Università Ca'Foscari Venezia

Artificial Intelligence and Data Engineering

Master Degree Thesis

A New Compressing Technique for Labeled Trees

Professor
Nicola Prezza

Author
Davide Tonetto
Student ID 884585

Academic year
2024/2025

Abstract

Contents

Chapter 1

Introduction

1.1 Project overview

The increasing availability of large structured datasets, such as those found in XML documents, biological data, and hierarchical knowledge bases, has led to the need for efficient compression techniques for trees. Traditional compression methods, such as general-purpose text compression algorithms, often fail to exploit the hierarchical structure of trees effectively. Consequently, specialized tree compression techniques have been developed to address this issue.

Among the most prominent techniques for tree compression, the *Extended Burrows-Wheeler Transform* [6] extends the classical Burrows-Wheeler Transform to labeled trees, leveraging their structural properties to achieve significant compression. Another notable approach includes *Re-Pair-based compression* [17], which applies grammar-based compression to the tree structure.

Despite these advancements, existing techniques may not be optimal when dealing with trees characterized by a high degree of repetitiveness. Many real-world datasets, such as versioned documents or biological phylogenies, contain repeated substructures that can be exploited to achieve better compression. The aim of this thesis is to study a novel compression technique designed to efficiently handle such highly repetitive trees. We will implement and evaluate this method, comparing it with existing state-of-the-art approaches to determine its effectiveness in different scenarios.

1.2 State of the art

We begin with a brief overview of the state of the art for labeled tree compression. Subsequently, the next chapter will provide a detailed examination of the Extended Burrows-Wheeler Transform.

The Extended Burrows-Wheeler Transform [6] is a data structure designed for efficient compression and indexing of labeled trees. The XBWT works by linearizing a labeled tree into two arrays: one captures the structural properties of the tree and the other stores its labels. This transformation allows for efficient representation, navigation, and querying of the tree. The key advantage of the 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. The XBWT provides significant improvements in both compression ratio and query performance compared to traditional compression schemes, making it a valid resource for intensive applications.

Tree Re-Pair is a grammar-based compression technique specifically adapted for tree structures [17]. It extends the principles of the original Re-Pair algorithm introduced in [16], specifically designed for sequences and to handle the hierarchical nature of trees. The core idea of the tool is to identify frequently occurring patterns within the tree and represent them more compactly. The process involves the linearization of the tree (e.g., using a specific traversal order) and then the application of the Re-Pair logic. In this way, it finds the most frequent pair of adjacent elements (which could represent nodes, labels, or structural components depending on the linearization) in the sequence. The pair is then replaced by a new non-terminal symbol, and the corresponding production rule is added to a grammar. All this process is then repeated until no more pairs occur frequently enough or some other stopping criterion is met. The final output is a relatively small grammar (a set of production rules) and a sequence of symbols (including the newly introduced non-terminals) that can be used to reconstruct the original tree. An application of Tree Re-Pair to XML documents can be found in [18].

1.3 Challenges

In order to develop an effective tree compression scheme that can exploit repetitive structures, we need to address several key challenges:

- **Identification of repetitive structures:** The first step in compressing repetitive trees is to identify the repeated substructures efficiently. This requires the development of algorithms capable of detecting and representing these structures in a compact way.
- **Optimization of representation:** Once the repetitive structures have been identified, the challenge is to represent them in an optimized way that minimizes the overall size of the compressed tree. This involves finding the most efficient encoding for the repeated substructures.

We will address these challenges by developing a novel tree compression scheme that leverages the deterministic finite automata minimization algorithm to identify and compress repetitive structures in trees and the Minimum Weight Perfect Bipartite Matching algorithm to optimize their representation. Both algorithms are known for their efficiency and effectiveness in solving similar problems and are well-suited to the task at hand. They will be integrated into a comprehensive compression pipeline that can handle trees with repetitive structures efficiently.

1.4 The structure of the thesis

The thesis is structured as follows:

1. **Chapter 1** provides an overview of the project and its objectives.
2. **Chapter 2** contains the theoretical background on labeled trees necessary to understand the content of the subsequent chapters.
3. **Chapter 3** provides a detailed examination of the Extended Burrows-Wheeler

Transform and its implementation that we will use as a benchmark for comparison.

4. **Chapter 4** introduces the Hopcroft algorithm for minimizing deterministic finite automata and an algorithm derived from it for minimizing acyclic deterministic finite automata in linear time which we will use to identify repetitive structures in trees.
5. **Chapter 5** presents the concept of matching in bipartite graphs and the Minimum Weight Perfect Bipartite Matching problem, which we will use in our compression scheme to chain repetitive structures.
6. **Chapters 6** gives an overview of the proposed tree compression scheme, outlining the pipeline used to compress trees with repetitive structures.
7. **Chapter 7** describes how the tree compression problem can be reduced to the Minimum Weight Perfect Bipartite Matching problem, providing a formal definition of the problem and discussing its complexity.
8. **Chapter ??** describes the implementation details of the proposed method and the experimental setup used to evaluate its performance.
9. **Chapter ??** provides a detailed analysis of the experimental results, discussing the performance of the proposed method in various scenarios.
10. **Chapter ??** discusses the implications of our findings and outlines possible directions for future research.

Chapter 2

Theoretical Background on Labeled Trees

Before delving into specific compression techniques, it is essential to establish a solid theoretical foundation regarding labeled trees. These structures are fundamental for representing hierarchical data across diverse fields, from bioinformatics to document processing. This chapter provides the necessary background, defining labeled trees, exploring their common applications, and introducing the core concepts behind their compression and indexing. Understanding these principles, including the role of succinct data structures and the information-theoretic limits of compression, is crucial for appreciating the challenges and advancements in handling large-scale tree-structured data effectively, which forms the basis for the work presented in this thesis.

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, which extends the classical Burrows-Wheeler Transform (BWT) to handle labeled trees efficiently.

Before the advent of XBWT, Kosaraju [14] 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 [12] 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

The Extended Burros-Wheeler Transform

This chapter delves into the Extended Burrows-Wheeler Transform (XBWT) introduced Ferragina et al. [6], a prominent state-of-the-art technique for labeled tree compression. Understanding the principles and performance of XBWT is crucial as it will serve as the primary benchmark against which we will evaluate the novel compression scheme proposed in this thesis. By establishing a baseline with a well-regarded method like XBWT, we can effectively demonstrate the potential advantages and contributions of our new approach, particularly for trees exhibiting high repetitiveness.

Disclaimer: The fundamental definitions, properties, and algorithms related to the Extended Burrows-Wheeler Transform presented in this chapter are based on the work introduced by Ferragina et al. 'Compressing and Indexing Labeled Trees, with Applications' [6].

3.1 Introduction to XBWT

In 2005, Ferragina et al. [6] 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. Let's start by defining the XBWT.

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 chapter aims to explore the XBWT data structure and 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.

3.1.1 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.

3.1.2 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.

3.1.3 XBWT implementation overview

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.

3.2 Definition of XBWT

The **Extended Burrows-Wheeler Transform** is a data structure designed to efficiently compress and index *labeled trees*. Inspired by the classical Burrows-Wheeler Transform (BWT) [1] 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.*

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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 $S_{\text{last}}[j_1, j_2]$.

