2021

Algorithms and Lower Bounds for Ordering Problems on Strings

Daniel Gibney
University of Central Florida

ALGORITHMS AND LOWER BOUNDS FOR ORDERING PROBLEMS ON STRINGS

by

DANIEL GIBNEY
M.S. University of Central Florida, 2018

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2021

Major Professor: Sharma V. Thankachan

ABSTRACT

This dissertation presents novel algorithms and conditional lower bounds for a collection of string and text-compression-related problems. These results are unified under the theme of ordering constraint satisfaction. Utilizing the connections to ordering constraint satisfaction, we provide hardness results and algorithms for the following: recognizing a type of labeled graph amenable to text-indexing known as Wheeler graphs, minimizing the number of maximal unary substrings occurring in the Burrows-Wheeler Transformation of a text, minimizing the number of factors occurring in the Lyndon factorization of a text, and finding an optimal reference string for relative Lempel-Ziv encoding.

TABLE OF CONTENTS

LIST OF FIGURES viii

LIST OF TABLES xi

CHAPTER 1: INTRODUCTION 1

Burrows-Wheeler Transform Runs Minimization 2

Overview 2

Results 5

Wheeler Graph Recognition 6

Overview 6

Results 11

Lyndon Factor Minimization 12

Overview 12

Results 15

Optimal Reference for Relative Lempel-Ziv Encoding 15

Overview 15

Results 17

CHAPTER 2: BWT-RUNS MINIMIZATION 19

Preliminaries: L-reductions 19

Hardness of Alphabet Ordering 20

Reduction Phase 1 21

Reduction Phase 2 24

Proof of Corollary 1 29

Constrained Alphabet Ordering 29

Reducing to a Simpler Problem 29

Solving the Tuple Ordering Problem in Linear Time 31

An Example of the Effectiveness of CAO 33

CHAPTER 3: WHEELER GRAPH RECOGNITION 34

NP-completeness of Wheeler Graph Recognition 34

The Betweenness Problem 34

Reduction from Betweenness to Wheeler Graph Recognition 35

NP-completeness of Wheeler Graph Recognition on d-NFAs 37

Wheeler graphs and Queue Number 42

Queue Number 42

An Exponential Time Algorithm 44

Optimization Variants to Wheeler Graph Recognition 48

The Wheeler Graph Violation Problem is APX-hard 48

The Reduction of FAS to WGV 49

The Wheeler Subgraph Problem is in APX 53

A Class of Graphs with Linear Time Solution for Recognition 56

PQ-trees 58

Detecting One-Queue DAGs 59

Linear Time Solution 59

Discussion and Open Problems 63

CHAPTER 4: LYNDON FACTOR OPTIMIZATION 65

Preliminaries 65

Hardness of Lyndon Factor Minimization 69

NP-Completeness of Lyndon Factor Minimization 69

ETH Hardness of Lyndon Factor Minimization 72

Inapproximability of Lyndon Factor Minimization 75

Hardness of Lyndon Factor Maximization 79

NP-Completeness of Lyndon Factor Maximization 79

Inapproximability of Lyndon Factor Maximization 80

Open Problems 84

CHAPTER 5: OPTIMAL REFERENCE FOR RELATIVE LEMPEL-ZIV 85

Hardness Results 85

Warm Up: Polynomially-Sized Alphabets 85

Hardness Over a Binary Alphabet 87

Bounds in Terms of the δ-Measure 91

Open Problems 93

CHAPTER 6: CONCLUSION 94

LIST OF REFERENCES 95

LIST OF FIGURES

Figure 1.1: Column L shows the BWT of *mississippi*. The number of runs $r = 9$. 3

Figure 1.2: A Wheeler graph with $\sigma = 3$. Ordering on edge labels: red (solid) < blue (long-dash) < green (short-dash). 9

Figure 1.3: In a proper ordering the above configurations cannot occur with edges that have the same label. 10

Figure 2.1: The modified incidence matrix for the graph G. Each of the first m rows is for an edge. The bottom $2\ell = 8m$ rows are added as are the outer two most columns. 22

Figure 2.2: The graph G constructed for the tuple ordering instance $(0, 1, 2)$, $(0, 1)$, (2), $(1, 2, 3)$. 32

Figure 3.1: An example of the reduction with input list 1, 2, 3, 4, 5, 6 and the triples $(5, 2, 3)$, $(1, 5, 2)$, $(4, 5, 6)$. 35

Figure 3.2: Vertex Z_1 and Z_2 could be for clauses (x_1, x_2, x_3), $(x_2, \overline{x_3}, x_4)$. Each 'betweenness' constraint adds a layer. $(x_4, X, \overline{x_4})$ constraint shown. . . 39

Figure 3.3: A k-gadget replacing directed labeled edge (u, v, k). 46

Figure 3.4: A heavy(bold) edge in Figure 3.5 is actually $k + 1$ subdivided edges. . 50

Figure 3.5: Reduction from FAS to WGV where $T = 1, 2, 3, 4, 5, 6$ and the inequalities are $5 < 3$, $1 < 5$, and $6 < 4$. 51

Figure 3.6: On the left is an example of a small graph that has full spectrum outputs and the unique string traversal property, but is not a Wheeler graph. On the right is an example of a small graph that has both properties and is a Wheeler graph. 57

Figure 3.7: In the figure, p-nodes are represented by circles and q-nodes by rectangles. The orderings represented by this PQ-Tree are orderings where 1 can be reversed with 2, the leaves 3,4, and 5 can be permuted arbitrarily, and the order of the sets of leaves 1,2 and 3,4 5, can be swapped. 58

Figure 3.8: An example Wheeler graph that meets the criteria for this section. Red (solid) edges correspond to edges labeled 1, and blue (dashed) edges correspond to edges labeled 2. 61

Figure 3.9: The tree resulting from Algorithm 2 applied to the Wheeler graph in Figure 3.8. An oval in the tree corresponds to a set of vertices in the Wheeler graph. The labels for these vertices are shown inside each oval. For each set of vertices inside an oval, the strings obtained by concatenating the edge labels on the path from the source is the same. These strings are shown to the side of each oval within the tree. In the tree, the edge colors indicated which type of edge was taken at each step along a path to that set. 62

Figure 5.1: The graph above contains an Eulerian walk. Using reference string $R = v_1v_2v_3v_4$, we show its corresponding set of texts and their encoding. Here $r + p = 4 + 5 = 2|E| + 1$. 86

Figure 5.2: The graph above does not contain an Eulerian walk. Using reference string $R = v_1v_2v_3v_4$, we show its corresponding set of texts and their encoding. Here $r + p = 4 + 6 > 2|E| + 1$. Furthermore, $r + p > 2|E| + 1$ for any reference string. 87

LIST OF TABLES

Table 3.1: Possible relative orderings of a_k, b_k, c_k, Z_k, X subject to (a_k, Z_k, b_k) and (c_k, X, Z_k). 41

Table 3.2: Orderings implied by all-equal assignment that are impossible while satisfying constraints. 41

CHAPTER 1: INTRODUCTION

String algorithms and text-indexing have seen a large number of recent breakthroughs. These breakthroughs are fundamental and range from a newfound understanding of dictionary-based compression [79, 80, 85, 101, 110, 122] to highly functional text-indexes requiring space based on repetitiveness, rather than entropy [52, 78, 86, 109, 111]. The new results also include fine-grained complexity results for several classical string problems [1, 2, 3, 10, 20, 21, 32], helping to justify the lack of improvements on long-standing algorithms.

But with this progress has come many new open questions. Among the most pressing of these are questions of how to optimize the performance of these new techniques. Doing so will help make results that stand currently as theoretical more applicable in practice and likely to find popular implementations. The problems considered in this dissertation are the result of attempting to answer these questions. Perhaps it is no surprise that many of the optimization problems are computationally hard. As such, much of this work is devoted to proving conditional lower bounds. Still, exploring what is required to prove these lower bounds highlights which facets make these problems intractable. The knowledge gained here can be leveraged to help design algorithms that either approximate optimal solutions or work well for a less general set of instances.

We first briefly give some background for each problem, give its formalization, and state our main results around it. Where it is necessary for understanding the statement of the results, the needed technical background is presented in this introduction. Additional, specific technical background needed for proofs is provided in the respective sections.

Burrows-Wheeler Transform Runs Minimization

The work presented here first appeared in the 28th Annual European Symposium on Algorithms, ESA 2020 [16].

Overview

The Burrows-Wheeler Transform (BWT) is an essential building block in the fields of text compression and indexing with a myriad of applications in bioinformatics and information retrieval [92, 93, 95, 108]. Since it was first published in 1994 [22], it has been utilized to provide the popular compression algorithm bzip2 and has been adapted to provide powerful compressed text indexing data structures, such as the FM-index [46]. The FM-index is the backbone of many widely used bioinformatics tools like Burrows-Wheeler Aligner (BWA) [94], SOAP2 [95], Bowtie [92]. Hence, improvements to the algorithmic aspects of this transformation and related data structures can have a significant impact on the research community.

The BWT of a text $T[1, n]$, denoted by $BWT(T)$ is a reversible transformation which can be defined as follows: sort the circular shifts of T in lexicographical order and place the sorted circular shifts in a matrix. By reading the last column of this matrix from top to bottom we obtain $BWT(T)$. To make the transformation invertible a new symbol \$ (lexicographically smaller than others) is appended to T prior to sorting the circular shifts. See Figure 1.1 for an example. Historically, the BWT was introduced for the purpose of text compression [22], where its effectiveness is based on symbols with shared preceding context forming long *runs* (maximal unary substrings).

Recently, the number of runs "r" in the BWT has become of increasing interest. This can be

F		L
$	mississipp	i
i	$mississip	p
i	ppi$missis	s
i	ssippi$mis	s
i	ssissippi$	m
m	ississippi	$
p	i$mississi	p
p	pi$mississ	i
s	ippi$missi	s
s	issippi$mi	s
s	sippi$miss	i
s	sissippi$m	i

Figure 1.1: Column L shows the BWT of *mississippi*. The number of runs $r = 9$.

attributed to the fact that many modern text collections are highly repetitive, which makes their compression effective via the BWT followed by Run-Length encoding (i.e., in space proportional to r). This raised an interesting question: can we also index the text in space propositional to r? Note that the FM-index needs space proportional to n (i.e., $\approx n \log \sigma$ bits, where σ is the alphabet size). The data-structure community has made great strides in making the size of a BWT-based index proportional to r rather than n [11, 18, 51, 81, 88, 115]. The first such index was developed by Mäkinen and Navarro in 2005 [99]. However, it lacked the ability to efficiently locate the occurrences of a pattern within space $\tilde{O}(r)$. After a decade of related research [100, 51], we now have fully functional suffix trees in space proportional to r, developed by Gagie *et al.* [52]. Also note that the recent optimal BWT construction algorithm for highly repetitive texts is parameterized by r [78]. A technique reducing the value of this parameter r would have a significant impact on a large body of work.

A natural way to minimize r is to change the lexicographic ordering assigned to symbols of

the alphabet. To demonstrate that this can have an impact on r, consider as an example the text *mississippi* with the usual ordering $\$ < i < m < p < s$ where $r = 9$, but with the ordering $\$ < s < i < p < m$ we have $r = 8$. In fact, there exist string families in which r differs by a factor of $\Omega(\log n)$ for different orderings. This problem of reordering the alphabet is clearly fixed-parameter tractable in alphabet size σ and has a trivial $O(\sigma!\ n)$ time solution. This may be adequate when σ is small as in DNA sequences. However, this is far from satisfactory from a theoretical point of view, or even from a practical point when the alphabet is slightly larger, such as in protein sequences, natural language texts, etc.

A work in 2018 on block sorting based transformations by Giancarlo *et al.* gives a theoretical treatment of alphabet ordering in the context of the Generalized BWT [55]. It was shown that for any alphabet ordering, r is at most twice the number of runs in the original text, a result which then holds for the standard BWT as well. Note however that this gives no lower bound on r, and thus gives no results on the approximability of the run minimization problem. There have been multiple previous attempts to develop other approaches to alphabet ordering. In bioinformatics, the role of ordering on proteins was considered in [126] with approaches evaluated experimentally. Similar heuristic approaches evaluated through experiments were done in [4]. Researchers have also considered more restricted versions of this problem. For example, one can try to order a restricted subset of the alphabet, or limit wherein the ordering symbols can be placed. On this problem, heuristics have been utilized. Software tools like BEETL utilize these techniques to handle collections of billions of reads [33]. Another related work in [23] shows, how to permute a given set of strings in linear time, such that the number of runs in the BWT of the (long) string obtained by concatenating the input strings, separated by the same delimiter symbol is minimized.

Even more recently, a work by Giancarlo *et al.,* considered the case where ordering is assigned to the nodes of a string's suffix tree, to minimize the number of runs in the BWT [56].

Interestingly, this problem can be solved in polynomial time. Although their technique can potentially minimize the number of runs in the BWT to an even greater extent than modifying the ordering on the alphabet, it also requires storing the order for each of these nodes, which can require more space. We leave open the problem of finding a trade-off between the strategy of ordering the alphabet and ordering the nodes of the suffix tree.

Given the lack of success with attacking the main problem from the upper bound side, perhaps it is best to approach the problem from the perspective of lower bounds and hardness. To this end, we show why a provably efficient algorithm has been evasive.

Let Σ denotes the alphabet and $\sigma = |\Sigma|$. A run in a string T is a maximal unary sub-string. Let $\rho(T)$ be the number of runs in T. The problem we are interested in is defined as follows.

Problem 1 (Alphabet Ordering (AO))**.** *Given a string $T[1, n]$ and an integer t, decide whether there exists an ordering of the symbols in its alphabet such that $\rho(BWT(T)) \leq t$.*

Results

The first result is the hardness of the problem.

Theorem 1. *The alphabet ordering problem is NP-complete and its corresponding minimization problem is APX-hard.*

The problem can be solved in $n \cdot \sigma! = n \cdot 2^{O(\sigma \log \sigma)}$ time naively. However, any significant improvement seems unlikely as per the Exponential Time Hypothesis (ETH) [96].

Corollary 1. *Under ETH, AO cannot be solved in time $poly(n) \cdot 2^{o(\sigma)}$.*

It is known that $\rho(BWT(T))$ can be lower bounded by the size of string attractor γ, a recently proposed compressibility measure [80]. Kempa and Kociumaka showed that $\rho(BWT(T))$ can

be upper bounded by $O(\gamma \log^2 n)$ [79]. However, γ is independent of the alphabet ordering and the following result is immediate.

Corollary 2. *Any alphabet ordering is an $O(\log^2 n)$-approximation for AO.*

We also introduce a specialization of AO, one where we impose more constraints on the ordering given to alphabet symbols.

Problem 2 (Constrained Alphabet Ordering (CAO))**.** *Given a set of d strings $T_1, \ldots, T_d$ of total length N, find an ordering π on the symbols $\$_i$ $(1 \leq i \leq d)$ such that $\$_{\pi(1)} \prec \$_{\pi(2)} \ldots \prec \$_{\pi(d)} \prec 0 \ldots \prec \sigma - 1$ and $\rho(BWT(T_1\$_1T_2\$_2 \ldots T_d\$_d))$ is minimized.*

We call $\$_1, \$_2, \ldots, \$_d$ *special symbols*. In Section 2, we provide an example where an optimal ordering of special symbols removes a factor of $\Omega(\log_\sigma d)$ in the number of runs, demonstrating that this can be a worthwhile preprocessing step. We refer to [33] for an immediate use case in bioinformatics, where the input is a large collection of DNA reads.

Theorem 2. *The constrained alphabet ordering problem can be solved in linear time.*

Wheeler Graph Recognition

The work presented here first appeared in the 28th Annual European Symposium on Algorithms, ESA 2019 [58].

Overview

Within the last two decades, there has been the development of Burrows-Wheeler Transform (BWT) [22] based indices for compressing a diverse collection of data structures. This

list includes labeled trees [123], certain classes of graphs [45, 114], and sets of multiple strings [48, 102]. These new techniques have motivated the search for a set of general conditions under which a structure can be indexed by a BWT based index, which itself led to the recent introduction of Wheeler graphs by Gagie *et al.* [50] (also see [8]). A Wheeler graph is a directed graph that has edge labels and satisfies two simple axioms related to the ordering of its vertices. Although not general enough to encompass all BWT-based structures (e.g., [54]), Gagie *et al.* demonstrated that Wheeler graphs offer a unified way of modeling several BWT based data structures such as representations of de Bruijn graphs [19, 38], generalized compressed suffix arrays [123], multi-string BWTs [103], XBWTs [45], wavelet matrices [31], and certain types of finite automaton [5, 15, 70]. They also showed that there exists an encoding of a Wheeler graph $G = (V, E)$ which requires only $2(e + n) + e \log \sigma + \sigma \log e + o(n + e \log \sigma)$ bits where σ is the size of the edge label alphabet, $e = |E|$, and $n = |V|$. This encoding allows for the efficient traversal of multiple edges while processing characters in a string, using an algorithm similar to the backward search in the FM-index [47]. This allows for near optimal time matching of patterns to paths on an indexed Wheeler graph.

Since their introduction, Wheeler graphs have been the subject of significant study. This includes the study of the languages that are accepted by automata that are also Wheeler graphs [7], as well as the extension of a technique for compression known as tunneling to the BWTs of Wheeler graphs [8]. It is important to note, however, that not all directed edge labeled graphs are Wheeler graphs. In fact, conditional lower bounds show that matching patterns to paths on DAGs with maximum total degree three and binary alphabets should require quadratic time, even with arbitrary polynomial preprocessing of the graph [42, 43, 57]. Despite being the subject of an increasing amount of research, it was not clear how to recognize whether a given graph is a Wheeler graph. This fact made the authors of [50] explicitly pose the question of *how to efficiently detect whether a graph is a Wheeler graph.*

The question is of both theoretical and practical value, as it might be the first step before attempting to apply some compression scheme to a given graph. For example, one could use the existence of a *Wheeler subgraph* to encode a graph. To do so, one maintains an encoding of the subgraph using the framework presented in [50] in addition to an adjacency list of the edges not included in the encoding. Depending on the size of the subgraph, such an encoding might provide large space savings at the cost of a modest time trade-off while traversing the graph. This concept also motivates the portion of the paper where we look at two *optimization versions* of this problem that seek subgraphs of the given graph that are Wheeler graphs. These problems turn out to be computationally difficult as well. As a positive result, we show that, for a constant sized alphabet, the problem of finding a maximum Wheeler subgraph admits a polynomial-time algorithm that outputs a solution with size within some constant factor of optimal. We also show that the problem of recognizing Wheeler graphs is similar to that of identifying the queue number of a graph. This helps to indicate a class of graphs where the problem becomes computationally tractable.

We first give the definition of a Wheeler graph. The notation (u, v, k) is used for the directed edge from u to v with label k. We will assume the usual ordering on the edge labels which come from the alphabet $\{1, 2, ..., \sigma\}$.

Definition 1. *A Wheeler graph is a directed graph with edge labels where there exists an ordering π on the vertices such that for any two edges (u, v, k) and (u', v', k'):*

1. $k < k' \implies v <_\pi v'$;
2. $(k = k') \wedge (u <_\pi u') \implies v \leq_\pi v'$.

Additionally, vertices with in-degree zero must be placed first in the ordering.

We consider an ordering of the vertices of the graph a *proper ordering* if it satisfies the axioms of the Wheeler graph definition. See Figure 1.2 for an illustration. One critical property of Wheeler graphs is called *path coherence*. This property is characterized by the fact that if you start at any consecutive range of vertices under the proper ordering π, and traverse the graph by following edge labels matching the characters in a string P, then when finished processing P the vertices ended on will form a consecutive range. This property is key to allowing the efficient traversal of multiple edges simultaneously, as well as achieving a compressed representation of the graph.

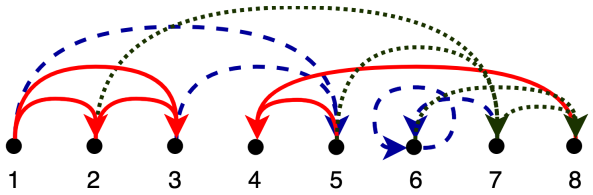


Figure 1.2: A Wheeler graph with $\sigma = 3$. Ordering on edge labels: red (solid) $<$ blue (long-dash) $<$ green (short-dash).

The following list of properties of Wheeler graphs can be deduced from Definition 1.

Property 1. *All edges inbound to a vertex v have the same edge label.*

Property 2. *In a proper ordering all vertices with the same inbound edge label are ordered consecutively.*

Property 3. *A vertex can have multiple outbound edges with the same label. It is also possible for a vertex to have more than σ inbound or outbound edges.*

Property 4. *Two edges with the same label, (u, v, k) and (u', v', k), where $u < u'$ and $v' < v$ are called a monochromatic rainbow. No monochromatic rainbows can exist in a proper ordering (see Figure 1.3).*

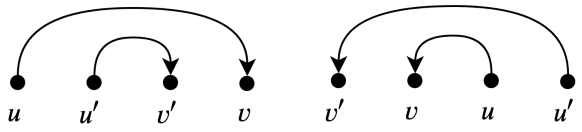


Figure 1.3: In a proper ordering the above configurations cannot occur with edges that have the same label.

The first question we wish to answer is given a directed graph with edge labels, does there exist a proper ordering π for its vertices? We define this problem formally as the following.

Problem 3 (Wheeler Graph Recognition)**.** *Given a directed edge labeled graph $G = (V, E)$, decide whether G is a Wheeler graph.*

Although we do not demand it here, ideally, a solution to the above problem would also return a proper ordering.

Next, we define two optimization versions of Problem 3 where we seek to find Wheeler subgraphs.

Problem 4 (Wheeler Graph Violation (WGV))**.** *Given a directed edge labeled graph $G = (V, E)$, identify the smallest $E' \subseteq E$ such that $G' = (V, E \backslash E')$ is a Wheeler graph.*

We also consider the dual of this problem.

Problem 5 (Wheeler Subgraph (WS))**.** *Given a directed edge labeled graph $G = (V, E)$, identify the largest $E'' \subseteq E$ such that $G'' = (V, E'')$ is a Wheeler graph.*

Results

We show that the problem of recognizing whether a given graph is a Wheeler graph is NP-complete, even for an edge alphabet of size $\sigma = 2$.

Theorem 3. *The Wheeler Graph Recognition Problem is NP-complete for any $\sigma \geq 2$.*

This result holds even when the input is a directed acyclic graph (DAG) and when the number of edges leaving a vertex with the same label is at most five.

We also relate the notion of queue number to Wheeler graphs, allowing us to place a bound on the number of edges of any Wheeler graph.

We provide an exponential time algorithm which solves the recognition problem on a graph $G = (V, E)$ in time $2^{O(n+e\log\sigma)}$ where $n = |V|$ and $e = |E|$. It uses the idea of enumerating through all possible encodings of Wheeler graphs (of bounded size), and the fact that we can test whether there exists an isomorphism between two undirected graphs in sub-exponential time. This technique also gives us exact algorithms with the same time complexity for the optimization variants discussed here.

Theorem 4. *Recognizing whether $G = (V, E)$ is a Wheeler graph can be done in time $2^{e\log\sigma+O(n+e)}$, where $n = |V|$, $e = |E|$, and σ is the size of the edge label alphabet.*

We examine the optimization variants of this problem called Wheeler Graph Violation (WGV) and Wheeler Subgraph (WS). We show via a reduction of the Minimum Feedback Arc Set problem that the Wheeler Graph Violation problem is APX-hard, and assuming the Unique Games Conjecture, cannot be approximated within a constant factor. This holds even when the graph is a DAG. On the other hand, we show that the Wheeler Subgraph

problem is in the complexity class APX for $\sigma = O(1)$. We do so by providing a poly-time algorithm whose solution size is $\Omega(1/\sigma)$ times the optimal value.

Theorem 5. *Conditioned on the Unique Games conjecture, for every constant $C \geq 1$, it is NP-hard to find a C-approximation to WGV, implying WGV is not in APX.*

Theorem 6. *There exists a linear time $\Omega(1/\sigma)$-approximation algorithm for WS.*

In the final part of this chapter, using PQ-trees and ideas similar to those used in detecting if the queue number of a DAG is one, we demonstrate a class of graphs where Wheeler graph recognition can be done in linear time.

Lyndon Factor Minimization

The work presented here first appeared in the 38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021 [59].

Overview

This chapter establishes several strong hardness results on the problem of finding an ordering on a string's alphabet that either minimizes or maximizes the number of factors in that string's Lyndon factorization. In doing so, we demonstrate that these ordering problems are sufficiently complex to model a wide variety of ordering constraint satisfaction problems (OCSPs). Based on this, we prove that (i) the decision versions of both the minimization and maximization problems are NP-complete, (ii) for both the minimization and maximization problems there does not exist a constant approximation algorithm running in polynomial time under the Unique Game Conjecture and (iii) there does not exist an algorithm to solve

the minimization problem in time $poly(|T|) \cdot 2^{o(\sigma \log \sigma)}$ for a string T over an alphabet of size σ under the Exponential Time Hypothesis (essentially the brute force approach of trying every alphabet order is hard to improve significantly).

