Space-Efficient Algorithms for Strings and Prefix-Sortable Graphs

Jarno N. Alanko

Doctoral dissertation, to be presented for public examination with the permission of the Faculty of Science of the University of Helsinki, in Auditorium PII, Porthania, on May 3, 2020, at 16 o’clock.

University of Helsinki
Finland
Abstract

Space-efficient data structures are an active field of research that has found many applications in combinatorial pattern matching and bioinformatics. The idea is to build data structures that occupy space close to an information-theoretic lower bound, or even less, but still support efficient query operations. In the past few decades, compact index structures have been designed for a variety of different types of data, including bit vectors, strings, trees and graphs, to name a few prominent applications.

In this thesis, we design and apply compact data structures for problems related to bioinformatics, and advance the theory of Wheeler graphs, which are a class of graphs that admit a compact indexing data structure. The work is based on four published papers.

In the first two papers, we propose compact solutions for two problems on strings. In Paper I, we design and implement an algorithm that computes the classical greedy approximation for the shortest common superstring problem. In Paper II, we design and implement data structures for storing a variable-order Markov model in a compact and queryable form.

The last two papers of the thesis expand the theory of Wheeler graphs. In Paper III, we extend the theory into finite state automata, leading to a number
of interesting algorithms for recognizing, sorting, determinizing and minimizing automata that are Wheeler graphs. We also show how to turn any acyclic automaton into the minimum equivalent Wheeler graph automaton. In Paper IV, we propose a method to compress a Wheeler graph while retaining the indexing functionality, by generalizing a recently introduced method of tunneling from the Burrows-Wheeler transform to Wheeler graphs.

Computing Reviews (2012) Categories and Subject Descriptors:

Theory of computation → Design and analysis of algorithms → Data structures design and analysis → Data compression

Theory of computation → Design and analysis of algorithms → Data structures design and analysis → Pattern matching

Applied computing → Life and medical sciences → Bioinformatics

General Terms:
Data structures, Data compression, Bioinformatics

Additional Key Words and Phrases:
Wheeler graph, Burrows-Wheeler transform, Balanced parentheses, Tunneling, Variable-order Markov models, Regular languages, Deterministic finite-state automata
Acknowledgements

I would like to thank all the numerous people I have had the pleasure of working with. Every author on the papers in this thesis is distinct, except for myself. This was made possible by my supervisor Veli Mäkinen, who gave me the freedom to pursue the range of topics I was interested in, and to build my own network of people to work with. I thank Myers lab of the Max Planck Institute of Molecular Cell Biology and Genetics for hosting me to work on Paper II, Navarro’s group in the University of Chile for hosting me to work on Paper IV, and the EU RISE project BIRDS for funding the visit Chile. I thank the Academy of Finland and the Doctoral Programme in Computer Science of University of Helsinki for funding my PhD studies.

I would like to thank my mentors and co-authors Fabio Cunial and Djamal Belazzougui for guiding me all the way back from when I was a MSc student, and my co-author Travis Gagie for connecting me with Nicola Prezza for work that would eventually become Paper III of the thesis. I am grateful for my co-author Louisa Seelbach Benkner and my friend Jani Koskinen for feedback on the thesis manuscript, and my friend Mohsin Md. Khan for peer support throughout our four-year PhD journey.

A special thank you goes to my co-workers at Skrolli magazine for helping me improve my writing skills to a higher standard, and to the Finnish competitive programming community for helping me keep my programming skills sharp, and my ego low.

Helsinki, May 2020
Jarno N. Alanko
Original Papers

This thesis is based on four original publications, referred as papers I to IV in the thesis.

I Jarno Alanko and Tuukka Norri. **Greedy Shortest Common Superstring Approximation in Compact Space.** In *Proceedings of the 24th International Symposium on String Processing and Information Retrieval (SPIRE 2017).*


IV Jarno Alanko, Travis Gagie, Gonzalo Navarro and Louisa Seelbach Benkner. **Tunneling on Wheeler Graphs.** In *Proceedings of the 2019 Data Compression Conference (DCC 2019).*
Contents

1 Introduction 1
  1.1 Overview of the thesis 3
  1.2 Original papers and individual contributions 4

2 Preliminaries 7
  2.1 Strings and substrings 7
  2.2 Bit vectors 8
    2.2.1 Rank and select 8
    2.2.2 Balanced parentheses 9
  2.3 Suffix data structures 9
  2.4 The Burrows-Wheeler transform 12
  2.5 The bidirectional Burrows-Wheeler index 13

3 Greedy Superstring Approximation in Compact Space 15
  3.1 Ukkonen’s linear time superstring algorithm 17
  3.2 Superstrings in compact working space 19
  3.3 Results and discussion 21

4 Data Structures for Large Variable Order Markov Models 25
  4.1 Learning VOMMs from data 27
    4.1.1 Selecting contexts 27
    4.1.2 Estimating probability distributions for contexts 29
  4.2 The framework of Belazzougui and Cunial 30
  4.3 Pruning the topology 32
  4.4 Construction 33
  4.5 Results and discussion 35
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Wheeler Graphs and Regular Languages</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>The Wheeler conditions</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Wheeler graphs and sortability of path label sets</td>
<td>42</td>
</tr>
<tr>
<td>5.3</td>
<td>Finite state automata and Wheeler graphs</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>Tunneling on Wheeler graphs</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>Generalized tunneling</td>
<td>52</td>
</tr>
<tr>
<td>6.2</td>
<td>The special case of the BWT</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>Discussion</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>61</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The invention of high-throughput DNA sequencing machines has inundated the world with sequence data. These machines produce sequences of nucleotides, which can be seen as strings from an alphabet consisting of letters A, C, G and T. Searching for substrings and combinatorial patterns in this type of data is a crucial part of modern bioinformatics.

Fortunately, decades of computer science research has been devoted for the types of pattern matching problems that commonly arise in bioinformatics. The most basic problem is to find all occurrences of a query string in a reference string. An optimal algorithm for the problem was found already in the 70’s by Knuth, Morris and Pratt [38]. The text indexing problem is a slightly more demanding problem, in which we are asked to preprocess a reference string so that any number of query strings can be searched in time that is linear only in the lengths of the queries and the number of occurrences of the queries in the reference. This can be done for example by constructing a data structure known as the suffix tree, which is a fundamental string processing data structure introduced by Weiner in 1973 [75]. The suffix tree is fast to build and use, but the memory usage tends to be high.

More lightweight data structures have been invented to address the memory issue, most importantly the suffix array and the longest common prefix array. These are simple integer arrays of length \( n \), where \( n \) is the length of the reference data, still capable of supporting linear-time text indexing. However, it was discovered that the space usage of the indexing problem can still be significantly improved by using the Burrows-Wheeler transform (BWT) of the reference string. The BWT is an invertible permutation of the input string with strong connections to the suffix array and the suffix tree. The permuted string can be indexed to
support pattern matching queries against the original string, using only a sub-linear number of extra bits. The resulting data structure is called the FM-index, and it is used in some of the most popular software tools used in bioinformatics, e.g. the alignment tools Bowtie 2 [41], BWA [43] and SOAP2 [44].

When the number of distinct characters of the reference string is small, the FM-index has a large space advantage compared to the suffix array. In its basic form, the suffix array of a string of length \( n \) takes \( n \log n \) bits to store, whereas the FM-index takes only \( n \log \sigma + \sigma \log n + o(n \log \sigma) \) bits using a wavelet tree, where \( \sigma \) is the number of distinct characters in the reference string. In the typical application of nucleotide sequences, where \( \sigma = 4 \), the space is an order of magnitude smaller compared to the suffix array.

The space advantage comes at a cost, however. The query complexity is slower by a factor of \( O(\log \sigma) \) and the FM-index can only count the number of occurrences of a query pattern, but not efficiently locate them in the reference string. Useful time-space tradeoffs have been found to support locating occurrences of patterns (see e.g. [53] for a survey) but even then, the FM-index cannot easily solve all problems that a fully functional suffix tree is able to solve. For example, some problems depend crucially on pointers called suffix links, which are commonly stored with the suffix tree, but are hard to emulate on an FM-index.

Fortunately there exists a compromise between the FM-index and a suffix tree: a compressed suffix tree, such as the one by Sadakane [64] or Ohlebusch et al. [55]. These data structures are able to simulate the whole functionality of a suffix tree, using only \( O(n \log \sigma) \) bits of space, at a cost of slower query times and a higher constant coefficient in the space consumption compared to the FM-index. A compressed suffix tree does not represent the tree explicitly, but instead the tree is typically decomposed into three parts: a compressed suffix array, a data structure encoding the shape of the tree, and a data structure encoding the lengths of the edges of the tree. The result is a set of data structures that has a relatively small memory footprint, but also the expressive power of the suffix tree.

A whole field of compact data structures has emerged to support data structures like the compressed suffix tree. A data structure is said to be compact if it takes space close to an information-theoretic lower bound. For example, consider the set \( X \) of all possible strings of length \( n \) with characters from an alphabet of \( \sigma \) distinct characters. The number of strings in \( X \) is \( \sigma^n \), so any fixed-length encoding scheme for strings in \( X \) must use at least \( \log_2 \sigma^n = n \log_2 \sigma \) bits per string, because otherwise two distinct strings would have the same encoding. In general, any data structure for elements of a set \( X \) must take at least \( \log_2 |X| \)
bits, or otherwise two distinct elements $x_1$ and $x_2$ of $X$ would end up with the same data structure bit-by-bit, and therefore we would have lost the information to distinguish $x_1$ and $x_2$. The compressed suffix tree is an example of a compact data structure, whereas the regular suffix tree is not.

Generally, a data structure is considered compact even if it uses a linear number of extra bits on top of the information-theoretic lower bound. A succinct data structure on the other hand is allowed to use only a sublinear number of extra bits. The FM-index is an example of a succinct data structure, because the extra space is only $\sigma \log_2 n + o(n \log \sigma)$ bits (sublinear in $n$) larger than the lower bound of $n \log_2 \sigma$ for storing strings of length $n$ without loss of information.

Even smaller structures are called compressed data structures. These are able to go below the basic information-theoretic lower bound on average by assuming that the data is drawn from a non-uniform probability distribution over the set of possible data $X$. An example of such a data structure is the run-length compressed FM-index, which takes less than $n \log_2 \sigma$ bits if the data is known to be repetitive on average in a way that is exposed by the Burrows-Wheeler transform.

In this thesis, we design and apply compact, succinct and compressed data structures for pattern matching problems in strings and labeled graphs. The research of the thesis has been published in four peer-reviewed papers labeled papers I-IV. The next section provides an overview of the thesis.

1.1 Overview of the thesis

There are many problems that cannot be solved efficiently with the FM-index alone, but do not need the full power of the suffix tree either. In the first half of this thesis, we identify and solve two such problems. In the second half, we explore the properties of a generalization of the FM-index, called the Wheeler graph index.

The chapters are organized as follows. Chapter 1 is the introduction to the thesis, which you are reading right now. Chapter 2 gives a survey of the preliminary knowledge required in the subsequent chapters. The rest of the chapters each cover one of the original papers of the thesis.

In Chapter 3, we consider the problem of finding the shortest common superstring of a set of strings. The problem is NP-hard, but a greedy heuristic is known to work well in practice. The greedy heuristic can be implemented in linear time using the Aho-Corasick automaton, a data structure loosely related to the suffix tree. Unfortunately, the space of the automaton is $\Omega(n \log n)$ bits,
which exceeds the information-theoretic lower bound of $n \log \sigma$. We show how to implement the greedy heuristic in compact $O(n \log \sigma)$ bits space, using the BWT, a succinct data structure representing the shape of the suffix tree, and some small auxiliary arrays for bookkeeping required in the greedy heuristic.

In Chapter 4, we consider the problem of representing variable order Markov models (VOMM) in a compressed form that allows evaluating the model on a query string efficiently. For this too, we apply the BWT and a succinct data structure for the shape of the suffix tree, coupled with some marking bit vectors. The index is compressed by taking advantage of substring redundancy present in repetitive reference strings.

In Chapter 5, we study the class of graphs called Wheeler graphs [29]. A Wheeler graph is a type of a labeled directed graph that can be indexed for pattern matching queries in compact space, similarly to the FM-index. Intuitively, these are graphs where the nodes can be sorted by incoming path labels in a specific sense. The full definition is surprisingly subtle, and therefore we devote a part of Chapter 5 for introducing Wheeler graphs and exposing the connections to sorting. The purpose of this expository material is to familiarize the reader with Wheeler graphs, so that the reader is able to dive into our papers on Wheeler graphs more easily. The rest of Chapter 5 summarizes the contributions of Paper III, in which we lay groundwork for a theory of Wheeler graphs in the context of finite state automata.

In Chapter 6, we generalize BWT tunneling [5] to Wheeler graphs. Tunneling is an attempt to exploit redundancy in a BWT that is not captured by other known compressors for the BWT. In Paper IV, we generalize the concept to Wheeler graphs, bringing the benefits of tunneling to all data structures that can be interpreted as a Wheeler graph. We also make progress in the special case of the tunnelled BWT, showing how to support full FM-index functionality on top of the compressed representation.

### 1.2 Original papers and individual contributions

Below is a list of the original papers that form the basis of this thesis, and my contributions to each of the papers.

**Paper I**: This paper presents the space-efficient superstring algorithm discussed in Chapter 3. I was mostly responsible for the design of the algorithm, while the other author was mostly responsible for the implementation and experiments.
1.2 Original papers and individual contributions

**Paper II**: In this paper, we present the framework for variable-order Markov models discussed in Chapter 4. My main contribution to the paper is the practical implementation, whereas the design is joint work with all three authors.

**Paper III**: This paper extends the theory of Wheeler graphs to finite state automata. The results are summarized at the end of Chapter 5. This work was done jointly among all the authors, with the fourth author leading the project.

**Paper IV**: In this paper, we generalize the concept of BWT tunneling to Wheeler graphs. Chapter 6 defines our notion of generalized tunneling and summarizes our results. All authors contributed equally to the work.
1 Introduction
Chapter 2

Preliminaries

This thesis stands on the shoulders of a large amount of research on string processing algorithms. In this chapter, we survey the fundamental string processing tools and concepts required to understand the papers of the thesis. Throughout the thesis we assume the RAM-model of computation with word size $O(\log n)$, where $n$ is the size of the input.

