Artificial Intelligence and Data Engineering

Algorithms for Massive Data

Project: **The XBW-Transform for Labeled Trees**

Professor
Nicola Prezza

Author
Davide Tonetto
Student ID 884585

Academic year
2024/2025

Contents

Chapter 1

Introduction

The Extended Burrows-Wheeler Transform (XBWT) is a data structure designed to efficiently compress and index labeled trees. A labeled tree is a data structure where each node is assigned a label from a given alphabet, and the tree can have an arbitrary shape and degree.

XBWT works by linearizing a labeled tree into two coordinated arrays: one capturing the structural properties of the tree and the other storing its labels. This transformation allows for efficient representation, navigation, and querying of the tree. The key advantage of XBWT lies in its ability to compress labeled trees while supporting a wide range of operations, such as parent-child navigation and sophisticated path-based searches, in (near-)optimal time and space.

One of the primary applications of XBWT is in the compression and indexing of hierarchical data formats, such as XML documents. It provides significant improvements in both compression ratio and query performance compared to traditional tools, making it an invaluable resource for data-intensive applications in fields like bioinformatics, information retrieval, and big data analytics.

This project aims to implement the XBWT data structure and explore its applications in the context of labeled trees. We will start by providing an overview of the theoretical foundations of the XBWT. Finally, we will describe and compare the algorithms for constructing the XBWT and demonstrate its use in compressing and indexing labeled trees.

Let's start with a quick overview of the XBWT and its theoretical foundations.

1.0.1 How XBWT Works

The transformation process of XBWT is as follows:

1. **Path Sorting:** The labeled tree is linearized by sorting its nodes based on the *paths* from each node's parent to the root. The resulting order groups nodes with similar upward paths together, clustering related labels.
2. **Array Construction:** Two arrays, S_{last} and S_α, are generated:
   - S_{last} stores structural information, such as whether a node is the last child of its parent. This encodes the tree's structure without the need for explicit pointers.
   - S_α stores the labels of the nodes in the sorted order determined by their upward-path sorting.

3. **Compression:** Both S_{last} and S_α are highly compressible due to the clustering of similar labels and structural redundancy.

1.0.2 Key Properties of XBWT

The XBWT has several key properties that make it an effective tool for labeled tree compression and indexing:

- **Succinctness:** The XBWT representation of a labeled tree uses space close to the *information-theoretic lower bound*, which is $2t - \Theta(\log t) + t \log |\Sigma|$ bits for a tree with t nodes and an alphabet of size $|\Sigma|$.
- **Efficient Querying:** XBWT supports a range of navigational operations, such as finding the parent, child, or subtree of a node in near-optimal time.
- **Scalability:** XBWT is particularly useful for large-scale hierarchical data, such as XML documents or phylogenetic trees, where both compression and fast querying are critical.

1.1 Project implementation

The XBWT data structure will be implemented in C++ using the Succinct Data Structure Library 2.0 (SDSL) for efficient representation and manipulation of compressed data structures. We will develop two algorithms for constructing the XBWT: one efficient linear-time recursive algorithm and one more straightforward iterative algorithm. Also, we will implement the necessary data structures and algorithms for navigating and querying the XBWT, such as parent-child navigation and path-based searches.

The code is available on GitHub at `https://github.com/davide-tonetto-884585/XBWT`. The project will be structured as follows:

- **XBWT.hpp**: File containing the class definition and implementation for the generic XBWT data structure.
- **LabeledTree.hpp**: File containing the class definition and implementation for the generic labeled tree data structure used to feed and test the XBWT.
- **main.cpp**: Main file containing the test cases and examples for the XBWT implementation.
- **experiments.cpp**: File containing the experiments and performance evaluation of the XBWT construction algorithms and compression efficiency.
- **CMakeLists.txt**: CMake configuration file for building the project.

The project will be developed and tested on a Linux environment using the GCC compiler and the CMake build system.

Chapter 2

Theoretical Background on Labeled Trees

2.1 Labeled Trees

A **labeled tree** is a rooted, ordered, hierarchical data structure in which every node is assigned a label from a predefined alphabet Σ. The structure consists of nodes connected by edges, forming a directed acyclic graph. Formally, a labeled tree T with t nodes can be defined as $T = (V, E, \ell)$, where:

- V is the set of nodes,
- $E \subseteq V \times V$ is the set of directed edges, and
- $\ell : V \to \Sigma$ is a labeling function that assigns a label $\ell(u) \in \Sigma$ to each node u.

In the case of ordered labeled tree, the children of a node in the tree are ordered, meaning their positions relative to each other matter. A labeled tree can have arbitrary degrees and shapes, and the alphabet Σ used for labels can be of arbitrary size.

2.1.1 Applications of Labeled Trees

Labeled trees are widely used in computer science and data representation due to their hierarchical structure and flexibility in modeling relationships. Prominent applications include:

1. **XML Data Representation:** XML documents are often modeled as labeled trees, where each element is a node labeled by its tag, and hierarchical nesting represents parent-child relationships.
2. **JSON Data Representation:** JSON objects can be viewed as labeled trees, with keys as labels and values as children.
3. **Bioinformatics:** Labeled trees are used to represent phylogenetic trees, genome annotations, and hierarchical clustering.
4. **Compiler Design:** Abstract Syntax Trees (ASTs) for programming languages are labeled trees that capture the structure of code.
5. **File Systems:** The directory structure of file systems can be viewed as labeled trees.

Efficient representation, navigation, and querying of labeled trees are essential for many applications, motivating the development of specialized data structures and algorithms.

2.2 Compressing and Indexing Labeled Trees

The goal of compressing and indexing labeled trees is to design a compressed storage scheme for a labeled tree T with t nodes that allows for efficient navigation operations in T, as well as fast search and retrieval of subtrees or paths within T. To be effective, the compressed representation should minimize the space required to store the tree while supporting a wide range of operations in (near-)optimal time.