A Lyndon word is a string that is lexicographically strictly smallest among all of its cyclic shifts. Letting $\circ$ denote concatenation, the Lyndon factorization of a string T is the factorization of T into Lyndon words T_1, T_2, ..., T_f that are lexicographically non-increasing and $T = T_1 \circ T_2 \circ \ldots \circ T_f$. For example, the Lyndon factorization of $0, 1, 0, 0, 2, 1, 1, 0, 0, 1, 0, 1, 1, 2$ is $(0, 1), (0, 0, 2, 1, 1), (0, 0, 1, 0, 1, 1, 2)$, assuming the usual ordering, $0 < 1 < 2$.

Lyndon words and Lyndon factorization are well-studied, and play an important role in string algorithms [12, 13, 35, 87, 104, 107], algebra and combinatorics [27, 69, 90], and data compression [49, 72, 76, 124, 125]. As an example, it was shown in [105] that local suffixes inside each Lyndon factor can be sorted independently and then merged to construct a string's suffix array. As another example, Lyndon factorization is used in both the construction of a string's bijective Burrows-Wheeler transform (BBWT) [60] and in performing pattern matching on indexes built from the string's BBWT [14], where the number of steps used to locate occurrences of a pattern P depends on the number of Lyndon factors within a particular suffix of P. Because of such applications, it would be beneficial to be able to control the number of factors in the Lyndon factorization of a string. Unfortunately, the Lyndon factorization of a string is unique under a fixed ordering of its alphabet [97]. However, it can vary under different alphabet orderings. For instance, if we change the alphabet ordering to $2 < 0 < 1$ in our example above, we obtain the Lyndon factorization $(0, 1)$, (0), (0), $(2, 1, 1, 0, 0, 1, 0, 1, 1)$, (2). This leads to the following problems:

Problem 6 (Lyndon Factor Minimization - Decision Version)**.** *Given an integer t and text T over alphabet Σ, does there exist an ordering on Σ such that the number of Lyndon factors*

of T is at most t?

Problem 7 (Lyndon Factor Maximization - Decision Version)**.** *Given an integer t and text T over alphabet Σ, does there exist an ordering on Σ such that the number of Lyndon factors of T is at least t?*

We will also consider the *optimization variants* of these problems. The objective cost of a solution is the number of factors in its Lyndon factorization. In particular, for the minimization problem, a λ-approximation for $\lambda > 1$, is a polynomial-time algorithm that outputs an alphabet ordering where the number of factors is at most λ times the minimum possible number of factors over all possible alphabet orderings. Similarly, for the maximization problem, a λ-approximation for $\lambda < 1$, is a polynomial-time algorithm that outputs an alphabet ordering where the number of factors is at least λ times the maximum number of possible factors over all possible alphabet orderings.

These problems were first considered by Clare and Daykin, who proposed a polynomial-time greedy algorithm that can be adjusted to provide either a small number of factors or a large number of factors [29]. Through experiments, the authors showed that the number of factors can be significantly affected by their algorithm. Another approach that uses evolutionary algorithms to find alphabet orderings to optimize the number of Lyndon factors was considered in [30] and in [98]. Again, it was shown that there is often a significant effect on the number of factors, which can be controlled by the use of different fitness functions within the evolutionary algorithms. These techniques, although appearing to have a significant impact on the number of factors, do not provide any approximation guarantee.

Although the Lyndon factors of a string determine the structure of its BBWT, we see no clear relation between the number of Lyndon factors of a string and the number of maximal unary substrings occurring in its BWT. Moreover, the techniques applied here seem quite

different from those used in Chapter 2.

Results

Theorem 7. *The decision version of Lyndon Factor Minimization is NP-complete.*

Theorem 8. *Under the Exponential Time Hypothesis, the optimization version of Lyndon Factor Minimization cannot be solved in time* $poly(|T|) \cdot 2^{o(|\Sigma| \log |\Sigma|)}$.

Theorem 9. *Under the Unique Games Conjecture, the optimization version of Lyndon Factor Minimization does not admit a* λ*-approximation for any constant* $\lambda > 1$.

Theorem 10. *The decision version of Lyndon Factor Maximization is NP-complete.*

Theorem 11. *Under the Unique Games Conjecture, the optimization version of Lyndon Factor Maximization does not admit a* λ*-approximation for any constant* $\lambda < 1$.

We leave open whether it is possible to have a result similar to Theorem 8 for Lyndon Factor Maximization.

Optimal Reference for Relative Lempel-Ziv Encoding

Overview

The final technical chapter of this dissertation is different than the previous sections, in that the problem addressed here is ostensibly not an ordering problem. However, we show that the problem is hard enough to allow for reductions from hard ordering problems, namely the Spanning Eulerian Subgraph problem and the Hamiltonian Path problem. Through

these reductions, we prove the hardness of this optimization problem arising from text-compression. The problem is to find an optimal reference for Relative Lempel-Ziv (relative-LZ) encoding. In addition to these hardness results, we present two positive approximation results conditioned on a plausible conjecture, and bounds on the cost of an optimal solution in terms of another important compressibility measure, the δ-measure.

Relative-LZ encodes a collection of strings $\mathcal{T} = \{T_1, \ldots, T_n\}$ using a reference string R. Each string in the collection is encoded using a set of 'pointers' to R each consisting of a starting and ending index in the reference. More formally, a relative-LZ encoding of $\mathcal{T}$ consists of a reference string R and an ordered set tuples (x_1, y_1), (x_2, y_2), ..., (x_p, y_p) where we consider some of the tuples marked to represent the start of a new text. The text $T_i \in \mathcal{T}$ is equal to $T_i = R[x_{i'}, y_{i'}] \circ R[x_{i'+1}, y_{i'+1}] \circ \ldots \circ R[x_j, y_j]$ where $(x_{i'}, y_{i'})$ is the i^{th} marked tuple in the encoding and (x_j, y_j) precedes the $(i+1)^{th}$ marked tuple. We refer to each tuple (x_i, y_i) as a single pointer. The typical (and optimal) way these pointers to R are assigned is in a greedy fashion, extending each substring as far left as possible. As an illustration, if $\mathcal{T}$ consists of texts $T_1 = abba$, $T_2 = babaa$, and the reference string is $R = abaa$, we encode T_1 with the pointers $(1, 2), (2, 3)$ and T_2 with the pointers $(2, 3), (2, 4)$.

Relative-LZ encoding was first formalized in [89], where it was shown to be an effective method for building compressed indices when the reference string is a well chosen one for the collection. Further data structures based on relative-LZ encoding were developed in [39, 112, 120]. As the name suggests, relative-LZ is closely related to other Lempel-Ziv encodings. The one most relevant here is LZ77 [128]. LZ77 decodes strings from left-to-right and works by using references to the previously decoded string, with an additional character for each pointer used to extend each newly added substring by one character. One advantage over relative-LZ when compared to LZ77 is the working space required by the encoding algorithm. The most space-efficient algorithm known for LZ77 requires linear space [77],

whereas the straight-forward algorithm for creating a relative-LZ encoding requires only working space proportional to the size of the reference.

Clearly the more representative the reference string R is of the set of the collection the fewer pointers will be needed. At the same time, a space efficient encoding should consist of small reference string, as well as a small number of pointers. That is, a good reference should minimize a sum related to $r + p$, where $r = |R|$ and p is the total number of pointers. This leads to the computational problem of finding the optimal reference string for a give collection. We formalize this below.

Problem 8 (The Optimal Reference Problem - Decision Version)**.** *Given a set of texts $\mathcal{T} = \{T_1, ..., T_n\}$ of total size $N = \sum_{i=1}^{n} |T_i|$, an integer t, and (not necessarily constant) values $\alpha \geq 1$ and $\beta \geq 1$, does there exist a reference string R such that the Relative-LZ encoding of $\mathcal{T}$ using R has $\alpha r + \beta p \leq t$, where $r = |R|$ and p is the total number of pointers?*

This problem has been studied previously [53, 71]. In [53] the effectiveness of randomly sampling from the collection was analyzed. In that work some conditions on the text that will lead to effective relative-LZ compression were established. Other work based on random sampling to obtain a reference was done in [71] where performance was evaluated experimentally on large test collections. However, neither of these previous studies seem to give guidance on how to find an optimal reference string.

Results

Our first result establishes the computational complexity of this problem.

Theorem 12. *The Optimal Reference problem is NP-complete, even over binary alphabets.*

Theorem 13 relates an algorithm that solves the Optimal Reference problem over a single string to an algorithm that solves the Optimal Reference problem over arbitrarily many strings.

Theorem 13. *An algorithm for the Optimal Reference problem on one string provides a 2-approximation algorithm for the Optimal Reference problem over arbitrarily many strings having the same time complexity.*

Finally, we present some bounds on the optimal cost of a relative-LZ encoding in terms of the δ-measure. The δ-measure was introduced in [122] and put more into the context of measuring repetitiveness in [85]. The definition of the δ-measure is as follows. Letting d_k denote the number of distinct substrings of T of length k, $\delta = \max_k \frac{d_k}{k}$. The δ-measure lower bounds many of the other known measures of compressibility. This includes the size of the LZ77 parse tree z, that is $\delta \leq z$ [109].

We have the following result.

Theorem 14. $\max\left(\delta, 2\sqrt{\alpha\beta}\sqrt{N}\right) \leq \alpha r^* + \beta p^* = OPT \leq 2(\alpha\delta)^{\frac{1}{3}}(\beta N)^{\frac{2}{3}}$.

CHAPTER 2: BWT-RUNS MINIMIZATION

The work presented here first appeared in the 28th Annual European Symposium on Algorithms, ESA 2020 [16].

Preliminaries: L-reductions

Our inapproximability results use L-reductions [34]. We will be reducing a problem A, with some known inapproximability results, to a new problem B. We will use the following notation:

- $\mathrm{OPT}_A(x)$ denotes the cost of an optimal solution to the instance x of Problem A.
- $c_A(y)$ denotes the cost of a solution y to an instance x of Problem A (suppressing the x in the notation $c_A(x, y)$).
- Since all problems presented here are minimization problems the approximation ratio can be written as $R_A(x, y) = \frac{c_A(y)}{\mathrm{OPT}_A(x)}$, which is ≥ 1.
- Let $f_A(x) = x'$ be a mapping of an instance x of Problem A to instance x' of Problem B.
- Let y' be a solution to instance $x' = f_A(x)$ and $g_B(y') = y$ be the mapping of a solution y' to a solution y for instance x.

Taking x, y, x' y' as above, an L-reduction is defined by the pair of functions (f_A, g_B), computable in polynomial time, such that there exist constants $\alpha, \beta > 0$, where for all x and

y the following two conditions hold:

$$\mathrm{OPT}_B(f_A(x)) \leq \alpha\,\mathrm{OPT}_A(x) \;\text{ and }\; c_A(g_B(y')) - \mathrm{OPT}_A(x) \leq \beta\Big(c_B(y') - \mathrm{OPT}_B(f_A(x))\Big).$$

As a result, $R_B(x', y') = 1 + \varepsilon$ implies $R_A(x, y) \leq 1 + \alpha\beta\varepsilon = 1 + O(\varepsilon)$. L-reductions preserve APX-hardness [117].

Hardness of Alphabet Ordering

We will demonstrate a sequence of L-reductions from the $(1, 2)$-TSP Cycle problem, where the aim is to find a Hamiltonian cycle of minimum weight through an undirected *complete graph* on n vertices where all edges have weights either 1 or 2. The $(1, 2)$-TSP Cycle problem is APX-hard, even with only $\Theta(n)$ edges of weight 1 [118]. The first reduction is to $(1, 2)$-TSP Path, where the goal is to find a Hamiltonian path of minimum weight, rather than a cycle.

Lemma 1. *(1,2)-TSP Path is APX-hard, even with only $\Theta(n)$ edges of weight 1.*

Proof. We will give an approximation preserving reduction from $(1, 2)$-TSP to $(1, 2)$-TSP Path. By the APX-hardness of $(1, 2)$-TSP Cycle, we obtain Lemma 1.

Let x be the input graph G for $(1, 2)$-TSP Cycle and let f_A map the graph G to an identical graph G'. Let g_B map the $(1, 2)$-TSP Path y' given to G' to the cycle in G obtained by connecting the end points of the path with an edge of weight at most 2. Hence the cost $c_B(y')$ is always at most the cost $c_A(g_B(y'))$. At the same time, the weight $\mathrm{OPT}_A(x)$ of an optimal cycle in G is bound above by the weight $\mathrm{OPT}_B(f_A(x))$ of an optimal path in G' plus

2. Thus, $c_B(y') \leq c_A(g_B(y'))$ and $\mathrm{OPT}_A(x) \leq \mathrm{OPT}_B(f_A(x)) + 2$. Therefore,

$$\frac{\mathrm{OPT}_B(f_A(x))}{c_B(y')} \leq 1 + \varepsilon \implies \frac{\mathrm{OPT}_A(x)}{c_A(g_B(y'))} \leq \frac{\mathrm{OPT}_B(f_A(x)) + 2}{c_B(y')} \leq 1 + \varepsilon + \frac{2}{n} \leq 1 + O(\varepsilon).$$

□

We proceed to present our reduction which consists of two phases.

Reduction Phase 1

Given a complete graph on n vertices and $m = \Theta(n)$ edges of weight 1 as input to (1,2)-TSP Path, remove all edges of weight 2. We call the resulting graph G. Construct the incidence matrix for G (a row for each edge, and a column for each vertex, where the two 1's in a row indicate which two vertices are incident to the edge for that row). Then add 2ℓ rows of all 0's to bottom of the matrix, where $\ell = 4m$. Next, add two additional columns c_s and c_t where $c_s[i] = 1$ if $i \in \{m+2, m+4, \ldots, m+2\ell\}$ and 0 otherwise, and $c_t[i] = 1$ if $i \in \{m+1, m+3, \ldots, m+2\ell-1\}$ and 0 otherwise (see Figure 2.1). We call this matrix M.

We now present an intermediate problem that we call Column Ordering (CO), which is: given a matrix M constructed as above, find an optimal ordering on the columns so as to *minimize the number of runs in its linearization.* We will use M_π to denote the matrix M with the ordering π applied to its columns and $L(M_\pi)$ to denote the string obtained by concatenating the rows of M_π from top to bottom. We call $L(M_\pi)$ the linearization of M_π.

Next, we describe the function which maps solutions of our instance of Column Ordering back to a solution of $(1,2)$-TSP Path. Ignoring the added columns c_s and c_t, the ordering π induces a collection of disjoint paths in G, which we call P, where two vertices form an edge

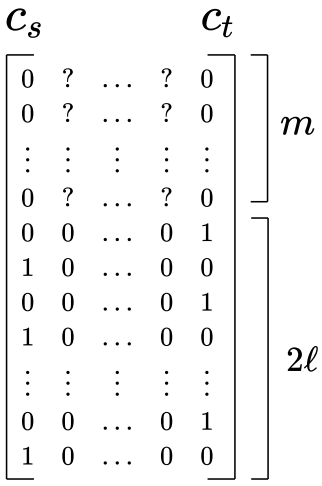


Figure 2.1: The modified incidence matrix for the graph G. Each of the first m rows is for an edge. The bottom $2\ell = 8m$ rows are added as are the outer two most columns.

if their columns are adjacent and there exists a row with 1's in both columns. Given P we create a (1,2)-TSP Path by connecting the paths in P with $|P| - 1$ edges of weight 2. Note that this can be done in linear time.

Lemma 2. *If c_s and c_t are the first and last columns of M_π respectively, then the cost of our CO solution is $\rho(L(M_\pi)) = 2m_1 + 4(m - m_1) + 2\ell + 1 = 4m - 2m_1 + 2\ell + 1$, where m_1 is the number of rows whose edges are in the collection of paths P. The corresponding cost of the solution to (1,2)-TSP Path is $m_1 + 2(n - 1 - m_1) = 2(n - 1) - m_1$.*

Proof. Ignoring the first run of $L(M_\pi)$ for the moment, every row in M_π corresponding to an edge in P contributes two runs to $\rho(L(M_\pi))$ (e.g. $0 \ldots 0110 \ldots 0$). Any row whose edge is not in P and not in the bottom 2ℓ rows, contributes four (e.g. $0 \ldots 010 \ldots 010 \ldots 0$) and there are $m - m_1$ such (rows) edges. The extra 2ℓ rows in total contribute 2ℓ runs. Adding the '+1' term for the start of $L(M_\pi)$ gives the desired expression. The second statement follows from the TSP Path having m_1 edges of weight 1 and the $n - 1$ edges in total needed to form a Hamiltonian path. □

Lemma 3. *If c_s and c_t are not the first and last columns respectively, then the solution to CO is sub-optimal.*

Proof. If c_t is first and c_s is last, then one extra run is contributed over c_s being first and c_t last, while maintaining the rest of the ordering to be the same. In any configuration where either c_s or c_t are not ends of the matrix, the bottom rows will contribute at least 3ℓ runs. Letting m_1^* denote the optimal number of edges of P, then the optimal $\rho(L(M_{\pi^*}))$ is $4m-2m_1^*+2\ell+1 < 4m+2\ell \leq 3\ell$. Note that the first inequality is strict since we can always find at least one edge for P. □

It is immediate from Lemmas 2 and 3 that an optimal solution for CO is one which maximizes m_1, and this provides an optimal solution for (1,2)-TSP Path. We now must show that our reduction is also an L-reduction. Lemmas 4 and 5 consider the two possible cases.

Lemma 4. *If c_s and c_t are the first and last columns respectively in a solution to CO, then the L-reduction conditions hold.*

Proof. By Lemmas 2 and 3, the optimal cost for the instance of CO can be expressed as $4m-2m_1^*+2\ell+1$ and the optimal cost for the instance of (1,2)-TSP Path as $2(n-1)-m_1^*$. To prove Condition (i), we need to show there exists an $\alpha > 0$ such that

$$4m - 2m_1^* + 2\ell + 1 \leq \alpha(2(n-1) - m_1^*)$$

Since $m = \Theta(n)$ there exists a constant $C > 1$, such that for n large enough $m \leq Cn$. The left hand side can be bounded above by $4Cn - 2m_1^* + 8Cn + 1 = 12Cn - 2m_1^* + 1$ (recall $\ell = 4m$). Since $m_1^* \leq n-1$ it is easy to find such an α for $n \geq 2$. Below is the inequality for

Condition (ii), which is true for $\beta \geq 1/2$.

$$(2(n-1) - m_1) - (2(n-1) - m_1^*) \leq \beta\Big((4m - 2m_1 + 2\ell + 1) - (4m - 2m_1^* + 2\ell + 1)\Big)$$

□

Lemma 5. *If c_s and c_t are not the first and last columns respectively in a solution to CO, the L-reduction conditions still hold.*

Proof. Condition (i) holds since the optimal solution values to the overall problem have not changed. For Condition (ii), we consider the two scenarios:

- **Scenario 1**: c_s or c_t are not at the far ends of M_π. Then the cost of the solution for CO, which is at least 3ℓ, exceeds the cost for any solution considered in Lemma 4. Furthermore, any corresponding solution for (1,2)-TSP Path has already been considered in Lemma 4, where now the right-hand is larger than it was in Lemma 4.
- **Scenario 2**: c_t is the first column of M_π and c_s is the last. Then, again, we have already considered a solution in Lemma 4 which has solution cost one less for CO and yet had the same solution cost for (1,2)-TSP Path.

This completes the proof. □

Reduction Phase 2

Given the matrix M as constructed in Phase 1 from G, we will now construct a string T as input to the problem AO. It is easier to describe T in terms of its substrings, which are created by iterating through the matrix M as follows:

- For $1 \leq j \leq n+2$, $1 \leq i \leq m+2\ell$: if $M_{i,j} = 1$ output the substring $10^{i+1}2C_j$
- For $1 \leq j \leq n+2$: output the substring $0^{m+2\ell+2}2C_j$
- Append to each substring created above a unique $\$_i$ symbol ($1 \leq i \leq 2m+2\ell+n+2$).

The string T is the concatenation of these substrings in any order and $|T| = O(n^2)$. The alphabet set Σ is $\{0,1,2\} \cup \{C_1, C_2, \ldots, C_{n+2}\} \cup \{\$_1, \$_2, \ldots, \$_{2m+2\ell+n+2}\}$ and $\sigma = \Theta(n)$.

Given a solution π to this instance of AO we use the relative ordering given to the C_i symbols as the ordering for the columns of M_π. For the analysis of why this works, we define some properties that we would like $BWT(T)$ and π to have. For any symbol $a \in \Sigma$ we will call the maximal set of indices where the F column of the sorted circular shift matrix has only a's as the *a-block*. Our goal will be to 'simulate' the linearization of $L(M_\pi)$ within the 0-block of $BWT(T)$. We let C_s and C_t denote the symbols for columns c_s and c_t respectively.

The following are the key properties that an optimal solution π^* will have:

1. For a fixed j, all C_j symbols are placed adjacently in $BWT(T)$;
2. All 2 symbols are placed adjacently in $BWT(T)$;
3. The symbol 2 is adjacent to the symbol 0 in the ordering;
4. The $\$_i$ symbols are ordered in such a way as to minimize the number of runs of 1 in the 0-block of $BWT(T)$.
5. The symbols C_s and C_t are both positioned at the beginning and end respectively of the alphabet ordering given to the C_i symbols.

The 0-block of $BWT(T)$ will consist of 0's, 1's, and $\$_i$ symbols. All $\$_i$ symbols will be adjacent within the 0-block. This is since the $\$_i$ symbols succeeded by 0, are all succeeded by the substring $0^{m+2\ell+2}2$ and every occurrence of $0^{m+2\ell+2}2$ preceded by a $\$_i$ symbol (when T is viewed as a circular string). Let r_0 denote the number of runs created in the 0-block of $BWT(T)$, minus the number of $\$_i$ symbols in the 0-block of $BWT(T)$.

Lemma 6. *Unless all of the above properties hold, the solution to AO is suboptimal.*

Proof. If any of Properties 1-3 are violated, we can exchange our solution with one which maintains the value r_0 but reduces the runs created in other blocks. This is since the alphabet ordering can be modified to have these properties, while at the same time maintaining the relative orderings of symbols within the 0-block. In the case of Property 4, given that Properties 1-3 hold, modifying the solution so that the property holds can only decrease r_0, while it maintains the number of runs created in other blocks. Assuming properties 1-4 hold, there are two possibilities, either C_s and C_t are extremal or they are not.

- In the case of being extremal, if $C_s < C_t$, then by Property 4, the $2\ell = 8m$ instances of 1's in the bottom 2ℓ rows of M_π shall correspond to $4m$ runs of two consecutive 1's in the 0-block of $BWT(T)$. The upper rows of M_π shall correspond to at most $2m$ runs of 1's in the 0-block of $BWT(T)$. Hence, in the 0-block there are at most $6m+1$ runs of 1's making at most $6m+2$ runs of zeros to surround them, so that $r_0 \leq 12m+3$. In the case where $C_t < C_s$, one additional run of 1's is created over the same configuration where the positions of C_s and C_t are swapped.

- In the case of them not being extremal, considering only the last 2ℓ rows of M_π, there are $8m$ runs of lonely 1's in the 0-block of $BWT(T)$, and at least $8m+1$ runs of 0's to surround them, leading to $r_0 \geq 16m+1$.

This completes the proof. □

As mentioned earlier, we aim to have a substring of $BWT(T)$ within the 0-block which is the same as $L(M_\pi)$ except for the lengths of its runs, i.e., the number of runs will be the same. We will call this substring the simulation of $L(M_\pi)$.

Lemma 7. *If all Properties 1-5 hold, then* $r_0 = \rho(L(M_\pi)) - 1$ *and* $\rho(BWT(T)) = r_0 + \sigma - 1$.

Proof. We will first show that when Properties 1-5 hold, $r_0 = \rho(L(M_\pi)) - 1$, i.e., that the simulation works. Within the 0-block of $BWT(T)$, row i is simulated by the characters preceding each substring $0^{i+1}2$. Note that they all appear consecutively in the 0-block. Within the simulation of the i^{th} row, if the value of the j^{th} column of M_π is 0, then the characters preceding substrings of the form $0^{i+1}2C_j$ are all 0. If the value of the j^{th} column of M is 1, then there exists a single substring of the form $0^{i+1}2C_j$ preceded by a 1, and the remaining substrings of the form $0^{i+1}2C_j$ are all preceded by 0. Note that all characters preceding $0^{i+1}2C_j$ are consecutive within the i^{th} row, however, the unique \$'s following each substring allow the characters following each $0^{i+1}2C_j$ to have their orders swapped. Because of Property 5, in the column ordering of M_π there will never be a run of more than two consecutive 1's in $L(M_\pi)$. Hence, when Property 4 is applied, we know that 1's which would are adjacent in $L(M_\pi)$ are adjacent in the 0-block. Combining all these observations gives us that $L(M_\pi)$ is successfully simulated within the 0-block. The '-1' term in the expression for r_0 arises due to Property 2. This is since the 0 symbol in 0-block of $BWT(T)$ that is adjacent to the 2-block does not contribute a run. We have shown $r_0 = L(M_\pi) - 1$.