2.1 Strings and substrings

A string $T$ is a sequence of characters. The set of distinct characters in $T$ is called the alphabet of $T$. By convention, we denote the alphabet of a string with $\Sigma$ and the size of the alphabet with $\sigma$. In this thesis, the alphabet is always ordered, so the alphabet can be indexed so that $\Sigma[i]$ refers to the $i$-th character in the order of the alphabet. The length of string $T$ is denoted with $|T|$ and the character at index $i$ is denoted with $T[i]$. A substring of $T$ starting from index $i$ and ending at index $j$ is denoted with $T[i..j]$. A string of the form $T[i..|T|]$ for some $i$ is called a suffix of $T$ and a string of the form $T[1..i]$ for some $i$ is called a prefix of $T$. The reverse of $T$ is denoted with $\overline{T}$.

We follow some conventions for choosing variable names for strings. Generally, upper case Latin letters $A, B, \ldots$ are preferred, unless the string plays the role of a substring, in which case lower case Greek letters $\alpha, \beta, \ldots$ are generally used. For text indexing, the string that is being indexed is denoted with $T$ and a query string against the index is generally denoted with $Q$. Single characters are denoted with lower case Latin letters $a, b, \ldots$. The concatenation of two strings $A$ and $B$ is denoted with $AB$. In this notation either $A$ or $B$, or both, can also be replaced
with single characters. The notation $A^k$ means string $A$ repeated $k$ times. If $A$ is a set, then $A^k$ means the set of all strings of length $k$ using $A$ as the alphabet, and $A^*$ means the set of all strings of any length using $A$ as the alphabet.

A substring $c\alpha$ of $T$ with $c \in \Sigma$ and $\alpha \in \Sigma^*$ is called a left-extension of $\alpha$. Correspondingly, a substring of the form $\alpha c$ is called a right-extension of $\alpha$. A substring that has two or more left-extensions is called left-maximal, and a substring with two or more right-extensions is called right-maximal. A substring that is both left- and right-maximal is called maximal. A substring that occurs at least twice in $T$ is called a repeat.

2.2 Bit vectors

A bit vector is an array consisting of zeroes and ones. Many compact data structures are based on efficient implementations of primitive operations on bit vectors. In this section, we describe the bit vector operations used in this thesis. The symbol $B$ is used to denote the bit vector being indexed.

2.2.1 Rank and select

The rank operation takes an index $i$ and returns the number of ones in $B[1..i]$. The result is denoted with $\text{rank}_B(i)$. The select operation is an inverse of the rank operation. It takes an integer $j$, and returns the first position $i$ such that $\text{rank}_B(i) = j$. This is denoted with $\text{select}_B(j)$. If no such position $i$ exists, the value of the select operation can be defined to whatever is convenient in the specific application.

These two operations can be generalized to arrays where the elements are from an alphabet of size $\sigma$. In this case, the rank and select take an extra parameter which specifies which character we are counting. We denote with $\text{rank}_A(i, c)$ the number of occurrences of character $c$ in $A[1..i]$ and by $\text{select}_A(j, c)$ the first position $i$ such that $\text{rank}_A(i, c) = j$, with implementation-defined behaviour if no such position exists.

The rank and select operations for a bit vector $B$ can be implemented in $O(1)$ time using $o(|B|)$ bits of extra space [36], and in $O(\log \sigma)$ time for an array $A$ with $\sigma$ distinct elements using $o(|A| \log \sigma)$ bits of extra space [34].
2.2.2 Balanced parentheses

Bit vectors can be used to describe a tree succinctly. Recall that a tree is a connected graph with no cycles. In this thesis, trees always have a designated root node. The neighbor of a node that is the closest to the root is called the parent of the node (the root itself does not have a parent), and the other neighbors are the called children of the node. Nodes with no children are called leaves. Nodes that are not leaves are called internal nodes.

There exists a few alternative ways to represent a rooted tree succinctly [52], but the one we use in this thesis is the balanced parentheses representation (BPR). The BPR is defined recursively such that a leaf node is represented by ( ) and a non-leaf node is represented by $(C_1 \ldots C_k)$, where $C_1 \ldots C_k$ is the concatenation of the representations of the children of the node. This representation also defines an order for the children by the order of the concatenation. The BPR of the tree is the representation of the root node. See Figure 2.1 for an example. The parentheses sequence can be encoded as a bit vector by replacing open parentheses with ones and closing parentheses with zeroes.

The identifier of a node in the BPR is the pair of parentheses enclosing the children of the node, or just the open parenthesis of the pair. The BPR can be indexed in succinct space to support a wide variety of operations that take as input the identifier of a node. For example, is possible to support navigational operations such as moving to the parent or a child of a node, or taking the lowest common ancestor of two nodes. Other useful operations include queries for the depth of a node in the tree, finding the matching parenthesis for a given parenthesis, counting the number of leaves in a range and jumping to a leaf with a given rank. Nodes can also be marked such that we can support finding the lowest marked ancestor. All these operations can be made to run in constant time. See e.g. the book "Compact data structures: A practical approach" by Navarro [52] for details on all these operations.

2.3 Suffix data structures

The suffix tree is a fundamental data structure that exposes the substring structure of a string. It is an important tool for reasoning about string algorithms, although more lightweight data structures are usually preferred in most practical applications. It can be seen as a compact version of a simpler data structure called the suffix trie. A suffix trie for a string $T$ with alphabet $\Sigma$ is the smallest tree with edges labeled with characters from $\Sigma$, such that for each suffix $\alpha$ of $T$,
Figure 2.1: A tree with the root drawn at the top. The balanced parentheses representation of the tree is 
\[ (((()())(()()())(())))). \]
The sequence of parentheses is the same as the one generated by walking through the tree in depth-first left-to-right order, appending an open parenthesis every time the walk moves down, and a closing parenthesis every time the walk backtracks up, and enclosing the whole resulting sequence in one more pair of parentheses.

there exists a simple path starting from the root such that the concatenation of edge labels on the path is \( \alpha \). The children of a node are ordered from left to right according to the order of the edge labels in the alphabet.

The suffix tree is a suffix trie modified such that all non-branching paths of nodes are contracted into single edges. The label of an edge resulting from a contraction is the concatenation of the original edge labels on the path, starting from the label closest to the root. We call the concatenation of edge labels on a path from the root to some node \( v \) the path label of \( v \), and denote it with \( \ell(v) \). The locus (plural loci) of a substring \( \alpha \) of \( T \) is a pair \( (v,d) \), where \( v \) is the node furthest possible from the root such that \( \alpha \) is a prefix of \( \ell(v) \), and \( d \) is the length of \( \alpha \). The distinct loci in the suffix tree are in a one-to-one correspondence with the nodes of the suffix trie.

We denote the suffix tree of \( T \) with \( ST_T \). The subscript \( T \) is omitted when it is clear from context. The suffix tree can be constructed in linear time [25] for a string with integer alphabet. It can be represented in \( O(|T| \log |T|) \) bits of space e.g. by storing the original string \( T \), the list of children for each node, and representing edge labels as pairs of integers such that \((i,j)\) corresponds to string \( T[i..j] \).

A unique terminator character \( \$ \notin \Sigma \) that is smaller than all characters in \( \Sigma \) is usually appended to \( T \) before building the suffix tree. With this, the suffix tree has the useful property that every suffix \( \alpha \) of \( T \) corresponds to a distinct leaf in the suffix tree, such that the path label of the leaf is \( \alpha \). In this thesis, we always assume that \( T \) contains the terminator symbol. The leaves are labeled with an
integer specifying the starting position of the corresponding suffix in \( T \). The path labels of the internal nodes are exactly the right-maximal substrings of \( T \).

**Suffix links** are pointers from a locus \((c, d)\) to the locus \((\alpha, d - 1)\). If \( \alpha \) is right-maximal, then the link is *explicit*, otherwise the link is *implicit*. The inverse pointers, i.e. those from \((\alpha, d - 1)\) to \((c, d)\), are called *Weiner links*. All loci except for the root have exactly one outgoing suffix link, but nodes can have multiple outgoing Weiner links. The explicit suffix links form a tree called the *suffix link tree*.

The **suffix array** of \( T \) is an array \( SA_{T}[1..|T|] \) such that \( SA_{T}[i] \) is the starting position of the suffix with rank \( i \) in a lexicographically sorted list of all suffixes of \( T \). Equivalently, it is the list of the integer labels of the leaves of \( ST_{T} \), read from left to right. The subscript \( T \) in \( SA_{T} \) is dropped when it is clear from context. The suffix array is often used in combination with the **longest common prefix array** \( LCP_{T}[1..(|T| - 1)] \), which is an array such that \( LCP_{T}[i] \) is the length of the longest common prefix between suffixes starting from \( SA_{T}[i] \) and \( SA_{T}[i + 1] \).

Suffix structures like the suffix tree and the suffix array facilitate efficient substring searching algorithms on \( T \). The basis for these algorithms is that every substring of \( T \) is a prefix of some suffix of \( T \), and moreover, the suffixes that are prefixed by some fixed string \( \alpha \) are consecutive in the lexicographically sorted list of suffixes. We call the range of suffixes in the sorted list that are prefixed by \( \alpha \) the *lexicographic range* or the *lexicographic interval* of \( \alpha \). The term *suffix array interval* is also used in the literature. The rank of a suffix in the lexicographically sorted list of suffixes is called the lexicographic rank of that suffix.

Substring search algorithms typically work by repeatedly extending the search pattern to the right, starting from the empty string, while maintaining the lexicographic range or the suffix tree locus of the current pattern. The starting positions of the occurrences can be read from the suffix array, given the lexicographic range.

The **Aho-Corasick automaton** (AC-automaton) is a suffix data structure somewhat related to the suffix tree. The difference is that while the suffix tree indexes a reference string \( T \) and can be used to find the occurrences of any query string, the AC-automaton indexes a set of query strings and can be used to find all occurrences of the query strings in any reference string \( T \). AC-automaton for a set of strings \( S \) is the trie of \( S \) augmented with some extra information. First, the nodes \( v \) such that \( \ell(v) \in S \) are marked. Second, two kinds of extra pointers are added to the tree: *failure links* and *output links*. All nodes, except for the root, have an outgoing failure link: the failure link from node \( v \) points to the node \( u \) such that \( \ell(u) \) is a proper suffix of \( \ell(v) \) and \( \ell(u) \) is the longest possible. The failure links are analogous to the suffix links in a suffix tree. The output links are
pointers that point from a node \( v \) to the first marked node on the path of failure links from \( v \) (if exists). The automaton can be used to search for the strings of \( S \) in any string \( T \) in time that is linear in \( |T| \) plus the number of occurrences found.

2.4 The Burrows-Wheeler transform

The Burrows-Wheeler transform (BWT) of a string \( T \) is the string formed by listing all rotations of \( T \), sorting them in lexicographic order, and taking the last character of each rotation. See Figure 2.2 for an example. Usually it is assumed that the last character of \( T \) is a unique terminator \( \$ \) that is smaller than all other characters in \( T \). This guarantees that the transformation is invertible. When the terminator is present, the BWT can also be defined in terms of the suffix array as follows: \( BWT[i] = T[SA[i] - 1] \) if \( SA[i] \neq 1 \), and \( BWT[i] = T[|T|] \) otherwise.

The FM-index is a data structure based on the BWT of \( T \), that can be used to search for substrings of \( T \). It consists of two parts: the BWT of \( T \) with support for rank queries (see Section 2.2), and an array \( C \) indexed by the characters of the alphabet of \( T \) such that \( C[c] \) is the number of positions in \( T \) that have a character smaller than \( c \). These data structures are enough to compute the LF-mapping (also known as the backward step operation), which maps the lexicographic rank of a suffix \( T[i..|T|] \) to the lexicographic rank of suffix \( T[(i-1)..|T|] \). In the special case where the LF-mapping is applied to the first suffix \( T[1..|T|] \), the LF-mapping returns the lexicographic rank of the last suffix \( T[|T|..|T|] \). The formula is \( LF(i) = C[BWT[i]] + rank_{BWT}(i, BWT[i]) \). The LF-mapping can also be interpreted as a mapping between corresponding characters in the first and the last column in the Burrows-Wheeler matrix (see Figure 2.2). The \( C \)-array can be seen as a compact way of representing the \( F \) column in Figure 2.2.

The substring search algorithm of the FM-index reads characters of the query string \( Q \) right-to-left, while maintaining the lexicographic range of the suffix of \( Q \) that has been processed so far. Suppose we have processed the suffix \( \alpha \) of \( Q \), the current lexicographic range is \([l..r]\), and the next character to be processed is \( c \). The updated lexicographic range after processing \( c \) is \([LF(i)..LF(j)]\), where \( BWT[i] \) and \( BWT[j] \) are the first and the last occurrences respectively of character \( c \) in \( BWT[i..j] \). If \( c \) does not occur in \( BWT[i..j] \), then \( c\alpha \) does not occur in \( T \). This update is called the left-extension operation.
2.5 The bidirectional Burrows-Wheeler index

The bidirectional Burrows-Wheeler index (BiBWT index for short) is a recent [10, 40], more expressive variant of the FM-index. It is defined as follows. Suppose $T$ is a string terminated with the symbol $\$\$ smaller than all other character in $T$. The BiBWT index of $T$ consists of the $BWT$ of $T$, the $\overline{BWT}$ of the reverse $\overline{T}$, and the $C$-array of the standard FM-index. We denote the $BWT$ of $\overline{T}$ with $\overline{BWT}$.