Let u be a node in the labeled tree T and let $c \in \Sigma$. We define the following navigation operations on T:

- **Navigational queries:** ask for the parent of u, the i-th child of u, or the label of u. The last two operations might be restricted to the children of u with a specific label c.
- **Path queries:** retrieve the nodes in the subtree rooted at u (any possible order should be implemented).
- **Subpath queries:** ask for the (number of occurrences of) nodes of T that descend from a labeled subpath P. Which may be anchored anywhere in the tree (i.e., not necessarily in its root).

A naive solution to index labeled trees is to store the tree in a straightforward manner, such as a list of nodes with their labels and parent-child relationships using pointer in $O(tlogt)$. However, this representation is not space-efficient and does not support fast navigation or query operations.

Many data structures have been proposed to compress and index labeled trees, each with its trade-offs in terms of space usage, query performance, and supported operations. One of the most successful approaches is the Extended Burrows-Wheeler Transform (XBWT), which extends the classical Burrows-Wheeler Transform (BWT) to handle labeled trees efficiently.

Before the advent of XBWT, Kosaraju [**kosaraju1989efficient**] proposed a method to index labeled trees by extending the concept of prefix sorting, which is commonly applied to strings, to work with labeled trees by leveraging the structure of tries (prefix trees). To achieve this, he introduced the idea of constructing a suffix tree for a reversed trie allowing subpath queries in $O(|P| \log |\Sigma| + occ)$ time, where occ is the number of occurrences of P in T but still requiring $O(t \log t)$ space and so not being compressed.

2.3 Succinct Data Structures for Trees

In order to compress the index of labeled trees, we need to avoid the use of pointers and store the tree in a space-efficient manner. Succinct data structures are a class of compressed data structures that support efficient navigation and query operations on the compressed data. These structures are designed to use close to the information-theoretic lower bound on space while providing fast access to the original data. They were first introduced by Jacobson [**jacobson1989space**] and have been applied to various problems in string processing, graph theory, and data compression.

2.3.1 Information-Theoretic Lower Bound for Trees

The information-theoretic lower bound for storing an unlabeled tree with t nodes is given by:

- The number of binary unlabeled trees with t nodes is given by the Catalan number $C_t = \frac{1}{t+1}\binom{2t}{t}$ that can can be approximated as $C_t \approx \frac{4^t}{t^{3/2}\sqrt{\pi}}$ using Stirling's approximation.
- The entropy (or the information-theoretic minimum number of bits to encode the structure of the tree) is the logarithm (base 2) of the total number of trees, which is $-\log_2 C_t \approx 2t - \frac{1}{2}\log_2 \pi t^3$.
- The correction term $\frac{1}{2}\log_2 \pi t^3$ grows slower that the linear term $2t$, we can say that $-\frac{1}{2}\log_2 \pi t^3 = -\Theta(\log t)$.
- The information-theoretic lower bound for storing an unlabeled tree with t nodes is $2t - \Theta(\log t)$ bits.

Then for labeled trees, the labels assigned to each node must be stored, which requires an additional space:

- Let Σ denote the alphabet of labels, and let $|\Sigma|$ be the size of the alphabet.
- Each node in the tree requires $\log_2 |\Sigma|$ bits to store its label.
- Therefore, for t nodes, the total space required to store the labels is $t \log_2 |\Sigma|$ bits.

Combining the structural representation and the labeling, the information-theoretic lower bound for storing a labeled tree is:

$$2t - \Theta(\log t) + t \log_2 |\Sigma| \text{ bits}$$

Chapter 3

Extended Burrows-Wheeler Transform

In 2005, Ferragina et al. [**ferragina2009compressing**] observed that succinctness could be achieved for labeled trees by exploiting an index scheme that fit into a space proportional to the entropy-compressed edge labels plus the succinct tree's topology. This observation was the starting point for the development of the Extended Burrows-Wheeler Transform (XBWT). Let's start by defining the XBWT.

3.1 Definition of XBWT

The **Extended Burrows-Wheeler Transform (XBWT)** is a data structure designed to efficiently compress and index *labeled trees*. Inspired by the classical Burrows-Wheeler Transform (BWT) [**burrows1994block**] for strings, the XBWT extends these principles to hierarchical structures, enabling efficient storage, navigation, and querying of trees. It is particularly effective for trees where each node has a label drawn from an alphabet Σ and the tree structure has an arbitrary shape and degree.

Given an ordered labeled tree T of arbitrary fan-out, depth and shape, with n internal nodes and l leaves (t nodes in total) and alphabet Σ. Let u be a node in T, we define the following information:

- $last[u]$: is a binary value that is 1 if u is the last (rightmost) child of its parent, and 0 otherwise.
- $\alpha[u]$: denotes the label of node u plus one bit that is 1 if u is a leaf and 0 otherwise.
- $\pi(u)$: is the string obtained by concatenating the labels of the nodes on the path from u's parent to the root of T (the root has an empty π component).

Then, to define the XBWT, a sorted multi-set S consisting of t triplets (one per node of T) is built, where each triplet is of the form $(last[u], \alpha[u], \pi(u))$ for some node u in T. S is built by traversing T in a pre-order fashion, for each visited node u, the triplet $(last[u], \alpha[u], \pi(u))$ is added to S, then S is stably sorted in accordance with the lexicographic order of the π component of the triplets.

Theorem 1. *The XBWT of a labeled tree T consist of the two arrays $\{S_{last}, S_\alpha\}$ after sorting, and takes $2t + t\log|\Sigma|$ bits of space.*

3.2 Properties of XBWT

The following two properties of the ordered multi-set S are crucial for the indexing scheme, they immediately follow from the composition of the transform and from the way S is built.

