Università
Ca'Foscari
Venezia

Artificial Intelligence and Data Engineering

Master Degree Thesis

A New Compression Technique for Repetitive Tries

Supervisor
Nicola Prezza

Co-supervisor
Alessio Campanelli

Author
Davide Tonetto
Student ID 884585

Academic year
2024/2025

Abstract

Contents

Chapter 1

Introduction

The problem of compressing large sets of strings, or finite languages, is a fundamental challenge in computer science with applications in areas like bioinformatics, natural language processing, and data indexing. A finite language can be naturally represented by an acyclic deterministic finite automaton (ADFA). A special case of ADFAs are tries. In this representation, each string in the language corresponds to a unique path from the root to a final state. Compressing the language is therefore equivalent to compressing its corresponding trie structure.

Traditional compression algorithms often fail to exploit the inherent structural properties of tries. To address this, specialized techniques have been developed. Among the most prominent is the *Extended Burrows-Wheeler Transform (XBWT)* [12], which extends the classical Burrows-Wheeler Transform [4] to labeled trees and can be applied to tries to achieve significant compression by capturing their structural regularities.

However, existing techniques may not be optimal when dealing with tries that exhibit a high degree of repetitiveness. Such is the case for languages containing many strings with shared substrings, leading to tries with large, identical subtrees. Many real-world datasets, such as genomic databases or dictionaries of related terms, generate such highly repetitive structures. This thesis introduces and analyzes a novel compression technique specifically designed to exploit these repetitions. The core idea is to identify and merge identical subtrees by reducing the trie to its minimal deterministic finite automaton (DFA) representation. We implement this method and evaluate its performance against state-of-the-art approaches like XBWT, assessing its effectiveness on various datasets.

1.1 Challenges and Contributions

The primary goal of this thesis is to develop a data structure that both compresses a given finite language and efficiently supports indexing queries, such as navigational and subpath queries (see Definition 12). This challenge involves navigating a fundamental trade-off between compression and indexability, which we explore in detail in Section 2.6. Two straightforward approaches highlight the extremes of this spectrum:

- **Full Compression, Difficult Indexing:** One could minimize the input trie into the smallest possible equivalent DFA using an algorithm like Revuz's [32]. While this yields optimal compression through DAG compression (see Section 2.1), indexing the resulting general DFA is a notoriously difficult problem. As shown by Equi et al. [11], pattern matching on general DFAs requires super-polynomial time unless the Strong Exponential Time Hypothesis (SETH) fails,

making this approach unsuitable for most indexing purposes.

- **Full Indexability, No Compression:** At the other extreme, the input trie itself can be used as an index. Tries are Wheeler graphs [14], specifically 1-sortable automata (Definition 21), a property that makes them highly amenable to efficient indexing [7]. While this provides excellent query performance through the co-lexicographic ordering of states (see Definition 23), it offers no compression, as even highly repetitive subtrees are stored explicitly.

This thesis proposes a novel algorithm that finds a sweet spot in this trade-off, which we develop throughout Section 2.4. The central idea is to partially minimize the input trie while ensuring the resulting automaton remains efficiently indexable. We achieve this by leveraging the theory of p-sortable graphs (see Section 2.6), developing a method that strategically increases the sortability parameter p just enough to enable significant compression. The motivation for this approach is rooted in the observation that a small increase in p can lead to substantial compression. As noted by Policriti et al. [27], there are cases where increasing p from 1 to 2 allows for an exponential reduction in the automaton's size, a phenomenon we explore in detail in Section 2.6.

Our compression scheme, presented in Section 2.4, works by partitioning the trie's nodes into a predefined number p of chains and then optimizing this partition to merge the maximum number of equivalent states while ensuring p-sortability. To make this optimization problem more concrete, we can frame it as a string partitioning problem. Consider the sequence of nodes in the trie, when read in co-lexicographic order (see Definition 23), as a single long string. The "character" corresponding to each node is its Myhill-Nerode equivalence class (see Theorem 3), which determines if it can be merged with other nodes. The task is to partition this string of nodes into p substrings such that the number of runs (Definition 1) is minimized. Minimizing the number of runs directly corresponds to maximizing the number of merged states, yielding a compact p-sortable automaton. We call this problem the String Partitioning Problem (Definition 4).

Definition 1 (Run)**.** *Let $S = s_1 s_2 \dots s_n$ be a string over an alphabet Σ. A substring $S[i..j] = s_i s_{i+1} \dots s_j$ (where $1 \leq i \leq j \leq n$) is a run if it satisfies the following conditions:*

1. ***Homogeneity:*** *All characters in the substring are identical, i.e., $s_k = s_i$ for all k such that $i \leq k \leq j$.*
2. ***Maximality:*** *The substring cannot be extended to the left or right without violating homogeneity. That is:*
   - *If $i > 1$, then $s_{i-1} \neq s_i$.*
   - *If $j < n$, then $s_{j+1} \neq s_j$.*

Definition 2. *For a string S over an alphabet Σ, and a set $I = \{i_1, i_2, \dots, i_k\} \in [n]$ with $i_1 < i_2 < \dots < i_k$, we define $S[I] := s_{i_1} s_{i_2} \dots s_{i_k}$ as the substring indexed by I.*

Definition 3. *Let $\tau(S)$ be the number of runs in a string S, i.e. $\tau(S) = 1 + |\{i \in [n-1] : S_i \neq S_{i+1}\}|$.*

Definition 4 (String Partitioning Problem)**.** *Let $\mathcal{I} = (S, p)$ be an instance of the String Partitioning Problem where S is a string of length n over an alphabet Σ, and p is a positive integer. The output of the problem is a partition $I_1, \ldots, I_p$ of $[n]$ such that $\sum_{i=1}^{p} \tau(S[I_i])$ is minimized.*

Example 1.1: String Partitioning

Let us consider the string $S = \texttt{2213122152}$ of length 10. The number of runs in S is 8, given by the decomposition `(22)(1)(3)(1)(22)(1)(5)(2)`. We want to partition the set of indices $\{1, \ldots, 10\}$ into two sets, I_1 and I_2, to minimize the total number of runs in the corresponding substrings.

A possible partition is:

- $I_1 = \{3, 5, 8, 9\}$
- $I_2 = \{1, 2, 4, 6, 7, 10\}$

This partition yields the following substrings:

- The substring corresponding to I_1 is $S[I_1] = \texttt{1115}$. This substring has 2 runs: (`111`) and (`5`).
- The substring corresponding to I_2 is $S[I_2] = \texttt{223222}$. This substring has 3 runs: (`22`), (`3`), and (`222`).

The total number of runs for this partition is $2 + 3 = 5$. This is a reduction from the original 8 runs in S. This example illustrates how partitioning a string's indices can reduce the total number of runs in the resulting substrings.

This thesis shows that the String Partitioning Problem can be reduced to the problem of finding a minimum weight perfect matching on a bipartite graph (MWPBM), allowing us to use efficient, well-studied algorithms to find the optimal solution. The result is a compressed automaton that is p-sortable by construction and thus supports efficient queries using the data structure developed by Cotumaccio et al. [7]. As our experimental results in Chapter 4 will show, this method is particularly effective for the highly repetitive datasets common in real-world applications, achieving a balance of compression and indexability that prior methods could not attain.

1.2 Structure of The Thesis

Chapter 2

Preliminary Concepts

2.1 Basic notation and concepts

Alessio: As of now this is a stash of general stuff that is important to explain in a general section.

Definition 5 (String)**.** *A **string** is a sequence of characters $S = s_1 s_2 \dots s_n$ drawn from an alphabet Σ.*

We use $|S| = n$ to denote the lengthh of S. The set of all strings is represented as Σ^*, which also contains the empty string ε.
The notation $S[i] = s_i$ denotes the i-th character of S, and $S[i..j]$ the substring $s_i s_{i+1} \dots s_j$, for $1 \leq i, j \leq n$. Therefore, a prefix of S is a substring of the type $S[1..j]$, while a suffix is a substring $S[i..n]$.

Alessio: Explain subsequences.

Definition 6 (Tree)**.** *A **rooted tree** is a triple $T = (V, E, r)$ where:*

- *V is a finite set of vertices (or nodes),*
- *$E \subseteq V \times V$ is a set of edges such that $|E| = |V| - 1$ and the underlying graph is connected and acyclic,*
- *$r \in V$ is the root vertex.*

Alessio: $t = |V|$ **Alessio:** Add notation and definitions relative to general trees here (depth, degree, children, leaves, subtree, etc...).

Before computing the information-theoretic lower bound for labeled trees, it is essential to define the concept of worst-case entropy, which provides a formal measure of the minimum number of bits required to represent any object from a given set. As detailed by Navarro [29], this is a fundamental concept in data structure design.

Definition 7 (Worst-case entropy)**.** *Let U be a universe of combinatorial objects. The worst-case entropy of U is*

$$H_{wc}(|U|) = \lceil \log_2(|U|) \rceil$$

This definition establishes that the minimum number of bits to uniquely identify any object in a set U is the logarithm of the size of U, rounded up to the nearest integer. We now apply this principle to determine the lower bound for labeled trees.

2.2 Finite State Automata

Finite automata are ...

Let $L \subseteq \Sigma^*$ be a finite set of strings. L can be represented in many different ways, such as enumeration, context-free grammars, regular expressions, or automata.

Definition 8 (Non-deterministic Finite Automaton)**.** *A* ***non-deterministic finite automaton*** *(NFA) is a 5-tuple* $N = (Q, \Sigma, \delta, q_0, F)$ *where:*

- Q *is a finite set of states,*
- Σ *is a finite alphabet,*
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ *is the transition function, where* $\mathcal{P}(Q) = \{A | A \subseteq Q\}$ *is the powerset of* Q*,*
- $q_0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is the set of final (accepting) states.*

An NFA processes an input string $S \in \Sigma^*$ one symbol at a time, starting from q_0 and following the transitions specified by δ. Let $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ be the extension of δ to strings. Then, for all $u \in Q$, $a \in \Sigma$, and $\alpha \in \Sigma*$:

$$\hat{\delta}(u, \varepsilon) = \{u\}, \qquad \hat{\delta}(u, \alpha a) = \bigcup_{v \in \hat{\delta}(u,\alpha)} \delta(v, a).$$

Therefore, $\delta(q_0, \alpha)$ denotes the set of states that can be reached from the start state q_0 by reading α. A string S is *accepted* if $\delta(q_0, S) \cap F \neq \emptyset$, or, in other words, if the automaton ends in any state $q \in F$ after processing the entire string. The set of all strings accepted by N is called the *language* of the automaton, and is denoted as

$$\mathcal{L}(N) = \{S \in \Sigma^* \mid \delta(q_0, S) \cap F \neq \emptyset\}.$$

Alessio: Maybe explain that automata can be represented labeled directed graphs, and add an example with 2 subfigures (a) NFA, (b) DFA, that accept the same language

Deterministic Finite Automata are a special case of NFAs, where each state has exactly one outgoing transition per character, i.e., $|\delta(q, a)| = 1$.

Definition 9 (Deterministic Finite Automaton)**.** *A* ***deterministic finite automaton*** *(DFA) is a 5-tuple* $D = (Q, \Sigma, \delta, q_0, F)$ *where:*

- Q *is a finite set of states,*
- Σ *is a finite input 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.*

Notice how the only difference between Definition 8 and Definition 9 is the return value of the transition function. As a consequence, the expansion of δ to strings can be simplified to

$$\hat{\delta}(u, \alpha a) = \delta(\hat{\delta}(\alpha), a)$$

Similarly to NFAs, the language of a DFA is the set of strings that end at a state $q \in F$ after being processed:

$$\mathcal{L}(D) = \{S \in \Sigma^* \mid \hat{\delta}(q_0, a) \in F\}$$

Alessio: Explain why DFAs are better than NFAs for indexing reasons and with respect to the objective of the thesis. Also mention disadvantages.

2.2.1 Minimization

A fundamental prerequisite for these algorithms is determinism, as they require that from any given state, each symbol corresponds to at most one transition. Tries inherently provide this guarantee, since from any node, there is at most one outgoing edge for each symbol in the alphabet. This deterministic nature is therefore what allows us to apply powerful automata minimization techniques to compress the tree structure.

2.3 Labeled Trees and Tries

Before delving into specific compression techniques, it is essential to establish a solid theoretical foundation regarding labeled trees and, in particular, tries. These structures are fundamental for representing hierarchical data across diverse fields, from bioinformatics to document processing. This chapter provides the necessary background, defining these two data structures, exploring their common applications, and introducing the core concepts behind their compression and indexing.

2.3.1 Definition

Alessio: Why the definition of the LT does not have Σ, while the NFA/DFA do?

Definition 10 (Labeled tree)**.** *A* ***labeled tree*** *is a quadruple* $T = (V, E, \lambda, r)$*, where:*

- V*,* E*, and* r *follow the same definition of the rooted tree (Definition 6),*
- $\lambda : E \to \Sigma$ *is an edge-labeling function.*

In the case of *ordered* labeled trees, the children of each node are totally ordered, while the degree and shape of the tree, as well as the size of the alphabet Σ, are unconstrained.

While labeled trees encompass a broad class of hierarchical structures, this thesis focuses specifically on tries. Tries are a special case of DFAs, restricted to a tree-shaped structure, and are therefore widely used in Computer Science for representing finite sets of strings.

Definition 11 (Trie)**.** *A **trie** is a 5-tuple $T = (V, E, \lambda, r, F)$ where:*

- *V is a finite set of vertices,*
- *$E \subseteq V \times V$ is a set of edges such that (V, E) forms a rooted tree,*
- *$\lambda : E \to \Sigma$ is an edge-labeling function,*
- *$r \in V$ is the root vertex,*
- *$F \subseteq V$ is the set of final (accepting) vertices.*

Notice that, when expressing a trie as a DFA, each node should have a child for each character of the alphabet Σ. However, for ease of notation and representation, we will only consider nodes and edges that will eventually lead to a final state. For this reason, our definition of trie is closer to the definition of labeled trees rather than DFAs.

Key properties of a trie include:

1. **Determinism**: For every vertex $v \in V$ and every symbol $a \in \Sigma$, there is at most one edge $(v, u) \in E$ such that $\lambda((v, u)) = a$.
2. **String Representation**: For any vertex $v \in V$, let $str(v)$ be the string obtained by concatenating the labels on the unique path from the root r to v. We set $str(r) = \varepsilon$.
3. **Language Correspondence**: The language represented by the trie is exactly L, i.e., $L = \{str(v) \mid v \in F\}$.
4. **Prefix Property**: The set of all prefixes of words in L coincides with $\{str(v) \mid v \in V\}$.

The determinism property ensures that each string prefix identifies a unique path from the root, which makes tries well suited for our compression techniques. **Alessio:** You should briefly explain also why the other properties are important.

Alessio: Which is the cause and the effect? We want to compress subtrees so we rely on trie properties, or we want to compress repetitive texts, which can be represented with tries, therefore the compression is like merging the subtrees? This focus on tries is motivated by our compression strategy, which relies on the identification and merging of identical subtrees. To formally identify these subtrees, we compute the Myhill-Nerode equivalence classes of the trie's states (see Theorem 3), a task that can be accomplished using algorithms for DFA minimization.

Given that a trie is a special case of DFA, we can apply automaton minimization techniques to compress its structure. This process, known as **DAG compression** [cite section/papers], consists of transforming a tree into a directed acyclic graph (DAG) by identifying and merging its isomorphic subtrees. In the context of tries, this is equivalent to finding and merging states with the same right language, which is the standard procedure in DFA minimization. The result is the smallest possible automaton that recognizes the same language. **Alessio:** What is the right language?

2.3.2 Applications

Alessio: This subsection should be revised, as the examples provided use node labeling instead of edge labeling. **Alessio:** Also consider moving this to the introduction, to show why this problem is important. 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.

2.3.3 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 optimal (i.e., $O(1)$) or (near-)optimal time.

