionary substitution methods. These compressors have no proven guarantees on compression or running time for navigation operations. We do not discuss the specifics of this application further. There are many surveys on the web including www.cs.uiowa.edu/˜rlawrenc/research/Students/SN_04_XMLCompress.pdf.

ending into the root. This concept has found uses in tree matching [18]. A pointer-based representation of the suffix tree of $\mathcal{T}$ is simple to build, but wasteful. Our xbw transform may be thought of as the compressed representation for the suffix tree of the tree. There have been a number of recent results on succinctly representing the suffix tree of a string [7, 12, 22, 13], but the structural properties of the string are crucially used in building, inverting and searching the suffix tree. For managing arbitrary trees with labels (not just suffix tree of a string) or going beyond, managing the suffix tree of such arbitrary trees as we do, new approaches and algorithms were needed.

The paper is organized as follows. In Section 2, we introduce our xbw transform and describe its structural and optimality properties. In Section 3, we present $O(t)$ time algorithms to convert $\mathcal{T}$ into xbw($\mathcal{T}$) and vice versa. In Section 4, we present our (near-)optimal succinct data structure using xbw transform that supports navigational operations in $O(1)$ time. In Section 5, we present our results for optimal subpath queries. In Section 6, we present results on tree entropy and compression.

2 The xbw transform for labeled trees

Let $\mathcal{T}$ be an ordered tree of arbitrary fan-out, depth and shape. We assume $\mathcal{T}$ has n internal nodes labeled with symbols drawn from set $\Sigma_N = \{A, B, C, \ldots\}$.[4] We also assume that $\mathcal{T}$ has ℓ leaves labeled with symbols drawn from set $\Sigma_L = \{a, b, c, \ldots\}$. We assume that $\Sigma_N \cap \Sigma_L = \emptyset$, so that internal nodes can be distinguished from leaves directly from their labels, and we set $\Sigma = \Sigma_N \cup \Sigma_L$. For the sake of presentation we assume that Σ is the set of labels effectively used in $\mathcal{T}$'s nodes, and that these symbols are encoded with the integers in the range $[1, |\Sigma|]$ (Σ_N's symbols first). This means that a preliminary bucket sorting step may be needed to compute this labeling. The overall size of $\mathcal{T}$ is $t = n + \ell$ nodes. For each node u, we let $\alpha[u] \in \Sigma$ denote the label of u, and $\pi[u]$ the string obtained by concatenating the symbols on the *upward path* from u's parent to the root of $\mathcal{T}$. Notice that $\pi[u]$ is formed by labels of internal nodes only. Since sibling nodes may be labeled with the same symbol, many nodes in $\mathcal{T}$ may have the same π-string (see Fig. 1).

To define the xbw transform we build a sorted multiset $\mathcal{S}$ consisting of t *triplets*, one for each tree node. To build $\mathcal{S}$ we first visit $\mathcal{T}$ in pre-order and, for each visited node u, we insert the triplet $s[u] = \langle \mathsf{last}[u], \alpha[u], \pi[u] \rangle$ in $\mathcal{S}$, where $\mathsf{last}[u]$ is a binary flag set to 1 iff u is the last child of its parent in $\mathcal{T}$. Then, we stably sort $\mathcal{S}$ lexicographically according to the π-component of its triplets. Note that the same triplet may appear more than once because sibling nodes may have the same label (see Fig. 1). Hence, the stability of the sorting procedure is needed to preserve the identity of triplets after the sorting step. 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 $\mathcal{S}$. See Fig. 1 for a running example.

[4] In some applications (e.g. XML data) the labels on nodes may be strings. We can assume that such strings have been renamed with atomic symbols.

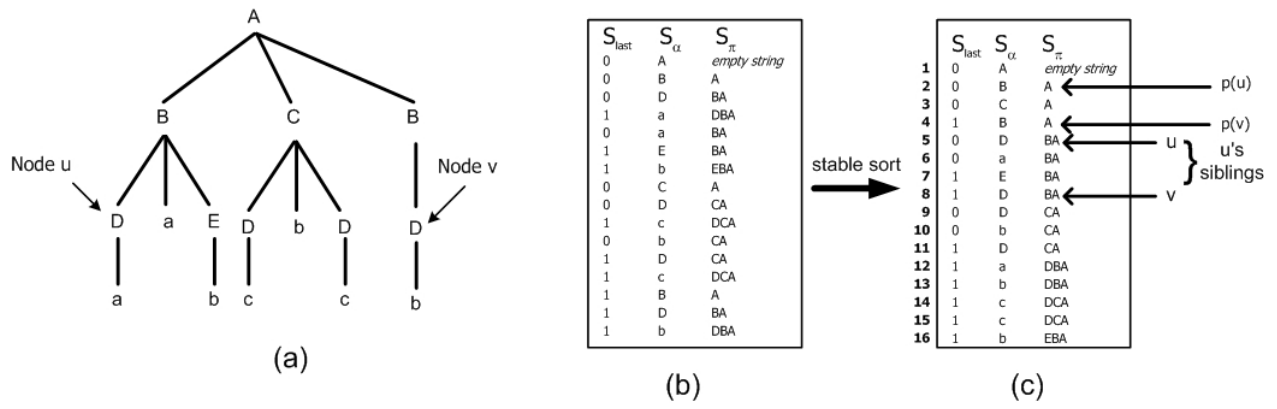
The sorted set $\mathcal{S}[1, t]$ satisfies the following "compositional" properties: $\mathcal{S}_{\mathsf{last}}$ has n bits set to 1 (one for each internal node); the other $t - n$ bits are set to 0; $\mathcal{S}_\alpha$ contains all the labels of the nodes of $\mathcal{T}$; $\mathcal{S}_\pi$ contains all the upward labeled paths of $\mathcal{T}$. Each path is repeated a number of times equal to the number of its offsprings. Thus $\mathcal{S}_\alpha$ is a lossless serialization of the labels of $\mathcal{T}$ whereas $\mathcal{S}_{\mathsf{last}}$ provides information on the grouping of the children of $\mathcal{T}$'s nodes. The structural properties of $\mathcal{T}$'s can be inferred from $\mathcal{S}$'s sorting:

