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. Trees are a natural choice for representing hierarchical data due to their ability to model parent-child relationships and nested structures. For instance, an XML document is inherently a tree, where tags are nested to create a structured hierarchy. Similarly, file systems are organized as trees of directories and files, and biological data, such as phylogenetic trees, use this structure to represent evolutionary relationships. Given their ubiquity in representing complex data, developing effective compression methods for trees is of paramount importance. Traditional compression methods, such as general-purpose text compression algorithms, often fail to effectively exploit the hierarchical structure of trees. 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* [7] 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* [18], 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. This thesis aims to study a novel compression technique designed to efficiently handle such highly repetitive trees. We implement and evaluate this method, comparing it with existing state of the art approaches to determine its effectiveness in different scenarios.

1.2 Background and State of The Art

Before the advent of the XBWT, Kosaraju [15] 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.

The Extended Burrows-Wheeler Transform (XBWT) [7] is a data structure designed for efficient compression and indexing of ordered node-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.

Another notable approach is Tree Re-Pair [18], a grammar-based compression technique adapted for tree structures. It extends the principles of the original Re-Pair algorithm [17] to handle the hierarchical nature of trees by identifying and compactly representing frequently occurring patterns. 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 [19]. While Tree Re-Pair is effective for general tree compression, this thesis focuses on developing a novel technique specifically tailored for highly repetitive trees. Therefore, we use the XBWT as our primary benchmark for comparison, as it represents a well-established and high-performance baseline in the field.

1.3 Challenges and Contributions

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 compactly.
- **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 address these challenges by developing a novel tree compression scheme that first leverages the well-known automata minimization algorithm to identify repetitive structures. This algorithm efficiently groups together similar subtrees, enabling us to identify and compress them with high efficiency. The tree is treated as a

deterministic finite automaton (DFA) where the root is the initial state and the leaves are the final states. Since trees are acyclic graphs, this structure is a specific type of DFA known as a Directed Acyclic Word Graph (DAWG). For this reason, we focus on an adaptation of Revuz's algorithm, which is specifically designed for minimizing acyclic DFAs, to make it more efficient for our purposes.

Then, we optimize the representation of these structures. Our approach is to partition the tree nodes into chains and apply Run-Length Encoding (RLE), a compression technique that stores sequences of identical data as a single value and a count. The key challenge is to create partitions that maximize the effectiveness of RLE. We prove that this optimization problem can be reduced to the Minimum Weight Perfect Bipartite Matching (MWPBM) problem. MWPBM is a classic graph theory problem focused on finding a pairing of all nodes in a bipartite graph such that the sum of the weights of the connecting edges is minimized. By modeling our partitioning problem as a bipartite graph, we can use efficient algorithms for MWPBM to find the optimal representation and achieve a higher compression ratio.

1.4 Structure of The Thesis

This thesis is structured to guide the reader from the foundational concepts of tree compression to the development and evaluation of our novel approach. The goal is to build a clear understanding of why each component of our proposed pipeline is necessary and how they fit together.

The logical flow is as follows:

- We begin in **Chapter 2** by establishing the necessary theoretical background on labeled trees.
- In **Chapter 3**, we examine the Extended Burrows-Wheeler Transform (XBWT), a state-of-the-art tree compression technique. This serves as a benchmark and highlights the opportunity for improvement, particularly in handling highly repetitive structures.
- To address this, we introduce a new approach based on automata. **Chapter 4** describes how we use Deterministic Finite Automata (DFA) to model the tree's structure and apply Hopcroft's algorithm to minimize this automaton, effectively identifying all unique subtrees (i.e., the repetitions).
- Once repetitions are identified, we need an efficient way to store them. **Chapter 5** introduces the Minimum Weight Perfect Bipartite Matching problem, which we use to find an optimal way to chain the identified repetitive structures, minimizing the overall compressed size.
- **Chapter 6** unites these concepts, presenting the complete pipeline of our proposed compression scheme.
- Finally, **??????** describe the implementation, present the experimental results of our method against the benchmark, and discuss our conclusions and future work.

Chapter 2

Labeled Trees

2.1 Introduction and Motivation

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.

Definition 1. *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.*
- *$\ell : V \to \Sigma$ is a labeling function that assigns a label $\ell(u) \in \Sigma$ to each node u.*

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

2.2 Applications

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 documents 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 a labeled tree.

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

2.3 Compression and Indexing

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 as a list of nodes with their labels and parent-child relationships using pointers in $O(t \log t)$. 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 (Section 1.2).

2.3.1 Information-Theoretic Lower Bound

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 Burrows-Wheeler Transform

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' [7].

3.1 Introduction and Motivation

This chapter explores the Extended Burrows-Wheeler Transform (XBWT), introduced by Ferragina et al. [7], a state of the art technique for labeled tree compression. Understanding the principles and performance of the 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 the XBWT, we can effectively demonstrate the potential advantages and contributions of our new approach, particularly for trees exhibiting high repetitiveness.

In 2005, Ferragina et al. [7] introduced an innovative approach to labeled tree compression by transforming it into a more tractable string compression problem. Their key contribution, the Extended Burrows-Wheeler Transform (XBWT), is a sophisticated data structure that achieves highly efficient compression by combining entropy-compressed edge labels with a succinct representation of the tree topology. This elegant solution not only simplifies the compression process but also maintains the structural relationships essential for tree operations.

The 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 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.

One of the primary applications of the XBWT is in compressing and indexing 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.

3.1.1 Key Aspects

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:** The XBWT supports a range of navigational operations, such as finding the parent, child, or subtree of a node in near-optimal time.
- **Scalability:** The XBWT is particularly useful for large-scale hierarchical data, such as XML documents or phylogenetic trees, where both compression and fast querying are critical.

3.2 Definition

The **Extended Burrows-Wheeler Transform** is a data structure designed to efficiently compress and index *ordered node-labeled trees*. Inspired by the classical Burrows-Wheeler Transform (BWT) [2] 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.

Definition 2 (Node informations)**.** *Let T be an ordered node-labeled tree 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:*

- *$last(u)$: 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.*
- *$\pi(u)$: the string obtained by concatenating the labels of the nodes on the UPWARD PATH from u's parent to the root of T (the root has an empty π component). Note that $\pi(u) = \pi(u') \circ label(u')$ where $\circ$ is the concatenation operator.*

The definition of the XBWT relies on a sorted multi-set S, which contains a triplet $(last(u), \alpha(u), \pi(u))$ for each node u in the tree T.

The construction of S is a two-step process. First, an intermediate multi-set is created by traversing the tree T in pre-order and generating a triplet $(last(u), \alpha(u), \pi(u))$ for each node. Second, this multi-set is stably sorted according to the lexicographical order of the 'π' component to produce the final multi-set S.

Theorem 1. *The XBWT of a labeled tree T consists of two arrays, S_{last} and S_α. These are constructed from the sorted multi-set S of triplets. Specifically, for each*

i from 1 to t, $S_{last}[i]$ is the 'last' component of the i-th triplet in S, and $S_\alpha[i]$ is the 'α' component. The total space required is $2t + t \log |\Sigma|$ bits.

S_π (for each i from 1 to t, $S_\pi[i]$ is the 'π' component of the i-th triplet in S), therefore is not needed after the construction of the XBWT. However, in the following discussion, we will still refer to it as it possesses some important properties.

3.3 Properties

The XBWT's effectiveness as an indexing structure stems from a key property of the sorted multi-set S. This property, along with its consequences, arises directly from the transform's definition and the sorting process.

3.3.1 Key Property: Grouping by Parent

The fundamental property of the XBWT is that the children of any node u in the tree T form a contiguous block in the sorted multi-set S. Let $u_1, \ldots, u_z$ be the children of node u in their original order. Their corresponding triplets will appear consecutively in S in that same order.

> **Example 1:**
> Consider the node u in Figure 3.1. Looking at Table 3.1, we can see that its children form a contiguous block in positions $[5, 6, 7]$ of the sorted multi-set S.

This grouping provides several important consequent properties:

Unary Degree Encoding: The subarray S_{last} for the block of children $[u_1, \ldots, u_z]$ encodes the degree of u in unary. Specifically, $S_{\text{last}}[u_z] = 1$ and $S_{\text{last}}[u_i] = 0$ for $1 \leq i < z$.

> **Example 2:**
> Consider the node u in Figure 3.1. Looking at Table 3.1, we can observe that $S_{\text{last}}[5 \ldots 7] = \{0, 0, 1\}$, which encodes the degree 3 in unary notation, matching the number of children of node u.

Preservation of Sibling Order: If two nodes u and v have the same label, and the triplet for u precedes the triplet for v in S, then the entire block of children of u will also precede the block of children of v.

> **Example 3:**
> Consider nodes u and v in Figure 3.1. In Table 3.1, node u appears at index 2 while node v appears at index 4 in the sorted multi-set S. Following the preservation of sibling order property, all children of u (occupying positions $[5, 6, 7]$) appear before the child of v (at position 8).

Path-based Indexing: This property extends to entire paths. For any label $c \in \Sigma$, all triplets whose π-components are prefixed by c form a contiguous block in S. If u is the i-th node with label c in S_α, its children's block is located within the larger block of all nodes with paths prefixed by c. This block is delimited by the $(i-1)$-th and i-th '1's in the corresponding section of S_{last}.

Example 4:
Let's examine nodes u and v in Figure 3.1, both labeled 'B'. In the sorted multi-set S shown in Table 3.1, u is the first node with label 'B' (at index 2), and v is the second (at index 4).

The children of all nodes labeled 'B' form a contiguous block in S. In this case, the children of both u and v are located in the range $[5, 8]$. We can distinguish between the children of u and the children of v using the S_{last} array:

- The block of children for u (the first 'B' node) starts at the beginning of the range (index 5) and ends at the position of the first '1' in $S_{\text{last}}[5 \dots 8]$.
- The block of children for v (the second 'B' node) starts after the first '1' and ends at the position of the second '1' in $S_{\text{last}}[5 \dots 8]$.

3.3.2 Other Properties

Additional properties of the XBWT components include:

- The first triplet in S always corresponds to the root of the tree T.
- S_{last} contains n ones (for internal nodes) and l zeros (for leaves).
- S_α is a permutation of the node labels in T.

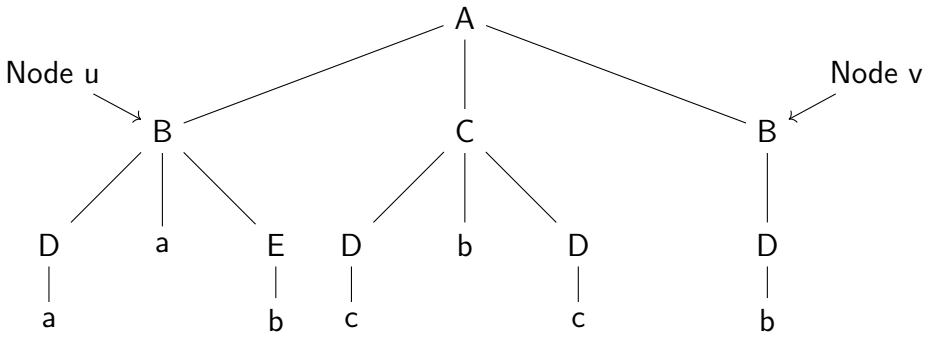


Figure 3.1: A labeled tree T where $\Sigma_N = \{A, B, C, D, E\}$ and $\Sigma_L = \{a, b, c\}$. Notice that $\alpha(u) = \alpha(v) = B$ and $\pi(u) = \pi(v) = A$.

	$S_{\mathbf{last}}$	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

Table 3.1: The multi-set S for the tree shown in Figure 3.1, obtained by stably sorting triplets according to their 'π' components. In this representation, nodes u and v from the original tree T appear at indices 2 and 4, respectively. The children's block of node u occupies positions 5 through 7, while node v's single child is located at index 8.

3.4 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. [7] 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 [14]. 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.

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 the characters are different, their lexicographical order is immediately determined. If they are equal, the algorithm 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. Given a value $j \in \{0, 1, 2\}$, 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. j is chosen in such a way that the number of nodes of the shrunk tree 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 of 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 [14], 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`, 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 Inversion

The ability to invert the XBWT is fundamental to its utility as a compression technique. Invertibility guarantees that the original tree can be perfectly reconstructed from its transformed representation (S_{last} and S_α). This ensures that the compression is lossless, meaning no information is lost during the process, which is a critical requirement for most applications.