The $BWT$ and $\overline{BWT}$ are symmetric in the following way: $BWT[i]$ is the character that precedes the suffix of $T$ with lexicographic rank $i$ among the suffixes of $T$, while $\overline{BWT}[i]$ is the character that follows the prefix of $T$ with colexicographic rank $i$ among the prefixes of $T$, where the $\$\$ is now at the start of $T$ instead of the end. The colexicographic order is defined just like the standard lexicographic order, but characters are compared from right to left instead of left to right. The range of suffixes of $T$ in the lexicographic order that are prefixed by $\alpha$ is called the lexicographic range of $\alpha$, and the range of prefixes of $T$ (where the $\$\$ is at the start of $T$) that are suffixed by $\alpha$ is called the colexicographic range of $\alpha$. While the FM-index only supports a left-extension, the BiBWT supports both left- and right extensions. The right-extension is implemented as a left-extension on $\overline{BWT}$ from the colexicographic range.

The substring search on the BiBWT maintains both a lexicographic and a colexicographic interval. This can be done efficiently using the same data structures required for substring search in FM-index (see [10] for details). If so far we have searched the string $\alpha$ and the corresponding interval pair is $([\ell, r], [\ell', r'])$, then the characters to the left of $\alpha$ in $T$ can be read from $BWT[\ell..r]$ and the characters to the right of $\alpha$ in $T$ can be read from $\overline{BWT}[\ell'..r']$. This information lets us know whether $\alpha$ is left-maximal or right-maximal (or both, or neither).

One very useful property of the BiBWT index is that it can be used to iterate the interval pairs of all right-maximal repeats of $T$. These correspond to all internal nodes of the suffix tree of $T$. Many algorithms that work on the suffix tree only need to visit the lexicographic intervals of all internal nodes of the suffix tree in any order. The BiBWT index can run any such algorithm. The right-maximal repeats are iterated by starting from the empty string and recursively taking all left-extensions leading to right-maximal substrings. This will give the intervals of all right-maximal strings, because when a left-extension becomes non-right-maximal, all the subsequent left-extensions from there must also be non-right-maximal. Symmetrically of course, it is possible to iterate all left-maximal repeats.
Figure 2.2: The block of characters on the left is the *Burrows-Wheeler matrix* of $T = ABBACABACBCCB\$. The rows of the matrix are the rotations $T$ sorted in lexicographic order. The leftmost column is called the F-column and the rightmost column is called the L-column, or the BWT. The picture on the right shows the LF-mapping, i.e. how the characters in the L-column correspond to characters in the F-column.

One important application of the right-maximal substring enumeration algorithm is that it can be used to build the balanced parentheses representation of the suffix tree of $T$ in linear time [8]. This is done by counting how many of the enumerated intervals start and end at a given position $i$, and then iterating $i = 1, \ldots, |T|$, appending an open parenthesis for every interval starting point, and a closing parenthesis for every interval ending point.
Chapter 3

Greedy Superstring Approximation in Compact Space

The shortest common superstring problem is a classical problem in string processing. It is defined as follows. Let $S = S_1, \ldots, S_m$ be a set of strings from an alphabet $\Sigma$ of size $\sigma$. The problem is to find the shortest string that contains every string in $S$ as a substring. If there are multiple solutions, it is enough to return one.

The problem was shown NP-hard by Maier and Storer [47] in 1977. However a simple greedy algorithm works well in practice. The basic greedy heuristic is based on merging strings with a large suffix-prefix overlap. The suffix-prefix overlap of an ordered pair of strings $(S_1, S_2)$ is defined to be the longest string $\alpha$ that is both a suffix of $S_1$ and a prefix of $S_2$. If no such overlap exists, $\alpha$ is the empty string. Now we can write $S_1 = \beta\alpha$ and $S_2 = \alpha\gamma$ and the merge of $S_1$ and $S_2$ is defined as $\text{Merge}(S_1, S_2) = \beta\alpha\gamma$. String $S_1$ is called the left side of the merge and $S_2$ is called the right side of the merge.

The greedy algorithm finds the pair of distinct strings $(S_i, S_j) \in S \times S$ with the longest suffix-prefix overlap and assigns $S := (S \setminus \{S_i, S_j\}) \cup \text{Merge}(S_i, S_j)$. If there are multiple pairs with the same overlap length, then an arbitrary choice is made among them. This is repeated until $S$ contains only one string, which is now a superstring of all strings in the original $S$.

To analyze the greedy algorithm, it is useful to think about the overlap graph of $S$. The overlap graph of $S$ is a weighted directed graph $G = (V, E)$ where $V = S$ and $E = \{(S_i, S_j) \mid i \neq j \wedge (S_i, S_j) \in S \times S\}$ such that the weight of edge $(S_i, S_j)$ is the length of the longest suffix-prefix overlap from $S_i$ to $S_j$. In terms of the overlap graph, the greedy algorithm works by building a subset of edges
Greedy Superstring Approximation in Compact Space

\(E' \subseteq E\), starting from the empty set and considering edges in \(E\) in decreasing order of weight, adding edge \(e = (S_i, S_j)\) if it does not create a cycle or a branch in \(E'\). Since the overlap graph is a complete graph, the set \(E'\) always forms a Hamiltonian path at the end of the algorithm. A superstring is constructed by merging the strings in the order of the Hamiltonian path.

The longer the Hamiltonian path, the shorter the resulting superstring, and if also \(S\) is factor-free, then any Hamiltonian path of maximum length gives a superstring of shortest possible length. A set of strings is said to be factor-free if no string in the set is a substring of another. We can always preprocess \(S\) to be factor-free by deleting duplicates, and strings \(S_i \in S\) that are a proper substring of another string \(S_j \in S\), because including \(S_j\) in the superstring will also automatically include \(S_i\).

To our knowledge, the first mention of the greedy algorithm is found in Gallant’s 1982 PhD thesis [30]. He thought that the algorithm would always produce a superstring of length at most 1.5 times of an optimal superstring. In 1989 Turner [71] showed this to be false by giving an input where the approximation ratio is close to 2. Around the same time, Ukkonen and Tarhio [69] also analyzed the algorithm and proposed the greedy conjecture, which is that the length of the superstring produced by the greedy algorithm is always at most two times the length of an optimal superstring. An example of an adversarial input where the greedy heuristic performs badly is the set of three strings \(\{ab^h, b^{h+1}, b^ha\}\) for some parameter \(h\), where the shortest superstring is \(ab^{h+1}a\). The greedy heuristic may produce \(ab^hab^{h+1}\), and therefore the approximation ratio is \((2h + 3)/(h + 3)\), which approaches 2 in the limit as \(h\) goes to infinity. The open problem is to decide whether the bound of 2 is tight.

In 1991 Blum et al. [12] showed that the greedy algorithm achieves an approximation ratio of at least 4 in the worst case, and in 2005 Kaplan and Shafir [37] improved this to 3.5. The result of Kaplan and Shafir is still the best that has been found, and thus the conjecture of Ukkonen and Tarhio still remains open after over 30 years of research on superstrings. More complicated approximation algorithms have been proposed that have been proven to achieve approximation bounds better than 3.5. Over the years, this has slowly been improved, and the current best bound is \(2 + 11/30\) by Paluch in 2014 [57]. We have compiled a timeline of approximation bound results in Figure 3.1.

The basic greedy superstring algorithm still works remarkably well in practice. Experiments of Romero et al. [62] show that for variants of the greedy algorithm, the approximation is within a few percent of the optimum for strings sampled from real DNA data. Ma [46] has offered an explanation for the good practical
performance using smoothed analysis, where the input is assumed to have a small amount of random noise. In such analysis, the average approximation ratio was proven to be $1 + o(1)$.

Efficient implementations for the greedy algorithm are therefore still relevant today. Since the algorithm is so simple, it can be implemented very efficiently. Already in 1990 Ukkonen [72] showed how to implement the algorithm in $\Theta(n \log(\min(\sigma, |S|)))$ time and $\Theta(n \log n)$ bits of space, where $\sigma$ is the size of the alphabet and $n = \sum_{S \in \mathcal{S}} |S|$. The algorithm relies on the Aho-Corasick data structure (see Chapter 2) and linear length lists of integers, hence the $\Theta(n \log n)$ bit space complexity. This was considered optimal at the time, because this was before the FM-index data structure was discovered. Back then, space in string algorithms was generally counted in terms of the number of integers needed to store in RAM, which is the optimal $O(n)$ in Ukkonen’s solution.

However, for large $n$ and small alphabets like the DNA alphabet, the $O(n \log n)$ bit space complexity can be a factor 15 or more larger than the size of the input data, which is $|\log \sigma|$ bits per character when characters are encoded with simple fixed-length codewords. Our contribution is to present the first algorithm that implements the greedy heuristic in $O(n \log \sigma)$ bits of space and time. We achieve this result by adapting Ukkonen’s approach to use only the FM index and the balanced parentheses representation of the suffix tree topology.

### 3.1 Ukkonen’s linear time superstring algorithm

As described in the previous section, the greedy algorithm selects edges of the overlap graph $G = (V, E)$ in decreasing order of weight such that introducing new edges does not introduce cycles or branches. Implementing this heuristic efficiently is a non-trivial task. Edge weights of $G$ correspond to lengths of suffix-prefix overlaps, which can be computed efficiently by utilizing the failure links of the Aho-Corasick structure.

As a preprocessing step, Ukkonen’s superstring algorithm removes from $\mathcal{S}$ all duplicates and all strings that are a proper substring of another string in $\mathcal{S}$, and builds the Aho-Corasick automaton of the reduced $\mathcal{S}$. Then it picks suffix-prefix overlaps in decreasing order of length by iterating the states of the automaton in reversed breadth-first order from the root. Each state $v$ corresponds to a prefix $\alpha$ of a string in $\mathcal{S}$, and the strings that are suffixed by $\alpha$ correspond to states that can be reached from $v$ by following the failure links in the reverse direction. Ukkonen maintains information to quickly access the strings that are suffixed by
Figure 3.1: A timeline of approximation bounds for the shortest common superstring problem by year of publication. The blue line shows the two proven bounds for classical simple greedy algorithm: 4.0 by Blum et al. in 1991 [12, 13] and 3.5 by Kaplan and Shafir in 2005 [37]. The orange line shows best proven bounds for any type of superstring algorithm. These bounds are 3.0 by Blum et al. in 1991 [12, 13], $2 + \frac{8}{9}$ by Teng and Yao in 1993 [70], $2 + \frac{5}{6}$ by Czumaj et al. in 1994 [20, 21], $2 + \frac{50}{63}$ by Kosaraju et al. in 1994 [39], $2 + \frac{4}{3}$ and then $2 + \frac{2}{3}$ by Armen and Stein in 1995 and 1996 [3, 4], $2 + \frac{25}{42}$ by Breslauer et al. in 1997 [15], $2 + \frac{1}{2}$ Sweedyk in 2000 [68], $2 + \frac{11}{23}$ Mucha in 2013 [51] and $2 + \frac{11}{30}$ by Paluch in 2014 [57].
α, given ν. This gives access to all suffix-prefix overlaps in decreasing order of length.

Ukkonen also maintains some information to know which edges of the overlap graph are still allowed to be added to set of edges \( E' \) we are constructing. An overlap \((S_i, S_j)\) is forbidden in \((V, E')\) if any of the following hold:

(i) \( S_i \) has out-degree 1 in \( E' \),

(ii) \( S_j \) has in-degree 1 in \( E' \),

(iii) there is a path from \( S_j \) to \( S_i \) in \( E' \).

The first two conditions prevent the formation of branches in \( E' \), and the last condition prevents the formation of cycles in \( E' \). Information to keep track of the conditions can be easily maintained during the course of the algorithm using some auxiliary arrays. The details are found in Ukkonen’s paper [72].

### 3.2 Superstrings in compact working space

In this section we outline our approach to implement Ukkonen’s greedy linear time superstring algorithm in \( O(n \log \sigma) \) space. A key insight is that while we cannot afford to store one integer per character, we can afford to store one integer per string, assuming the strings are distinct. This is based on the following lemma.

**Lemma 3.1.** Let there be \( m \) distinct non-empty strings with combined length \( n \) from an alphabet of size \( \sigma > 1 \). Then \( m \log n \in O(n \log \sigma) \). \(^1\)

*Proof.* If \( m \leq \sqrt{n} \), the result is trivial, so let us assume \( m \geq \sqrt{n} \). To avoid rounding, let us also assume that \( m \) is a power of \( \sigma \). The number of possible distinct strings of length up to \( \log \sigma m - 2 \) characters is \( \sigma + \ldots + \sigma^{\log \sigma m - 2} \), which is less than \( \sigma^{\log \sigma m - 1} \leq \frac{1}{2}m \). Therefore at least \( \frac{1}{2}m \) of the strings have length at least \( \log \sigma m - 1 \), and so \( \frac{1}{2}m(\log \sigma m - 1) \leq n \). Since \( m \leq n \), we have \( m \log \sigma m \leq 3n \). Now by the assumption \( m \geq \sqrt{n} \), we have \( \log \sigma m \geq \frac{1}{2} \log \sigma n \), which takes us to \( m \log \sigma n \leq 6n \). Applying the logarithm base change formula \( \log_c n = \log_c n / \log_c \sigma \) to any constant base \( c \) gives \( m \log_c n \leq 6n \log_c \sigma \). \( \square \)

---

\(^1\)There is a mistake in the statement of the Lemma in Paper I. In the paper, the Lemma says that the length \( n \) is measured in bits, where in fact it is measured in characters. The statement in this thesis is correct, and the proof has been rewritten for this thesis to be more simple and readable.
This lemma applies to our algorithm because we can delete duplicates as a preprocessing step. We also remove all strings that are substrings of other strings in $S$. After the preprocessing we concatenate the remaining strings in lexicographic order into $T = S_1 S_2 S_3 \cdots S_m$ where $\$ is a character not found in $\Sigma$, smaller than all characters in $\Sigma$. We build the BWT of $T$, indexed for rank and select queries, and build the balanced parentheses representation of the suffix tree topology of $T$, indexed for suffix link queries. This can be done in $O(n \log \sigma)$ time and space [8]. Next, we pick edges of the overlap graph while avoiding conditions (i), (ii) and (iii) described in the previous section.