1. The first triplet of $\mathcal{S}$ refers to the root of $\mathcal{T}$.
2. Let u' and u'' be two nodes of $\mathcal{T}$ such that $\pi[u'] = \pi[u'']$. Nodes u' and u'' have the same depth in $\mathcal{T}$, and u' is to the left of u'' iff the triplet $s[u']$ precedes the triplet $s[u'']$ in $\mathcal{S}$.
3. Let $u_1, \ldots, u_c$ be the children of a node u in $\mathcal{T}$. The triplets $s[u_1], \ldots, s[u_c]$ lie contiguously in $\mathcal{S}$ following this order. Moreover, the last triplet $s[u_c]$ has its last-component set to 1, whereas all the other triplets have their last-component set to 0.
4. Let v_1, v_2 denote two nodes of $\mathcal{T}$ such that $\alpha[v_1] = \alpha[v_2]$. If $s[v_1]$ precedes $s[v_2]$ in $\mathcal{S}$, then the children of v_1 precede the children of v_2 in $\mathcal{S}$.

Example 1 In Fig. 1 we have two nodes labeled B, whose upward path is A. These nodes have a total of $q = 4$ children which are stored in the subarray $\mathcal{S}[5, 8]$. The bits $\mathcal{S}_{\mathsf{last}}[7] = \mathcal{S}_{\mathsf{last}}[8] = 1$ separate the three children of u from the unique child of v. ∎

Definition 1 *Given a labeled tree $\mathcal{T}$ we denote by* xbw*($\mathcal{T}$) the pair $\langle \mathcal{S}_{\mathsf{last}}, \mathcal{S}_\alpha \rangle$.* ∎

Recall that $\mathcal{S}_{\mathsf{last}}$ is a binary string of length t and $\mathcal{S}_\alpha$ is a permutation of the t labels associated to the nodes of $\mathcal{T}$. Later we show that from $\mathcal{S}_{\mathsf{last}}$ and $\mathcal{S}_\alpha$ we can recover $\mathcal{T}$, hence xbw is an invertible transform. Our transform takes $t \log |\Sigma|$ bits for $\mathcal{S}_\alpha$, plus t bits for $\mathcal{S}_{\mathsf{last}}$. If internal nodes and leaves are labeled using one unique alphabet (ie. $\Sigma_N = \Sigma_L$), we need an additional bit array of length t to distinguish between leaves and internal nodes in $\mathcal{S}_\alpha$. The overall space is then $2t + t \log |\Sigma|$. This is optimal up to lower order terms since the information theoretic lower bound on the space for representing a t-node ordinal tree is $2t - O(\log t)$ bits [15], and the term $t \log |\Sigma|$ is the optimal cost of representing the labels in the worst case.



(b)

S_{last}	S_α	S_π
0	A	*empty string*
0	B	A
0	D	BA
1	a	DBA
0	a	BA
1	E	BA
1	b	EBA
0	C	A
0	D	CA
1	c	DCA
0	b	CA
1	D	CA
1	c	DCA
1	B	A
1	D	BA
1	b	DBA

(c)

	S_{last}	S_α	S_π
1	0	A	*empty string*
2	0	B	A
3	0	C	A
4	1	B	A
5	0	D	BA
6	0	a	BA
7	1	E	BA
8	1	D	BA
9	0	D	CA
10	0	b	CA
11	1	D	CA
12	1	a	DBA
13	1	b	DBA
14	1	c	DCA
15	1	c	DCA
16	1	b	EBA

Figure 1. (a) A labeled tree $\mathcal{T}$. Notice that $\alpha[u] = \alpha[v] = D$ and $\pi[u] = \pi[v] = BA$. (b) The multiset $\mathcal{S}$ as a result of the pre-order visit of $\mathcal{T}$. (c) $\mathcal{S}$ after the stable lexicographic sort executed on the π's component of its triplets. $\mathcal{S}_{\mathsf{last}}$ marks the triplets corresponding to nodes which are rightmost children.

Hereafter we continue to assume that leaves and internal nodes are labeled with two distinct alphabets since this simplifies the algorithms. The space cost of the additional bit array does not asymptotically influence the final bounds.