Finally, the fact that $\rho(BWT(T)) = r_0 + \sigma - 1$ follows from Properties 1-3 which cause every symbol except 1 to contribute exactly one run to $\rho(BWT(T))$ outside of the simulation (1's first appearance is within the simulation). □

Lemma 8. *If all Properties 1-5 hold, the L-reduction conditions are satisfied.*

Proof. By Lemma's 6 and 7 we have the optimal cost for AO being $r_0^* + \sigma - 1$ and optimal cost for CO as $r_0^* + 1$. For Condition (i) note that $\sigma = \Theta(n)$ and because there are at most 5 runs created by each row, $m + 2\ell \leq r_0^* \leq 5(m + 2\ell)$, so that $r_0^* = \Theta(n)$. Hence, we can find an α such that $r_0^* + \sigma - 1 \leq \alpha(r_0^* + 1)$. For Condition (ii), we have $(r_0 + 1) - (r_0^* + 1) \leq \beta((r_0 + \sigma - 1) - (r_0^* + \sigma - 1))$ with $\beta = 1$. □

Lemma 9. *If any of Properties 1-5 are violated, the L-reduction conditions are satisfied.*

Proof. Condition (i) is satisfied since optimal values for the overall problem are unchanged. For Condition (ii), if any of the first four properties are violated, we have already shown in Lemma 8 that the inequality holds in the harder case where $\rho(L(M_\pi))$ has the same value but the overall number of runs in $BWT(T)$ is less. If the first four properties hold and the fifth property does not hold, there are two cases. In the first case, if C_t is ordered first and C_s last, then swapping C_s and C_t modifies both sides of the inequality for Condition (ii) by the same amount. In the second case, if either C_s or C_t are not ordered first or last, the left hand side of the inequality in Condition (ii), that is $\Big(\rho(L(M_\pi)) - \rho(L(M_{\pi^*}))\Big)$, will be large, as this corresponds to the columns c_s and c_t not being first or last. However, the right-hand side $\Big((r_0 + \sigma - 1) - (r_0^* + \sigma - 1)\Big)$ will be large as well, perhaps even larger as there may exist runs of three of four 1's in $L(M_\pi)$ that cannot be simulated in the 0-block of $BWT(T)$. In particular, $r_0 \geq \rho(L(M_\pi)) - 1$ and $\rho(L(M_{\pi^*})) = r_0^* + 1$, so that with $\beta = 1$

$$\rho(L(M_\pi)) - \rho(L(M_{\pi^*})) \leq (r_0 + 1) - \rho(L(M_{\pi^*})) \leq \beta\Big((r_0 + \sigma - 1) - (r_0^* + \sigma - 1)\Big).$$

□

We have shown an L-reduction from (1,2)-TSP Path to AO. This combined with Lemma 1 completes the proof for Theorem 1.

Proof of Corollary 1

Assuming ETH, there exists no $2^{o(n)}$ time algorithm for Hamiltonian Path Problem [36]. Our reduction allows us to determine the minimum number of paths in G needed to cover all the vertices and can hence solve Hamiltonian Path. This can be done by first constructing an incidence matrix for G and then applying the rest of the reduction as in Section 2. Since the alphabet size σ is linear in n and $|T| = \Theta(n^2)$, an $|T|^{O(1)} \cdot 2^{o(\sigma)}$ time algorithm for AO would imply an $2^{o(n)}$ time algorithm for Hamiltonian Path, a contradiction.

Constrained Alphabet Ordering

Reducing to a Simpler Problem

Recall that we wish to find an ordering on the special symbols $\$_1, \ldots, \$_d$ such that the number of runs in the BWT of $T = T_1\$_1 \ldots T_d\$_d$ is minimized and the \$ symbols are lexicographically before other symbols. We will consider our alphabet to be over integers that are bounded by $N^{O(1)}$, where $N = |T|$. Let s be an arbitrary substring of T without \$ symbols. The symbols in T which are followed by $s\$_i$ will form a contiguous portion of $BWT(T)$. However, their ordering within that contiguous portion is determined by the relative ordering given to $\$_i$ symbols. Hence, we can arrange the symbols within this portion of $BWT(T)$ so that identical symbols are placed adjacently.

For example, let $c_1 s\$_1$, $c_2 s\$_2$, ..., $c_t s\$_t$ be substrings of T. The symbols c_1, c_2, ... c_t will

be contiguous in $BWT(T)$ in some order. Now, suppose that $c_2 = c_4 = c_7$. By rearranging the $\$_2$, $\$_4$, and $\$_7$ to be adjacent within the relative ordering of the \$ symbols, we can make c_2, c_4, and c_7 appear consecutively. Taking this one step further, we can also change the relative ordering of $\$_2$, $\$_4$, and $\$_7$, so that if the substrings $\alpha c_2 s\$_2$, $\beta c_4\$_4$, and $\alpha s\$_7$ occur in T, then the two α's will be adjacent in the contiguous portion of $BWT(T)$ corresponding to the substrings $c_2 s\$_2$, $c_4 s\$_4$, and $c_7 s\$_7$.

Hence, the set of symbols $B_s = \{x \mid xs\$_i \text{ is a substring of } T \text{ for some } i \in [1, d]\}$ can be modeled as a tuple where each symbol appears only once within the tuple. Along with each symbol x in B_s, we will maintain a set $\Delta_s^x = \{\$_i \mid xs\$_i \text{ is a substring of } T\}$. We will arrange all non-empty tuples B_s in the lexicographic ordering of s. As such, these tuples can be constructed by first assigning any ordering to the \$ symbols (where they are lexicographically first in the alphabet) and then using the longest common prefix (LCP) between consecutive suffixes in lexicographic order. These values are obtained directly from the longest common prefix array. The suffix array and longest common prefix array can both be constructed in linear time assuming an integer alphabet of size $N^{O(1)}$ [44]. We will define the problem of ordering the symbols within these tuples as a new problem.

Problem 9 (Tuple Ordering (TO))**.** *Given a list of tuples $t_1, \ldots, t_q$ in a fixed order, each containing a subset of symbols from Σ, order the symbols in each tuple such that the total number of runs in the string formed by their concatenation $t_1 \cdot t_2 \cdot \ldots \cdot t_q$ is minimized (not considering '(', ')' and commas, of course).*

We will show that TO can be solved in linear time. To map solutions of TO back to solutions of CAO, a tuple for B_s needs to maintain pointers to each tuple B_{xs}, where x is a symbol. Then given a solution to TO, we start with the tuple for B_ε. The ordering given to symbols within this tuple provides us with a partial ordering on the \$ symbols. The symbols in

Δ_ε^x associated with the first symbol x within the tuple are ordered before the symbols Δ_ε^y associated with the second symbol y, etc. Then for a symbol x, the tuple for B_x provides a refinement of this partial ordering. In particular, it provides a partial ordering on Δ_ε^x. To recover the total ordering on \$ symbols, we recursively refine the partial ordering at our current tuple by examining all of the tuples which the current tuple points to. Note that this works since for a given tuple for B_s, the sets Δ_s^x are disjoint. The time required to recover this solution is proportional to N.

Solving the Tuple Ordering Problem in Linear Time

We show how to reduce the TO problem to the single-source shortest path problem on a DAG G, which is constructed as follows. For each tuple t_i, create two sets of vertices L_i and R_i, both of size $|t_i|$, such that for each symbol $c \in t_i$, there exists a vertex with label c in L_i as well as in R_i. Between each pair of vertices $u \in L_i$ and $v \in R_i$, where the label of u is not equal to the label of v, create a directed edge of weight 1 from u to v. If $|t_i| = 1$, then create a directed edge of weight 1 from the unique vertex in L_i to the unique vertex in R_i. For each R_i and L_{i+1} $(1 \leq i \leq q-1)$, and each pair $u \in R_i$ and $v \in L_{i+1}$, create a directed edge from u to v, with weight 1 if they have the same label, and weight 2 otherwise. Finally, create a start vertex s and directed edges of weight 1 from s to each vertex in L_1, and an end vertex e with directed edges of weight 1 from each vertex in R_q to e. See Figure 2.2 for an illustration.

Clearly, the shortest path from s to e is the one with the fewest edges of weight 2, and this path gives us a tuple ordering which minimizes the number of runs created by the tuples. To obtain this ordering, for a tuple t_i, place as the left-most symbol the label of the vertex used in L_i within the shortest path, and the right-most symbol the label of the vertex used

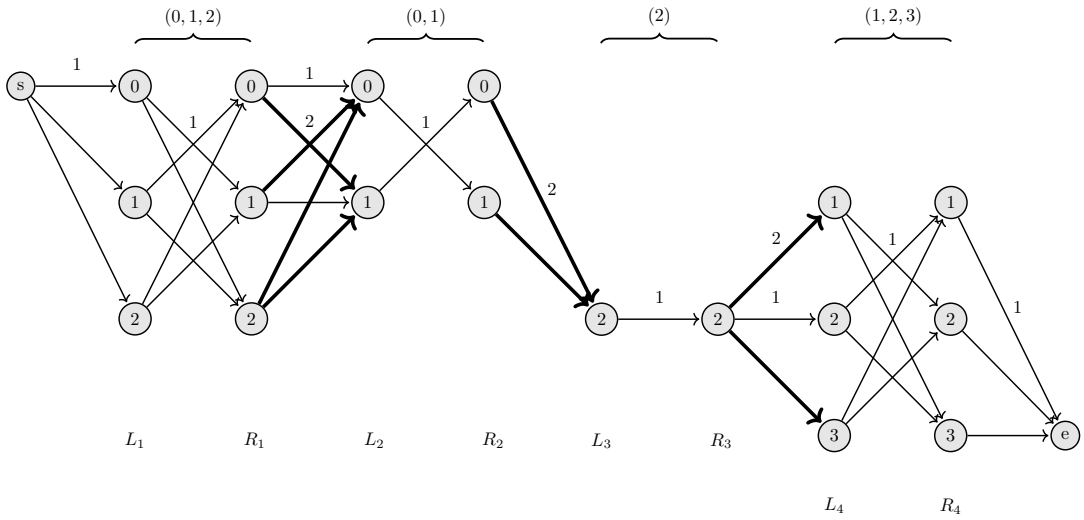


Figure 2.2: The graph G constructed for the tuple ordering instance $(0, 1, 2)$, $(0, 1)$, (2), $(1, 2, 3)$.

in R_i within the shortest path. The other symbols can be ordered arbitrarily. Because G a DAG, this shortest path can be found in time proportional to the number of edges, which is $O(\sigma^2 q)$. Next, we show how to solve this in time proportional to the number of vertices in the graph G.

Rather than constructing the edges in G, we can work from left-to-right maintaining the shortest path from s to the vertices in our current level of G, either L_i or R_i. Suppose our current level is L_i and we wish to extend the solution to the level R_i. Assuming $|t_i| \geq 2$, we identify the vertices v_1 and v_2 in L_i with the first and second shortest paths (they may have the same length) from s, respectively. For each vertex u in R_i, if the label of u is not the same as the label for v_1, we make the shortest path to u the path from s to v_1, then the edge from v_1 to u, otherwise we make it the path from s to v_2, then the edge from v_2 to u. If $|t_i| = 1$, we make the shortest path from s to u the path from s to the unique vertex v in L_i, then the edge from v to the unique vertex u. To extend a solution from R_i to L_{i+1}, we

first identify the vertex v_1 in R_i with the shortest path from s. For each vertex u in L_{i+1}, if a vertex with matching label v_u exists in R_i, we take as the shortest path to u the shorter of the following two paths: (i) the path from s to v_1, then from v_1 to u, or (ii) the path from s to v_u, then from v_u to u. If no such vertex with matching label exists in R_i, take as the shortest path from s to u the path from s to v_1, then from v_1 to u.

An Example of the Effectiveness of CAO

Lastly, we provide an example where the \$ symbol ordering greatly reduces the number of runs in the BWT. Let d be the number of strings and n the length of the strings. It is possible for a set of special symbols to be ordered such that the number of runs is $\Omega(nd)$. Let $\sigma = 2$ and $d = \sigma^n$. Consider the d distinct binary strings concatenated with special symbols in lexicographic order. Under the ordering $\$_1 < \$_2 \ ... \ < \$_d$, the string $BWT(T)$ alternates between the \$'s, 0's, and 1's, yielding $\Omega(nd)$ runs. On the other hand, for this same case, arranging the \$'s in the optimal ordering will give $O(d)$ runs in total. This is since for any substring s of T, the contiguous section of $BWT(T)$ containing the characters preceding $s\$_i$ for $i \in [1, d]$ contains at most the start of two runs. For example, with $n = 3$, we would have $T = 000\$_1001\$_2010\$_3011\$_4100\$_5101\$_6110\$_7111\$_8$. The number of runs in $BWT(T)$ under the naive ordering $\$_1 < \$_2 < \ldots < \$_8$, is 32 with $BWT(T) = 010101010101\$_8\$_101\$_2\$_3010101\$_4\$_501\$_6\$_7$. The number of runs using an optimal ordering $\$_3 < \$_5 < \$_2 < \$_7 < \$_4 < \$_6 < \$_1 < \$_8$ is 19 with $BWT(T) = 00001111110001\$_8\$_101\$_2\$_3001110\$_4\$_501\$_6\$_7$.

CHAPTER 3: WHEELER GRAPH RECOGNITION

The work presented here first appeared in the 28th Annual European Symposium on Algorithms, ESA 2019 [58].

NP-completeness of Wheeler Graph Recognition

We first show a simple reduction from the Betweenness problem to Wheeler Graph Recognition. Although straightforward, it requires graphs with either $O(n)$ sources or $O(n)$ edges with the same label leaving a single vertex. In Section 3, by expanding on the techniques used in the first reduction we show that even if these quantities are limited to at most five the recognition problem remains NP-complete.

The Betweenness Problem

The Betweenness problem is an ordering constraint satisification problem first established as NP-complete by Opatrný in 1979 [116]. Like our problem, it deals with finding a total ordering on a set of elements. The input to the Betweenness problem is a list of distinct elements $T = t_1, \ldots, t_n$ and a collection of $k < n^3$ ordered triples of $(t_1^1, t_2^1, t_3^1), (t_1^2, t_2^2, t_3^2), \ldots (t_1^k, t_2^k, t_3^k)$ where every element in a triple is in T. The elements in the list T should be ordered so that the middle element in each triple appears somewhere between the other two elements in that triple. The elements in each triple are not required to be consecutive in the ordering. The decision problem is determining whether such an ordering exists.

As an example, consider the input $T = 1, 2, 3, 4, 5, 6$, and triples $(5, 2, 3)$, $(1, 5, 2)$, $(4, 5, 6)$,

$(4, 6, 2)$. An ordering that satisfies the given triples is 1, 4, 5, 6, 2, 3. An ordering that does not satisfy the given triples is 1, 2, 3, 4, 5, 6 since it violates the triples $(5, 2, 3)$, $(1, 5, 2)$, and $(4, 6, 2)$.

Reduction from Betweenness to Wheeler Graph Recognition

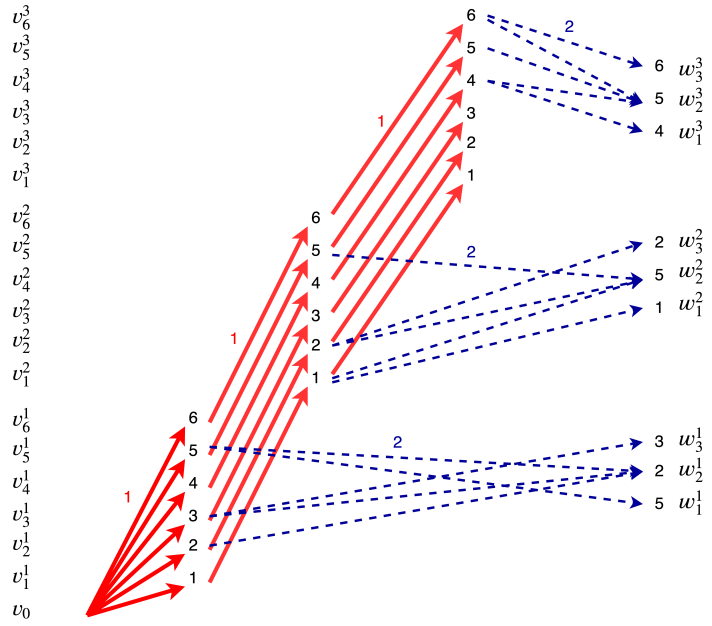


Figure 3.1: An example of the reduction with input list 1, 2, 3, 4, 5, 6 and the triples $(5, 2, 3)$, $(1, 5, 2)$, $(4, 5, 6)$.

Suppose we are given as input to the Betweenness problem the list $t_1, t_2, \ldots, t_n$, and triples $(t_1^1, t_2^1, t_3^1), (t_1^2, t_2^2, t_3^2), \ldots, (t_1^k, t_2^k, t_3^k)$. We construct a DAG of size $O(nk)$ as follows:

- Create a source vertex v_0 and vertices v_i^j for $1 \leq i \leq n$ and $1 \leq j \leq k$.
- For each triple (t_1^j, t_2^j, t_3^j), create a vertex for each element of the triple, we call them w_1^j, w_2^j, and w_3^j respectively.

- Create the edges $(v_0, v_i^1, 1)$ and edges $(v_i^j, v_i^{j+1}, 1)$ for $1 \leq i \leq n,\ 1 \leq j \leq k-1$.
- Create the following edges for $1 \leq i \leq n,\ 1 \leq j \leq k$:
  - $(v_i^j, w_1^j, 2)$ and $(v_i^j, w_2^j, 2)$ if $t_i = t_1^j$;
  - $(v_i^j, w_2^j, 2)$ if $t_i = t_2^j$;
  - $(v_i^j, w_3^j, 2)$ and $(v_i^j, w_2^j, 2)$ if $t_i = t_3^j$.

The intuition is that the vertices with inbound red (solid) edges labeled 1 represent the permutation of the elements in T. The vertices with the inbound blue (dashed) edges labeled 2 represent the elements in each triple. Consider the graph formed by the reduction as laid out in the same fashion as shown in Figure 3.1. The ordering is obtained from the layout as follows: vertices with inbound red edges are ordered from bottom-to-top, followed by the vertices with inbound blue edges also ordered from bottom-to-top. In the layout, any red edges crossing red edges, or blue edges crossing blue edges, correspond to forbidden configurations. Hence, in a proper ordering, these crossings should not occur.

Lemma 10. *An instance of the Betweenness problem has an ordering satisfying all of the constraints iff the graph constructed as above is a Wheeler graph.*

Proof. The edges labeled 1 force the ordering given to v_i^1 to be repeated k times, once for each constraint. Otherwise, the second inequality in the Wheeler graph axioms is violated. Similarly, the edges with label 2 enforce that the only valid orderings of the vertices representing elements in T are orderings that satisfy the betweenness constraints. In particular, a monochromatic rainbow of edges with label 2 is avoided iff for the j^{th} constraint (t_1^j, t_2^j, t_3^j), the given vertex ordering has $v_{i_2}^j$ ordered between $v_{i_1}^j$ and $v_{i_3}^j$, where $t_{i_1} = t_1^j$, $t_{i_2} = t_2^j$, and $t_{i_3} = t_3^j$. □

The last statement in the proof can be observed in Figure 3.1 where the top-most betweenness constraint gadget has the vertices for 4, 5, and 6 in a constraint satisfying order. One can easily check that reversing the positions of the vertices for 4 and 6 will avoid dashed blue edges crossing dashed blue edges, avoiding a monochromatic rainbow when the bottom-to-top ordering described earlier is applied. However, any order where 5 is not between 4 and 6 will not satisfy this property. Theorem 3 then follows directly from Lemma 10.

NP-completeness of Wheeler Graph Recognition on d-NFAs

Now we restrict the number of edges with the same label that can leave a single vertex. We adopt the terminology used by Alanko *et al.,* and consider the problem of recognizing whether a d-NFA is also a Wheeler graph [6]. A d-NFA is defined as follows:

Definition 2. *A d-NFA G is an NFA where the number of edges with the same character leaving a vertex is at most d. We refer to the value d as the non-determinism of G.*

Here an NFA contains a single start state, from which we assume each vertex is reachable.

The results in this section are in contrast to the recent work of Alanko *et al.,* who showed that it can be recognized in polynomial time whether a 2-NFA is a Wheeler graph [6]. Their result coupled with the observation that the reduction in Section 3 requires a $n^{\Theta(1)}$-NFA suggests an interesting question about what role non-determinism plays in the tractability of Wheeler graph recognition. To this end, we prove Theorem 15.

Theorem 15. *The Wheeler Graph Recognition Problem is NP-complete for d-NFA's, $d \geq 5$.*

The strategy of the proof will be to reduce the NP-complete problem 4-NAESAT to Wheeler Graph Recognition. In 4-NAESAT each clause is of length 4, and an instance is satisfiable iff

there exists a truth assignment such that each clause contains both a true literal and a false literal. We start with 4-NAESAT to obtain a 3-NAESAT instance with the special property highlighted by Lemma 11.

Lemma 11. *An instance ϕ of 4-NAESAT can be reduced in poly-time to an instance ϕ' of 3-NAESAT where a variable occurring in the middle of a clause appears at most twice in ϕ'.*

Proof. Convert the 4-NAESAT instance ϕ to a 3-NAESAT instance ϕ' by converting each clause (a_k, b_k, c_k, d_k) into the clauses (a_k, w_k, b_k) and $(c_k, \overline{w_k}, d_k)$ where w_k is a new variable. One can quickly check that both clauses have a satisfying not-all-equal assignment iff it is not the case that $a_k = b_k = c_k = d_k$. We also note that the variable used in the middle of the clauses, w_k, is used only twice in all of ϕ'. □

For convenience, we define the set of 3-NAESAT instances where any variable occurring in the middle of a clause occurs at most twice in the whole Boolean expression as 3-NAESAT*. We next describe the construction of a one source DAG from an instance of 3-NAESAT*. Suppose we are given an instance ϕ of 3-NAESAT* with variables $x_1, x_2, \ldots, x_n$ and the clauses (a_k, b_k, c_k) where we assume a_k, b_k, c_k can represent either a Boolean variable or its negation. We create a single source DAG G based on ϕ. The first step creates a *menorah like* structure which allows for the vertices representing x_i and $\overline{x_i}$ to swap places in G, but otherwise fixes the positions of the vertices. We begin by adding the vertices which represent our variables, $x_1, \ldots, x_n, X, \overline{x_1}, \ldots, \overline{x_n}$; (the role of X will become clear). Next, we add the structure to constrain their possible positions (see Figure 3.2 for an example).

Add to G the vertices:

- $s_1^0 \ldots, s_n^0$;

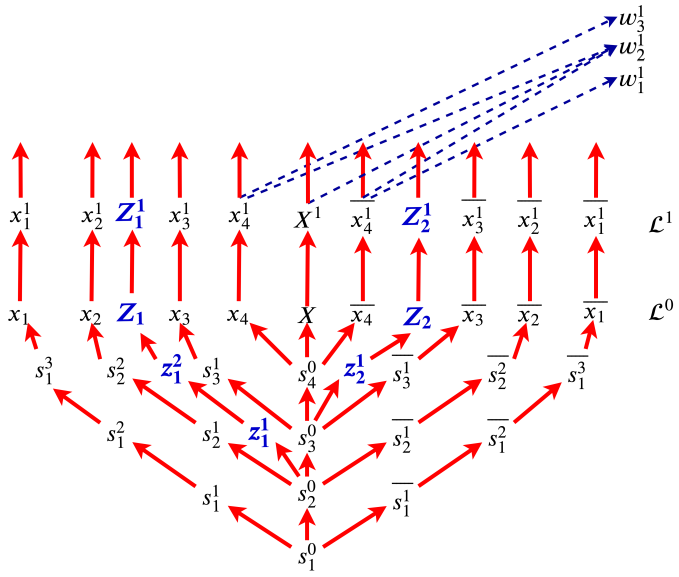


Figure 3.2: Vertex Z_1 and Z_2 could be for clauses (x_1, x_2, x_3), $(x_2, \overline{x_3}, x_4)$. Each 'betweenness' constraint adds a layer. $(x_4, X, \overline{x_4})$ constraint shown.

- For $1 \leq i \leq n-1$, $1 \leq j \leq n-i$: s_i^j and $\overline{s_i^j}$;

Add to G the red (solid) edges:

- $(s_1^0, s_2^0, 1), \ldots (s_n^0, X, 1)$;
- For $1 \leq i \leq n-1$, $1 \leq j \leq n-i$: $(s_i^{j-1}, s_i^j, 1)$ and $(\overline{s_i^{j-1}}, \overline{s_i^j}, 1)$;
- For $1 \leq i \leq n$: $(s_i^{n-i}, x_i, 1)$ and $(\overline{s_i^{n-i}}, \overline{x_i}, 1)$;

For clause k, denoted (a_k, b_k, c_k), we add a vertex Z_k. Suppose the middle variable of the clause, b_k, is x_h (positive or negated), then we add the vertices z_k^j for $1 \leq j \leq n-h$, and red edges $(s_h^0, z_k^1, 1), (z_k^1, z_k^2, 1) \ldots (z_k^{n-h}, Z_k, 1)$.