3.2.1 Property 1

1. S_{last} has n 1s (one for each internal node) and l 0s (one for each leaf).
2. S_α is a permutation of the labels of the nodes in T.
3. S_π contains all the upward labeled paths of T consisting internal node labels only. Also, each path is repeated a number of times equal to the number of its offsprings.

3.2.2 Property 2

1. The first triplet of S refers to the root of T.
2. The triplet of node u precedes the triplet of node v in S iff either $\pi[u] < \pi[v]$ or $\pi[u] = \pi[v]$ and u precedes v in the pre-order traversal of T.
3. Let $u_1, \ldots, u_z$ be the children of node u in T, then the triplets of $u_1, \ldots, u_z$ are consecutive in S following this order. Moreover, the subarray $S_{\text{last}}[u_1 \ldots u_z]$ provides the unary encoding of u's degree, namely $S_{\text{last}}[u_z] = 1$ and $S_{\text{last}}[u_i] = 0$ for $1 \leq i < z$.
4. Let u, v be two nodes in T having the same label $\alpha[u] = \alpha[v]$, then if the triplets of u precedes the triplets of v in S, then the contiguous block of children of u in S precedes the contiguous block of children of v in S.

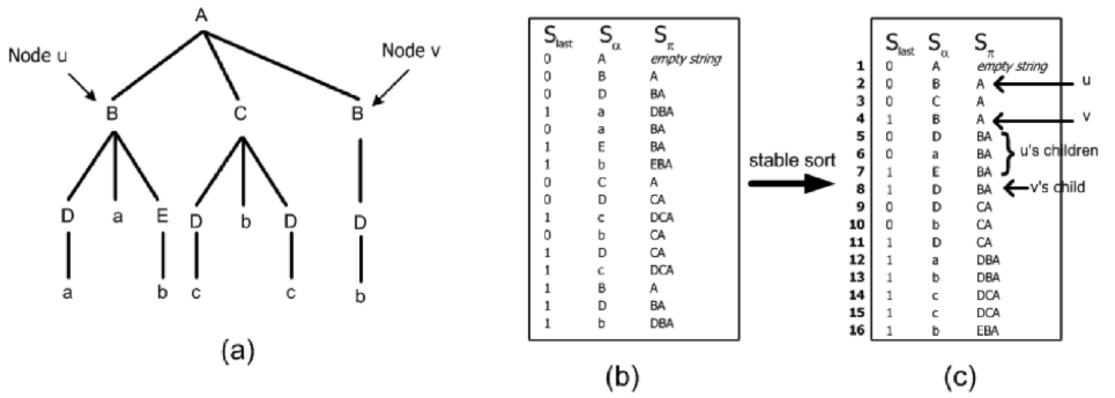
3.2.3 Property 3

Let $c \in \Sigma$ be an internal node label, and let $S[j_1, j_2]$ be all triplets whose π-components are prefixed by c. If u is the i-th node labeled c in S_α, its children occur contiguously within $S[j_1, j_2]$ and delimited by the $i-1$-th and i-th bit set to 1 in $S_{\text{last}}[j_1, j_2]$.

3.3 XBWT Construction

A naive approach to build the XBWT would be to explicitly construct S through the concretization of π-strings and then sort it using a stable sorting algorithm. However, this approach would require $\Theta(t^2)$ space in the worst case, which is not feasible for large deep trees. To overcome this issue, Ferragina et al. [**ferragina2009compressing**] proposed a more efficient algorithm that builds S in linear time and $O(t \log t)$ space.

The linear time algorithm is called **pathSort**, it is based on a generalization of the Skew algorithm for suffix array construction of strings [**karkkainen2006linear**]. Let's see briefly how the Skew algorithm works.



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

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

3.3.1 Skew Algorithm

The Skew algorithm is an efficient method for constructing the suffix array of a string in linear time. A suffix array is a data structure that lists the starting indices of all the suffixes of a string in lexicographical order, and it is widely used in various string processing algorithms.

Algorithm Overview

1. Divide the String

The algorithm begins by partitioning the indices of the string into three groups based on their modulo 3 value:

- S_0: Indices congruent to 0 mod 3.
- S_1: Indices congruent to 1 mod 3.
- S_2: Indices congruent to 2 mod 3.

The suffixes starting at positions in S_1 and S_2 are combined into a single group called S_{12}.

2. Sort Suffixes in S_{12}

To sort the suffixes in S_{12}, the algorithm considers the triplets of characters starting at each position in S_{12}. These triplets are sorted using a linear-time sorting algorithm, such as radix sort, and then renamed by assigning each triplet an integer value representing its rank in the sorted order. If all triplets are unique, the sorting is complete; otherwise, the same procedure is applied recursively to the sequence of ranks obtained.

3. Sort Suffixes in S_0

Once the suffixes in S_{12} are sorted, the algorithm proceeds to sort the suffixes in S_0. To compare two suffixes starting at positions i and j in S_0, it compares the first characters of their respective substrings. If these are equal, it compares the suffixes starting at positions $i+1$ and $j+1$, whose ranks are already known from the sorting of S_{12}.

4. Merge the Sorted Orders

Finally, the sorted orders of the suffixes in S_0 and S_{12} are merged to obtain the complete suffix array of the original string. This merging process can be performed in linear time, ensuring the overall efficiency of the algorithm.