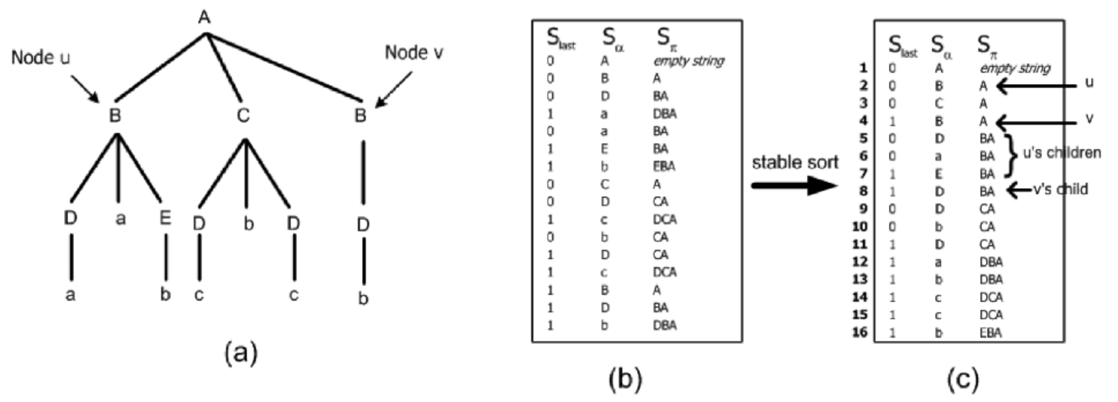
3.4 XBWT Construction

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. [6] 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 [13]. Let's see briefly how the Skew algorithm works.

3.4.1 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.



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.

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.

3.4.2 PathSort Algorithm

The pseudocode of the pathSort algorithm is shown in Algorithm 1. 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 [13], 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.
```

Recursive Step of PathSort

At each recursive step, the algorithm constructs the array `IntNodes` (as shown in Figure 3.1-(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.

3.5 Inverting the XBWT

Property 3 3.3.3 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 2 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 $i = 1, \dots, |\Sigma_N|$ 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 $i = 1, \dots, |\Sigma_N| - 1$ 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 $i = 1, \dots, t$ 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 $J$.
```

3.6 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.

3.7 Indexing a compressed labeled tree

In order to implement the efficient operations listed in 2.2 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 2 and 3.

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 3.1.
- **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 3.1.
- **GetDegree(i)**: returns the number of children of u.
- **GetCharDegree(i, c)**: returns the number of children of u labeled c.
- **GetParent(i)**: returns the position in 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 3.1.

- **GetSubtree(i)**: returns the node labels of the subtree rooted at u. Any possible order (i.e., pre, in, post) may be implemented.
- **SubPathSearch(P)**: determines the range $S[\text{First}, \text{Last}]$ of nodes, which are immediate descendants of each occurrence of the labeled path $P = c_1c_2\cdots c_k$ 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 3.1.

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 (3.3.3). 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 6 is based on the Property 3 (3.3.3) 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 $i \leftarrow 2, \ldots, k$ do
6:     $k_1 \leftarrow \mathrm{rank}_{c_i}(S_\alpha, First - 1)$; $z_1 \leftarrow \mathrm{select}_{c_i}(S_\alpha, k_1 + 1)$        ▷ first entry in $S_\alpha[First, t]$ labeled $c_i$
7:     $k_2 \leftarrow \mathrm{rank}_{c_i}(S_\alpha, Last)$; $z_2 \leftarrow \mathrm{select}_{c_i}(S_\alpha, k_2)$        ▷ 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:    $First \leftarrow$ GetRankedChild($z_1, 1$)        ▷ get the first child of $S[z_1]$
12:    $Last \leftarrow$ GetRankedChild($z_2$, 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.

3.8 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`.

3.8.1 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.

3.8.2 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` [9]. 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 [20].

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 [10]. 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.

3.8.3 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.

3.8.4 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.

3.8.5 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.

3.8.6 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.

3.9 Experiments

Davide T.: Questa sezione andrà riadattata una volta che avremo deciso quali esperimenti fare con l'altro algorimo

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 3.1 and Table 3.2. The source code for the experiments can be found in the `experiments.cpp` file.

3.9.1 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 3.1.

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 3.1: Performance comparison between PathSort and Naive Sort algorithms.

3.9.2 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 3.1, 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 3.2 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 3.2: 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.

3.9.3 Conclusions

From the results shown in Table 3.1, 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.

Chapter 4

Hopcroft's algorithm for Minimization of DFA

Tree compression schemes that effectively exploit repetitive structures require efficient techniques for identifying and representing such repetitions in a compact manner. In our approach, we leverage deterministic finite automata (DFA) minimization as a foundational tool for recognizing and compressing recurring patterns within trees. Among the various DFA minimization algorithms, Hopcroft's algorithm stands out due to its optimal time complexity and efficiency in reducing state redundancies. This chapter provides the necessary theoretical background on Hopcroft's algorithm and DFA minimization, explaining their relevance to our proposed compression scheme. Also, we introduce a linear-time algorithm for minimizing acyclic deterministic finite automata, which is particularly suitable for identifying repetitive structures in trees. Let's start by introducing what are deterministic finite automata and why minimizing them is important for tree compression.

4.1 Deterministic Finite Automata (DFA)

Definition 1 (Deterministic Finite Automaton). *A deterministic finite automaton (DFA) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where:*

- *Q is a finite set of states*
- *Σ is a finite set of input symbols (alphabet)*
- *$\delta : Q \times \Sigma \to Q$ is the transition function*
- *$q_0 \in Q$ is the initial state*
- *$F \subseteq Q$ is the set of final (accepting) states*

The DFA processes an input string s one symbol at a time by starting from the initial state q_0 and following transitions based on the input symbols. The string s is accepted if the DFA ends in an accepting state after processing all input symbols, otherwise, it is rejected. The language recognized by a DFA is the set of all strings that lead to an accepting state. DFA are widely used in various applications, including lexical analysis, pattern matching, and formal language theory.

4.1.1 DFA Minimization

The process of automata minimization consists in reducing the number of states in a DFA while preserving the language accepted by the DFA. The minimization of DFA is crucial for a variety of applications, such model checking, hardware design, and compilers, as it produces a more effective and compact representation of the automaton allowing for faster processing and reduced memory usage.

The minimization of DFA is a well-studied problem in automata theory, and there are several algorithms available for this purpose. One of the most popular algorithms for DFA minimization is the Hopcroft's algorithm, which was proposed by John Hopcroft in 1971 [11]. The Hopcroft's algorithm is an efficient and simple algorithm that can minimize a DFA in $O(n \log n)$ time, where n is the number of states in the DFA.

4.2 The Need for DFA Minimization in the novel scheme

A key challenge in tree compression is the identification of structurally similar or identical subtrees. By modeling repetitive substructures as states in a DFA, we can transform the problem into one of minimizing redundant states, thereby reducing the size of the representation. DFA minimization ensures that equivalent substructures are merged efficiently, leading to a more compact encoding.

The minimized DFA provides a canonical representation of the repetitive structures, which can then be leveraged in our compression pipeline. This theoretical foundation enables us to systematically identify and encode tree patterns, ultimately improving the compression efficiency.

In the subsequent sections, we delve into the formal definition of the Hopcroft's algorithm. Then, we introduce an algorithm for minimizing acyclic DFAs in linear time, which is particularly relevant for identifying repetitive structures in trees.

4.3 Hopcroft's Minimization Algorithm

Minimization of DFA is a classical and widely studied problem in automata theory and Formal Languages. It consists of finding the unique (up to isomorphism) finite automaton with the minimal number of states, recognizing the same regular language of a given DFA.

The algorithm is introduced in Algorithm **??** and works as follows:

1.

Alessio: You should also give an intuition/explanation on how it works with words

The algorithm enables to compute equivalence classes of nodes in $O(n \log n)$, in particular, the Myhill-Nerode equivalence classes **Alessio:** cite. The Myhill-Nerode theorem states that a language is regular if and only if it has a finite number of Myhill-Nerode equivalence classes. This theorem provides a powerful tool for determining the regularity of languages and is a cornerstone of automata theory. Let's formalize the concept of equivalence classes and the Myhill-Nerode theorem.