3 Converting $\mathcal{T}$ to $\mathsf{xbw}(\mathcal{T})$ and vice versa

First we will focus on converting $\mathcal{T}$ to $\mathsf{xbw}(\mathcal{T})$; this task is simpler than going the other way. For the computation of $\mathsf{xbw}(\mathcal{T})$, explicitly building $\mathcal{S}$ would require too much space and time. In the case of a degenerate tree consisting of a single path of t nodes, the overall size of $\mathcal{S}_\pi$ would be $\Theta(t^2)$. The construction of $\mathcal{S}$ must be therefore not explicit. In this section we describe an algorithm which runs in optimal $O(t)$ time and uses $O(t \log t)$ bits of space. The algorithm, summarized in Fig. 2, sorts the π-components using a straightforward generalization of the *skew algorithm* for suffix array construction [17]. The only non-trivial step of this algorithm is the recursion (Step 3) in which we sort the paths starting at nodes at levels $\not\equiv j \pmod 3$. The parameter j is chosen in such a way that the number of nodes being at level $\equiv j \pmod 3$ is at least $t/3$.[5] We use radix sort to assign a lexicographic name to each path according to its first three symbols (obtained using the parent pointers). Then we build a new "contracted" tree whose labels are the names previously assigned. Because of the choice of j, the new tree will have at most $2t/3$ nodes. Note that the new tree will have a larger fan-out than the original one, but this does not affect the algorithm that only uses parent pointers. The correctness of our algorithm can be easily proved. The running time satisfies the recurrence $T(t) = T(2t/3) + \Theta(t)$ and is therefore $\Theta(t)$. Once the π-components have been sorted, the construction of $\mathcal{S}_\alpha$ and $\mathcal{S}_{\mathsf{last}}$ is trivial.

```
Algorithm PathSort($\mathcal{T}$)
 1. Create the array IntNodes$[1, n]$ initially empty.
 2. Visit the internal nodes of $\mathcal{T}$ in preorder. Let $u$ denote the
    $i$th 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 $n/3$. Sort recursively
    the upwards paths starting at nodes at levels $\not\equiv j \pmod 3$.
 4. Sort the upward paths starting at nodes at levels $\equiv j \bmod 3$
    using the result of Step 3.
 5. Merge the two sets of sorted paths.
```

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

Next we show how to reconstruct the tree $\mathcal{T}$ given $\mathsf{xbw}(\mathcal{T}) = \langle \mathcal{S}_{\mathsf{last}}, \mathcal{S}_\alpha \rangle$ in $O(t)$ time. Our algorithm works in three phases. In the first phase we build an array $\mathsf{F}[1, |\Sigma_N|]$ which approximates $\mathcal{S}_\pi$ at its first symbol. For every internal-node label $x \in \Sigma_N$, $\mathsf{F}[x]$ stores the position in $\mathcal{S}$ of the first triplet whose π-component is prefixed by x. For the example in Fig. 1 we would have $\mathsf{F}[\mathtt{B}] = 5$, since the 5-th triplet in $\mathcal{S}$ is the first one having its π-component prefixed by $\mathtt{B}$. In the second phase we build an array $\mathsf{J}[1, t]$ which allows us to "jump" in $\mathcal{S}$ from any node to its first child. We set $\mathsf{J}[i] = j$ if $\mathcal{S}[i]$ is an internal node and $\mathcal{S}[j]$ is the first child of $\mathcal{S}[i]$. If $\mathcal{S}[i]$ is a leaf we set $\mathsf{J}[i] = -1$. For the example in Fig 1 we would have $\mathsf{J}[5] = 12$, since node u has corresponding triplet $\mathcal{S}[5]$ and first (and unique) child

[5]In the original Skew Algorithm, 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.

represented by $\mathcal{S}[12]$. The third phase recovers the original tree $\mathcal{T}$ using the array J.

We now give details of the three phases. Algorithm BuildF in Fig. 3 computes the array $\mathsf{F}[1, |\Sigma_N|]$ in $O(t)$ time as follows. It first initializes an array C so that $\mathsf{C}[i]$ stores the number of occurrences of symbol i in $\mathcal{S}_\alpha$. Then it computes inductively $\mathsf{F}[i+1]$ given $\mathsf{F}[i]$. The base case is obvious: Step 2 sets $\mathsf{F}[1] = 2$ since $\mathcal{S}_\pi[1]$ is the empty string, which occurs only once in $\mathcal{S}_\pi$, and $\mathcal{S}_\pi[2]$ is therefore prefixed by symbol 1. For the general case assume that $\mathsf{F}[i]$ is the position of the first entry in $\mathcal{S}_\pi$ prefixed by symbol i (step 4). We know that there are $\mathsf{C}[i]$ internal nodes labeled by i and that their children occur contiguously in $\mathcal{S}$ starting at position $\mathsf{F}[i]$. Hence, the last entry in $\mathcal{S}_\pi$ prefixed by i is the one corresponding to the $\mathsf{C}[i]$-th 1 in $\mathcal{S}_{\mathsf{last}}$ counting from position $\mathsf{F}[i]$. The loop in Steps 6–8 serves the purpose of counting $\mathsf{C}[i]$ 1's starting from $\mathcal{S}_{\mathsf{last}}[\mathsf{F}[i]]$ and therefore the value $\mathsf{F}[i+1]$ is correctly set at Step 9.

```
Algorithm BuildF(xbw($\mathcal{T}$))
 1. for $i = 1, \ldots, t$ do $\mathsf{C}[\mathcal{S}_\alpha[i]]$ += 1;
 2. $\mathsf{F}[1] = 2$;          { $\mathcal{S}_\pi[1]$ is the empty string }
 3. for $i = 1, \ldots, |\Sigma_N| - 1$ do
 4.    $j = \mathsf{F}[i]$;
 5.    $s = 0$;
 6.    while ($s \neq \mathsf{C}[i]$) do
 7.       if ($\mathcal{S}_{\mathsf{last}}[j++] == 1$)
 8.          $s$++;         { One set of siblings has passed}
 9.    $\mathsf{F}[i+1] = j$;
10. return F.
```

Figure 3. Algorithm to compute $\mathsf{F}[1, |\Sigma_N|]$ such that $\mathsf{F}[i] = j$ iff $\mathcal{S}_\pi[j]$ is the first entry of S prefixed by i.

