A New Compression Technique for Repetitive Tries

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

Master's Thesis Defence

Computer Science and Information Technology
Artificial Intelligence and Data Engineering

6th November 2025

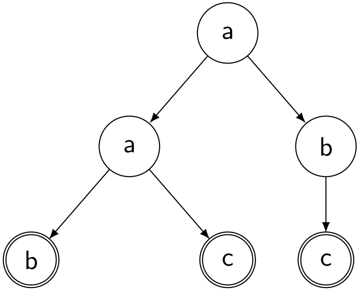
Candidate: Davide Tonetto **Supervisor**: Nicola Prezza **Co-Supervisor**: Alessio Campanelli

Introduction and Motivation

Motivation

- Tries are fundamental data structures for representing large sets of strings.
- Efficient for prefix-based queries, but often very large in memory.
- Typical applications:
  - Autocomplete and predictive text
  - Spell checking
  - IP routing
  - Bioinformatics (DNA pattern matching)
- Objective: reduce memory footprint while keeping efficient queries.

Example: Trie Representation



Each path from the root node a to a leaf represents one string in the language $\{aab, aac, abc\}$.

Challenges

- Tries can contain large, identical subtrees.
- Standard compression (e.g., XBWT, LZ77, top tree) does not exploit structural repetitions.

Fundamental trade-off between compression and indexability:

- **Full Indexability, No Compression:** the input trie itself can be used as an index. It offers no compression, as even highly repetitive subtrees are stored explicitly
- **Full Compression, Difficult Indexing:** maximum compression but very poor indexability.

Goal: find the sweet spot between these two extremes.

Theoretical Background

Tries, DFAs and Minimization

- A trie can be seen as an **acyclic DFA**.
- Minimizing a DFA merges **Myhill–Nerode equivalent states**.
- Revuz' algorithm allows linear-time minimization for acyclic DFAs.

Example: reduction of an acyclic DFA by merging equivalent subtrees.

XBWT (Extended Burrows–Wheeler Transform)

- Extends the classic BWT from strings to labeled trees.
- Encodes the tree as two arrays:
  - S_α: node labels
  - S_{last}: structure bits
- Enables compressed storage and prefix queries.

The children of each node form a contiguous block in S_α.

Proposed Compression Scheme

Motivation for a New Approach

- Full minimization $\Rightarrow$ excellent compression but unindexable.
- Uncompressed trie $\Rightarrow$ fully indexable but memory-hungry.

Our approach:

Partial Minimization

Produce a smaller automaton that remains indexable by allowing partial merging of equivalent subtrees.

Key concept: **p–sortable automata**.

p–Sortable Automata

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.

Let $\mathcal{A}$ be a p–sortable automaton. There exists a compressed data structure for A that supports subpath queries on a query word α of length m in $O(mp^2 \log\log(p|\Sigma|))$ time. The space required is:

- $\log(|\Sigma|) + \log p + 2$ bits per edge if A is a DFA.
- $\log(|\Sigma|) + 2\log p + 2$ bits per edge if A is an NFA.

The Intuition

- Parameter p interpolates between the two extremes:
  - $p = 1$: Wheeler graph $\Rightarrow$ maximum indexability, less compression.
  - $p \to \infty$: more compression, less indexability.
- Increasing p improves compression but complicates indexing.

Example of a 2–sortable automaton: partial merging with preserved order.

Compression Pipeline

1. Sort nodes by co–lexicographic order.
2. Partition nodes into p subsequences respecting the order.
3. Merge consecutive Myhill–Nerode equivalent states within each partition.
4. Construct the resulting compressed p–sortable automaton.

String Partitioning Problem

Consider a sting s.

Run

Maximal contiguous subsequence of identical symbols within s.

String Partitioning Problem

Partition s into p subsequences minimizing number of runs.

Example: String Partitioning Problem

$$s = 2213122152$$

of length 10. The number of runs in S is 8, given by the decomposition:

$$(22)\,(1)\,(3)\,(1)\,(22)\,(1)\,(5)\,(2)$$

A possible partition is:

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

This yields the following subsequences:

- $S[I_1] = 1115 \Rightarrow 2$ runs: (111), (5)
- $S[I_2] = 223222 \Rightarrow 3$ runs: (22), (3), (222)

Total runs: $2 + 3 = 5 \Rightarrow$ reduction from the original 8 runs in S.