Definition 2 (Equivalence Relation)**.** *For a language $L \subseteq \Sigma^*$ and any strings $x, y \in \Sigma^*$, we say x is equivalent to y with respect to L (written as $x \approx_L y$) if and only if*

Algorithm 8 Hopcroft's Algorithm: DFA Minimization ($\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$)

```
1:  $\Pi \leftarrow \{F, Q \setminus F\}$
2:  for all $a \in \Sigma$ do
3:      $\mathcal{W} \leftarrow \{(\min(F, Q \setminus F), a)\}$ Alessio: how does min work given the two sets?
4:  end for
5:  while $\mathcal{W} \neq \emptyset$ do
6:      choose and delete any $(C, a)$ from $\mathcal{W}$
7:      for all $B \in \Pi$ do
8:          if $B$ is split from $(C, a)$ then
9:              $B' \leftarrow \delta_a^{-1}(C) \cap B$
10:             $B'' \leftarrow B \setminus \delta_a^{-1}(C)$
11:             $\Pi \leftarrow \Pi \setminus \{B\} \cup \{B', B''\}$
12:             for all $b \in \Sigma$ do
13:                 if $(B, b) \in \mathcal{W}$ then
14:                     $\mathcal{W} \leftarrow \mathcal{W} \setminus \{(B, b)\} \cup \{(B', b), (B'', b)\}$
15:                 else
16:                     $\mathcal{W} \leftarrow \mathcal{W} \cup \{(\min(B', B''), b)\}$
17:                 end if
18:             end for
19:         end if
20:     end for
21: end while
```

for all strings $z \in \Sigma^*$:

$$xz \in L \Leftrightarrow yz \in L$$

That is, strings x and y are equivalent if they have the same behavior with respect to the language L - either they both lead to acceptance or both lead to rejection when any suffix z is appended.

Theorem 2 (Myhill-Nerode theorem)**.** *Let L be a language over an alphabet Σ. Then L is regular if and only if there exists a finite number of Myhill-Nerode equivalence classes for L. Specifically, the number of equivalence classes is equal to the number of states in the minimal DFA recognizing L.*

Alessio: You should explain what does it mean for a language to be regular.

4.4 Minimization of acyclic DFA in linear time

For our purpose, we will focus on a specific type of finite automaton: an acyclic deterministic finite automaton (or DAWG). An acyclic DFA is one where there are no cycles within the transitions. This property simplifies the minimization process since it ensures that every state can be reached from the start state through a unique path.

Let's start by giving the notion of directed acyclic word graph (DAWG):

Definition 3 (DAWG)**.** *A* ***DAWG*** *(directed acyclic word graph) or automaton $\mathcal{A}$*

is defined by the following 5-uple:

$$\mathcal{A} = (Q, \Sigma, F, T, q_0),$$

where

- Q *is a set of states;*
- Σ *is an alphabet of finite cardinal denoted by* $|\Sigma|$*;*
- q_0 *is the initial state;*
- T *is the subset of terminal states of* Q*;*
- F *is a function of* $Q \times \Sigma$ *into* Q *defining the transitions (arcs) of the automaton.*

In this section, we will discuss an efficient algorithm for minimizing acyclic deterministic finite automata in linear time on the number of states [21]. The minimization process involves identifying and merging equivalent states. Two states are considered equivalent if they have the same set of reachable final states, meaning that from any state q, there is a path to a final state in both states. This equivalence relation partitions the DAWG into disjoint sets of states, each representing an equivalence class. The purpose of using this approach is then to apply it to the input tree for our pipeline since the problem can directly be applied to trees where the leaf nodes are considered as final states, the root as the initial state and the edges of the tree are considered directed from node to its children.

4.4.1 The minimization algorithm

Alessio: Section 2.3 is already called "minimization of...", consider removing this section title or rename one of the two. The minimization algorithm introduced in [21] operates by labeling each state with a unique identifier that represents the structure of the automaton from that state onward. It proceeds in the following steps:

1. **Height Computation:** The height of each state is determined, where the height of a state is the length of the longest path from that state to a final state.
2. **State Labeling:** Each state is labeled based on the structure of its transitions. The label consists of:
   - Whether the state is final or not.
   - The transitions, recorded as ordered pairs of symbols and target state identifiers.
3. **Lexicographic Sorting:** States at each height level are sorted lexicographically based on their labels using a bucket sort technique.
4. **Merging Equivalent States:** After sorting, states with identical labels are merged, ensuring that equivalent states are unified.

Chapter 5

Min-Weight Perfect Bipartite Matching

5.1 Problem definition

Given a weighted bipartite graph $G = (V, E)$ (remember that a bipartite graph is a graph whose vertices can be divided into two disjoint sets V_1 and V_2 such that every edge connects a vertex in V_1 to a vertex in V_2), let's define the concept of a matching.

Definition 4 (Matching)**.** *A matching $M \in E$ is a collection of edges such that every vertex of V is incident to at most one edge of M. In other words, a matching is a set of edges such that no two edges share a common vertex.*

If a vertex v has no edge of M incident to it then v is said to be exposed (or unmatched). A matching is perfect if no vertex is exposed; in other words, a matching is perfect if its cardinality is equal to $|V_1| = |V_2|$ [8].

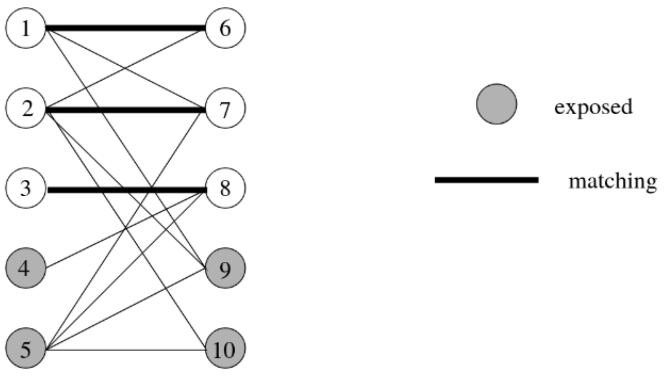


Figure 5.1: Example of a perfect matching in a bipartite graph.

The problem of finding a minimum weight perfect matching in a bipartite graph is a well-known problem in combinatorial optimization. The problem can be formulated as follows:

Definition 5 (Minimum weight perfect matching in bipartite graphs (MWPBM))**.** *Given a weighted bipartite graph $G = (V, E)$, where $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$, find a perfect matching M such that the sum of the weights of the edges in M is minimized. The weight of a matching is the sum of the weights of the edges in the matching. The weight of an edge $e = (u, v)$ is denoted by $w(e)$. This problem is also called **the assignment problem**.*

5.1.1 The existence of perfect matchings in bipartite graphs

In this subsection a theorem is introduced that states a condition for the existence of perfect matchings in bipartite graphs. This theorem will be useful in the following chapter to proof our reduction [19].

Let's start with the definition of the **Tutte matrix** of a bipartite graph.

Definition 6 (Tutte matrix). *The Tutte matrix of bipartite graph $G = (U, V, E)$ is an $n \times n$ matrix M with the entry at row i and column j*

$$M_{i,j} = \begin{cases} 0, & \textit{if } (u_i, u_j) \notin E \\ x_{i,j}, & \textit{if } (u_i, u_j) \in E \end{cases} \tag{5.1}$$

The determinant of the Tutte matrix is useful in testing whether a graph has a perfect matching or not, as the following theorem shows.