Property 'Path-based Indexing' (Subsection 3.3.1) ensures that the two arrays 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.

Algorithm 2 operates in three main steps. First, it constructs two auxiliary arrays, F and J, which are crucial for navigating the tree structure within the compressed format.

- **The F array:** This array maps each character $c \in \Sigma$ to the index of the first occurrence in S of a triplet whose π-component is prefixed by c. It essentially marks the starting points of blocks of nodes that share the same initial path label.
- **The J array:** For each entry i in S, $J[i]$ stores the index in S corresponding to the first child of the node represented by $S[i]$. If $S[i]$ represents a leaf, $J[i]$ is set to a sentinel value (e.g., -1).

Example 5: F and J arrays
Considering the XBWT in Table 3.1, the F array would map 'A' to index 2 (for node r), 'B' to index 5 (for the children of nodes with label 'B'), and so on. For the J array, let's take the node u at index 2 in S. Its first child is at index 5. Therefore, $J[2]$ would be 5.

Finally, the algorithm employs 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(`XBWT[`T`]`)

F = BuildF(`XBWT[`T`]`);
J = BuildJ(`XBWT[`T`]`, F);
Create node r and set $Q = \{\langle 1, r\rangle\}$; ▷ Q is a stack
while $Q \neq \emptyset$ **do** ▷ We still have nodes to create in T
    $\langle i, u\rangle$ = pop(Q);
    $j = J[i]$; ▷ Take the block of u's children in S
    if $j = -1$ **then** ▷ u is a leaf of T
        continue;
    end if
    Find first $j' \geq j$ such that $S_{\text{last}}[j'] = 1$; ▷ $S[j, j']$ are the children of u in T
    for $h = j'$ downto j **do** ▷ Recall that Q is a stack
        Create the node v labeled $S_\alpha[h]$;
        Attach v as first child of u;
        push($\langle h, v\rangle$, Q);
    end for
end while
return node r.

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

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

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

for $i = 1, \ldots, t$ **do**
    if $S_\alpha[i] \in \Sigma_L$ **then**
        $J[i] = -1$; ▷ $S_\alpha[i]$ is a leaf label
    else
        $z = J[S_\alpha[i]]$;
        while $S_{\text{last}}[z] \neq 1$ **do** ▷ Reach the last child of $S_\alpha[i]$
            $z = z + 1$;
        end while
        $F[S_\alpha[i]] = z + 1$;
    end if
end for
return J.

3.6 Compressing Labeled Trees

The XBWT[T] exhibits a local homogeneity property on the string S_α, specifically, node labels get distributed over S_α in accordance with a pattern that clusters closely the labels that descend from 'similar' upward paths sharing long prefixes. 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 [2] when applied to labeled trees.

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 compress the two strings that compose XBWT[T], by exploiting their fine specialties. Of course, many choices are possible for C_α and C_{last}, each having implications on the algorithmic time and compression bounds.

In general, the following theorem holds:

Theorem 2. *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.*

Since $H_k(S_\alpha) \leq (\log|\Sigma|) + 1$,[6] the above bound is at most $t(\log|\Sigma| + 3) + o(t)$ bits, and can be significantly better than the information-theoretic lower bound and the plain storage of XBWT[T] (both taking $2t + t\log|\Sigma|$ bits), depending on the distribution of the labels among its nodes.

3.7 Indexing a Compressed Labeled Tree

In order to implement the efficient operations listed in Section 2.3 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: $A[1] = 0$, $A[j] = 1$ 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, through rank and select operations over A, we can succinctly implement the array F employed in 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 the node at index i. If the child does not exist, it returns -1.

> **Example 6:**
> In Table 3.1, `GetRankedChild(2, 2)` returns 6.

GetCharRankedChild(i, c, k): Returns the position in S of the k-th child labeled c of the node at index i. If the child does not exist, it returns -1.

> **Example 7:**
> In Table 3.1, `GetCharRankedChild(1, B, 2)` returns 4.

GetDegree(i): Returns the total number of children of the node at index i in S.

GetCharDegree(i, c): Returns the number of children of the node at index i in S that have the label c.

GetParent(i): Returns the position in S of the parent of the node at index i. If the node is the root (at index 1), it returns -1.

Example 8:
In Table 3.1, `GetParent(8)` returns 4.

GetSubtree(i): Retrieves the labels of all nodes in the subtree rooted at the node at index i in S. The labels can be returned in any standard traversal order (e.g., pre-order, in-order, or post-order).

SubPathSearch(P): For a given labeled path $P = c_1c_2 \cdots c_k$, this function finds the range $S[\text{First} \dots \text{Last}]$ containing the immediate children of all nodes that match the path P. Meaning that all strings in $S_\pi[\text{First} \dots \text{Last}]$ are prefixed by the reversed path $P^R = c_k \cdots c_2c_1$, as the strings in S_π are constructed using upward paths.

Example 9:
In Table 3.1, `SubPathSearch(BD)` results in the range [12, 13], and `SubPathSearch(AB)` gives the range [5, 8].

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 exploits directly the properties described before, in particular Property 'Path-based Indexing' (Subsection 3.3.1). 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 $F[c]$ is obtained through a select operation on A (line 6). By Property 'Path-based Indexing', the children of $S[i]$ are located at the r-th block of children following position $F[c]$. Lines $8-9$ identify this block.

Example 10:
Let's walk through an example using Table 3.2. Consider the node u at index 2 labeled with B. To find its children:

1. First, we compute $r = 1$ since this is the first occurrence of B in S_α up to position 2.
2. Next, we find $y = F[B] = 5$, which marks the start of the block containing children of all nodes labeled B.
3. Then, we count $z = 1$ ones in S_{last} up to position $y - 1$.

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

Table 3.2: The multi-set S for the tree shown in Figure 3.1, obtained by stably sorting triplets according to their 'π' components. In this representation, nodes u and v from the original tree T appear at indices 2 and 4, respectively. The children's block of node u occupies positions 5 through 7, while node v's single child is located at index 8. Also, the auxiliary binary array A is shown.

4. Finally, the children block is delimited by the $z+r-1 = 1$st and $z+r = 2$nd ones in S_{last}, giving us the range $[5, 7]$.

This range $[5, 7]$ indeed contains the three children of the node at index 2, as we can verify from the tree structure in Figure 3.1.

Algorithm 5 GetChildren(i)

```
1: if S_α[i] ∈ Σ_L then
2:     return −1                                   ▷ S[i] is a leaf
3: end if
4: c ← S_α[i]                                      ▷ S[i] is labeled c
5: r ← rank_c(S_α, i)
6: y ← select_1(A, c)                              ▷ y = F[c]
7: z ← rank_1(S_last, y − 1)
8: First ← select_1(S_last, z + r − 1) + 1
9: Last ← select_1(S_last, z + r)
10: return (First, Last)
```

GetParent(i)

Algorithm 6 is based on Property 'Path-based Indexing' (Subsection 3.3.1) and it is the inverse of the GetChildren method. In 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 'Path-based Indexing' 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 indeed the parent of $S[i]$.

Example 11:
Let's illustrate how to find a node's parent using Table 3.2. Consider node v located at index 4 with label B. The process to find its parent involves:

1. Computing $c = \text{rank}_1(A, 4) = 1$, which tells us the parent has label 'A' (as A contains exactly one 1 up to position 4).
2. Locating $y = F[A] = 2$, which indicates where the block of children for nodes labeled 'A' begins.
3. Calculating $k = \text{rank}_1(S_{\text{last}}, 4-1) - \text{rank}_1(S_{\text{last}}, 2-1) = 0$, meaning no complete child blocks appear before position 4.
4. Therefore, v's parent is the first ($(k+1)$-th) occurrence of 'A' in S_α, corresponding to index 1 (the root of $\mathcal{T}$).

This example demonstrates how the XBWT structure efficiently encodes parent-child relationships using just the S_{last} and S_α arrays.

Algorithm 6 GetParent(i)

```
1: if i = 1 then
2:     return −1                          ▷ S[i] is the root of T
3: end if
4: c ← rank_1(A, i)
5: y ← select_1(A, c)
6: k ← rank_1(S_last, i − 1) − rank_1(S_last, y − 1)
7: p ← select_c(S_α, k + 1)
8: return p
```

SubPathSearch(P)

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.

Example 12:
Consider the tree in Figure 3.1, and let $P = BD$. The algorithm SubPathSearch(P) returns the range $[12, 13]$ through the following steps:

1. Initially, $First = F[B] = 5$ and $Last = F[C] - 1 = 8$. The range $S[5, 8]$ contains all nodes descending from paths prefixed by B.
2. For $c_2 = D$:
   - Compute $k_1 = 0$ and $k_2 = 2$
   - This yields $z_1 = 5$ and $z_2 = 8$
   - The first child of $S[5]$ is at position 12
   - The last (and only) child of $S[8]$ is at position 13
3. Therefore, the algorithm returns the range $[12, 13]$

Note that both the number of offspring and the number of occurrences of subpath P are 2, as evidenced by the two occurrences of 1 in $S_{\text{last}}[12, 13]$.

Algorithm 7 SubPathSearch(P)

```
1: First ← F(c_1); Last ← 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 ← 2, ..., k do
6:     k_1 ← rank_{c_i}(S_α, First − 1); z_1 ← select_{c_i}(S_α, k_1 + 1)        ▷ first entry in S_α[First, t] labeled c_i
7:     k_2 ← rank_{c_i}(S_α, Last); z_2 ← select_{c_i}(S_α, k_2)        ▷ last entry in S_α[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 ← GetRankedChild(z_1, 1)        ▷ get the first child of S[z_1]
12:    Last ← GetRankedChild(z_2, GetDegree(z_2))        ▷ get the last child of S[z_2]
13: end for
14: return (First, Last)
```

3.8 Implementation

The XBWT data structure has been implemented in C++ using the Succinct Data Structure Library 2.0 (SDSL) for efficient representation and manipulation of com-

pressed 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 implementation of the XBWT is based on the descriptions provided in the previous sections. Also, it 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 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 takes as input a labeled tree. It constructs directly a compressed indexing scheme based on the Extended Burrows-Wheeler Transform of the tree as described in the previous sections.
- For the XBWT to work, it is important 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. This can be achieved by remapping the alphabet of the labels. If the original alphabet Σ does not satisfy this property, we can create a new alphabet Σ' where the condition holds. The process involves partitioning the original alphabet into two disjoint sets: Σ_I, containing the labels of internal nodes, and Σ_L, containing the labels of leaf nodes. Subsequently, a new mapping is created that preserves the relative lexicographical order within each set but ensures that every label in Σ_L is mapped to a value greater than any label in Σ_I. For example, if we map the labels to integers, we can assign integers from 1 to $|\Sigma_I|$ to the labels in Σ_I (sorted lexicographically) and integers from $|\Sigma_I| + 1$ to $|\Sigma_I| + |\Sigma_L|$ to the labels in Σ_L (also sorted lexicographically). This re-encoding ensures that the requirement is met before the construction of the XBWT.
- 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 respects the order of the labels in the alphabet and allows simplifying and reducing the space needed to store the labels in the compressed data structure. For this reason, the constructor of the XBWT class takes as input a generic labeled tree.
- All the operations introduced in Section 3.7 are implemented in the XBWT class.

3.8.2 Succinct Data Structures

The implementation of the XBWT relies heavily on succinct data structures to achieve space efficiency while maintaining fast query operations. In particular, we use succinct data structures to compress the two main arrays of the XBWT: S_α and S_{last}. These arrays, which can be quite large for substantial trees, benefit significantly from compression.

The compression is achieved through the Succinct Data Structure Library (SDSL), which provides efficient implementations of various compressed data structures. For S_{last}, which is a binary sequence, we utilize a compressed bit vector that supports fast rank and select operations. For S_α, which contains labels from a potentially large alphabet, we employ a wavelet tree structure that provides both compression and efficient query capabilities.

The 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` [10]. Let's see the implementation details of the SDSL data structures used in the XBWT implementation.

RRR Bit Vector

The RRR bit vector is 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 [22].