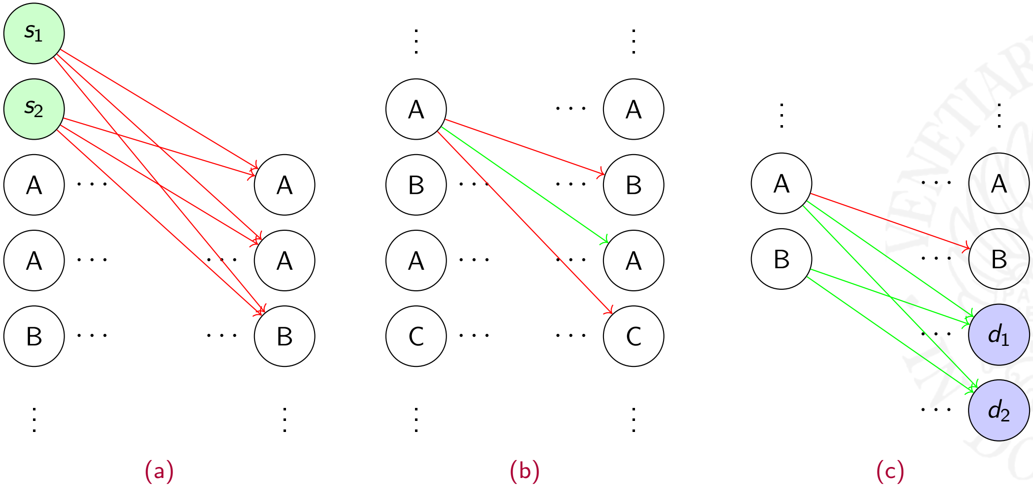
Reduction to Bipartite Graph Matching

- The String Partitioning Problem can be reduced to:

  Minimum Weight Perfect Bipartite Matching (MWPBM)

- Nodes correspond to string characters; edges encode run boundaries cost between characters.
- Allows to use efficient, well-studied algorithms to find the optimal solution.

Example: Min-Weight Bipartite Graph Matching



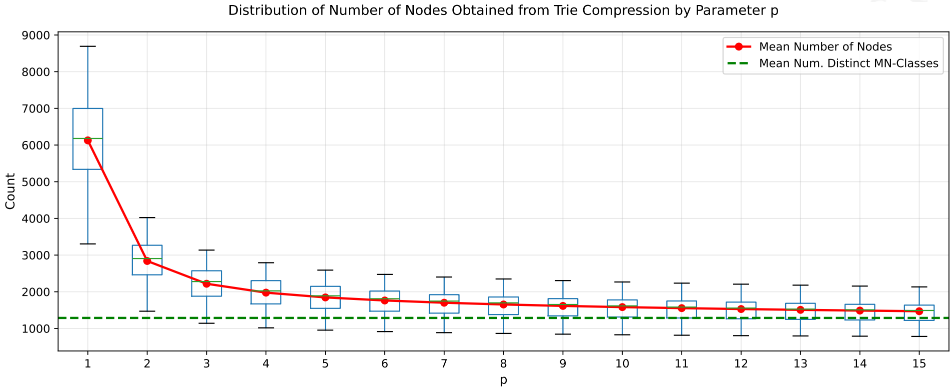
Implementation and Experiments

Experimental setup

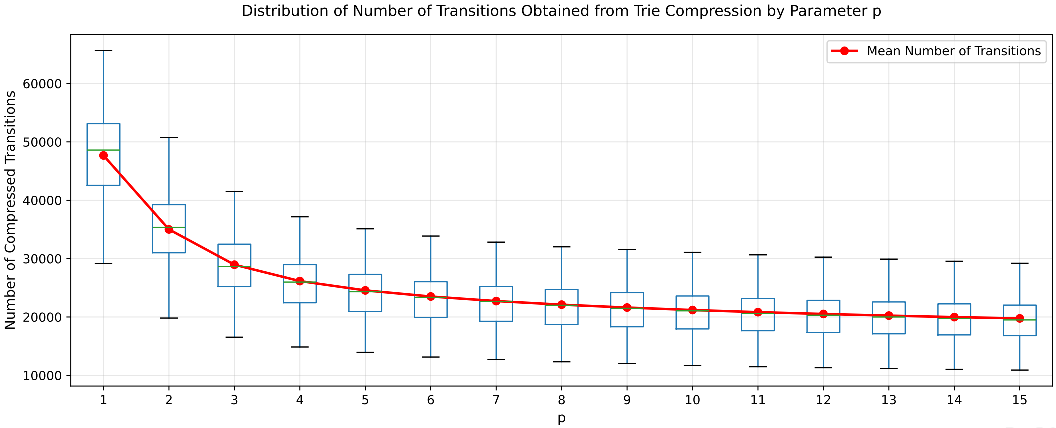
- Implemented in **C++** for performance.
- **CPU and RAM**: Apple M4 Pro, 24 GB.
- **Dataset**: synthetic generated tries to control repetitiveness.

Experimental Results

- Increasing p from 1 to 2 halves the number of states.
- Compression ratio improves rapidly, approaching the minimal number of states for larger p.



Experimental Results



Conclusions

Conclusions

- Proposed a new trie compression method based on p–sortable automata.
- Achieves:
  - Significant memory reduction on repetitive data.
  - Efficient querying capability.
  - Flexibility via parameter p.
- Opens new directions for compressed automata and string indexing.

Future Work

- **Scalability**: Investigate improvements to the proposed reduction for the String Partitioning problem to develop more scalable and efficient solutions for large-scale applications.
- **DFA Construction**: Explore methods to directly construct a p—sortable deterministic finite automaton from the pipeline, potentially by developing a pruning strategy for the output NFA.
- **DFA Minimization**: Minimize the size of the returned automaton, potentially by providing explicit guarantees of minimality of the returned p—sortable DFA.