Theorem 3 (Existence of perfect matchings in bipartite graphs). *Given a bipartite graph G and the Tutte matrix M for G then the following equivalence holds:*

$$Det(M) \neq 0 \iff \textit{There exists a perfect matching in } G$$

Proof. We have the following expression for the determinant:

$$\text{Det}(M) = \sum_{\pi \in S_n} (-1)^{sgn(\pi)} \prod_{i=1}^{n} M_{i,\pi(i)}$$

where S_n is the set of all permutations on $[n]$, and $sgn(\pi)$ is the sign of the permutation π. There is a one-to-one correspondence between a permutation $\pi \in S_n$ and a (possible) perfect matching

$$\{(u_1, v_{\pi(1)}), (u_2, v_{\pi(2)}), \cdots, (u_n, v_{\pi(n)})\} \text{ in } G.$$

Note that if this perfect matching does not exist in G (i.e., some edge $(u_i, v_{\pi(i)}) \notin E$), then the term corresponding to π in the summation is 0. So we have

$$\text{Det}(M) = \sum_{\pi \in P} (-1)^{sgn(\pi)} \prod_{i=1}^{n} x_{i,\pi(i)}$$

where P is the set of perfect matchings in G. This is clearly zero if $P = \emptyset$, i.e., if G has no perfect matching. If G has a perfect matching, there is a $\pi \in P$ and the term corresponding to π is

$$\prod_{i=1}^{n} x_{i,\pi(i)} \neq 0.$$

Additionally, there is no other term in the summation that contains the same set of variables. Therefore, this term is not cancelled by any other term. So in this case, $\text{Det}(M) \neq 0$. □

5.1.2 Problem formulation

The problem of finding a minimum weight perfect matching in a bipartite graph can be formulated as an integer linear program (ILP), i.e.an optimization problem in which the variables are restricted to integer values and the constraints and the objective function are linear as a function of these variables. Given a matching M, let x be its incidence vector where $x_{ij} = 1$ if edge (i, j) is in the matching, and $x_{ij} = 0$ otherwise. Then, the problem can be formulated as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in E} w_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{j\in V_2} x_{ij} = 1, \quad \forall i \in V_1 \\
& \sum_{i\in V_1} x_{ij} = 1, \quad \forall j \in V_2 \\
& x_{ij} \in \{0,1\}, \quad \forall (i,j) \in E
\end{aligned} \tag{5.2}$$

Notice that any solution to this integer program corresponds to a matching and therefore this is a valid formulation of the minimum weight perfect matching problem in bipartite graphs.

The linear program relaxation of the above integer program is as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in E} w_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{j\in V_2} x_{ij} = 1, \quad \forall i \in V_1 \\
& \sum_{i\in V_1} x_{ij} = 1, \quad \forall j \in V_2 \\
& 0 \le x_{ij} \le 1, \quad \forall (i,j) \in E
\end{aligned} \tag{5.3}$$

The set of feasible solutions to the constraints in (P) forms a polytope. When optimizing a linear constraint over a polytope, the optimum will be achieved at one of the "corners" or extreme points of the polytope. An extreme point x of a set Q is an element $x \in Q$ that cannot be expressed as $\lambda y + (1 - \lambda)z$ with $0 < \lambda < 1$, $y, z \in Q$, and $y \neq z$. (This concept will be formalized and discussed in more detail when we cover polyhedral theory.)

In general, even if all the coefficients of the constraint matrix in a linear program are either 0 or 1, the extreme points of a linear program are not guaranteed to have all coordinates integral. This is not surprising since the general integer programming problem is NP-hard, while linear programming is solvable in polynomial time. Consequently, there is no guarantee that the value Z_{IP} of an integer program is equal to the value Z_{LP} of its LP relaxation. However, since the integer program is more constrained than the relaxation, we always have $Z_{IP} \geq Z_{LP}$, implying that Z_{LP} is a lower bound on Z_{IP} for a minimization problem. Moreover, if an optimal solution to a linear programming relaxation is integral, then it must also be an optimal solution to the integer program.

In our problem, the constraint matrix has a special form that lead to the following result:

Theorem 4. *Any extreme point of (P) is a* $0-1$ *vector and, hence, is the incidence vector of a perfect matching.*

Consequently, the polytope

$$\begin{aligned} P = \{x : \sum_{j\in V_2} x_{ij} &= 1, \quad \forall i \in V_1, \\ \sum_{i\in V_1} x_{ij} &= 1, \quad \forall j \in V_2, \\ 0 \le x_{ij} &\le 1, \quad \forall (i,j) \in E\} \end{aligned} \tag{5.4}$$

is called the bipartite perfect matching polytope.

5.2 Solutions to the problem

There are several algorithms to solve the problem of finding a minimum weight perfect matching in a bipartite graph. The first algorithm to solve this problem was proposed by Kuhn in 1955 [15]. The algorithm is based on the Hungarian method, which is a combinatorial optimization algorithm that solves the assignment problem in polynomial time. In the original paper the complexity of the algorithm was $O(n^4)$, but later Dinic and Kronrod [4] showed that the algorithm can be implemented in $O(n^3)$ time.

The Hungarian method is a powerful algorithm, however, the algorithm is not very intuitive and can be difficult to implement. In recent years, several other algorithms have been proposed to solve the problem of finding a minimum weight perfect matching in a bipartite graph. In 1970, Edmonds and Karp [5] proposed an algorithm that solves the problem in $O(nm + n^2 \log n)$ time. In 1989 Gabow and Tarjan [7] proposed an algorithm that solves the problem in $O(\sqrt{n}m \log(nW))$ time, where n, m and W denote the number of vertices, number of edges, and largest magnitude of a cost; costs are assumed to be integral. The algorithms work by scaling. Lastly, in 2009, Sankowski and Piotr [22] introduced a randomized algorithm that solves the problem in $O(Wn^w)$ time, where w is the exponent of matrix multiplication, and W is the highest edge weight in the graph.

In 2022, Chen, Li, et al. [2] proposed a new solution to the Minimum-Cost Flow problem that woks in almost-linear time, precisely in $O(m^{1+o(1)})$ time. The minimum-cost flow problem is a classic combinatorial graph problem that find numerous applications in engineering and scientific computing. This result is important also for our problem, since the maximum weight perfect matching problem can be reduced to the minimum-cost flow problem, allowing to solve the problem in almost-linear time.

5.3 Implementation used in this work

In this section, we will present an implementation of the Gabow and Tarjan algorithm to solve the problem of finding a minimum weight perfect matching in a bipartite graph. The algorithm is based on scaling and is a generalization of the Hungarian method. The algorithm works by scaling the edge weights and then finding a perfect matching in the scaled graph.

Chapter 6

Tree compression scheme focused on repetitive structures

As introduced in the first chapter of this thesis, the main goal of this work is to develop a new tree compression scheme that is able to exploit the presence of repetitive structures in the input tree. The idea is to design a compression algorithm that is able to identify and compress the repetitive parts of the input tree, while still being able to represent the non-repetitive parts of the tree in a compact way. The main motivation behind this work is to improve the compression performance of tree compression algorithms when dealing with trees that contain repetitive structures. In this chapter, we provide an overview of the proposed tree compression scheme.

6.1 The novel compression scheme

Let T be an ordered tree of arbitrary fan-out, depth, and shape. T consists of n internal nodes and ℓ leaves, for a total of $t = n + \ell$ nodes. Every node of T is labeled with a symbol drawn from an alphabet Σ. We assume that Σ is the set of labels effectively used in the nodes of T and that these labels are encoded with the integers in the range $[1, |\Sigma|]$. Then we need to define the array π where, for each node u, $\pi(u)$ is the string obtained by concatenating the labels on the **upward path** from the parent of u to the root of the tree (root has an empty π component).

The following pipeline is used to compress the tree T:

1. Initially the array π is computed for the tree T by traversing the tree in a preorder fashion. Then the nodes are stably sorted by the lexicographic order of their π strings. In order to sort the nodes, the **Path Sort** algorithm introduced in [6] is used, allowing to sort nodes in linear time and $O(t \log t)$ space. An implementation of this algorithm in C++ is available in the author's GitHub in the following repository: `https://github.com/davide-tonetto-884585/XBWT`.

2. Then, using a variant of the Hopcroft algorithm for minimization of DFA [11] that works over directed acyclic graphs [21] and so over trees, the nodes are partitioned into equivalence classes where two nodes are equivalent (has the same class) if they have the same subtree rooted at them.

3. Given a width p, the nodes previously sorted are then divided into p chains with the aim of minimizing the run-length encoding of each chain (by considering the equivalence classes). In order to do so we reduced this problem to the Minimum Perfect Bipartite Matching problem, which can be solved in polynomial time as introduced in [2] and [22].