In the second phase of the inversion procedure we compute the array $\mathsf{J}[1, t]$ in $O(t)$ time. We make use of the properties of xbw transform that the k-th occurrence of the symbol $j \in \Sigma_N$ in $\mathcal{S}_\alpha$ corresponds to the k-th group of siblings counting from position $\mathsf{F}[j]$. This allows us to compute J with a simple scan of the array $\mathcal{S}_\alpha$ (see pseudocode in Fig. 4). Indeed, to avoid the use of an additional auxiliary array, we update the entries of F by maintaining the invariant that, for every $j \in \Sigma_N$, $\mathsf{F}[j]$ is the position of the first child of the next occurrence of symbol j in $\mathcal{S}_\alpha$ (step 9).

Finally, in the third phase we recover $\mathcal{T}$ given $\mathcal{S}_\alpha$, $\mathcal{S}_{\mathsf{last}}$ and the array J. Of course there are several possible representations of the tree $\mathcal{T}$. In the case we wish to list the nodes of $\mathcal{T}$ in depth-first order, we can use a stack Q in which pop/push pairs $\langle i, u\rangle$ where i denotes the position of the node in $\mathcal{S}$ and u denotes the node label. For each popped node $\langle i, u\rangle$, we can use the arrays J and $\mathcal{S}_\alpha$ to locate its children (if any) and insert them in the stack in reverse order (details in the full paper). Summarizing,

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

Figure 4. Algorithm to compute $\mathsf{J}[1, t]$ such that $\mathsf{J}[i] = j$ if $\mathcal{S}[j]$ is the first child of $\mathcal{S}[i]$, and $\mathsf{J}[i] = -1$ if $\mathcal{S}[i]$ is a leaf.

Theorem 1 *Given a tree $\mathcal{T}$ consisting of t nodes labeled with symbols drawn from an alphabet Σ, we can compute* xbw*($\mathcal{T}$) in optimal $O(t)$ time. Given* xbw$(\mathcal{T})$ *we can reconstruct $\mathcal{T}$ in optimal $O(t)$ time.* ∎

4 Navigational Operations on xbw($\mathcal{T}$)

We show that using a succinct representation of $\mathcal{S}_\alpha$ and $\mathcal{S}_{\mathsf{last}}$—plus an additional array we will introduce shortly—we can efficiently navigate over $\mathcal{T}$ following node labels or the ranking of nodes among their siblings. An important ingredient of our solution is the use of rank and select operations over arbitrary sequences. Given a sequence $S[1, t]$ over an alphabet Σ, $\mathsf{rank}_c(S, q)$ is the number of times the symbol $c \in \Sigma$ appears in $S[1, q] = s_1 s_2 \ldots s_q$, and $\mathsf{select}_c(S, q)$ is the position of the q-th occurrence of the symbol $c \in \Sigma$ in S.

Lemma 2 *Let s denote a sequence over the alphabet Σ.*

1. *If $\Sigma = \{0, 1\}$, the data structure in [26, Theorem 3.3] supports $\mathsf{rank}_1(s, q)$ (when $s[q] = 1$) and select_1 queries in $O(1)$ time using $\log \binom{|s|}{m} + o(m) + O(\log\log|s|)$ bits, where m is the number of 1's in s.*
2. *For $|\Sigma| = O(\mathrm{polylog}(t))$, the generalized wavelet tree in [23] supports rank_c and select_c queries in $O(1)$ time using $|s|H_0(s) + o(|s|)$ bits of space, where $H_0(s)$ denotes the 0th order empirical entropy of the sequence s.*
3. *For general Σ, the wavelet tree in [13] supports rank_c and select_c queries in $O(\log|\Sigma|)$ time using $|s|H_0(s) + o(|s|)$ bits of space, where $H_0(s)$ denotes the 0th order empirical entropy of the sequence s.*

These data structures support the retrieval of s_i for any i in the same time as the rank *and* select *queries.* ∎

Based on the array F defined in Section 3, we define the binary array $\mathcal{A}[1,t]$ such that $\mathcal{A}[j]=1$ iff $j=\mathsf{F}[x]$ for some $x\in\Sigma_N$. In other words, the ones in $\mathcal{A}$ mark the positions j such that the first symbol of $\mathcal{S}_\pi[j]$ differs from the first symbol of $\mathcal{S}_\pi[j-1]$. Given an index i, $1\le i\le t$, the algorithm GetChildren in Fig. 5 computes the range of rows $(\mathsf{First},\mathsf{Last})$ such that $\mathcal{S}[\mathsf{First}],\mathcal{S}[\mathsf{First}+1],\ldots,\mathcal{S}[\mathsf{Last}]$ are the children of node $\mathcal{S}[i]$. The algorithm returns -1 if $\mathcal{S}[i]$ is a leaf. Let c denote the label of node $\mathcal{S}[i]$ (Step 2). We first compute $\mathsf{F}[c]$, the first position of $\mathcal{S}_\pi$ whose starting symbol is c (Step 3). Then we compute k, the number of occurrences of c in $\mathcal{S}_\alpha$ up to and including the position $\mathcal{S}_\alpha[i]$ (Step 4). By the properties of the xbw transform, the children of $\mathcal{S}[i]$ are the k-th group of siblings counting from position $\mathsf{F}[c]$. Steps 5–7 correctly compute the position of the first and last child of $\mathcal{S}[i]$.