Now we wish add a set of *betweenness* type constraints on any proper ordering given of the vertices $\mathcal{L}^0 = \{x_1, \ldots, X, \overline{x_n} \ldots \overline{x_1}, Z_1, Z_2, \ldots\}$. Given a constraint (y_1, y_2, y_3) we insist y_2 be between y_1 and y_3 in the ordering. This can be done by adding a layer of new vertices $\mathcal{L}^1 = \{x_1^1, \ldots, X^1, \overline{x_n^1} \ldots \overline{x_1^1}, Z_1^1, Z_2^1, \ldots\}$ with red(solid) edges labeled 1 from vertices in layer $\mathcal{L}^0$ to their corresponding vertices in $\mathcal{L}^1$. We use the same gadget that was used in Section 3. Consider adding a betweenness constraint on the vertices y_1, y_2, y_3 in $\mathcal{L}^1$. Add the vertices w_1^1, w_2^1, and w_3^1 and the blue(dash) edges $(y_1^1, w_1^1, 2)$, $(y_2^1, w_2^1, 2)$, $(y_3^1, w_3^1, 2)$, $(y_1^1, w_2^1, 2)$ and $(y_3^1, w_2^1, 2)$. Additional betweenness constraints can be similarly enforced by adding a new layer $\mathcal{L}^2$ on top of $\mathcal{L}^1$ with a new gadget. We add the betweenness constraints $(x_i, X, \overline{x_i})$ for $1 \leq i \leq n$ fixing X, and then betweenness constraints (a_k, Z_k, b_k) and (c_k, X, Z_k) for every clause (a_k, b_k, c_k).

Before proving the correctness of the reduction, we make the observation that because any variable occurring in the middle of a clause occurs as most **twice** in the whole expression, the maximum number of edges leaving a vertex s_i^0 is bounded by $3 + \mathbf{2} = 5$. All of the other vertices have at most three edges with the same label leaving them.

Lemma 12. *The leveled graph G constructed as above from an instance ϕ of 3-NAESAT* is a Wheeler graph iff ϕ is satisfiable.*

Proof. Given a truth assignment that satisfies the 3-NAESAT* instance ϕ, put the vertices in $\mathcal{L}^0$ whose variables are assigned the value T on the left side of X (as in Figure 3.2), and the vertices whose variables are assigned F on the right side of X. For example, if $x_1 = T$ and $x_2 = F$, the two left-most vertex on level $\mathcal{L}^0$ would be x_1 followed by $\overline{x_2}$. For all of the possible not-all-equal arrangements of the variables for a_k, b_k, and c_k, relative to X, we will always be able to find a place in ordering for Z_k that respects the betweenness constraints. For instance, if the variable for b_k is x_h, this is possible because Z_k is able to 'freely pivot' around

Table 3.1: Possible relative orderings of a_k, b_k, c_k, Z_k, X subject to (a_k, Z_k, b_k) and (c_k, X, Z_k).

Possible Orderings (a_k has variable x_j and c_k has variable x_h)		
$a_k b_k c_k$	$j < h$	$h < j$
FFT	$c_k \dots X \dots b_k, Z_k \dots a_k$	$c_k \dots X \dots b_k, Z_k \dots a_k$
FTF	$b_k, Z_k \dots X \dots c_k \dots a_k$	$b_k, Z_k \dots X \dots a_k \dots c_k$
TFF	$a_k \dots \overline{b_k}, Z_k \dots X \dots b_k \dots c_k$	$a_k \dots \overline{b_k}, Z_k \dots X \dots b_k \dots c_k$
FTT	$c_k \dots b_k \dots X \dots \overline{b_k}, Z_k \dots a_k$	$c_k \dots b_k \dots X \dots \overline{b_k}, Z_k \dots a_k$
TFT	$a_k \dots c_k \dots X \dots Z_k, b_k$	$c_k \dots a_k \dots X \dots Z_k, b_k$
TTF	$a_k \dots Z_k, b_k \dots X \dots c_k$	$a_k \dots Z_k, b_k \dots X \dots c_k$

Table 3.2: Orderings implied by all-equal assignment that are impossible while satisfying constraints.

Impossible Orderings (a_k has variable x_j and c_k has variable x_h)		
$a_k b_k c_k$	$j < h$	$h < j$
TTT	$a_k \dots b_k \dots c_k \dots X$	$c_k \dots b_k \dots a_k \dots X$
FFF	$X \dots c_k \dots b_k \dots a_k$	$X \dots a_k \dots b_k \dots c_k$

the vertex s_h in the spine of menorah structure to find betweenness constraint respecting position immediately to left or right of x_h or $\overline{x_h}$. This can be confirmed by examining all possible cases, as is shown in Table 3.1. That is, for clause (a_k, b_k, c_k) Table 3.1 shows all possible not-all-equal truth assignments, and the corresponding relative orderings of $\mathcal{L}^0$ that we can apply to the vertices to satisfy the Wheeler graph axioms.

In the other direction, assume G is a Wheeler graph so we have a proper ordering on the vertices of G. The proper ordering of the menorah structure is fixed with the exception of z_i^j vertices and the ordering duplicated across layers $\mathcal{L}^0, \mathcal{L}^1, \dots$. We will show that in a proper ordering of the vertices the ordering given to $\mathcal{L}^0$ must have every clause in ϕ getting a not-all-equal assignment when we apply the following map: vertices for variable on the left of X in $\mathcal{L}^0$ map back to T assignment for that variable, and vertices for a variable to the

right of X in $\mathcal{L}^0$ map back to an F assignment for that variable.

Suppose to the contrary that this mapping did not provide a valid not-all-equal assignment. Then $\mathcal{L}^0$ was given an ordering where either the variables for a_k, b_k, and c_k are all on the left or the right side of X. The possible arrangements for this are presented in Table 3.2. In contrast to the cases listed in Table 3.1, for all cases listed in Table 3.2, placing Z_k between a_k and b_k violates the constraint (c_k, X, Z_k), which by our reduction implies it violates a Wheeler graph constraint as well. This contradicts our assumption that we have a proper ordering on the vertices. We conclude that a proper ordering of the vertices of G must map back to a truth assignment that gives each clause in ϕ a not-all-equal assignment. □

This leaves open the complexity of the recognition problem for 3-NFA's and 4-NFA's.

Wheeler graphs and Queue Number

Queue Number

The concept of queue number and queue layout were introduced by Heath and Rosenberg originally for undirected graphs in [68] and later expanded to directed graphs in [67]. Consider the vertices of the graph given a total ordering. We will say that we can process the edges using k queues if we can iterate through the vertices in the given ordering, and every time the tail of an edge is encountered that edge is enqueued in one of the queues, and when the head of that same edge is encountered that edge is then dequeued. Over all possible orderings of the vertices, there is some ordering that requires the minimum number of queues to perform this processing. That minimum number of queues is called the queue number of the graph. The problem of detecting whether a graph is a one-queue DAG was shown to be

solvable in linear time by Heath and Pemmaraju [66, 67, 68]. Using a few additional steps, we can extend their techniques to a specific subset of Wheeler graphs.

Theorem 16. *The Wheeler graph recognition problem can be solved in linear time for an edge alphabet of size $\sigma = 1$ on graphs without self-loops.*

Proof. When $\sigma = 1$ and the graph has no self-loops, any proper Wheeler ordering is either a topological ordering or a reversed topological ordering. Hence, the problem of finding a one-queue ordering and a proper Wheeler ordering are almost equivalent. The only difference is that for a proper Wheeler ordering all of the vertices with in-degree zero must be placed first. We can overcome this difference and reduce our problem to detecting one-queue DAGs. Let $V_0 \subset V$ represent all vertices in V with in-degree zero. Create a new vertex u with in-degree zero and add an edge from u to each vertex in V_0. Since a valid one-queue ordering is a topological ordering, v_0 must be first in the one-queue ordering. Moreover, any vertices in the $V - V_0$ must be in the one-queue ordering after the last position given to a vertex in V_0, otherwise a rainbow is created. Thus, the above modification ensures that only one-queue orderings on V place the vertices in V_0 before any vertices in $V - V_0$, so that we also obtain a proper Wheeler ordering. □

We can use results on the queue number of undirected simple graphs to obtain an upper bound on the number of edges which can be in a Wheeler graph. An undirected simple graph has queue number q if there exists an ordering on the vertices where we can process them with q queues so that when an edge is first encountered it is enqueued, and when that edge is encountered again it is dequeued, i.e., no rainbows for edges in the same queue.

Theorem 17. *The number of edges in a Wheeler graph is at most $3\sigma n - \sigma(2\sigma + 1)$.*

Proof. The number of edges in a undirected graph with queue number q is at most $2qn -$

$q(2q+1)$ [40]. By removing self-loops and the edge orientations, the Wheeler graph becomes an undirected graph with queue number at most σ. Each label adds at most n self-loops, contributing in total at most σn additional edges. □

An Exponential Time Algorithm

We can apply the encoding introduced by Gagie *et al.* [50] to develop exponential time algorithms to solve all of the problems presented in this paper. The idea is to enumerate over all possible encodings of Wheeler graphs with the proper number of vertices, edges, and labels, checking whether the encoding is isomorphic with the given graph. This idea exploits the fact that having such a space-efficient encoding also implies having a limited search space of Wheeler graphs, and that graph isomorphism can be checked in sub-exponential time. This proves Theorem 4.

Before describing the algorithm that proves Theorem 4 we need to describe the encoding of a Wheeler graph given in [50]. A Wheeler graph can be completely specified by three bit vectors. Two bit vectors I and O both of length $e + n$ and a bit vector L of length $e \log \sigma$. We assume that the vertices of the Wheeler graph G are listed in a proper ordering $x_1 <_\pi x_2 <_\pi \ldots <_\pi x_n$. The array I is of the form $0^{\ell_1}10^{\ell_2}1 \ldots 0^{\ell_n}1$ and O is of the form $0^{k_1}10^{k_2}1 \ldots 0^{k_n}1$. Here ℓ_i is the out-degree of x_i whereas k_i is the in-degree of x_i. The array L indicates which character symbol is assigned to each edge. Specifically, the i^{th} character in L gives us the label of the edge corresponding to the i^{th} zero in O. In [50] an additional C array is added, and these arrays are equipped with additional rank and select structures to allow for efficient traversal as is done in the FM-index [47]. For our purposes, however, the arrays O, I, and L are adequate.

The outline of the algorithm is given below as Algorithm 1. It essentially enumerates all bit vectors of a given length, checks whether or not the bit vector encodes a valid Wheeler graph, and if so then checks whether the encoding matches our given graph G. Let S represent the set of all possible encodings we wish to check. Note that $|S| \leq 2^{2(e+n)+e\log\sigma}$.

Algorithm 1 IdentifyWheelerGraph(G)

```
for all (O, I, L) ∈ S do
    if (O, I, L) defines a valid wheeler graph G' then
        convert G to undirected graph α(G)
        convert G' to undirected graph α(G')
        if α(G) and α(G') are isomorphic then
            return "Wheeler Graph"
        end if
    end if
end for
return "Not a Wheeler Graph"
```

The Wheeler graph corresponding to the encoding can be extracted by working from right to left reading the array I. For each zero in I, we know which symbol should be on the inbound edge going into the corresponding vertex. We only need to decide where the edge's tail was. Let k be the edge label and j be the index of the label k in L that is furthest to the right in L and yet to be used. If no such j exists we reject the encoding. When assigning the tail for an edge, take as the tail the vertex x_i where $i = rank_1(O, select_0(O, j))$. We call the graph constructed in this way G'.

We now wish to check whether G' and G are the same graphs, only with a reordering of the vertices, that is G' is the result of applying an isomorphism to G. Unlike the typical isomorphism for labeled graphs, where a bijection between the symbols on the edge alphabet is all that is required, here we wish for the adjacency and the label on the edge to be preserved in the mapping between G and G'. Specifically, we wish to know if there exists a bijective function $f : V(G) \to V(G')$, such that if $u, v \in V(G)$ are adjacent via an edge (u, v, k) with

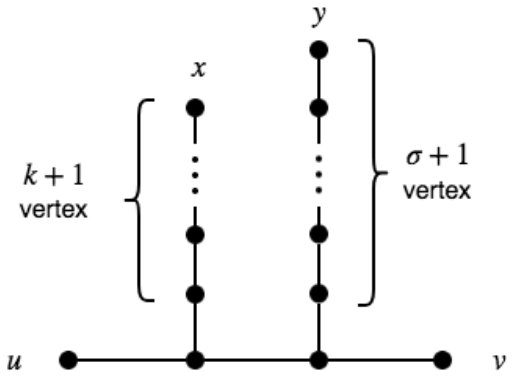


Figure 3.3: A k-gadget replacing directed labeled edge (u, v, k).

label k in G, then $f(u)$ and $f(v)$ are also adjacent via an edge $(f(u), f(v), k)$ with label k in G'. Using ideas similar to those presented by Miller in [106], this problem can be reduced in polynomial time to checking whether two undirected graphs are isomorphic.

Lemma 13. *Checking whether the direct edge labeled graph G' is edge label preserving isomorphic to G can be reduced in polynomial time to checking if two undirected graphs are isomorphic.*

Proof. Define the transformation α from directed edge labeled graph G to undirected graph $\alpha(G)$ as follows: For every directed edge (u, v, k) replace it with the k-gadget in Figure 3.3.

Assume that there exists an edge label preserving isomorphism f from $V(G)$ to $V(G')$. This implies that when α is applied to G' the same gadget is used to replace the edge $(f(u), f(v), k)$ as the gadget used to replace the edge (u, v, k) in G. Therefore, the function f can be naturally extended to an isomorphism $\tilde{f}$ on the vertices of $\alpha(G)$ providing an isomorphism between $\alpha(G)$ and $\alpha(G')$. Now, consider the case where g is an isomorphism between $\alpha(G)$ and $\alpha(G')$. We wish to show that G and G' must be related by an isomorphism preserving edge labels. We define a n-tuple of numbers for each vertex $v \in V(\alpha(G))$, $\beta(v) =$

$(a_1, a_2, \ldots, a_n)$ where a_i is the number of vertices with graph distance (the number of edges) i from v. In Figure 3.3, $\beta(x) = (1, 1, \ldots, 1, 2, \ldots)$ where the leading 1's are repeated $k+1$ times. Notice first that $\beta(v) = \beta(g(v))$, that is $\beta(v)$ is invariant under g. Also, $\beta(y) = (1, 1, \ldots, 1, 2, \ldots)$ where the leading 1's are repeated $\sigma + 1$ times. For example, when $k = 1$, we have $\beta(y) = (1, 1, 2, \ldots)$. Observe that for any vertex $u \in V(G)$ of degree d we have that $\beta(\alpha(u)) = (d, 2d, \ldots)$. It follows that any vertex which is an x vertex of a k-gadget is mapped by g onto an x vertex of a k-gadget. Similarly, any vertex which is a y vertex of a k-gadget is mapped by g onto a y vertex of a k-gadget. Hence, k-gadgets are mapped by g onto k-gadgets. This also implies that vertices in $V(\alpha(G))$ originally in G are mapped by g onto vertices in $V(\alpha(G'))$ which were originally in $V(G')$. If we restrict g to only the vertices originally in $V(G)$, then g provides us with an isomorphism between G and G'. The reduction clearly takes polynomial time. □

The final step in this algorithm is to check whether $\alpha(G)$ and $\alpha(G')$ are isomorphic. Using well established techniques this can be done in time $2^{\sqrt{n'}+O(1)}$ where n' is the number of vertices in $\alpha(G)$ [9]. The total time complexity of Algorithm 1 is the number of bit strings tested, multiplied by the time it takes to (1) validate whether the bit string encodes a Wheeler graph G' and decode it, (2) convert G and G' to undirected graphs $\alpha(G)$ and $\alpha(G')$, and (3) test whether $\alpha(G)$ and $\alpha(G')$ are isomorphic. This yields an overall time complexity of $|S|n^{O(1)}2^{\sqrt{n+2e(\sigma+1)}+O(1)}$, i.e., $2^{e \log \sigma + O(n+e)}$ for Algorithm 1.

Optimization Variants to Wheeler Graph Recognition

The Wheeler Graph Violation Problem is APX-hard

In this section, we show that obtaining an approximate solution to the WGV problem that comes within some constant factor of the optimal solution is NP-hard. We do this through a reduction that shows that WGV is at least as hard as solving the Minimum Feedback Arc Set problem (FAS). FAS in its original formulation is phrased in terms of a directed graph where the objective is to find the minimum number of edges that need to be removed in order to make the directed graph a DAG. A slightly different formulation proves more useful for us. Letting $F_\pi = \{(v_i, v_j) \in E \mid \pi(v_i) > \pi(v_j)\}$ we have the following:

Lemma 14 (Younger [127])**.** *Determining a minimum feedback arc set for $G = (V, E)$ is equivalent to finding an ordering π on V for which $|F_\pi|$ is minimized.*

From this, we can present the equivalent formulation of FAS.

Definition 3 (Minimum Feedback Arc Set (FAS))**.** *The input is a list $T = t_1 t_2 \dots t_n$ of n numbers and a set of k inequalities of the form $t_i < t_j$. This task is to compute an ordering π on T so that the number of inequalities violated is minimized.*

Interestingly, we could not have used FAS for proving that the Wheeler graph recognition problem is NP-complete, as FAS is fixed-parameter tractable in terms of the size of the feedback arc set [26]. Indeed, setting the size of the feedback arc-set to zero is equivalent to checking if the given graph is a DAG and the problem becomes solvable in linear time.

On the other hand, it has been shown that FAS is APX-hard, meaning that every problem in APX is reducible to it [75]. It also implies, assuming NP $\neq$ P, that there is a constant

$C \geq 1$ such that there is no polynomial time algorithm which provides a C-approximation. The reduction provided in this section implies:

Theorem 18. *The WGV problem is APX-hard.*

In addition, Guruswami *et al.* demonstrated that assuming the Unique Games Conjecture holds, and NP $\neq$ P, there is no constant $C \geq 1$ such that a polynomial-time algorithm's approximate solution to FAS is always a factor C from the optimal solution. We state this as a lemma.

Lemma 15 (Guruswami *et al.* [62])**.** *Conditioned on the Unique Games Conjecture, for every $C \geq 1$, it is NP-hard to find a C-approximation to FAS.*

An approximation preserving reduction from FAS to WGV combined with Lemma 15 proves the other main result of this section, Theorem 5.

The Reduction of FAS to WGV

Let $T = t_1, t_2, \ldots, t_n$ and inequalities $t_1^1 < t_2^1, t_1^2 < t_2^2, \ldots, t_1^k < t_2^k$ be the input to FAS.

We define a *heavy edge* between the vertices u and v with label ℓ as $k+1$ subdivided edges between u and v each with label ℓ. That is, a heavy edge between u and v with label ℓ consists of the edges (u, w_i, ℓ) and (w_i, v, ℓ) for $1 \leq i \leq k+1$. See Figure 3.4 for an illustration. We use the following steps to create a graph (which is a DAG):

- Create a vertex v_0 and vertices v_i^j for $1 \leq i \leq n+1$ and $1 \leq j \leq k$.
- For each inequality $t_1^j < t_2^j$ create a vertex for both t_1^j and t_2^j, labeled w_1^j and w_2^j, respectively.

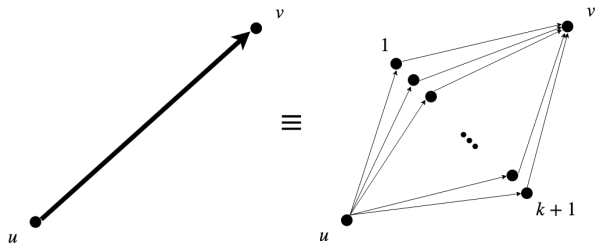


Figure 3.4: A heavy(bold) edge in Figure 3.5 is actually $k+1$ subdivided edges.

- Create heavy edges $(v_0, v_i^1, 1)$ for $1 \leq i \leq n+1$ and heavy edges $(v_i^j, v_i^{j+1}, 1)$ for $1 \leq i \leq n+1,\ 1 \leq j \leq k-1$.
- Create heavy edges $(v_0, w_1^1, 2)$, and heavy edges $(v_{n+1}^j, w_2^j, 2)$ and $(v_{n+1}^j, w_1^{j+1}, 2)$ for $1 \leq j \leq k-1$, and heavy edge $(v_{n+1}^k, w_2^k, 2)$.
- Add the regular (not heavy) edges $(v_i^j, w_1^j, 2)$ if $t_i = t_1^j$, and $(v_i^j, w_2^j, 2)$ if $t_i = t_2^j$ for $1 \leq i \leq n$, $1 \leq j \leq k$.

An example of the reduction is given in Figure 3.5. The intuition is that the vertices with an inbound heavy edge labeled 1 represent the permutation of the elements in T. The heavy edges labeled 1 force the permutation to be duplicated k times, once for each constraint. The vertices with the inbound edge label 2 represent the elements in each inequality. We will show that this is an approximation preserving reduction.

Let E' be an optimal solution to WGV and $G' = (V, E \backslash E')$. Let π represent a proper ordering on the vertices of G'. Lemma 16 indicates that, other than permuting the ordering found on the vertices v_i^j for $1 \leq i \leq n$ (with the ordering duplicated for $1 \leq j \leq k$), the ordering for the vertices in Figure 3.5 is fixed.

We say an edge (u, v, k) violates a Wheeler graph axiom if any the following hold:

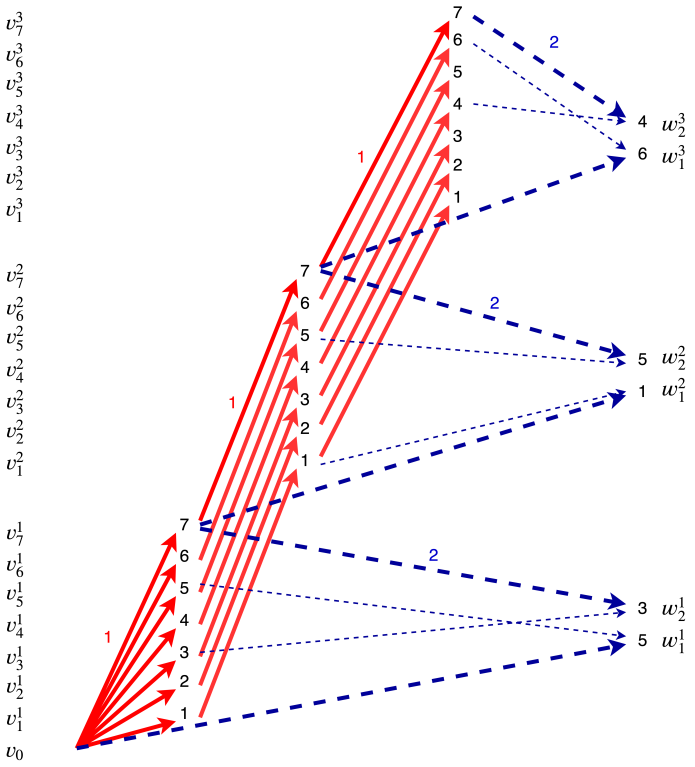


Figure 3.5: Reduction from FAS to WGV where $T = 1, 2, 3, 4, 5, 6$ and the inequalities are $5 < 3$, $1 < 5$, and $6 < 4$.

1. there exists an edge (u', v', k') with $k < k'$ and $v \geq_\pi v'$;
2. there exists an edge (u', v', k') with $k = k'$ and $u <_\pi u'$ and $v' < v$;
3. the in-degree of u is zero and there exist $w \in V$ with in-degree one or greater and $w <_\pi u$.

Lemma 16. *Let ϕ represent a permutation of the set $[n+1]$. Any ordering π which provides a solution to the constructed instance of WGV with at most k edges violating the axioms is of the form*

$$v_0, v^1_{\phi(1)}, v^1_{\phi(2)}, \ldots v^1_{\phi(n+1)}, \ldots v^k_{\phi(1)}, v^k_{\phi(2)}, \ldots v^k_{\phi(n+1)}, w^1_1, w^1_2, w^2_1, w^2_2, \ldots w^k_1, w^k_2.$$

Proof. The ordering given in Figure 3.5 causes at most k edges to violate, so we know that $|E'| \leq k$. If any of the w vertices is placed before a v vertex in π that causes $k+1$ edges to violate (1), implying $|E'| \geq k+1$, a contradiction. Similarly, v_0 must be placed first in the ordering.

For the v vertices, a v^j vertex must precede a v^{j+1} vertex in π, for $j \geq 1$. Suppose otherwise for the sake of contradiction. Take the lowest ordered such v_i^{j+1} that is preceding a v^j vertex. If v_t^j follows v_i^{j+1} in the ordering, then the heavy edge $(v_i^j, v_i^{j+1}, 1)$ violates (2) due to the edge $(v_t^{j-1}, v_t^j, 1)$ when $j \geq 2$ and $(v_0, v_t^j, 1)$ when $j = 1$, since $v_t^{j-1} <_\pi v_i^j$ and $v_i^{j+1} <_\pi v_t^j$. This causes $k+1$ violations, a contradiction. The same ordering that was found on the vertices $v_1^j, v_2^j, \ldots v_{n+1}^j$ must be duplicated across the vertices $v_1^{j+1}, v_2^{j+1}, \ldots v_{n+1}^{j+1}$. Again for the sake of contradiction, suppose otherwise and take the lowest ordered vertex v_i^{j+1} in the second group which violates the ordering of the first. Supposing, v_t^j is element preceding v_i^j in the ordering, then the heavy edge $(v_t^j, v_t^{j+1}, 1)$ violates (2) due to edge $(v_i^j, v_i^{j+1}, 1)$ since $v_t^j <_\pi v_i^j$ and $v_i^{j+1} <_\pi v_t^{j+1}$. This creates $k+1$ violations.