4. Lastly, the resulting deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA) can be indexed using the indexing scheme introduced by Cotumaccio et al. [3]. Also, the chains may be compressed using some techniques such as run-length encoding, Huffman encoding, and Elias-Fano encoding.

Chapter 7

Reduction to the assignment problem

7.1 Introduction

In this chapter, we will show how we can reduce the problem of finding the optimal partition of the nodes of a tree T given their equivalence classes into p with $p \leq |E| \leq t$ chains to the Minimum Weight Perfect Bipartite Matching problem, where E is the set of equivalence classes of the nodes of T and t is the number of nodes of the tree. This reduction will allow us to solve the problem in polynomial time as shown in the previous chapter.

Then we will show how to optimize the reduction by introducing some constraints that will allow us to reduce the number of edges in the bipartite graph, and we will also show how to move from the Minimum Weight Perfect Bipartite Matching problem to the more studied Maximum Weight Perfect Bipartite Matching problem without losing generality.

7.2 The reduction

7.2.1 Bipartite graph construction

Let T be a tree with t nodes and p, which is the number of chains we want to partition the nodes into. Let E be the set of equivalence classes of the nodes of T. We can construct a bipartite graph $G = (V, E)$ such that vertices are divided in two disjoint sets $V = V_1 \cup V_2$ in the following way:

Definition 7. *The two sets V_1 and V_2 of the bipartite graph G are constructed in the following way:*

- *V_1 contains $t + p$ nodes composed by p source nodes $s_1, s_2, \ldots, s_p$ and the t elements of E ordered.*
- *V_2 contains $t + p$ nodes composed by the t elements of E ordered and p destination nodes $d_1, d_2, \ldots, d_p$.*

Then the edges of the graph G will be constructed in the following way:

1. *The sources nodes $s_i \in V_1$ for $i = 1, 2, \ldots, p$ are connected to the first p nodes with distinct equivalence class in V_2 with weight* 1.
2. *Each of the t nodes of the tree in V_1 is connected to the first p (at most) nodes with distinct class in V_2 (and without the same class of the considered node) coming after it in the ordering of the nodes with weight* 1.

3. *Each of the t nodes of the tree in V_1 is also connected to the first node with its same class in V_2 coming after it in the ordering of the nodes of t with weight 0 iff there is one, otherwise p edges with weight 0 are added to each of the destination nodes $d_i \in V_2$ for $i = 1, 2, \ldots, p$.*

Notice that it is important to consider the order of the nodes of the two sets V_1 and V_2 as stated above, because we will need to connect the source nodes to the destination nodes in a way that will allow us to find the optimal partition of the nodes of the tree. In Figure 7.4 the nodes are ordered from top to bottom. An example of the node structure is shown in Figure 7.1.

Notice also that when we talk about the same V_1 node placed in V_2 we are referring to the corresponding node in V_2 that derives from the same node in the original tree T since the nodes of the tree are placed ordered in both sets V_1 and V_2. In Figure 7.1, 7.4 and 7.2 the node's correspondence is achieved by putting the two nodes at the same level.

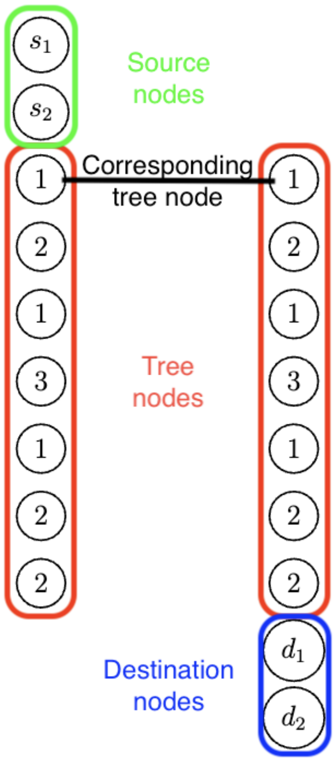


Figure 7.1: Example of a bipartite graph nodes constructed from a tree with the following equivalence classes $E = \{1, 2, 1, 3, 1, 2, 2\}$. The nodes are ordered from top to bottom.

Definition 8 (Bipartite graph properties)**.** *The resulting bipartite graph G will have $2t + 2p$ nodes and $O(t(p + 1) + p^2 + tp)$ edges, where the $O(t(p + 1))$ edges come from the tree nodes, the $O(p^2)$ edges come from the sources since each source node is connected to p nodes, and the $O(tp)$ edges come from the destination nodes since in the worst case we have t distinct equivalence class and so all the nodes are connected to the destination nodes. The weight of the edges will be 0 or 1.*

Let's see a small example for each case, consider $p = 2$. In Figure 7.2-(a) there is an example for the sources' edges, as stated before, for each source p nodes with weight

1 are created and connected to the first p nodes with distinct equivalence class in V_2.

In Figure 7.2-(b) there is an example for the tree nodes' edges, for each node in the tree T edges with weight 1 are created and connected to the first p nodes with distinct equivalence class in V_2 after the corresponding node in V_2 (coming after the node itself in the ordering), and edges with weight 0 are created and connected to the first node with the same class in V_2 after the corresponding node in V_2. As we can see from the image, we consider the first node in V_1 labelled 1 that is connected to the node 2 with weight 1 and to the node 3 with weight 1, and to the second node labelled 1 in V_2 with weight 0.

Lastly, in Figure 7.2-(c) there is an example for the destination nodes' edges, we start by considering the first node in V_1 that is labelled 1, it is connected to the node 2 with weight 1, then since there is no node with the same class in V_2 we connect it to the destination nodes d_1 and d_2 with weight 0. The same is done for the second node in V_1 that is labelled 2 since no nodes are coming after it in the order it is connected to the destination nodes d_1 and d_2 with weight 0.

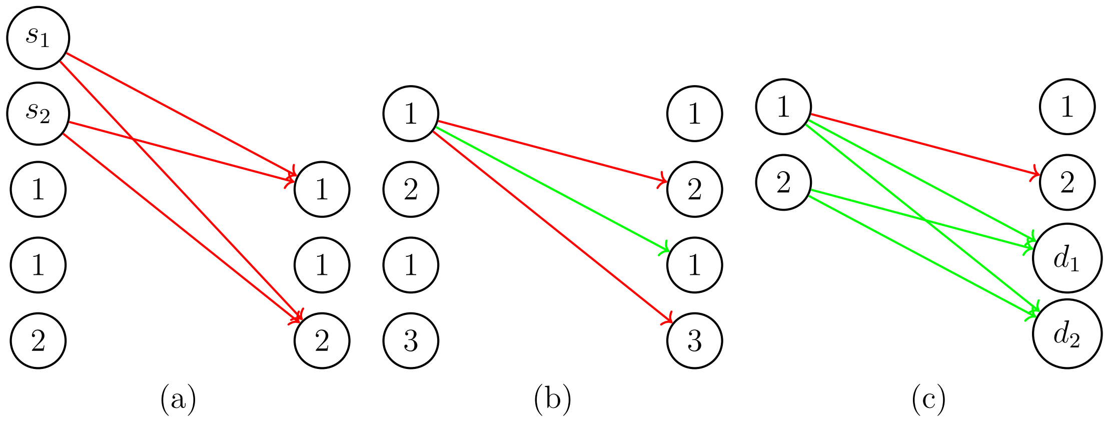


Figure 7.2: Considering $p = 2$ these three examples show how to connect nodes in the bipartite graph in the case of sources (a), tree nodes (b), and destinations (c).

7.2.2 Proof of correctness

In order to prove the correctness of the reduction, it is important to start by defining the problem we want to solve.