For each string, we create an iterator that iterates the suffixes of the strings that occur at least twice in $T$, in decreasing order of length. An iterator for string $S_i$ is a quadruple $(i, \ell, r, d)$, where $[\ell, r]$ is the lexicographic range of the current suffix $\alpha$ of $S_i$, and $d$ is the length of $\alpha$. We use the BWT to initialize the iterators to the lexicographic ranges of the longest right-maximal suffixes of the strings. The iterators are put into a priority queue such that the keys are the $d$ values.

The main loop of the algorithm pops an iterator $(i, \ell, r, d)$ with the highest $d$ value from the priority queue and tries to merge $S_i$ to a string to the right using an overlap of length $d$, while avoiding conditions (i), (ii) and (iii). If successful, the merge is added as an edge to $E'$, and the iterator is removed from the priority queue. Otherwise, we take a suffix link from $[\ell, r]$ using the tree topology machinery we have, decrement $d$, and push the iterator back to the priority queue.

Since the iterators are removed from the priority queue immediately after successful merge, all merges we consider during the course of the algorithm will always avoid condition (i). It remains to show how to maintain information to find merges that avoid conditions (ii) and (iii).

To keep track of condition (ii), we use a bit vector $\text{rightavailable}$ of length $m$ to mark which strings have been used at the right side of a merge. The bit vector is initialized to ones, meaning that all strings are available initially. To keep track of condition (iii), we have two integer vectors $\text{rightend}$ and $\text{leftend}$ of length $m$ such that $\text{rightend}[i]$ is the node reached by following edges in $E'$ starting from node $i$, to the end of the path and correspondingly, $\text{leftend}[i]$ is the node reached by following edges in $E'$ backwards from node $i$ to the start of the path. These are well defined because every node has indegree and outdegree at most 1 in $E'$, and there are no cycles. These vectors and the iterators fit into $O(n \log \sigma)$ space by Lemma 3.1.

Suffixes that are not right-maximal in $T$ do not need to be considered because they are all followed by $\$ and hence cannot be prefixes of others strings, because in that case they would be a duplicate or a substring of another string, but those were removed in preprocessing.
3.3 Results and discussion

Now we will describe how to find and select available merges. Suppose we are considering iterator \((i, \ell, r, d)\) currently representing suffix \(\alpha\). We will first find all right-available substrings that have \(\alpha\) as a prefix. These strings start at suffixes of \(T\) that are preceded by dollars, or in other terms, they are the suffixes of \(T\) at lexicographic positions \(j \in [\ell, r]\) such that \(BWT[j] = \$\). Since the strings are concatenated in lexicographic order in \(T\), the \(k\)-th dollar in the BWT is the dollar preceding the \(k\)-th string \(S_k\) in the concatenation. This means we can use rank queries on the dollars of the BWT to find the contiguous range of strings \(S_{\ell'}, \ldots, S_{r'}\) that are prefixed by \(\alpha\).

Now the problem reduces to finding positions \(j\) in \(\text{rightavailable}[\ell'...r']\) such that \(\text{rightavailable}[j] = 1\), and checking condition (iii) for each such position. If we find a position \(j\) that satisfies condition (iii), we add \((S_i, S_j)\) to \(E'\), delete the iterator of \(S_i\) and proceed with the main loop by popping the next iterator. As Ukkonen observed in 1990 [72], it is enough check condition (iii) for only two positions \(j\) with \(\text{rightavailable}[j] = 1\), because if the first position we try satisfies condition (iii), the second one we try is guaranteed not to. However, we still need to find the two positions \(j\) quickly. This can be done in \(O(\log n)\) time by using dynamic rank and select data structures on \(\text{rightavailable}\). Since we do only \(m - 1\) merges in the course of the whole algorithm, the total time consumed in finding one-bits in \(\text{rightavailable}\) is \(O(m \log n)\), which is in \(O(n \log \sigma)\) by Lemma 3.1.

See for Figure 3.2 for more details and example of the merge finding subroutine and Paper I for the full details of the whole algorithm. Paper I also shows how to improve the \(O(\log n)\) time to find a one-bit in an interval in \(\text{rightavailable}\) to \(O(1)\) using a union-find data structure for runs of consecutive zeros.

### 3.3 Results and discussion

We implemented a prototype of our algorithm in C++ on top of the SDSL-library [32]. A compressed suffix tree (CST) from the SDSL library was used to implement the suffix links and the FM-index. We modified the code so that only the parts of the CST that we require were built, namely the FM-index, the balanced parentheses support and a bit vector indicating the leftmost child of each node.

For preprocessing, we sort the strings, which allows also allows us to detect and remove duplicate strings quickly and put us into the conditions of Lemma
Greedy Superstring Approximation in Compact Space

Figure 3.2: Pictured on the left is the overlap graph of the lexicographically sorted set $S = \{\text{ACAC}, \text{ACGG}, \text{ACTG}, \text{CCAC}, \text{TAAC}, \text{TGCC}\}$. The weights of the edges (i.e. the lengths of the overlaps) are encoded in the colors. The blue edges have weight 1, the orange edges have weight 2, and the edges with weight 0 are not drawn. On the right is the BWT of the dollar-separated concatenation $S$, and below that are the three arrays for the bookkeeping information $E'$. Suppose we are running our superstring algorithm, the bold edges have been chosen so far, and we are considering suffix AC of string CCAC, i.e. string with index 3. The box on the BWT marks the lexicographic interval of AC. We see that the first, second and third dollars in the BWT fall inside the box. This means that strings with indices 0, 1 and 2 in $S$ are prefixed by AC. The interval 0..2 in the array rightavailable then tells us that strings 0 and 2 are available for a merge because they are marked with one-bits. We cannot choose string 2, because an edge from string 3 to string 2 would cause a cycle 3-2-5. We can see this from the fact that rightend[2] = 3. However, choosing the edge from 3 to 0 does not run into such a problem. In general we also need to check that we do not merge a string to itself. In our example, the algorithm will select the edge from 3 to 0, and do the following updates: rightavailable[0] := 0, leftend[rightend[0]] := leftend[3] and rightend[leftend[3]] := rightend[0]. The information in leftend and rightend is allowed to go out of date for nodes in the middles of paths, but the information for nodes at the ends of paths in always kept up to date, which is all the information we need about the paths.
3.3 Results and discussion

3.1. We use the CST to detect and flag strings that are proper substrings of other strings in $S$. These will be treated as deleted in the algorithm.

We ran some preliminary experiments on DNA reads from the human gut microbial gene catalogue project [58]. The alphabet of the dataset is \{A, C, G, T, N\}. After index construction, the peak memory usage is roughly 4.5 bytes per character in the input. The peak at index construction is almost 4 times as large, but it is due to the SDSL library using a pointer-based construction algorithm. More efficient construction algorithms like the ones described by Belazzougui [8] could be plugged in to bring the construction space down. We estimate that an implementation of Ukkonen’s Aho-Corasick based algorithm would need at the very least $4n \log n$ bytes, which is already at least twice that of ours in our largest test set.

The time usage was roughly 4 seconds per megabyte for index construction and 0.7 seconds per megabyte for overlap enumeration and substring construction.

One potential application for our space-efficient superstring algorithm is compression of DNA sequence reads. DNA reads have many overlaps naturally, so we could try to compress a read set by computing a superstring and encoding the start and end points of the reads using bit vectors.

Future work could also include benchmarking different superstring algorithms against our algorithm. At the very least, it would be interesting to see how a well-engineered implementation our algorithm matches against a similarly carefully crafted implementation of Ukkonen’s superstring algorithm.
Chapter 4

Data Structures for Large Variable Order Markov Models

Markov models are a basic tool in biological sequence analysis. They model sequences as random processes following simple rules where the key assumption is that the process has limited memory, i.e. the distribution of the next symbol in the sequence depends only on a few of the previous symbols. This is called the Markov property. The previous symbols that determine the distribution of the next symbol are called the current state or the current context of the process. The model is composed of a list of transition probabilities between all possible states. The transition probabilities are usually learned automatically from training data.

In fixed order models, the state of the chain is the last $k$ characters of the current sequence, for some constant $k$. The parameter $k$ is called the order of the model. If the alphabet of the sequence is $\Sigma$, then the set of states is $\Sigma^k$, i.e. the set of all substrings of length $k$, and the transition function can be described with a function of the form $f : \Sigma^k \times \Sigma \rightarrow \mathbb{R}$ such that $f(\alpha, c)$ gives the probability that context $\alpha$ is followed by symbol $c$. When a symbol $c$ is appended, the state of the chain is changed by dropping the first character of the old state and appending the new character $c$.

The size of a fixed order model is exponential in $k$ because storing the value of $f(\alpha, c)$ for all $\alpha \in \Sigma^k$ and $c \in \Sigma$ means storing up to $|\Sigma|^{k+1}$ distinct probabilities. However, it is often the case that for some contexts, only the last $l$ characters for some $l < k$ affect the probability, and the rest are irrelevant, or in other words, changing any of the first $l$ characters does not affect the probability distribution from the context. In this case the effective order of the model at this context is
and the model could be stored more efficiently as a \textit{variable order Markov model} (VOMM).

A VOMM is defined by choosing a set of contexts $C_1, \ldots, C_m$ where the lengths of the contexts can vary, and assigning a probability distribution over $\Sigma$ for each context. This makes it possible to include long contexts in the model without having the exponential blowup in memory. Now if we are given a query string $Q$ and are asked to predict the next character, we could, for example, take the longest context that is a suffix of $Q$ and look up the probability distribution for that context. Another approach is to take a mixture of the distributions of all contexts that are a suffix of $Q$. A model that predicts characters like this is called an interpolated Markov Model (IMM).

To find all contexts that match a suffix of a query, the contexts have to be stored in a suitable data structure. One option is to store the contexts in a trie\footnote{A trie for a set of strings is an edge-labeled tree, where root-to-leaf paths correspond to the input strings. See also Section 2.3.} containing the reverse strings of all contexts. This way we can find all contexts that are a suffix of $Q$ by taking characters from the end of $Q$ one by one, and walking down the trie with those characters as far as possible. The nodes of the trie store the probability distributions associated with the contexts.

The trie-solution is however far from optimal. In 1996 Ron et al. \cite{63} improved on the idea by introducing the concept of the probabilistic suffix tree (PST)\footnote{Similar ideas had been presented as early as 1983 by Rissanen \cite{60}, but in a slightly different form.}. Unlike the name suggests, the data structure is completely deterministic. The purpose of the tree is to learn and represent a VOMM such that the probabilities are easily accessible when using the model for prediction. The PST is essentially a suffix tree (see Chapter 2.3) of the training data, where nodes corresponding to contexts are marked and assigned a probability distribution over the alphabet.

The original algorithm to construct the PST by Ron et al. required $\Omega(LN^2)$ time in the worst case, where $L$ is the length of the longest context and $N$ is the length of the training data. This was improved first to $O(LN \log N)$ by Mazeroff et al. \cite{50} and then to $O(N)$ by Apostolico and Bejerano \cite{2}. Apostolico and Bejerano also give an algorithm to compute the probability of every individual character of a query string $Q$ against the model in $O(|Q|)$ time, whereas previously this took $O(|Q|L)$ time. The size of the alphabet is assumed to be constant in all these results.

Later, Leonardi \cite{42} proposed a sparse variant of the PST where contexts can be grouped into equivalence classes, and showed that classification accuracy
could be improved this way. Even more recently, Lin et al. [45] proposed a variant of the PST that is based on a suffix array instead of a suffix tree, which is smaller and takes less time to build. In 2015, Shagrehi et al. [65, 66] designed a small PST variant for natural language processing applications based on a compressed suffix tree. This takes only $O(N \log \sigma)$ bits of space, where $\sigma$ is the size of the alphabet, whereas normal pointer-based suffix trees take $\Theta(N \log N)$ bits. However, their data structure does not support the linear whole-string prediction algorithm of Apostolico and Bejerano. Independently at the same time, Cunial and Belazzougui designed an $O(N \log \sigma)$ space framework for string kernels, including support for simulating a PST [9]. Their framework supports a variety of context selection criteria and $O(|Q| \log \sigma)$ time whole-string prediction. However no implementation was given.

Our work continues on this line of research. We rely on tools from string processing and compact data structures literature to implement a PST in $O(m \log \frac{N}{m} + r(\log \sigma + \log \frac{N}{r}))$ bits, where $m$ is the number of maximal repeats and distinct left-extensions thereof in the training data ($m \leq N$), $r$ is the number of runs of the Burrows-Wheeler transform of the training data ($r \leq N$) and $\sigma$ is the size of the alphabet, such that we can support whole-string prediction in $O(|Q|(\log \frac{N}{\min(m,r)} + \log \sigma))$ time. A caveat is that the index works only for context selection criteria which select only maximal repeats or left extensions of maximal repeats. However we argue that most reasonable context selection criteria will have this property. If this property does not hold, we have to settle for $O(N \log \sigma)$ bits of space, but then on the positive side, we can do faster prediction in $O(|Q| \log \sigma)$ time.

4.1 Learning VOMMs from data

There are two key choices in training a VOMM: How to select contexts, and how to estimate the probability distributions from the contexts. Let us consider both of these questions in turn.

4.1.1 Selecting contexts

Lin et al. [45] simply select all substrings of the training data as contexts. It might be worthwhile to try to pick contexts more carefully to increase the signal-to-noise ratio of the model. Here we present a sample of context selection criteria found in the literature.
One principle is to select strings of the form $c\alpha$ where $c \in \Sigma$ and $\alpha \in \Sigma^*$ such that the empirical distributions of characters to the right of $c\alpha$ and $\alpha$ are significantly different in the training data. This is a signal that the distribution to the right of $c\alpha$ carries important information. The similarity of two distributions can be measured with the Kullback-Leibler divergence\textsuperscript{3} between the distributions. This leads to the following context selection criterion [16, 17, 73]:

- Denote the distribution of characters to the right of $\alpha$ with $D(\alpha)$ and denote the Kullback-Leibler divergence between distributions $d_1$ and $d_2$ with $KL(d_1, d_2)$. Fix a threshold $\tau > 0$. Contexts are all strings of the form $c\alpha$ such that:

$$KL(D(c\alpha), D(\alpha)) \geq \tau$$

(4.1)

Another variant of this is to measure the difference between two distributions using a $p$-norm [17]

- Denote the distribution of characters to the right of $\alpha$ with $D(\alpha)$. Fix a threshold $\tau > 0$ and a number $p$. Contexts are all strings of the form $c\alpha$ such that:

$$||D(\alpha) - D(c\alpha)||_p \geq \tau$$

(4.2)

A third variant is based on the concept of empirical entropy [60, 73, 74].