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

Figure 5. Algorithm for computing the range $(\mathsf{First},\mathsf{Last})$ such that $\mathcal{S}[\mathsf{First}],\mathcal{S}[\mathsf{First}+1],\ldots,\mathcal{S}[\mathsf{Last}]$ are the children of node $\mathcal{S}[i]$. The algorithm returns -1 if $\mathcal{S}[i]$ is a leaf of $\mathcal{T}$.

Example 2 Consider Fig. 1 and pick the node $p(v)$, the parent of v associated with the triplet $\mathcal{S}[4]$. We have $i=4$, $c=\mathtt{B}$, and $y=5$ since $\mathcal{S}[5]$ is the first triplet having π-component that starts with B. We also have $k=2$ and $z=1$. Hence, the children of $p(v)$ are the second group of children counting from $\mathcal{S}[5]$. The algorithm reports correctly $\mathsf{First}=\mathsf{select}_1(\mathcal{S}_{\mathsf{last}},2)+1=8$ and $\mathsf{Last}=\mathsf{select}_1(\mathcal{S}_{\mathsf{last}},3)=8$. Thus, $p(v)$ has a single child (i.e. v) located at $\mathcal{S}[8]$. ■

Given an index i, $1\le i\le t$, the algorithm GetParent in Fig. 6 computes the index p such that $\mathcal{S}[p]$ is the parent node of $\mathcal{S}[i]$. The algorithm returns -1 if $i=1$, since $\mathcal{S}[1]$ is the root of $\mathcal{T}$ (Step 1). Otherwise, it first finds the symbol c associated with the parent of $\mathcal{S}[i]$. This symbol is $\mathcal{S}_\pi[i]$ and we retrieve it by means of array $\mathcal{A}$ (Step 2). In Steps 3–4 we compute the number k of groups of siblings from position $\mathsf{F}[y]$ to position i. By the properties of xbw transform we know that the index p of i's parent is the position of the $(k+1)$-th occurrence of symbol c in $\mathcal{S}_\alpha$ (Step 5).

```
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 6. Algorithm for computing the index p such that $\mathcal{S}[p]$ is the parent of $\mathcal{S}[i]$. The algorithm returns -1 if $\mathcal{S}[i]$ is the root of $\mathcal{T}$.

Example 3 In Fig. 1 consider the node v associated with the triplet $\mathcal{S}[8]$. We have $c=2$ (symbol B), $y=5$, $k=1$, thus we correctly compute $p=\mathsf{select}_{\mathtt{B}}(\mathcal{S}_\alpha,2)=5$. ■

There is a technical detail to achieve constant time rank and select queries over $\mathcal{S}_\alpha$. Recall that Σ_N is mapped onto the range $[1,|\Sigma_N|]$. Let $\mathcal{B}$ be a binary sequence of length $|\Sigma_N|t$ such that, for $c\in\Sigma_N$, we have $\mathcal{B}[t(c-1)+i]=1$ iff $\mathcal{S}_\alpha[i]=c$. In other words, $\mathcal{B}$ consists of $|\Sigma_N|$ segments of length t such that the x-th segment is a bitmap for the occurrences of the symbol x in $\mathcal{S}_\alpha$. Let occ be an array of size n storing in $\mathsf{occ}[c]$ the number of occurrences in $\mathcal{S}_\alpha$ of symbols smaller than $c\in\Sigma_N$. It is easy [12] to verify that for $c\in\Sigma_N$

$$\begin{aligned}\mathsf{rank}_c(\mathcal{S}_\alpha,i) &= \mathsf{rank}_1(\mathcal{B},c|\Sigma_N|+i)-\mathsf{occ}[c];\\ \mathsf{select}_c(\mathcal{S}_\alpha,i) &= \mathsf{select}_1(\mathcal{B},\mathsf{occ}[c]+i)-c|\Sigma_N|.\end{aligned}$$

Using rank_1 and select_1 operations over $\mathcal{B}$ we can answer in $O(1)$ time rank_c and select_c queries over $\mathcal{S}_\alpha$ for any symbol $c\in\Sigma_N$ (note that algorithms GetChildren and GetParent do not need rank_c and select_c operations for symbols c associated to leaves). Consequently, using succinct representation for arrays $\mathcal{S}_{\mathsf{last}}$, $\mathcal{A}$, $\mathcal{B}$, and occ we can conclude:

Theorem 3 *For any alphabet Σ, there exists a succinct representation for $\mathsf{xbw}(\mathcal{T})$ that takes at most $2t\log(|\Sigma|)+O(t)$ bits and supports* `parent(u)`, `child(u,i)` *and* `child(u,α)` *queries in $O(1)$ time. We can also recover $\mathcal{T}$ in $O(t)$ time.*

Proof: The only non trivial point is the space bound. We notice that $\mathcal{B}$ has length $t|\Sigma_N|$ and contains exactly n ones since n is the number of internal nodes of $\mathcal{T}$. By Lemma 2 (item 1) the succinct representation of $\mathcal{B}$ takes $\log\binom{|\Sigma_N|t}{n}+o(n)+O(\log\log(t|\Sigma_N|))$ bits. Using elementary calculus, this is upper bounded by $t\log|\Sigma_N|+O(t)$. We note that the rank_1 query over $\mathcal{B}$ are always executed on bits set to 1.

We represent the values in array occ in unary and then build the data structure of Lemma 2 (item 3) with $\Sigma=$