Definition 9 (CHAINS-DIVISION problem). *Given a tree T with t nodes, and given the equivalence classes E coming from the Hopcroft algorithm applied to the tree T, the sorted order of the nodes in T according to the upward path π, and the number of chain $p \leq t$, we want to find the optimal partition of the nodes of T into p chains with $p \leq t$ such that the run length encoding of each chain is minimized.*

Let's give a formal definition of run length encoding.

Definition 10 (Run length encoding). *Given a sequence $S = \{s_1, s_2, \ldots, s_n\}$, the run length encoding of S is the sequence $R = \{r_1, r_2, \ldots, r_m\}$ where r_i is the number of times the element s_i is repeated in S. It allows us to represent the sequence S in a more compact way.*

So, we aim to divide the nodes of the tree into p chains such that the run length encoding of the chains is minimized meaning that we want to minimize the number of distinct equivalence classes in each chain. Follows the definition of chain.

Definition 11 (Chains)**.** *Given a tree T with t nodes and p chains, a chain C is a sequence of nodes $C = \{c_1, c_2, \dots, c_m\}$ such that c_i is a node of T for $i = 1, 2, \dots, m$ and $m \le t$. Also, Each node of T is in exactly one chain and the nodes of the chain are ordered according to the upward path π of the tree.*

Let's start by stating the following lemma.

Lemma 1. *Exactly $|E|$ nodes of the tree T of the set V_1 are connected each to all the destination nodes $d_i \in V_2 \quad \forall i = 1, 2, \dots, p$ with weight 0. Where E is the set of equivalence classes of the nodes of T coming from the Hopcroft algorithm applied to the tree T.*

Proof. Since the destination nodes $d_i \in V_2$ are connected to the nodes of the tree T with weight 0 only if there is no other node with the same class in V_2 coming after the node in the ordering, then for sure there are $|E|$ nodes of the tree T in V_1 that have no other node with the same class in V_2 coming after the node in the ordering. □

Lemma 2. *The optimal solution of the CHAINS-DIVISION problem for an instance $\mathcal{I}$ for a tree T is always greater or equal to the number of equivalence classes coming from the Hopcroft algorithm applied to the tree T.*

Proof. Since we aim to minimize the run length encoding of the chains, and the minimum cost of a chain is 1, then the optimal cost of the *CHAINS-DIVISION* problem for the tree T is always greater or equal to the number of equivalence classes since if we dispose them in $|E|$ chains we will have a cost of $|E|$ since each chain contains only nodes with the same class, and if we dispose them in $p < |E|$ chains we will have a cost greater or equal to $|E|$ since we will have to put at least two nodes with the same class into one chain or more than one. □

Lemma 3. *The solutions for the CHAINS-DIVISION problem for the instances where the number p of chains is greater than $|E|$ are not better than the solutions for the instances where $p \le |E|$.*

Proof. The proof comes directly from lemma 2. □

Lemma 4. *A perfect matching for a bipartite graph G constructed as stated in Definition 7 exist for each possible instance of the CHAINS-DIVISION problem.*

Proof. The proof comes from the construction of the bipartite graph G and from theorem 3. We are going to proof that the bipartite graph G constructed as stated in Definition 7 has a Tutte matrix (definition 6) with determinant different from 0 and so a perfect matching for G exists.

We know that a $n \times n$ matrix M has $Det(M) \neq 0$ if and only if it has full rank ($rank(M) = n$), or equivalently if it has n linearly independent rows or columns. We can see that the bipartite graph G has $2t + 2p$ nodes and $O(t(p + 1) + p^2 + tp)$

$$\begin{array}{c|ccccccccc}
 & 1 & 2 & 1 & 3 & 1 & 2 & 2 & d_1 & d_2 \\
\hline
s_1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}$$

Figure 7.3: Example of a Tutte matrix for a bipartite graph in figure 7.4-(a). As we can see the matrix has full rank and so a perfect matching exists.

edges, and so the Tutte matrix of G will have $2t + 2p$ rows and $2t + 2p$ columns. The columns of M are all independent since each node of G in V_1 is connected only to nodes in V_2 that are greater than u in the ordering. Also each node is connected to at least one node in V_2 and at most $p + 1$ distinct nodes with distinct class in V_2. Those conditions on the edges are sufficient to get a full rank matrix and so a perfect matching for G exists. □

In figure 7.3 the Tutte matrix for the bipartite graph in Figure 7.4-(a) is shown.

Lemma 5. *Given a bipartite graph G constructed as stated in Definition 7 from a tree T, for each node $u \in V_1$ coming from the nodes of T it is impossible for u to be connected to node $v \in V_2$ coming from the nodes of T such as $v < u$ in the order of the nodes of the tree T.*

Proof. The proof comes from the construction of G where the nodes of V_1 are always connected to the nodes of V_2 coming after them in the ordering of the nodes of the tree T. □

Now we can prove the correctness of the reduction.

Theorem 5. *An optimal solution of an instance $\mathcal{I}$ with $p \leq |E|$ of the problem defined in Definition 9 is equivalent to an optimal solution of the Minimum Weight Perfect Bipartite Matching for the instance $r(\mathcal{I})$ where $r : \mathcal{I}_{CHAINS-DIVISION} \to \mathcal{I}_{MWPBM}$ is the reduction function that maps an instance of the problem defined in Definition 9 to an instance of the Minimum Weight Perfect Bipartite Matching problem defined in Definition 5 constructed as stated in Definition 7.*

Proof. Let $p \leq t$ be the number of chains we want to partition the nodes of a given tree T into. Let E be the set of equivalence classes of the nodes of T, from lemma 3 we consider that $p \leq |E|$. We can construct a bipartite graph $G = (V, E)$ such that vertices are divided in two disjoint sets $V = V_1 \cup V_2$ as stated in Definition 7. We will show that the optimal solution of the Minimum Weight Perfect Bipartite Matching problem for the graph G is equivalent to the optimal solution of the CHAINS-DIVISION problem for the tree T.

From definition 7 we know that V_1 contains $t+p$ nodes including $s_1, s_2, \ldots, s_p$ and the t elements of E ordered, and V_2 contains $t+p$ nodes including the t elements of E ordered and $d_1, d_2, \ldots, d_p$.

The source nodes are necessary to distinguish the chains, by starting from each source node we can follow the edges of the nodes and reach the destination nodes in order to retrieve the nodes of a chain. The source nodes are all connected to the first p nodes with distinct equivalence class in V_2 with weight 1 since each chains has at least one node and a minimum cost of 1.

The destination nodes are necessary to close the chains, by connecting the nodes of the tree to the destination nodes we can ensure that the chains are closed and allows to get a perfect matching otherwise we would have $|V_1| > |V_2|$ and so we would have unmatched nodes. From lemma 1 we know that exactly $|E|$ nodes of the tree T in V_1 are connected to all the destination nodes $d_i \in V_2 \quad \forall i = 1, 2, \ldots, p$ with weight 0. This is done in order to ensure that the chains are closed and that the nodes of the tree are connected to the destination nodes in order to have a perfect matching. The weight of the edges is 0 since if we conclude a chain no additional cost is needed.

We know from definition 11 that the order of the nodes of each chain must follow the order given by the upward path π of each node in the tree T. This is ensured by lemma 5 that states that it is impossible for a node $u \in V_1$ coming from the nodes of T to be connected to a node $v \in V_2$ coming from the nodes of T such as $v < u$ in the order of the nodes of the tree T. This is done in order to ensure that the nodes are connected in the right order.