3.3.2 PathSort Algorithm

The pseudocode of the pathSort algorithm is shown in Algorithm **??**. As we can see the algorithm is based on the Skew algorithm, but it is adapted to work on labeled trees. The main idea is to recursively sort the upward subpaths of the tree starting at nodes in levels $\not\equiv j \pmod 3$, then sort the upward subpaths starting at nodes in levels $\equiv j \pmod 3$ using the result of the previous step, and finally merge the two sets of sorted subpaths by exploiting their lexicographic names. The value of j is chosen in such a way that the number of nodes in `IntNodes` whose level is $\equiv j \pmod 3$ is at least $t/3$ so that a constant fraction of upward paths are ensured to be dropped at each recursive step. Is important to note that:

1. The height of the new (contracted) tree shrinks by a factor three, hence the node naming requires the radix sort over triples 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 $R(t) = R(2t/3) + \Theta(t) = \Theta(t)$;
3. following an argument similar to [**karkkainen2006linear**], the names of the dropped subpaths can be computed in $O(t)$ time from the names of the non dropped subpaths, by radix sorting.

Algorithm 1 PATHSORT(T)

```
1: Create the array IntNodes[1, $t$], initially empty.
2: Visit the internal nodes of $T$ in pre-order. Let $u$ denote the $i$-th visited node.
3: Write in IntNodes[$i$] the symbol $\alpha[u]$, the level of $u$ in $T$, and the position in
   IntNodes of $u$'s parent.
4: 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 upward subpaths starting at
   nodes in levels $\not\equiv j \pmod 3$.
5: Sort the upward subpaths starting at nodes in levels $\equiv j \pmod 3$ using the
   result of Step 3.
6: Merge the two sets of sorted subpaths by exploiting their lexicographic names.
```

3.3.3 Recursive Step of PathSort

At each recursive step, the algorithm constructs the array `IntNodes` (as shown in Figure **??**-(b)), which stores the triplets $(\alpha[u], \text{level}(u), \text{parent}(u))$ for every internal node u in the given tree T.

Next, the algorithm selects a value j such that the number of nodes in `IntNodes` with depth $\equiv j \pmod 3$ is at least $t/3$. Based on this choice, two separate arrays are created:

- `IntNodesAtPosJ`, containing nodes at levels $\equiv j \pmod 3$,
- `IntNodesNotAtPosJ`, containing nodes at levels $\not\equiv j \pmod 3$

For each node u in `IntNodesNotAtPosJ`, the algorithm extracts the upward path consisting of the first three ancestors of u. These paths are then sorted using radix sort. If the sorted upward paths contain duplicates, the algorithm recursively calls the PathSort function on a new contracted tree, where nodes are renamed according to their sorted paths. Otherwise, if all upward paths are unique, the nodes in `IntNodesAtPosJ` are sorted and subsequently merged with `IntNodesNotAtPosJ` using lexicographic ordering, following the same merging strategy as in the Skew algorithm.

3.4 Inverting the XBWT

Property 3 **??** ensures that the two array S_{last} and S_α of the XBWT can be used to reconstruct the original tree T. The algorithm to invert the XBWT is linear in time and requires $O(t \log t)$ bits of space.

The algorithm **??** initially builds the array F that stores the first entry in S whose π-component is prefixed by a symbol x (F approximates S_π at its first symbol). Then, it exploits the array F to efficiently build the array J that stores the position in S of the first child of each node in T. Finally, the algorithm deploys the array J to simulate a depth-first visit of T, creates its labeled nodes, and properly connects them to their parents.

Algorithm 2 RebuildTree(`xbw[`T`]`)

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

Algorithm 3 BuildF(`xbw[`T`]`)

```
1: for $i = 1, \ldots, |\Sigma_N|$ do
2:     $C[S_\alpha[i]] \leftarrow C[S_\alpha[i]] + 1$; ▷ Count the occurrences of node labels
3: end for
4: $F[1] = 2$; ▷ $S_\pi[1]$ is the empty string
5: for $i = 1, \ldots, |\Sigma_N| - 1$ do ▷ Consider just the internal-node labels
6:     $s = 0$; $j = F[i]$;
7:     while $s \neq C[i]$ do ▷ Not all blocks of children have been passed
8:         if $S_{\text{last}}[j++] = 1$ then $s++$; ▷ One further block of children has passed
9:         end if
10:    end while
11:    $F[i+1] = j$;
12: end for
13: return $F$.
```

Algorithm 4 BuildJ(`xbw[`T`]`, F)

```
1: for $i = 1, \ldots, t$ do
2:     if $S_\alpha[i] \in \Sigma_L$ then
3:         $J[i] = -1;$                                  ▷ $S_\alpha[i]$ is a leaf label
4:     else
5:         $z = J[S_\alpha[i]];$
6:         while $S_{\text{last}}[z] \neq 1$ do $z++;$    ▷ Reach the last child of $S_\alpha[i]$
7:         end while
8:         $F[S_\alpha[i]] = z + 1;$
9:     end if
10: end for
11: return $J$.
```

3.5 Compressing Labeled Trees

Let the k-context of a node u in a tree T be defined as the first k symbols of the π-component of the triplet associated with u. We denote this k-long prefix as $\pi_k[u]$. Thus, $\pi_k[u]$ represents the subpath of length k leading to u in T, or equivalently, the node u descends from a subpath labeled as $\pi_k[u]$, where the nodes in $\pi_k[u]$ are encountered in an upward direction.

The XBW[T] exhibits a local homogeneity property on the string S_α, which can be demonstrated through the concept of k-contexts on trees. This property mirrors the strong local homogeneity exhibited by strings under the Burrows-Wheeler Transform [Burrows and Wheeler 1994] when applied to labeled trees. Specifically, node labels in T are distributed across S_α in a manner that clusters together those labels originating from "similar" upward paths that share long prefixes.