For the w vertices, the vertex w_1^1 must be ordered first in the w block, else $(v_0, w_1^1, 2)$ and $(v_{n+1}^1, w_2^1, 2)$ cause $k+1$ violations. The vertex w_2^1 must precede w_1^2, else the heavy edge $(v_{n+1}^1, w_2^1, 2)$ and edge $(v_{i_1}^2, w_1^2, 2)$ where $t_{i_1} = t_1^2$ cause $k+1$ violations since $v_{n+1}^1 <_\pi v_{i_1}^2$ but $w_1^2 <_\pi w_2^1$. The vertex v_{n+1}^1 must proceed the vertex $v_{i_2}^1$ where $t_{i_2} = t_2^1$. Otherwise the edges $(v_{n+1}^1, w_1^2, 2)$ and $(v_{i_2}^1, w_2^1, 2)$ cause $k+1$ violations since $v_{n+1}^1 <_\pi v_{i_2}^1$ but $w_2^1 <_\pi w_1^2$. We inductively proceed up to w_1^k and w_2^k. □

Let $f(x)$ refer to the reduction described above applied to an instance x of FAS, creating an instance $f(x)$ of WGV. We also refer to the solution to either of these problems as OPT$(\cdot)$, and $val(\cdot)$ as the cost of a solution. For FAS, $val(\cdot)$ is the number of violated inequalities. For WGV, $val(\cdot)$ is the number of violating edges.

Lemma 17. *Given an instance x of FAS, a solution to the instance $f(x)$ of WGV that has $\ell \leq k$ axiom violating edges yields a solution to x with ℓ violated inequalities.*

Proof. Suppose we have a solution to y' for WGV instance $f(x)$ with $val(y') = \ell \leq k$. By Lemma 16 the ordering of the vertices only differs from the expected ordering by the ordering given to $v_1^j, \ldots, v_n^j, v_{n+1}^j$. Ignore the vertex v_{n+1}^j and apply the remaining ordering to T. Any edge that has to be removed is one of the two edges $(v_{i_1}^j, w_1^j, 2)$ and $(v_{i_2}^j, w_2^j, 2)$, where $t_{i_1} = t_1^j$ and $t_{i_2} = t_2^j$, and where $v_{i_2}^j <_\pi v_{i_1}^j$ and $w_1^j <_\pi w_2^j$. This implies for our solution to x the j^{th} inequality has $t_{i_2} < t_{i_1}$, so we do not satisfy the inequality $t_{i_1} < t_{i_2}$. On the other hand, if it holds for the edges $(v_{i_1}^j, w_1^j, 2)$ and $(v_{i_2}^j, w_2^j, 2)$ that $v_{i_1}^j <_\pi v_{i_2}^j$, this implies the inequality is satisfied. □

The next lemma is an immediate consequence of Lemma 17.

Lemma 18. *Given an instance x of FAS, a C-approximation to the solution OPT($f(x)$) yields a C-approximation to the solution OPT(x).*

Theorem 18 follows from Lemma 18 and Theorem 5 follows from Lemma 15 and Lemma 18.

The Wheeler Subgraph Problem is in APX

The dual problem to WGV is the problem of finding the largest subgraph of G which is a Wheeler graph. This problem is called Wheeler Subgraph problem, abbreviated WS. Unlike WGV, this problem yields a $\Theta(1)$-approximate solution for constant σ.

We first prove the result for $\sigma = 1$. We then apply this result to get an approximation for $\sigma > 1$. The proof for $\sigma = 1$ uses a branching of a directed graph. A branching is a set of

arborescence where an arborescence is a directed, rooted tree with all maximal paths starting at the root. A branching is called spanning if every vertex in V is included in exactly one arborescence in the branching.

Lemma 19. *There exists a linear time $\Theta(1)$-approximation algorithm for WS when the alphabet size σ is one.*

Proof. Remove all singleton vertex (vertex with in-degree and out-degree zero), and let n' be the number of remaining vertex. By doing this we know that the number of edges in the remaining graph is at least $n' - 1$. Next, remove any edges that are self-loops. Then let V_0 be set of vertices with out-degree greater than zero. There are three cases:

Case: $|V_0| \leq n'/2$: Take a branching $\mathcal{F}$ such that each vertex with in-degree greater than zero is included in some arborescence whose root is in V_0. This is always possible, as can be shown by induction on the number of vertices not in V_0. In particular, if you take a vertex u not in V_0, since there are no singleton vertex, u has in-degree greater than zero. Removing u and applying the claim as an inductive hypothesis to the remaining graph $G - \{u\}$, you get that u has some edge from a vertex in $G - \{u\}$, which can be used to add u to an arborescence whose root is in V_0. Let $|\mathcal{F}|$ denote the total number of arborescences in $\mathcal{F}$. Since $|V_0| \leq n'/2$, it follows that $|\mathcal{F}| \leq n'/2$ as well.

We create a planar leveling $(L_0, L_1, \ldots)$ of $\mathcal{F}$ by aligning all roots of the branching on level L_0 in an arbitrary order. Then set L_i to be all of the vertices which are distance i from some root in L_0. Because these are trees, we can order the vertices in levels in such a way that the leveling is planar. For our purposes say the levels increase from left to right. We claim that $\mathcal{F}$ is a Wheeler graph and that we can obtain a proper ordering π for the vertices of $\mathcal{F}$ from this leveling. This is since, starting with V_0, we can read the order the vertices on each level from the bottom to top before proceeding right to the next level.

The number of edges in $\mathcal{F}$, denoted $e(\mathcal{F})$, is equal to $n' - |\mathcal{F}|$. And since $|\mathcal{F}| \leq n'/2$, we have that $e(\mathcal{F}) \geq n'/2$. At the same time, by Theorem 17 the optimal number of edges, denoted $|E^*|$ (including the $O(n')$ self-loops we removed earlier) is $O(n')$. This makes the ratio of the optimal solution value over the branching solution value is bounded. In particular, $|E^*|/e(\mathcal{F}) \leq O(n')/(n'/2) = O(1)$. The construction of the branching, the planar leveling, and the extracting π can all be done in linear time.

Case $|V_0| > n'/2$: Take one outbound edge from each vertex in V_0. This is possible by the definition of V_0. We obtain a Wheeler graph with $|V_0| > n'/2$ edges. This gives us a solution with an approximation ratio of $|E^*|/|V_0| < O(n')/(n'/2) = O(1)$. In either case, we have an approximate solution with $\Theta(|E^*|)$ edges. □

Next, we consider when $\sigma > 1$. Suppose $G^* = (V, E^*)$ is the optimal solution for G. Then $E^* = E_1^* \cup E_2^* \ldots E_\sigma^*$ where $E_k^* = \{(u, v, k) \in E^*\}$. Let $G_k = (V, E_k)$ where $E_k = \{(u, v, k) \in E\}$ and let $G'_k = (V, E'_k)$ be the optimal solution for G_k. Then, since $|E_k^*| \leq |E'_k|$ we have

$$|E^*| = \sum_{k=1}^{\sigma} |E_k^*| \leq \sigma \cdot \max_k |E_k^*| \leq \sigma \cdot \max_k |E'_k|.$$

Applying the result for $\sigma = 1$ (Lemma 19), we can approximate $\max_k |E'_k|$ with a solution having $\alpha \cdot \max_k |E'_k|$ edges for some constant $\alpha \leq 1$. Therefore,

$$\frac{\alpha}{\sigma}|E^*| \leq \alpha \max_k |E'_k| \leq \max_k |E'_k| \leq |E^*|.$$

So the solution provides $\Omega(1/\sigma)$-approximation for G as well.

We close this section by noting that the algorithm presented in Section 3 also provides us with an exponential time solution to the two optimization problems defined in Section 3. The

solution is to iterate over all possible subsets of edges in E, take the corresponding induced subgraph, and apply Algorithm 1 to identify if the induced subgraph is isomorphic to a Wheeler graph. For both the WGV and WS problems the optimal solution is the encoding with the fewest edges removed. The resulting time complexity is the same as in Theorem 4 with the addition of one e term in the exponent. We have shown the following:

A Class of Graphs with Linear Time Solution for Recognition

As mentioned before, in [6] it was shown by Alanko *et al.* that that exists an algorithm that solves the recognition problem on 2-NFA's using linear time. This works by reducing the recognition problem to a 2-SAT instance that can then be efficiently solved. The most immediate modification to the reduction that allows for higher non-determinism fails to ensure that a solution obtained from the 2-SAT instance is a proper ordering. Here we allow for arbitrary levels of non-determinism but place two rather stringent conditions on the graphs so that our techniques will work. It is also important to note that the motivation of pattern matching on graphs is not particularly well suited for this class of graphs (one of the main motivations of the work of Alanko *et al.*). We will see that these the graphs can be easily converted into equivalent (from the pattern matching perspective) DFA's which are trees and hence also Wheeler graphs. Instead, we take the view point that these are ordering problems, where the edges of the graph form a type of constraint and the vertices need to be ordered in a way as to satisfy these constraints. The below characteristics make this ordering problem solvable in polynomial time.

First, our graph G must have at least one vertex with in-degree zero. We let W_0 denote the set of vertices with in-degree zero. Next, we make two definitions which describe the characteristics we require in order to solve the problem efficiently.

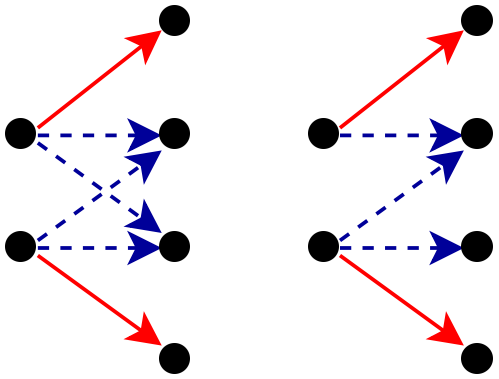


Figure 3.6: On the left is an example of a small graph that has full spectrum outputs and the unique string traversal property, but is not a Wheeler graph. On the right is an example of a small graph that has both properties and is a Wheeler graph.

Definition 4. *We consider a graph G to have* full spectrum outputs *if for every vertex v of out-degree greater than zero, every label appears on an edge leaving from v.*

Definition 5. *A graph G has the* unique string traversal property *if for every vertex v, all paths from W_0 to v form the same string when the path's edge labels are concatenated.*

In Figure 3.6 we see a simple example of two graphs that satisfy both of the given definitions, however one is a Wheeler graph and the other is not. Further, it can be seen from the reductions used in this work that even when graphs satisfy the unique string traversal property, the problem remains NP-hard. We leave open whether the problem is NP-hard when restricted to instances that have full spectrum outputs but not the unique string traversal property.

These two characteristics make this problem tractable using techniques similar to those used to detect one-queue DAGs. Based on this, we provide a linear time solution for this special case. Before presenting the solution, we introduce an essential data structure, and the process by which it is used detect whether a DAG has queue number one.

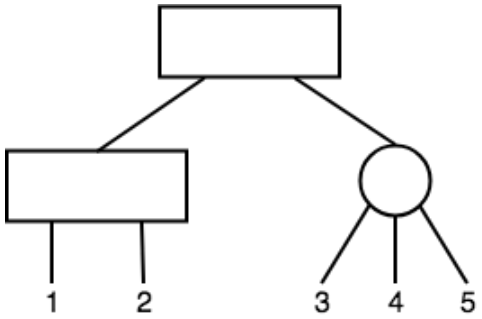


Figure 3.7: In the figure, p-nodes are represented by circles and q-nodes by rectangles. The orderings represented by this PQ-Tree are orderings where 1 can be reversed with 2, the leaves 3,4, and 5 can be permuted arbitrarily, and the order of the sets of leaves 1,2 and 3,4 5, can be swapped.

PQ-trees

PQ-trees were introduced by Booth and Lueker for the purpose of solving the consecutive ones problem [17], and have since found applications in a wide range of problems including planarity detection, detecting interval graphs, and graph embedding [17, 28, 64, 74, 91]. PQ-trees represent a set of possible orderings of the leaves which are subject to certain constraints. These constraints specify that some subset of the leaves must be contiguous in the ordering. The trees are made up of three types of nodes, p-nodes, q-nodes, and leaves. The p-nodes allow for arbitrary permutations of their child nodes, whereas q-nodes only allow for the reversal of their child nodes. The leaves represent the actual elements whose ordering we are interested in. See Figure 3.7 for an example.

A universal PQ-tree is a p-node v where all of the leaves are v's children. The ϵ-tree, T_ϵ is a special tree which represents the empty set of orderings. We can take the intersection of two PQ-trees in time proportional to the sum of the two tree sizes [17]. The resulting PQ-tree represents the intersection of the orderings represented by each PQ-tree. Deletion of a leaf

can be done in constant time.

Detecting One-Queue DAGs

The problem of detecting whether a directed graph has queue number one can be solved in linear time, but the solution is non-trivial. We give a rough idea of how the algorithm works here. Details of the algorithm are given in [66, 67]. It begins by taking a leveling $V_1, \ldots, V_k$ of the vertices in the DAG. Starting with a universal PQ-tree whose leaves are the vertices in V_1, it then makes the leaves of the PQ-tree to be the vertices in V_2 according to adjacency, with possible duplicates. Then the leaves that should be in correspondence (duplicate leaves) in V_2 are merged into the same leaf. If at any point the merging step fails, we obtain the ϵ-tree and conclude that the DAG does not have queue number one. If we get to the final level without a merging step returning the ϵ-tree, the DAG has queue number one. For convenience, we will call the combined steps of transforming the leaves and merging the leaves from one level to the next *pushing*. The intuition behind this procedure is that when the level-k has been pushed to, the PQ-tree captures all possible orderings of V_k such that a one queue layout of levels one through k is possible. This interpretation of the process is very useful for understanding the algorithm presented next.

Linear Time Solution

The basic approach to solving this problem is to use a depth-first search, treating sets of vertices as a single vertex. These vertex sets will have PQ-trees pushed across them in a similar fashion as was done in solution described above. The situation is slightly more complicated here as we have multiple edge types. This results in a tree structure, rather than a path of vertex sets. We will label the vertices representing vertex sets with capital

letters and label the PQ-tree for a vertex set $W \subset V$ as T_W.

We split the algorithm into two parts. The first part is to create a tree where vertex sets play the role of vertices. It is a depth-first search using the edges between neighborhoods as connecting edges. The pseudocode is given in Algorithm 2. $N_k(W)$ denotes the set of neighbors of the set W connected by an edge with label k. The function CREATEVERTEX takes a set of vertices and creates a new instance of a vertex class that can maintain pointers to its parent, children, internal vertices, and a string. Lemma 20 can be proven by applying induction to the number of edge labels, σ.

Algorithm 2 CreateNeighborhoodGraph

Require: Vertex set W with adjacency information

```
1: function CreateNeighborhoodGraph(W):
2:     for all k ∈ [σ] do
3:         if N_k(W) ≠ ∅ then
4:             W_k ← createVertex(N_k(W))
5:             W_k.parent ← W
6:             W_k.string ← k ∘ W.string                    ▷ Concatenate
7:             W.children.add(CreateNeighborhoodGraph(W_k))
8:         end if
9:     end for
10:    return W
11: end function
```

Lemma 20. *If the given graph G is a Wheeler graph, in a proper ordering, the vertex sets obtained as above are ordered by the lexicographical ordering of their strings.*

An example of a tree obtained from Algorithm 2 is shown in Figure 3.9. The vertex sets are disjoint due to the unique string traversal property. During Algorithm 2 we can identify if the graph satisfies the unique string traversal property by checking that vertex in V gets included into exactly one vertex set.

Moving forward, the next portion of the algorithm is a recursive procedure that starts with

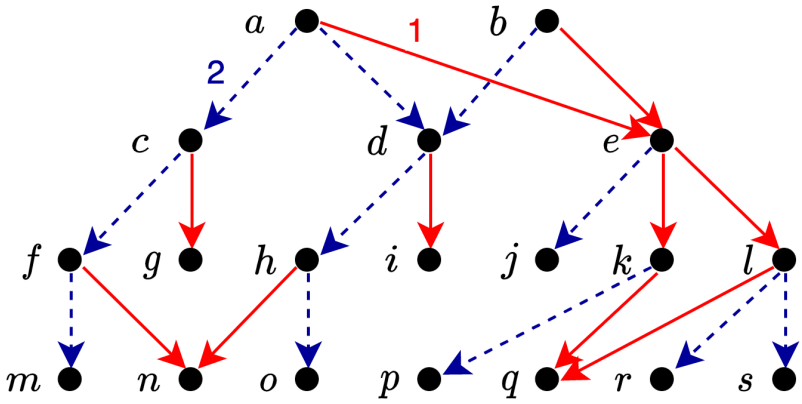


Figure 3.8: An example Wheeler graph that meets the criteria for this section. Red (solid) edges correspond to edges labeled 1, and blue (dashed) edges correspond to edges labeled 2.

the set of vertices having in-degree zero. Pseudocode is given in Algorithm 3. The first step removes vertices in W with out-degree zero. This is necessary since when we push a PQ-tree back up to W, these vertices will not be leaves in the resulting PQ-tree, making computing the intersection in future steps impossible. Hence, from here we consider W as containing no out-degree zero vertex.

We next demonstrates how Algorithm 3 works. Let V' be the vertices processed prior to reaching W and assume inductively that the PQ-tree T_W represents all orderings of W such that if we fixed any one of these orderings there still exists a proper ordering of the vertices in V'. Then after performing the first line of the for-loop, the PQ-tree T_{W_1} represents all orderings of W_1 such that if we fixed any one of these orderings there still exists a proper ordering of the vertices in $V' \cup W$. After performing the second line in the for-loop, T_{W_1} now represents all orderings of W_1 such that if we fixed any one of these orderings there still exists a proper ordering of the vertices in $V' \cup W$ and vertices that are descendants of W_1. After completing the third line in the loop, T_W represents all orderings of W such that if we fixed any one of them there still exists a proper ordering of the vertices in $V' \cup W_1$

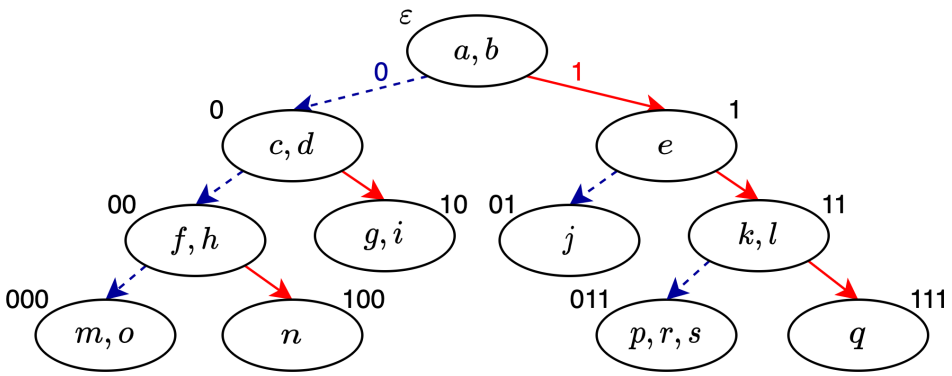


Figure 3.9: The tree resulting from Algorithm 2 applied to the Wheeler graph in Figure 3.8. An oval in the tree corresponds to a set of vertices in the Wheeler graph. The labels for these vertices are shown inside each oval. For each set of vertices inside an oval, the strings obtained by concatenating the edge labels on the path from the source is the same. These strings are shown to the side of each oval within the tree. In the tree, the edge colors indicated which type of edge was taken at each step along a path to that set.

and any descendants of W_1. We repeat this process for each of W's children. When finally returned, T_W represents all orderings of W such that there exists working orderings on V' and all descendants of W. The pseudocode for the whole algorithm is given in Algorithm 4.

The full spectrum output condition is necessary to apply this algorithm. We need that every vertex in W maps onto some vertex in each of W's children. Thanks to this property when the PQ-tree T_{W_i} gets pushed back from a child W_i and the new PQ-tree T_W is created, all the vertices in W are also leaves in T_W, and we can take the intersection with the previous PQ-tree for W.

Time Complexity: Each set of edges between two vertex sets has PQ-trees pushed across it twice. These pushes can be done in time proportional to the number of edges. In addition, all intersections can be done in time proportional to the number of vertices. As a result of these two facts, the overall algorithm can be performed in linear time. We have demonstrated

Algorithm 3 Propagating PQ-Trees

Require: PQ-Tree T_W and corresponding vertex set W.

function PROPAGATEPQTREES(T_W, W):
    Remove out-degree zero vertex from W and corresponding leaves in T_W
    for all $W_i \in W.children$ **do**
        $T_{W_i} \leftarrow$ PUSH(T_W, W_i) ▷ Push PQ-Tree down to child.
        $T_{W_i} \leftarrow$ PROPAGATEPQTREES(T_{W_i}, W_i) ▷ Recursively apply to children
        $T_W \leftarrow T_W \cap$ PUSH(T_{W_i}, W) ▷ Push PQ-tree up from child and take intersection
    end for
    return T_W
end function

Algorithm 4 Detecting Wheeler graphs

Require: Full spectrum graph $G = (V, E)$ with unique string traversal property.

function DETECTWHEELERGRAPH(G):
    Let W_0 denote the set of all in-degree zero vertex in G
    $W_0 \leftarrow$ CREATEVERTEX(W_0)
    $W_0 \leftarrow$ CREATENEIGHBORHOODGRAPH(W_0)
    $W_0.string \leftarrow$ "ϵ"
    Construct T_{W_0}, the universal tree with leaves W_0
    if PROPAGATEPQTREES(W_0, T_{W_0}) $\neq T_\epsilon$ **then**
        return "Wheeler graph"
    else
        return "Not a Wheeler graph"
    end if
end function

the following:

Theorem 19. *It can be determined in linear time if a directed edge labeled graph with full spectrum outputs and the unique string traversal property is a Wheeler graph.*

Discussion and Open Problems

We have shown that recognizing Wheeler graphs is indeed a hard problem in general. We have also shown a special case where the recognition problem can be performed efficiently. The most important directions to expand this research appear to be in identifying more classes

of graphs where this can be done in polynomial time. We can also ask for improvements on the algorithms presented here. Specifically, we ask:

- Is the Wheeler graph recognition problem NP-complete for 3-NFA and 4-NFA?
- For which other classes of graphs can Wheeler graph recognition be done efficiently?
- Is there a fixed parameter tractable exponential time algorithm for any of the hard problems given in this paper?
- Can we provide a better approximation algorithm for the optimization variants?

Constructive answers to these questions will contribute to our knowledge on finding vertex orderings "close" to that required for a Wheeler graph. It will aid in our ability to apply BWT based indices to various structures, as well as our ability to find useful compressible subgraphs.

CHAPTER 4: LYNDON FACTOR OPTIMIZATION

The work presented here first appeared in the 38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021 [59].

Our main line of attack here is to model a more abstract class of ordering constraint satisfaction problems (OCSPs), a subject of extensive research in its own right [24, 25, 63, 65, 113, 121]. Our work shows that a solver for these Lyndon factorization problems would be powerful enough to solve difficult OCSP instances. Our results make use of strings that allow us to model different constraint satisfaction problems and thus prove our hardness results.

Preliminaries

We again denote the concatenation of the strings u and v using the '$\circ$' symbol, writing their concatenation as $u \circ v$. However, we omit '$\circ$' where the concatenation is clear from context. Throughout this chapter, we will use '$<$' and '$>$' to refer to alphabet order between symbols, the lexicographic order between strings, and the usual ordering between real numbers. Again, context will make it clear which type of order is meant. A suffix of a string T is a string v such that $T = u \circ v$ for some string u. The suffix array of a string $T[1, n]$ is a length n array where the i^{th} element is equal to the starting index of the i^{th} lexicographically smallest suffix of T. The inverse suffix array is defined as the length n array such that i^{th} element is the position of $T[i, n]$ in the suffix array, i.e., the lexicographic rank of $T[i, n]$.

The Lyndon factorization (defined in Chapter 1) of a string can be computed in linear time. This can be done using the well known Duval's algorithm [41], or by using the inverse suffix array, which can be constructed in linear time [83]. Lemma 21 makes it clear why the latter

technique works.

Lemma 21 (Theorem 2.2 [105])**.** *The starting index, i, of a suffix in T that is lexicographically smaller than any suffix starting at index $j < i$ is an index where a Lyndon factor begins.*

In other words, as we scan the inverse suffix array from left-to-right, an index i where the inverse suffix array value is smaller than any seen thus far marks the start of a Lyndon factor. Moreover, if a Lyndon factor starts at index i in T, the next Lyndon word must be this factor. We aim to use this to construct strings where the number of Lyndon factors tells us something about the number of constraints satisfied within an OCSP. The definition of an OCSP used here is less general than the one given in [61], but still sufficient for our purposes.

Definition 6. *An OCSP of arity k is specified by a set $\Lambda \subseteq S_k$ where S_k is the set of permutations of $\{1, 2, ..., k\}$. An instance of such an OCSP consists of a set of variables, $V = \{x_1, \ldots, x_n\}$, and m constraints, C_1, ..., C_m, each of which is an ordered k-tuple of V. The objective is to find a global ordering σ of V that maximizes $\sum_{i=1}^{m} \chi_\Lambda(\sigma_{|C_i})$, where $\sigma_{|C_i} \in S_k$ is the ordering of the k elements of C_i induced by the global ordering σ, and $\chi_\Lambda(\sigma_{|C_i}) = 1$ if $\sigma_{|C_i} \in \Lambda$ and 0 otherwise. If $\chi_\Lambda(\sigma_{|C_i}) = 1$, we say that C_i is satisfied.*

Note that $m \leq n!/(n-k)! \leq n^k$. Additionally, we will only consider OCSP instances where each variable appears in at least two constraints. Under this last assumption, we can relate the number of variables, n, to the number of clauses, m.