Given the construction of the bipartite graph G we can see that the optimal solution of the Minimum Weight Perfect Bipartite Matching problem for the graph G is equivalent to the optimal solution of the CHAINS-DIVISION problem for the tree T since the chains are closed and the nodes are connected in the right order. The weight of the edges 0 allow the problem to chain subsequent nodes with the same class without adding additional cost. Also from the definition of matching (4) no node of G is exposed and so the matching is perfect as proved in lemma 4. The fact that each node in V_1 is connected to at least p edges allow the problem to compare all the possible chains and so to find the optimal solution. □

7.2.3 Full example

Consider The example in Figure 7.4 where we have a tree T with $t = 7$ nodes, $p = 2$ chains and the equivalence classes $E = \{1, 2, 1, 3, 1, 2, 2\}$ sorted accordingly to the upward path π of each node of the tree. We can construct the two distinct sets V_1 and V_2 of the bipartite graph G as follows: $V_1 = \{s_1, s_2, 1, 2, 1, 3, 1, 2, 2\}$ and $V_2 = \{1, 2, 1, 3, 1, 2, 2, d_1, d_2\}$. The edges of the graph G will be constructed as stated in Definition 7. In Figure 7.4-(a) we have the resulting bipartite graph, and in Figure 7.4-(b) we have one of the possible minimum perfect matchings for the graph in (a) having weight 4. A the end we can see that the optimal partition of the nodes of the tree T is $C_1 = \{1, 1, 1, 2, 2\}$ and $C_2 = \{2, 3\}$ with a total cost of 4, this can be obtained starting from the sources and by following the edges of the nodes, jumping to the corresponding node in V_1 and following the edges again until we reach the destination nodes.

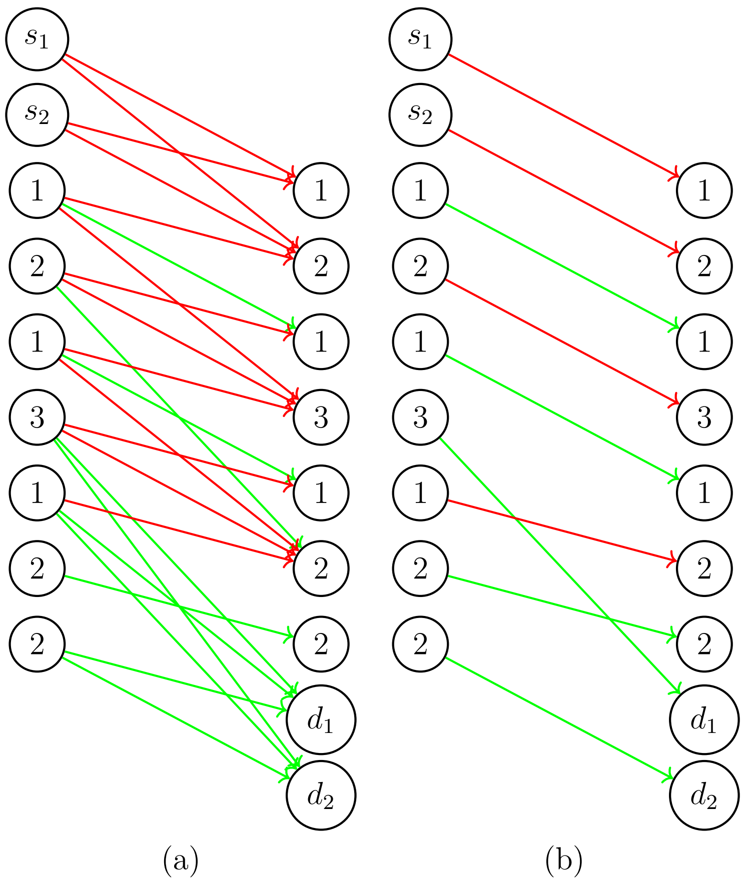


Figure 7.4: Example of a reduction for the sorted nodes' equivalency classes $E = \{1, 2, 1, 3, 1, 2, 2\}$. In (a), we have the resulting bipartite graph constructed from E. In (b), we have the resulting perfect matching for the graph in (a) having weight 4. Green edges weigh 0, while red edges weigh 1.

7.3 Heuristics and Improvements

Some changes can be made to the reduction in order to optimize it and to reduce the number of edges in the bipartite graph. Here are some of the improvements that can be made.

- **Sources' edges optimization**:

  Lemma 6. *The sources' edges can be optimized by connecting each source only to the smaller node (considering the order of the nodes) in V_2 coming from the tree T that is not connected to any other source.*

  Proof. Since the source nodes are needed to distinguish the chains as starting points, we need that each source is connected to at least one node in V_2 coming from the tree T. Having the sources connected to the first p nodes with distinct equivalence class in V_2 is not necessary since allows us just to invert the chains starting from each source and so it is redundant. We can connect each source to the smaller node in V_2 coming from the tree T that is not connected to any other source since we need to connect each source to at least one node in V_2 coming from the tree T and this will allow us to distinguish the chains. □

  This will reduce the number of edges coming from the sources from $O(p^2)$ to $O(p)$. In Figure 7.5-(a) the removed edges are shown in green.

- **Tree nodes' edges optimization 1**:

  Lemma 7. *The tree nodes' edges can be optimized by removing the edges of tree nodes that are connected to nodes in V_2 already linked to a source node in V_1.*

  Proof. From definition 4 we know that a matching $M \in E$ is a collection of edges such that every vertex of V is incident to at most one edge of M. In other words, a matching is a set of edges such that no two edges share a common vertex. Given that, in all the solutions to the problem all sources will be connected to exactly one node in V_2 coming from the tree T and so we can remove the edges of the tree nodes that are connected to nodes in V_2 already linked to a source node in V_1 since they will not be part of the final matching. □

  This will reduce the number of edges by a factor of $O(p-1)$. In Figure 7.5-(a) the removed edges are shown in blue.

- **Tree nodes' edges optimization 2**:

  Lemma 8. *The tree nodes' edges can be optimized by removing the edges with weight* 1 *starting from a node $u \in V_1$ to a node $v \in V_2$ if the node u has another edge with weight* 0 *connected to a node $z \in V_2$ such that $z < v$ in the ordering of the nodes.*

  Proof. To proof the lemma since every time we have an edge with weight 0 between two nodes of V_1 and V_2 it means that those two nodes have the same equivalence class and so there is no need to add additional cost trying to connect that node to nodes with different classes coming after in the ordering, as that would only increase the cost without providing any benefit to the solution. Also, we know that connecting nodes with the same class is always the best choice for optimizing the run length encoding of each chain. □

  In Figure 7.5-(a) the removed edges are shown in red.

In figure 7.5-(b) we can see the resulting bipartite graph for the example shown in previous section after the optimizations.

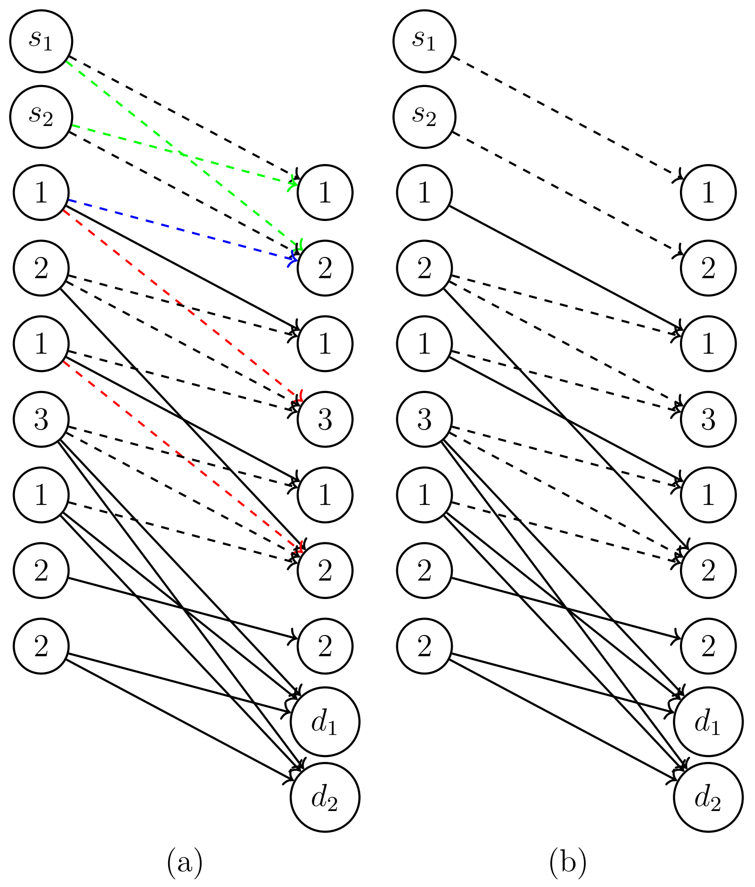


Figure 7.5: Example of a reduction for the sorted nodes' equivalency classes $E = \{1, 2, 1, 3, 1, 2, 2\}$ applying also the heuristics showed. In (a), the edges removed are shown in green for lemma 6, blue for lemma 7, and red for lemma 8. In (b), we have the resulting bipartite graph after the heuristics applied. Dashed edges weigh 1, while solid edges weigh 0.