To illustrate this, let us consider two arbitrary nodes u and v in T, and examine their contexts $\pi[u]$ and $\pi[v]$. Given the sorting of S, the greater the length of the shared prefix between $\pi[u]$ and $\pi[v]$, the closer the corresponding labels $\alpha[u]$ and $\alpha[v]$ will be in the string S_α. These closely spaced labels are expected to be few in number, resulting in S_α exhibiting local homogeneity. As a consequence, we can leverage the advanced algorithmic techniques developed for BWT-based compression methods to achieve efficient compression.

At the end, the XBWT is used for turning the labeled tree compression problem into a string compression problem. To this aim, two string compressors C_α and C_{last} are used to squeeze the two strings that compose XBW[T], by exploiting their fine specialties. Of course, many choices are possible for C_{last} and C_α, each having implications on the algorithmic time and compression bounds.

In general, let C_α be a k-th order string compressor that compresses any string w into $|w|H_k(w) + |w| + o(|w|)$ bits, taking $O(|w|)$ time; and let C_{last} be an algorithm that stores S_{last} without compression. With this simple instantiation, the labeled tree T can be compressed within $tH_k(S_\alpha) + 2t + o(t)$ bits and takes $O(t)$ optimal time.

3.6 Indexing a compressed labeled tree

In order to implement the efficient operations listed in **??** using the compressed arrays S_{last} and S_α of XBWT, we need that the chosen compressors C_α and C_{last} support the following operations:

Given a string $S[1,t]$ over alphabet Σ

- $rank_c(S,q)$: gives the number of times the symbol $c \in \Sigma$ appears in $S[1,q]$.
- $select_c(S,i)$: gives the position of the i-th occurrence of the symbol $c \in \Sigma$ in S.

The compressed indexing of XBWT$[T]$ will be based on three compressed data structures that support rank and select queries over the two strings S_α and S_{last}, and over an auxiliary binary array $A[1,t]$ defined as: $A[1] = 1$, $A[j] = 1$ if and only if the first symbol of $S_\pi[j]$ differs from the first symbol of $S_\pi[j-1]$. Hence, A contains at most $|\Sigma|+1$ bits set to 1 out of t positions. It is also easy to see that, by means of rank and select operations over A, we can succinctly implement the array F deployed in the algorithms **??** and **??**.

The following methods are supported by the compressed index:

- **GetRankedChild(i, k)**: returns the position in S of the k-th child of u; the output is -1 if this child does not exist. As an example, `GetRankedChild(2, 2) = 6` in Figure **??**.
- **GetCharRankedChild(i, c, k)**: returns the position in S of the triplet representing the k-th child of u among the ones whose label is c. The output is -1 if this child does not exist. As an example, `GetCharRankedChild(1, B, 2) = 4` in Figure **??**.
- **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 S of the triplet representing the parent of u. The output is -1 if $i = 1$ (the root). As an example, `GetParent(8) = 4` in Figure **??**.
- **GetSubtree(i)**: returns the node labels of the subtree rooted at u. Any possible order (i.e., pre, in, post) may be implemented.
- **SubPathSearch(P)**: determines the range $S[\text{First}, \text{Last}]$ of nodes, which are immediate descendants of each occurrence of the labeled path $P = c_1 c_2 \cdots c_k$ in T. Note that all strings in $S_\pi[\text{First}, \text{Last}]$ are prefixed by P^R. As an example, `SubPathSearch(BD) = [12, 13]` and `SubPathSearch(AB) = [5, 8]` in Figure **??**.

It is important to note that their time complexity is dependent on the specific implementation for rank and select over the compressed strings S_α and S_{last}.

Let's now see how to implement some of the above methods (from which the others can be derived) using the rank and select operations over the compressed strings S_α and S_{last}.

GetChildren(i)

Algorithm 5 GetChildren(i)

```
 1: if $S_\alpha[i] \in \Sigma_L$ then
 2:     return $-1$                                        ▷ $S[i]$ is a leaf
 3: end if
 4: $c \leftarrow S_\alpha[i]$                             ▷ $S[i]$ is labeled $c$
 5: $r \leftarrow \text{rank}_c(S_\alpha, i)$
 6: $y \leftarrow \text{select}_1(A, c)$                   ▷ $y = F[c]$
 7: $z \leftarrow \text{rank}_1(S_{\text{last}}, y - 1)$
 8: First $\leftarrow \text{select}_1(S_{\text{last}}, z + r - 1) + 1$
 9: Last $\leftarrow \text{select}_1(S_{\text{last}}, z + r)$
10: return (First, Last)
```

The algorithm exploits directly the properties described before, in particular the Property 3 (**??**). The rank operation at line 5 is used to get the number r of nodes labeled c up to position i in S_α. Then, the position $F[c]$through a select operation on A (line 6). By Property 3, the children of $S[i]$ are located at the r-th block of children following position $F[c]$. Lines $8 - 9$ identify this block.

GetParent(i)

Algorithm 6 GetParent(i)

```
1: if $i == 1$ then
2:     return $-1$                                         ▷ $S[i]$ is the root of $\mathcal{T}$
3: end if
4: $c \leftarrow \text{rank}_1(A, i)$
5: $y \leftarrow \text{select}_1(A, c)$
6: $k \leftarrow \text{rank}_1(S_{\text{last}}, i - 1) - \text{rank}_1(S_{\text{last}}, y - 1)$
7: $p \leftarrow \text{select}_c(S_\alpha, k + 1)$
8: return $p$
```

Algorithm **??** is based on the Property 3 (**??**) and it is the inverse of the GetChildren method. At line 4 the algorithm computes the label c of the parent of $S[i]$ that prefixes the upward path leading to $S[i]$. Then, the parent of $S[i]$ is searched among the nodes labeled c in S_α by exploiting Property 3 in a reverse manner. Namely, the number k of children-blocks in the range $S[y, i]$ is computed, these are children of nodes labeled c and preceding i in the stable sort of S. Then, the k-th occurrence of c in S_α is selected, which is properly the parent of $S[i]$.