```
Algorithm $\mathsf{SubPathSearch}(q_1 q_2 \cdots q_k)$
 1. $i = k$;
 2. $\mathsf{First} = \mathsf{F}(q_k)$; $\mathsf{Last} = \mathsf{F}(q_k + 1) - 1$;
 3. while $(\mathsf{First} \leq \mathsf{Last})$ and $(i \geq 2)$ do
 4.    $i = i - 1$;
 5.    $z = \mathsf{rank}_1(\mathcal{S}_{\mathsf{last}}, \mathsf{F}(q_i) - 1)$;          { $z$ is # of nodes with label smaller than $q_i$ }
 6.    $k_1 = \mathsf{rank}_{q_i}(\mathcal{S}_\alpha, \mathsf{First} - 1)$;            { # occurrences of $q_i$ in $\mathcal{S}_\alpha[1, \mathsf{First} - 1]$ }
 7.    $\mathsf{First} = \mathsf{select}_1(\mathcal{S}_{\mathsf{last}}, z+k_1-1)+1$;   {starting position of the $(z + k_1)$-th group of siblings }
 8.    $k_2 = \mathsf{rank}_{q_i}(\mathcal{S}_\alpha, \mathsf{Last})$;                { # of occurrences of $q_i$ in $\mathcal{S}_\alpha[1, \mathsf{Last}]$ }
 9.    $\mathsf{Last} = \mathsf{select}_1(\mathcal{S}_{\mathsf{last}}, z + k_2)$;      { ending position of the $(z + k_2)$-th group of siblings }
10. if $(\mathsf{First} > \mathsf{Last})$ then
11.    return "no path prefixed by $q_1 q_2 \cdots q_k$";
12. else return $(\mathsf{First}, \mathsf{Last})$.
```

Figure 7. Compute $\mathcal{S}[a, b]$ as the set of triplets representing the parents of nodes with paths prefixed by $q_1 ... q_k$.

$\{0, 1\}$. It is easy to see that $\mathsf{occ}[c]$ can be computed in constant time by means of rank_1 and select_1 operations over the unary sequence. The theorem follows by observing that $\mathcal{S}_\alpha$ takes $t \log(|\Sigma|)$ bits and the succinct representation of $\mathcal{A}$, $\mathcal{S}_{\mathsf{last}}$ and occ takes $O(t)$ bits. ∎

To achieve entropy bounds for the storage of $\mathcal{S}_\alpha$ we can resort to the (generalized) wavelet tree of Lemma 2 (items 2 and 3) for implementing the rank and select operations on $\mathcal{S}_\alpha$. In this case, the arrays $\mathcal{B}$ and occ are no longer needed.

Theorem 4 *For any alphabet Σ, there exists a succinct representation for* $\mathsf{xbw}(\mathcal{T})$ *that takes at most $tH_0(\mathcal{S}_\alpha) + 2t + o(t)$ bits and supports* `parent`(u), `child`(u, i) *and* `child`(u, α) *queries in $O(\log |\Sigma|)$ time. If $|\Sigma| = O(\text{polylog}(t))$, navigational queries take $O(1)$ time.* ∎

Since $H_0(\mathcal{S}_\alpha) \leq \log |\Sigma|$, we note that for polylogarithmic alphabets Theorem 4 improves Theorem 3.

5 Subpath Search in $\mathsf{xbw}(\mathcal{T})$

We now consider the subpath search problem of finding the nodes u of $\mathcal{T}$ whose *upward* path, i.e., the path obtained by concatenating the labels from u to the root, is prefixed by the string $\beta = q_1 \dots q_k$, with $q_i \in \Sigma_N$. (Subpath queries with downward paths can be handled by reversing the path in the query.) Our solution also implicitly counts the number of subpaths matching β in $\mathcal{T}$. Because of the sorting of $\mathcal{S}$, the triplets corresponding to the nodes in the output of the subpath query are contiguous in $\mathcal{S}$. We denote their range with $\mathcal{S}[\mathsf{First}, \mathsf{Last}]$, so that $\mathcal{S}_\pi[\mathsf{First}, \mathsf{Last}]$ are exactly the rows of $\mathcal{S}_\pi$ prefixed by β.

Algorithm $\mathsf{SubPathSearch}$ in Fig. 7 computes the range $[\mathsf{First}, \mathsf{Last}]$ in k phases numbered from k to 1. Each phase preserves the following invariant: *At the end of the i-th phase the parameter* First *points to the first row of $\mathcal{S}$ such that $\mathcal{S}_\pi[\mathsf{First}]$ is prefixed by $q_i \cdots q_k$ and the parameter* Last *points to the last row of $\mathcal{S}$ such that $\mathcal{S}_\pi[\mathsf{Last}]$ is prefixed by $q_i \cdots q_k$.* For $i = k$ we determine First and Last via the array F such that $\mathsf{F}[c]$ is the first position of $\mathcal{S}_\pi$ prefixed by the symbol c. The inductive step assumes that $[\mathsf{First}, \mathsf{Last}]$ is the range of rows whose π-component is prefixed by $q_{i+1} \cdots q_k$, and then computes the range of rows $[\mathsf{First}', \mathsf{Last}']$ whose π-component is prefixed by $q_i q_{i+1} \cdots q_k$ as follows. Let k_1 (resp. k_2) denote the first (resp. last) occurrence of symbol q_i in $\mathcal{S}_\alpha[\mathsf{First}, \mathsf{Last}]$ (hence we have $\mathsf{First} \leq k_1 \leq k_2 \leq \mathsf{Last}$ and $\mathcal{S}_\alpha[k_1] = \mathcal{S}_\alpha[k_2] = q_i$). By properties of the xbw transform, First' is the position of the first child of node $\mathcal{S}[k_1]$ and Last' is the position of the last child of $\mathcal{S}[k_2]$. The main loop of algorithm $\mathsf{SubPathSearch}$ (Steps 3–9) computes the new range $[\mathsf{First}', \mathsf{Last}']$ using essentially the same computation as $\mathsf{GetChildren}$.

Theorem 5 *Using the same data structures described in Theorem 4, we can perform a subpath search for a string β in $O(|\beta| \log |\Sigma|)$ time for a general alphabet Σ, and in optimal $O(|\beta|)$ time if $|\Sigma| = O(\text{polylog}(t))$.* ∎

6 Tree Entropy and Compression