The space needed for an RRR 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.

This data structure is used to represent the S_{last}, the additional bit in S_α, and A arrays of the XBWT.

Wavelet Tree

The Wavelet tree is 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 [11]. The `sdsl::wt_int` uses the RRR 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 bit vectors the space needed by integer Wavelet tree 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.

This data structure is used to represent the S_α array of the XBWT.

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 Table 3.3 and Table Table 3.4. The source code for the experiments can be found in the `experiments.cpp` file.

3.9.1 Construction Performance

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 Table 3.3. **Alessio:** Spiega subito i risultati.

Alessio: I numeri vanno sempre allineati a destra, così si capisce meglio chi è più grande di chi. Inoltre, anche il numero di cifre dopo la virgola deve essere sempre lo stesso, così la virgola è fissa e si possono leggere meglio i dati. Su questo faccio io, tu fai sulla prossima :)

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	0.310	0.350
100000	754	0.690	1.250
500000	2246	4.700	16.460
900000	2658	8.510	34.20

Table 3.3: Performance comparison between PathSort (P.S.) and Naive Sort (N.S.) algorithms.

3.9.2 Space Analysis

To evaluate the space savings achieved through XBWT compression, we conducted experiments on the same set of randomly generated trees used for the construction

performance tests. For each tree, we compared the memory usage (in bytes) of three representations: the plain tree, the uncompressed XBWT, and the compressed XBWT.

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

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

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

Table Table 3.4 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.

Alessio: Oltre ai punti di prima, metti la percentuale anche per UXBWT, magari non come un'altra colonna ma metti tra parentesi. Te lo faccio sulle prime righe per la C.XBWT. Se ti piace, ricorda di spiegare cosa sono i numeri tra parentesi nella descrizione.

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 3.4: 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 Table 3.3, 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 , giving speedup of more than $4\times$.

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

DFA Minimization

4.1 Introduction and Motivation

Tree compression schemes that effectively exploit repetitive structures require efficient techniques for identifying and representing such repetitions compactly. A powerful approach to this problem is to view a tree as a finite language, where each path from the root to a leaf represents a word. Such a language can be recognized by a Deterministic Finite Automaton (DFA). More specifically, since trees are inherently acyclic, they can be represented by Acyclic Deterministic Finite Automata (ADFAs).

The problem of finding and compressing identical subtrees is thus equivalent to minimizing the corresponding DFA. 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 identify and encode tree patterns systematically, ultimately improving the compression efficiency.

While general-purpose minimization algorithms like Hopcroft's are highly efficient for any DFA, the specific structure of ADFAs allows for even faster, linear-time algorithms. In this context, we focus on Revuz's algorithm, which is specifically designed to minimize ADFAs and is therefore particularly well-suited for compressing tree structures.

This chapter provides the necessary theoretical background on DFA minimization. We will first introduce the concepts of DFAs and their minimization, followed by a detailed look at both Hopcroft's algorithm as a general solution and Revuz's algorithm as a specialized, linear-time solution for acyclic graphs, which is central to our tree compression methodology.

4.2 Deterministic Finite Automata

Definition 3 (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. DFAs are widely used in various applications, including lexical analysis, pattern matching, and formal language theory.

4.2.1 DFA Minimization

The process of automata minimization consists of 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 as 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 Hopcroft's algorithm, which was proposed by John Hopcroft in 1971 [13]. 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.

4.3 Hopcroft's Minimization Algorithm

DFA minimization 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.

Algorithm Algorithm 8 works by iteratively refining a partition of the states until no further refinement is possible, meaning all states within each set of the partition are indistinguishable. The final partition represents the equivalence classes, which correspond to the states of the minimal DFA. Here's a step-by-step explanation based on the provided pseudocode:

1. **Initialization:** The algorithm starts with an initial partition P containing two sets: the set of final states F and the set of non-final states $Q \setminus F$. These are the coarsest sets of potentially distinguishable states. A working set W is initialized, typically containing the set of final states F (or the smaller of the two initial sets as an optimization). W holds the sets that are used as "splitters" to refine the partition P. These sets are called "splitters" because they are used to partition other sets into smaller, more refined ones. A set $A \in W$ acts as a criterion to distinguish states: if for a given symbol, some states in a set Y transition to a state in A and others do not, then Y is split.
2. **Refinement Loop:** The algorithm iterates as long as the working set W is not empty. In each iteration, a set A (a "splitter") is removed from W. Then, for each input symbol $c \in \Sigma$:

Algorithm 8 Hopcroft's Algorithm for DFA Minimization

Require: $M = (Q, \Sigma, \delta, q_0, F)$

```
function HopcroftMinimization(M)
    P ← {F, Q \ F}                                   ▷ Initial partition
    W ← {F}                                          ▷ Working set initialized with final states
    while W ≠ ∅ do
        Remove a set A from W
        for all c ∈ Σ do
            X ← {q ∈ Q | δ(q, c) ∈ A}                ▷ Predecessors of A via c
            for all Y ∈ P such that X ∩ Y ≠ ∅ and Y \ X ≠ ∅ do
                Replace Y in P with Y1 = X ∩ Y and Y2 = Y \ X
                if Y ∈ W then
                    Replace Y in W with Y1 and Y2
                else
                    Add the smaller of Y1 and Y2 to W
                end if
            end for
        end for
    end while
    return the minimized DFA built from partition P
end function
```

- Calculate the set $X = \{q \in Q \mid \delta(q, c) \in A\}$. This is the set of all states that transition into the set A upon reading symbol c.
- For each set Y currently in the partition P, check if Y needs to be split by X. A split is necessary if some states in Y are in X and some are not (i.e., $X \cap Y \neq \emptyset$ and $Y \setminus X \neq \emptyset$). This indicates that states in Y are distinguishable based on whether their c-transition leads into A.
- If Y needs to be split, replace Y in the partition P with two new sets: $Y_1 = X \cap Y$ (states in Y that transition into A) and $Y_2 = Y \setminus X$ (states in Y that do not transition into A).
- Update the working set W: If the original set Y was in W, remove Y and add both new sets Y_1 and Y_2 to W. If Y was not in W, add only the smaller of the two new sets (Y_1 or Y_2) to W. This optimization helps maintain the algorithm's efficiency.

3. **Termination:** The loop continues until the working set W is empty. At this point, no set in the partition P can be further refined. The partition P now contains the final equivalence classes of states.
4. **Result:** The final partition P defines the states of the minimized DFA. Each set in P corresponds to a single state in the minimal DFA, and transitions are defined based on the original DFA's transitions between these sets.

The algorithm enables computing equivalence classes of nodes in $O(n \log n)$, in particular, the Myhill-Nerode equivalence classes [21, 20]. 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 4 (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 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.

Definition 5 (Regular Language)**.** *A language L over an alphabet Σ is called a* ***regular language*** *if it can be recognized by a deterministic finite automaton (DFA).*

Theorem 3 (Myhill-Nerode theorem [21, 20])**.** *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.*

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. An ADFA is a DFA where the transition graph contains no cycles. This structure is also commonly known as a Directed Acyclic Word Graph (DAWG) when used to represent a set of strings. The acyclic property is key, as it simplifies the minimization process significantly.

In this section, we will discuss an efficient algorithm for minimizing acyclic deterministic finite automata in linear time on the number of states [23]. Notice that we will use the following notation for DAWG: for a state q and a symbol a, the transition $\delta(q, a)$ will be denoted as $q.a$. This notation is extended to words, so for a word $w = w_1 w_2 \dots w_n$, $q.w$ is the state reached from q by following the path labeled by w. A word w is accepted by the automaton if $q_0.w \in F$.

Let's start by giving some definitions and theorems introduced in the paper [23].

Definition 6 (Height function)**.** *For a state s in an automaton, the height $h(s)$ is defined as the length of the longest path starting at s and going to a final state.*

$$h(s) = \max\{|w| : s.w \text{ is final}\}$$

This height function induces a partition Π_i of Q, where Π_i denotes the set of states of height i.

Definition 7 (Distinguished set)**.** *We say that a set Π_i is distinguished if no pair of states in Π_i are equivalent.*

4.4.1 Algorithm

The minimization algorithm introduced in [23] 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 (Definition 6).
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.

   Therefore, the resulting structure of each state s is the following:

   $$label(s) = (F/NF, l_1, nl_1, l_2, nl_2, \ldots, l_k, nl_k)$$

   where F/NF indicates final/non-final, l_k is the k-th transition symbol, and nl_k is the (renumbered) target state name.

   > **Example 13:**
   > Consider a state s at height $i = 2$ that is non-final and has two transitions: one on symbol 'a' to a state t_1 (which belongs to an equivalence class that has been renumbered to 5) and another on symbol 'b' to a state t_2 (belonging to an equivalence class renumbered to 8). The label for state s would be $(NF, a, 5, b, 8)$.

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.

In detail, the algorithm minimizes a DAWG by leveraging the concept of state height. The algorithm partitions the states based on their height. It then processes these groups in increasing order of height, starting from height 0.

The core idea relies on the 'height property': If every Π_j with $j < i$ is distinguished, then two states p and q in Π_i (the set of states with height i) are equivalent if and only if for every letter a in the alphabet Σ, the transitions $p.a$ and $q.a$ lead to the same state (or both are undefined).

The algorithm iteratively ensures that each Π_i is distinguished. It starts with Π_0, where all states are trivially equivalent (as they are final states with no outgoing paths to other final states contributing to height) and merges them. Then, for each subsequent height i, it sorts the states in Π_i based on their transitions. Specifically, states are grouped based on the target states of their transitions for each symbol in the alphabet. Since all lower levels ($j < i$) are already distinguished by the inductive step, states in Π_i that have identical transitions for all symbols (leading to equivalent states in lower levels) are themselves equivalent according to the height property. These equivalent states are then merged.

This process uses a specialized lexicographic sorting technique (related to bucket sort) optimized for this task, which helps achieve linear time complexity relative to the size of the automaton (number of states and transitions). The algorithm leverages a technique similar to the one presented in [1] for testing tree isomorphism. Specifically, Revuz adopts a renumbering scheme during the lexicographic sorting phase to optimize both time and space complexity.

Renumbering Scheme

The core minimization algorithm relies on sorting states at the same height level based on their transitions. The challenge arises when performing the lexicographic sort (using repeated bucket sorts) on these labels, specifically concerning the nl_i components. If the actual state numbers (ranging from 1 to $|Q|$, the total number of states) are used directly as the values for nl_i:

- The range of values for these components becomes large (1 to $|Q|$). This forces the bucket sort step that handles these components to use a bucket array of size $|Q|$, potentially making the sort non-linear in the size of the automaton if $|Q|$ is large compared to the number of transitions e.
- Alternatively, representing state numbers as strings of digits would increase the length of the labels by a factor proportional to $\log |Q|$, again potentially breaking the overall linear time complexity.

The renumbering scheme overcomes this issue by assigning temporary, small integer names to the target states (nl_i) during the sorting process for each height level Π_i. The key idea is that when sorting states at height i, we only need to distinguish between the equivalence classes of the target states nl_j from lower levels ($j < i$), as those levels have already been processed and minimized. The renumbering scheme ensures that the bucket array size is bounded by the maximum between $|\Sigma|$ (the alphabet size) and $|E_i|$ (the total number of edges from states at height i). The renumbering function adopted is presented in Algorithm 9.

Algorithm 9 Renumbering Function

```
1: function RENUMBER(s, h, n)
2:     if s.ch ≠ h then
3:         s.ch = h
4:         s.num = n
5:         n = n + 1
6:     end if
7:     return s.num
8: end function
```

4.4.2 Pseudocode

The paper presents [23] the core minimization logic and a more detailed sorting (or distinguishing) algorithm separately. Algorithm 10 shows a combined representation based on the 'Final algorithm' section of the paper.

Algorithm 10 Minimization Algorithm for DAWGs

Calculate height $h(s)$ for every state s.
Create partitions $\Pi_i = \{s \in Q \mid h(s) = i\}$.
Merge all states in Π_0.
for $i := 1$ to $h(q_0)$ **do** ▷ q_0 is the initial state
    ▷ Distinguish states in Π_i using the sorting/distinguishing algorithm below
    Put states in Π_i into a list L. ▷ May pre-split by Final/NonFinal
    Call Distinguish(L)
    Merge resulting groups of equivalent states identified by Distinguish.
end for

Algorithm 11 distinguishes states within a given height level Π_i. It iteratively refines partitions based on transitions. The sorting happens component by component.