We focus on the following operations on labeled trees, as they will be useful in future sections:

Definition 12 (Tree Operations)**.** *Given a labeled tree $T = (V, E, \lambda, r)$, a node $u \in V$, and a symbol $s \in \Sigma$, we define the following fundamental operations:*

- ***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 s.*
- ***Visualization queries:*** *retrieve the nodes in the subtree rooted at u (any possible order should be implemented).*
- ***Subpath queries:*** *given a word $\alpha \in \Sigma^*$, let $\hat{\delta}(q_0, \alpha)$ be the set of vertices reached by processing α in a NFA fashion. Subpath queries can solve:*
  - *Existence: determine whether $\hat{\delta}(q_0, \alpha) \neq \emptyset$, i.e., check if it exists a path in T obtained by following the symbols of α.*

- *Counting: compute the size of* $\hat{\delta}(q_0, \alpha)$.
- *Location: return a representation of all the vertices in* $\hat{\delta}(q_0, \alpha)$. **Alessio:** *What do you mean with representation?*

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 $O(t \log t)$ bits. However, this representation is not space-efficient and does not support fast query operations. In order to implement fast, compressed tree indexes, we first have to understand which are the information-theoretic lower bounds for their lossless representation.

Lemma 1. *The information-theoretic lower bound for storing a labeled tree* T *with* t *nodes over an alphabet* Σ *is* $2t + t \log_2 |\Sigma| - \Theta(\log t)$ *bits.*

Proof. The total information required to store a labeled tree can be decomposed into two components: the space needed to encode the tree's structure and the space needed to encode the labels on its nodes.

1. Structural Information (Unlabeled Tree): The number of distinct unlabeled binary trees with t nodes is given by the t-th Catalan number, $C_t = \frac{1}{t+1}\binom{2t}{t}$. Using Stirling's approximation for factorials, the Catalan number can be approximated as:

$$C_t \approx \frac{4^t}{t^{3/2}\sqrt{\pi}}$$

Then, the worst-case entropy (or the information-theoretic minimum number of bits to encode the structure of the tree) is:

$$\log_2 C_t \approx \log_2 \left(\frac{4^t}{t^{3/2}\sqrt{\pi}} \right) = \log_2(4^t) - \log_2(t^{3/2}\sqrt{\pi}) = 2t - \frac{3}{2}\log_2 t - \frac{1}{2}\log_2 \pi$$

The lower-order terms can be expressed using Big Theta notation as $\Theta(\log t)$. Therefore, the space required for the structure is $2t - \Theta(\log t)$ bits.

2. Labeling Information: For a tree with t nodes and an alphabet Σ, each node must be assigned a label. To distinguish between $|\Sigma|$ possible labels, a minimum of $\log_2 |\Sigma|$ bits is required for each node. Consequently, the total space required to store the labels for all t nodes is:

$$t \log_2 |\Sigma| \text{ bits}$$

Finally, by adding the space required for the structure and the labels, the total information-theoretic lower bound for storing a labeled tree is the sum of the two components:

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

This completes the proof. □

2.3.4 State of The Art

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 (Subsection 2.3.4).

The field of tree indexing and compression has evolved through two main paradigms: succinct data structures that achieve space-optimal representations, and compression techniques that exploit structural repetitions.

In the realm of succinct tree structures, early work by Kosaraju [22] proposed a method to index labeled trees by extending the concept of prefix sorting from strings to labeled trees using trie structures. He introduced the idea of constructing a suffix tree for a reversed trie, enabling subpath queries in $O(|P| \log |\Sigma| + occ)$ time, where occ is the number of occurrences of P in T. However, this approach still required $O(t \log t)$ space (where t is the number of nodes of the tree) and thus was not compressed.

A significant advancement in this direction came with the Extended Burrows-Wheeler Transform (XBWT) [12], 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 capturing the structural properties of the tree and 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 navigation, visualization and subpath queries (see Definition 12), within (near-)optimal time bounds and entropy-bounded space. The XBWT provides significant improvements in both compression ratio and query performance compared to traditional compression schemes, making it a valuable resource for intensive applications.

Complementing succinct approaches, tree compression has been extensively studied through different paradigms that exploit structural repetitions in distinct ways. One of the classical approaches is *DAG compression*, which represents a tree as a minimal directed acyclic graph (DAG) by identifying and merging identical rooted subtrees. Concretely, whenever two identical subtrees occur, only one copy is stored and all occurrences point to it. The resulting structure can be exponentially smaller than the original tree and can be computed in linear time. DAG compression has been widely used in programming languages, binary decision diagrams, and XML representations [3].

Another line of research extends the well-known LZ77 factorisation from strings to trees. Here, the tree is decomposed into edge-disjoint fragments, each being either a single node or a copy of a fragment that appeared earlier in a breadth-first traversal. Each fragment is thus defined by pointers to earlier occurrences, much like in the string version of LZ77. This factorisation uniquely determines the tree, and by minimising the number of fragments one obtains a compressed representation. Importantly, such factorizations can be computed in polynomial time (and in linear time for restricted variants), and they yield representations no larger than the smallest tree grammar, thus bridging block compression and

grammar-based compression [15].

More recently, top tree compression has been proposed as a method that combines the advantages of subtree sharing and grammar-like approaches. The key idea is to build a hierarchical top tree decomposition, where the input tree is recursively partitioned into clusters that capture connected patterns. These clusters are then merged following a restricted set of operations, producing a binary decomposition tree whose internal repetitions are turned into subtree repeats. Finally, this decomposition is compressed using standard DAG compression, resulting in a so-called top DAG. This approach achieves close-to-optimal worst-case bounds, can be exponentially more succinct than DAG compression, and crucially, supports a wide range of navigational queries (e.g., parent, child, depth, nearest common ancestor) in logarithmic time directly on the compressed representation [3].

Another notable approach is Tree Re-Pair [25], a grammar-based compression technique adapted for tree structures. It extends the principles of the original Re-Pair algorithm [24] 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 [26].

In summary, DAG compression is efficient but limited to subtree repeats, LZ77 factorisation captures more general repetitions while relating closely to grammar-based methods such as Tree Re-Pair, and top tree compression strikes a balance by exploiting both subtree and pattern repeats while still enabling efficient query support.

This thesis focuses on developing a novel technique specifically tailored for highly repetitive tries. Our approach leverages the structural properties unique to tries and their repetitive patterns. Therefore, we use the XBWT as our primary benchmark for comparison, as it represents a well-established and high-performance baseline specifically designed for trie compression in the field.

2.4 The Extended Burrows-Wheeler Transform

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 Applica-

tions' [12].

2.4.1 Introduction and Motivation

This chapter explores the Extended Burrows-Wheeler Transform (XBWT), introduced by Ferragina et al. [12], a state-of-the-art technique for labeled tree compression. Our interest in the XBWT has many reasons.

First, it provides a powerful theoretical foundation for indexing labeled trees and tries. Tries are a special case of a broader class of graphs known as Wheeler graphs, and the XBWT can be understood as a compressed index specifically for this special case. This context is crucial, as the work of Cotumaccio et al. [6] later generalized the principles of the XBWT to the larger class of p-sortable graphs.

Second, the XBWT serves 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 and theoretically significant 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. [12] 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 capturing the structural properties of the tree and 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 navigation, visualization and subpath queries (see Definition 12), within (near-)optimal time bounds and entropy-bounded 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.

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 worst-case entropy (Lemma 1), 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 the navigational queries (Definition 12) in optimal time $O(1)$ if $|\Sigma| = O(polylog(t))$, otherwise in $O(\log \log^{1+\epsilon} |\Sigma|)$ time. Whereas, given $s \in \Sigma^*$, subpath queries (Definition 12) are supported in $O(|s|)$ if $|\Sigma| = O(polylog(t))$, otherwise in $O(|s| \log \log^{1+\epsilon} |\Sigma|)$ 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.

2.4.2 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) [4] 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 13. *Let T be a totally 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 in the total order, and 0 otherwise.*
- *$\alpha(u)$: denotes the label of node u concatenated with 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. Formally, we define $\pi(u)$ recursively as:*

$$\pi(u) = \begin{cases} \varepsilon & \text{if } u \text{ is the root} \\ label(parent(u)) \circ \pi(parent(u)) & \text{otherwise} \end{cases}$$

where $\circ$ is the concatenation operator and ε is the empty string.

If the leaf-label alphabet Σ_L is disjoint from the internal-node label alphabet Σ_N, the additional indicator bit in $\alpha(u)$ is redundant and may be omitted. In that case, we identify $\alpha(u)$ with the node label, i.e., $\alpha(u) := label(u)$.

The definition of the XBWT relies on a sequence 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 vector of triplets is created by traversing the tree T in pre-order and generating a triplet $(last(u), \alpha(u), \pi(u))$ for each node. Second, it is stably sorted according to the lexicographical order of the 'π' component to produce the final sequence S (see Table 2.1 for an example).

Theorem 1. *The XBWT of a labeled tree T consists of two arrays, S_{last} and S_α. These are constructed from the sequence 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. See Figure 2.1 for an example of the tree T and its corresponding sequence S.

2.4.3 Properties

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

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 sequence 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 2.1:**
> Consider the node u in Figure 2.1. Looking at Table 2.1, we can see that its children form a contiguous block in positions $[5, 6, 7]$ of the sequence 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 2.2:**
> Consider the node u in Figure 2.1. Looking at Table 2.1, 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 2.3:**
> Consider nodes u and v in Figure 2.1. In Table 2.1, node u appears at index 2 while node v appears at index 4 in the sequence 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 2.4:
Let's examine nodes u and v in Figure 2.1, both labeled 'B'. In the sequence S shown in Table 2.1, 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 $S_{\text{last}}[5 \dots 8]$.

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.

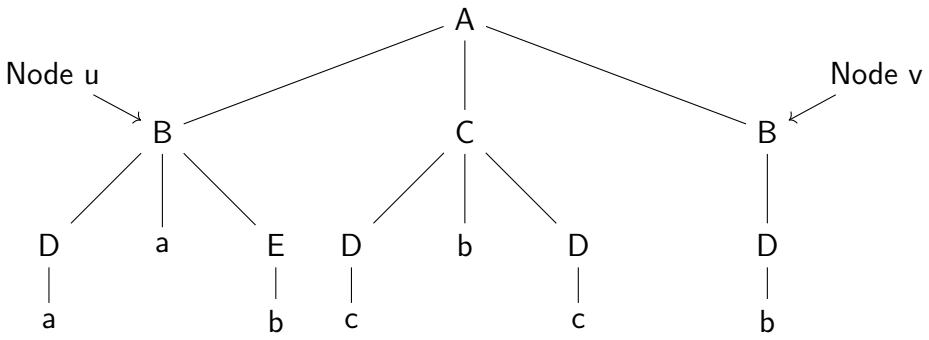


Figure 2.1: A labeled tree T where $\Sigma_N = \{A, B, C, D, E\}$ and $\Sigma_L = \{a, b, c\}$. Notice that $\alpha(u) = \alpha(v) = B$ and $\pi(u) = \pi(v) = A$.

	$S_{\mathbf{last}}$	S_α	S_π
1	0	A	ϵ
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 2.1: The sequence S for the tree shown in Figure 2.1, 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.

2.4.4 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 deep trees. To overcome this issue, Ferragina et al. [12] proposed a more efficient algorithm that builds S in linear time and $O(t \log t)$ bits of working space.

The linear time algorithm is called **pathSort**, it is based on a generalization of the Skew algorithm for suffix array construction of strings [20]. Let's see briefly how the Skew algorithm works.

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

Step 0: Construct a Sample. For $k \in \{0, 1, 2\}$, define the index sets

$$B_k \;=\; \{\, i \in [0, n] \mid i \bmod 3 = k \,\}.$$

Let $C = B_1 \cup B_2$ be the set of sample positions, and let S_C denote the set of sample suffixes.

Step 1: Sort Sample Suffixes. For $k \in \{1, 2\}$, construct the string R_k whose characters are the triples $[\, t_i \, t_{i+1} \, t_{i+2} \,]$ for $i \in B_k$ in increasing order of i. The last character of R_k is unique because we pad with sentinels so that $t_{n+1} = t_{n+2} = 0$. Let $R = R_1 R_2$ be the concatenation of R_1 and R_2. Then the nonempty suffixes of R correspond to the sample suffixes in S_C in an order-preserving way: sorting the suffixes of R yields the order of S_C. To sort the suffixes of R, first radix sort the characters of R (the triples) and rename them by their ranks to obtain a string R'. If all characters are distinct, their order directly gives the order of suffixes. Otherwise, sort the suffixes of R' recursively using the same DC3 procedure. Once S_C is sorted, assign ranks to sample suffixes: for $i \in C$, let $\text{rank}(S_i)$ be the rank of S_i within S_C. Additionally, define $\text{rank}(S_{n+1}) = \text{rank}(S_{n+2}) = 0$. For $i \in B_0$, $\text{rank}(S_i)$ is undefined.

Step 2: Sort Nonsample Suffixes. Represent each nonsample suffix S_i with $i \in B_0$ by the pair

$$\Big(t_i,\ \text{rank}(S_{i+1})\Big).$$

Note that $\text{rank}(S_{i+1})$ is always defined for $i \in B_0$ by the previous step. Then, radix sort these pairs to obtain the order of S_{B_0}.

Step 3: Merge. Merge the two sorted sets S_C and S_{B_0} using a standard comparison-based merging. To compare a suffix $S_i \in S_C$ against a suffix $S_j \in S_{B_0}$, distinguish two cases:

$$\text{if } i \in B_1: \quad S_i \le S_j \iff \Big(t_i,\ \text{rank}(S_{i+1})\Big) \le_{\text{lex}} \Big(t_j,\ \text{rank}(S_{j+1})\Big),$$
$$\text{if } i \in B_2: \quad S_i \le S_j \iff \Big(t_i,\ t_{i+1},\ \text{rank}(S_{i+2})\Big) \le_{\text{lex}} \Big(t_j,\ t_{j+1},\ \text{rank}(S_{j+2})\Big).$$

The ranks used above are defined in all cases by Step 1. Each comparison inspects $O(1)$ characters/ranks, and the two-way merge advances each pointer at most once, so the merge runs in linear time overall.

Time Complexity. Excluding the recursive call, all steps are linear-time via radix sorting and a single-pass merge. The recursion operates on a string of length $2n/3$, so the overall time satisfies

$$T(n) \;=\; T(2n/3) + O(n) \;=\; O(n).$$

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 [20], 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: Initialize the array of triplets `IntNodes`$[1..t]$.
2: Visit the internal nodes of T in pre-order. For the i-th visited internal node u, set `IntNodes`$[i] \leftarrow \big(\alpha(u), \text{level}(u), \text{parent}(u)\big)$.
3: Let $j \in \{0, 1, 2\}$ be such that the number of nodes in `IntNodes` whose level is $\equiv j \pmod 3$ is at least $t/3$. Sort recursively the upward subpaths starting at nodes in levels $\not\equiv j \pmod 3$.
4: Sort the upward subpaths starting at nodes in levels $\equiv j \pmod 3$ using the result of Step 3.
5: 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.