SubPathSearch(P)

Algorithm 7 SubPathSearch(P)

```
1: $First \leftarrow F(c_1)$; $Last \leftarrow F(c_1 + 1) - 1$
2: if $First > Last$ then
3:     return "$P$ is not a subpath of $T$"
4: end if
5: for $i \leftarrow 2, \dots, k$ do
6:     $k_1 \leftarrow \text{rank}_{c_i}(S_\alpha, First - 1)$; $z_1 \leftarrow \text{select}_{c_i}(S_\alpha, k_1 + 1)$    ▷ first entry in $S_\alpha[First, t]$ labeled $c_i$
7:     $k_2 \leftarrow \text{rank}_{c_i}(S_\alpha, Last)$; $z_2 \leftarrow \text{select}_{c_i}(S_\alpha, k_2)$    ▷ last entry in $S_\alpha[1, Last]$ labeled $c_i$
8:     if $z_1 > z_2$ then
9:         return "$P$ is not a subpath of $T$"
10:    end if
11:    $First \leftarrow \text{GetRankedChild}(z_1, 1)$    ▷ get the first child of $S[z_1]$
12:    $Last \leftarrow \text{GetRankedChild}(z_2, \text{GetDegree}(z_2))$    ▷ get the last child of $S[z_2]$
13: end for
14: return $(First, Last)$
```

We assume that $P = c_1 c_2 \cdots c_k$ algorithm SubPathSearch computes the range $[First, Last]$ in $|P| = l$ phases, each one preserving the following invariant:

- Invariant of Phase i. At the end of the phase, $S_\pi[First]$ is the first entry prefixed by $P[1,i]^R$, and $S_\pi[Last]$ is the last entry prefixed by $P[1,i]^R$, 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 $F[c_1]$ and $F[c_1 + 1] - 1$, which point to the first and last entry of S_π prefixed by c_1 (by definition of array F). Since we do not have the F array, we implement these operations via rank and select queries over array A. Let us assume that the invariant holds for Phase $i - 1$, and prove that the i-th iteration of the for-loop in algorithm SubPathSearch preserves the invariant. More precisely, let $S_\pi[First, Last]$ be all entries prefixed by $P[1, i - 1]^R$. So $S[First, Last]$ contains all nodes descending from $P[1, i - 1]$. SubPathSearch determines $S[z_1]$ (respectively $S[z_2]$) as the first (respectively last) node in $S[First, Last]$ that descends from $P[1, i-1]$ and is labeled c_i, if any. Then it jumps to the first child of $S[z_1]$ and the last child of $S[z_2]$. From Property 2 (item 2), and the correctness of algorithms GetChildren and GetDegree, we infer that the positions of these two children are exactly the first (respectively last) entry in S whose π-component is prefixed by $P[1, i]^R$.

The time complexity of the SubPathSearch algorithm is $O(l)$, where l is the length of the input path P.

Chapter 4

Implementation of the XBWT

The implementation of the XBWT is based on what has been described in the previous chapters. The implementation is written in C++ and is available on GitHub at the following link: `https://github.com/davide-tonetto-884585/XBWT`.

4.1 Implementation choices

Follows a list of the main choices made during the implementation of the XBWT:

- The implementation is not focused on working for a specific kind of data such as XML documents or JSON files, but it is designed to work with any kind of labeled tree.
- The construction method of the XBWT class takes as input a labeled tree, and construct directly a compressed indexing scheme for it based on the Extended Burrows-Wheeler Transform of the tree as described in the previous chapters.
- In order for the XBWT to work we assume that the labels of the leaf nodes of the given labeled tree are lexicographically greater than the labels of the internal nodes. This is necessary to ensure that the navigational and search operations work correctly.
- The implementation is based on the Succinct Data Structure Library (SDSL) to handle the compressed data structures generated by the XBWT. The SDSL library provides efficient implementations of various compressed data structures and algorithms, which are essential for representing and querying the XBWT efficiently.
- The labels of the alphabet are encoded as integers, starting from 0 to $|\Sigma| - 1$, where $|\Sigma|$ is the cardinality of the alphabet. This encoding respect the order of the labels in the alphabet and allows simplifying and reduce the space needed to store the labels in the compressed data structures. For this reason the constructor of the XBWT class takes as input a generic labeled tree.

4.1.1 Succinct Data Structure Library (SDSL)

The Succinct Data Structure Library (SDSL) is a C++ library that provides efficient implementations of various compressed data structures and algorithms. It is used in this project to handle the compressed data structures generated by the XBWT. The SDSL library provides a wide range of succinct data structures, such as bit vectors, wavelet trees, and compressed suffix arrays, which are essential for representing and querying the XBWT efficiently. The library is available at `https://github.com/`

`simongog/sdsl-lite` [**gbmp2014sea**]. Let's see the implementation details of the SDSL data structures used in the XBWT implementation.

sdsl::rrr_vector

The `sdsl::rrr_vector` is a class of the Succinct Data Structure Library (SDSL), designed to provide space-efficient representations of bit vectors while supporting efficient rank and select operations. This data structure implements the RRR (Raman, Raman, and Rao) encoding method, which compresses bit vectors by partitioning them into fixed-size blocks and encoding each block based on its population count (the number of 1s) and specific configuration [**raman2002succinct**].