Algorithm 11 Distinguish Algorithm

function DISTINGUISH(List_of_States)
    Place List_of_States into QUEUE2 ▷ Queue of lists of potentially equivalent states
    $k := 0$ ▷ Represents the component index of the label being compared
    repeat
        Move QUEUE2 to QUEUE1; Clear QUEUE2
        $k := k + 1$
        while QUEUE1 not empty **do**
            Let L be the first list in QUEUE1
            Clear Buckets $Q[1 \ldots m]$ ▷ m depends on alphabet size and renumbered state names
            Clear NONEMPTY list of bucket indices
            while L not empty **do**
                Let S be the first state in L
                Determine the k-th component value v for state S
                if $Q[v]$ is empty **then**
                    Add v to NONEMPTY
                end if
                Move S from L to bucket $Q[v]$
            end while
            for each index v in NONEMPTY **do**
                if Bucket $Q[v]$ contains more than one state **then**
                    Add $Q[v]$ as a list to QUEUE2 ▷ These still need further distinguishing
                else if k corresponds to the end-of-label marker 'S' **then**
                    Merge all states in $Q[v]$ (they are equivalent)
                end if
            end for
        end while
    until QUEUE2 is empty ▷ No more lists need refinement
end function

Note: The pseudocode in the paper is slightly intertwined with the bucket sort details. This representation attempts to capture the logic described. The renumbering function is called implicitly when accessing nl_j. The handling of merging might occur after the Distinguish procedure completes for level i, or potentially within it as groups become fully distinguished. The pseudocode merges states within bucket Q[$], which relates to the end-of-string marker in the generic sort; for state labels, equivalence is confirmed when a group remains together after checking all label components.

Example 14:
Now we are going to see an example of reduction for a given DAWG. The DAWG is represented in figure Figure 4.1 and, as we can notice, it is also a valid ordered rooted tree with $n = 11$ nodes, $e = 10$ edges, and the following alphabet: $\Sigma = \{0, 1\}$. The node a is the root of the tree and the initial state of the automaton, while the leaf nodes e, g, h, i, l, m are final states. It is important to note that while the algorithm applies to any DAWG, our focus is on those that are also trees, as this is the specific case relevant to our work.

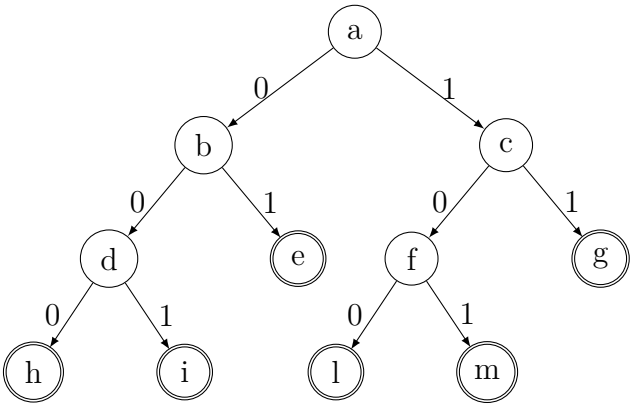


Figure 4.1: Example DAWG to be minimized

Now, let's apply the minimization algorithm step by step:

1. **Height Computation:** First, we compute the height of each state. The height is the length of the longest path to a final state. The final states (e, g, h, i, l, m) have a height of 0. For the other states, the height is calculated as follows:

   - $h(d) = 1 + \max(h(h), h(i)) = 1 + 0 = 1$
   - $h(f) = 1 + \max(h(l), h(m)) = 1 + 0 = 1$
   - $h(b) = 1 + \max(h(d), h(e)) = 1 + \max(1, 0) = 2$
   - $h(c) = 1 + \max(h(f), h(g)) = 1 + \max(1, 0) = 2$
   - $h(a) = 1 + \max(h(b), h(c)) = 1 + \max(2, 2) = 3$

   This gives us the following partitions based on height:

   - $\Pi_0 = \{e, g, h, i, l, m\}$
   - $\Pi_1 = \{d, f\}$

- $\Pi_2 = \{b, c\}$
- $\Pi_3 = \{a\}$

2. **Processing Π_0:** All states in Π_0 are final and have no outgoing transitions, so they are all equivalent. We merge them into a single class, let's call it $D = \{e, g, h, i, l, m\}$. After this step, we have a new state D which is final.
3. **Processing Π_1:** Now we examine the states in Π_1: d and f. We check their transitions:
   - State d: $\delta(d, 0) = h \in D$ and $\delta(d, 1) = i \in D$.
   - State f: $\delta(f, 0) = l \in D$ and $\delta(f, 1) = m \in D$.

   Since both states transition to the same equivalence class (D) for both symbols 0 and 1, they are equivalent. We merge them into a new class, $C = \{d, f\}$.
4. **Processing Π_2:** Next, we process the states in Π_2: b and c.
   - State b: $\delta(b, 0) = d \in C$ and $\delta(b, 1) = e \in D$.
   - State c: $\delta(c, 0) = f \in C$ and $\delta(c, 1) = g \in D$.

   Both states have transitions to class C on symbol 0 and to class D on symbol 1. Therefore, b and c are equivalent. We merge them into a new class, $B = \{b, c\}$.
5. **Processing Π_3:** Finally, we process Π_3, which contains only state a. There is nothing to compare it with, so it forms its class, $A = \{a\}$.

After applying the algorithm, we obtain the minimized DAWG represented in figure Figure 4.2. Each node of the original DAWG is represented by a node in the minimized DAWG (equivalence classes). The edges represent transitions between these nodes. The root node A is the initial state of the minimized DAWG, while the node D is the final state.

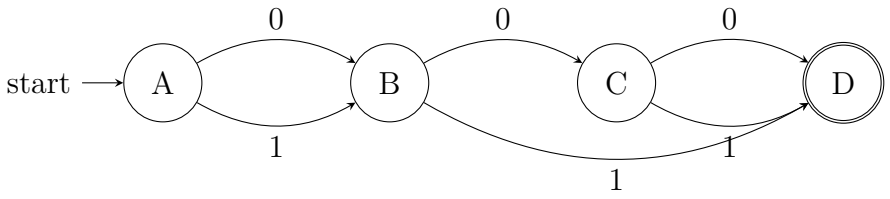


Figure 4.2: Minimized DAWG

The equivalence classes of the nodes are listed in table Table 4.1.

Class	States
A	a
B	b, c
C	d, f
D	e, g, h, i, l, m

Table 4.1: Equivalence classes of the nodes

Chapter 5

Min-Weight Perfect Bipartite Matching

5.1 Introduction and Motivation

In this chapter, we delve into the problem of finding a minimum-weight perfect matching in a bipartite graph (MWPBM problem), a fundamental challenge in combinatorial optimization with wide-ranging applications. This problem, often referred to as the assignment problem, seeks to pair elements from two distinct sets in the most efficient way possible, minimizing the total cost of the pairings. The principles and algorithms discussed here are not only of theoretical importance but also have practical relevance in fields such as logistics, scheduling, and resource allocation.

Our approach is to partition the tree nodes into chains and apply Run-Length Encoding (RLE), a compression technique that stores sequences of identical data as a single value and a count. The key challenge is to create partitions that maximize the effectiveness of RLE. We prove that this optimization problem can be reduced to the MWPBM problem. By modeling our partitioning problem as a bipartite graph, we can use efficient algorithms for MWPBM to find the optimal representation and achieve a higher compression ratio.

5.2 Bipartite Graph

Definition 8. *A graph $G = (V, E)$ is called bipartite if its vertex set V can be partitioned into two disjoint subsets $V = V_1 \cup V_2$ such that every edge in E has the form (v_1, v_2) where $v_1 \in V_1$ and $v_2 \in V_2$.*

In other words, the vertices of the graph can be divided into two separate groups such that all edges connect a vertex from the first group to a vertex from the second group. An example of a bipartite graph is shown in Figure 5.1.

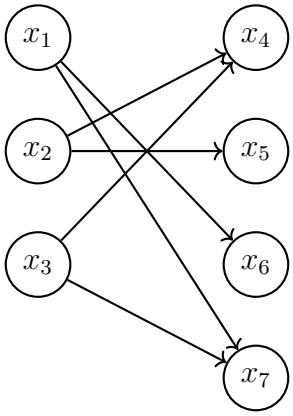


Figure 5.1: Example of a bipartite graph $G = (V, E)$ where $V_1 = \{x_1, x_2, x_3\}$, $V_2 = \{x_4, x_5, x_6, x_7\}$ and $E = \{(x_1, x_6), (x_1, x_7), (x_2, x_4), (x_2, x_5), (x_3, x_4), (x_3, x_7)\}$.

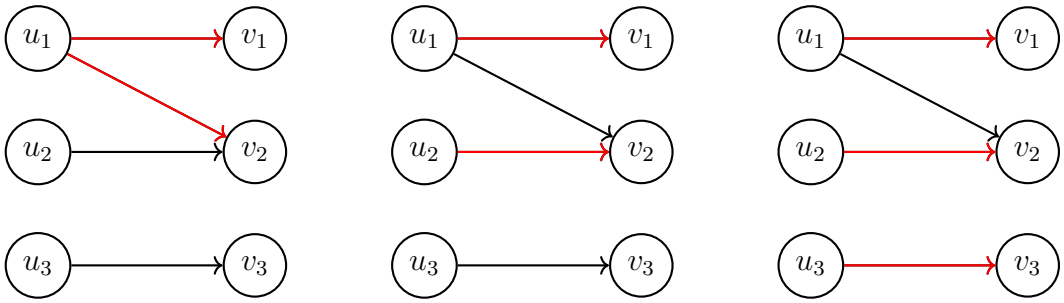
5.3 Problem Definition

Given a weighted bipartite graph $G = (V, E)$ (Definition 8), let's define the concept of a matching.

Definition 9 (Matching)**.** *Given a bipartite graph $G = (V, E)$, a matching $M \subseteq E$ 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 $|V_1| = |V_2|$ [9].

Example 15:
In Figure 5.2, we illustrate three distinct scenarios. Subfigure (a) depicts a set of edges that does not constitute a valid matching, as vertex u_1 is incident to more than one edge, namely (u_1, v_1) and (u_1, v_2), violating the definition of a matching. Subfigure (b) presents a valid, yet non-perfect matching; here, vertices u_3 and v_3 are exposed, meaning they are not incident to any edge in the matching. Finally, subfigure (c) shows a perfect matching, where every vertex in the graph is incident to exactly one edge in the matching, satisfying the condition $|M| = |V_1| = |V_2| = 3$.



(a) Not a matching (b) A non-perfect matching (c) A perfect matching

Figure 5.2: Examples of a non-matching (a), a non-perfect matching (b), and a perfect matching (c) in a bipartite graph. The edges in the set M are highlighted in red.

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 10 (Minimum weight perfect matching in bipartite graphs)**.** *Given a weighted bipartite graph $G = (V, E)$, 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 $w(e)$. We define the weight of a matching M as follows:

$$w(M) = \sum_{e \in M} w(e) \tag{5.1}$$

Example 16:
Consider the weighted bipartite graph in Figure 5.3. The goal is to find a perfect matching with the minimum possible total weight. Both subfigures show a valid perfect matching; however, only one of them has the minimum weight.

- Subfigure (a) shows the perfect matching $M_a = \{(u_1, v_2), (u_2, v_1), (u_3, v_3)\}$. Its total weight is $w(M_a) = w(u_1, v_2) + w(u_2, v_1) + w(u_3, v_3) = 2 + 1 + 1 = 4$. This is a valid perfect matching, but it is not optimal.
- Subfigure (b) shows the perfect matching $M_b = \{(u_1, v_1), (u_2, v_2), (u_3, v_3)\}$. Its total weight is $w(M_b) = w(u_1, v_1) + w(u_2, v_2) + w(u_3, v_3) = 1 + 1 + 1 = 3$.

Since $w(M_b) < w(M_a)$, the matching in (b) is a minimum weight perfect matching for this graph, while the matching in (a) is a non-minimum perfect matching.