Lemma 22. *For OCSPs with arity k constraints, n variables, and m constraints, where every variable appears in at least two clauses, $n \leq \frac{k}{2}m$.*

Proof. Since every variable appears in at least two constraints,

$$2n \leq \sum_{i=1}^{n} (\text{the number of times variable } x_i \text{ appears in total}) = km.$$

□

One of the simplest OCSPs is the *Maximum Acyclic Subgraph Problem* (MAS), where $k = 2$, making constraints of the form (x_i, x_j), and where $\Lambda = \{(\begin{smallmatrix}1&2\\1&2\end{smallmatrix})\}$ (using two-line permutation notation). That is, Λ contains only the identity permutation that orders $x_i < x_j$. For example, an instance of MAS could be $V = \{x_1, x_2, x_3, x_4, x_5\}$ and $C_1 = (x_1, x_3)$, $C_2 = (x_5, x_2)$, $C_3 = (x_3, x_4)$, $C_4 = (x_2, x_1)$. An ordering σ that puts the variables in the order $x_4 < x_5 < x_3 < x_2 < x_1$ would yield $\chi_\Lambda(\sigma_{|C_1}) = \chi_\Lambda((\begin{smallmatrix}1&2\\2&1\end{smallmatrix})) = 0$, $\chi_\Lambda(\sigma_{|C_2}) = \chi_\Lambda((\begin{smallmatrix}1&2\\1&2\end{smallmatrix})) = 1$, $\chi_\Lambda(\sigma_{|C_3}) = \chi_\Lambda((\begin{smallmatrix}1&2\\2&1\end{smallmatrix})) = 0$, $\chi_\Lambda(\sigma_{|C_4}) = \chi_\Lambda((\begin{smallmatrix}1&2\\1&2\end{smallmatrix})) = 1$, making its objective value 2.

The dual minimization problem of MAS is the *Feedback Arc Set* (FAS). Recall the aim of FAS is to minimize the objective value of a solution, which is defined as the number of constraints being violated, i.e., $m - \sum_{i=1}^{m} \chi_\Lambda(\sigma_{|C_i})$. The problem is otherwise identical. The following hardness result for FAS is used when proving Theorem 9.

Recall from Lemma 15 that conditioned on the Unique Games Conjecture [82], for every constant $C > 1$, it is NP-hard to find a C-approximation for FAS. We will use the term Unique-Games-hard here to refer to problems that, conditioned on the Unique Games conjecture, are NP-hard.

We can always assume that at least half of the constraints in an instance of MAS can be satisfied. To see this, take an arbitrary ordering of the variables. Either this ordering or its reversal must satisfy at least $m/2$ constraints. This is just a specific instance of a more general result. We can always assume our optimal solution satisfies at least $|\Lambda|m/k!$

constraints. Since the expected number of constraints satisfied by a random ordering on the variables is $|\Lambda|m/k!$, we know the maximum number of constraints satisfied by any ordering is bounded below by this quantity. It turns out, however, that finding a solution that does better than this expected value is computationally difficult. We give a simplified statement of the main result in [61], maintaining only the pertinent details for our problem.

Theorem 20 ([61])**.** *For an OCSP with arity k, for every constant $\varepsilon > 0$, it is Unique-Games-hard to find an ordering for the variables that achieves a ratio of satisfied constraints over total constraints that is at least $|\Lambda|/k! + \varepsilon$.*

Our results also make use of the *Betweenness Problem*. Using this new formalization, in this problem $k = 3$ and $\Lambda = \{\left(\begin{smallmatrix}1&2&3\\1&2&3\end{smallmatrix}\right), \left(\begin{smallmatrix}1&2&3\\3&2&1\end{smallmatrix}\right)\}$. Recall that for a constraint (x_i, x_j, x_k) to be satisfied either $x_i < x_j < x_k$ or $x_k < x_j < x_i$. For example, the ordering $x_4 < x_5 < x_3 < x_2 < x_1$ satisfies the constraint (x_1, x_2, x_5), but not the constraint (x_4, x_2, x_5). By applying Theorem 20 to the Betweenness problem, we obtain that it is Unique-Games-hard to achieve a ratio of satisfied constraints to total constraints better than $2/3! = 1/3$.

For hardness under the Exponential Time Hypothesis (ETH) [73], we will use a result by Kim and Gonçalves appearing in [84]. An Arity k Permutation CSP as defined in [84] is a OCSP where Λ consists of the identity permutations, $\Lambda = \{\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right), \left(\begin{smallmatrix}1&2\\1&2\end{smallmatrix}\right), \ldots, \left(\begin{smallmatrix}1&2&\cdots&k\\1&2&\cdots&k\end{smallmatrix}\right)\}$, and constraints *up to* arity k are allowed. This is different from our definition of OCSPs, where all constraints are of exactly arity k. The differences between these two definitions are accommodated for whenever Lemma 23 is used. In [84] the authors prove the following.

Lemma 23 ([84])**.** *Assuming ETH, there is no $2^{o(n \log n)}$-algorithm for Arity 4 Permutation CSP (and thus for Arity k Permutation CSP, $k \geq 4$).*

Hardness of Lyndon Factor Minimization

The first reduction is from the Betweenness problem to the Lyndon Factor Minimization Problem. It is used to demonstrate NP-completeness. An alternative proof can be done with a reduction from MAS. Our reasoning for choosing one over the other is we believe that the Betweenness problem provides a good initial illustration of the power of a hypothetical solver to these Lyndon factorization problems. It also provides a warm-up for the techniques used in Section 4. Moreover, we will use a reduction from MAS as a short proof to illustrate NP-completeness for the maximization problem, before introducing a more involved reduction to prove an inapproximability result.

NP-Completeness of Lyndon Factor Minimization

Suppose we are given as input an instance ϕ of the Betweenness problem consisting of n variables x_1, x_2, ..., x_n and m constraints C_1, C_2, ..., C_m. Let $F(T)$ denote the number of Lyndon factors of a string T under the alphabet ordering currently under consideration. We will use $F_T(T_1)$ to denote the number of Lyndon factors of T starting within the *first occurrence* of the substring T_1 of T. The subscript T is to remind us that the factors starting in T_1 are sensitive to the other symbols in T. By a *run* of a symbol, we mean a maximal unary substring containing that symbol.

Lemma 24. *Let T be any string of the form $T = T_1 \circ (x_0)^\alpha \circ (x_1^\gamma\ x_2^\gamma\ \dots\ x_n^\gamma)^\beta$ where T_1 is over the alphabet $\{x_0, \dots, x_n\}$, α is greater than the length of any run of x_0 in T_1, γ is greater than the length of any run of any symbol other than x_0 in T_1, and $\beta > 1$. If x_0 is the smallest symbol in the ordering, then $F(T) \leq F_T(T_1) + 1$.*

Proof. If T_1 does not end with an x_0, then the first x_0 in the $(x_0)^\alpha$ marks the start of a new

Lyndon factor in T since $(x_0)^\alpha$ is lexicographically smaller than any preceding suffix. Then this factor includes the remaining suffix of T. In this case $F(T) = F_T(T_1) + 1$. If T_1 contains a suffix consisting of only x_0's, then a new Lyndon factor must start at the first of these x_0's, and again this factor contains the remaining suffix of T. In this case, $F(T) = F_T(T_1)$. □

Lemma 25. *Let T be defined as in Lemma 24. If x_0 is not the smallest symbol in the ordering, $F(T) \geq \beta - 1$.*

Proof. In this case, the smallest symbol must be one of $x_1, \ldots, x_n$. Suppose the smallest is x_i. Then the first symbol in the first x_i^γ marks the beginning of a Lyndon factor. This factor is of the form $x_i^\gamma\ x_{i+1}^\gamma\ \ldots\ x_n^\gamma\ x_1^\gamma\ \ldots x_{i-1}^\gamma$ and is repeated at least $\beta - 1$ times. In particular, the suffix $x_{i+1}^\gamma\ \ldots\ x_n^\gamma$ is preceded by $\beta - 1$ factors of the form $x_i^\gamma\ x_{i+1}^\gamma\ \ldots\ x_n^\gamma\ x_1^\gamma\ \ldots\ x_{i-1}^\gamma$. □

Lemmas 24 and 25 will be useful in proving that x_0 must be smallest in an optimal ordering. We now introduce our constraint gadgets.

Lemma 26. *Let x_0 be the smallest symbol in T. For $i, j, k > 0$, consider the first instance of a substring S of T where*

$$S = x_0^\eta\ x_j\ x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_i\ x_0^\eta\ x_k\ x_0^\eta\ x_i\ x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_j\ x_0^\eta\ x_j$$

and η is larger than the length of any run of x_0 preceding S in T, and S is immediately followed by the run $x_0^{\eta+1}$. The symbols in this first instance of S make up three complete Lyndon factors if x_j is ordered between x_i and x_k, and four complete Lyndon factors otherwise.

Proof. Since the number of times x_0 is repeated is more than the length of any previous run, it must be the case that a new factor begins at the start of S. The six possible cases and their corresponding factorizations are:

$$x_0 < x_i < x_j < x_k : (x_0^\eta\ x_j), (x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_i\ x_0^\eta\ x_k), (x_0^\eta\ x_i\ x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_j\ x_0^\eta\ x_j)$$

$$x_0 < x_i < x_k < x_j : (x_0^\eta\ x_j), (x_0^\eta\ x_i\ x_0^\eta\ x_j), (x_0^\eta\ x_i\ x_0^\eta\ x_k), (x_0^\eta\ x_i\ x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_j\ x_0^\eta\ x_j)$$

$$x_0 < x_j < x_i < x_k : (x_0^\eta\ x_j\ x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_i\ x_0^\eta\ x_k\ x_0^\eta\ x_i\ x_0^\eta\ x_i), (x_0^\eta\ x_j), (x_0^\eta\ x_j), (x_0^\eta\ x_j)$$

$$x_0 < x_k < x_i < x_j : (x_0^\eta\ x_j), (x_0^\eta\ x_i\ x_0^\eta\ x_j), (x_0^\eta\ x_i), (x_0^\eta\ x_k\ x_0^\eta\ x_i\ x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_j\ x_0^\eta\ x_j)$$

$$x_0 < x_j < x_k < x_i : (x_0^\eta\ x_j\ x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_i\ x_0^\eta\ x_k\ x_0^\eta\ x_i\ x_0^\eta\ x_i), (x_0^\eta\ x_j), (x_0^\eta\ x_j), (x_0^\eta\ x_j)$$

$$x_0 < x_k < x_j < x_i : (x_0^\eta\ x_j\ x_0^\eta\ x_i), (x_0^\eta\ x_j\ x_0^\eta\ x_i), (x_0^\eta\ x_k\ x_0^\eta\ x_i\ x_0^\eta\ x_i\ x_0^\eta\ x_j\ x_0^\eta\ x_j\ x_0^\eta\ x_j)$$

Notice that only in the first and last orderings where the constraint is satisfied are there three factors. The other cases have four. □

For each constraint $C_t = (x_i, x_j, x_k)$ in the instance ϕ of the Betweenness problem, where $1 \leq t \leq m$, we construct the gadget from Lemma 26,

$$S(C_t) \coloneqq x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_j\ x_0^t\ x_j.$$

We next define $S(\phi) \coloneqq S(C_1) \circ S(C_2) \circ \ldots \circ S(C_m) \circ (x_0)^{m+1} \circ (x_1^2\ x_2^2\ \ldots\ x_n^2)^\beta$ where $\beta = 3m+3$.

Lemma 27. *The string $S(\phi)$ has an alphabet ordering yielding at most $3m+1$ Lyndon factors iff there exists a variable ordering satisfying all constraints in ϕ.*

Proof. Assuming there exists a constraint satisfying variable ordering for ϕ, make x_0 the smallest symbol and order the remaining symbols $x_1, \ldots, x_n$ according to the variable ordering. By Lemma 26, each of the substrings $S(C_t)$ for $1 \leq t \leq m$ contributes three factors, and by the analysis in Lemma 24 the remaining suffix contributes one additional factor. This creates $3m+1$ factors in total.

Conversely, assume that no variable ordering exists that satisfies the constraints. If x_0 is

the smallest symbol, then at least one $S(C_t)$ gadget contributes four factors while the others contribute at least three. The remaining suffix contributes one factor making the number of factors at least $4 + 3(m-1) + 1 = 3m + 2$. If x_0 is not the smallest symbol, then by Lemma 25, the number of factors is at least $\beta - 1 = (3m + 3) - 1 = 3m + 2$. □

Since determining if there exists a variable ordering satisfying all constraints in an instance of the Betweenness problem is NP-hard [116], determining whether there exists an alphabet order where there are at most $3m + 1$ Lyndon factors is NP-hard as well. With a symbol ordering as a polynomial sized certificate, the problem is clearly in NP, proving Theorem 7.

ETH Hardness of Lyndon Factor Minimization

Here we reduce Arity 4 Permutation CSP to Lyndon Factor Minimization. Assume for the moment that x_0 is the smallest symbol, and that each substring $S(C_t)$ (yet to be defined) is followed by a run of x_0 longer than any run of x_0 that precedes it.

For an arity 2 constraint $C_t = (x_i, x_j)$, we construct a string using the symbols x_0, x_i, and x_j that has either 3 or 4 factors depending on the ordering on the variables. We will demonstrate which orderings create which factorizations. The string we construct is $S(C_t) = x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i$, which has the factorizations for different orderings,

Ordering	Factorization	# factors
$x_i < x_j$:	$(x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_i)$	3
$x_j < x_i$:	$(x_0^t\ x_i)(x_0^t\ x_i)(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i)$	4

Slightly more involved are the strings to model arity 3 constraints $C_t = (x_i, x_j, x_k)$,

$S(C_t) = x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i.$, where

Ordering	Factorization	# factors
$x_i < x_j < x_k$:	$(x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_i)$	3
$x_i < x_k < x_j$:	$(x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_i)$	4
$x_j < x_i < x_k$:	$(x_0^t\ x_i)(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i)$	4
$x_k < x_i < x_j$:	$(x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i)$	4
$x_j < x_k < x_i$:	$(x_0^t\ x_i)(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i)$	4
$x_k < x_j < x_i$:	$(x_0^t\ x_i)(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_i)$	4

The most involved is the gadget for an arity 4 constraint $C_t = (x_i, x_j, x_k, x_h)$,
$S(C_t) = x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i$
which has the following factorizations depending on the ordering given to its symbols:

Ordering (‘<’ omitted)	Factorization	#
x_i, x_j, x_k, x_h	$(x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k)(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)$	3
x_i, x_j, x_h, x_k	$(x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k)(x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k)(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)$	4
x_i, x_k, x_j, x_h	$(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)$	4
x_i, x_h, x_j, x_k	$(x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k)(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)$	4
x_i, x_k, x_h, x_j	$(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)$	4
x_i, x_h, x_k, x_j	$(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)$	4
x_j, x_i, x_k, x_h	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)$	4
x_j, x_i, x_h, x_k	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)$	4
x_k, x_i, x_j, x_h	$(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_h, x_i, x_j, x_k	$(x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k)(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_k, x_i, x_h, x_j	$(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_h, x_i, x_k, x_j	$(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_j, x_k, x_i, x_h	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)$	4
x_j, x_h, x_i, x_k	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)$	4
x_k, x_j, x_i, x_h	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_h, x_j, x_i, x_k	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_k, x_h, x_i, x_j	$(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_h, x_k, x_i, x_j	$(x_0^t\ x_i\ x_0^t\ x_j)(x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_j, x_k, x_h, x_i	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)$	4
x_j, x_h, x_k, x_i	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)$	4
x_k, x_j, x_h, x_i	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_h, x_j, x_k, x_i	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_k, x_h, x_j, x_i	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4
x_h, x_k, x_j, x_i	$(x_0^t\ x_i)(x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)(x_0^t\ x_h\ x_0^t\ x_j\ x_0^t\ x_i\ x_0^t\ x_k\ x_0^t\ x_i\ x_0^t\ x_j\ x_0^t\ x_i)$	4

The string construction for the overall reduction is almost identical to the one for ϕ in Section 4. We only need to select β to be slightly different. We let $\beta = 4m + 3$. This is enough to ensure that in an optimal solution x_0 must be the smallest symbol. If x_0 is smallest, in the worst-case, when all constraints are not satisfied, there are at most $4m + 1$ Lyndon factors. If x_0 is not smallest, as shown in Lemma 25, the number of factors is at least $\beta - 1 = 4m + 2$. Then, with x_0 as the minimum, each ordering on $x_1, \ldots, x_n$ gives us $3s + 4(m - s) + 1 = 4m + 1 - s$ factors, where s is the number of satisfied constraints when using the corresponding variable ordering in ϕ. Therefore, an optimal ordering for the n variables of ϕ is obtained by an order on the $(n+1)$ symbols which minimizes the number of Lyndon factors in the string. This combined with Lemma 23 proves Theorem 8.

Inapproximability of Lyndon Factor Minimization

We will perform an approximation preserving reduction from FAS to Lyndon Factor Minimization. Recall that for FAS the arity k of the constraints is 2, so that constraints are of the form (x_i, x_j) and Λ consists of the identity permutation. In other words, the constraint is only satisfied if $x_i < x_j$. The cost of the solution will be the number of violated constraints, which we wish to minimize. Our gadget for constraint $C_t = (x_i, x_j)$ will be

$$S(C_t) = (x_0^t \; x_i) \circ (x_0^t \; x_j)^{\alpha - 1}$$

where $\alpha > 1$ will be chosen later. The whole string for our reduction will be

$$T = S(\phi) = S(C_1) \circ S(C_2) \circ \ldots \circ S(C_m) \circ (x_0)^{m+1} \circ (x_1^2 \; x_2^2 \; \ldots \; x_n^2)^{\beta}$$

where $\beta = \alpha m + 3$. By Lemma 25, if x_0 is not smallest, then $F(T) \geq \beta - 1$. We consider next what happens in our constraint gadgets when x_0 is smallest.

Lemma 28. *If x_0 is smallest and $x_i < x_j$ then $F_T(S(C_t)) = 1$.*

Proof. Since x_0^t is the longest run of x_0 seen so far, the start of $S(C_t)$ marks the smallest suffix seen so far when traversing T from left to right. Then, since $x_j > x_i$, the start of all substrings of the form $x_0^t\ x_j$ do not mark the start of the smallest suffix seen so far. □

Lemma 29. *If x_0 is smallest and $x_j < x_i$ then $F_T(S(C_t)) = \alpha$.*

Proof. Again, since x_0^t is the longest run of x_0 seen so far, the start of $S(C_t)$ marks the smallest suffix seen so far when traversing T from left to right. However, now the start of each substring of the form $x_0^t\ x_j$ marks the start of the smallest suffix seen so far (recall after the last $x_0^t\ x_j$ there will be a longer run of x_0 than has been seen before). Hence, there are $\alpha - 1$ additional factors created. □

Lemma 30. *Any alphabet ordering where x_0 is smallest has fewer factors than an alphabet ordering where x_0 is not the smallest.*

Proof. If x_0 is smallest, $F(T) = s + \alpha(m - s) + 1$ where s is the number of satisfied constraints and the $+1$ arises from the last factor, $(x_0)^{m+1} \circ (x_1^2\ x_2^2\ \ldots\ x_n^2)^\beta$. Because $\alpha > 1$, this is upper bounded by the case when $s = 0$ so that $F(T) \leq \alpha m + 1$. On the other hand, if x_0 is not smallest $F(T) \geq \beta - 1 = \alpha m + 2$. □

Henceforth, we only need to worry about the case when x_0 is the smallest. Our aim is to show that a constant approximation algorithm for Lyndon Factor Minimization allows us to construct a constant approximation algorithm for FAS. If our hypothetical approximation

algorithm for Lyndon Factor Minimization ever returned a solution where x_0 is not smallest, we add the additional step of replacing that solution with any solution where x_0 is smallest, obtaining a solution that performs even better. Then our modified algorithm maintains being an approximation algorithm for Lyndon Factor Minimization (perhaps with an even smaller approximation factor).

Let s_F^* denote the number of constraints satisfied in an optimal solution of ϕ for FAS and let s_L^* denote the number of constraints in ϕ satisfied by the variable ordering obtained from our optimal, factor minimizing, alphabet order for the corresponding instance of Lyndon Factor Minimization. Also, let s denote the actual number of constraints satisfied by the variable ordering obtained from our approximate factor minimizing alphabet order for the corresponding instance of Lyndon Factor Minimization. A λ-approximation for Lyndon Factor Minimization with $\lambda > 1$ gives the following set of inequalities:

$$s_L^* + \alpha(m - s_L^*) + 1 \leq s + \alpha(m - s) + 1 \leq \lambda(s_L^* + \alpha(m - s_L^*) + 1).$$

Which can be equivalently written as

$$(m - s_L^*) + \frac{s_L^* + 1}{\alpha} \leq (m - s) + \frac{s + 1}{\alpha} \leq \lambda(m - s_L^*) + \lambda\frac{s_L^* + 1}{\alpha}. \tag{4.1}$$

We will show that by taking α large enough we can ensure $s_L^* = s_F^*$.

Lemma 31. *With $\alpha = 2(m + 1) + 1$, we have that $s_L^* = s_F^*$.*

Proof. The cost of an optimal solution of ϕ is $m - s_F^*$. The solution for ϕ we get from mapping our solution for Lyndon factorization back to ϕ must have at least as many violated constraints as the optimal solution for ϕ, i.e., $m - s_L^* \geq m - s_F^*$, and so $s_F^* \geq s_L^*$. Let us suppose for the sake of contradiction that $s_F^* \geq s_L^* + 1$. This implies $m - s_L^* - (m - s_F^*) \geq 1$.

Then, using in addition that $\frac{s_F^*+1}{\alpha} \leq \frac{m+1}{\alpha} \leq \frac{1}{2}$, we obtain

$$\frac{s_F^*+1}{\alpha} - \frac{s_L^*+1}{\alpha} \leq \frac{1}{2} < 1 \leq m - s_L^* - (m - s_F^*),$$

which implies that

$$m - s_F^* + \frac{s_F^*+1}{\alpha} < m - s_L^* + \frac{s_L^*+1}{\alpha}.$$

Or, written more naturally as the cost of a Lyndon Factor Minimization Problem's solution,

$$s_F^* + \alpha(m - s_F^*) + 1 < s_L^* + \alpha(m - s_L^*) + 1.$$

But then this implies that the ordering on $x_1, \ldots, x_n$ that is used to obtain the optimal solution for ϕ creates fewer Lyndon factors than our supposedly optimal solution for Lyndon Factor Minimization, a contradiction. □

Let us now upper bound $m - s$ (our approximate solution cost when the solution is mapped back to FAS) in terms of $\lambda(m - s_F^*)$. Combining the inequalities in (4.1) with Lemma 31, and the fact that $s_F^* = s_L^* \leq m$ when $\alpha = 2(m+1)+1$, we get that

$$m - s \leq m - s + \frac{s+1}{\alpha} \leq \lambda(m - s_L^*) + \lambda\frac{s_L^*+1}{\alpha} \leq \lambda\left(m - s_F^* + \frac{1}{2}\right).$$

The case where $m = s_F^*$ can easily be solved in polynomial time, so we can consider that check added to our hypothetical solution as well. Hence, we assume $m - s_F^* \geq 1 > 1/2$ and,

$$m - s_F^* \leq m - s \leq \lambda\left(m - s_F^* + \frac{1}{2}\right) < \lambda(m - s_F^* + m - s_F^*) = 2\lambda(m - s_F^*).$$

We have shown that a λ approximation for Lyndon Factor Minimization allows us to obtain, at worst, a 2λ approximation for FAS. Moreover, the α value we need to do this is polynomial

in m so that the whole reduction is done in polynomial time. This polynomial time constant approximation algorithm is better then what is allowed by Lemma 15 under the Unique Games Conjecture. This completes the proof of Theorem 9.

Hardness of Lyndon Factor Maximization

Our approach will be similar to the one taken for minimization. First, we introduce some gadgetry for the NP-completeness proof that is later expanded upon to create an inapproximability result. As of now, we have not yet found gadgets to establish the same ETH hardness for the maximization problem.