- Denote the number of occurrences of string $\alpha$ in the training data with $f(\alpha)$ and denote the empirical entropy of distribution $d$ with $H(d)$. Fix a threshold $\tau > 0$. Contexts are all strings of the form $\alpha$ such that:

$$f(\alpha)H(\alpha) - \sum_{c \in \Sigma} f(c\alpha)H(c\alpha) \geq \tau$$

(4.3)

Simpler criteria based only on fractions involving substring counts and lengths have also been used successfully (see Bejerano and Yona [7] and references therein). For example:

- Denote the number of occurrences of string $\alpha$ in the training data with $f(\alpha)$. Fix thresholds $\tau_1 > 0$, $\tau_2 > 0$, $\tau_3 < 1$ and $\tau_4 > 1$. Contexts are all strings of the form $c\alpha$ such that there exists $b \in \Sigma$ such that all of the following hold:

\textsuperscript{3}If $P$ and $Q$ are both distributions defined over a finite domain $X$, then the Kullback-Leibler divergence from $Q$ to $P$ is defined as $KL(P, Q) = \sum_{x \in X} P(x) \log(P(x)/Q(x))$. 
4.1 Learning VOMMs from data

\[
\frac{f(c\alpha)}{|T| - |c\alpha| + 1} \geq \tau_1
\]  

(4.4)

\[
f(c\alpha b) / f(c\alpha) \geq \tau_2
\]  

(4.5)

\[
\frac{f(c\alpha b) / f(c\alpha)}{f(ab) / f(\alpha)} \in (0, \tau_3] \cup [\tau_4, \infty)
\]  

(4.6)

It can be easily checked that all the four criteria surveyed here have the property that \( \alpha \) is a maximal repeat. A maximal repeat is a substring of the training data that can be extended in at least two ways both to the left and to the right (see Section 2.1).

One of the main reasons to prefer a VOMM over a fixed order model is that the contexts can freely expand or shrink to their "best version", and this best version is usually in close vicinity to a maximal repeat, because this is where we start to lose or gain new occurrences. We formalize this intuition by assuming that contexts are always maximal repeats, or one-character left extensions of maximal repeats.

4.1.2 Estimating probability distributions for contexts

The second part of training a VOMM is to learn probability distributions for the characters that follow each context. A good starting point is to use the empirical distributions in the data. Let \( f(\alpha) \) be the number of occurrences of substring \( \alpha \) in the training data. We can estimate the probability that context \( \alpha \) is followed by character \( c \) with the empirical probability:

\[
Pr(c|\alpha) = \frac{f(\alpha c)}{f(\alpha)}
\]  

(4.7)

However there is a well-known problem with this, called the zero-frequency problem or the sparse data problem. That is, if an event has never occurred, how should we estimate its probability? Formula 4.7 assigns a probability of exactly zero for such an event, but this ignores prior knowledge that in most applications, almost no event is completely impossible. A more likely explanation for the absence of an event is that our sample size was too small to observe all outcomes. The smaller our sample size is for a context, the less we should trust the empirical probabilities. One simple and practical modification to Formula 4.7 to address
this is to add pseudocounts, i.e. add a positive number \( s \) to all counts and thus define the probabilities as:

\[
Pr(c|\alpha) = \frac{f(\alpha c) + s}{f(\alpha) + s}\sigma
\]

(4.8)

This is also called additive smoothing or Laplace smoothing. This corresponds to having a Dirichlet prior on the counts (see e.g. Chapter 11.5 of Durbin et al. [23] for the mathematical details).

More sophisticated ways of smoothing the probabilities exist, like the Good-Turing estimator [33] developed for cracking German ciphers during World War II, and Kneser-Ney smoothing [65, 66, 54] used in natural language processing. In interpolated Markov models, the zero frequency problem is not as serious because the prediction is a smooth mixture of estimates from all contexts that match the end of the current sequence.

In our work, we assume that the probability distributions are estimated with formulae such as 4.7 and 4.8 that depend mainly on the counts of \( \alpha \) and \( \alpha c \) in the training data, as these counts are easily accessible from the PST.

### 4.2 The framework of Belazzougui and Cunial

In this section we outline the whole-string scoring algorithm of Belazzougui and Cunial [9]. The algorithm is described on a high level in terms of the following basic operations:

- **RightExtend**: Takes a string \( \alpha \) and a character \( c \) and returns string \( \alpha c \).
- **Count**: Takes a string \( \alpha \) and returns the number of occurrences of \( \alpha \).
- **LeftContract**: Takes a string \( \alpha \) and returns the longest proper suffix of \( \alpha \) that is left-maximal. A suffix is considered proper if it is not equal to \( \alpha \).
- **GetContext**: Takes a string \( \alpha \) and returns the longest suffix of \( \alpha \) that is a context.

The pseudocode of the whole-string scoring algorithm is in Algorithm 1. The algorithm prints the probability of each position in the query. For concreteness, probabilities are estimated using the raw empirical probabilities from the longest matching context (line 5). Other probability estimation methods could be plugged in at line 5. The loop invariant is that at the start of iteration \( i \) of
the for-loop, \( \alpha \) is the longest substring of \( Q \) that ends at position \( i - 1 \) in \( Q \) and also occurs somewhere in \( T \). The longest valid context ending at position \( i - 1 \) of \( Q \) must therefore be a suffix of \( \alpha \). The code can be easily modified for IMM prediction by retrieving all matching contexts by repeatedly executing \( \beta \leftarrow \text{GetContext}(\text{LeftContract}(\beta)) \) after line 4, evaluating the probabilities from each context, until we arrive at the empty string.

Algorithm 1 High level view of the whole-string prediction algorithm of Belazzougui and Cunial. The string \( Q \) is the input to the algorithm.

1: \( \alpha \leftarrow \text{empty string} \)
2: \textbf{for} \( i \in 1, \ldots, |Q| \) \textbf{do}
3: \( c \leftarrow Q[i] \)
4: \( \beta \leftarrow \text{GetContext}(\alpha) \)
5: \textbf{print} \( \text{Count}(\text{RightExtend}(\beta, c)) / \text{Count}(\beta) \)
6: \textbf{while} \( |\alpha| > 0 \) and \( \text{Count}(\text{RightExtend}(\alpha, c)) = 0 \) \textbf{do}
7: \( \alpha \leftarrow \text{LeftContract}(\alpha) \)
8: \( \alpha \leftarrow \text{RightExtend}(\alpha, c) \)

In the actual implementation these operations do not operate on strings, but rather on their colexicographic intervals and representations of their corresponding nodes in the topology of the suffix tree \( \overline{ST} \) of \( T \). A reader unfamiliar with these concepts is recommended to study Chapter 2 for an overview before proceeding with the rest of this chapter.

The main data structures used by Belazzougui and Cunial [9] are the Burrows-Wheeler transform of \( \overline{T} \), denoted with \( \overline{BWT} \), and the balanced parentheses representation (BPR) of the topology of \( \overline{ST} \). Substrings are handled in two different complementary representations: as colexicographic intervals, and as pairs of parentheses in the BPR. It is possible to convert between the two representations quickly on demand. If we have a colexicographic interval \([\ell..r]\), then the corresponding pair of parentheses in the BPR is the lowest common ancestor of leaves with ranks \( \ell \) and \( r \), and if we have a pair of matching parentheses, the corresponding colexicographic interval is the range between the ranks of the leftmost and rightmost leaf inside the parentheses of the node. These operations can be done in constant time with succinct data structures [52].

Our terminology in this text is such that when we refer to the node of a string \( \alpha \), we mean the shallowest node \( v \) of \( \overline{ST} \) such that \( \overline{\alpha} \) is a prefix of the path label of \( v \). This means that the path label of \( v \) is the reverse string of \( \overline{\alpha} \), where
the notation $\hat{\alpha}$ denotes the left-saturation of $\alpha$, i.e. the shortest left-maximal substring of $T$ that has $\alpha$ as a suffix.

The operations of Algorithm 1 are implemented as follows. The operation RightExtend is just a standard FM-index left extension on $\overline{BWT}$. Operation Count is the length of the colexicographic interval of the string. Operation LeftContract from $\alpha$ is implemented with a parent operation in the $\overline{ST}$ topology from the node of $\alpha$. Operation GetContext essentially boils down to a nearest marked ancestor query on the BPR, where we have marked all nodes that correspond to contexts.

### 4.3 Pruning the topology

Our main theoretical contribution is to show how to implement Algorithm 1 in space proportional only to the number of maximal repeats and BWT runs of the training data. We follow the framework of Belazzougui and Cunial, except that we observe that it is not necessary to store the whole topology of $\overline{ST}$. Assuming contexts are maximal repeats or one character left-extensions thereof, we can get away with only storing the part of the topology that contains maximal repeats and their children, and encoding the leaves with a bit vector. We denote this pruned tree with $\overline{ST}^*$. We build a data structure to support a new operation Expand. The operation takes a colexicographic interval of a string $\alpha$, and returns the colexicographic interval of the longest suffix of $\hat{\alpha}$ that is represented in $\overline{ST}^*$. This is based on a bit vector that marks where the pruning has occurred in $\overline{ST}$. See Figure 4.1 for more details.

The following modifications are now made to Algorithm 1. On line 4, before executing GetContext, we expand the colexicographic interval of $\alpha$. This takes us to the nearest non-pruned ancestor of the node of $\alpha$. Now that our string lies within the topology that we have stored, we can transform the colexicographic range back to the node representation and run GetContext.

The other modification is at line 7. On the first round of the loop, before executing LeftContract, we expand the interval of $\alpha$. This takes us to the interval of the longest suffix of $\alpha$. This is a child of a maximal repeat in $\overline{ST}$. It is guaranteed that none of the nodes between $\alpha$ and the expansion of $\alpha$ could have been right-extended by $c$. This is because of Observation 4.1 below and the fact that the nodes we skip over cannot be maximal repeats because they have been pruned out. Therefore the while-loop would have eventually taken us to
this node in any case, and the correctness of the algorithm is preserved. Now that our string lies within the topology we have stored, we can transform the colexicographic range back to the node representation and proceed as normal in the while-loop.

**Observation 4.1.** Let $\alpha_1$ be the string $\alpha$ at line 6 of Algorithm 1 and let $\alpha_2$ be the string $\alpha$ at line 8 before executing the right extension. If the while-loop of Algorithm 1 executes at least one iteration, then $\alpha_2$ is the empty string or a maximal repeat.

**Proof.** Suppose $\alpha_2$ is not the empty string. Since by assumption the loop was executed at least once, $\alpha_1c$ is not a substring of $T$, so $\alpha_1$ must be followed by some other character $d \neq c$. This means $\alpha_2$ is also followed by $d$ as it is a suffix of $\alpha_1$. When we exit the while-loop, $\alpha_2c$ is a substring of $T$ and so $\alpha_2$ is followed by two distinct characters $c$ and $d$, and therefore $\alpha_2$ is right-maximal. The string $\alpha_2$ is also left-maximal because it corresponds to a node of $ST$.

### 4.4 Construction

We use a bidirectional BWT index of $T$ [10] to enumerate the colexicographic intervals of all left-maximal substrings of $T$, and Belazzougui’s counter-based approach [8] to build the part of the topology excluding the pruned nodes. Next we iterate intervals of maximal repeats and their left extensions using the bidirectional BWT index again, and mark the nodes that are contexts by testing some of the context criteria listed in Equations 4.1-4.6. Finally we index the BPR to support queries for parents and lowest marked ancestors using standard succinct data structures [52], and for string depth queries using a run-length coded version of a suffix link tree trick described in [9]. The string depth queries are needed in the details of GetContext, which we have omitted for brevity, and in some IMM scoring variants.

To compress the index, the pruning bit vector described in Figure 4.1 is encoded as a rank/select index for sparse bit vectors using the method of Okanohara and Sadakane [56], which takes $O(m \log \frac{n}{m})$ bits of space, where $n$ is the length of the vector and $m$ is the number of one-bits in the bit vector. The number of one-bits is linear in the number of nodes in the pruned topology. The BWT is stored as a run-length coded FM-index [48, 49].

To save even more space we can also limit the lengths of contexts to some constant $k$. In this case we only need to store the topology of $ST$ down to depth
Figure 4.1: This figure depicts the topology of the suffix tree of the reverse of aabbaabaabbbbaabbaabaaba$. The edge labels are not drawn to reduce clutter. The leaves are ordered from left to right according to the lexicographic order of their path labels. The green nodes are all maximal repeats or children of maximal repeats. The white nodes will be pruned out. Below the tree is a bit vector marking the leftmost leaf of every green node, and below that a numbering for the leaves. The bit vector is used to implement the Expand operation by expanding the leaf interval of the current node (i.e. the colexicographic interval) to the left to the closest one-bit, and to the right to the last zero-bit before the next one-bit. For example, suppose we are at the node that is the parent of leaves 5 and 6 and want to find the nearest green ancestor. Expanding the interval (5..6) yields the interval (2..7), which is indeed the interval of the nearest green ancestor.
k. Small modifications are then needed in Algorithm 1, which are described in Paper II.

If we keep the bidirectional BWT after construction, we can try different types of context markings easily by reusing the topology data structures we have already built. The only things that need to be rebuilt are the context markings and the lowest marked ancestor data structure.

4.5 Results and discussion

The data structures and algorithms described in the chapter were implemented in C++. We experimented with three levels of compression: no pruning, pruning to left extensions of maximal repeats, and pruning to left extensions of maximal repeats down from a given maximum depth.

We compare our implementation to the those of Lin et al. [45], Bejerano [6], Shareghi et al. [65, 66] and Leonardi [42]. Bejerano and Leonardi prune the index so that the size is dependent on the number of contexts selected. Lin et al. and Shareghi et al. on the other hand store the entire training data in some form and merely mark the contexts in their data structure, similarly to what we do.