(a) A non-minimum perfect matching.

(b) A minimum perfect matching.

Figure 5.3: Example of a non-minimum perfect matching (a) and minimum perfect matching (b) in a weighted bipartite graph. Dashed edges have weight 2 while solid edges 1. The edges in a matching are highlighted in red.

5.4 Hall's Marriage Theorem

Hall's Marriage Theorem [12] provides a necessary and sufficient condition for the existence of a matching in a bipartite graph that saturates one side of the partition. It's often stated in the context of finding pairings (like marriages) between two sets of entities. This theorem will be crucial in proving the correctness of our reduction from the tree partitioning problem to the minimum weight perfect bipartite matching problem. Specifically, we will use it to show that our constructed bipartite graph always admits a perfect matching, ensuring that our reduction is valid.

Definition 11 (Neighborhood)**.** *For a subset of vertices $W \subseteq V_1$, the **neighborhood** of W, denoted by $N(W)$, is the subset of all vertices in V_2 that are adjacent to at least one vertex in W.*

$$N(W) = \{v \in V_2 \mid \exists u \in W \wedge \{u, v\} \in E\}$$

Theorem 4 (Hall's Marriage Theorem [12])**.** *Let $G = (V, E)$ be a bipartite graph. There exists a perfect matching M in G if and only if for every subset $W \subseteq V_1$, the following condition holds:*

$$|N(W)| \geq |W|$$

This condition is known as **Hall's condition**.

In simpler terms, a matching that covers all vertices in V_1 exists if and only if every group of vertices chosen from V_1 collectively has at least as many neighbors in V_2 as there are vertices in the chosen group.

5.5 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 P of the above integer program is as follows:

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

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 "corner" 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 $y \neq z$.

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 all have integral coordinates. 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 leads to the following result:

Theorem 5. *Any extreme point of P is a $0 - 1$ vector; hence, it is the incidence vector of a perfect matching.*

Consequently, the polytope

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

is called the bipartite perfect matching polytope (see Lecture notes by [9]).

5.6 Proposed Solutions

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 [16]. 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 [5] showed that the algorithm can be implemented in $O(n^3)$ time.

The Hungarian method is a powerful algorithm; however, it is not very intuitive and can be difficult to implement. In recent years, several other algorithms have been proposed to solve this problem. In 1970, Edmonds and Karp [6] proposed an algorithm that solves the problem in $O(nm + n^2 \log n)$ time. In 1989 Gabow and Tarjan [8] 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 [24] 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. [3] proposed a new solution to the Minimum Cost Flow problem that works in almost-linear time, precisely in $O(m^{1+o(1)})$ time. The minimum-cost flow problem is a classic combinatorial graph problem with numerous applications in engineering and scientific computing. This result is important for our problem since the maximum weight perfect matching problem can be reduced to minimum-cost flow, enabling an almost-linear time solution.

5.7 Implementation

Davide T.: Da completare una volta deciso come procedere

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

Project Overview

As introduced in the first chapter, the primary goal of this thesis is to develop a novel tree compression scheme that effectively leverages repetitive structures within the input tree. The proposed algorithm is designed to identify and compactly represent these recurring patterns, thereby improving compression performance, particularly for highly repetitive trees. This chapter provides an overview of the proposed compression scheme.

6.1 Compression Scheme Pipeline

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 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 (the root has an empty π component).

The following pipeline is used to compress the tree T:

1. Initially, the array π is computed for T by traversing the tree in a pre-order fashion. Then the nodes are stably sorted by the lexicographic order of their π strings. To sort the nodes, the **Path Sort** algorithm introduced in Subsection 3.4.2 is used, enabling the sorting of nodes in linear time and $O(t \log t)$ space. This sorting step is the one used in the XBWT transform described in Chapter 3.

2. Then, using the Algorithm 10 for minimizing acyclic deterministic finite automata (ADFAs) described in Section 4.4, the nodes are partitioned into equivalence classes where two nodes are equivalent if they have the same subtree rooted at them.

3. Given a width p, the previously sorted nodes are then divided into p chains to minimize the run-length encoding of each chain (considering the equivalence classes). To do so, we reduce this problem — which we call *Chains-Division problem* — to the Minimum Perfect Bipartite Matching problem (see Chapter 5), which can be solved in polynomial time as described in Section Section 5.6.

4. Lastly, the resulting deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA) can be indexed using the scheme introduced by Cotumaccio et al. [4]. Also, the chains may be compressed using techniques such

as run-length encoding, Huffman encoding, and Elias-Fano encoding (further details will be explained in the next chapters).

6.2 Reducing the Chains-Division Problem to the Assignment Problem

In this section, we will show how we can reduce the problem of finding the optimal partition of the nodes of a labeled tree T given their equivalence classes into p chains to the Minimum Weight Perfect Bipartite Matching problem (see Definition 10). We define $\mathcal{C}$ as the set of equivalence classes of the nodes of T, and t as 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.

In particular, we show that, given a tree T and the number of chains p, we can construct a bipartite graph $G = (V, E)$ in which a perfect matching (Definition 9) always exists. In turn, a perfect matching with minimum weight enables us to retrieve the optimal partition of the nodes in T into p chains, such that the run-length encoding of each chain is minimized.

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.

6.2.1 Chains-Division Problem Definition

It is essential to begin by defining the problem we aim to solve.

Definition 12 (Chains-Division Problem)**.** *Given a labeled tree T, the equivalence classes $\mathcal{C}$, the stable order of the nodes in T according to the upward path π as defined in Definition 2, and an integer parameter $p \in [2, t]$, find the optimal partition of the nodes of T into p chains such that the run-length encoding of each chain is minimized.*

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

Definition 13 (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.

> **Example 17:**
> Let $S = \{A, A, B, B, B, C, C, A, A\}$. The run-length encoding of S would be $R = \{(A, 2), (B, 3), (C, 2), (A, 2)\}$. The length of the RLE, which is the value we want to minimize, is $|R| = 4$.

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 we want to reduce the number of distinct equivalence classes in each chain. Follows the definition of a chain.

Definition 14 (Chains)**.** *Given a tree T, a chain C is a sequence of nodes $C = \{c_1, c_2, \ldots, c_m\}$ such that $C \subseteq V$. Additionally, each node of T belongs to exactly one chain, and the nodes in the chain are ordered according to the upward path π (as defined in Definition 2) of each node c_i.*

Note that, following the XBWT definition, two sibling nodes are comparable. The stable sorting algorithm respects the original sibling order, meaning the node that appears first among its siblings in a pre-order traversal will also come first in the sorted sequence.

Example 18: Chains-Division Problem

Consider a tree T with 7 nodes having the following equivalence classes: $E = \{A, B, A, C, A, B, B\}$, where the nodes are ordered according to their upward paths. Let's say we want to divide these nodes into $p = 2$ chains.

Non-optimal division: If we divide the nodes into chains $C_1 = \{A, B, A, C\}$ and $C_2 = \{A, B, B\}$, the run-length encoding would be:

- C_1: $(A, 1), (B, 1), (A, 1), (C, 1)$ - requiring 4 pairs
- C_2: $(A, 1), (B, 2)$ - requiring 2 pairs

Total RLE cost: $4 + 2 = 6$

Optimal division: A better division would be $C_1 = \{A, A, A\}$ and $C_2 = \{B, C, B, B\}$, with run-length encoding:

- C_1: $(A, 3)$ - requiring 1 pair
- C_2: $(B, 1), (C, 1), (B, 2)$ - requiring 3 pairs

Total RLE cost: $1 + 3 = 4$

This example demonstrates how grouping nodes of the same equivalence class in chains minimizes the total run-length encoding cost. The optimal solution can be found by reducing this problem to the MWPBM problem as described in this chapter.

6.2.2 Bipartite Graph Construction

Now, we will show how to construct a bipartite graph that allows us to solve the CHAINS-DIVISION problem.

Definition 15 (Bipartite graph construction)**.** *Let T be a tree with t nodes, and p the number of chains we want to partition the nodes into. Let $\mathcal{C}$ 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:*

- *V_1 contains $t + p$ nodes composed by p source nodes $s_1, s_2, \ldots, s_p$ (referred to collectively as $\mathcal{S}$) followed by the t elements (referred to collectively as $\mathcal{T}_1$) of $\mathcal{C}$. The nodes in V_1 follow a strict ordering $s_1 \prec s_2 \prec \cdots \prec s_p \prec u_1 \prec u_2 \prec \cdots \prec u_t$, where u_i are the tree nodes ordered according to the upward path π as defined in Definition 2.*
- *V_2 contains $t + p$ nodes composed by the t elements (referred to collectively as*

$\mathcal{T}_2$) of $\mathcal{C}$ followed by p destination nodes $d_1, d_2, \ldots, d_p$ (referred to collectively as $\mathcal{D}$). The nodes in V_2 follow a strict ordering $v_1 \prec v_2 \prec \cdots \prec v_t \prec d_1 \prec d_2 \prec \cdots \prec d_p$, where v_i are the tree nodes ordered according to the upward path π as defined in Definition 2.

Then the edges of the graph G are constructed in the following way:

1. *The $\mathcal{S}$ nodes are connected to the first p nodes with distinct equivalence class in V_2, with weight 1.*
2. *Let $u_i \in \mathcal{T}_1$. We define $\mathcal{C}(u_i)$ as the equivalence class of node u_i. For each node u_i, we construct the following edges:*

   - For the first p nodes $v_j \in \mathcal{T}_2$ such that $j > i$ and $\mathcal{C}(v_j) \neq \mathcal{C}(u_i)$, we add an edge (u_i, v_j) with weight 1. If there are fewer than p nodes in V_2 with distinct equivalence classes, we stop earlier.

   - Let $v_k \in \mathcal{T}_2$ be the first node in the ordering such that $k > i$ and $\mathcal{C}(v_k) \neq \mathcal{C}(u_i)$, we add an edge (u_i, v_k) with weight 0. If such a node does not exist, we add p edges (u_i, d_j) with weight 0 for each $j = 1, 2, \ldots, p$, where $d_j \in \mathcal{D}$.

Notice that it is important to consider the order of the nodes of the two sets V_1 and V_2 as stated in the definition, 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. An example of the node structure is shown in Subsection 6.2.2.

Notice also that when we talk about the same $\mathcal{T}_1$ node placed in $\mathcal{T}_2$, we are referring to the corresponding node in $\mathcal{T}_2$ that derives from the same node in the original tree T since the nodes of the tree are ordered in both sets $\mathcal{T}_1$ and $\mathcal{T}_2$. In Figures 6.2, 6.3 and 6.5, the node's correspondence is achieved by putting the two nodes at the same level.

Example 19: Vertices

This example illustrates the structure of the bipartite graph vertices for the tree shown in Figure 6.1. The nodes in this tree are labeled by their equivalence classes obtained from the minimization of the corresponding DAWG in Subsection 4.4.2. Our goal is to partition the tree's nodes into $p = 2$ chains.

Figure 6.2 shows the corresponding bipartite graph. The graph is composed of:

- Two source nodes (s_1, s_2) on the left, representing the start of each chain.
- Two sets of tree nodes, representing the partitions $\mathcal{T}_1$ (left column) and $\mathcal{T}_2$ (right column). These nodes are ordered based on the `pathSort` algorithm. The labels (A, B, C, D) correspond to the equivalence classes from the original tree.
- Two destination nodes (d_1, d_2) on the right, representing the end of each chain.

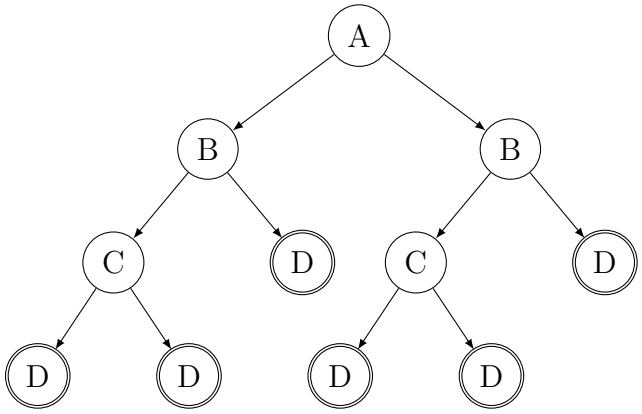


Figure 6.1: Tree resulting from DAWG minimization of Figure 4.1. Each node is labeled with its equivalence class.