The space needed by `sdsl::rrr_vector` for a bit vector of length n with m set bits is $nH_0+o(n)$ ($\approx \lceil \log \binom{n}{m} \rceil$). The rank support is provided by `sdsl::rank_support_rrr` adding 80 bits and requiring $O(\log k)$ time for rank queries, where k is the number of set bits. The select support is provided by `sdsl::select_support_rrr` adding 64 bits and requiring $O(\log n)$ time for select queries.

sdsl::wt_int

The `sdsl::wt_int` is a class of the Succinct Data Structure Library (SDSL) that implements wavelet trees designed to efficiently handle sequences over large alphabets, such as integer sequences. It provides a space-efficient representation while supporting fast access, rank, and select operations. The wavelet tree is a balanced binary tree that recursively partitions the alphabet into two equal-sized subsets and encodes the sequence based on the partitioning [**grossi2003high**]. The `sdsl::wt_int` uses the RRR compressed bit vectors or other succinct representations for storing the bit vectors in each node of the wavelet tree. This makes the structure space-efficient.

In the case of RRR compressed bit vectors the space needed by `sdsl::wt_int` for a sequence of length n over an alphabet of size σ is $nH_0(S)+o(n \log \sigma)+\Theta(\sigma \log n)$ bits, where $H_0(S)$ is the zero-order empirical entropy of the sequence S. Also supports query access, rank and select operations in $O(\log \sigma)$ time.

4.1.2 Details of the XBWT Class Elements

The XBWT class utilizes several data structures from the SDSL library to efficiently represent and query the compressed data. Below are the details of the main elements used in the class:

- `sdsl::rrr_vector<> SLastCompressed`: This is a compressed bit vector that stores the S_{last} array of the XBWT.
- `sdsl::wt_int<sdsl::rrr_vector<» SAlphaCompressed`: This is a wavelet tree built on top of a compressed bit vector. The wavelet tree is used to compress and index the S_α array of the XBWT.
- `sdsl::rrr_vector<> SAlphaBitCompressed`: Another compressed bit vector used to store the additional bit of S_α needed to distinguish between internal and leaf nodes.

- `sdsl::rrr_vector<> ACompressed`: A compressed bit vector representing the A array of the XBWT used to in the F array of the XBWT.
- `sdsl::rrr_vector<>::rank_1_type SLastCompressedRank`: A rank support structure for the `SLastCompressed` bit vector, allowing efficient rank queries.
- `sdsl::rrr_vector<>::select_1_type SLastCompressedSelect`: A select support structure for the `SLastCompressed` bit vector, allowing efficient select queries.
- `sdsl::rrr_vector<>::rank_1_type ACompressedRank`: A rank support structure for the `ACompressed` bit vector.
- `sdsl::rrr_vector<>::select_1_type ACompressedSelect`: A select support structure for the `ACompressed` bit vector.
- `std::unordered_map<T, unsigned int> alphabetMap`: An hash map that maps each label in the alphabet to a unique integer.
- `unsigned int cardSigma`: The cardinality of the alphabet Σ.
- `unsigned int cardSigmaN`: The cardinality of the Σ_N alphabet. Where Σ_N is the set of labels that appear in the internal nodes of the labeled tree.
- `unsigned int maxNumDigits`: The maximum number of digits that has the integer code associated to the greater label in the alphabet (needed to sort the labels in the alphabet).

The overall space complexity of the XBWT class can be derived from the space complexity of the compressed data structures used in the class.

4.2 Construction of the XBWT

The construction of the XBWT is done by the constructor of the XBWT class. The constructor takes as input a generic labeled tree and constructs the compressed indexing scheme using the linear pathSort (also the naive construction method can be used by passing the boolean flag `usePathSort = false`). The construction process is divided into the following steps:

1. **Alphabet Encoding**: The first step is to encode the labels of the alphabet as integers. The labels are sorted in lexicographical order and assigned a unique integer code starting from 1 to $|\Sigma|$. Two hash maps are used to map each label to a unique integer and vice versa.
2. **Construct `intNodes` array**: The next step is to construct the `intNodes` array as described in the previous chapters. `intNodes` is an array of triplets of length t in which node is represented as a triplet containing the node's label, its level, and the index of its parent node in the array (from 1 to t, root has parent 0). The nodes are inserted in preorder traversal of the labeled tree.
3. **Sort `intNodes` array:** Call the `pathSort` or `upwardStableSortConstruction` (naive method) method to get the sorted array of nodes `intNodes`.

4. **Construct $S_{\textbf{last}}$ array**: Construct the S_{last} array by iterating over the sorted `intNodes` array.
5. **Construct S_α array**: Construct the S_α array by iterating over the sorted `intNodes` array, along with the additional bit array to distinguish between internal and leaf nodes.
6. **Construct A array**: Construct the A array by iterating over the sorted `intNodes` array.
7. **Construct rank and select support structures**: Construct the rank and select support structures for the compressed bit vectors.

4.3 Navigational Operations

The XBWT class provides several navigational operations to traverse the labeled tree and retrieve information about the nodes. The navigational operations implemented are:

- `getChildren(unsigned int i)`: This method returns a pair of integers representing the indices of the leftmost and rightmost children of the node at index `i`.
- `getRankedChild(unsigned int i, unsigned int k)`: This method returns the index of the `k`-th child of the node at index `i`.
- `getCharRankedChild(unsigned int i, T label, unsigned int k) const`: This method returns the index of the `k`-th child of the node at index `i` with the specified label.
- `getDegree(unsigned int i)`: This method returns the degree (number of children) of the node at index `i`.
- `getCharDegree(unsigned int i, T label)`: This method returns the number of children of the node at index `i` with the specified label.
- `getParent(unsigned int i)`: This method returns the index of the parent of the node at index `i`.
- `getSubtree(unsigned int i, unsigned int order = 0)`: This method returns a vector containing the labels of the nodes in the subtree rooted at index `i`. The `order` parameter specifies the traversal order (e.g., preorder, post-order).

All the methods refer to the index of the nodes in S_{last} and S_α arrays.