2.4.5 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 2.4.3) 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 2.5: F and J arrays
Considering the XBWT in Table 2.1, 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`]`)

```
1: F = BuildF(XBWT[T]);
2: J = BuildJ(XBWT[T], F);
3: Create node r and set Q = {⟨1, r⟩};                    ▷ Q is a stack
4: while Q ≠ ∅ do                                  ▷ We still have nodes to create in T
5:     ⟨i, u⟩ = pop(Q);
6:     j = J[i];                                   ▷ Take the block of u's children in S
7:     if j = −1 then                                              ▷ u is a leaf of T
8:         continue;
9:     end if
10:    Find first j′ ≥ j such that S_last[j′] = 1;   ▷ S[j, j′] are the children of u in T
11:    for h = j′ downto j do
12:        Create the node v labeled S_α[h];
13:        Attach v as first child of u;
14:        push(⟨h, v⟩, Q);
15:    end for
16: end while
17: return node r.
```

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

```
1: for i = 1, ..., |Σ_N| do
2:     C[S_α[i]] ← C[S_α[i]] + 1;                ▷ Count the occurrences of node labels
3: end for
4: F[1] = 2;                                      ▷ S_π[1] is the empty string
5: for i ∈ {1, ..., |Σ_N| − 1} do                 ▷ Consider just the internal-node labels
6:     s = 0; j = F[i];
7:     while s ≠ C[i] do                          ▷ Not all blocks of children have been passed
8:         j = j + 1;
9:         if S_last[j] = 1 then                  ▷ One further block of children has passed
10:            s = s + 1;
11:        end if
12:    end while
13:    F[i + 1] = j;
14: end for
15: return F.
```

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

```
1: for i = 1, ..., t do
2:     if S_α[i] ∈ Σ_L then
3:         J[i] = −1;                             ▷ S_α[i] is a leaf label
4:     else
5:         z = J[S_α[i]];
6:         while S_last[z] ≠ 1 do                 ▷ Reach the last child of S_α[i]
7:             z = z + 1;
8:         end while
9:         F[S_α[i]] = z + 1;
10:    end if
11: end for
12: return J.
```

2.4.6 Compressing Labeled Trees

The XBWT[T] exhibits a local homogeneity property on S_α: specifically, the labels $\alpha(u)$ of nodes u whose upward paths $\pi(u)$ share long common prefixes appear in S_α in contiguous (or tightly bounded) clusters. This phenomenon can be formalized via the notion of k-contexts on trees. This property mirrors the strong local homogeneity exhibited by strings under the Burrows-Wheeler Transform [4] 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 ([12], Theorem 4)**.** *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 $H_k(S_\alpha) \leq (\log|\Sigma|) + 1$ (Recall that when a unique alphabet is used to label both internal nodes and leaves, one needs a further bit to distinguish between them. The additional term $+1$ in the bound takes this into account), 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.

2.4.7 Indexing a Compressed Labeled Tree

In order to implement the efficient operations listed in Subsection 2.3.3 using the compressed arrays S_{last} and S_α of XBWT, we need the chosen compressors C_α and C_{last} to 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.

In this section, let $S := (S_{\text{last}}, S_\alpha)$ denote the XBWT[T] obtained after the construction phase. 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 2.6:**
> In Table 2.2, `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 2.7:
In Table 2.2, `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 2.8:
In Table 2.2, `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 $S[\text{First} \dots \text{Last}]$ such that all strings in $S_\pi[\text{First} \dots \text{Last}]$ are prefixed by the reversed path $P^R = c_k \cdots c_2c_1$.

Example 2.9:
In Table 2.2, `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 2.4.3). The rank operation at line 5 is used to get the number r of nodes labeled c up to position i in S_α. Then, the position $F[c]$ 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 2.10:
Let's walk through an example using Table 2.2. 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.

	A	$S_{\mathbf{last}}$	S_α	S_π
1	0	0	A	ϵ
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 2.2: The sequence S for the tree shown in Figure 2.1, 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.

3. Then, we count $z = 1$ ones in S_{last} up to position $y - 1$.
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 2.1.

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 2.4.3) 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 2.11:
Let's illustrate how to find a node's parent using Table 2.2. 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 2.12:
Consider the tree in Figure 2.1, 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 the number of occurrences of subpath P is 2, as evidenced by the two occurrences of 1 in range $S_{\text{last}}[12, 13]$.

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)
```

2.4.8 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 compressed data structures. We will develop two algorithms for constructing the XBWT: one efficient linear-time recursive algorithm and one more straightforward iterative algorithm. Also, we will implement the necessary data structures and algorithms for navigating and querying the XBWT, such as parent-child navigation and path-based searches.

The 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`.

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.
- 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.

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 composing 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` [17]. 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 [31].

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 S_{last}, a dedicated binary array B_α that stores the additional information associated with each entry of S_α (i.e., $B_\alpha[i] = 1$ if the i-th symbol in S_α corresponds to a leaf and 0 otherwise), and the A array 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 [18]. 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.

If RRR-compressed bitvectors are used for the internal bitmaps, a wavelet tree over a sequence $S \in \Sigma^n$ (with $|\Sigma| = \sigma$) occupies $nH_0(S) + o(n \log \sigma) + \Theta(\sigma \log n)$ bits of space, where $H_0(S)$ is the zero-order empirical entropy of S, and it supports access, rank, and select queries in $O(\log \sigma)$ time.

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

2.4.9 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 2.3 and Table Table 2.4. The source code for the experiments can be found in the `experiments.cpp` file.

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 2.3. **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	34.200

Table 2.3: Performance comparison between PathSort (P.S.) and Naive Sort (N.S.) algorithms.

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 repre-

sented 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 2.4 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 2.4: 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.

Conclusions

From the results shown in Table Table 2.3, 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×.

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.

2.5 Finite State Automata

2.5.1 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.

2.5.2 DFA Minimization

The process of automata minimization consists of reducing the number of states in a DFA while preserving its accepted language. 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 [19]. 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.

The algorithm enables computing equivalence classes of nodes, in particular, the Myhill-Nerode equivalence classes [30, 28]. 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 14 (Myhill-Nerode 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.

Theorem 3 (Myhill-Nerode theorem [30, 28])**.** *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.*

Throughout this section let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. For $q \in Q$ and $a \in \Sigma$, we adopt the shorthand $q.a := \delta(q, a)$. We extend δ to words by the usual recursion:

$$\delta^*(q, \varepsilon) = q, \qquad \delta^*(q, wa) = \delta\Big(\delta^*(q, w),\, a\Big) \quad \text{for } w \in \Sigma^*,\ a \in \Sigma.$$

For a word $w = w_1 w_2 \dots w_n \in \Sigma^*$, we then write $q.w := \delta^*(q, w)$ for the (unique) state reached from q by reading w. A word w is accepted by M iff $\delta^*(q_0, w) \in F$.

Also, Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing L. For states (nodes) $u, v \in Q$, we say that u and v are MN-equivalent iff

$$\forall \alpha \in \Sigma^* :\ u.\alpha \in F \iff v.\alpha \in F.$$

2.5.3 Revuz' Minimization Algorithm

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. 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 edges [32].

Let's begin by providing some definitions needed to understand the algorithm.

Definition 15 (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.

Lets now define the canonical label of each state that will be necessary to identify MN-equivalent states. For $s \in Q$, let $l_1 < \cdots < l_k$ be the symbols of the outgoing transitions defined at s (listed in increasing order of Σ). With $b = \mathrm{F}$ if $s \in F$ and $b = \mathrm{NF}$ otherwise, we set

$$\mathrm{l}(s) := \Big(b,\, l_1,\, s.l_1,\, l_2,\, s.l_2,\, \ldots,\, l_k,\, s.l_k\Big).$$

Also, the algorithm uses a function R to map the labels of states to a new signature. This function is defined as follows:

Definition 16 (Signature map R)**.** *Let $N[\,\cdot\,]$ be the current renaming array that assigns to each state its equivalence class identifier Myhill-Nerode. For a state s labeled*

$$\mathrm{l}(s) \;=\; \Big(b,\, l_1,\, s.l_1,\, l_2,\, s.l_2,\, \ldots,\, l_k,\, s.l_k\Big),$$

where $b \in \{F, NF\}$, $l_i \in \Sigma$ (listed in increasing order), and $nl_i \in Q$, define

$$R(\mathrm{l}(s)) \;=\; \Big(b,\, l_1,\, N[s.l_1],\, l_2,\, N[s.l_2],\, \ldots,\, l_k,\, N[s.l_k]\Big).$$

It is importanto to notice that, since the automaton is acyclic, every transition $s \xrightarrow{a} t$ strictly decreases the height: $h(t) < h(s)$. The main loop of Algorithm 8 processes levels in increasing order $i = 0, 1, \ldots$, so by the time we handle a state $s \in \Pi_i$, all its targets t lie in $\bigcup_{j<i} \Pi_j$ and have already been assigned a Myhill-Nerode equivalence class.

Algorithm 8 RevuzMinimization(ADFA)

Require: ADFA $M = (Q, \Sigma, \delta, q_0, F)$
Ensure: Minimal DFA $M' = (\{1, \ldots, n\}, \Sigma, \delta', N[q_0], F')$ with $F' = \{N[q] \mid q \in F\}$ and $\delta'(N[q], a) = N[\delta(q, a)]$