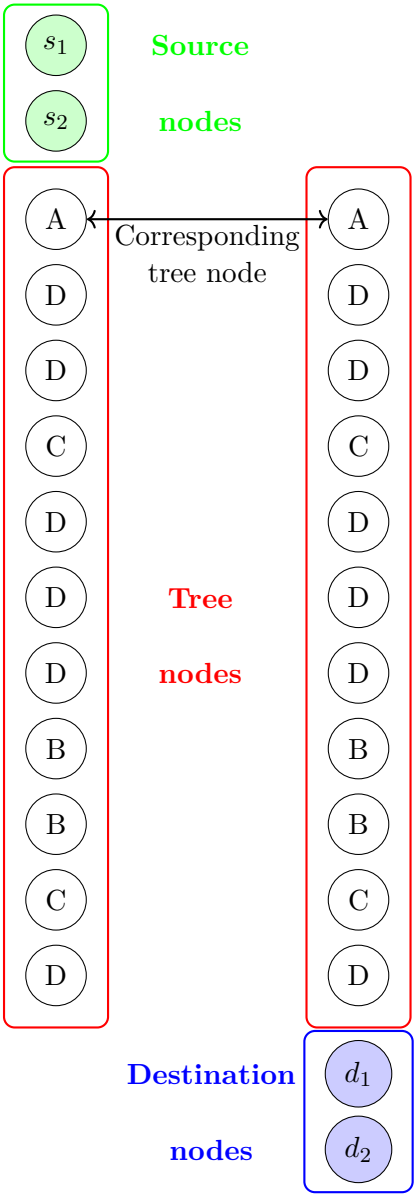


Figure 6.2: Corresponding bipartite graph structure for the tree in Figure 6.1 with $p = 2$. The nodes are ordered from top to bottom using Algorithm 1.

Example 20: Edges

Let's see a small example for each case. Consider $p = 2$. In Figure 6.3-(a), there is an example for the sources' edges. As stated before, for each source, p edges with weight 1 are created and connected to the first p nodes with distinct equivalence classes in $\mathcal{T}_2$.

In Figure 6.3-(b), there is an example for the tree nodes' edges. For each node in $\mathcal{T}_1$, edges with weight 1 are created and connected to the first p nodes with distinct equivalence class in $\mathcal{T}_2$ after the corresponding node in $\mathcal{T}_2$ (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 $\mathcal{T}_2$ after the corresponding node in $\mathcal{T}_2$. As we can see from the image, we consider the first node in $\mathcal{T}_1$ labelled A that is connected to the node labelled B with weight 1, and to the node labelled C with weight 1, and to the second node labelled A in V_2 with weight 0.

Lastly, in Figure 6.3-(c) there is an example for the destination nodes' edges. We start by considering the node in $\mathcal{T}_1$ that is labeled A, which is connected to a node labeled B with weight 1. Then, since there is no node with the same class in $\mathcal{T}_2$, 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 B 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.

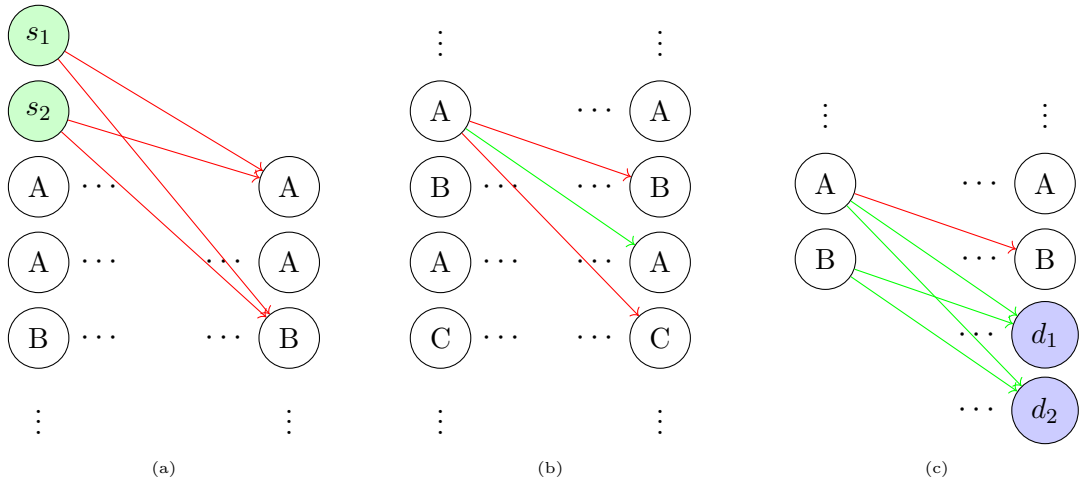


Figure 6.3: Examples of the connection construction in the bipartite graph for $p = 2$, showing the cases for source nodes $\mathcal{S}$ (a), internal tree nodes $\mathcal{T}_1$ and $\mathcal{T}_2$ (b), and destination nodes $\mathcal{D}$ (c). Red arrows indicate edges with weight 1, while green arrows indicate edges with weight 0.

Before we present the proof of the correctness of the reduction, let us state the following theorem regarding the number of edges in the bipartite graph resulting from Definition 15. This theorem is essential for understanding the complexity of the final algorithm employed to solve the MWPBM problem and so, the CHAIN-DIVISION problem.

Theorem 6 (Bipartite graph properties)**.** *The bipartite graph G constructed as stated in Definition 15 has $2t + 2p$ nodes and $O(t(p + 1) + p^2 + tp)$ edges.*

Proof. 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

classes meaning that all the nodes are connected to the destination nodes. □

6.2.3 Proof of Correctness

In this section, we present the proof of the correctness of the reduction introduced in the previous sections. Let's start by stating the following lemmas.

Lemma 1. *The optimal solution of an instance $\mathcal{I}$ of the CHAINS-DIVISION problem for a tree T is always greater than or equal to $|\mathcal{C}|$.*

Proof. To minimize the run-length encoding of the chains, we note that the minimum cost of a chain is 1. Consequently, the optimal cost of the *CHAINS-DIVISION* problem for the tree T is always greater than or equal to the cardinality of the set of equivalence classes $\mathcal{C}$ This is because if we partition them into $p = |\mathcal{C}|$ chains, the cost will be equal to $|\mathcal{C}|$, since each chain contains only nodes belonging to the same class. Conversely, if we partition them into $p < |\mathcal{C}|$ chains, the cost will be greater than or equal to $|\mathcal{C}|$ since we will need to include at least two nodes from different classes within a single chain. □

Claim 1. *The solutions for the CHAINS-DIVISION problem for the instances where the number p of chains is greater than $|\mathcal{C}|$ are not better than the solutions for the instances where $p \leq |\mathcal{C}|$.*

Proof. The proof builds upon Lemma 1. If we use a number of chains $p > |\mathcal{C}|$, we would have at least $p - |\mathcal{C}|$ empty chains, since there are only $|\mathcal{C}|$ non-empty equivalence classes of nodes to partition. As the minimum cost for any chain is 1, these empty chains contribute to the total cost. An optimal arrangement would involve $|\mathcal{C}|$ chains, each containing nodes from a single equivalence class, costing $|\mathcal{C}|$, and $p - |\mathcal{C}|$ empty chains, each costing 1. The total cost would be $|\mathcal{C}| + (p - |\mathcal{C}|) = p$. Since $p > |\mathcal{C}|$, this cost is greater than the optimal cost of $|\mathcal{C}|$ achievable with $p = |\mathcal{C}|$ chains. Therefore, any solution with $p > |\mathcal{C}|$ is suboptimal. □

Therefore, for the proof of the reduction, we will only consider instances of the problem where $p < |\mathcal{C}|$, as they do not present a trivial solution.

Lemma 2. *Given a bipartite graph G constructed as stated in Definition 15, for each node $u_i \in \mathcal{T}_1$ it is impossible for u_i to be connected to a node $v_j \in \mathcal{T}_2$, such that $j \leq i$ in the order of the nodes.*

Proof. The proof comes from the construction of G (Definition 15) where the nodes of $\mathcal{T}_1$ are always connected to the nodes of $\mathcal{T}_2$ coming after them. □

Lemma 3. *In the bipartite graph G constructed as per Definition 15, for every node $u \in \mathcal{T}_1$ $|N(\{u\})| \geq 1$.*

In other words, every node in V_1 is connected to at least another node in V_2.

Proof. Let $u_i \in \mathcal{T}_1$ be an arbitrary node. We analyze the construction of its outgoing edges based on Definition 15. There are two mutually exclusive cases for u_i:

1. There exists at least one node $v_k \in \mathcal{T}_2$ with $k > i$ that has the same equivalence class as u_i, i.e., $\mathcal{C}(v_k) = \mathcal{C}(u_i)$. In this case, the construction specifies that an edge is added between u_i and the first such node v_k. This guarantees u_i has at least one neighbor.
2. There are no nodes $v_k \in \mathcal{T}_2$ with $k > i$ that share the same equivalence class as u_i. This occurs when u_i is the last node of its equivalence class in the specified ordering. In this scenario, the construction adds p edges from u_i to each of the destination nodes $d_j \in \mathcal{D}$. Since $p \geq 2$, u_i is connected to at least two nodes.

In either case, any node $u_i \in \mathcal{T}_1$ is guaranteed to have at least one outgoing edge. Therefore, its neighborhood is non-empty. □

Lemma 4. *For any pair of distinct nodes $u_i, u_j \in \mathcal{T}_1$ such that $u_i \prec u_j$, $N(\{u_i\}) \not\subseteq N(\{u_j\})$.*

Proof. By Lemma 3, the set of neighbors $N(\{u_i\})$ and $N(\{u_j\})$ are non-empty. We will show that $N(\{u_i\}) \not\subseteq N(\{u_j\})$.

By construction, u_i is connected to v_{i+1}. This holds whether v_{i+1} is the next node in the same equivalence class or one of the first p nodes in a different class. Thus, $v_{i+1} \in N(\{u_i\})$.

From Lemma 2, any neighbor v_k of u_j must have $k > j$. Since $i < j$, we have $i + 1 \leq j$. This means v_{i+1} cannot be a neighbor of u_j, as $i + 1 < k$. Therefore, $v_{i+1} \in N(\{u_i\})$ but $v_{i+1} \notin N(\{u_j\})$, which proves that $N(\{u_i\}) \not\subseteq N(\{u_j\})$. □

Lemma 5. *For any pair of distinct nodes $u_i, u_j \in \mathcal{T}_1$ such that $u_i \prec u_j$ and u_j is not the last node of its equivalence class in the order, then $N(\{u_j\}) \not\subseteq N(\{u_i\})$.*

Proof. Since u_j is not the last node of its equivalence class, by construction, it must be connected to the first node $v_k \in \mathcal{T}_2$ with $k > j$ such that $\mathcal{C}(v_k) = \mathcal{C}(u_j)$. Therefore, $v_k \in N(\{u_j\})$.

By construction, it follows that u_i cannot be connected to v_k. This is due to the existence of another node $v_l \in \mathcal{T}_2$ with $i < l \leq j$, such that $\mathcal{C}(v_l) = \mathcal{C}(u_k)$. Consequently, we conclude that $v_k \notin N(\{u_i\})$, as the edges of u_i connect to nodes in $\mathcal{T}_2$, all of which belong to distinct classes.

In summary, we have identified a node v_k that is included in $N(u_j)$ but excluded from $N(u_i)$, thus proving that $N(u_j) \not\subseteq N(u_i)$. □

Lemma 6. *For every possible instance of the CHAINS-DIVISION problem, a perfect matching exists in the bipartite graph G constructed as specified in Definition 15.*

Proof. The proof comes from the construction of the bipartite graph G and from Theorem 4. We are going to prove that G satisfies Hall's condition (see Theorem 4) and so, since by construction $|V_1| = |V_2|$, a perfect matching for G exists.