4.4 Search Operations

The XBWT class provides search operation `subPathSearch(const std::vector<T> &path)` that searches for a subpath in the XBWT structure. It uses the compressed vectors to determine the range of positions corresponding to the nodes whose upward path is prefixed by a given vector reversed.

Chapter 5

Experiments and Conclusions

The experiments have been run on a machine with an AMD Ryzen 9 5600Hs CPU with 24 GB of RAM. The results are shown in Table **??** and Table **??**. The source code for the experiments can be found in the `experiments.cpp` file.

5.1 Experiments

5.1.1 Construction Performance of the XBWT

To evaluate the performance of the implemented algorithms, we conducted a series of experiments on randomly generated trees created using the Python library `networkx`. The trees were generated with sizes ranging from 100 to 900,000 nodes. For each tree, we executed the construction algorithms 10 times, measuring the average execution time for both the linear *PathSort* (P.S.) algorithm and the naive *UpwardStableSort* (N.S.) algorithm used for constructing the XBWT. This approach allowed us to compare their performance across different tree sizes and assess their scalability.

The results are shown in Table **??**.

Nodes	**Depth**	**P.S. Time (s)**	**N.S. Time (s)**
100	22	0.002	0.001
500	45	0.004	0.002
1000	74	0.006	0.003
5000	175	0.028	0.015
10000	288	0.056	0.053
50000	486	0.31	0.35
100000	754	0.69	1.25
500000	2246	4.7	16.46
900000	2658	8.51	34.2

Table 5.1: Performance comparison between PathSort and Naive Sort algorithms.

5.1.2 Space Analysis of the XBWT

To evaluate the space savings achieved through XBWT compression, we conducted experiments on the same set of randomly generated trees used for the construction performance tests. For each tree, we compared the memory usage (in bytes) of three representations: the plain tree, the uncompressed XBWT, and the compressed XBWT.

The plain tree representation consists of the simple balanced parenthesis encoding of the tree structure combined with the edge labels. For example for tree in Figure **??**, the plain tree representation would be:

`(A(B(D(a))(a)(E(b)))(C(D(c))(b)(D(c)))(B(D(b)))).`

By *uncompressed XBWT*, we refer to the XBWT arrays S_{last} and S_α (including the additional bit) stored without any compression. Specifically, S_{last} is represented as a plain bitvector (`sdsl::bit_vector`), and S_α is stored as a wavelet tree (`sdsl::wt_int`) with plain bitvectors (`sdsl::bit_vector`). In contrast, the *compressed XBWT* representation stores S_{last} and S_A as compressed RRR bitvectors (`sdsl::rrr_vector`), and S_α as a wavelet tree with RRR bitvectors, as described in the previous chapter.

Table **??** reports the sizes (in bytes) for each representation of the trees across different sizes. The last column highlights the space savings achieved by the compressed XBWT compared to the plain tree representation, expressed as a percentage. These results illustrate the substantial space reductions achieved through compression, especially as the tree size increases.

Nodes	**Plain tree (B)**	**U. XBWT (B)**	**C. XBWT (B)**	**Saving (%)**
100	390	424	496	-27.18
500	2390	1112	1136	52.47
1000	4890	2242	2056	57.96
5000	28890	12911	10400	64
10000	58890	45625	21848	62.90
50000	338890	175146	123216	63.64
100000	688890	349478	259376	62.35
500000	3888890	1850850	1451570	62.67
900000	7088890	3480190	2718570	61.65

Table 5.2: Space analysis of the XBWT. Plain tree is the size in bytes of the tree in the simple balanced parenthesis representation plus the edge labels, U. XBWT is the size in bytes of the tree in the uncompressed XBWT, and C. XBWT is the size in bytes of the tree in the compressed XBWT. The last column shows the space-saving percentage between plain tree and compressed XBWT.

5.2 Conclusions

From the results shown in Table **??**, we can draw several conclusions about the performance of the PathSort (P.S.) algorithm compared to the Naive Sort (N.S.) algorithm and the space savings achieved by compressing the XBWT.

Firstly, the PathSort algorithm consistently outperforms the Naive Sort algorithm in terms of execution time, especially as the number of nodes increases. For smaller trees, the difference in execution time between the two algorithms is minimal. However, as the number of nodes grows, the PathSort algorithm demonstrates significantly better scalability. For instance, with 900,000 nodes, the PathSort algorithm takes 8.51 seconds, whereas the Naive Sort algorithm takes 34.2 seconds.

Secondly, the depth of the tree appears to increase with the number of nodes, which is expected in randomly generated trees. This increase in depth does not seem to adversely affect the performance of the PathSort algorithm as much as it does the Naive Sort algorithm.

For small trees, the compressed XBWT does not always provide immediate savings due to the overhead of succinct data structures. For instance, for 100 nodes, the compressed representation is larger than the plain tree, showing a -27.18% increase in space. However, as the number of nodes increases, the compression becomes more effective, achieving savings of over 60% for large trees.

The space reduction becomes particularly evident for trees with more than 500 nodes. These results confirm that the compressed XBWT provides a scalable and space-efficient alternative for storing and indexing labeled trees. The efficiency gains are particularly beneficial for applications requiring large-scale tree processing, such as bioinformatics and text indexing.

In conclusion, the PathSort algorithm is a more efficient choice for constructing the XBWT, especially for larger trees, and the compression method provides significant space savings, making the overall process more efficient in terms of both time and space.

5.2.1 Future Work

There are several directions for future work that could be explored to further improve the performance of the algorithm. One possible avenue is to investigate the impact of different tree structures on the performance of the algorithm. For instance, it would be interesting to see how the algorithm performs on trees with different branching factors or depths.

Also, the algorithm could be parallelized in order to take advantage of multicore processors and further improve the scalability of the algorithm. This could potentially reduce the execution time even further, especially for very large trees.