The locality principle exploited in universal compressors for strings is that each element of a sequence depends most strongly on its nearest neighbors, that is, predecessor and successor. The *context* of a symbol s is therefore defined on strings as the substring that *precedes* s. A k-context is a context of length k. The larger is k, the better should be the prediction of s given its k-context. The theory of Markov random fields [28, 33] extends this principle to more general mathematical structures, including trees, in which case a symbol's nearest neighbors are its ancestors, its children or any set of *nearest* nodes. In what follows we extend, in a natural way, the notion of k-context for string data to the notion of k-context for labeled-tree data. We let $\pi[u]$ be the *context* of node u; the *k-context* of u is the k-long prefix of $\pi[u]$ denoted by $\pi_k[u]$. The context $\pi_k[u]$ should be a *good predictor* for the labels that might be assigned to u. A larger k induces a better prediction. Like on string data, we postulate that *similarly labeled nodes descend from similar contexts*, and that *the similarity of contexts is proportional to the length of their shared prefix*. Our goal now is to show that node labels get distributed in $\mathsf{xbw}(\mathcal{T})$ according to a pattern that *clusters closely the similar labels*. 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 of u and v in $\mathcal{S}_\alpha$*. This is therefore a powerful homogeneity property over $\mathcal{S}_\alpha$ that is the analog of the properties of the standard BW transform [2] on strings for labeled trees.

To measure the compressibility of a labeled tree, we proceed as follows. Let β be a string drawn from an alphabet of h symbols, and let b_i be the number of occurrences of the ith symbol in β. We define the zeroth order empirical entropy *on strings* as usual: $H_0(\beta) = -\sum_{i=1}^{h}(b_i/|\beta|)\log(b_i/|\beta|)$. Given H_0, we define the *k-th order empirical entropy on labeled trees*. For any positive integer k, we use the notation $\mathtt{cover}[\rho]$ to denote the string of symbols labeling nodes in $\mathcal{T}$ whose context is prefixed by the string ρ. We can then define $H_k(\mathcal{T}) = \sum_{\rho\in\Sigma^k} |\mathtt{cover}[\rho]|\, H_0(\mathtt{cover}[\rho])$. It is not difficult to see that $H_k(\mathcal{T})$ extends to labeled trees the notion of kth order empirical entropy introduced for strings, with the k-contexts reinterpreted "vertically" by reading the symbols labeling the upward paths.

Given the structure of $\mathsf{xbw}(\mathcal{T})$ it is easy to observe that, for any k, the concatenation of strings $\mathtt{cover}[\rho]$ for all possible $\rho \in \Sigma^k$ taken in lexicographic order, gives the string $\mathcal{S}_\alpha$. This is therefore exactly the same strong property holding for the BW transform (see [20, Eqn. (9)]), here generalized to labeled trees. As a consequence and given the definition of H_k, we have reduced the *tree compression* problem up to $H_k(\mathcal{T})$ to a *string compression* problem up to H_0. This allows us to use all the machinery developed for string data.

We are left with the problem of locating the strings $\mathtt{cover}[\rho]$ within $\mathcal{S}_\alpha$. For a fixed k, it is enough to use the *longest common prefix* (lcp) information we can derive from the algorithm PathSort of Section 3. Indeed the substrings $\mathtt{cover}[\rho]$ are delimited by lcp-values smaller than k, and provide a partition of $\mathcal{S}_\alpha$. As an alternative, we can use an adaptation of the compression boosting technique [8] which finds an optimal partition of $\mathcal{S}_\alpha$ ensuring a compression bounded by $H_k(\mathcal{T})$ *for any* positive k (details in the full paper).

Theorem 6 *Let $\mathcal{A}$ be a compressor that compresses any string w into $|w|H_0(w) + \mu|w|$ bits. The string $\mathsf{xbw}(\mathcal{T})$ can be compressed in $tH_k(\mathcal{T}) + t(\mu+2) + o(t) + g_k$ bits, where g_k is a parameter that depends on k and on the alphabet size (but not on $|w|$). The bound holds for any positive k, whereas the approach is independent of k. The time and space complexities are both $O(t)$ plus the time and space complexities of applying $\mathcal{A}$ over a t-symbol string.* ■

An example of a compressor $\mathcal{A}$ is the classical Arithmetic Coder which achieves $\mu \approx .01$. A lower bound of $2t - O(\log t)$ can be trivially derived for the representation of (the t-ordinal tree) $\mathcal{T}$ by dropping its labels and just considering its structure. Our solution is off from this lower bound by $(H_k(\mathcal{T}) + \mu + 2)/2$ factor which depends on the entropy of the labels distribution over $\mathcal{T}$. It is never worse than the $\log|\Sigma|$ factor we obtained in earlier sections, but could be significantly better depending on the tree entropy.

7 Concluding Remarks

We have introduced the xbw transform which is a new approach to compressing and indexing tree-shaped data. We list three data structural and compression problems whose solution, combined with our tree transform, would extend our results even more.