Our index with no pruning takes approximately 3 bytes per character of $T$. This can be up to 10 times smaller than Bejerano’s and Leonardi’s when we select a large number of contexts. However if the number of contexts is small, Bejerano’s and Leonardi’s indexes become smaller than ours, as they will be able to prune aggressively. Our unpruned index is between 3 to 4 times smaller than the index of Lin et al., and approximately as small as the index of Shareghi et al (which is to be expected as they use similar techniques to us). Pruning to maximal repeats saves us between 30% and 50% of the space on datasets that are not particularly repetitive, and approximately 90% in highly repetitive collections. With additional pruning by depth, the index can shrink by a further 50%, or more. For detailed space breakdowns and data from the experiments, including construction time and memory, the time required for prediction, variants on different probability estimate functions, experiments on an alternative way to store the lengths of maximal repeats, and more, we refer the reader to Paper II.

Finally, a few words about applying large VOMMs in practice. Very long contexts can be problematic because there are usually not many occurrences of them in the training data, and therefore the probabilities need to be smoothed heavily, or else we risk overfitting the data. If distributions are smoothed heavily, they are not very informative, and so the longest matching context might not
be the best for prediction. We recommend using some kind of an interpolation scheme for large VOMMs.

There are many difficult choices to make when building a VOMM. Which type of context selection criteria should we use and what are the best values for its tunable parameters? How should the probability estimates be smoothed to deal with sparse data? Should the predictions be made from the longest matching context, or from some interpolation of distributions from all matching contexts? In case we want to interpolate, which interpolation method should we choose and how should we tune the parameters of that method? It would be interesting future work to evaluate different models systematically to find out which ones work the best for different types of data. We recommend exploring the space of possible models as much as possible to find the best match for the data at hand.
Chapter 5

Wheeler Graphs and Regular Languages

Wheeler graphs are a new concept that explains many existing succinct data structures. As a first approximation, Wheeler graphs are directed edge-labeled graphs such that the nodes can be sorted by incoming path labels. That is, if node $u$ comes before node $v$ in the sorted order, then path labels to $u$ must be smaller or equal to path labels to $v$, where path labels are compared colexicographically, i.e. right-to-left. The full definition is more tricky however. It is simple on the surface, but disguises a surprising amount of complexity underneath. We will study the full definition in detail in this chapter.

Wheeler graphs are useful because they admit a kind of a generalized Burrows-Wheeler transform which allows succinct storage of the graph with support for path queries. Some examples of data structures that can be seen to operate on a Wheeler graph are the FM-index [27], the XBWT data structure for trees [26], the BOSS data structure for de Bruijn graphs [14], the GCSA data structure for directed acyclic graphs [67], the positional Burrows-Wheeler transform [22] and the wavelet matrix [19].

After Wheeler graphs were introduced by Gagie et al. [29] in 2017, a couple of papers have appeared that rely on the Wheeler graph formalism. Gibney and Thankachan [31] showed that the problem of recognizing whether a graph is Wheeler is NP-complete. They also obtained a couple of approximability and inapproximability results around the problem. Bentley et al. [11] showed NP-hardness of the problem of ordering source nodes of a Wheeler graph to minimize the number of runs in the generalized BWT of the graph. Gagie et al. [28]
used the Wheeler graph framework for compressing sets of DNA sequence reads. Papers III and IV of this thesis add to the growing body of work on this topic.

We start this chapter with a slow introduction to Wheeler graphs aimed at non-experts of the topic. Emphasis is placed on building intuition on the concept. The purpose of this is to equip the reader with the required knowledge and intuition to ease into our two papers on Wheeler graphs. We summarize the results of Paper III at the end of this chapter and the results of Paper IV in the next chapter.

5.1 The Wheeler conditions

We will now introduce the definition of Wheeler graphs as a generalization of the colexicographic order for strings. Recall that the colexicographic order of strings is the same as the lexicographic order, but the characters are compared from right to left\(^1\). For brevity, we shorten the word colexicographic to just *colex*.

Let \( s \) and \( t \) be strings of length \( n \) and \( m \) respectively. If \( s[n] \neq t[m] \), the colex order \( < \) of \( s \) and \( t \) is the same as the order of the last characters, or if the last characters are the same, the colex order is defined recursively to be the same as colex the order of \( s[1..n-1] \) and \( t[1..m-1] \). Written in equations, we have:

\[
\begin{align*}
    s[n] \neq t[m] & \Rightarrow (s < t \iff s[n] < t[m]) . \quad (5.1) \\
    s[n] = t[m] & \Rightarrow (s < t \iff s[1..n-1] < t[1..m-1]) . \quad (5.2)
\end{align*}
\]

The base case of the recursion is the empty string, which is defined to be smaller than all other strings. Now we are ready for the generalization. A graph is Wheeler if the nodes can be ordered by an order \( < \) which satisfies the following three conditions. First, for all pairs of edges \( e = (u, v), e' = (u', v') \), with single-character labels \( \lambda(e) \) and \( \lambda(e') \) respectively, we have:

\[
\lambda(e) \neq \lambda(e') \Rightarrow (v < v' \iff \lambda(e) < \lambda(e')) , \quad (5.3)
\]

This is analogous to Equation 5.1. Further, for all pairs of edges \( e = (u, v), e' = (u', v') \) such that \( u \neq u' \) and \( v \neq v' \), the order needs to satisfy the following condition:

\[
\lambda(e) = \lambda(e') \Rightarrow (v < v' \iff u < u') , \quad (5.4)
\]

\(^1\)The sorting direction is arbitrary but the literature has adopted the convention to use the colexicographic order instead of the lexicographic order for Wheeler graphs. The subpath search queries always run in the opposite direction to the sorting direction.
5.1 The Wheeler conditions

This is analogous to Equation 5.2. Finally, source nodes of the graph must come before non-source nodes of the graph in the order $<$, that is, for any pair of nodes $u, v$, we require:

$$\text{indegree}(v) = 0 \land \text{indegree}(u) \neq 0 \rightarrow v < u.$$  \hfill (5.5)

This is analogous to the recursion base case in colexicographic ordering.

Conditions 5.3, 5.4 and 5.5 are called the Wheeler conditions. We refer to them as conditions $W1$, $W2$ and $W3$ respectively. The conditions can be summarized as follows: nodes are ordered by incoming labels, with ties broken by the origins of the edges, such that source nodes always come before non-source nodes. The definition is a generalization from the colex order on strings because if we model strings $s$ and $t$ as non-branching paths in a graph, then the Wheeler conditions require that the final nodes on the paths are ordered according to the colex order of $s$ and $t$.

The original Wheeler graph paper [29] has a slightly different looking set of conditions, but they are logically equivalent to our conditions. Their formulation replaces Equations 5.3 and 5.4 with the following conditions:

$$\lambda(e) < \lambda(e') \rightarrow v < v',$$  \hfill (5.6)

$$(\lambda(e) = \lambda(e')) \land (u < u') \rightarrow v \leq v',$$  \hfill (5.7)

The definitions can be shown equivalent with a basic case analysis. However our formulation suits the purposes of our exposition better. Gagie et al. also allow multigraphs, i.e. graphs where multiple edges can have the same origin and destination, but we restrict our attention to graphs without edge multiplicities.

In general there can be multiple orders of nodes satisfying the Wheeler conditions, or there might not exist even a single Wheeler order. If there exists a valid Wheeler order, we say that the graph is Wheeler. If the Wheeler order is fixed, the rank of a node in the order is called the Wheeler rank of the node.

A visual intuition for Wheeler graphs is that they are graphs such that if nodes are duplicated and drawn like shown in Figure 5.1, then no edges with the same label cross each other. In other words, if we iterate the nodes in the Wheeler order, and list all edges with an outgoing label $c$, for any fixed choice of $c$, then the destinations of the edges are increasing in the Wheeler order.

An immediate consequence of Equation 5.3 is that all incoming edges to a node must have the same label, because otherwise the node would have to be strictly smaller than itself. A graph where for all nodes, all incoming edges to the node have the same label, is called input consistent. Because all Wheeler
Figure 5.1: Above: a Wheeler graph. The nodes are sorted left to right. Below: the same graph, but all nodes have been duplicated so that the bottom row has only outgoing edges and the top row has only incoming edges. Edges have been colored by label. Wheeler condition $W_1$ says that the destination of all edges with label $a$ must be before the destination of all edges with label $b$. Condition $W_2$ is equivalent to saying that edges with the same label must not cross in this representation. Condition $W_3$ says that the nodes in the upper row with no edge connected to them must come first in the order (only $v_0$ in this case).
5.1 The Wheeler conditions

graphs are input consistent, it is possible to think that the labels of the graph are actually on the nodes, not on the edges, by simply pushing the label of an edge to the destination of the edge. Any nodes with no incoming edges can be thought have a label $\$ \not\in \Sigma$ such that $\$ is smaller than any character in $\Sigma$.

One interesting point of Gagie et al. [29] is that Wheeler graphs are a class of nondeterministic finite state automata (NFA) which can be simulated efficiently. In general to simulate a NFA, we need to keep track of which states are currently active, so simulating an automaton with $n$ states on a string of length $m$ can take $\Omega(nm)$ time. Wheeler graphs on the other hand have a property called path coherence, which is that if we take a contiguous range of nodes in the Wheeler order and follow all edges with some common label $c$, then the resulting range of nodes is also contiguous. This means that the subset of active states in NFA simulation can be represented with two integers representing the start and the end point of the interval.

The main motivation for Wheeler graphs is that they can be indexed for efficient subpath queries. A subpath query on a graph $G = (V, E)$ takes a string $\alpha$ and returns a representation of all nodes $v$ such that there is a subpath to $v$ with label $\alpha$. Recently, strong evidence was found [24] that the problem is hard to solve in general graphs in faster than quadratic time: if an algorithm with running time $O(|E||\alpha|^{1-\varepsilon})$ or $O(|E|^{1-\varepsilon}|\alpha|)$ for any positive $\varepsilon$ existed, then the currently widely accepted strong exponential time hypothesis would be false.

However, in Wheeler graphs it is possible to build a data structure that solves the subpath query problem in $O(|\alpha| \log \sigma)$ time for a query $\alpha$. The data structure takes only $O(n \log \sigma)$ bits of space, but efficient construction algorithms are known only for special cases such as trees and de Bruijn graphs. That is, for some graphs, the data structure might exist, but we might not be able to recognize or construct it in polynomial time.

The subpath search algorithm is based on the path coherence property of Wheeler graphs mentioned above. By path coherence, the set of nodes at the ends of subpaths labeled with $\alpha$ are always a contiguous range in a Wheeler order. Therefore if we know the interval of nodes of a string $\alpha$, then we can find the interval of nodes of $\alpha c$ for any character $c$ by following the first and the last edge labeled with $c$ in the range, and taking all nodes in between the destination nodes of these two edges. Here the edge ordering is such that edges are ordered by the Wheeler order of the origins, with ties broken by destinations. This extension by one character can be repeated any number of times to search for a string of any length, starting from the empty string, whose range is defined to be all nodes
in the graph. Figures 5.2 and 5.3 illustrate the data structure and the search procedure.

5.2 Wheeler graphs and sortability of path label sets

We will now establish some precise statements between Wheeler graphs and sorting by incoming path labels. First we introduce some notation. For a positive integer $k$, we denote by $P_k(v)$ the set of labels of paths that end in $v$ and either start at a source node or have length $k$. If $v$ is a source node, we define $P_k(v) = \{ \varepsilon \}$, for all $k$, where $\varepsilon$ is the empty string.

We write $A < B$ for path label sets $A$ and $B$ iff for all $\alpha \in A$ and $\beta \in B$ we have $\alpha < \beta$. We call a graph strictly sortable if there exists a positive integer $k$ such that for all pairs of nodes $u, v$, exactly one of the following is true $P_k(u) < P_k(v)$ or $P_k(v) < P_k(u)$. Any strictly sortable and input-consistent graph is Wheeler:

**Lemma 5.1.** If an input-consistent graph is strictly sortable, it is Wheeler.

*Proof.* Consider the order $<$ defined by $u < v$ iff $P_k(u) < P_k(v)$ with a large enough $k$ so that the sets $P_k$ are disjoint for all nodes. Such $k$ always exists by strict sortability. We claim that $<$ is a Wheeler order. Due to proof technicalities we select $k$ that is one larger than necessary to make the sets disjoint. Let us now verify the three Wheeler conditions **W1**, **W2** and **W3**.

- **W1**: Take any two edges $(u', u)$ and $(v', v)$ with edge labels $c_u, c_v$ respectively, such that $c_u < c_v$. By input consistency, all path labels to $u$ end with $c_u$ and all path labels to $v$ end with $c_v$, and therefore $P_k(u) < P_k(v)$ and so $u < v$, as required.

- **W2**: Take two edges $(u', u)$ and $(v', v)$ having the same label $c$ such that $u < v$. Take $\alpha c \in P_k(u)$, $\beta c \in P_k(v)$. By strict sortability $\alpha < \beta$, and then also $P_{k-1}(u') < P_{k-1}(v')$ because $\alpha \in P_{k-1}(u')$, $\beta \in P_{k-1}(v')$. Therefore also $P_k(u') < P_k(v')$ and so $u' < v'$, as required.

- **W3**: Suppose $u$ has indegree 0 and $v$ has a positive indegree. Then for all $k$, $P_k(u) = \{ \varepsilon \}$ and $\varepsilon \not\in P_k(v)$, where $\varepsilon$ is the empty string. String $\varepsilon$ is by definition smaller any other string, which means $P_k(u) < P_k(v)$, and so $u < v$, as required.