NP-Completeness of Lyndon Factor Maximization

We perform a reduction from the dual of FAS, the Maximum Acyclic Subgraph Problem (MAS). Recall MAS is identical to FAS except for the cost of a solution now being the number of constraints satisfied, which we wish to maximize. For constraint $C_t = (x_i, x_j)$, we define our constraint gadget as $S(C_t) = x_0^{t+1}\ x_j\ x_0^{t+1}\ x_i$ (note the reversal of i and j). The entire string formed by our instance ϕ of FAS is

$$T = S(\phi) = (x_0\ x_1\ x_2\ \ldots\ x_n) \circ S(C_1) \circ S(C_2) \circ \ldots \circ S(C_m) \circ (x_0)^m.$$

Lemma 32. *If x_0 is not the smallest symbol in the ordering, then $F(T) \leq n + m$.*

Proof. Suppose $x_i \neq x_0$ is the smallest symbol. Then the first Lyndon factor starting with x_i occurs in the prefix $(x_0\ x_1\ \ldots\ x_n)$. Subsequent Lyndon factors must begin with x_i. The prefix contributes at most n factors and there are at most m remaining occurrences of x_i. □

Lemma 33. *In an ordering where x_0 is smallest, $F(T) = 2s + (m - s) + 1 + m$, where s is the number of constraints satisfied in MAS by the ordering given to $x_1, \ldots, x_n$.*

Proof. For a substring $S(C_t)$, if $C_t = (x_i, x_j)$ is not satisfied (i.e., $x_i > x_j$) then $F_T(S(C_t)) = 1$. If it is satisfied (i.e., $x_i < x_j$) then $F_T(S(C_t)) = 2$. The prefix $x_0\ x_1\ x_2\ \ldots\ x_n$ contributes exactly one additional factor. The suffix $(x_0)^m$ contributes m factors. □

Lemma 34. *Any ordering where x_0 is the smallest has more factors than an ordering where x_0 is not the smallest.*

Proof. By Lemma 22, we can assume that $n \leq m$. Then by Lemma 32, we have that if x_0 is not smallest, $F(T) \leq n + m \leq 2m$. By Lemma 33, if x_0 is smallest then $F(T) = 2s + (m - s) + 1 + m = s + 2m + 1 > 2m$. □

The value $F(T)$ is maximized by an alphabet order which has the largest possible number of satisfied constraints, say s^*. This gives $(s^* + 2m + 1)$ Lyndon factors. Clearly, this solution also provides an ordering satisfying the maximum number of constraints in our MAS instance. Since MAS is NP-hard, we have shown Lyndon Factor Maximization is NP-hard as well. The decision problem is in NP using the ordering on $x_1\ \ldots\ x_n$ as a polynomial sized certificate, and this remains NP-hard as it could be used to solve the optimization problem. This completes the proof of Theorem 10.

Inapproximability of Lyndon Factor Maximization

First, let us describe the OCSP from which we are reducing. Let $k > 1$ be the arity of the constraints, which we will specify later. Each constraint will be satisfied iff the variables in that constraint have one of the $(k - 1)!$ orderings where the last variable is ordered

first, i.e., for constraint $(x_{i_1}, x_{i_2}, \ldots, x_{i_{k-1}}, x_{i_k})$, the ordering over those variables will have $x_{i_k} < x_{i_j}$ for $j \in [1, k-1]$. More formally, $\Lambda = \{\left(\begin{smallmatrix} 1 & 2 & \ldots & k-1 & k \\ z_1 & z_2 & \ldots & z_{k-1} & 1 \end{smallmatrix}\right) \mid \cup_{i=1}^{k-1}\{z_i\} = \{2, \ldots, k\}\}$. According to Theorem 20, it is Unique-Games-Hard to find an approximation which beats $|\Lambda|m/k! = (k-1)!m/k! = m/k$ constraints being satisfied.

Our constraint gadget is of the form

$$S(C_t) = (x_0^{t+1}\ x_{i_1}) \circ (x_0^{t+1}\ x_{i_2}) \circ \ldots \circ (x_0^{t+1}\ x_{i_{k-1}}) \circ (x_0^{t+1}\ x_{i_k})^{\alpha}$$

and our overall string constructed from our instance ϕ of OCSP is

$$T := S(\phi) = (x_0\ x_1\ x_2\ \ldots\ x_n) \circ S(C_1) \circ S(C_2) \circ \ldots \circ S(C_m) \circ (x_0), \quad \text{where } \alpha = mn.$$

Lemma 35. *If x_0 is not smallest then $F(T) \leq n + m$.*

Proof. Let $x_i \neq x_0$ be the smallest symbol instead. Then the prefix $(x_0\ x_1\ x_2\ \ldots\ x_n)$ contributes at most n factors, and each remaining factor must begin with x_i. We will show that there is at most 1 factor starting in each constraint gadget. For a given constraint containing x_i, if $x_i \neq x_{i_k}$ this is immediate. On the other hand, if $x_i = x_{i_k}$ then only its first occurrence can form a smaller suffix of T than those preceding it. In more detail, since $x_0 > x_i = x_{i_k}$, we have $x_{i_k}\ (x_0^t\ x_{i_k})^{\alpha-1}x_0 < x_{i_k}\ (x_0^t\ x_{i_k})^{\alpha-2}x_0 < x_{i_k}\ (x_0^t\ x_{i_k})^{\alpha-3}x_0 < \ldots$. Note that this is the reason for the final x_0 appended to T. □

Lemma 36. *If x_0 is smallest, and in constraint $C_t = (x_{i_1}, \ldots, x_{i_k})$ the symbol x_{i_k} is smallest among $x_{i_1}\ \ldots\ x_{i_k}$, then $F_T(S(C_t)) \geq \alpha$.*

Proof. Since $x_0^{t+1}x_{i_k} < x_0^{t+1}x_{i_j}$ for $j \in [1, k-1]$, and the substring following $S(C_t)$ is either x_0^{t+2} (or the final x_0 of T), the start of **each run** of x_0 in the substring $(x_0^{t+1}x_{i_k})^{\alpha}$ marks the

start of a suffix smaller than any of those preceding it. □

Lemma 37. *If x_0 is the smallest in the ordering, then $F(T) \geq \alpha s + 1$ where s is the number of clauses in ϕ satisfied by the ordering given to $x_1, \ldots, x_n$. This is larger than the number of factors from any ordering where x_0 is not the smallest.*

Proof. By Lemma 36, when x_0 is the smallest each of the satisfied constraint gadgets contributes at least α factors. In addition, the lone x_0 symbol at the end of T forms its own factor. For the second statement, we can always assume our approximate solution satisfies at least 1 constraint, hence $s \geq 1$ and $\alpha s + 1 \geq mn + 1 > m + n$, which by Lemma 35 is an upper bound on the number of factors when x_0 is not smallest. □

From here we only need to consider when x_0 is smallest, for the same reasoning as given in Section 4. Now, suppose we have a λ-approximation with $\lambda < 1$ for Lyndon Factor Maximization. Let s_L^* be the number of constraint gadgets satisfied from our optimal solution of Lyndon factor maximization, and s the number from the approximate solution. Then,

$$\lambda(\alpha s_L^* + 1 + y_L^*) \leq \alpha s + 1 + y \leq \alpha s_L^* + 1 + y_L^*$$

where y_L^* represents the number of additional factors contributed beyond $\alpha s_L^* + 1$ and y represents the number of factors beyond $\alpha s + 1$ for our approximate solution. We can equivalently write the above expression as

$$\lambda s_L^* \left(1 + \frac{1}{\alpha s_L^*} + \frac{y_L^*}{\alpha s_L^*}\right) \leq s \left(1 + \frac{1}{\alpha s} + \frac{y}{\alpha s}\right) \leq s_L^* \left(1 + \frac{1}{\alpha s_L^*} + \frac{y_L^*}{\alpha s_L^*}\right). \tag{4.2}$$

Lemma 38. *For all $s \in [1, m]$, and for the corresponding y value as described above,*

$$1 \leq \left(1 + \frac{1}{\alpha s} + \frac{y}{\alpha s}\right) \leq 3.$$

Proof. We first bound y from above. Any factor in a constraint gadget begins at the start of a run x_0. In a satisfied constraint gadget, there are $k-1$ such runs outside of the $(x_0^{t+1} x_{i_k})^\alpha$ substring. Hence, each satisfied constraint gadget contributes at most $k-1$ additional factors beyond α. A constraint gadget that is not satisfied, i.e., has $x_{i_j} < x_{i_k}$ for some $j \neq k$, has the gadget's last factor beginning at the start of the substring $(x_0^{t+1}\ x_{i_j})$. This implies the substring $(x_0^{t+1}\ x_{i_k})^\alpha$ does not split into different factors. Therefore, an unsatisfied constraint gadget again contributes at most $k-1$ factors. Because of this, the m constraint gadgets contribute at most $k-1$ additional factors in total and $y \leq m(k-1)$. Finally, $\alpha = mn$, hence

$$\frac{y}{\alpha s} \leq \frac{y}{\alpha} \leq \frac{m(k-1)}{\alpha} \leq \frac{mn}{\alpha} = 1 \quad \text{and} \quad \frac{1}{\alpha s} \leq \frac{1}{\alpha} = \frac{1}{nm} \leq 1.$$

□

Let s_C^* be the number of constraints satisfied in an optimal solution to ϕ. Like in Section 4, we know that $s \leq s_C^*$ and $s_L^* \leq s_C^*$, Using Lemma 38 we can easily make them differ by at most a constant factor.

Lemma 39. *Using the definitions above, it holds that $s_C^* \leq 3s_L^*$.*

Proof. For the sake of contradiction, assume instead that $s_C^* > 3s_L^*$. Applying the ordering given by the optimal solution of ϕ to the symbols $x_1, \ldots, x_n$, and letting y_C^* be defined as above but for s_C^*, we have

$$s_C^* \left(1 + \frac{1}{\alpha s_C^*} + \frac{y_C^*}{\alpha s_C^*}\right) > s_C^* > 3s_L^* \geq s_L^* \left(1 + \frac{1}{\alpha s_L^*} + \frac{y_L^*}{\alpha s_L^*}\right)$$

However, this implies $\alpha s_C^* + 1 + y_C^* > \alpha s_L^* + 1 + y_L^*$. Thus, s_L^* couldn't have been the number of constraints satisfied in an optimal solution to our Lyndon Factor Maximization instance, since using whichever ordering was used for the solution to ϕ would have given us more factors, a contradiction. □

By Lemma 39, we have $\frac{1}{3}s_C^* \leq s_L^*$. Multiplying both sides by $\lambda/3$, we obtain $\frac{\lambda}{9}s_C^* \leq \frac{\lambda}{3}s_L^*$. By Lemma 38 and our starting inequality in (4.2) we also have that

$$\lambda s_L^* \leq \lambda s_L^* \left(1 + \frac{1}{\alpha s_L^*} + \frac{y_L^*}{\alpha s_L^*}\right) \leq s\left(1 + \frac{1}{\alpha s} + \frac{y}{\alpha s}\right) \leq 3s.$$

From which we obtain $\frac{\lambda}{3}s_L^* \leq s$. Combining these inequalities with the fact that $s \leq s_C^*$, we get $\frac{\lambda}{9}s_C^* \leq s \leq s_C^*$. That is, a λ-approximation algorithm for Lyndon Factor Maximization provides at least a $\lambda/9$ -approximation algorithm for this set of OCSP problems.

To finish the proof of Theorem 11, suppose for the sake of contradiction there exists a λ-approximation algorithm for Lyndon factor maximization for some constant $\lambda < 1$. Consider the set of OCSPs problems described in beginning of Section 4 with arity k such that $1/k < \lambda/9$. With our reduction, we obtain a polynomial-time algorithm that can find a solution with approximation ratio better than $|\Lambda|/k! = 1/k$, proving the Unique Games Conjecture false by Theorem 20.

Open Problems

We leave open the problem of establishing similar ETH hardness results for the maximization problem. We also leave open the problem of finding a (non-constant factor) approximation algorithm for either the minimization or maximization problem.

CHAPTER 5: OPTIMAL REFERENCE FOR RELATIVE LEMPEL-ZIV

Hardness Results

As a warm up, we begin with a simple reduction for the case where the alphabet is polynomially-sized and $\alpha = \beta = 1$. Here we reduce the Eulerian Walk problem described below to the Optimal Reference problem. These ideas are then expanded on in a reduction with a more complicated proof from the Hamiltonian path problem to the Optimal Reference problem over a binary alphabet.

Warm Up: Polynomially-Sized Alphabets

The problem of determining whether a directed graph $G = (V, E)$ has a subset of edges $E' \subseteq E$ such that $G' = (V, E')$ has a spanning Eulerian walk, i.e., one that includes every vertex and traverses every edge in E' exactly once (vertices can be visited multiple times) is NP-complete. This is through a trivial reduction from the same problem for Eulerian circuits, proven NP-complete in [37].

Reduction: To reduce from the Eulerian Walk Problem to Optimal Reference we add a string $T_{i,j} = v_i v_j$ to $\mathcal{T}$ for every directed edge $(v_i, v_j) \in E$. The following two lemmas prove the reduction's correctness.

Lemma 40. *For all reference strings R, we have $r + p \geq 2|E| + 1$.*

Proof. If $v_i v_j$ is a substring of R then $T_{i,j}$ requires only one pointer. If $v_i v_j$ is not a substring

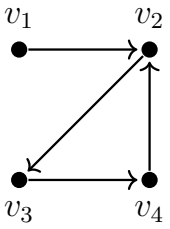
in R, then $T_{i,j}$ requires two pointers. It follows that $p = 2|E| - a$ where a is the number of substrings $v_i v_j$ in R corresponding to edges in G. At the same time, $a \leq r - 1$, making $r + p = r + 2|E| - a \geq 2|E| + 1$. □

Lemma 41. *There exists a reference string R such that $r + p = 2|E| + 1$ iff G has a subset of edges $E' \subseteq E$ where $G' = (V, E')$ has a spanning Eulerian walk.*

Proof. If there exists such a set of edges E', we can make $a = r - 1$ by listing the vertices in the order given by the Eulerian walk. This makes $r + p = r + 2|E| - a = 2|E| + 1$.

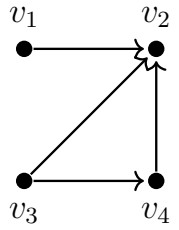
Conversely, if there exists an R such that $r + p = 2|E| + 1$, then $r + 2|E| - a = 2|E| + 1$, implies $a = r - 1$. Hence, all substrings of length two in R correspond to a distinct string $T_{i,j}$, and therefore to a distinct edge $(v_i, v_j) \in E$. Moreover, since every symbol v_i has to appear in R, the trail obtained from R has to encounter all vertices in V, making it the desired spanning Eulerian walk. □

See Figures 5.1 and 5.2 for examples.



String	Encoding
v_1v_2	$(1, 2)$
v_2v_3	$(2, 3)$
v_3v_4	$(3, 4)$
v_4v_2	$(4, 4), (2, 2)$

Figure 5.1: The graph above contains an Eulerian walk. Using reference string $R = v_1v_2v_3v_4$, we show its corresponding set of texts and their encoding. Here $r + p = 4 + 5 = 2|E| + 1$.



String	Encoding
v_1v_2	$(1,2)$
v_3v_2	$(3,3)(2,2)$
v_3v_4	$(3,4)$
v_4v_2	$(4,4),(2,2)$

Figure 5.2: The graph above does not contain an Eulerian walk. Using reference string $R = v_1v_2v_3v_4$, we show its corresponding set of texts and their encoding. Here $r + p = 4 + 6 > 2|E| + 1$. Furthermore, $r + p > 2|E| + 1$ for any reference string.

Hardness Over a Binary Alphabet

In this section we obtain a stronger hardness result via a reduction from the Hamiltonian Path problem on a directed graph $G = (V, E)$ where each vertex has a total degree of at most four. This problem is proven NP-complete in [119].

Recall the cost function $\alpha r + \beta p$. The problem is clearly trivial if $\alpha = 0$ or $\beta = 0$. We will show that the problem is NP-complete for any α and β where $1 \leq \alpha \leq \beta$, i.e., a pointer is more expensive than a new symbol in the reference. Let $x \geq 2$ be a positive integer to be fixed later. For $i \in [1, |V|]$, let $B(i)$ be the encoding of i obtained by taking a binary encoding of i on $\lceil \log |V| \rceil$ bits, inserting $x-1$ number of 0's before every bit in the encoding, then padding both sides with "11". Let $K = x\lceil \log |V| \rceil + 4$, the length of the encoding.

We construct the input $\mathcal{T}$ to the Optimal Reference problem as follows: Create an empty text collection $\mathcal{T}$. Then for all $s, t \in [1, K]$, $i \in [1, |V|]$, add $0^s B(i) 0^t$ to $\mathcal{T}$. Also, for all $(v_i, v_j) \in E$, add $0^K B(i) 0^K B(j) 0^K$ to $\mathcal{T}$.

Lemma 42. *For all $i \in [1, |V|]$, the string $B(i)$ is a substring of any optimal reference R^*.*

Proof. If $B(i)$ is not a substring of R^* then the texts $0^s B(i) 0^t$ for $s, t \in [1, K]$ each require

at least two pointers, contributing $2K^2$ to p. However, by appending $0^K B(i)0^K$ to R^* we increase r by $2K + |B(i)| = 3K$ and decrease p by at least K^2. This is due to the texts $0^s B(i)0^t$ for $s, t \in [1, K]$ now each requiring only one pointer. Since $\beta K^2 > 3\alpha K$, this reduces the cost, contradicting the optimality of R^*. □

Lemma 43. *There exists an optimal reference R^* such that for all $i \in [1, |V|]$ the string $0^K B(i)0^K$ is a substring of R^*.*

Proof. Let $\tilde{R}$ be an optimal reference not containing $0^K B(i)0^K$. By Lemma 42, we know that $B(i)$ is a substring of $\tilde{R}$. Let $s, t \in [0, K]$ be such that $s + t$ is maximized and $0^s B(i)0^t$ is a substring of $\tilde{R}$. We assume WLOG that $s = \min(s, t) < K$. Inserting $K - s$ number of 0's immediately before $0^s B(i)0^t$ in $\tilde{R}$ and $K - t$ number of 0's immediately after $0^s B(i)0^t$ in $\tilde{R}$ increases r by $2K - s - t$. The number of texts that change from requiring at least two pointers to only one is at least $K(K - s)$. We have $2K - s - t \leq 2K - 2s$. Since $\beta K(K - s) \geq \alpha(2K - 2s)$, therefore inserting these 0's in R lowers the overall cost. □

Lemma 44. *There exists an optimal reference R^* such that for all $i \in [1, |V|]$, the string $0^K B(i)0^K$ occurs in R^* exactly once.*

Proof. By Lemma 43, an optimal reference R^* contains at least one $0^K B(i)0^K$ for every $i \in [1, |V|]$. We will show that deleting any additional instances of the substring $0^K B(i)0^K$ will not increase $\alpha r + \beta p$.

Removing an additional instance of $0^K B(i)0^K$ from R^* will not change the number of pointers needed for the texts of the form $0^s B(i)0^t$, since they require only one pointer from a single instance of $0^K B(i)0^K$ in R^*.

We next show that the minimum number of pointers needed for a text $0^K B(j)0^K B(h)0^K$ where $i \neq j$ and $h \neq i$ will not change either. If the whole of the extra instance of $0^K B(i)0^K$

is being referred to, then since $0^K B(i)0^K$ is not a substring of $0^K B(j)0^K B(h)0^K$, at least two pointers are being used. We can instead use the substrings $0^K B(j)$ and $0^K B(h)0^K$ as pointers, which by Lemma 43 are substrings of R^*.

For a text $0^K B(i)0^K B(j)0^K$ or $0^K B(j)0^K B(i)0^K$, removing the additional $0^K B(i)0^K$ substring from R^* may increase the number of pointers needed from one to two. However, the number of such texts is bound by the total-degree of v_i. Hence, by completely deleting from R^* the substring $B(i)$ contained in the extra instance of $0^K B(i)0^K$, we increase p by at most the total-degree of v_i. Recall that this is bounded by four. The deletion therefore contributes an additional cost of at most 4β to the objective value. At the same time, it decreases r by $|B(i)| = K$. To contradict the optimality of R^* we need that $\alpha K > 4\beta$, which is accomplished by letting $x = \max(2, \lceil(4\frac{\beta}{\alpha} - 3)/\log|V|\rceil)$ in $K = x\lceil\log|V|\rceil + 4$. □

Lemma 45. *Consider an optimal reference R^* of the form $S_1 \circ 0^K B(\pi(1))0^K \circ S_2 \circ 0^K B(\pi(2))0^K \circ S_3 \circ \ldots \circ S_{|V|} \circ 0^K B(\pi(|V|))0^K \circ S_{|V|+1}$ for some permutation π over $1, ..., |V|$, where S_h's are arbitary strings and we allow the substrings 0^K to be merged when S_h is the empty string. Then all substrings S_h for $h \in [1, |V| + 1]$ can be removed without increasing $\alpha r + \beta p$.*

Proof. Suppose the substring S_h is needed to decrease the number of pointers to a text $0^K B(i)0^K B(j)0^K$ from two to one. However, unless one of the following cases applies, the text $0^K B(i)0^K B(j)0^K$ will still require at least two pointers:

- S_h has the substring $0^k B(i)0^K B(j)0^K$;
- S_h has the suffix $B(i)$ and is followed by $0^K B(\pi(h))0^K$ where $\pi(h) = j$;
- S_h has the prefix $B(j)$ and is preceded by $0^K B(\pi(h-1))0^K$ where $\pi(h-1) = i$.

However, by Lemma 44, we can assume R^* only contains one copy of $0^K B(i)0^K$ and $0^K B(j)0^K$.

Hence, none of the above cases can apply. This implies S_h can be deleted from R^*, decreasing r while leaving p unaltered, a contradiction. □

Lemma 45 allows us to assume that the optimal reference R^* for our collection is of the form $0^K B(\pi(1))0^K B(\pi(2))0^K \ldots 0^K B(\pi(|V|))0^K$ for some permutation π over 1, ..., $|V|$. To complete the proof of Theorem 12, let λ be the number of substrings of R^* of the form $0^K B(i)0^K B(j)0^K$ for some $(v_i, v_j) \in E$. Then the number of pointers $p = K^2|V| + \lambda + 2(|E| - \lambda)$, the length of the reference $r = 2K|V| + K$, and the total cost

$$\alpha r + \beta p = \alpha(2K|V| + K) + \beta(K^2|V| + 2|E| - \lambda).$$

This is clearly minimized when λ is maximized. Moreover, G has a Hamiltonian path iff there exists a permutation π where every substring of R^* of the form $0^K B(\pi(i))0^K B(\pi(j))0^K$ corresponds with an edge $(v_{\pi(i)}, v_{\pi(j)}) \in E$, making $\lambda = |V| - 1$ and the objective cost $\alpha(2K|V| + K) + \beta(K^2|V| + 2|E| - |V| + 1)$. This completes the proof of Theorem 12.

The following Lemma directly proves Theorem 13 and relates the problem on a collection of strings to the problem on a single string.

Lemma 46. *Let $T = T_1 \circ \ldots \circ T_n$, i.e, the concatenation of all strings in $\mathcal{T}$. Then, an optimal reference R° for T provides a 2-approximation of the optimal reference R^* for $\mathcal{T}$.*

Proof. Let OPT^* be the optimal cost for encoding $\mathcal{T}$ using R^*, and OPT° be the optimal cost for encoding T using R°. We have $OPT^\circ \leq OPT^*$ since the reference string R^* can be used on T with at most as many pointers as was needed for $\mathcal{T}$. At the same time, $OPT^* \leq \beta n + OPT^\circ$ since the encoding used for T can be used for $\mathcal{T}$ while only adding at most n more pointers, one for the beginning of each text. Finally, observing that $\beta n \leq \beta p^* \leq OPT^*$ we obtain $OPT^* \leq \beta n + OPT^\circ \leq \beta n + OPT^* \leq 2OPT^*$. □

Bounds in Terms of the δ-Measure

We first establish straight forward lower bounds on the cost of an optimal relative-LZ encoding.

Lemma 47. $\alpha r^* + \beta p^* = OPT \geq 2\sqrt{\alpha\beta}\sqrt{N}$.

Proof. The optimization problem $\min_{r,p} \alpha r + \beta p$ s.t. $rp \geq N$ can be solved using standard techniques, e.g. Lagrange multipliers. Doing so, one finds $r^* = \sqrt{\frac{\beta}{\alpha}}\sqrt{N}$ and $p^* = \sqrt{\frac{\alpha}{\beta}}\sqrt{N}$. □

Lemma 48. *For a single text T, $\delta(T) \leq \alpha r^* + \beta p^*$.*

Proof. Let T be the concatenation of the texts in $\mathcal{T}$. For any string S we use $LZ_{77}(S)$ to denote the number of phrases in the LZ77 encoding of S.

Consider decoding the LZ77 encoding of $R^* \circ T$ where R^* is the optimal relative-LZ reference for T. The substring T is decoded using pointers to the prefix R^* in addition to pointers to the already decoded prefix of T. From this we can infer that p^* must be at most the number of pointers used by LZ77 to encode T and $LZ_{77}(R^* \circ T) \leq LZ_{77}(R^*) + p^*$. Combined with properties of the δ-measure [109] we have,

$$\delta(T) \leq \delta(R^* \circ T) \leq LZ_{77}(R^* \circ T) \leq LZ_{77}(R^*) + p^* \leq \alpha r^* + \beta p^*.$$

□

Lemma 49. *For a single text T, $\alpha r^* + \beta p^* \leq 2(\alpha\delta(T))^{\frac{1}{3}}(\beta N)^{\frac{2}{3}}$.*

Proof. Let $\delta_k = \frac{d_k}{k}$ where d_k is the number of distinct substrings of T of length k. Note that

$\delta = \max_k \delta_k$. For any k, by choosing R to be the concatenation of all distinct substrings of T of length k we have $r = kd_k$ and $p \leq \frac{N}{k}$. Next,

$$\alpha r^* + \beta p^* \leq \alpha r + \beta p \leq \alpha k d_k + \beta \frac{N}{k} = \alpha k^2 \delta_k + \beta \frac{N}{k}$$

Setting $k = \left(\frac{\beta N}{\alpha \delta}\right)^{\frac{1}{3}}$ and using that $\delta_k \leq \delta$,

$$\alpha k^2 \delta_k + \beta \frac{N}{k} = \alpha^{\frac{1}{3}} \frac{\delta_k}{\delta^{\frac{2}{3}}} (\beta N)^{\frac{2}{3}} + (\alpha \delta_k)^{\frac{1}{3}} (\beta N)^{\frac{2}{3}} \leq 2(\alpha\delta)^{\frac{1}{3}} (\beta N)^{\frac{2}{3}}.$$

□

This bounds the cost of an optimal relative-LZ encoding and proves Theorem 14.