To verify Hall's condition, we need to prove that for any subset $W \subseteq V_1$ we have that $|N(W)| \geq |W|$, where $N(W)$ is the neighborhood of W (Definition 11). We have the following cases:

1. $W \subseteq \mathcal{S}$: Let W be a subset of S of size k. By construction, every source node $s_i \in \mathcal{S}$ is connected to the same set of p nodes in V_2, which are the first p nodes with distinct equivalence classes in the ordering. Therefore, for any non-empty $W \subseteq \mathcal{S}$, the neighborhood $N(W)$ consists of exactly these p nodes, so $|N(W)| = p$. From Lemma 1 and Claim 1, we only consider instances where $p \leq |\mathcal{C}|$, ensuring that at least p such nodes exist. Since $|\mathcal{S}| = p$, we have $|W| = k \leq p$. Thus, $|N(W)| \geq |W|$.

2. $W \subseteq \mathcal{T}_1$: Let $W = \{u_{i_1}, \ldots, u_{i_k}\} \subseteq \mathcal{T}_1$ with $i_1 < i_2 < \cdots < i_k$. We analyze two subcases:

   Case A: No node in W is the last of its class. By Lemmas 4 and 5, for any two nodes $u_a, u_b \in W$ with $a < b$, we have both $N(\{u_a\}) \not\subseteq N(\{u_b\})$ and $N(\{u_b\}) \not\subseteq N(\{u_a\})$. This implies that each node in W contributes at least one unique neighbor to the total neighborhood $N(W)$. Therefore, $|N(W)| \geq |W|$.

   Case B: At least one node in W is the last of its class. Let $u_j \in W$ be a node that is the last of its equivalence class. By construction, u_j is connected to all p destination nodes $\mathcal{D}$, which means $\mathcal{D} \subseteq N(W)$ and thus $|N(W)| \geq p$.

   For such a node u_j, its neighborhood $N(\{u_j\})$ may be a subset of $N(\{u_a\})$ for some $u_a \in W$ with $a < j$, since Lemma 5 does not apply. However, the reverse is not true, as Lemma 4 guarantees $N(\{u_a\}) \not\subseteq N(\{u_j\})$. This asymmetry ensures that $N(\{u_a\})$ always contributes at least one neighbor not present in $N(\{u_j\})$. This property, combined with the fact that all nodes that are last of their class are connected to the p destination nodes, is sufficient to guarantee that $|N(W)| \geq |W|$ since $p \geq 2$.

   In both cases, Hall's condition $|N(W)| \geq |W|$ is satisfied for any $W \subseteq \mathcal{T}_1$.

3. $W = W_S \cup W_U$, where $W_S \subseteq \mathcal{S}, W_U \subseteq \mathcal{T}_1$: The neighborhood of W_S consists of p nodes, as established in case 1 ($W \subseteq \mathcal{S}$). By Lemma 2, the neighbors of any node in W_U appear later in the node ordering than the node itself. Since all nodes in $\mathcal{T}_1$ are ordered after the source nodes, the neighbors of W_U are distinct from the neighbors of W_S. Specifically, $N(W_S)$ consists of the first p nodes with distinct equivalence classes.

   We now analyze two subcases for nodes in W_U: If W_U contains a node $u_i \in \mathcal{T}_1$ that is not the last of its equivalence class, then there exists at least one node $v_j \in \mathcal{T}_2$ with $i < j$ and $\mathcal{C}(u_i) = \mathcal{C}(v_j)$ that is connected to u_i but not to any source node. This contributes additional neighbors to $N(W)$, ensuring $|N(W)| \geq |W|$. If W_U contains a node $u_i \in \mathcal{T}_1$ that is the last of its equivalence class, then by construction, u_i is connected to all destination nodes. This guarantees $|N(W)| \geq |W|$.

   In both subcases, Hall's condition $|N(W)| \geq |W|$ is satisfied.

□

We can now prove the correctness of the reduction. Consider a perfect matching M in G. Therefore, $|V_1| = |V_2|$ and M is perfect, every node in V_1 is matched to exactly one node in V_2, and vice versa. The matching M consists of $t + p$ edges. Due to

the construction of G (Definition 15) and Lemma 2 (a node $u_i \in \mathcal{T}_1$ only connect to a node $v_j \in \mathcal{T}_2$ with $i < j$ or to destination nodes $\mathcal{D}$), the matching M naturally decomposes into p paths starting from the source nodes $s_1, \ldots, s_p$ and ending at the destination nodes $d_1, \ldots, d_p$. Each path traverses a sequence of nodes corresponding to the nodes of the original tree T. Specifically, a path starting at s_i will match it to a node $u_a \in \mathcal{T}_2$. Then, the corresponding node of $u_a \in \mathcal{T}_1$ can be matched to a node $u_b \in \mathcal{T}_2$ (where $b > a$). This continues until a node $u_x \in \mathcal{T}_1$ is matched to a destination node $d_k \in \mathcal{D}$. This forms a sequence $s_i \to u_a \to u_b \to \cdots \to u_x \to d_k$. Following this technique, we will retrieve all the optimal chains from the solution of the MWPBM problem.

Example 21:
Consider the example in Figure 6.4. It shows a bipartite graph constructed from a tree (not shown) and a perfect matching within it. The solid arrows represent the edges of the perfect matching, where an edge (u, v) signifies that node v follows node u in a path. The dashed arrows link the segments of the paths by connecting a node's representation in V_2 to its corresponding representation in V_1.

The matching partitions the nodes into two distinct paths, differentiated by color:

- **Path 1 (red):** Starting from source s_1, the matching edge (s_1, A) leads to the first node, A. The dashed arrow from this node in V_2 points to the same node A in V_1, which is then matched with another A in V_2. Following the next dashed arrow to the final A in V_1, we see it is matched with the destination d_1. This sequence traces the path $s_1 \to A \to A \to d_1$ leading to the chain $C_{red} = \{A, A\}$.
- **Path 2 (blue):** Starting from source s_2, the matching edge (s_2, B) leads to node B. The dashed arrow connects to the next B in V_1, which is matched with destination d_2, tracing the path $s_2 \to B \to d_2$ leading to the chain $C_{blue} = \{B\}$.

This demonstrates how a perfect matching in the bipartite graph yields a valid partition of the original tree's nodes into paths from sources to destinations.

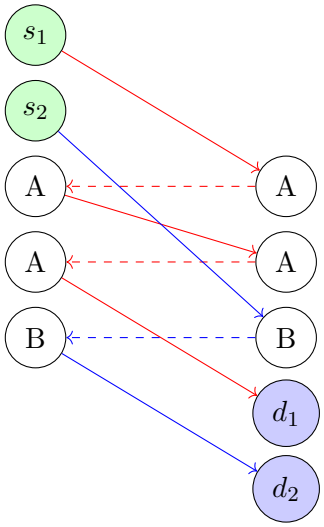


Figure 6.4: An example of a perfect matching (solid lines) in the constructed bipartite graph. The matching defines a partition into two paths (red and blue), which are traced by following the solid and dashed arrows.

Theorem 7. *An optimal solution of an instance $\mathcal{I}$ with $p \leq |\mathcal{C}|$ of the CHAINS-DIVISION problem (Definition 12) is equivalent to an optimal solution of the MWPBM problem (Definition 10) 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 MWPBM problem for a bipartite graph G constructed as stated in Definition 15.*

Proof. Let $\mathcal{I} = (T, \mathcal{C}, p)$ be an instance of the CHAINS-DIVISION problem, where T is a tree with t nodes, $\mathcal{C}$ is the set of equivalence classes, and p is the target number of chains. We assume $p \leq |\mathcal{C}|$, per Claim 1. Let $G = r(\mathcal{I})$ be the bipartite graph constructed according to Definition 15. We will demonstrate a bijection between the set of valid chain partitions of T and the set of perfect matchings in G, such that the cost of a partition equals the weight of its corresponding matching.

First, we establish the existence of a perfect matching. By construction, the graph G is bipartite with partitions V_1 and V_2 such that $|V_1| = |V_2| = t + p$. Lemma 6 ensures that a perfect matching exists in G.

Let $\mathcal{P} = \{C_1, \ldots, C_p\}$ be a valid partition of the nodes of T into p chains. We can construct a perfect matching $M_{\mathcal{P}}$ in G as follows: For each chain $C_k = [u_1, \ldots, u_{m_k}]$, we construct a path in G: match s_k to $u_1 \in \mathcal{T}_2$. Then match $u_i \in \mathcal{T}_1$ to $u_{i+1} \in \mathcal{T}_2$ for $i = 1, \ldots, m_k - 1$. Finally, match $u_{m_k} \in \mathcal{T}_1$ to one of the available destination nodes $d_j \in \mathcal{D}$. Since we have p chains and p source/destination nodes, and every tree node is in exactly one chain, this process uses all $t + p$ nodes in V_1 and V_2, forming a perfect matching. The weight of this matching is given by:

$$\begin{aligned} W(M_{\mathcal{P}}) &= \sum_{(u,v) \in M_{\mathcal{P}}} w(u, v) \\ &= p + |\{(u_i, u_j) \in M_{\mathcal{P}} \mid u_i \in \mathcal{T}_1, u_j \in \mathcal{T}_2, \mathcal{C}(u_i) \neq \mathcal{C}(u_j)\}| \end{aligned}$$

where p represents the contribution from the source nodes s_i, as each source node must be connected with weight 1 to start a chain. A class change occurs exactly when a path in the matching uses a weight-1 edge between tree nodes. Therefore,

$W(M)$ is exactly equal to the RLE cost of the partition defined by the matching $M_{\mathcal{P}}$.

Conversely, let M be a perfect matching in G. The structure of G ensures that M consists of p disjoint paths starting from source nodes $\{s_1, \ldots, s_p\}$ and ending at destination nodes $\{d_1, \ldots, d_p\}$. Each path defines an ordered chain of nodes from T. By Lemma 2, the node order within these chains is consistent with the original node ordering π. Thus, M maps to a valid partition of T. The cost of this partition is equal to $W(M)$.

Since there is a cost-preserving bijection between the set of all valid partitions and the set of all perfect matchings, an optimal solution to one problem corresponds to an optimal solution to the other. Therefore, finding a minimum weight perfect matching in G is equivalent to solving the CHAINS-DIVISION problem for T. □

Example 22:
Consider the example in Figure 6.5 where we have the bipartite graph for the tree in Figure 6.1 and $p = 2$. In Figure 6.5-(a) we have the resulting bipartite graph, and in Figure 6.5-(b) we have one of the possible minimum perfect matchings for the graph in (a) having weight 5. At the end we can see that the optimal partition of the nodes of the tree T is $C_1 = \{A, C, B, B, C\}$ and $C_2 = \{D, D, D, D, D, D\}$ with a total cost of 5, 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.

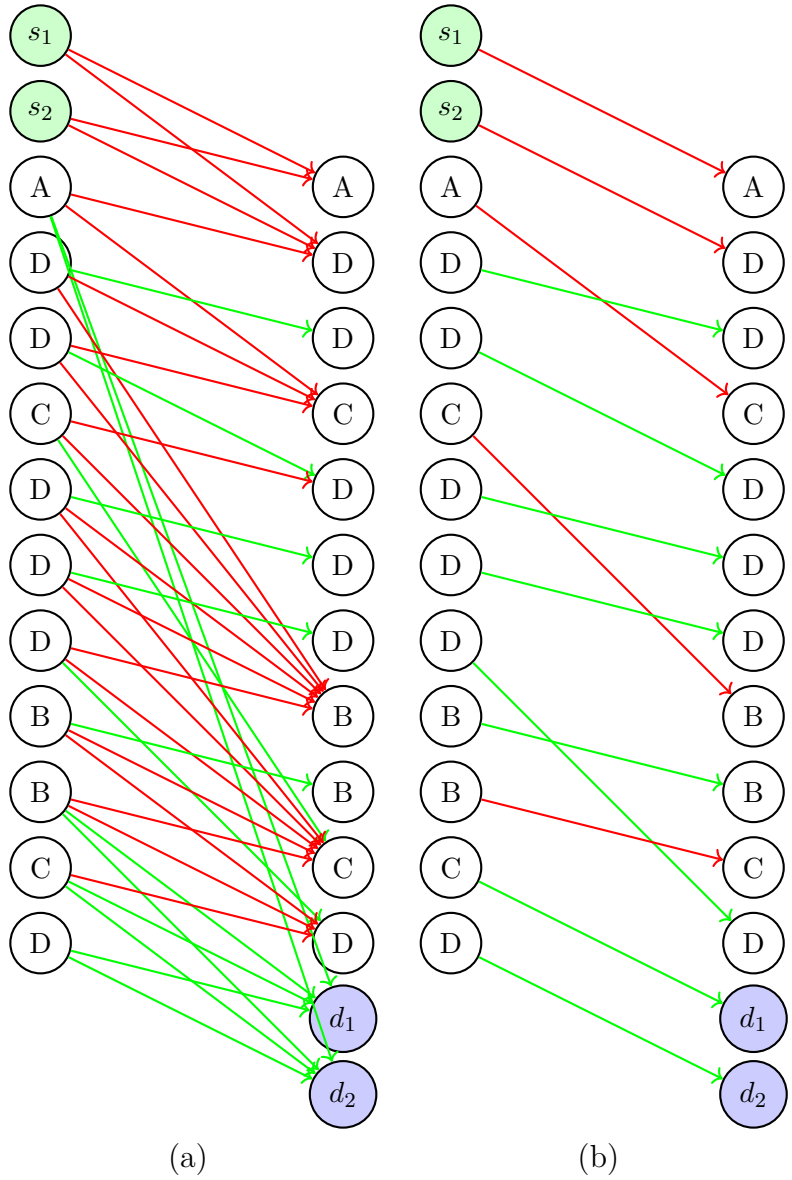


Figure 6.5: Example of a reduction for the tree in Figure 6.1. In (a), we have the resulting bipartite graph constructed. In (b), we have the resulting perfect matching for the graph in (a). Green edges weigh 0, while red edges weigh 1.