7.4 Moving to Maximum weight perfect bipartite matching

In this section, we will discuss how to slightly modify the reduction process to move from a minimum weight perfect bipartite matching problem to a maximum weight perfect bipartite matching problem. This will be helpful in solving the problem more efficiently by using some known algorithms to solve the maximum weight perfect bipartite matching problem.

Theorem 6. *An optimal solution of an instance $\mathcal{I}$ with $p \leq |E|$ of the CHAINS-DIVISION problem is equivalent to an optimal solution of the Maximum Weight Perfect Bipartite Matching for the instance $r(\mathcal{I})$ where $r : \mathcal{I}_{CHAINS-DIVISION} \to \mathcal{I}_{MWPBM}$ is the reduction function that maps an instance of the CHAINS-DIVISION problem to an instance of the Maximum Weight Perfect Bipartite Matching problem constructed as stated in Definition 7 but with inverted weights (weight 0 becomes 1 and weight 1 becomes 0).*

Proof. Let M be a perfect matching in the bipartite graph G constructed as stated in Definition 7. Let $w(M)$ be the sum of the weights of the edges in the matching M. From the previous theorem, we know that the optimal solution of the

CHAINS-DIVISION problem is equivalent to finding a perfect matching M in G that minimizes $w(M)$.

Let G' be a bipartite graph constructed as G but with inverted weights (weight 0 becomes 1 and weight 1 becomes 0). Let M' be a perfect matching in G' and let $w'(M')$ be the sum of the weights of the edges in the matching M'. Let k be the number of edges in the matching.

We can see that for any matching M in G:

$$w'(M) = k - w(M)$$

This means that maximizing $w'(M)$ is equivalent to minimizing $w(M)$. Therefore, finding the maximum weight perfect matching in G' is equivalent to finding the minimum weight perfect matching in G, which in turn is equivalent to finding the optimal solution of the CHAINS-DIVISION problem. $\square$

Bibliography

[1] M. Burrows and D. Wheeler. *A block-sorting lossless data compression algorithm.* Tech. rep. 124. Digital Equipment Corporation, 1994 (cit. on p. 11).

[2] Li Chen et al. "Maximum flow and minimum-cost flow in almost-linear time". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS).* IEEE. 2022, pp. 612–623 (cit. on pp. 35, 38).

[3] Nicola Cotumaccio et al. "Co-lexicographically ordering automata and regular languages-Part I". In: *Journal of the ACM* 70.4 (2023), pp. 1–73 (cit. on p. 39).

[4] EA Dinic and MA Kronrod. "An algorithm for the solution of the assignment problem". In: *Soviet Math. Dokl.* Vol. 10. 6. 1969, pp. 1324–1326 (cit. on p. 35).

[5] Jack Edmonds and Richard M Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264 (cit. on p. 35).

[6] Paolo Ferragina et al. "Compressing and Indexing Labeled Trees, with Applications". In: *Journal of the ACM* 57 (2009). DOI: `10.1145/1613676.1613680` (cit. on pp. 1, 9, 12, 38).

[7] Harold N Gabow and Robert E Tarjan. "Faster scaling algorithms for network problems". In: *SIAM Journal on Computing* 18.5 (1989), pp. 1013–1036 (cit. on p. 35).

[8] Michel Goemans. *Advanced Algorithms: Matching Notes.* Lecture notes, Massachusetts Institute of Technology. 2009. URL: `https://math.mit.edu/~goemans/18433S09/matching-notes.pdf` (cit. on p. 32).

[9] Simon Gog et al. "From Theory to Practice: Plug and Play with Succinct Data Structures". In: *13th International Symposium on Experimental Algorithms, (SEA 2014).* 2014, pp. 326–337 (cit. on p. 20).

[10] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. "High-order entropy-compressed text indexes". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* USA: Society for Industrial and Applied Mathematics, 2003, pp. 841–850 (cit. on p. 21).

[11] John Hopcroft. "AN n log n ALGORITHM FOR MINIMIZING STATES IN A FINITE AUTOMATON". In: *Theory of Machines and Computations.* Ed. by Zvi Kohavi and Azaria Paz. Academic Press, 1971, pp. 189–196. ISBN: 978-0-12-417750-5. DOI: `https://doi.org/10.1016/B978-0-12-417750-5.50022-1`. URL: `https://www.sciencedirect.com/science/article/pii/B9780124177505500221` (cit. on pp. 28, 38).

[12] Guy Jacobson. "Space-efficient static trees and graphs". In: *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS).* Los Alamitos, CA: IEEE Computer Society Press, 1989, pp. 549–554 (cit. on p. 7).

[13] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. "Linear work suffix array construction". In: *Journal of the ACM* 53.6 (2006), pp. 918–936. DOI: `10.1145/1217856.1217858` (cit. on pp. 12, 14).

[14] S. R. Kosaraju. "Efficient tree pattern matching". In: *Proceedings of the 20th IEEE Foundations of Computer Science (FOCS).* 1989, pp. 178–183 (cit. on p. 6).

[15] Harold W Kuhn. "The Hungarian method for the assignment problem". In: *Naval research logistics quarterly* 2.1-2 (1955), pp. 83–97 (cit. on p. 35).

[16] N. Jesper Larsson and Alistair Moffat. "Off-line dictionary-based compression". In: *Proceedings DCC 2000. Data Compression Conference*. IEEE. 2000, pp. 296–305 (cit. on p. 2).

[17] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. "Tree structure compression with repair". In: *2011 Data Compression Conference*. IEEE. 2011, pp. 353–362 (cit. on pp. 1, 2).

[18] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. "XML tree structure compression using RePair". In: *Information Systems* 38.8 (2013), pp. 1150–1167 (cit. on p. 2).

[19] Viswanath Nagarajan. *Finding Perfect Matchings*. Lecture notes. 2004. URL: `https://www.cs.cmu.edu/afs/cs/academic/class/15859-f04/www/scribes/lec3.pdf` (cit. on p. 33).

[20] Rajeev Raman, V. Raman, and S. Srinivasa Rao. "Succinct Indexable Dictionaries with Applications to representations of k-ary trees and multi-sets". In: *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2002 (cit. on p. 21).

[21] Dominique Revuz. "Minimisation of acyclic deterministic automata in linear time". In: *Theoretical Computer Science* 92.1 (1992), pp. 181–189 (cit. on pp. 30, 38).

[22] Piotr Sankowski. "Maximum weight bipartite matching in matrix multiplication time". In: *Theoretical Computer Science* 410.44 (2009), pp. 4480–4488 (cit. on pp. 35, 38).

Web bibliography

[8] Michel Goemans. *Advanced Algorithms: Matching Notes*. Lecture notes, Massachusetts Institute of Technology. 2009. URL: `https://math.mit.edu/~goemans/18433S09/matching-notes.pdf` (cit. on p. 32).

[19] Viswanath Nagarajan. *Finding Perfect Matchings*. Lecture notes. 2004. URL: `https://www.cs.cmu.edu/afs/cs/academic/class/15859-f04/www/scribes/lec3.pdf` (cit. on p. 33).