```
1: Calculate height h(s) for every state s.
2: Create partitions Π_i = {s ∈ Q | h(s) = i}.
3: N[1, |Q|] = {1, 2, ..., |Q|};                    ▷ Renaming array
4: n = 0;
5: for i := 0 to h(q_0) do                          ▷ q_0 is the initial state
6:     Sort states in Π_i based on R(l(q)), q ∈ Π_i.
7:     n = n + 1;
8:     N[Π_i[1]] = n;
9:     for j := 2 to |Π_i| do
10:        if then R(l(Π_i[j])) ≠ R(l(Π_i[j − 1]))
11:            n = n + 1;
12:        end if
13:        N[Π_i[j]] = n;
14:    end for
15: end for
```

The algorithm proceeds level by level, from $i = 0$ up to the maximum height, ensuring that states at each level are correctly partitioned into Myhill-Nerode equivalence classes. For each level i, it groups the states in Π_i based on their signatures computed by the function R (see Definition 16). As explained before, when processing level i, the equivalence classes for all states in lower levels ($j < i$) have already been finalized. The signature $R(l(s))$ for a state s depends on its finality and the equivalence classes of its immediate successors. Therefore, two states $s, t \in \Pi_i$ have the same signature if and only if they are MN-equivalent. The algorithm assigns a unique class identifier to each group of states with the same signature.

The whole algorithm can be implemented to run in time $O(m)$ for an acyclic automaton with m edges. Heights may be computed in linear time by a bottom-up traversal. The lists of states of a given height are collected during this traversal. The signature of a state is easy to compute provided the edges starting in a state have been sorted (by a bucket sort for instance to remain within the linear time constraint). Sorting states by their signature again is done by a lexicographic sort [2].

Example 2.13:
Now we are going to see an example of reduction for a given ADFA. The ADFA is represented in figure Figure 2.2 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 ADFA, our focus is on those that are also trees, as this is the specific case relevant to our work.

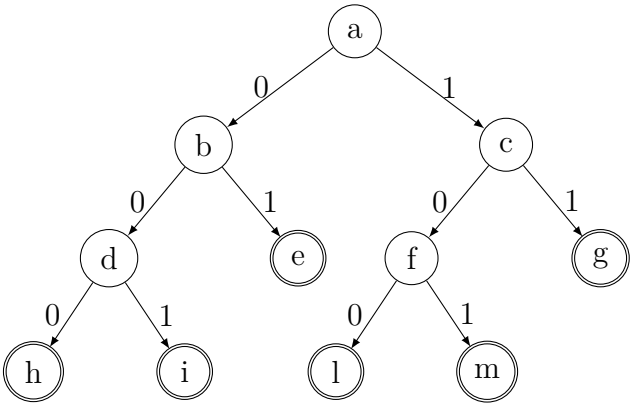


Figure 2.2: Example ADFA to be minimized

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 ADFA represented in Figure 2.3. Each node of the original ADFA is represented by a node in the minimized ADFA (equivalence classes). The edges represent transitions between these nodes. The root node A is the initial state of the minimized ADFA, while the node D is the final state.

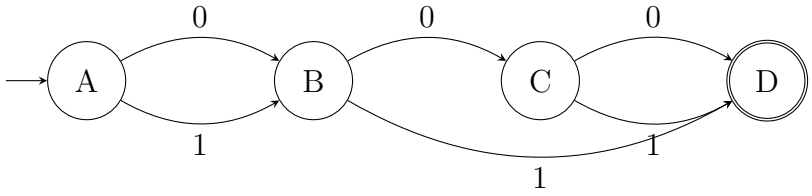


Figure 2.3: Minimized ADFA

The equivalence classes of the nodes are listed in Table 2.5.

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

Table 2.5: Equivalence classes of the nodes

2.6 Wheeler and p-sortable Graphs

2.6.1 Introduction and Motivation

As established in the introduction, the primary goal of this thesis is to find an effective balance between compressing a finite language and preserving its indexability. The two extremes—full DFA minimization and the raw input trie—are inadequate, as one sacrifices indexing for compression and the other sacrifices compression for indexing. The solution to this problem lies in a specific class of graphs that are structured enough to be indexed efficiently yet flexible enough to allow for significant compression. This chapter introduces the theoretical framework that underpins our approach: Wheeler graphs and their generalization, p-sortable graphs.

A crucial observation is that the input trie representing our language is already a highly structured object. It is a Wheeler graph, a type of graph that admits a special ordering on its nodes and edges, making it exceptionally well-suited for indexing. In

formal terms, a trie is a 1-sortable graph. This property explains both its powerful indexing capabilities and its inherent lack of compression.

The concept of p-sortability offers a way to navigate the trade-off. By controllably increasing the sortability parameter p, we can begin to merge MN-equivalent states (see Theorem 3), thereby compressing the graph. The resulting automaton is no longer a simple trie but a more general p-sortable graph that retains strong indexing properties. This chapter will formally define these concepts, which are the foundation of our algorithm for achieving a practical compromise between compression and indexability.

2.6.2 Orders

The core property that makes Wheeler and p-sortable graphs efficiently indexable is the existence of a specific ordering on their states. This ordering provides the necessary structure to navigate the automaton and answer queries quickly, a task that is computationally hard on general graphs. The fundamental ordering used in this context is the *co-lexicographic order* (co-lex), which compares states based on the labels of the paths that reach them. This section formally defines this order and the related concepts that are essential for understanding the structure of indexable automata.

Definition 17 (Co-lexicographic Order on Σ^*)**.** *The co-lex order $\preceq$ is defined as follows. Given two strings $\alpha, \beta \in \Sigma^*$, we say that $\alpha \preceq \beta$ if and only if either:*

- *α is a suffix of β, or*
- *there exist strings $\alpha', \beta', \gamma \in \Sigma^*$ and symbols $a, b \in \Sigma$, such that $\alpha = \alpha' a \gamma$, $\beta = \beta' b \gamma$, and $a \prec b$.*

Now, let us define the formal concept of partial order and the width of a partial order.

Definition 18 (Partial Order)**.** *A partial order is a binary relation $\leq$ over a set S that is reflexive, antisymmetric, and transitive. That is, for all $a, b, c \in S$:*

- *$a \leq a$ (reflexivity)*
- *if $a \leq b$ and $b \leq a$, then $a = b$ (antisymmetry)*
- *if $a \leq b$ and $b \leq c$, then $a \leq c$ (transitivity)*

A partial order $(S, \leq)$ can be visualized using a *Hasse diagram*. In a Hasse diagram, each element of S is represented by a node, and given $a, b \in S$ there is a line segment or curve going upward from a to b if $a \leq b$ and there is no element $c \in S$ such that $a \leq c \leq b$. The direction of the relation is implicitly understood to be upwards, so arrows are not needed. Also, if two elements of S are incomparable under $\leq$ they are displayed at the same level in the diagram. An example is given in Example 2.14.

Example 2.14:
In the example shown in Figure 2.4, the set is composed of the divisors of 12, and the relation is divisibility. An edge is drawn from a to b if a divides b and there is

no other element c in the set such that a/c and c/b. For instance, there is an edge from 2 to 4 because 2 divides 4, and no other element in the set is a multiple of 2 and a divisor of 4. There is no direct edge from 2 to 12 because the relationship is captured transitively through other elements, such as 2/4/12 or 2/6/12.

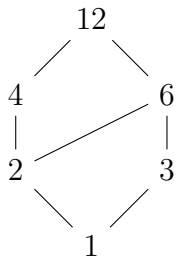


Figure 2.4: Hasse diagram for the set $\{1, 2, 3, 4, 6, 12\}$ with the "divides" relation.

Now we can define the concept of width of a partial order.

Definition 19 (Antichain)**.** *An antichain of a partial order $(S, \leq)$ is a subset of S where any two distinct elements are incomparable. That is, for any two distinct elements a, b in the antichain, neither $a \leq b$ nor $b \leq a$ holds.*

Definition 20 (Width of a partial order, [8])**.** *The width of a partial order $\leq$, denoted by width$(\leq)$, is the size of the largest possible antichain.*

By Dilworth's Theorem [8], the width of a partially ordered set $(S, \leq)$ is equal to the cardinality of its largest antichain; this can be equivalently defined as the minimum number of chains needed to partition S, where each chain is a totally ordered subset of S under the relation $\leq$.

2.6.3 Wheeler Graphs

With the concept of co-lex order established, we can now define the class of graphs that form the starting point of our work. A Wheeler automaton is an automaton where the states can be arranged in a strict, total order.

Definition 21 (Wheeler automaton, [14])**.** *A finite state automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ is a Wheeler automaton if there exists a total order $\leq$ on its set of states Q that satisfies the following axioms:*

1. *The initial state precede all other states in the order.*

For any two transitions $u \in \delta(u', a)$ and $v \in \delta(v', b)$:

2. $a < b \implies u \leq v$,
3. $a = b \wedge u' < v' \implies u \leq v$.

The order $\leq$ is called a Wheeler order.

Consequently, we define the concept of Wheeler language.

Definition 22 (Wheeler language)**.** *A Wheeler language L is a language accepted by a deterministic Wheeler automaton.*

The most important example for Wheeler automaton in this thesis is the trie. Any trie representing a finite language is a Wheeler automaton. The co-lexicographic order of the strings spelling the paths from the root to each node provides the required total ordering of the states. This is why tries are inherently indexable. However, this rigid structure also means they are uncompressed, as every unique path must be stored explicitly, even if it corresponds to a substring that appears many times in the language. Our work begins with this observation: we start with a Wheeler automaton (the trie) and seek to compress it while preserving efficient indexability.

2.6.4 The Co-lex Width of an Automaton

Now that we have the concepts of co-lex order and width, we can combine them to formally define the class of indexable automata that are central to this thesis. The width of the co-lex partial order on an automaton's states is the critical measure of its structural complexity from an indexing perspective.

The co-lex order can be extended to the set of states of an automaton. The idea of co-lex order on the states of an automaton was first introduced with the notion of Wheeler graphs by Gagie et al. [14] and was later generalized to arbitrary finite automata by Cotumaccio and Prezza [6], where a partial order replaces the total order.

Let $\lambda(q)$ denotes the set of labels of transitions entering in state q, and $\min \lambda(q)$ and $\max \lambda(q)$ represent the minimum and maximum element of the set, respectively. The definition of co-lex order on an automaton is as follows:

Definition 23 ([7])**.** *Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. A co-lex order on N is a partial order $\leq$ on Q that satisfies the following two axioms:*

1. *For every $u, v \in Q$, if $u < v$, then $\max \lambda(u) < \min \lambda(v)$*
2. *For every $a \in \Sigma$ and $u, v, u', v' \in Q$, if $u \in \delta(u', a)$, $v \in \delta(v', a)$ and $u < v$, then $u' \leq v'$*

The two axioms in Definition 23 allow for states of a finite automaton to be compared. When $\leq$ is total, we say that the co-lex order is a Wheeler order (introduced in [14] and Definition 21).

Consequently, we can introduce the concept of co-lex width of an automaton.

Definition 24 ([7])**.** *The co-lex width of an NFA N is the minimum width of a co-lex order on N, i.e.,*

$$width(N) = \min\{width(\leq) | \leq \text{ is a co-lex order on } N\}$$

The requirement of a Wheeler order is powerful but restrictive. Many automata, especially those resulting from DAG compression, may not satisfy it (see Example 2.15). This introduces a fundamental trade-off: while DAG compression minimizes an automaton's size, it can destroy the very structure that enables efficient indexing. In fact, it has been shown that indexing general graphs—and thus, highly compressed automata—to support fast string matching is computationally expensive, as showed in [11]. The second axiom of Definition 23 does not always enforce

an ordering between any two states, leading to a partial order instead of a total one. This gives rise to the more general notion of a *p-sortable automaton*:

Definition 25 (*p*-sortable automaton)**.** *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a finite-state automaton. We call $\mathcal{A}$ p-*sortable *if there exists a co-lexicographic order $\leq$ on Q such that Q can be partitioned into p chains $\{Q_i\}_{i=1}^{p}$, where each $(Q_i, \leq)$ is totally ordered.*

Under these definitions, a Wheeler automaton is a **1-sortable automaton**, as a total order has a width of 1 (the largest antichain is a single element).

Example 2.15:
State incomparability can arise in several situations. For example, consider two states u and u'.

- As illustrated in Figure 2.5-(a), if there are two same-labeled transitions to the target states u and v from two incomparable states u' and v', then u and v are also incomparable.
- Conflicting constraints from different labels can force incomparability, as shown in Figure 2.5-(b). An existing order on the sources of a-transitions (e.g., $u' < v'$) may require $u < v$ to satisfy the Wheeler axioms, while an order on the sources of b-transitions (e.g., $v'' < u''$) may require the opposite, $v < u$. Since both cannot be true, the targets u and v must be incomparable.

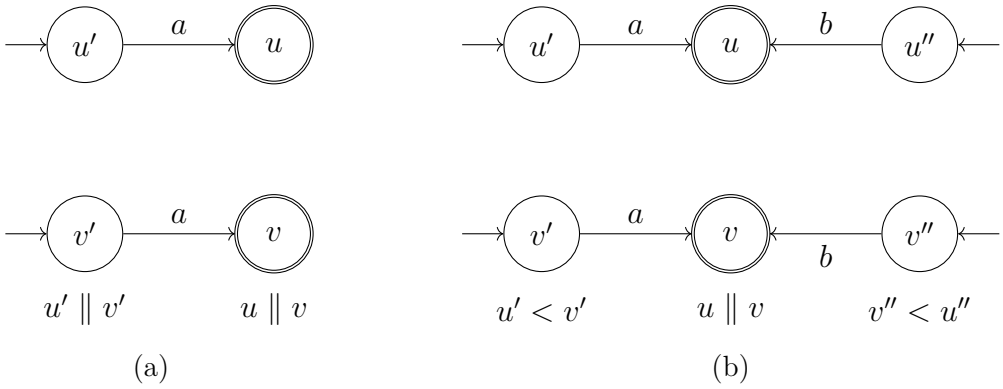


Figure 2.5: Examples of state incomparability in automata.

Example 2.16:
Consider the following DFA D of Figure 2.6-(a) and its partial co-lex order Figure 2.6-(b). The DFA is not Wheeler because states v_3 and v_5 and states v_4 and v_7 are incomparable (as shown in the Hasse diagram). D admits a partition into two chains of totally ordered states, for example one possible chain partition is given by:

- Chain 1: $v_1 \leq v_2 \leq v_3 \leq v_4$
- Chain 2: $v_5 \leq v_6 \leq v_7$

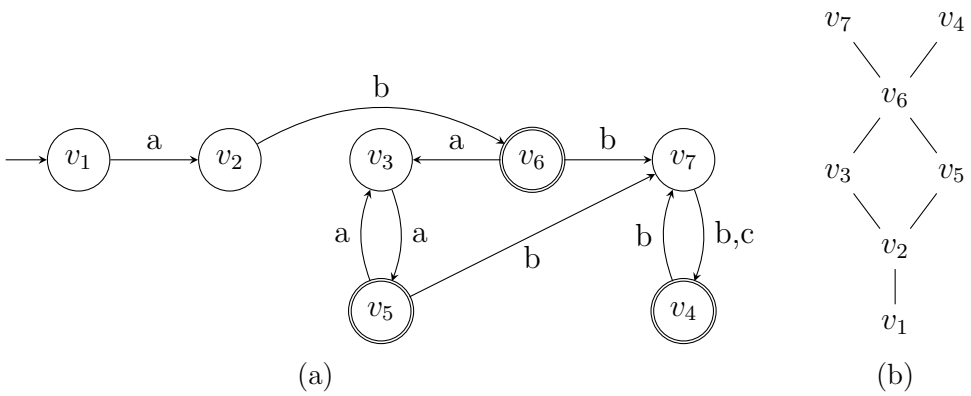


Figure 2.6: An example of a 2-sortable DFA and its corresponding Hasse diagram of the partial order.

We now introduce another important result by [27]. Let $D = (Q, \Sigma, \delta, s, F)$ denote the minimal DFA accepting a Wheeler language L, and let $D^w = (Q^w, \Sigma, \delta^w, s^w, F^w)$ denote the minimal Wheeler DFA (WDFA) accepting L, i.e. the DFA with the minimum number of states among all Wheeler DFAs accepting L. Since D^w is Wheeler for any two distinct states $q, q' \in Q^w$, the associated intervals $I_q, I_{q'}$ are disjoint. This property does not generally hold for D, where states may correspond to overlapping sets of prefixes. As a consequence, when transforming D into D^w, certain states of D may need to be *split* into several states in D^w, potentially leading to an exponential blow-up in the number of states.

Example 2.17: [27]

We now provide a simple example of an automaton D with width n that accepts a Wheeler language, yet its minimum equivalent Wheeler DFA, D^w, is exponentially larger. Let D be the automaton depicted in Figure 2.7.

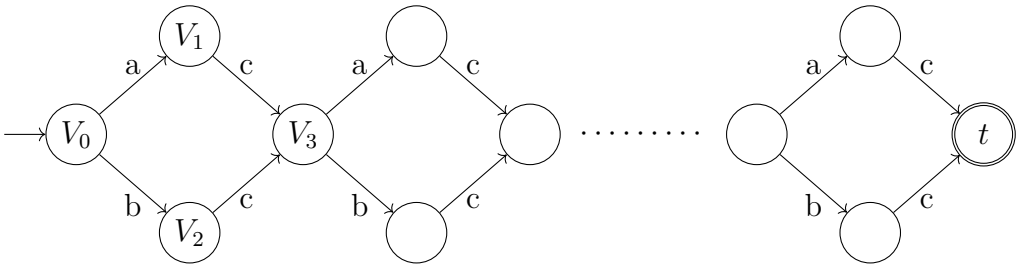


Figure 2.7: A DFA accepting a finite (and thus Wheeler) language, for which the minimal equivalent Wheeler DFA is exponentially larger.

The language $L = \mathcal{L}(D)$, being finite, is a Wheeler language [1]. However, any Wheeler automaton accepting L must have a number of states that is exponential in n. Since there are exponentially many such pairwise distinct strings leading to t, a Wheeler automaton must partition the set I_t into an exponential number of sub-intervals. This forces the state t to be "split" into exponentially many copies, leading to an exponential blow-up in the size of the minimal Wheeler DFA.

The previous example highlights a crucial trade-off: enforcing the strict ordering of a Wheeler DFA can lead to an exponential increase in the number of states compared to a minimal DFA.

Since this work aims to transform a trie (a 1-sortable automaton) into a more general p-sortable graph (with width $p > 1$) in a way that introduces DAG compression while maintaining efficient indexability, it is motivated by the powerful result of **??** leading to the fact that even a small increase in sortability (for example from $p = 1$ to $p = 2$) can yield exponential compression. This highlights the potential of exploring the trade-off between sortability and size, which is the central theme of this thesis.

2.6.5 Indexing Finite State Automata

Now we introduce the current state of the art in indexing finite state automata. In 2023, Cotumaccio et al. [6] introduced a compressed data structure for automata whose performance and space complexity are directly tied to the automaton's co-lex width p. This structure generalize the famous Burros-Wheeler transform [4].

Theorem 4 (Cotumaccio et al. [6])**.** *Let A be a p-sortable automaton. There exists a compressed data structure for A that supports subpath queries (Definition 12) on a query word α of length m in $O(mp^2 \log(p|\Sigma|))$ time. The space required is:*

- $\log(|\Sigma|) + \log p + 2$ *bits per edge for DFAs*
- $\log(|\Sigma|) + 2\log p + 2$ *bits per edge for NFAs*

This highlights a direct trade-off: both query time and space per edge depend on the width parameter p, which governs the automaton's compressibility.

To highlight the importance of this data structure, we recall that the final output of our compression pipeline is a p-sortable DAG compressed automaton with a controlled co-lex width p. This allows us to leverage these advanced indexing capabilities on the compressed automata produced by our method.

2.7 Min-Weight Perfect Bipartite Matching

2.7.1 Introduction and Motivation

The fundamental goal of our compression scheme is to transform an input trie into a compressed, p-sortable automaton by partitioning its nodes into p chains in an optimal way.

To make this optimization problem more concrete, we can frame it as a string partitioning problem. Imagine the sequence of nodes in the trie, when read in co-lexicographic order, as a single long string. The "character" corresponding to each node is its Myhill-Nerode equivalence class, which determines if it can be merged with other nodes. The task is to partition this string of nodes into p substrings such that the number of runs is minimized (consecutive nodes of the same MN-equivalence class are merged into a single run). For instance, a subsequence $AAABBA$ contains three runs. Minimizing the number of runs directly corresponds to maximizing the number of merged states, yielding a compact p-sortable automaton.

In this sections, we will provide the necessary background on the Minimum Weight Perfect Bipartite Matching (MWPBM) problem, a fundamental challenge in combinatorial optimization. We will then demonstrate in Section 2.4 a formal reduction from our partitioning problem to MWPBM. This reduction is the key to our method,

as it allows us to model our problem as a bipartite graph and leverage well-known, efficient algorithms to find the optimal solution for our compression task.

2.7.2 Bipartite Graphs

Definition 26. *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 2.8.

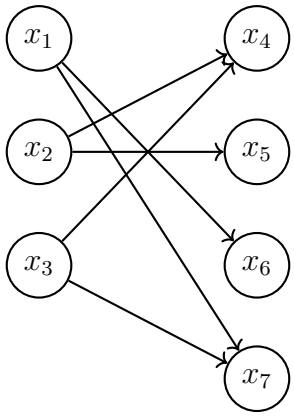


Figure 2.8: Example of a bipartite graph $G = (V, E)$ where $V_1 = \{x_1, x_2, x_3\}$, $V_2 = \{x_4, x_5, x_6, x_7\}$ and $E = \{(x_1, x_6), (x_1, x_7), (x_2, x_4), (x_2, x_5), (x_3, x_4), (x_3, x_7)\}$.

Definition 27 (Weighted Bipartite Graph)**.** *A weighted bipartite graph is a bipartite graph $G = (V, E, w)$, where w is a weight function $w : E \to \mathbb{R}$ that assigns a real-valued weight to each edge.*

In a weighted bipartite graph, each edge has a numerical value, or "weight", associated with it.

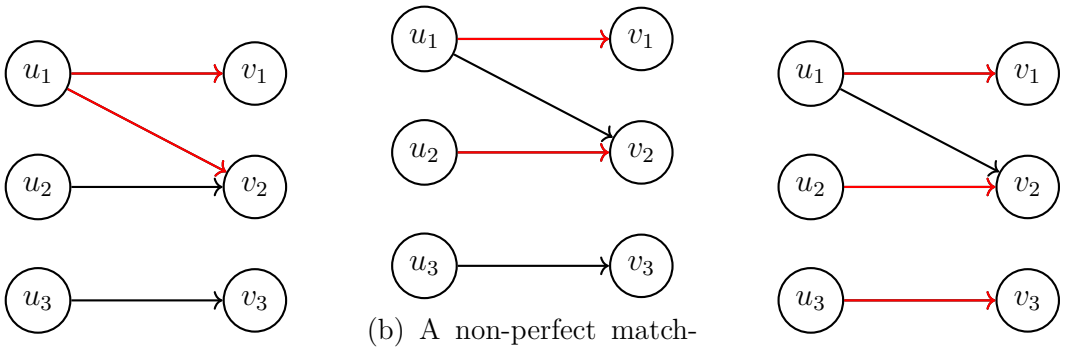
2.7.3 Problem Definition

Given a bipartite graph $G = (V, E)$ (Definition 26), let's define the concept of a matching.

Definition 28 (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 $|V_1| = |V_2|$ [16].

Example 2.18:
In Figure 2.9, 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 2.9: 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 weighted bipartite graph (Definition 27) is a well-known problem in combinatorial optimization. The problem can be formulated as follows:

Definition 29 (Minimum weight perfect matching in weighted bipartite graphs)**.** *Given a weighted bipartite graph $G = (V, E, w)$, find a perfect matching M such that the sum of the weights of the edges in M is minimized.*

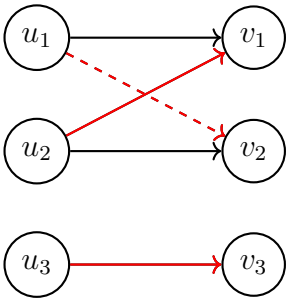
The weight of a matching is the sum of the weights of the edges in the matching. We define the weight of a matching M as follows:

$$w(M) = \sum_{e \in M} w(e) \tag{2.1}$$

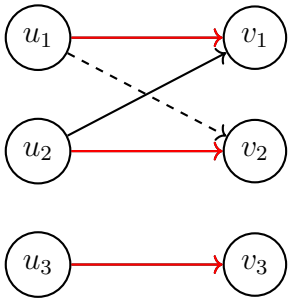
Example 2.19:
Consider the weighted bipartite graph in Figure 2.10. The goal is to find a perfect matching with the minimum possible total weight. Both subfigures show a valid perfect matching; however, only one of them has the minimum weight.

- 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 2.10: 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.

2.7.4 State of the Art

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 [23]. The algorithm is based on the Hungarian method, which is a combinatorial optimization algorithm that solves the MWPBM problem in polynomial time. In the original paper the complexity of the algorithm was $O(n^4)$ where n is the number of nodes in the bipartite graph. Later Dinic and Kronrod [9] 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 [10] proposed an algorithm that solves the problem in $O(nm + n^2 \log n)$ time, where m is the number of edges. In 1989 Gabow and Tarjan [13] proposed an algorithm that solves the problem in $O(\sqrt{n}m \log(nW))$ time, where W denote the highest edge weight in the graph; costs are assumed to be integral. The algorithms work by scaling. Lastly, in 2009, Sankowski and Piotr [33] introduced a randomized algorithm that solves the problem in $O(Wn^\omega)$ time, where ω is the exponent of matrix multiplication, and W is the highest edge weight in the graph.

In 2022, Chen, Li, et al. [5] proposed a nearly linear time algorithm for the Minimum Cost Flow (MCF) problem, running in $O(m'^{1+o(1)})$ on a network with m' edges. This is highly relevant as the MWPBM problem can be reduced to MCF. Specifically, an MWPBM instance on a bipartite graph with n vertices and m edges can be transformed into an MCF problem on a network with $m' = 2n + m$ edges.

Chapter 3

Tree Compression Scheme

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 trie. The proposed algorithm is designed to identify and compactly represent these recurring patterns, thereby improving compression performance, particularly for highly repetitive tries. This chapter provides an overview of the proposed compression scheme.

3.1 Compression Scheme Pipeline

The overall pipeline of our proposed method is outlined in Algorithm 9. It takes an input trie T and a width parameter p and produces a compressed, p-sortable automaton.

Algorithm 9 $CompressTrie(T, p)$

Require: Input trie T, width integer parameter p
Ensure: A compressed, p-sortable automaton $\mathcal{A}$
1: $V_{sorted} \leftarrow \text{PathSort}(T)$
2: $N[1, \ldots, |Q|] \leftarrow \text{ComputeEquivalenceClasses}(T)$
3: $G_{bipartite} \leftarrow \text{ConstructMWPBMInstance}(V_{sorted}, N, p)$
4: $M \leftarrow \text{SolveMWPBM}(G_{bipartite})$
5: $C[1, \ldots, p] \leftarrow \text{ExtractChainsFromMatching}(M, V_{sorted})$
6: $\mathcal{A} \leftarrow \text{CollapseChains}(C, N)$
7: **return** $\mathcal{A}$

The first step of the pipeline (line 2) is to establish a total order on the nodes of the trie. This is achieved by sorting the nodes co-lexicographically using the **Path Sort** algorithm, which we detail in Subsection 2.4.4. This sorting is fundamental, as it arranges the nodes in the order required for a Wheeler automaton.

Next, the algorithm identifies which nodes are candidates for merging (line 3). This is done by partitioning the nodes into equivalence classes based on the structure of the subtrees rooted at each node. Two nodes are in the same class if and only if their subtrees are isomorphic. This is equivalent to computing the Myhill-Nerode equivalence classes for the finite language accepted by the trie, a process we adapt from Revuz's algorithm for minimizing acyclic DFAs (Subsection 2.5.3).

The core of the algorithm lies in lines 5-7, where we solve the **String Partitioning Problem**. The input to this problem is the string formed by concatenating the Myhill-Nerode classes of the sorted nodes. As detailed in Section 3.3, we reduce this

problem to finding a Minimum Weight Perfect Bipartite Matching (MWPBM). A bipartite graph is constructed where edge weights correspond to the cost of placing two characters adjacently in a subsequence. By finding a perfect matching with minimum weight using standard algorithms (Subsection 2.7.4), we can reconstruct a partition of the string into p subsequences that minimizes the total number of runs (Definition 1), thereby maximizing compression.

Finally, the DAG compressed automaton is constructed (line 8). The algorithm iterates through each of the p subsequences and "collapses" any consecutive sequence of nodes belonging to the same Myhill-Nerode equivalence class into a single state. This compression, which we describe in Section 3.2, produces the final p-sortable automaton, which can then be indexed for efficient querying (Subsection 2.6.5).

Example 3.1:
In our running example, we begin with the tree shown in Figure 3.1. Each node in this tree is labeled with its corresponding Myhill-Nerode equivalence class, as determined in Example 2.13. By traversing the nodes in co-lexicographic order, we construct a string S where each character represents the Myhill-Nerode equivalence class of a node:

$$S = \text{ABCDDCBDDDD}$$

This string S then becomes the input to the String Partitioning Problem (see **??**), where the objective is to partition it into p subsequences while minimizing the total number of runs.

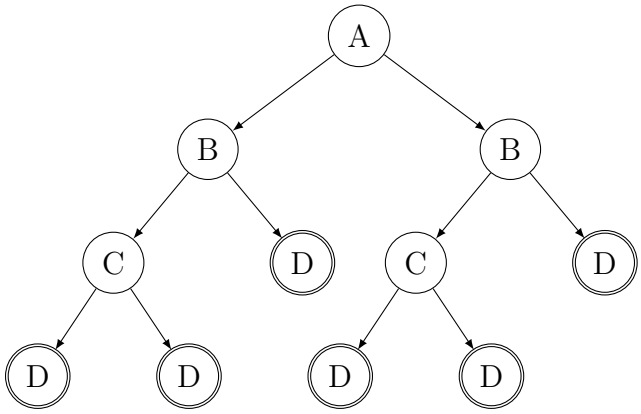


Figure 3.1: Tree ADFA of Figure 2.2. Each node is labeled with its equivalence class.

3.2 Collapsing Nodes in Chains

This chapter introduces a crucial optimization technique called **node collapsing**, designed to reduce the size of the trie. The String Partitioning Problem provides us with a set of subsequences whose characters correspond to the nodes of the original tree. Within these subsequences, we often find consecutive nodes that are redundant from a language-theoretic perspective, specifically because they belong to the same Myhill-Nerode equivalence class.

The core idea is to merge any such sequence of consecutive, equivalent nodes into a single representative node. This operation simplifies the graph structure while

preserving its essential connectivity. The new representative node inherits all unique incoming and outgoing transitions from the nodes it replaces, ensuring that the overall language accepted by the graph remains unchanged.