\qed
5.2 Wheeler graphs and sortability of path label sets

Figure 5.2: **Left:** The de Bruijn graph of the string abcabcbabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabcbaabc
Figure 5.3: **Example of a path query.** This figure continues from Figure 5.2. Suppose we want to search for a path labeled with ab. We search the characters one by one while maintaining an interval of nodes in the Wheeler order. We start from the interval (0..6) containing all nodes. Next we follow the first and the last a-edges (orange) from the upper row to the lower row. This gives us the interval (0..2). Then we follow the first and last b-edges (green) in the node same interval (0..2) from the upper row to the bottom row. Both of these lead to node 3, so our new interval is (3..3), which means that node 3 is the only node with incoming paths labeled by ab. Full data structure details and formulas to make this happen can be found e.g. in [29].
Strict sortability allows us to immediately classify some important graph classes as Wheeler. Any graph that is a single cycle of length $n$ and has at least one edge label that occurs only once is Wheeler because it is strictly sortable with $k = n$. This corresponds to the FM-index of a dollar-terminated input string, where dollar is the unique edge label. Any deterministic trie where edges are directed away from the root is also strictly sortable, because all path labels from the root are unique. This explains the XBWT data structure. A de Bruijn graph of order $d$ is strictly sortable with $k = d$, which can be used to derive the BOSS data structure for de Bruijn graphs. The GCSA data structure for DAGs works by modifying the DAG until it is strictly sortable.

If we do not have strict sortability, the situation becomes murkier, but sortability is still relevant in the following sense. For any two nodes $u$ and $v$, if there is at least one string in $P_k(u) \setminus P_k(v)$, for some $k$, then the order of $u$ and $v$ can be determined by comparing $\alpha$ to any string in $P_k(v)$. This is formalized in the Lemma below.

**Lemma 5.2.** Suppose we have a Wheeler graph. Take any pair of nodes $u, v$ such that for some $k$ we have $P_k(u) \setminus P_k(v) \neq \emptyset$. Take $\alpha \in P_k(u) \setminus P_k(v)$ and $\beta \in P_k(v)$. Now any Wheeler order satisfies $u < v \iff \alpha \prec \beta$.

**Proof.** Let $\gamma$ be the longest string such that we can write $\alpha = \alpha' \gamma$ and $\beta = \beta' \gamma$. Then there exists paths $u_1, u_2, \ldots, u_{|\gamma|}, u$ and $v_1, v_2, \ldots, v_{|\gamma|}, v$ both spelling $\gamma$. It must be that $u_i \neq v_i$ for all $i = 1, \ldots, |\gamma|$, because otherwise $\alpha \in P_k(v)$, contradicting $\alpha \in P_k(u) \setminus P_k(v)$. Nodes $u_1$ and $v_1$ must have different incoming edge labels, because otherwise $\gamma$ was not the longest possible (both nodes cannot be sources because then we would have $\alpha = \beta$ and therefore $\alpha \in P_k(v)$). Therefore by **W1**, the order of $u_1$ and $v_1$ is the same as the order of the last characters of $\alpha'$ and $\beta'$, so we can write that any Wheeler order has $u_1 < v_1$ iff $\alpha' \prec \beta'$. If one of $\alpha'$ or $\beta'$ is the empty string, this still holds by **W3**, and both cannot be empty or else $\alpha = \beta$. Since, $u_i \neq v_i$ for all $i = 1, \ldots, |\gamma|$, we can propagate Wheeler condition **W2** forward through the paths to conclude that $u < v$ iff $\alpha' \prec \beta'$, which is equivalent to $\alpha \prec \beta$. \hfill $\Box$

Lemma 5.2 can be used to infer the Wheeler order of any pair of nodes $u$ and $v$ such that $P_k(u) \neq P_k(v)$ by finding suitable $\alpha$ and $\beta$ for the Lemma, and ordering the nodes according to the colex order of $\alpha$ and $\beta$. In a non-Wheeler graph, the Lemma might imply both $u < v$ and $v < u$ with different choices of $\alpha$ and $\beta$, proving by contradiction that the graph is not Wheeler. The contradiction happens if path label sets of $u$ and $v$ are interleaved colexicographically, that is,
Figure 5.4: Left: a graph that is almost Wheeler. Right: a matrix where the rows correspond to distinct path labels in the graph, and columns to the nodes of the graph. The rectangles show which nodes have which path labels. Lemma 5.2 implies the following partial ordering: $v_1, v_2 < v_3, v_4 < v_5 < v_6 < v_7 < v_8, v_9 < v_{10} < v_{11} < v_{12}$. Nodes with equal path label sets cannot yet be ordered with Lemma 5.2. From Wheeler condition $W2$ we can derive that $(v_{11} < v_{10}) \leftrightarrow (v_1 < v_2) \leftrightarrow (v_5 < v_6)$. But according to the partial order, $(v_{11} < v_{10})$ is false and $(v_5 < v_6)$ is true, a contradiction, so the graph is not Wheeler.

for example if there are strings $\alpha < \beta < \gamma$ such that $\alpha$ and $\gamma$ are in $P_k(u)$, and $\beta$ is in $P_k(v)$ but not in $P_k(u)$.

However, the Lemma is not yet a sufficient characterization of non-Wheeler graphs, as there are non-Wheeler graphs that do not violate it. See Figure 5.4 for an example. The difficulty comes when multiple nodes have the same path label set. Lemma 5.2 says nothing about the relative order within such a set of nodes, but the order cannot be arbitrary either. Finding a quickly decidable characterization of Wheeler graphs is unlikely in the light of the Wheeler graph recognition NP-hardness result of Gibney and Thankachan [31].
5.3 Finite state automata and Wheeler graphs

In this section we summarize the main results of Paper III, without going into the details due to the large size of the paper. Since recognizing Wheeler graphs is NP-complete in general, our paper focuses on the more tractable special case of input-consistent graphs that represent a state graph of a finite state automaton. This way we are also able to benefit from deep results from automata theory.

Some restrictions are made on the automata to keep the theory clean. We assume that every state of the automaton is reachable from the initial state and the transition function of the automaton is allowed to be partial, i.e. states are not required to have outgoing transitions with all edge labels of the alphabet. Any input that would require a transition that is not defined is considered not accepting. In theory this is usually handled so that every undefined transition would go to an implicit absorbing fail-sink state, but we do not want to assume the existence of such state, because it might break the Wheelerness of the graph. We also assume that the automaton does not contain epsilon-transitions. The size of an automaton is defined as the number of edges, denoted with $n$.

One of our first algorithmic results is to show that in non-deterministic automata (NFA) with a limited degree of nondeterminism, it is possible to decide in polynomial time whether the automaton is a Wheeler graph. We say that an NFA has nondeterminism degree $d$ if every state has at most $d$ outgoing edge labels for each character $c \in \Sigma$. We call an such an automaton a $d$-NFA. We show that if a graph is a 2-NFA, we can decide whether it is Wheeler in $O(|V|^2 + |E|^2)$ time by a reduction to 2-SAT, and if it turned out to be Wheeler, we can return the Wheeler order. Our reduction works by creating a variable $x_{uv}$ for each pair of nodes $u$ and $v$, which is intended to be true if and only if $u < v$ in the Wheeler order. We then encode the Wheeler conditions with clauses of two variables each, along with clauses that enforce that the variables produce a valid total order. In general, the clauses that enforce a total order require a clause with three variables, but in the case $d = 2$, those problematic clauses are shown to be unnecessary. This result is in contrast with the work of Gibney and Thankachan [31], who showed that the Wheeler graph recognition problem is NP-hard if $d \geq 5$. It remains to open find out the complexity of the problem for parameters $d = 3$ and $d = 4$.

Recognizing Wheeler graphs turns out to be even easier in 1-NFAs, i.e. deterministic finite state automata (DFA). We denote the the set of path labels from the initial state to a node $v$ with $I_v$. DFAs have the nice property that all sets $I_v$ are disjoint. This means that Lemma 5.2 completely determines the Wheeler order, if it exists. We can simply take a representative string from each $I_v$, order
the strings according to the colex order of the representatives, and check whether the order is a valid Wheeler order. If it is not valid, then the graph cannot be Wheeler. A linear-time algorithm for checking whether a given order is a Wheeler order is given in the paper. We can also find the candidate order in linear time, by taking any spanning tree of the graph, and ordering the nodes with the XBWT sorting algorithm. Additionally, we give an $O(n \log n)$ time online algorithm that works if the graph is acyclic. The algorithm is online in the sense that it adds the nodes to the constructed Wheeler graph one-by-one in any topological order, while maintaining a valid Wheeler order of the existing nodes.

These results show that deterministic automata are easy to handle. It also turns out that we can transform any non-deterministic Wheeler automaton into a deterministic one recognizing the same language, in cubic time, by just running the standard powerset construction algorithm to determinize an automaton. In general this may blow up the size of the automaton exponentially, but it turns out that in Wheeler graphs, the size of the output automaton is always small. The number of states can be bounded to $O(n^2)$ by the path coherence property of Wheeler graphs: the set of active states at any time in the powerset construction is a contiguous interval of nodes in the Wheeler order, and the number of such intervals is at most quadratic in the number of nodes. Further, it turns out that the number of states is only linear because the intervals have a structure such that an interval cannot be a strict subinterval of another, unless they share start points or end points. This is based on a variant of Lemma 5.2. We call such a family of intervals a prefix-suffix family.

We also consider the problem of minimizing Wheeler automata. The problem is to find the smallest Wheeler automaton that recognizes the same language as a given Wheeler automaton. The minimum Wheeler automaton for a language turns out to be unique, and it is characterized as follows: Adjacent states in the Wheeler order are Myhill-Nerode inequivalent, or have different incoming edge labels. Otherwise they could be merged, preserving the language of the automaton and the Wheeler property. This characterization can be used to minimize a Wheeler automaton by finding the Myhill-Nerode equivalence classes of the nodes and merging all runs of Myhill-Nerode equivalent nodes with the same incoming edge label. The equivalence classes can be found in $O(n \log n)$ time by Hopcroft’s

---

2This is consistent with the Wheeler graph recognition NP-hardness result of Gibney and Thankachan [31], because the determinization can turn a non-Wheeler graph into a Wheeler graph.

3Two states $u$ and $v$ are Myhill-Nerode inequivalent iff there exists a string $\alpha$ such that the automaton accepts $\alpha$ if started from $u$ but rejects $\alpha$ if started from $v$, or vice versa.
We also present an algorithm that takes any acyclic DFA as an input and returns the minimum Wheeler DFA recognizing the same language. It works by first minimizing the DFA and then processing the graph in topological order, maintaining the invariant that the part of the graph that has been processed so far is Wheeler, by splitting nodes that violate the Wheeler conditions. This takes time proportional to the size of the input automaton and the output automaton. We show that the size of the output automaton may be $O(2^{n/4})$ in the worst case, where $n$ is the size of the input automaton.

Finally, we study properties of Wheeler languages. We say that a language is Wheeler iff it can be recognized with a Wheeler automaton of a finite size. It turns out that not all regular languages are Wheeler. Intuitively, non-Wheeler regular languages are such that there are two states $u$ and $v$ in the minimum deterministic automaton for the language such that there is an infinite sequence of strings $\alpha_1 \prec \beta_1 \prec \alpha_2 \prec \beta_2, \ldots$ such that strings $\alpha_1, \alpha_2, \ldots$ lead to $u$ and strings $\beta_1, \beta_2, \ldots$ lead to state $v$. This means $u$ and $v$ cannot be sorted, and no finite amount of splitting of states $u$ and $v$ can fix this. This is formalized in a theorem characterizing Wheeler languages analogously to the Myhill-Nerode theorem of finite languages.
Chapter 6

Tunneling on Wheeler graphs

The Burrows-Wheeler transform is remarkably effective at exposing redundancy in a string. If a string is repetitive, its BWT tends to have many long runs where all characters are the same. This is easily exploited for compression by applying run-length coding, which replaces each run with a pair describing the length and the character of the run.

Tunneling is a new method that tries to compress the BWT better than the run-length compressed representation. It was introduced by Baier in 2018 [5]. The insight is that if the string is repetitive, runs of the BWT are often linked in sequence by the LF-mapping in the sense that if \( BWT[i..j] \) is a run, then also \( BWT[LF(i) .. LF(j)] \) is often a run. A sequence of runs \( BWT[LF^0(i) .. LF^0(j)], \ldots, BWT[LF^\ell(i) .. LF^\ell(j)] \), where \( LF^k \) means the LF-mapping iterated \( k \) times, is called a block of width \( j - i + 1 \) and length \( \ell + 1 \). Blocks can be compressed into tunnels.

Tunnel compression is based on the fact that characters in a run in the \( L \)-column of the Burrows-Wheeler matrix map to a contiguous range in the \( F \)-column. Therefore if we know where the first position of the run maps to, we know where all positions in the run map to. This is the redundancy that tunneling seeks to exploit. This is exploited with a pair of bit vectors that let us remove any position \( i \) from the \( L \)-column and corresponding position \( LF(i) \) from the \( F \)-column such that the original LF-mapping can still be computed for positions other than \( i \). This can be repeated many times to remove any number of positions. With this, Baier removes all but the first position of all runs in a block, except for the first and the last run of the block. Thanks to the two bit vectors mentioned earlier, no information is lost in the process, and thus we can decompress to recover the original BWT.
We do not go into the details of how Baier’s tunneling works, but instead give an alternative formulation of the same idea in terms of Wheeler graphs. This brings the benefits of tunneling to any data structure that is based on a Wheeler graph. We also study the possibility of using a tunneled Wheeler graph for subpath searching in the original graph. We show how to decide whether there exists a path in the original graph with a given path label, in time that is linear in the length of the search query. In the special case corresponding to a Wheeler graph of a string, we also show how to implement a fully functional FM-index on top of a tunneled BWT.

This chapter is built on top of the material developed in Sections 5.1 and 5.2 of this thesis. It is advisable to read those sections before proceeding with this chapter.

### 6.1 Generalized tunneling

In this section we generalize Baier’s tunneling to Wheeler graphs. The idea is to merge isomorphic subgraphs satisfying certain restrictions. Suppose we have a Wheeler graph \( G = (V, E) \). The nodes are integers \( 1, \ldots, |V| \) such that the order of the integers matches a Wheeler order of \( G \).