6.2.4 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.

Lemma 7 (Sources' edges optimization)**.** *The sources' edges can be optimized by connecting each source only to the smallest node in $\mathcal{T}_2$ 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 exactly one node in $\mathcal{T}_2$. Having the sources connected to the first p nodes with distinct equivalence classes in $\mathcal{T}_2$ is not necessary since it allows us just to invert the chains starting from each source, and so it is redundant.

Moreover, Hall's condition $|N(W)| \geq |W|$ still holds for any subset $W \subseteq V_1$ in the optimized graph. This follows trivially from the fact that case 1 of Lemma 6 still guarantees that $|N(W)| \geq |W|$ for any subset $W \subseteq \mathcal{S}$. □

Example 23:
Consider the bipartite graph shown in Figure 6.6. The red solid arrows represent edges with weight 1, while the red dashed arrows represent edges that can be optimized away according to Lemma 7.

In this example, we can observe the application of the optimization rule:

- The first tree node A has a solid edge to the first destination node A and a dashed edge to destination node B to be removed. This since source s_1 is already connected to node A.
- Similarly, the second source node s_2 has a dashed edge to the first destination node A, which can be removed because s_1 is already connected to node A with the same weight.

After applying the optimization, only the solid edges remain, resulting in a reduced bipartite graph that maintains the same optimal matching cost while having fewer edges to consider during the matching algorithm.

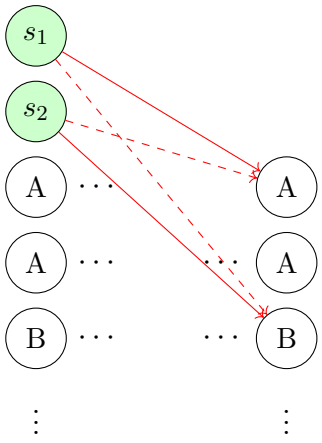


Figure 6.6: Bipartite graph after applying Sources' edges optimization. Dashed edges from the original graph have been removed according to Lemma 7, reducing the number of edges while preserving optimality. Red edges have weight 1.

This will reduce the number of edges coming from the sources from p^2 to p.

Lemma 8 (Tree nodes' edges optimization 1)**.** *The tree nodes' edges can be optimized by removing the edges of the nodes in $\mathcal{T}_1$ that are connected to nodes in $\mathcal{T}_2$ already linked to a source node in V_1.*

Proof. This optimization follows directly from Lemma 7. From Definition 9 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 $\mathcal{T}_2$. Therefore, we can remove the edges of the nodes in $\mathcal{T}_1$ that are connected to nodes in $\mathcal{T}_2$ already linked to a source node in V_1 since they will not be part of the final matching. □

Example 24:
Consider the bipartite graph shown in Figure 6.7. The red arrows represent edges with weight 1, while green arrows represent edges with weight 0. Dashed edges represent edges that can be optimized away according to Lemma 8.

In particular, in Figure 6.7, we can observe that the dashed edge from the first node A in $\mathcal{T}_1$ can be optimized away since it is connected to B in $\mathcal{T}_2$, which is already linked to a source node in V_1.

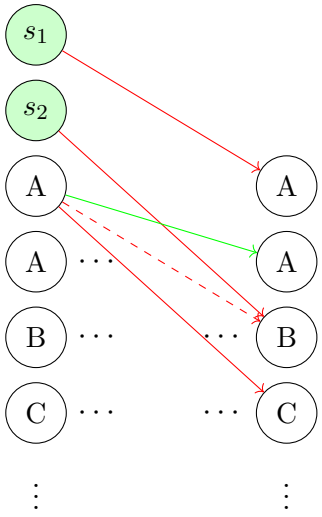


Figure 6.7: Bipartite graph after applying tree nodes' edges optimization 1. Dashed edges from the original graph have been removed according to Lemma 8, reducing the number of edges while preserving optimality. Red edges have weight 1, while green edges have weight 0.

This will reduce the number of edges by a factor of $p - 1$.

Lemma 9 (Tree nodes' edges optimization 2)**.** *Edges of nodes in $\mathcal{T}_1$ can be optimized by removing the edges with weight 1 starting from a node $u \in \mathcal{T}_1$ to a node $v \in \mathcal{T}_2$ if the node u has another edge with weight 0 connected to a node $z \in \mathcal{T}_2$ such that $z \prec v$ in the ordering of the nodes.*

Proof. Let M be any perfect matching. Suppose M contains an edge (u, v) that satisfies the conditions of the lemma, i.e., $u \in \mathcal{T}_1$, $v \in \mathcal{T}_2$, $w(u, v) = 1$, and there exists an edge (u, z) with $w(u, z) = 0$ for some $z \in \mathcal{T}_2$ with $z \prec v$.

We will show that this edge (u, v) is not necessary for an optimal solution. Since M is a perfect matching, z must be matched with some node $u' \in \mathcal{T}_1$, so $(u', z) \in M$. Note that $u' \neq u$ and that $u' \prec z$ for Lemma 2.

Consider an alternative matching $M' = (M \setminus \{(u, v), (u', z)\}) \cup \{(u, z), (u'', v)\}$. This is a valid perfect matching where u is matched with z, and $u'' \in \mathcal{T}_1$ is matched with v. By construction and from Lemma 2 we know that $u \prec u' \preceq u'' \prec v$.

The weight of this new matching is $W(M') = W(M) - w(u, v) - w(u', z) + w(u, z) + w(u'', v)$. By substituting the known weights $w(u, v) = 1$ and $w(u, z) = 0$, we get: $W(M') = W(M) - 1 - w(u', z) + 0 + w(u'', v) = W(M) + (w(u'', v) - w(u', z)) - 1$.

The construction of the graph ensures that for any node $u'' \in \mathcal{T}_1$, the cost of connecting to v is either the same or one greater than connecting to a preceding node z. That is, $w(u'', v) - w(u', z) \in \{-1, 1\}$. This property arises from the problem

reduction, where moving to a subsequent node in the ordering can at most increment the cost by one.

Therefore, $w(u'', v) - w(u', z) \leq 1$, which implies $W(M') \leq W(M)$. Thus, the edge (u, v) can be removed from the graph without affecting the weight of the optimal solution.

Finally, we need to prove that Lemma 6 still holds. The proof follows from showing that the two fundamental neighborhood properties remain valid after edge removal:

1. Lemma 4 remains valid because it relies on the fact that each node $u_i \in \mathcal{T}_1$ is connected to $v_{i+1} \in \mathcal{T}_2$. The edge (u, v) we remove cannot be this critical edge (u_i, v_{i+1}) since, by our optimization condition, there exists a node $z \prec v$ connected to u with weight 0; the edge (u_i, v_{i+1}) is always the first valid connection for u_i in the ordering. Therefore, v cannot be v_{i+1} for the node u.
2. Lemma 5 is preserved because it concerns edges between nodes of the same equivalence class. Our optimization only removes an edge when there exists a better alternative to a preceding node, meaning that the same-class connectivity pattern is unaffected by this edge removal.
3. The edge removal does not affect source and destination nodes' connections, as we only remove tree node edges.

Since both Lemmas 4 and 5 remain valid and the source and destination connectivity is preserved, all conditions required by Lemma 6 continue to hold, ensuring the existence of a perfect matching in the optimized graph.

Alessio: La questione è che: se da un nodo con valore 1 voglio connettermi ad un nodo con valore 3, ma prima c'è un altro nodo con valore 1, tanto vale pasó che comunque ci sarà un collegamento tra quello nuovo e li 3 (da dimostrare). In questo modo, prendendo quel nodo, il costo della mia catena aumenta di 0, quindi non peggiora, e il coto di un altra catena potrebbe non aumentare, quindi togliamo una scelta ovviamente errata. **Davide T.:** ho provato a renderla più generale. così mi sembra funzionare □

Example 25:
Consider the bipartite graph in Figure 6.8. The red arrows represent edges with weight 1, while green arrows represent edges with weight 0. Dashed edges represent edges that can be optimized away according to Lemma 9.

In this case, we can observe the application of the optimization rule: The edge (A, C) is removed from the graph, as it is not necessary for an optimal solution. This is because the node A has another edge with weight 0 connected to a node B with $B \prec C$.

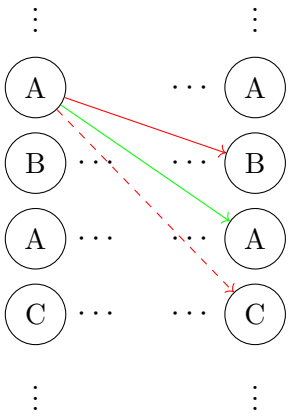


Figure 6.8: Bipartite graph after applying tree nodes' edges optimization 2. Dashed edges from the original graph have been removed according to Lemma 9, reducing the number of edges while preserving optimality. Red edges have weight 1, while green edges have weight 0.

6.2.5 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 8. *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 $r : \mathcal{I}_{CHAINS-DIVISION} \rightarrow \mathcal{I}_{MWPBM}$ 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 15 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 15. 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. □

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974 (cit. on p. 34).

[2] M. Burrows and D. Wheeler. *A block-sorting lossless data compression algorithm*. Tech. rep. 124. Digital Equipment Corporation, 1994 (cit. on pp. 10, 17).

[3] 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 p. 44).

[4] 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. 46).

[5] 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. 43).

[6] 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. 43).

[7] 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, 2, 9, 13).

[8] 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. 43).

[9] 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 pp. 40, 43).

[10] 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. 24).

[11] 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. 24).

[12] Philip Hall. "On representatives of subsets". In: *Journal of the London Mathematical Society* 10.1 (1935), pp. 26–30 (cit. on pp. 41, 42).

[13] 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 p. 30).

[14] 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. 13, 15).

[15] 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. 1).

[16] 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. 43).

[17] 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).

[18] 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).

[19] 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).

[20] John Myhill. *Finite automata and the representation of events*. Tech. rep. WADC Technical Report 57-624. Wright Air Development Center, 1957 (cit. on pp. 31, 32).

[21] Anil Nerode. "Linear automaton transformations". In: *Proceedings of the American Mathematical Society* 9.4 (1958), pp. 541–544 (cit. on pp. 31, 32).

[22] 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. 24).

[23] Dominique Revuz. "Minimisation of acyclic deterministic automata in linear time". In: *Theoretical Computer Science* 92.1 (1992), pp. 181–189 (cit. on pp. 32–34).

[24] Piotr Sankowski. "Maximum weight bipartite matching in matrix multiplication time". In: *Theoretical Computer Science* 410.44 (2009), pp. 4480–4488 (cit. on p. 44).

Web bibliography

[9] 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 pp. 40, 43).