In the following sections, we will formally define the collapsing procedure and then rigorously prove that this transformation is language-preserving (**??**), guaranteeing the correctness of our compression scheme.

3.2.1 How to Collapse Nodes

Now, we define the concept of collapsing two consecutive MN-equivalent nodes in a subsequence.

Definition 30 (Collapsing Two MN-equivalent States)**.** *Let $A = (Q, \Sigma, \delta, q_0, F)$ be the ADFA corresponding to the trie structure we are compressing. The operation* ***collapse(u, v)*** *is defined for two states $u, v \in Q$ if and only if they are consecutive in a subsequence and are MN-equivalent.*

This operation transforms A into a new automaton $A' = (Q', \Sigma, \delta', q_0', F')$ as follows:

1. ***State Merging:*** *A new state w is created to replace u and v. The new set of states is $Q' = (Q \setminus \{u, v\}) \cup \{w\}$.*
2. ***Transition Redirection:*** *Let $\phi : Q \to Q'$ be a state mapping function defined as follows:*

$$\phi(z) = \begin{cases} w & \text{if } z \in \{u, v\} \\ z & \text{otherwise} \end{cases}$$

   We treat δ as a set of triples of the form (q, a, r), where q is the source state, a is the transition label, and r is the destination state. The new transition function $\delta' : Q' \times \Sigma \to Q'$ is then formed by applying this mapping to the source and destination states of every transition in δ:

$$\delta' = \{(\phi(q), a, \phi(r)) \mid (q, a, r) \in \delta\}$$

3. ***Initial and Final States:*** *The new initial state is $q_0' = \phi(q_0)$, and the new set of final states is $F' = \{\phi(f) \mid f \in F\}$.*

Example 3.2:
Consider Example 3.7 where we obtained the chains $C_1 = \{A, C, C, B\}$ and $C_2 = \{B, D, D, D, D, D, D\}$ for the tree ADFA in Figure 2.2 by applying the reduction from String Partitioning to MWPBM. The nodes inside each chain are the following:

- $C_1 = \{a, d, f, c\}$
- $C_2 = \{b, h, l, e, i, m, g\}$

Applying the collapsing operation from Definition 30:

- For $C_1 = (a, d, f, c)$ with classes (A, C, C, B):
  - Block $B_1 = \{a\}$ (class A) $\to$ collapsed node v_1. The node a is the initial

state. It has two outgoing edges: $a \xrightarrow{0} b$ and $a \xrightarrow{1} c$. Since b and c collapse to w_1 and v_3 respectively, we obtain $v_1 \xrightarrow{0} w_1$ and $v_1 \xrightarrow{1} v_3$.

- Block $B_2 = \{d, f\}$ (class C) $\rightarrow$ collapsed node v_2. The outgoing edges of d and f are:

$$d \xrightarrow{0} h,\ d \xrightarrow{1} i, \qquad f \xrightarrow{0} l,\ f \xrightarrow{1} m.$$

After collapsing, we obtain:

$$v_2 \xrightarrow{0} w_2, v_2 \xrightarrow{1} w_2, \qquad v_2 \xrightarrow{0} w_2, v_2 \xrightarrow{1} w_2.$$

Since we have two identical edges we can keep only one of each.

- Block $B_3 = \{c\}$ (class B) $\rightarrow$ collapsed node v_3. It has two outgoing edges: $c \xrightarrow{0} f$ and $c \xrightarrow{1} g$. Since f and g collapse to v_2 and w_2 respectively, we obtain $v_3 \xrightarrow{0} v_2$ and $v_3 \xrightarrow{1} w_2$.

Result: $C_1' = (v_1, v_2, v_3)$ with classes (A, C, B). Here, v_1 is the initial state.

- For $C_2 = (b, h, l, e, i, m, g)$ with classes (B, D, D, D, D, D, D):
  - Block $B_1 = \{b\}$ (class B) $\rightarrow$ collapsed node w_1. It has two outgoing edges: $b \xrightarrow{0} d$ and $b \xrightarrow{1} e$. As d, e collapse to v_2 and w_2 respectively, we obtain $w_1 \xrightarrow{0} v_2$ and $w_1 \xrightarrow{1} w_2$.
  - Block $B_2 = \{h, l, e, i, m, g\}$ (all class D) $\rightarrow$ collapsed node w_2. The node w_2 collects all incoming edges formerly targeting any of b, h, l, e, i, m, g, and it is accepting.

  Result: $C_2' = (w_1, w_2)$ with classes (B, D), and w_2 is the unique accepting state for this example.

The collapsed chains preserve all distinct outgoing and incoming edges through the collapse map Φ, significantly reducing the space complexity from 11 nodes to 5 nodes total. The resulting 2-sortable automaton is shown in Figure 3.2.

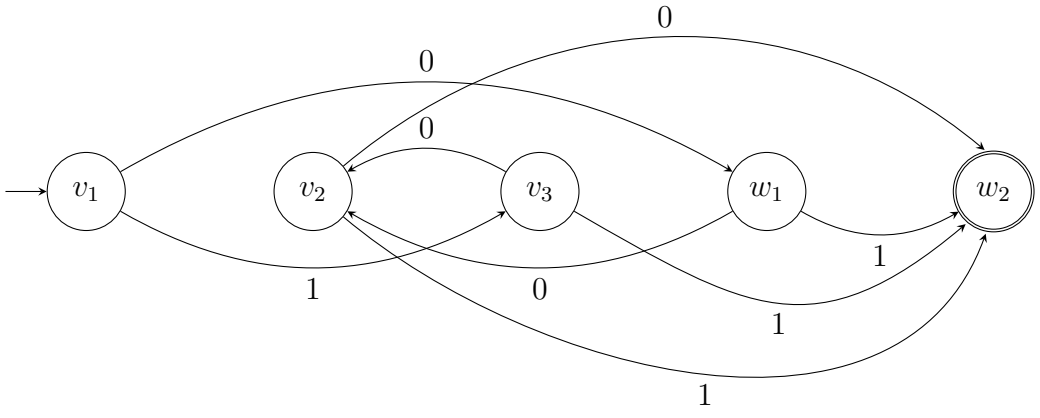


Figure 3.2: 2-sortable automaton obtained after collapsing equivalent nodes in chains C_1 and C_2.

3.2.2 Language Equivalence

Now, we need to prove that the language recognized by the p-sortable automaton obtained after collapsing two MN-equivalent nodes following Definition 30 is equivalent to the language of the original ADFA.

Lemma 2. *Let $M = (Q, \Sigma, \delta, q_0, F)$ be an automaton recognizing $L \subseteq \Sigma^*$. If two states $p, q \in Q$ correspond to the same Myhill–Nerode class for L (i.e., for all $w \in \Sigma^*$ we have $\delta(p, w) \in F \iff \delta(q, w) \in F$), then merging p and q into a single state yields an automaton (possibly nondeterministic) that still recognizes exactly L.*

Proof. By the Myhill-Nerode theorem, every state of M corresponds to a unique equivalence class of $\sim_L$, and L is exactly the union of those classes that intersect L. If p and q belong to the same equivalence class, then for every continuation $z \in \Sigma^*$ we have

$$\delta(p, z) \in F \iff \delta(q, z) \in F.$$

Thus replacing p with q (or vice versa) in any path does not affect whether the run ends in an accepting state. Therefore merging p and q does not alter the set of accepted strings, i.e. the recognized language remains L. □

Collapsing two MN-equivalent nodes as in Definition 30 preserves the language of the original ADFA and the resulting chains inherit a total order. This enables the application of the NFA indexing scheme of Cotumaccio et al. [7].

3.3 Reducing the String Partitioning Problem to the MWPBM Problem

In the previous chapters, we modeled the task of partitioning trie nodes into p chains as the String Partitioning Problem (see Definition 4). We now demonstrate that this problem can be solved in polynomial time by reducing it to the MWPBM problem. This section will first detail the construction of a bipartite graph from an instance of the String Partitioning Problem. Then, we will prove that a minimum-weight perfect matching in this graph directly corresponds to an optimal solution for the partitioning problem. Let's start with an example.

Example 3.3: String Partitioning Problem

Let $S = \text{AABACABB}$ be the input string and let $p = 2$ be the desired number of subsequences. Our goal is to partition the characters of S into two subsequences, S_1 and S_2, such that the total number of runs (Definition 1) is minimized.

Consider the following partition:

- S_1 is formed by taking the 1st, 2nd, 4th, and 6th characters of S: AAAA.
- S_2 is formed by the remaining characters (3rd, 5th, 7th, and 8th): BCBB.

The number of runs for each subsequence is:

- $\text{Runs}(S_1) = 1$ (the run is "AAAA").
- $\text{Runs}(S_2) = 3$ (the runs are "B", "C", "BB").

The total number of runs for this partition is $1 + 3 = 4$. An optimal solution to the String Partitioning Problem would be a partition that achieves the minimum possible total number of runs. In this case, 4 is indeed the optimal value.

3.3.1 Bipartite Graph Construction

Now, we will show how to construct a bipartite graph that allows us to solve the String Partitioning problem.

Definition 31 (Bipartite graph construction)**.** *Let S be a string of size n from an alphabet Σ and let p be the number of subsequences we want to partition S into. We can construct a weighted bipartite graph $G = (V, E, w)$ such that vertices are divided in two disjoint sets $V = V_1 \cup V_2$ in the following way:*

- *V_1 contains $n + p$ nodes composed by p source nodes $s_1, s_2, \ldots, s_p$ (referred to collectively as $\mathcal{S}$) followed by the n characters $a_1, a_2, \ldots, a_n$ (referred to collectively as $\mathcal{T}_1$) of S.*
- *V_2 contains $n + p$ nodes composed by the n characters $b_1, b_2, \ldots, b_n$ (referred to collectively as $\mathcal{T}_2$) of S followed by p destination nodes $d_1, d_2, \ldots, d_p$ (referred to collectively as $\mathcal{D}$).*

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

- ***Source Edges:** For each source node $s \in \mathcal{S}$ and each node $v_j \in \mathcal{T}_2$, an edge (s, v_j) is created with weight $w(s, v_j) = 1$.*
- ***Internal nodes Edges:** For each pair of indices i, j such that $1 \leq i < j \leq n$, an edge is created between $u_i \in \mathcal{T}_1$ and $v_j \in \mathcal{T}_2$. The weight of this edge, $w(u_i, v_j)$, is 0 if the characters S_i and S_j are the same, and 1 otherwise. Formally:*

$$w(u_i, v_j) = \begin{cases} 0 & \text{if } S_i = S_j \\ 1 & \text{if } S_i \neq S_j \end{cases}$$

- ***Destination Edges:** For each node $u_i \in \mathcal{T}_1$ and each destination node $d \in \mathcal{D}$, an edge (u_i, d) is created with weight $w(u_i, d) = 1$.*

Example 3.4: Vertices
Let's apply the reduction to the string $S =$ ABCDDCBDDDD from our running example (Example 3.1), with a target of $p = 2$ subsequences. Following the construction rules, we build a bipartite graph. The vertices of this graph are structured as shown in Figure 3.3.

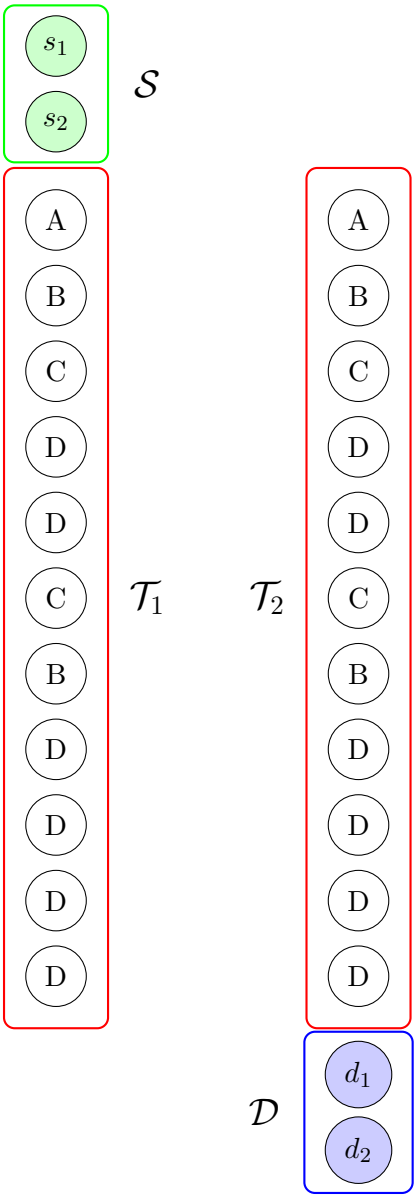


Figure 3.3: Bipartite graph vertices structure for string S = ABCDDCBDDDD with $p = 2$. The nodes are ordered from top to bottom.

Example 3.5: Edges

Let us see a small example for each case of Definition 31. Consider $p = 2$. In Figure 3.4-(a), there is an example for the sources' edges. As stated before, each source is connected with weight 1 to all nodes in $\mathcal{T}_2$.

In Figure 3.4-(b), we illustrate the edges from $\mathcal{T}_1$ to $\mathcal{T}_2$. These edges model the cost of appending a character to a subsequence. An edge from u_i to v_j (for $j > i$) has weight 1 if $S[i] \neq S[j]$ (starting a new run) and weight 0 if $S[i] = S[j]$ (extending an existing run). For instance, the node for the first 'A' connects to the nodes for 'B' and 'C' with weight 1, and to the node for 'A' with weight 0.

Lastly, Figure 3.4-(c) shows the destination edges. These edges terminate a subsequence. An edge from any node $u_i \in \mathcal{T}_1$ to any destination node $d_k \in \mathcal{D}$ has weights 0, ensuring that ending a chain does not increase the run count. For example, if the node for 'B' is the last element of a subsequence, it is matched with

a destination node, and this edge (u_B, d_k) contributes 0 to the total weight.

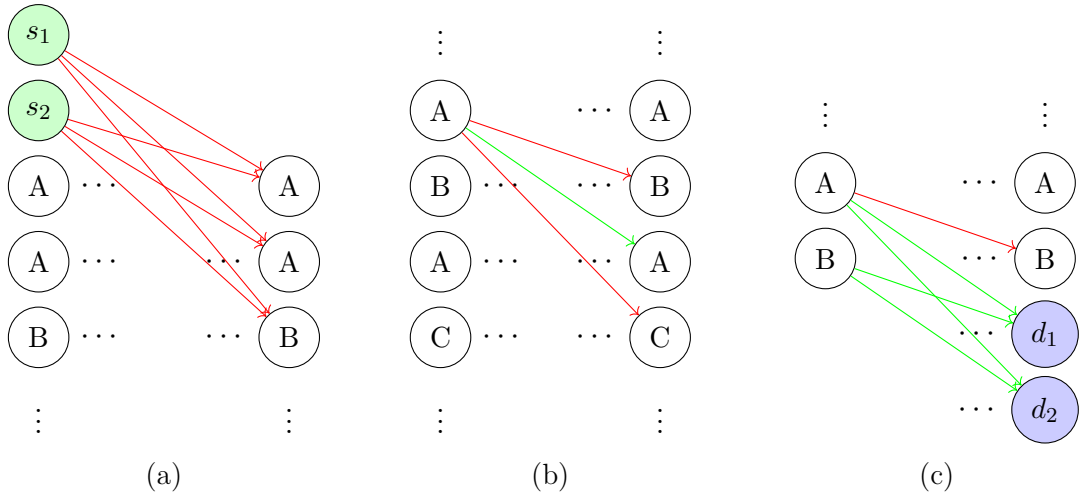


Figure 3.4: Examples of the connection construction in the bipartite graph for $p = 2$, showing the cases for source nodes $\mathcal{S}$ (a), internal nodes $\mathcal{T}_1$ and $\mathcal{T}_2$ (b), and destination nodes $\mathcal{D}$ (c). Red arrows indicate edges with weight 1, while green arrows indicate edges with weight 0.

Let us now state the following theorem regarding the number of edges in the bipartite graph resulting from Definition 31. This theorem is essential for understanding the complexity of the final algorithm employed to solve the MWPBM problem and, thus, the String Partitioning problem.

Theorem 5 (Bipartite graph properties)**.** *The bipartite graph G constructed as stated in Definition 31 has $2n + 2p$ nodes and $2np + \frac{n(n-1)}{2}$ edges.*

Proof. The total number of edges in the graph G is the sum of the edges from the three categories defined in the construction:

- **Source Edges:** There are p source nodes in $\mathcal{S}$ and n tree nodes in $\mathcal{T}_2$. Each source node connects to every node in $\mathcal{T}_2$, resulting in $p \times n = np$ edges.
- **Destination Edges:** There are n tree nodes in $\mathcal{T}_1$ and p destination nodes in $\mathcal{D}$. Each node in $\mathcal{T}_1$ connects to every node in $\mathcal{D}$, resulting in $n \times p = np$ edges.
- **Internal nodes Edges:** For each node $u_i \in \mathcal{T}_1$, edges are created to all nodes $v_j \in \mathcal{T}_2$ where $j > i$. The number of such edges is $\sum_{i=1}^{n-1}(i) = \frac{n(n-1)}{2}$.

Summing these up, the total number of edges is $2np + \frac{n(n-1)}{2}$. □

3.3.2 Proof

In this section, we will prove that a minimum-weight perfect matching in a bipartite graph constructed as stated in Definition 31 directly corresponds to an optimal solution for the String Partitioning problem defined as follows.

Definition 32 (String Partitioning Problem)**.** *Let $\mathcal{I} = (S, p)$ be an instance of the String Partitioning Problem where S is a string of length n over an alphabet Σ, and*

p is a positive integer. The output of the problem is a partition $\mathcal{P} = I_1, \ldots, I_p$ of $[n]$ such that $\delta(\mathcal{P}) = \sum_{i=1}^{p} \tau(S[I_i])$ is minimized.

We introduce the following notation:

Definition 33. *Let $r : \mathcal{I}_{StringPartitioning} \to \mathcal{M}_{MWPBM}$ be the reduction function that maps an instance $\mathcal{I} = (S, p)$ of the String Partitioning Problem to an instance $\mathcal{M} = (G)$ of the MWPBM Problem, where G is the bipartite graph constructed as stated in Definition 31.*

Lemma 3. *Let $\mathcal{I} = (S, p)$ be an instance of the String Partitioning Problem. Any partition $\mathcal{P} = I_1, \ldots, I_p$ of $[n]$ induces a perfect matching $M(\mathcal{P})$ in the bipartite graph $G = r(\mathcal{I})$.*

Proof. Let $\mathcal{P} = I_1, \ldots, I_p$ be a partition. Write I_q as $I_q = \{i_1, i_2, \ldots, i_k\}$ such that $i_1 < i_2 < \cdots < i_k$, then define