Suppose \( G_1, \ldots, G_w = (V_1, E_1), \ldots, (V_w, E_w) \) are isomorphic disjoint subgraphs of \( G \) such that each of the subgraphs has \( s \) nodes. We denote the nodes of subgraph \( G_i \) with \( V_i = \{v_{i,1}, \ldots, v_{i,s}\} \). The set of edges \( E_i \) is the set of all edges in \( G \) between all pairs of nodes in \( V_i \). The isomorphism between the subgraphs is such that nodes \( \{v_{i,j} \mid 1 \leq i \leq w\} \), where \( j \) is fixed, correspond to each other. The isomorphism preserves both the edge connectivity and edge labels of the subgraphs. These subgraphs form a block which can be compressed into a tunnel. We call \( w \) the width of the tunnel and \( s \) the size of the tunnel. In addition to this setup, we need to place the following restrictions on the types of subgraphs \( G_i \) in a block:

- **Shape:** All the subgraphs need to have a tree shape. That is, for each subgraph \( G_i \), there must exist a root node such there is exactly one path from the root node to all other nodes in the subgraph, and the root node has no incoming edges from within the subgraph.

- **Incoming edges:** The nodes in the subgraphs are not allowed to have incoming edges from outside of the subgraph, except for the root nodes.
6.1 Generalized tunneling

That is, if \((u, v) \in E\) such that \(u \not\in V_i\) but \(v \in V_i\), then \(v\) must be the root of \(G_i\).

- **Determinism**: If there is an edge \((u, v)\) with label \(c\) in \(E_i\), then there is no edge \((u, v')\) with label \(c\) in \(E\) with \(v' \neq v\).

- **Wheeler Adjacency**: The corresponding nodes between the subgraphs must be adjacent in the Wheeler order, that is, we need that for any fixed \(j\) and all \(i = 1, \ldots, w - 1\), it holds that \(v_{i,j} = v_{i+1,j} - 1\).

In fact, in this setup the adjacency requirement is the same as requiring that just the roots of the trees are adjacent in the Wheeler order. This is not shown in our main paper, so we show it here.

**Lemma 6.1.** Let \(G_1 = (V_1, E_1)\) and \(G_2 = (V_2, E_2)\) be disjoint isomorphic subgraphs of a Wheeler graph, both satisfying the shape and determinism constraints above. Denote the root nodes of the graphs with \(v_1\) and \(v_2\) respectively. Then \(G_1\) and \(G_2\) are Wheeler adjacent iff \(v_1\) and \(v_2\) are consecutive in the Wheeler order.

**Proof.** Suppose roots \(v_1\) and \(v_2\) are consecutive in the Wheeler order and suppose without loss of generality that \(v_1 < v_2\). Take any edges \((v_1, u_1, c) \in E_1\) and \((v_2, u_2, c) \in E_2\). Then \(u_1\) and \(u_2\) must also be consecutive: suppose for a contradiction that there is a node \(u'\) between \(u_1\) and \(u_2\). Then \(u'\) must have an incoming edge \((v', u', c)\) labeled with \(c\). Node \(v'\) is distinct from \(v_1\) and \(v_2\) because of the determinism constraint on the subgraphs. Node \(v'\) has to be between \(v_1\) and \(v_2\), or otherwise edge \((v', u', c)\) violates Wheeler condition \(W2\) (Equation 5.4) with either \((v_1, u_1, c)\) or \((v_2, u_2, c)\). But \(v_1 < v' < v_2\) contradicts the consecutivity of \(v_1\) and \(v_2\), so this shows that \(u_1\) and \(u_2\) must be consecutive. By induction, this shows that all corresponding nodes of \(G_1\) and \(G_2\) are consecutive. The other direction of the "iff" is trivial. \(\square\)

**Corollary 6.1.** A set \(G_1, \ldots, G_w\) of disjoint isomorphic subgraphs satisfying the shape and determinism constraints, with root nodes \(v_1, \ldots, v_w\), are Wheeler adjacent iff \(v_i = v_{i+1} - 1\) for all \(i = 1, \ldots, w - 1\). \(\square\)

In our paper we show that merging a set of subgraphs satisfying the requirements stated above preserves the Wheeler conditions. This means that we can merge any number of sets of disjoint subgraphs while maintaining the Wheeler conditions. Each set of merged subgraphs represents a kind of a generalized tunnel. With the help of some lightweight auxiliary data structures, it is possible to recover the
original graph. Namely, we need to mark which nodes are inside a tunnel, and for each tunnel and each character $c$, we need to count the number of incoming edges to the root of the tree, and the number of outgoing edges from a node inside the tunnel to outside of the tunnel. All this information can be stored with $3|E_t|$ bits of space using three bit vectors, where $E_t$ is the number of edges in the tunneled graph.

Now we can simulate traversal in the original graph $G$. We use the normal traversal algorithm for Wheeler graphs described in Section 5.1 with a small modification. Every time we enter a tunnel, we compute and store a tunnel offset, which is a number that represents which copy of the subgraph in the original graph we are entering. This can be computed from the bit vector that we have stored to represent incoming edges to tunnels. When we exit the tunnel, we can find out available edges by using the stored bit vector that encodes outgoing edges from tunnels, and use our tunnel offset to pick the right edge. Figure 6.1 gives some more details and illustrates the idea, and the full details are given in Paper IV.

Once we know how to traverse a tunneled Wheeler graph, it is easy to also search for subpaths in the graph with a given path label. The path search algorithm is just the normal Wheeler graph path searching algorithm, modified to navigate through tunnels as described above.

### 6.2 The special case of the BWT

The second part of our paper focuses on implementing the operations of a fully functional FM-index in the special case of a tunneled BWT. The basic FM-index of a string $T$ takes a query string $\alpha$ and returns the lexicographic range of $\alpha$ in $T$. This tells us how many times $\alpha$ occurs in $T$, but it does not yet tell us where the occurrences are in $T$. For this we would need to have the suffix array. If the lexicographic range of $\alpha$ is $[i..j]$, then the occurrences of $\alpha$ start at indices $SA[i], \ldots, SA[j]$, where $SA$ is the suffix array of $T$. However storing the whole suffix array explicitly takes $\Omega(n \log n)$ bits, where $n = |T|$. The whole point of the FM-index is to manage with only $O(n \log \sigma)$ bits of space, so the suffix array is too large.

The standard workaround is to sample only a subset of the positions of the suffix array. If we sample $[n/s]$ values of the suffix array for some sampling parameter $s$, then the space required is $O((n/s) \log n)$ bits. The usual choice is to set $s = \log n$, which gives total space complexity of $O(n)$. The sampled values are
Figure 6.1: Tunneling on general Wheeler graphs. In the graph on the top, there is a block consisting of three components, marked with blue, green and orange rectangles respectively. The numbers on the nodes correspond to the Wheeler order of the graph. Below is a version of the graph where the block has been compressed into a tunnel. The colors of the edges indicate which colored component we are entering to, or exiting from. Because we have a Wheeler graph, we know that nodes that have an edge to the entrance (node 2) of the tunnel are ordered so that the origins of blue edges come before origins of green edges, and the origins of green edges come before origins of orange edges. Correspondingly, the destination nodes of edges exiting the tunnel have the same relative order for each individual character. If we store for each character the number of edges of each color entering the tunnel, and the number of edges of each color exiting from each tunnel node, we can use the order property to deduce the colors of the edges. These edge counts can be stored compactly by encoding them in unary and concatenating the unary numbers. When the colors are known, no information is lost in the tunneling process, and the original graph can be recovered for the tunneled graph.
chosen so that we can reconstruct any suffix array value in time $O(s \log \sigma)$. We sample the suffix array at every position $i$ such that $SA[i]$ is a multiple of $s$. Now if we want to know the value of $SA[j]$ for some position $j$ that is not sampled, the method is to execute up to $s$ backward steps on the FM-index, taking us to the nearest sampled value, and add the number of backward steps taken to the sampled value.

Let us now cast this into the framework of Wheeler graphs. In the framework of Wheeler graphs, the FM-index corresponds to a graph that is just a single cycle, such that exactly one edge is labeled with the symbol $\$" that is smaller than all other edge labels. The problem of locating an occurrence translates to the problem of transforming the rank of a node in the Wheeler order to the distance of that node from the $\$" on the cycle. The sampling strategy corresponds to storing this distance for evenly spaced nodes on the cycle, so that we can map the Wheeler rank of a sampled node to the corresponding distance value. To avoid special cases, let us assume that the origin of the $\$"-edge is always sampled. Suppose now that we have the Wheeler rank of a node that is not sampled and want to find the distance of the node from the $\$. We use the Wheeler graph traversal operation to walk forward until we have the Wheeler rank of a sampled node, look up the stored distance, and subtract the number of edges we had to traverse. We are guaranteed to find a sampled node in at most $s$ steps. Our goal is to make this work even if the graph has been tunneled.

Each node in the tunneled graph corresponds to one or more nodes in the non-tunneled graph. A first attempt at sampling could be to sample all nodes that correspond to a sampled node in the non-tunneled graph. In case the sampled node falls inside a tunnel, the we also have store a number specifying which of the original nodes the sample corresponds to, i.e. the tunnel offset. Now we can always walk to the next sampled node using at most $s$ steps by using the tunneled traversal algorithm described in the previous section. However, the space taken by this solution is $\Omega((n/s) \log n)$ bits, which is unfortunately proportional to $n$, i.e. the size of the non-tunneled graph. We want to eliminate this linear dependency on $n$, because the purpose of tunneling is to compress the data structure to a space that is potentially much smaller than $n$.

We need a sampling scheme which has the following three properties. First, it should be possible to walk to a sampled node in a reasonable time, second, it should be possible to reconstruct a missing sample from a nearby sample and third, it should take only $O(n_t)$ space, where $n_t$ is the number of nodes in the tunneled graph. Our solution is to sample only nodes that are outside of tunnels, and store shortcut pointers to skip past long tunnels quickly.
Tunnels on a tunneled cycle-graph start with \( k \geq 2 \) nodes all pointing to the same node, followed by a non-branching path of nodes, ending in a node which has \( k \) outgoing edges. Consider the graph formed by taking the tunneled graph and contracting each non-branching path inside a tunnel into a single node. We call this the \textit{contracted graph}. A useful property is that this graph is Eulerian, i.e. there exists a path through the graph passing through every edge exactly once. One such path is the one which follows the path taken by the original cycle. We sample every \( s \) nodes on this path, except that if the sample would fall on a contracted tunnel-node, we sample the next node on the path. See Figure 6.2.

A good choice of sampling parameters gives us total space \( O(n_t) \) with a query time complexity of \( O(\log n \log n_t) \) using constant-time graph traversal steps with the help of a partial rank data structure. Other sampling ideas also allow us to support counting the number of nodes at the end of a given query path label in time \( O(\log^2 n_t) \) and to extract \( k \) consecutive characters from an arbitrary position in the original string in time \( O(k + \log n \log n_t) \). For details, we refer the reader to Paper IV.
Figure 6.2: Suffix array sampling on a tunneled BWT of a string. Let $\alpha = bcabba$ and $\beta = dbaa$. The string corresponding to the tunneled graph in the figure is $T = \alpha a b \beta (\alpha a \beta)^4 \$. Nodes inside tunnels are colored with orange and nodes with suffix array samples are drawn in blue. The skip pointers are not drawn. In the contracted graph, the orange nodes in non-branching paths are contracted to single nodes. After this, walking through the graph following the original string $T$ gives an Eulerian path around the graph. When we enter a contracted tunnel, we remember the relative rank of the node we came from among all nodes that point to the start of the tunnel. When we exit the contracted tunnel, we exit to the out-node with the same relative rank. This keeps us on the Eulerian path.
Chapter 7

Discussion

In this thesis, we have studied the possibilities of compact, succinct and compressed data structures in strings and prefix-sortable graphs. The work was published in four separate papers.

In Paper I, we proposed a method for constructing a greedy approximation of the shortest common superstring for a set of strings. This is the first published method that runs in space proportional to the information-theoretic lower-bound $O(n \log \sigma)$, where $n$ is the length of the input and $\sigma$ is the size of the alphabet. Possible future work could include adapting our solution to support different ways for making the arbitrary greedy choices and to different variants of the superstring problem [18, 61].

In Paper II, we built on the framework of Belazzougui and Cunial [9] to design and implement a practical and scalable set of data structures for representing and querying large variable-order Markov models. The size of our data structures scales linearly with the number of maximal repeats of the training data. This makes them the first known compressed set of data structures for probabilistic suffix trees with efficient query support. Future work could include finding principled ways for setting the various thresholds of the models, and making the data structures dynamic, allowing for example efficient insertion of new sequences to the model.

In Paper III, we applied the theoretical concept of Wheeler graphs [29] to finite-state automata, and thus laid foundations for a theory of prefix-sortable finite-state automata and languages. Our paper introduces two new key concepts: Wheeler automata and Wheeler languages. The theoretical cornerstone of our paper is a precise characterization of the minimum Wheeler automaton of a regular language. The theory leads us to a number of algorithmic results.
First, we show how to recognize Wheeler automata in polynomial time. Second, we show how to determinize a Wheeler automaton in polynomial time, and show that the size of the automaton increases only by a constant factor in the process. Third, we show how to sort a deterministic Wheeler automaton in an offline setting in linear time, or in an online setting with a logarithmic slowdown. Fourth, we show how to minimize an acyclic Wheeler automaton in linear time or a cyclic one with a logarithmic slowdown. Fifth, we show how to turn any deterministic finite-state automaton into the minimum equivalent Wheeler automaton, if it exists, in time $O(n + m \log m)$, where $n$ is the size of the input, and $m$ is the size of the output. Future work includes implementing and evaluating our algorithms and data structures in practice, as well as expanding the newly-found rich theory of Wheeler languages. We have already published a preprint of a continuation paper of the theory [1].

In Paper IV, we generalized the work of Baier [5] on BWT tunneling to Wheeler graphs. This allows us to define a compressed representation of Wheeler graphs that can still support pattern matching queries. The work remains purely theoretical, and implementing and evaluating the approach in practice is left for future work.
References


References


Reports are available on the e-thesis site of the University of Helsinki.


A-2016-4 J. Toivanen: Methods and Models in Linguistic and Musical Computational Creativity. 56+8+79 pp. (Ph.D. Thesis)