Problem 1. Design a *compressed* data structure for supporting constant time rank_c and select_c queries over a string drawn from an *arbitrary* alphabet. Here the term "compressed" means *up to* H_k; the problem is currently open even for H_0. This data structure used over $\mathcal{S}_\alpha$ would help us obtain *both* compression and indexing over labeled trees simultaneously.

Problem 2. In some applications, node labels are strings of arbitrary length, not just single character labels. A key example is the labeled tree structure representation of XML documents. In this setting entropy depends on both the *vertical* contexts (i.e. the labels on the upward path leading to a node) and the classical *horizontal* contexts (i.e. the preceding characters in a single node label). We leave open the problem of defining a new notion of entropy that takes into account both contexts, and a compression algorithm that achieves optimality with respect to this new notion in efficient time.

Problem 3. Still in the context of Problem 2, where node labels are strings of arbitrary length, it is an interesting problem to support an extension of the subpath search defined as follows. Given a subpath $\beta = q_1 \ldots q_k$ and a string s, we want to find all nodes that have upward path prefixed by β and whose labels contain s as a substring. This query finds applications in searches over XML documents.

References

[1] D. Benoit, E. Demaine, J. Ian Munro, R. Raman, V. Raman, and S. Rao. Representing Trees of Higher Degree. *WADS*, 1999. To appear in *Algorithmica*.

[2] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. *DEC TR No. 124*, 1994.

[3] R. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4), 843-850, 1988.

[4] J. Cheney. Statistical models for term compression. *IEEE DCC*, 2000.

[5] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. *IEEE DCC*, 2001.

[6] Y. Cohen and M.S. Landy and M. Pavel. Hierarchical coding of binary images. *IEEE Trans. Pattern Analysis and Mach. Intel.*, 284-298, 1985.

[7] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. *To appear in JACM*. See also *IEEE FOCS*, 390-398, 2000.

[8] P. Ferragina and R. Giancarlo and G. Manzini and M. Sciortino. Boosting Textual Compression in Optimal Linear Time. *To appear in JACM*. TR 13, Univ. of Palermo, 2004. See also *ACM-SIAM SODA 04* and *CPM 03*.

[9] E. Fredkin. Trie Memory. *Comm. ACM* 3(9), 490-499, 1960.

[10] R.F. Geary and R. Raman and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM-SIAM SODA*, 2004.

[11] R. Goldman and J. Widom. Dataguides: enabling query formulation and optimization in semi structured databases. *Proc. of VLDB*, 436–445, 1997.

[12] R. Grossi and J. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *ACM STOC*, 397-406, 2000.

[13] R. Grossi, A. Gupta and J. Vitter. High-Order Entropy-Compressed Text Indexes. *ACM-SIAM SODA*, 841-850, 2003.

[14] A. Hamou-Lhadj and T.C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. *Proc. WODA*, 2003.

[15] G. Jacobson. Space-efficient Static Trees and Graphs. *FOCS* 1989, 549–554. Also, thesis at CMU.

[16] J. Katajainen and M. Penttonen and J. Teuhola. Syntax-directed compression of program files. *Software Practice & Experience*, 16(3), 269-276, 1986.

[17] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. *Proc. ICALP*, 943-955, 2003.

[18] S. R. Kosaraju. Efficient Tree Pattern Matching. *IEEE FOCS*, 178-183, 1989.

[19] H. Liefke and D. Suciu. XMILL: An Efficient Compressor for XML Data. *ACM SIGMOD*, 153–164, 2000.

[20] G. Manzini. An Analysis of the Burrows-Wheeler Transform. *Journal of the ACM*, 48(3), 407-430, 2001.

[21] I. Munro and V. Raman. Succinct Representation of Balanced Parentheses, Static Trees and Planar Graphs. *IEEE FOCS 1997*, 118–126.

[22] J. I. Munro and V. Raman and S. Srinivasa Rao. Space Efficient Suffix Trees. *J. Algorithms*, 39(2), 205–222, 2001.

[23] G. Navarro, P. Ferragina, G. Manzini, and V. Mäkinen. Succinct representation of sequences. Technical Report TR/DCC-2004-5, Dept. of Computer Science. University of Chile, 2004.

[24] G.C. Necula and P. Lee. The design and implementation of a certifying compiler. *ACM SIGPLAN Conf. on Programming Lang. Design and Impl.*, 333-344, 1998.

[25] C. Nevill-Manning and I. Witten and D. Maulsby. Compression by induction on hierarchical grammars. *IEEE DCC*, 244-253, 1994.

[26] R. Raman and V. Raman and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *ACM-SIAM SODA*, 233-242, 2002.

[27] J.R. Rico-Juan and J. Calera-Rubio and R.C. Carrasco. Smoothing and Compression with Stochastic k-testable Tree Languages. *Pattern Recognition*, 2004.

[28] F. Spitzer. Markov random fields on an infinite tree. *Annals of Probability*, 3(3), 387–398, 1975.

[29] R. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *Computer Journal*, 29(4), 307-314, 1986.

[30] J. Tarhio. Context coding of parse trees. *IEEE DCC*, 1995.

[31] P. Weiner. Linear Pattern Matching Algorithms. *Proc. 14th IEEE Annual Symp. on Switching and Automata Theory*, 1-11, 1973.

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

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

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

[35] XQuery 1.0: An XML Query Language. W3C. Feb 2005. `http://www.w3.org/TR/xquery/`.