The above inequality is tight to within logarithmic factors on an average input in the following sense.

Lemma 50. *For a random binary string T, $\mathbb{E}[\delta] = \Omega(\frac{N}{\log N})$.*

Proof. Let $k = 2 \log N$, X_{ij} be 1 if $T[i..i+k-1] = T[j..j+k-1]$ and 0 otherwise, and X_i be the number of times $T[i..i+k-1]$ occurs in T. Then, $X_i = \sum_{j=1}^{N-k+1} X_{ij}$. Hence, the expected number of times a string X_i occurs in T is $\mathbb{E}[X_i] = \sum_{j=1}^{N-k+1} \mathbb{E}[X_{ij}]$. Using that $\mathcal{P}(X_{ij} = 1) = 2^{-k}$ for $j \neq i$ and 1 for $j = i$,

$$\mathbb{E}[X_i] = 1 + \frac{N-k}{2^k} = 1 + \frac{N-k}{N^2}.$$

Using Markov's inequality,

$$\mathcal{P}(X_i \geq 2) \leq \frac{1}{2} + \frac{1}{2N} - \frac{k}{2N^2} \leq \frac{1}{2} + \frac{1}{2N}.$$

Let Y_i be 1 if $T[i..i+k-1]$ occurs only once in T and 0 otherwise. Then,

$$\mathcal{P}(Y_i = 1) = 1 - \mathcal{P}(X_i \geq 2) \geq \frac{1}{2} - \frac{1}{2N}.$$

Since the number of distinct substrings of length k is at least the number substrings of length k occuring only once, $d_k \geq \sum_{i=1}^{N-k+1} Y_i$ and

$$\mathbb{E}[\delta] \geq \mathbb{E}[\delta_k] = \frac{\mathbb{E}[d_k]}{k} \geq \frac{1}{k} \sum_{i=1}^{N-k+1} \mathbb{E}[Y_i] \geq \frac{N-k+1}{k} \left(\frac{1}{2} - \frac{1}{2N} \right) = \Omega\left(\frac{N}{k} \right).$$

□

At the same time $\delta = O(N)$ since there are at most N distinct substrings of T for any length. Plugging into the inequality in Theorem 14 the lower bound on the expected value of δ, for the case with $\alpha = 1$, $\beta = \log r$, we have $OPT \in \Omega(\frac{N}{\log N}) \cap O(N \log^{\frac{2}{3}} r)$. That is, on random binary strings the inequality is tight to within logarithmic factors.

Open Problems

The main problem that this work leaves open is whether the Optimal Reference problem can be solved in polynomial time on a single string. Thanks to Theorem 13 a positive answer to this question would indicate a polynomial time 2-approximation algorithm for the general problem.

CHAPTER 6: CONCLUSION

This work has demonstrated that several novel problems arising from string algorithms and text compression are closely related to ordering constraint satisfaction problems. This was either explicitly in their formulation, or implicitly through reductions. Using these connections, we were able to establish strong hardness results for each problem.

After establishing these hardness results, we were able to derive several algorithms. In particular, for many of these problems, by restricting the set of possible inputs, adding constraints, or making reasonable conjectures, we provided polynomial time solutions or approximation algorithms.

We leave open for future research:

- For BWT-runs minimization, lifting the constraint that the $ symbols be ordered first in the constrained alphabet ordering problem while maintaining a polynomial-time algorithm;
- Finding whether Wheeler graph recognition is NP-complete for degree-three and degree-four DAGs (3-NFAs and 4-NFAs);
- Finding a wider class of graphs where Wheeler graph recognition can be performed in polynomial time;
- Establishing ETH hardness results for *minimizing* the number of Lyndon factors via alphabet ordering;
- Finding a polynomial time algorithm for the Optimal Reference problem on a single string.

LIST OF REFERENCES

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78, 2015. doi:10.1109/FOCS.2015.14.

[2] Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 375–388, 2016. doi:10.1145/2897518.2897653.

[3] Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, pages 39–51, 2014. doi:10.1007/978-3-662-43948-7_4.

[4] Jürgen Abel. Post BWT stages of the burrows-wheeler compression algorithm. *Softw., Pract. Exper.*, 40(9):751–777, 2010. doi:10.1002/spe.982.

[5] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

[6] Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 911–930, 2020. doi:10.1137/1.9781611975994.55.

[7] Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *CoRR*, abs/2002.10303, 2020. URL: `https://arxiv.org/abs/2002.10303`, `arXiv:2002.10303`.

[8] Jarno N. Alanko, Travis Gagie, Gonzalo Navarro, and Louisa Seelbach Benkner. Tunneling on wheeler graphs. In *Data Compression Conference, DCC 2019, Snowbird, UT, USA, March 26-29, 2019*, pages 122–131, 2019. `doi:10.1109/DCC.2019.00020`.

[9] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 171–183, 1983. `doi:10.1145/800061.808746`.

[10] Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 457–466, 2016. `doi:10.1109/FOCS.2016.56`.

[11] Hideo Bannai, Travis Gagie, et al. Online lz77 parsing and matching statistics with rlbwts. In *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[12] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017. `doi:10.1137/15M1011032`.

[13] Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing the bijective BWT. *CoRR*, abs/1911.06985, 2019. URL: `http://arxiv.org/abs/1911.06985`, `arXiv:1911.06985`.

[14] Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Indexing the bijective BWT. In *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, pages 17:1–17:14, 2019. doi:10.4230/LIPIcs.CPM.2019.17.

[15] Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, pages 88–100, 2010. doi:10.1007/978-3-642-13509-5_9.

[16] Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the complexity of bwt-runs minimization via alphabet reordering. In *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, pages 15:1–15:13, 2020. doi:10.4230/LIPIcs.ESA.2020.15.

[17] Kellogg Speed Booth. Pq-tree algorithms. Technical report, California Univ., Livermore (USA). Lawrence Livermore Lab., 1975.

[18] Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big bwts. *Algorithms for Molecular Biology*, 14(1):13, 2019.

[19] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 225–235, 2012. doi:10.1007/978-3-642-33122-0_18.

[20] Karl Bringmann, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). *ACM Trans. Algorithms*, 16(4):48:1–48:22, 2020. doi:10.1145/3381878.

[21] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 79–97, 2015. doi:10.1109/FOCS.2015.15.

[22] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *SRC Research Report*, 1994.

[23] Bastien Cazaux and Eric Rivals. Linking BWT and XBW via aho-corasick automaton: Applications to run-length encoding. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 24:1–24:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.24.

[24] Moses Charikar, Venkatesan Guruswami, and Rajsekar Manokaran. Every permutation CSP of arity 3 is approximation resistant. In *Proceedings of the 24th Annual IEEE Conference on Computational Complexity, CCC 2009, Paris, France, 15-18 July 2009*, pages 62–73, 2009. doi:10.1109/CCC.2009.29.

[25] Moses Charikar, Konstantin Makarychev, and Yury Makarychev. On the advantage over random for maximum acyclic subgraph. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings*, pages 625–633, 2007. doi:10.1109/FOCS.2007.47.

[26] Jianer Chen, Yang Liu, Songjian Lu, Barry O'Sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *J. ACM*, 55(5):21:1–21:19, 2008. doi:10.1145/1411509.1411511.

[27] Kuo Tsai Chen, Ralph H Fox, and Roger C Lyndon. Free differential calculus, iv. the quotient groups of the lower central series. *Annals of Mathematics*, pages 81–95, 1958.

[28] Norishige Chiba, Takao Nishizeki, Shigenobu Abe, and Takao Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *J. Comput. Syst. Sci.*, 30(1):54–76, 1985. doi:10.1016/0022-0000(85)90004-2.

[29] Amanda Clare and Jacqueline W. Daykin. Enhanced string factoring from alphabet orderings. *Inf. Process. Lett.*, 143:4–7, 2019. doi:10.1016/j.ipl.2018.10.011.

[30] Amanda Clare, Jacqueline W. Daykin, Thomas Mills, and Christine Zarges. Evolutionary search techniques for the lyndon factorization of biosequences. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 1543–1550, 2019. doi:10.1145/3319619.3326872.

[31] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015. doi:10.1016/j.is.2014.06.002.

[32] Raphaël Clifford, Allan Grønlund, Kasper Green Larsen, and Tatiana Starikovskaya. Upper and lower bounds for dynamic data structures on strings. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, pages 22:1–22:14, 2018. doi:10.4230/LIPIcs.STACS.2018.22.

[33] Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 05 2012. doi:10.1093/bioinformatics/bts173.

[34] Pierluigi Crescenzi. A short guide to approximation preserving reductions. In *Proceedings of Computational Complexity. Twelfth Annual IEEE Conference*, pages 262–273. IEEE, 1997.

[35] Maxime Crochemore and Dominique Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991. doi:10.1145/116825.116845.

[36] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. Lower bounds based on the exponential-time hypothesis. In *Parameterized Algorithms*, pages 467–521. Springer, 2015.

[37] Marek Cygan, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Ildikó Schlotter. Parameterized complexity of eulerian deletion problems. *Algorithmica*, 68(1):41–61, 2014. doi:10.1007/s00453-012-9667-x.

[38] Nicolaas Govert De Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49(49):758–764, 1946.

[39] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative lempel-ziv self-index for similar sequences. *Theor. Comput. Sci.*, 532:14–30, 2014. doi:10.1016/j.tcs.2013.07.024.

[40] Vida Dujmovic and David R. Wood. On linear layouts of graphs. *Discrete Mathematics & Theoretical Computer Science*, 6(2):339–358, 2004. URL: http://dmtcs.episciences.org/317.

[41] Jean-Pierre Duval. Génération d'une section des classes de conjugaison et arbre des mots de lyndon de longueur bornée. *Theor. Comput. Sci.*, 60:255–283, 1988. doi:10.1016/0304-3975(88)90113-2.

[42] Massimo Equi, Roberto Grossi, Veli Mäkinen, and Alexandru I. Tomescu. On the complexity of string matching for graphs. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata,*

Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece, volume 132 of *LIPIcs*, pages 55:1–55:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.55.

[43] Massimo Equi, Veli Mäkinen, and Alexandru I. Tomescu. Graphs cannot be indexed in polynomial time for sub-quadratic time string matching, unless SETH fails. In *SOFSEM 2021: Theory and Practice of Computer Science - 47th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2021, Bolzano-Bozen, Italy, January 25-29, 2021, Proceedings*, pages 608–622, 2021. doi:10.1007/978-3-030-67731-2_44.

[44] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

[45] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1):4:1–4:33, 2009. doi:10.1145/1613676.1613680.

[46] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

[47] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

[48] Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Trans. Algorithms*, 7(1):10:1–10:21, 2010. doi:10.1145/1868237.1868248.

[49] Isamu Furuya, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Lyndon factorization of grammar compressed texts revisited. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 24:1–24:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CPM.2018.24.

[50] Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for bwt-based data structures. *Theor. Comput. Sci.*, 698:67–78, 2017. doi:10.1016/j.tcs.2017.06.016.

[51] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1459–1477, 2018. doi:10.1137/1.9781611975031.96.

[52] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1), January 2020. doi:10.1145/3375890.

[53] Travis Gagie, Simon J. Puglisi, and Daniel Valenzuela. Analyzing relative lempel-ziv reference construction. In *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, pages 160–165, 2016. doi:10.1007/978-3-319-46049-9_16.

[54] Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pbwt: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms,*

SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.

[55] Raffaele Giancarlo, Giovanni Manzini, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Block sorting-based transformations on words: Beyond the magic BWT. In *Developments in Language Theory - 22nd International Conference, DLT 2018, Tokyo, Japan, September 10-14, 2018, Proceedings*, pages 1–17, 2018. doi:10.1007/978-3-319-98654-8_1.

[56] Raffaele Giancarlo, Giovanni Manzini, Giovanna Rosone, and Marinella Sciortino. A new class of searchable and provably highly compressible string transformations. In *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, pages 12:1–12:12, 2019. doi:10.4230/LIPIcs.CPM.2019.12.

[57] Daniel Gibney, Gary Hoppenworth, and Sharma V. Thankachan. Simple reductions from formula-sat to pattern matching on labeled graphs and subtree isomorphism. In *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, pages 232–242, 2021. doi:10.1137/1.9781611976496.26.

[58] Daniel Gibney and Sharma V. Thankachan. On the hardness and inapproximability of recognizing wheeler graphs. In *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, pages 51:1–51:16, 2019. doi:10.4230/LIPIcs.ESA.2019.51.

[59] Daniel Gibney and Sharma V. Thankachan. Finding an optimal alphabet ordering for lyndon factorization is hard. In *38th International Symposium on Theoretical Aspects of Computer Science, STACS 2021, March 16-19, 2021, Saarbrücken, Germany (Virtual Conference)*, pages 35:1–35:15, 2021. doi:10.4230/LIPIcs.STACS.2021.35.

[60] Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012. URL: http://arxiv.org/abs/1201.3077, arXiv:1201.3077.

[61] Venkatesan Guruswami, Johan Håstad, Rajsekar Manokaran, Prasad Raghavendra, and Moses Charikar. Beating the random ordering is hard: Every ordering CSP is approximation resistant. *SIAM J. Comput.*, 40(3):878–914, 2011. doi:10.1137/090756144.

[62] Venkatesan Guruswami, Rajsekar Manokaran, and Prasad Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 573–582, 2008. doi:10.1109/FOCS.2008.51.

[63] Venkatesan Guruswami and Yuan Zhou. Approximating bounded occurrence ordering csps. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 15th International Workshop, APPROX 2012, and 16th International Workshop, RANDOM 2012, Cambridge, MA, USA, August 15-17, 2012. Proceedings*, pages 158–169, 2012. doi:10.1007/978-3-642-32512-0_14.

[64] Bernhard Haeupler and Robert Endre Tarjan. Planarity algorithms via pq-trees (extended abstract). *Electronic Notes in Discrete Mathematics*, 31:143–149, 2008. doi:10.1016/j.endm.2008.06.029.

[65] Johan Håstad. Some optimal inapproximability results. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 1–10, 1997. doi:10.1145/258533.258536.

[66] Lenwood S. Heath and Sriram V. Pemmaraju. Stack and queue layouts of directed acyclic graphs: Part II. *SIAM J. Comput.*, 28(5):1588–1626, 1999. doi:10.1137/S0097539795291550.

[67] Lenwood S. Heath, Sriram V. Pemmaraju, and Ann N. Trenk. Stack and queue layouts of directed acyclic graphs: Part I. *SIAM J. Comput.*, 28(4):1510–1539, 1999. doi:10.1137/S0097539795280287.

[68] Lenwood S. Heath and Arnold L. Rosenberg. Laying out graphs using queues. *SIAM J. Comput.*, 21(5):927–958, 1992. doi:10.1137/0221055.

[69] Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theor. Comput. Sci.*, 307(1):173–178, 2003. doi:10.1016/S0304-3975(03)00099-9.

[70] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster compressed dictionary matching. *Theor. Comput. Sci.*, 475:113–119, 2013. doi:10.1016/j.tcs.2012.10.050.

[71] Christopher Hoobin, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB Endow.*, 5(3):265–273, 2011. URL: http://www.vldb.org/pvldb/vol5/p265_christopherhoobin_vldb2012.pdf, doi:10.14778/2078331.2078341.

[72] Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016. doi:10.1016/j.tcs.2016.03.005.

[73] Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

[74] Haitao Jiang, Cédric Chauve, and Binhai Zhu. Breakpoint distance and pq-trees. In *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, pages 112–124, 2010. doi:10.1007/978-3-642-13509-5_11.

[75] Viggo Kann. *On the approximability of NP-complete optimization problems.* PhD thesis, Royal Institute of Technology Stockholm, 1992.

[76] Juha Kärkkäinen, Dominik Kempa, Yuto Nakashima, Simon J. Puglisi, and Arseny M. Shur. On the size of lempel-ziv and lyndon factorizations. In Heribert Vollmer and Brigitte Vallée, editors, *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, volume 66 of *LIPIcs*, pages 45:1–45:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.STACS.2017.45.

[77] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time lempel-ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, pages 189–200, 2013. doi:10.1007/978-3-642-38905-4_19.

[78] Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1344–1357, 2019. doi:10.1137/1.9781611975482.82.

[79] Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1002–1013, 2020. doi:10.1109/FOCS46700.2020.00097.

[80] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.

[81] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840, 2018. doi:10.1145/3188745.3188814.

[82] Subhash Khot. On the unique games conjecture. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, page 3, 2005. doi:10.1109/SFCS.2005.61.

[83] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*, pages 186–199, 2003. doi:10.1007/3-540-44888-8_14.

[84] Eun Jung Kim and Daniel Gonçalves. On exact algorithms for the permutation CSP. *Theor. Comput. Sci.*, 511:109–116, 2013. doi:10.1016/j.tcs.2012.10.035.

[85] Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive measure of repetitiveness. In *LATIN 2020: Theoretical Informatics - 14th Latin American Symposium, São Paulo, Brazil, January 5-8, 2021, Proceedings*, pages 207–219, 2020. doi:10.1007/978-3-030-61792-9_17.

[86] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

[87] Manfred Kufleitner. On bijective variants of the burrows-wheeler transform. In *Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009*, pages 65–79, 2009. URL: `http://www.stringology.org/event/2009/p07.html`.

[88] Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Efficient construction of a complete index for pan-genomics read alignment. In *Research in Computational Molecular Biology - 23rd Annual International Conference, RECOMB 2019, Washington, DC, USA, May 5-8, 2019, Proceedings*, pages 158–173, 2019. `doi:10.1007/978-3-030-17083-7\_10`.

[89] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, pages 201–206, 2010. `doi:10.1007/978-3-642-16321-0\_20`.

[90] Pierre Lalonde and Arun Ram. Standard lyndon bases of lie algebras and enveloping algebras. *Transactions of the American Mathematical Society*, 347(5):1821–1830, 1995.

[91] Gad M. Landau, Laxmi Parida, and Oren Weimann. Gene proximity analysis across whole genomes via PQ trees[1]. *Journal of Computational Biology*, 12(10):1289–1306, 2005. `doi:10.1089/cmb.2005.12.1289`.

[92] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

[93] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[94] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics*, 26(5):589–595, 2010. doi:10.1093/bioinformatics/btp698.

[95] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[96] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Lower bounds based on the exponential time hypothesis. *Bulletin of the EATCS*, 105:41–72, 2011. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/92.

[97] M. Lothaire. *Combinatorics on words*, volume 17. Cambridge university press, 1997.

[98] Lily Major, Amanda Clare, Jacqueline W. Daykin, Benjamin Mora, Leonel Jose Peña Gamboa, and Christine Zarges. Evaluation of a permutation-based evolutionary framework for lyndon factorizations. In *Parallel Problem Solving from Nature - PPSN XVI - 16th International Conference, PPSN 2020, Leiden, The Netherlands, September 5-9, 2020, Proceedings, Part I*, pages 390–403, 2020. doi:10.1007/978-3-030-58112-1_27.

[99] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Combinatorial Pattern Matching, 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005, Proceedings*, pages 45–56, 2005. doi:10.1007/11496656_5.

[100] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of individual genomes. In *Research in Computational Molecular Biology, 13th Annual International Conference, RECOMB 2009, Tucson, AZ, USA, May 18-21, 2009. Proceedings*, pages 121–137, 2009. doi:10.1007/978-3-642-02008-7_9.

[101] Sabrina Mantaci, Antonio Restivo, Giuseppe Romana, Giovanna Rosone, and Marinella Sciortino. A combinatorial view on string attractors. *Theor. Comput. Sci.*, 850:236–248, 2021. doi:10.1016/j.tcs.2020.11.006.

[102] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the burrows wheeler transform and applications to sequence comparison and data compression. In *Combinatorial Pattern Matching, 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005, Proceedings*, pages 178–189, 2005. doi:10.1007/11496656_16.

[103] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the burrows-wheeler transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/j.tcs.2007.07.014.

[104] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting suffixes of a text via its lyndon factorization. In Jan Holub and Jan Zdárek, editors, *Proceedings of the Prague Stringology Conference 2013, Prague, Czech Republic, September 2-4, 2013*, pages 119–127. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2013. URL: http://www.stringology.org/event/2013/p11.html.

[105] Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Suffix array and lyndon factorization of a text. *J. Discrete Algorithms*, 28:2–8, 2014. doi:10.1016/j.jda.2014.06.001.

[106] Gary L. Miller. Graph isomorphism, general remarks. *J. Comput. Syst. Sci.*, 18(2):128–142, 1979. doi:10.1016/0022-0000(79)90043-6.

[107] Marcin Mucha. Lyndon words and short superstrings. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New*

Orleans, Louisiana, USA, January 6-8, 2013, pages 958–972, 2013. doi:10.1137/1.9781611973105.69.

[108] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[109] Gonzalo Navarro. Indexing highly repetitive string collections. *CoRR*, abs/2004.02781, 2020. URL: https://arxiv.org/abs/2004.02781, arXiv:2004.02781.

[110] Gonzalo Navarro, Carlos Ochoa, and Nicola Prezza. On the approximation ratio of ordered parsings. *IEEE Trans. Inf. Theory*, 67(2):1008–1026, 2021. doi:10.1109/TIT.2020.3042746.

[111] Gonzalo Navarro and Nicola Prezza. Universal compressed text indexing. *Theor. Comput. Sci.*, 762:41–50, 2019. doi:10.1016/j.tcs.2018.09.007.

[112] Gonzalo Navarro and Victor Sepulveda. Practical indexing of repetitive collections using relative lempel-ziv. In *Data Compression Conference, DCC 2019, Snowbird, UT, USA, March 26-29, 2019*, pages 201–210, 2019. doi:10.1109/DCC.2019.00028.

[113] Alantha Newman. Cuts and orderings: On semidefinite relaxations for the linear ordering problem. In *Approximation, Randomization, and Combinatorial Optimization, Algorithms and Techniques, 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2004, and 8th International Workshop on Randomization and Computation, RANDOM 2004, Cambridge, MA, USA, August 22-24, 2004, Proceedings*, pages 195–206, 2004. doi:10.1007/978-3-540-27821-4_18.

[114] Adam M. Novak, Erik Garrison, and Benedict Paten. A graph extension of the positional burrows-wheeler transform and its applications. *Algorithms for Molecular Biology*, 12(1):18:1–18:12, 2017. doi:10.1186/s13015-017-0109-9.

[115] Tatsuya Ohno, Kensuke Sakai, Yoshimasa Takabatake, I Tomohiro, and Hiroshi Sakamoto. A faster implementation of online rlbwt and its application to lz77 parsing. *Journal of Discrete Algorithms*, 52:18–28, 2018.

[116] Jaroslav Opatrny. Total ordering problem. *SIAM J. Comput.*, 8(1):111–114, 1979. doi:10.1137/0208008.

[117] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.*, 43(3):425–440, 1991. doi:10.1016/0022-0000(91)90023-X.

[118] Christos H. Papadimitriou and Mihalis Yannakakis. The traveling salesman problem with distances one and two. *Math. Oper. Res.*, 18(1):1–11, 1993. doi:10.1287/moor.18.1.1.

[119] Ján Plesník. The np-completeness of the hamiltonian cycle problem in planar digraphs with degree bound two. *Inf. Process. Lett.*, 8(4):199–201, 1979. doi:10.1016/0020-0190(79)90023-1.

[120] Simon J. Puglisi and Bella Zhukova. Relative lempel-ziv compression of suffix arrays. In *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, pages 89–96, 2020. doi:10.1007/978-3-030-59212-7_7.

[121] Prasad Raghavendra. Optimal algorithms and inapproximability results for every csp? In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria,*

British Columbia, Canada, May 17-20, 2008, pages 245–254, 2008. doi:10.1145/1374376.1374414.

[122] Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. doi:10.1007/s00453-012-9618-6.

[123] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 11(2):375–388, 2014.

[124] Kazuya Tsuruta, Dominik Köppl, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Grammar-compressed self-index with lyndon words. *CoRR*, abs/2004.05309, 2020. URL: https://arxiv.org/abs/2004.05309, arXiv:2004.05309.

[125] Yuki Urabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. On the size of overlapping lempel-ziv and lyndon factorizations. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 29:1–29:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.CPM.2019.29.

[126] Lianping Yang, Guisong Chang, Xiangde Zhang, and Tianming Wang. Use of the burrows–wheeler similarity distribution to the comparison of the proteins. *Amino acids*, 39(3):887–898, 2010.

[127] D Younger. Minimum feedback arc sets for a directed graph. *IEEE Transactions on Circuit Theory*, 10(2):238–245, 1963.

[128] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.