$$M(I_q) := \{(s_q, b_{i_1}), (a_{i_k}, d_q)\} \cup \{a_{i_j}, b_{i_{j+1}} : j \in [k-1]\}$$

and

$$M(\mathcal{P}) := \bigcup_{i=1}^{p} M(I_i).$$

To see that $M(\mathcal{P})$ is a perfect matching, observe that every node in G is covered exactly once. For each $q \in [p]$, the source s_q and destination d_q are used once in $M(I_q)$ and are not part of any other $M(I_z)$ for $z \neq q$. Furthermore, as $\mathcal{P}$ is a partition of $[n]$, every index $j \in [n]$ belongs to exactly one set I_q. The definition of $M(I_q)$ creates a chain of edges for the indices in I_q, covering each corresponding node a_j and b_j exactly once within that chain. Thus, every node in G is included in precisely one edge of $M(\mathcal{P})$. □

Lemma 4. *Let $\mathcal{I} = (S, p)$ be an instance of the String Partitioning Problem. Every perfect matching M on $G = r(\mathcal{I})$ induces a partition $\mathcal{P}(M)$ of $[n]$.*

Proof. Let M be a perfect matching on G. For a node u in the bipartite graph, we let $M(u)$ be the node matched to u in M. The matching M naturally defines p sets $I_q = \{i_1, \ldots, i_k\}$ such that $i_1 < i_2 < \cdots < i_k$, that can be inductively defined as follows: i_1 is such that $b_{i_1} = M(s_i)$, i_2 is such that $b_{i_2} = M(a_{i_1})$, $\ldots$, i_k is such that $t_i = M(a_{i_k})$. It remains to show that $I_1, \ldots, I_p$ form a partition of S.

Assume that an index $j \in [n]$ does not appear in any set I_q. By the construction of our bipartite graph, the node b_j corresponds to position j in the string. If j is not included in any partition, then b_j cannot be matched to any node in V_1, contradicting the requirement that M is a perfect matching.

Moreover, assume that an index $j \in [n]$ appears in two distinct sets I_i and I_z where $i \neq z$. Assume also, without loss of generality, that j is the smaller index in the sets I_i and I_z. For the definition of $M(I_q)$, where $I_q = \{i_1, i_2, \ldots, i_k\}$ such that $i_1 < i_2 < \cdots < i_k$, we know that $s_q = M(b_{i_1})$. This leads to a contradiction since if j is the smaller index in both I_i and I_z then it has to be matched with two distinct sources s_i and s_z. However, in a matching, each node can be incident to at most

one edge. Therefore, b_j cannot be matched to multiple nodes, which contradicts the definition of a matching. Hence, if $I_i \neq I_z$, then $I_i \cap I_z = \emptyset$.

Hence, for M to be a perfect matching, each index $j \in [n]$ must appear in exactly one set I_i, ensuring that the sets $\{I_1, I_2, \ldots, I_p\}$ form a partition of $[n]$. □

Lemma 5. *Let $\mathcal{I} = (S, p)$ be an instance of the String Partitioning Problem. Let moreover M be a perfect matching on $G = r(\mathcal{I})$. Then $W(M) = \delta(\mathcal{P}(M))$. Conversely, let $\mathcal{P}$ be a partition of S, then $\delta(\mathcal{P}) = W(M(\mathcal{P}))$.*

Proof. The weight $W(M)$ is the sum of the weights of all edges in the matching. By construction of the bipartite graph:

- Each source node s_i contributes weight 1 to connect to the first element of I_i,
- Each edge (a_j, b_k) with $j < k$ contributes weight 0 if $S_j = S_k$, or weight 1 if $S_j \neq S_k$,
- Each edge to destination node d_i contributes weight 0.

Therefore, let M be a perfect matching on G. By Lemma 4, M induces a partition $\mathcal{P}(M) = \{I_1, \ldots, I_p\}$ of $[n]$. The matching M naturally defines p sets $I_q = \{i_1, \ldots, i_k\}$ such that $i_1 < i_2 < \cdots < i_k$, that can be inductively defined as follows: i_1 is such that $b_{i_1} = M(s_i)$, i_2 is such that $b_{i_2} = M(a_{i_1})$, …, i_k is such that $t_i = M(a_{i_k})$. Thus, the following equation holds.

$$\begin{aligned} W(M) &= p + |\{(a_j, b_k) \in M : S_j \neq S_k\}| \\ &= p + \sum_{i=1}^{p} |\{j \in [|I_i| - 1] : S[I_i]_j \neq S[I_i]_{j+1}\}| \\ &= \sum_{i=1}^{p} \Big(1 + |\{j \in [|I_i| - 1] : S[I_i]_j \neq S[I_i]_{j+1}\}|\Big) \\ &= \sum_{i=1}^{p} \tau(S[I_i]) \\ &= \delta(\mathcal{P}(M)) \end{aligned}$$

The first equality holds because the total weight $W(M)$ is the sum of weights from the p source edges (each with weight 1) and the internal edges with weight 1. The second equality rewrites this count by summing the run boundaries for each partition set I_i. The third equality distributes the constant p into the summation. The fourth holds by the definition of $\tau(S[I_i])$ as the number of runs (1 plus the number of character changes). The final equality is by the definition of $\delta(\mathcal{P}(M))$.

Conversely, let $\mathcal{P} = \{I_1, \ldots, I_p\}$ be a partition of S. By Lemma 3, $\mathcal{P}$ induces a perfect matching $M(\mathcal{P}) = \bigcup_{q=1}^{p} M(I_q)$. Since the edge sets $M(I_q)$ are disjoint, the total weight of the matching is the sum of the weights of each part: $W(M(\mathcal{P})) = \sum_{q=1}^{p} W(M(I_q))$.

Let's analyze the weight of a single component $M(I_q)$ for a set $I_q = \{i_1, \ldots, i_k\}$ with $i_1 < \cdots < i_k$. The edges in $M(I_q)$ are (s_q, b_{i_1}), (a_{i_k}, d_q), and $(a_{i_j}, b_{i_{j+1}})$ for $j \in [k-1]$.

According to the weight definitions, $w(s_q, b_{i_1}) = 1$, $w(a_{i_k}, d_q) = 0$, and $w(a_{i_j}, b_{i_{j+1}})$ is 1 if $S_{i_j} \neq S_{i_{j+1}}$ and 0 otherwise. The weight of $M(I_q)$ is therefore:

$$\begin{aligned} W(M(I_q)) &= w(s_q, b_{i_1}) + \sum_{j=1}^{k-1} w(a_{i_j}, b_{i_{j+1}}) + w(a_{i_k}, d_q) \\ &= 1 + |\{j \in [k-1] : S_{i_j} \neq S_{i_{j+1}}\}| \\ &= \tau(S[I_q]) \end{aligned}$$

Summing over all sets in the partition, we directly equate the matching weight to the partition cost:

$$W(M(\mathcal{P})) = \sum_{q=1}^{p} W(M(I_q)) = \sum_{q=1}^{p} \tau(S[I_q]) = \delta(\mathcal{P})$$

This confirms that the weight of the induced matching is equal to the cost of the partition. □

Theorem 6. *Let $\mathcal{I} = (S, p)$ be an instance of the String Partitioning problem with $S = a_1 a_2 \dots a_n$. An optimal solution of $r(\mathcal{I})$ can be used to compute an optimal solution of $\mathcal{I}$.*

Proof. By the previous lemmas, we have established a bijective correspondence between partitions of S and perfect matchings in the bipartite graph G constructed by $r(\mathcal{I})$:

- Any partition $\mathcal{P}$ of S induces a perfect matching $M(\mathcal{P})$ in G (Lemma 3).
- Any perfect matching M in G induces a partition $\mathcal{P}(M)$ of S (Lemma 4).
- The weight equivalence holds: $W(M) = \delta(\mathcal{P}(M))$ and $\delta(\mathcal{P}) = W(M(\mathcal{P}))$ (Lemma 5).

Therefore, finding a minimum weight perfect matching M in G is equivalent to finding an optimal partition $\mathcal{P}$ for the String Partitioning Problem. □

In the following example, we show how a perfect matching can be used to retrieve a partition.

Example 3.6:
Consider the example in Figure 3.5, which shows a perfect matching for the instance $r(\mathcal{I})$ with $\mathcal{I} = (S = \text{AAB}, p = 2)$. The solid arrows represent the edges of the matching M. The dashed arrows are a visual aid showing the correspondence between a character's representation in $\mathcal{T}_2$ (on the right) and its representation in $\mathcal{T}_1$ (on the left), which is essential for tracing the paths.

We extract two substrings:

- **Substring 1 (red):**
  1. Start at source s_1. The matching edge is (s_1, v_1), where v_1 corresponds to the first character, $S_1 = \text{A}$. The subsequence is now "A".

2. Following the conceptual link to u_1, we find the matching edge (u_1, v_2), where v_2 corresponds to the second character, $S_2 = \text{A}$. The subsequence is now "AA".
3. Following the link to u_2, we find the matching edge (u_2, d_1). Since d_1 is a destination node, the path terminates.
4. The final subsequence is $S_1 = \text{'AA'}$.

- **Substring 2 (blue):**
  1. Start at source s_2. The matching edge is (s_2, v_3), where v_3 corresponds to the third character, $S_3 = \text{B}$. The subsequence is "B".
  2. Following the link to u_3, we find the matching edge (u_3, d_2). Since d_2 is a destination, the path terminates.
  3. The final subsequence is $S_2 = \text{B}$.

This example illustrates how the procedure correctly reconstructs the two substrings from the perfect matching, yielding the partition $\Pi = \{\text{AA}, \text{B}\}$.

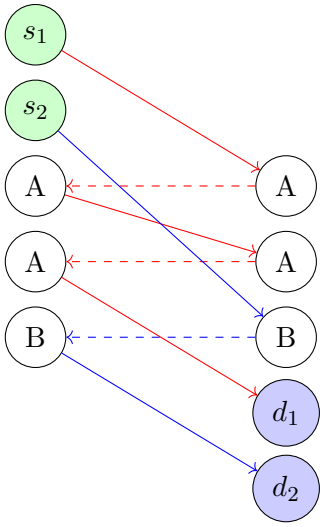


Figure 3.5: An example of a perfect matching (solid lines) in the constructed bipartite graph. The matching defines a partition into two paths (red and blue), which are traced by following the solid and dashed arrows.

Example 3.7:
Let's apply the reduction to the string $S = \text{ABCDDCBDDDD}$ from our running example (Example 3.1), with a target of $p = 2$ subsequences. In Figure 3.6 we have one of the possible minimum perfect matchings for the instance having weight 5.

We can trace the two paths from the source nodes to the destination nodes to obtain the following optimal partition:

- **Subsequence 1:** The path starting from s_1 traces the characters corresponding to indices (1, 3, 6, 7), yielding the subsequence $S_1 = \text{"ACCB"}$. The number of runs is $\text{runs}(S_1) = 3$.

- **Subsequence 2:** The path starting from s_2 traces the characters for indices (2, 4, 5, 8, 9, 10, 11), yielding the subsequence S_2 = "BDDDDDD". The number of runs is runs(S_2) = 2.

The total cost of this partition is the sum of the runs, $3 + 2 = 5$, which matches the weight of the perfect matching. This demonstrates how the reduction finds an optimal solution for the String Partitioning Problem.

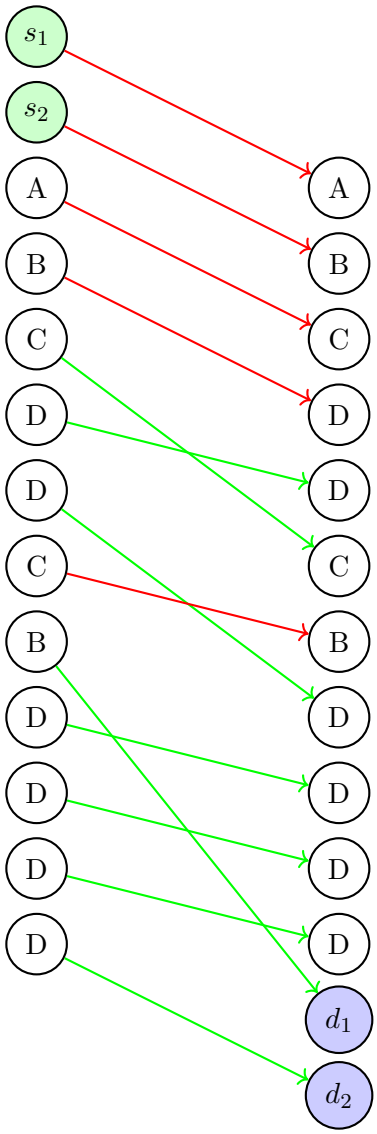


Figure 3.6: Example of an optimal perfect matching for the graph G of the MWPBM instance $r((S = ABCDDCBDDDD, p = 2))$. Green edges weigh 0, while red edges weigh 1.

3.3.3 Future Improvements

The reduction proposed in this thesis allows us to find an optimal solution for the String Partitioning Problem. However, it requires the construction of a graph with $O(n^2)$ edges, where n is the length of the input string. This is a significant drawback, as it makes the pipeline unfeasible for real-world datasets.

A more efficient bipartite graph construction is currently under study. The goal is

to reduce the number of edges in the graph to $O(np)$, while maintaining the same optimal solution cost as the reduction with $O(n^2)$ edges.

To improve the efficiency of the reduction, we are studying a more sparse construction for the bipartite graph, replacing the dense connection of each node $u_i \in \mathcal{T}_1$ to every node $v_j \in \mathcal{T}_2$ (where $j > i$). Consider an instance $\mathcal{I} = (S, p)$ of the String Partitioning problem. In the refined weighted bipartite graph $G = (V_1 \cup V_2, E, w)$ of $r(\mathcal{I})$, for each node $u_i \in V_1$, we define the edge set as follows:

1. **Zero-weight edge:** If there exists $j \in \{i+1, i+2, \ldots, n\}$ such that $S_i = S_j$, then we add edge $(u_i, v_{j'})$ with weight 0, where

$$j' = \min\{j \in \{i+1, i+2, \ldots, n\} : S_i = S_j\}$$

2. **Unit-weight edges:** Let $\mathcal{C}_i = \{c \in \Sigma : c \neq S_i\}$ be the set of characters distinct from S_i. For each character $c \in \mathcal{C}_i$, define

$$j_c = \min\{j \in \{i+1, i+2, \ldots, n\} : S_j = c\}$$

if such j exists. Then we add edges (u_i, v_{j_c}) with weight 1 for the first p distinct characters in $\mathcal{C}_i$.

This leads to a graph with $O(np)$ edges, where n is the size of S.

Example 3.8:
Consider the example in Figure 3.7, where $p = 2$. The first node labeled A in V_1 is connected to the first node labeled A in V_2 with weight 0. Also, it is connected only to the first p distinct nodes in V_2 with weight 1.

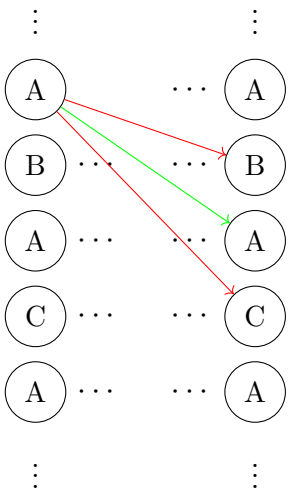


Figure 3.7

Chapter 4

Experiments

4.1 Experimental Setup

All experiments were conducted on a consistent hardware and software platform to ensure fair and reproducible comparisons. This section details the environment and datasets used in our evaluation.

4.1.1 Hardware and Software

The experimental platform is a MacBook Pro equipped with an Apple M4 Pro processor and 24 GB of RAM, running macOS. Our proposed algorithm was implemented in C++ and it is available at . We compiled it using Apple Clang 17.

In particular, we utilized the PathSort algorithm implementation of the XBWT proposed in Subsection 2.4.8. Moreover, we used a C++ implementation by Vladimir Kolmogorov of the minimum cost perfect matching algorithm described in [21]. The implementation is available at `https://pub.ista.ac.at/~vnk/software.html`.

4.1.2 Datasets

To systematically evaluate performance under controlled conditions, particularly with respect to repetitiveness, we generated a suite of synthetic labeled tries. Real-world datasets often lack ground-truth information about their repetitive structures, making it difficult to isolate the effects of this property. Our synthetic generation approach allows us to create trees with tunable characteristics.

The tries were generated using a custom Python script that constructs a tree and introduces repetitiveness by randomly copying subtries to different locations. At each step of the tree's growth, with a certain probability, a random existing subtrie is selected and duplicated as a new branch. This "copy-paste" mechanism allows us to create complex, highly repetitive structures that mimic patterns found in real-world data.

The generation process was controlled by the following key parameters:

Max Branching Factor The maximum number of children for any node.

Repetition Probability The probability of copying an existing subtrie instead of creating a new random branch.

Subtrie Depth Range The minimum and maximum depth of subtries eligible for copying.

Max Nodes The target maximum number of nodes in the generated trie.

Alphabet Size The number of unique characters in the alphabet.

Seed The seed used for random number generation to ensure reproducibility.

By varying these parameters, we can produce a diverse range of tries to thoroughly test the limits and behaviors of each compression algorithm.

4.2 Compression as a Parameter of p

Our first experiment investigates the relationship between the co-lexicographical width, p, of the automaton produced by our compression pipeline and the final compression size. The central hypothesis is that a larger value of p directly leads to a better compression ratio as shown in [27] (see **??**).

To generate automata with varying co-lex widths, we controlled the structural repetitiveness of the input tries using the `repetition_probability` parameter in our data generator. We conducted two main experiments:

1. **Low-Repetitivity Scenario:** We generated a set of 100 tries with a target size of 100,000 nodes, an alphabet size of 26 characters and a low repetition probability of 0.2.
2. **High-Repetitivity Scenario:** We generated a second set of 100 tries with a target size of 100,000 nodes, an alphabet size of 26 characters, but with a high repetition probability of 0.8.

We run our full compression pipeline on each trie with different values of p in range $[1, 15]$. We then report the maximum, mean, and minimum number of nodes and edges obtained after compression. By correlating the measured p with the final compression ratio in both scenarios, we aim to empirically demonstrate that the compression of our method is fundamentally governed by the co-lex width of the resulting automaton.

For each value of p, the experimental results are presented as boxplots, which provide a comprehensive statistical summary of the compression performance across all 100 trials. Each boxplot displays:

- The **median** (middle line): the central value that separates the upper and lower halves of the results.
- The **first quartile (Q1)** and **third quartile (Q3)** (box boundaries): representing the 25th and 75th percentiles, respectively, with the box containing the middle 50% of the data.
- The **interquartile range (IQR)**: the spread of the middle 50% of observations, calculated as Q3 - Q1.
- The **whiskers**: extending to the most extreme data points within 1.5 × IQR from the box boundaries.
- **Outliers**: individual points beyond the whiskers, representing unusually high or low compression ratios.

This visualization allows us to assess not only the central tendency of compression performance for each p value, but also the variability and distribution shape of the results across different trie structures. Additionally, the mean compression value for each p is highlighted with a red line plot that is overlaid on the boxplots to show the overall trend.

The results from our experiments, presented in Figure 4.1 and Figure 4.2, confirm that increasing the parameter p leads to a substantial improvement in compression for both low- and high-repetition datasets. As p grows, the number of nodes and transitions in the compressed automaton decreases significantly, demonstrating the effectiveness of the string partitioning approach in identifying and merging MN-equivalent nodes while keeping the co-lexicographical width of the automaton controlled.

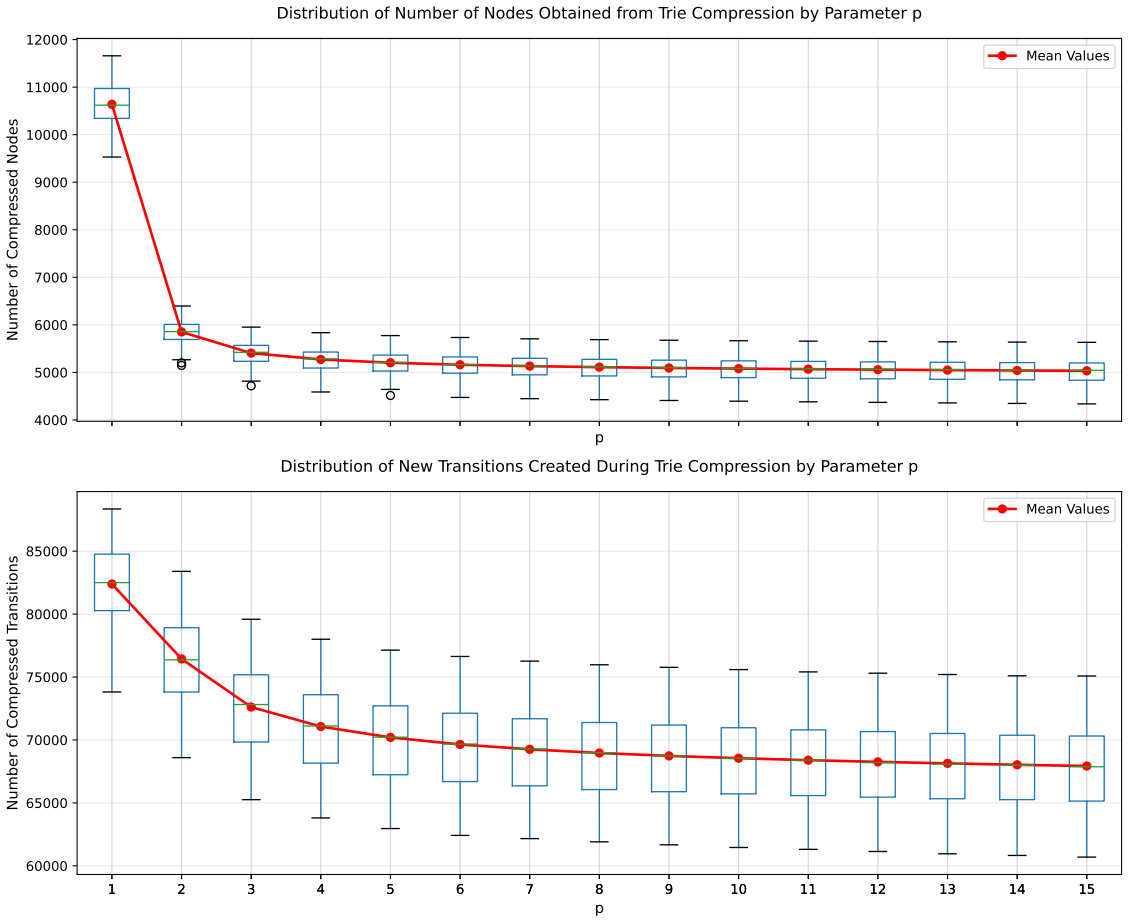


Figure 4.1: Experimental results for the low-repetition scenario. The top plot shows the number of nodes in the compressed trie as a function of p, while the bottom plot shows the number of transitions.

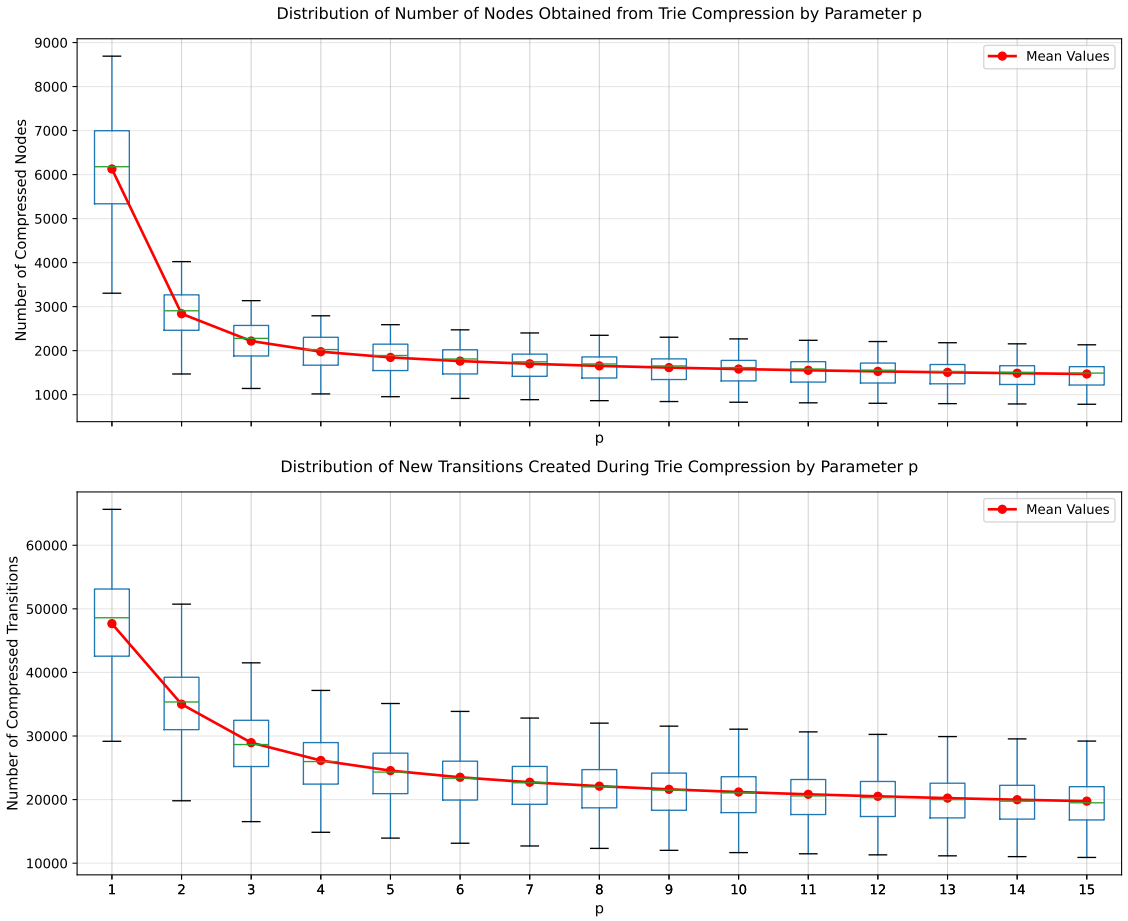


Figure 4.2: Experimental results for the high-repetition scenario. The top plot shows the number of nodes in the compressed trie as a function of p, while the bottom plot shows the number of transitions.

To provide a clear comparison between the two scenarios, Figure 4.3 shows the mean and standard deviation for both the low- and high-repetition datasets. The results confirm our hypothesis: tries with higher string repetitiveness achieve significantly better compression, as evidenced by the lower number of nodes and transitions for all values of p. This is expected, as a more repetitive trie allow for a more compact representation in the compressed automaton.

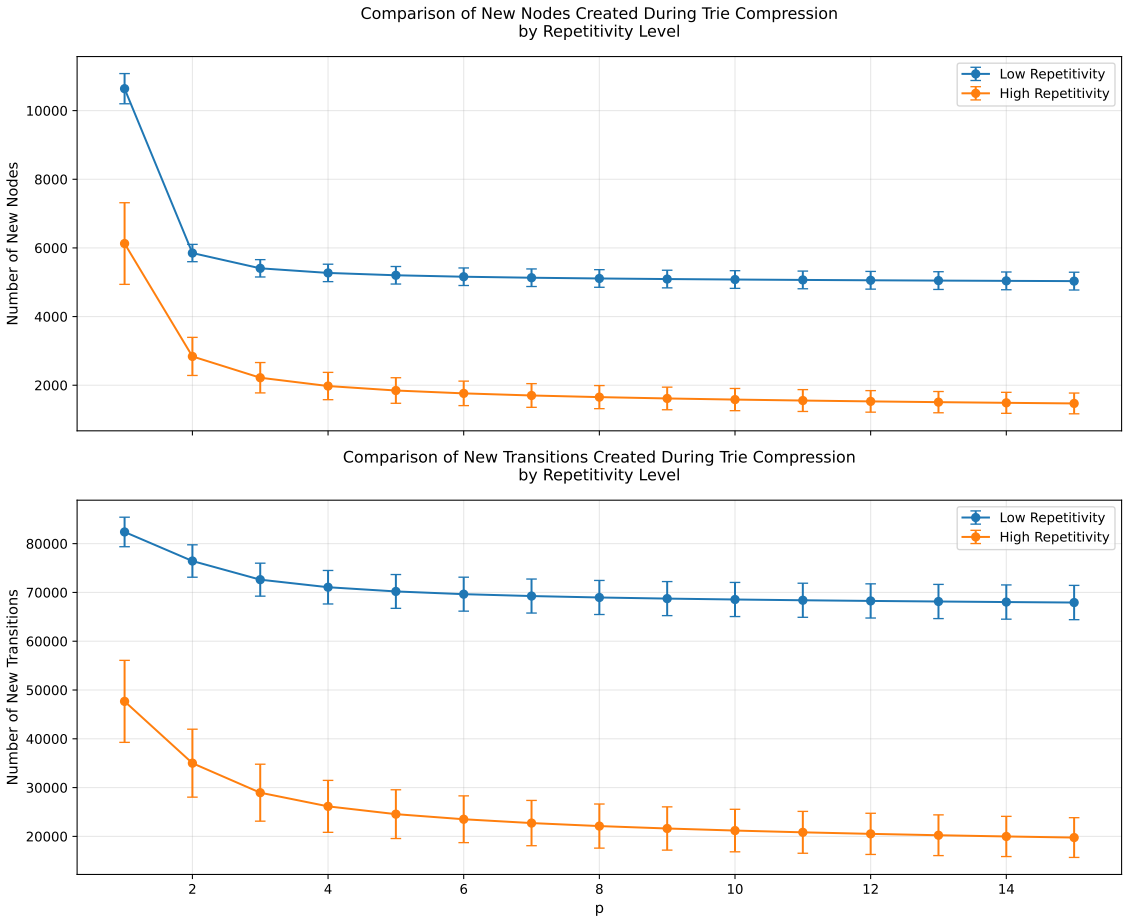


Figure 4.3: Comparison of compression performance between the low-repetition and high-repetition scenarios. The top plot compares the number of nodes, and the bottom plot compares the number of transitions.

Chapter 5

Conclusions and Future Works

This thesis has presented a preliminary study of a novel pipeline designed for trie compression. The initial results are promising and suggest a strong potential for this approach. Specifically, our findings demonstrate that a minor increase in the co-lexicographical width of the input trie can achieve significant compression, particularly when applied to tries with a high degree of internal repetitiveness.

A central contribution of this research is the development of a methodology to balance trie compression with indexability. We framed this challenge as a String Partitioning problem, aiming to identify the optimal p-sortable compressed automaton for a given input trie. The core of our approach lies in a strategic partitioning of the trie's nodes. This partitioning is designed to maximize compression while ensuring the resulting automaton is p-sortable, thus controlling balance between a compact representation and efficient indexing. Furthermore, we demonstrated that this optimization problem can be reduced to finding a minimum-weight perfect matching in a bipartite graph, which allows for the identification of optimal node pairings to achieve the highest degree of compression.

The reason we specifically aim for a p-sortable automaton is that it allows to find a balance between storage efficiency and usability. While maximum compression could be achieved by converting the trie into a minimal directed acyclic graph (DAG), such a structure is generally difficult to index, rendering it inefficient for search operations. A p-sortable automaton, however, allows for the creation of an effective index. This index enables fast querying and data retrieval directly on the compressed form, thereby avoiding the performance bottleneck of decompressing the data before use. This method produces a p-sortable automaton that can be indexed effectively, allowing for efficient subsequent querying and data retrieval operations without a notable sacrifice in performance. This approach offers a valuable compromise in the trade-off between the extremes of full, un-indexable compression and uncompressed, fully-indexable structures.

5.1 Future Works

The research presented in this thesis is preliminary, and several avenues for future work have been identified. The reduction we have proposed for the String Partitioning problem, while effective, exhibits a quadratic time complexity with respect to the number of nodes in the trie. This computational cost renders it inefficient for very large-scale applications. A critical next step is to investigate improvements to this reduction to develop more scalable and efficient solutions (see Subsection 3.3.3 for further details). Subsequently, these optimized algorithms should be rigorously

evaluated on real-world datasets to validate their performance and practical applicability.

Another significant direction for future research is the exploration of methods to produce a p-sortable deterministic finite automaton directly from our pipeline, rather than the current p-sortable non-deterministic finite automaton. A deterministic representation would offer further advantages in terms of space efficiency. One potential approach to achieve this is to develop a pruning strategy for the output NFA. Such a method would need to carefully remove states and transitions in a way that transforms the NFA into an equivalent DFA, ensuring that the recognized language remains unchanged.

The research and development of these future works will be continued during the author's PhD program.

Bibliography

[1] Jarno Alanko et al. “Wheeler languages”. In: *Information and Computation* 281 (2021), p. 104820 (cit. on p. 41).

[2] Jean Berstel et al. “Minimization of automata”. In: *arXiv preprint arXiv:1010.5318* (2010) (cit. on p. 34).

[3] Philip Bille et al. “Tree Compression with Top Trees”. In: *Information and Computation* 243 (Aug. 2015), pp. 166–177. ISSN: 08905401. DOI: 10.1016/j.ic.2014.12.012. URL: https://linkinghub.elsevier.com/retrieve/pii/S0890540114001643 (visited on 08/18/2025) (cit. on pp. 11, 12).

[4] M. Burrows and D. Wheeler. *A block-sorting lossless data compression algorithm*. Tech. rep. 124. Digital Equipment Corporation, 1994 (cit. on pp. 1, 14, 21, 42).

[5] 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. 45).

[6] Nicola Cotumaccio and Nicola Prezza. “On Indexing and Compressing Finite Automata”. In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, Jan. 2021, pp. 2585–2599. ISBN: 9781611976465. DOI: 10.1137/1.9781611976465.153 (cit. on pp. 13, 39, 42).

[7] 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 pp. 2, 3, 39, 51).

[8] Robert P Dilworth. “A decomposition theorem for partially ordered sets”. In: *The Dilworth Theorems: Selected Papers of Robert P. Dilworth*. Springer, 1990, pp. 7–12 (cit. on p. 38).

[9] 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. 45).

[10] 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. 45).

[11] Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. “Graphs Cannot Be Indexed in Polynomial Time for Sub-Quadratic Time String Matching, Unless SETH Fails”. In: *Theoretical Computer Science* 975 (Oct. 2023), p. 114128. ISSN: 03043975. DOI: 10.1016/j.tcs.2023.114128. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397523004413 (visited on 08/22/2025) (cit. on pp. 1, 39).

[12] 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, 11, 13, 17, 22).

[13] 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. 45).

[14] Travis Gagie, Giovanni Manzini, and Jouni Sirén. "Wheeler graphs: A framework for BWT-based data structures". In: *Theoretical computer science* 698 (2017), pp. 67–78 (cit. on pp. 2, 38, 39).

[15] Pawel Gawrychowski and Artur Jez. "LZ77 Factorisation of Trees". In: *LIPIcs, Volume 65, FSTTCS 2016* 65 (2016). Ed. by Akash Lal et al., 35:1–35:15. ISSN: 1868-8969. DOI: 10.4230/LIPICS.FSTTCS.2016.35. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSTTCS.2016.35 (visited on 08/18/2025) (cit. on p. 12).

[16] Michel Goemans. *Advanced Algorithms: Matching Notes*. Lecture notes, Massachusetts Institute of Technology. 2009. URL: https://math.mit.edu/~goemans/18433S09/matching-notes.pdf (cit. on p. 43).

[17] 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. 28).

[18] 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. 28).

[19] 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. 32).

[20] 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. 17, 19).

[21] Vladimir Kolmogorov. "Blossom V: A New Implementation of a Minimum Cost Perfect Matching Algorithm". In: *Mathematical Programming Computation* 1.1 (July 2009), pp. 43–67. ISSN: 1867-2949, 1867-2957. DOI: 10.1007/s12532-009-0002-8. URL: http://link.springer.com/10.1007/s12532-009-0002-8 (visited on 08/13/2025) (cit. on p. 62).

[22] 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. 11).

[23] 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. 45).

[24] 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. 12).

[25] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. "Tree structure compression with repair". In: *2011 Data Compression Conference*. IEEE. 2011, pp. 353–362 (cit. on p. 12).

[26] 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. 12).

[27] Giovanni Manzini et al. "The Rational Construction of a Wheeler DFA". In: *LIPIcs, Volume 296, CPM 2024* 296 (2024). Ed. by Shunsuke Inenaga and

Simon J. Puglisi, 23:1–23:15. ISSN: 1868-8969. DOI: 10.4230/LIPICS.CPM.2024.23. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CPM.2024.23 (visited on 08/23/2025) (cit. on pp. 2, 41, 63).

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

[29] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016 (cit. on p. 5).

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

[31] 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. 28).

[32] Dominique Revuz. "Minimisation of acyclic deterministic automata in linear time". In: *Theoretical Computer Science* 92.1 (1992), pp. 181–189 (cit. on pp. 1, 33).

[33] Piotr Sankowski. "Maximum weight bipartite matching in matrix multiplication time". In: *Theoretical Computer Science* 410.44 (2009), pp. 4480–4488 (cit. on p. 45).

Web bibliography

[16] Michel Goemans. *Advanced Algorithms: Matching Notes*. Lecture notes, Massachusetts Institute of Technology. 2009. URL: https://math.mit.edu/~goemans/18433S09/matching-notes.pdf (cit. on p. 43).